Journal of Signal Processing Systems (2019) 91:33–46
https://doi.org/10.1007/s11265-018-1416-1

Exploiting Task Parallelism with OpenCL: A Case Study

Pekka Jääskeläinen[1] · Ville Korhonen[1] · Matias Koskela[1] · Jarmo Takala[1] · Karen Egiazarian[1] · Aram Danielyan[2] · Cristóvão Cruz[2] · James Price[3] · Simon McIntosh-Smith[3]

Received: 1 April 2017 / Revised: 15 February 2018 / Accepted: 1 October 2018 / Published online: 15 October 2018

Abstract
While data parallelism aspects of OpenCL have been of primary interest due to the massively data parallel GPUs being on focus, OpenCL also provides powerful capabilities to describe task parallelism. In this article we study the task parallel concepts available in OpenCL and find out how well the different vendor-specific implementations can exploit task parallelism when the parallelism is described in various ways utilizing the command queues. We show that the vendor implementations are not yet capable of extracting kernel-level task parallelism from in-order queues automatically. To assess the potential performance benefits of in-order queue parallelization, we implemented such capabilities to an open source implementation of OpenCL. The evaluation was conducted by means of a case study of an advanced noise reduction algorithm described as a multi-kernel OpenCL application.

Keywords OpenCL · Task-level parallelism

1 Introduction

OpenCL is a widely-adopted programming standard for parallel heterogeneous systems. The goal of the standard is to support a wide range of heterogeneous platforms efficiently and provide source code portability across them. While data parallelism aspects of OpenCL have been of primary interest to its users due to the massively parallel GPU devices being on focus, OpenCL also provides extensive capabilities to describe heterogeneous task parallelism by means of pushing commands to one or more command queues controlling one or more devices, and using events, command queue barriers or kernel argument buffer data dependencies for synchronization.

We consider this side of the standard underutilized despite it being the feature to efficiently harness devices in heterogeneous platforms to collaboratively execute multi-kernel applications by reducing the "master role" of the host program. OpenCL *command queues* (CQ) provide a means to describe larger parts of the application structure to the OpenCL runtime, giving it an opportunity to optimize the execution at the higher level [9]. OpenCL 2.0 [10] introduced additional task-related features, allowing devices themselves to launch new kernels asynchronously, which blurs the original division of responsibilities between a "master host" and "slave devices".

In this article we study the task parallel concepts of OpenCL and evaluate how well vendor-specific implementations of OpenCL can currently exploit task parallelism. As we found out that the vendor implementations are not yet capable of extracting kernel-level task parallelism from in-order queues automatically, we also propose a task scheduling runtime which can analyze the data dependencies automatically and utilizes multicore processors with various memory hierarchies efficiently.

The rest of the article is organized as follows. In Section 2 we discuss the background of OpenCL and its potential in harnessing entire heterogeneous platforms. Section 3 describes how task parallelism can be expressed using OpenCL constructs, whereas Section 4 shows the steps to efficiently construct multi-device heterogeneous task graphs out of multiple command queues. Section 5 details the proposed command queue runtime. Section 6 evaluates the performance using a case study of an advanced noise reduction algorithm. Finally, conclusions and our future plans are presented in Section 8.

✉ Pekka Jääskeläinen
pekka.jaaskelainen@tut.fi

1 Tampere University of Technology, Tampere, Finland
2 Noiseless Imaging Ltd., Tampere, Finland
3 University of Bristol, Bristol, UK

2 Platform-Wide Execution of Heterogeneous Task Graphs

Due to its history as a standardized programming model for GPGPUs, OpenCL has been mostly used to reap quick performance increases from GPUs where the parallel performance is abundant, but requires data parallel kernels to exploit to the maximum. Therefore, a common parallel programming pattern in OpenCL accelerated applications has been a straightforward host-slave "single kernel at a time" offloading model where most of the application logic is written in standard C/C++ with only the accelerated parts calling the OpenCL API for getting speedups available with the more parallel devices. The parts of the application not suitable for parallel GPUs have been thus naturally mapped to the host processor, which is usually a general purpose CPU that can execute serial code faster due to higher clock frequencies, branch prediction and speculative execution.

But why should the programmers bother writing larger parts of the application as OpenCL kernels that are pushed to command queues when large speedups can be already reached by the simple model of CPU to GPU offloading? So far there has been only limited number of non-GPU based devices available that could have been efficiently programmed using OpenCL. However, more and more OpenCL support for devices originally designed for other tasks than graphics processing has appeared to the market from vendors such as Movidius (Myriad vision processing units [12]) and Texas Instruments (C6000 DSPs [15]). Synopsys also now provides OpenCL support for application-specific processors produced using their ASIP Designer tools [14]. Also FPGA vendors have acknowledged the benefits of a heterogeneous parallel programming standard for providing more efficient results from high-level synthesis [2, 3] with a rising trend of integrating configurable logic at the chip or package level with CPUs and GPUs [5, 16]. As this development progresses, and a wider diversity of OpenCL-supported devices are becoming accessible in a same platform, the performance benefits of structuring the OpenCL implementation application for efficient coordinated multi-kernel execution is becoming more tangible.

The power performance promise of heterogeneous computing can only be redeemed when the application is properly partitioned and each part mapped to a device with the best matching architecture. However, the costs of mapping kernels in the application to multiple devices must not shadow the power-performance benefits of utilizing a more suitable device for the task at hand. Such costs include the additional synchronization related communication needed when a consumer task is not residing in the same device as the producer task. This is costly especially in case the synchronization has to be done across different chips, but also adds to power consumption and traffic congestion in case of using on-chip interconnection networks. If the device communication requires operating system calls in the host, it also adds context switch overheads.

In addition to the lack of diversity in OpenCL supported devices, from our experience, another major reason for resorting to the simple single kernel offloading model has been the often quoted low level of the OpenCL programming model. Many programmers consider it too burdensome to describe the whole application logic using the OpenCL constructs, therefore programmers tend to implement most of the logic in the host program without calling OpenCL. This is emphasized in case of legacy applications that are accelerated using OpenCL.

In this aspect, OpenCL can be considered to have a bit of an identity problem; on one hand it is too low level as an end user programming model, leaving a lot of decisions such as kernel-to-device partitioning, or the style of data parallelism to the shoulders of the programmer. On another hand, it contains rather high level abstractions and "programmer-targeted features", such as two kernel description languages, instead of only defining a kernel intermediate representation for the compilers to target. In contrast, the more recent *Heterogeneous Systems Architecture (HSA)* [1] standard clearly sets itself to a lower level in the heterogeneous parallel software stack with very strictly defined hardware features, bit exact in-memory control structures, and synchronization protocols that the conformant heterogeneous platforms must implement.

If OpenCL is considered too low level to efficiently implement large applications with, or requires a lot of boilerplate code when accelerating legacy applications, should more focus be put on using it as a portable software stack layer below more programmer productive higher-level programming languages? In this use, OpenCL provides a benefit over HSA; its looser requirements to the underlying hardware platform resulting from the somewhat higher level abstractions. HSA requires a coherent shared virtual memory across the whole system of which power efficient and scalable implementation is considered an open research challenge [7, 8, 13]. While hardware based cache coherency is not a problem for many classes of heterogeneous platforms, it is a too strict requirement to place for a programming model used as a portability layer which is desired to cover also the most challenging *high performance low power* use cases and the highly embedded devices of "Internet-of-Things" use cases.

We believe OpenCL has untapped potential of being efficiently utilized as a portability layer for wide range of heterogeneous platforms that are capable of executing heterogeneous task graphs across various devices in an independent manner. Its task graph description capabilities are powerful enough to describe task graphs that are "heterogeneous", that is, which can utilize various type of devices and explicit synchronization that can be optimized by the runtime, and thus spread the execution across diverse heterogeneous platforms.

3 Task Parallel Concepts in OpenCL

OpenCL programs structure the computational parts of the application into *kernels* and specify that there must be no data dependencies between the "kernel instances" (*work-items*) by default. This allows the programmer to describe parallelism in the *single program multiple data (SPMD)* style. In this style, multiple parallel *work-items* execute the same kernel function in parallel with synchronization expressed explicitly by the programmer. Another concept utilizing the SPMD model is the *work-group (WG)* which bundles sets of work-items that can possibly synchronize with each other. The specification states that the groups itself can be executed completely independently from each other. These concepts allow exploiting scalable data level parallelism at multiple levels with a single kernel command; across work-items in a single WG and all the WGs in the *work-space*.

Thread-level parallelism can be utilized in OpenCL in multiple ways: First, as WGs are assumed to be data independent of each other, they can be executed as coarse-grained "embarrasingly parallel" tasks, to exploit multiple hardware threads (or cores) available in the devices. At the higher application abstraction level, an abstraction called *command queue (CQ)* is used for pushing tasks to the devices in the platform.

There are two modes of operation available for the CQs: *In-order* mode, which has an implicit ordering constraint derived from the order the commands are enqueued in. The other mode of execution is *out-of-order (OoO)* with which the command execution ordering is constrained by explicit synchronization commands and explicitly defined event dependencies.

The event based command synchronization follows the common event handling scheme: Each *event* object encapsulates an execution status of a command, which other commands can monitor. Command dependencies are formed by defining an *event wait list* when enqueuing a command. It contains a list of events that must signal the finished status before the enqueued command is ready for execution.

OpenCL versions 1.0-1.2 state that commands in in-order queues must be executed in the order they have been queued, even if an external observer couldn't tell the difference. However, starting from version 2.0, reordering the command execution also in case of in-order CQs is explicitly allowed as long as the execution semantics are preserved [9, 11]. This is fulfilled in case the updates on the memory objects accessed by the kernel commands are visible to succeeding reads defined by the command enqueue order, and if commands that have other side effects are not reordered.

The less constrained in-order CQ execution semantics of OpenCL 2.0 mean that the practical difference of in-order to OoO CQs is that the programmer can rely on data dependence analysis based on the buffer arguments of the kernel commands to enforce the ordering constraints. This is in contrast to OoO CQs with which one must mark command dependencies explicitly with events or by using *command queue barriers*.

Commands between multiple CQs, regardless if they are in-order or OoO CQs are assumed independent from each other unless synchronized by events. A single CQ always targets a single device, but a single device may be targeted by multiple CQs in the same OpenCL context to expose task parallelism. Utilizing multiple CQs controlling multiple devices, the programmer can communicate the higher level application logic to the OpenCL runtime which can then perform static and dynamic scheduling for performance improvements.

Multiple command queues and shared (sub-)buffers can be used to form multiple device kernel execution that relieves the host as shown in Fig. 1. In this style of execution, OpenCL buffers residing in a shared memory are used to pass data between successive kernel commands. It allows describing execution of task sequences across the devices in the platform without having to synchronize buffers with the host after each executed kernel like in the simplest "CPU to GPU offloading model".

Kernel sequences connected with buffers is an efficient means in OpenCL to implement such type of applications where the sequence of tasks to execute vary, for example, per each frame of a video stream, and where the different tasks benefit from different style of devices. The model resembles, but is different from the streaming model enabled by OpenCL 2.0 *pipes*. Pipes are designed for forming data flow style execution pipelines with kernels residing in the devices for long periods of time, processing packets as they appear from other kernels.

4 Converting Command Queues to Task Graphs

In task scheduling, the tasks of the application and their dependencies are commonly expressed as a task graph (also sometimes referred to as a macro-dataflow graph). A task

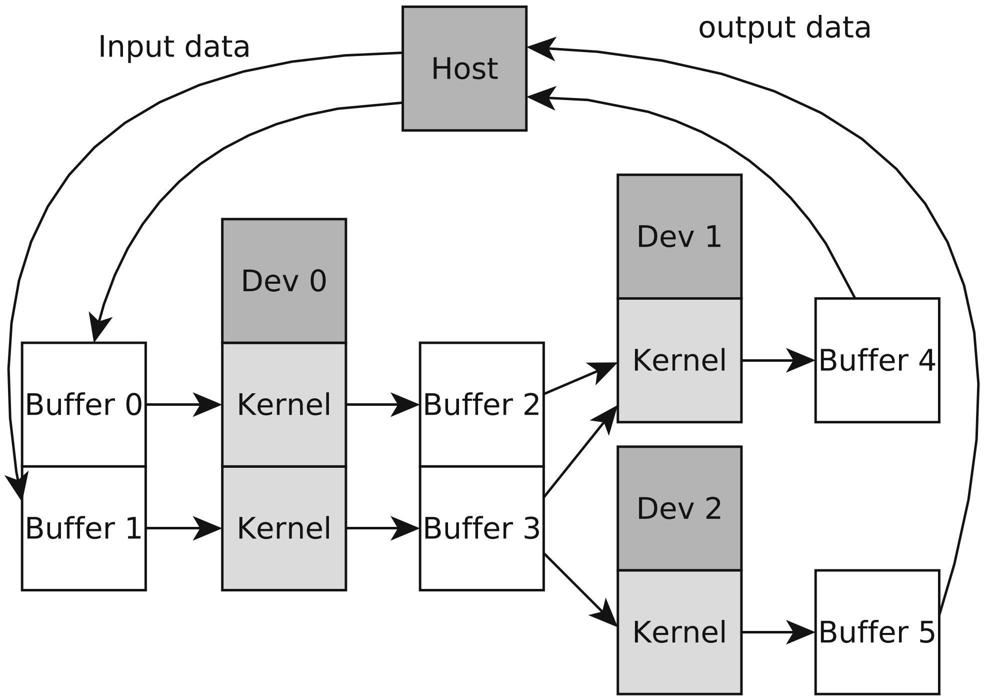

Figure 1 Example of a multi-device multi-kernel task sequence connected with buffers.

graph is defined as a *directed acyclic graph (DAG) G* consisting of a tuple of sets:

$$G = (V, E, C, T) \tag{1}$$

The graph topology is defined by set V which consists of the *tasks* in the application, and by the edge set E which defines communication direction or other ordering constraints between tasks. These two sets are enough to describe the semantics of the application to avoid illegal task execution ordering decisions by the scheduler. Additional information can be defined as labels by the set C that stores communication costs associated with edges, and by the set T which records the execution times of the tasks. [17]

The general task-scheduling problem is to manage the execution of the tasks in a set of processors in the platform as efficiently as possible, typically with the goal of minimizing the total execution time of the application or maximizing the energy efficiency. In order to accomplish its goal, the scheduler must make two decisions for each task: Which processor should execute each task (the *partitioning* decision) and at which time, or in which relative order to the other tasks (the *scheduling* decision).

When all G nodes and edges are labeled with their costs before the run time, that is, when the sets C and T are known before execution, the launch times of the tasks can be defined *statically* in advance. In this ideal situation there is no need for notifying the finishing of a task to the dependent tasks due to the common assumption of the tasks finishing early enough for the produced results to be valid and consumable by the dependent tasks. However, due to the dynamic nature of shared resource systems and data dependent task execution times that call for dynamic load balancing, fully static scheduling of task graphs to the extent of omitting explicit task synchronization is not practical. Therefore, task completion notification costs must be taken into account when designing an efficient task scheduling runtime system for executing both coarse and fine grained tasks.

In OpenCL, command queues, which are used to pass tasks to the runtime, are associated with a single OpenCL device. Thus, the partitioning problem of the tasks is partially delegated to the programmer or to a higher level programming model. Due to the relative simplicity of the contemporary OpenCL-programmable platforms, the OpenCL application partitioning decision is usually driven by the characteristics of the kernel at hand; massively data parallel kernels are mapped to GPUs, while kernels with more control oriented or uniformly executed parts are targeted to the CPU or a DSP. However, the OpenCL devices are often multicores themselves, leaving the choice of which *compute unit* inside the device to execute the task in to the runtime. Especially in case of multicore devices with non-shared cache hierarchy levels, the compute unit mapping choice for the work-group or a kernel command can make a big difference in terms of cache hit ratio. Clearly, also when scheduling for asymmetric multicore devices such as ARM's big.LITTLE CPUs that have multiple different types of cores with the same instruction-set architecture, the compute unit partitioning decision has a major impact.

In order to exploit the OpenCL buffer data locality, the runtime must ensure the buffers are not needlessly synced or moved around in the global memory hierarchy.

In addition, especially with applications involving a lot of small granularity kernels, it is important to reduce the global synchronization needs during execution, by ensuring that only the kernels that need to be notified of an event are notified while bothering the host or other devices as little as possible. In order to exploit application-level properties such as task dependencies and communication demands in task parallelization, the OpenCL runtime can utilize a task graph representing the kernels and their dependencies involved in the application.

4.1 Constructing the Task Graph

OpenCL standard, which is at the time of this writing at version 2.1, does not directly include a concept of a task graph. Therefore, the sets of G have to be populated from CQ abstractions such as events and commands while taking in account the special cases and differences in the versions of the standard.

When building a G from CQs, the set V is populated with nodes consisting of three different task types that can be directly converted from CQ concepts: *Memory transfer tasks* which move or synchronize buffer data, *synchronization tasks* which are used to explicitly restrict the execution order of commands, and *kernel tasks* for executing user defined compute tasks in a device.

In addition to the task types derived from CQ concepts, we use several other task types for improving the parallelism and optimizing the scheduling process: *Kernel compilation tasks* are added due to the possibility of OpenCL to build kernels online in the host program – presenting the compilation as tasks in the task graph enables the runtime to overlap kernel compilation with other tasks. Compilation task is implicitly added when a new kernel command is added with a dependency edge added to ensure compilation is finished before the kernel task can start. The kernel tasks are further split to *work-group tasks* for allowing fine-grained control of work-group execution across multiple compute-units.

While constructing the task nodes is a straightforward process, adding the edges require considering the different mixes of inorder and out-of-order queues. Extra care must be taken to extract as much task parallelism as possible while still preserving the application semantics as defined by each standard version. The edge set is constructed from two main data sources: Explicit *events* used to synchronize commands in one or more CQs and buffer *data* dependencies defined by the buffer arguments to kernel commands.

The rules for constructing the edges can be formalized to a set of conditions which are checked when considering two tasks $(v_a, v_b) \in V : v_a \neq v_b$ from command queues $(q_x, q_y) \in Q$, where Q represents the set of all command queues in the OpenCL context. In case at least one of the following conditions is fulfilled, an edge $v_a \rightarrow v_b$ is added to E.

$$\begin{aligned} q_x = q_y \wedge inorderq(q_x) \quad &\wedge \\ (OpenCL\_version < 2.0 \quad &\vee \\ (qafter(v_a, v_b) \wedge datadep(v_a, v_b)) & \end{aligned} \tag{2}$$

where $inorderq()$ is true in case the given command queue is an in-order queue. Function $qafter(v_a, v_b)$ returns true in case v_b was pushed to the CQ after v_a. Function $datadep(v_a, v_b)$ is true in case there is a data dependency between the two tasks according to their buffer arguments or shared program scope variables, a feature introduced in OpenCL 2.0. This condition handles nodes in the same in-order queue. An example of such a task graph is illustrated in Fig. 2. The example program writes kernel input data to the device memory, then launches the kernel B, which produces input for the next kernel C. Finally, the host reads back the results from the device. Edges between the nodes are implied by the in-order semantics of OpenCL 1.2.

$$(q_x \neq q_y \vee \neg inorderq(q_x)) \wedge waitson(v_a, v_b) \tag{3}$$

where $waitson(v_a, v_b)$ marks an event dependency such that v_b waits on a completion event produced by v_a. Out-of-order queues and event synchronization is handled by this condition. In case the two commands are from two different

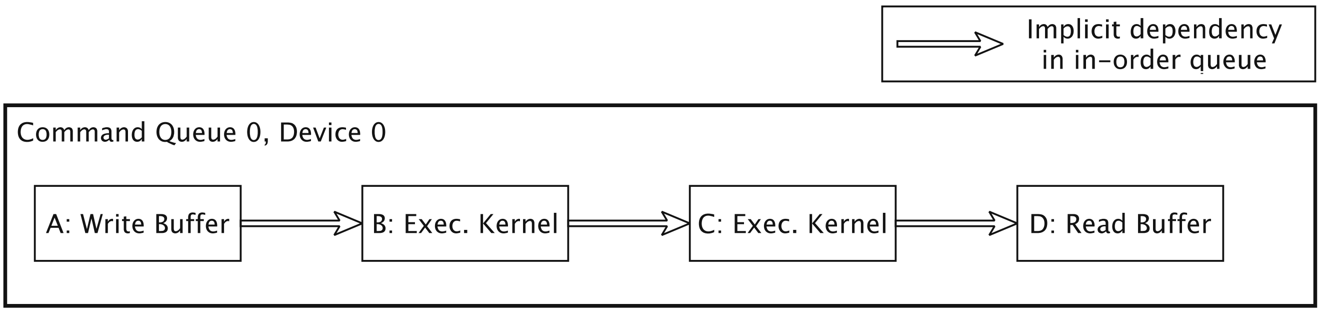
Implicit dependency in in-order queue
Command Queue 0, Device 0
A: Write Buffer
B: Exec. Kernel
C: Exec. Kernel
D: Read Buffer

Figure 2 Example of a simple task graph constructed from a single command queue in OpenCL 1.2. All tasks implicitly depend on results from the preceding task.

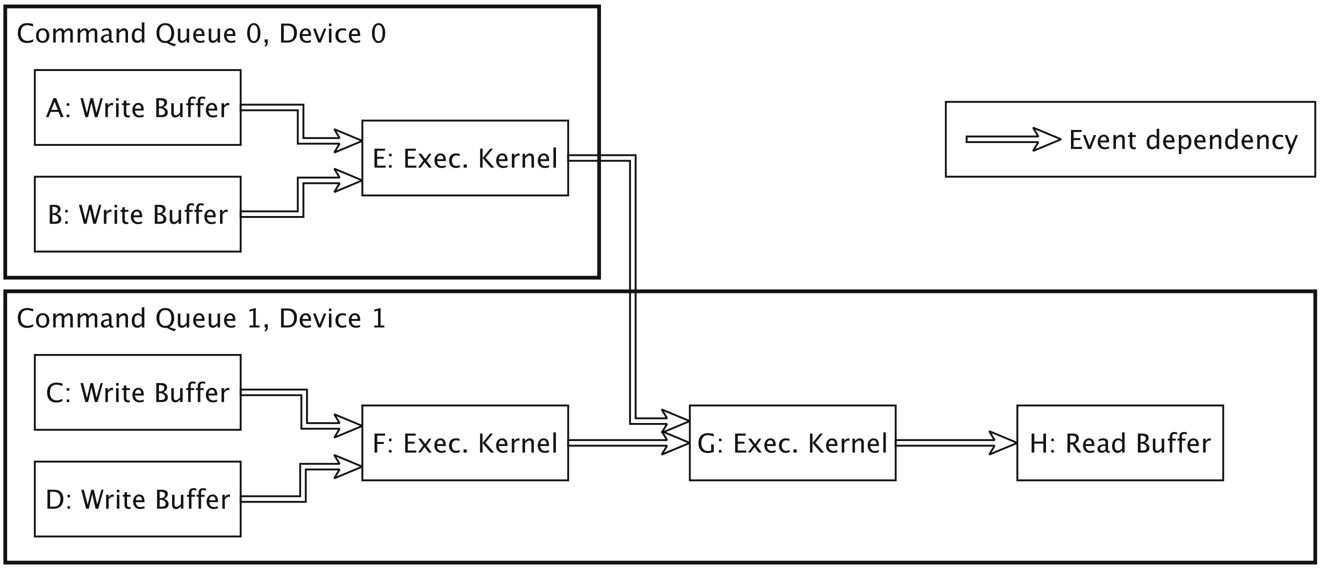

Figure 3 Example of a task graph extracted from a two-device program.

queues, they are always treated like they are from an out-of-order queue and must be event synchronized to ensure a predefined ordering. An example of a G built from two different queues possibly controlling two different devices is shown in Fig. 3. It can be seen that this application has task parallelism that can be exploited with multiple device or multiple compute unit execution. For example, the buffer commands can be executed in parallel in case there is enough memory bandwidth available, or overlapped with kernel execution from the other queue.

4.2 Command Queue Data Dependence Analysis

OpenCL programmers typically use in-order queues for their ease of use; the burden of extracting parallelism and enforcing adequate synchronization is placed to the runtime. Thus, in the edge creation conditions formulated in the previous section, the $datadep()$ function is placed major responsibility in extracting task parallelism from programs using in-order queues. The function returns true in case an edge should be added due to of either a known memory access conflict or because of being unable to prove there is no dependence between the given tasks. It relies on the fact that in OpenCL 2.0, it is legal to reorder the in-order queue commands in case semantical differences to the original order cannot be observed from the outside. In practice, state in OpenCL applications that is visible to the outside observers is transferred in *memory objects (MO)* the kernels receive as arguments and indirectly by means of *program scope variables (PSV)*. In this context *Pipe* memory objects are excluded, since their use cases differs greatly from normal *Buffer* and *Image* memory objects.

In case only considering any uses of MOs or PSVs, it is straightforward to add a set of edges to the task graph that constraint the execution order to prevent illegal command orderings. However, finer grained analysis that determines the read-write relationships of the accesses should be conducted for improving task parallelization opportunities.

When considering data dependencies between tasks, read-after-read dependencies can be ignored because they involve no data races which require serialization. Only the other access relationships, that is, write-after-read, read-after-write and write-after-write add restrictions to the execution ordering, and thus imply edges in the task graph. There are various ways to extract the access type information from CQ tasks. For memory transfer tasks, the access type is explicitly defined by the used API function. For example, *clWriteBuffer() writes* a MO to the device memory and *reads* it from the host memory, while *clReadBuffer()* performs an opposite direction memory transfer task.

Analyzing the MO usage of kernel tasks can be done by exploiting explicit information and by means of static kernel memory access analysis. In the task graph build process, there are three ways are used to derive the access type information:

1. At creation time, the MO can be flagged with CL_MEM_READ_ONLY or CL_MEM_WRITE_ONLY, indicating that any kernel task using the MO may only read it or write to it.
2. Restricted to *Image* MO's only: to check the arguments of the kernel task for *__read_only* and *__write_only* access qualifiers that have the same semantics with the corresponding MO flags, but can be defined at the granularity of a kernel task.

3. In case the buffer is in the *constant* address space, only read accesses are allowed.
4. When none of the above flags or qualifiers are present, read/write access is assumed by the standard. Static compiler analysis can be attempted to resolve whether kernel is only reading the data in the buffer.

PSVs are per-device global variables in kernels that have the same lifetime as the whole OpenCL application and once initialized, they can be read and written by other kernels in the same program module. PSVs are a completely device-side construct – no information of PSVs are given by the programmer at the host side. In addition to static kernel analysis, the following predicates can be derived from the specification text to reduce the number of task graph edges when a kernel is known to use PSVs:

1. If the kernel tasks v_a and v_b are not from the same program, there is no PSV-induced dependence.
2. If the kernel tasks v_a and v_b are not queued to the same device, there is no PSV-induced dependence.

5 Implementing a Task Scheduling Runtime

For this study, we extended an open source OpenCL implementation with a runtime that is able to extract task graphs from command queues and dynamically schedule their execution on shared memory single instruction set multicores (often referred to as CPUs).

There were two main challenges when developing the runtime: First, the point of time during the host application's execution affects the size of extracted task graphs heavily. This problem is discussed in the next subsection. Second, the mapping of kernels and work-groups to cores in various multicores is not trivial due to differing memory hierarchies and topologies. The proximity and the size of caches affects the resulting performance heavily. The implementation specifics resulting from this, with a portable OpenCL task scheduling algorithm are presented in the latter subsection.

5.1 Dynamic Construction of Task Graphs

In OpenCL applications, the extent of the task relationships that can be extracted to build a task graph that is beneficial in task scheduling is limited by calls made by the host program to OpenCL APIs such as *clFinish()* or *clFlush()*. When the program calls these functions, it communicates to the runtime that it assumes progress will be made in the execution of the CQ given as a parameter.

A common OpenCL host program idiom is an asynchronous execution style where one or more commands are pushed to a queue, and then *clFlush()* is called in hopes of kernels executing concurrently in a device while the host program is running and possibly pushing more tasks to the queue. This is typically done in a loop to produce a streaming style of execution using the command queues.

When *clFlush()* is called, it often means that more commands will be pushed to the CQ by the host application – otherwise it would have called *clFinish()* to indicate the finalization of the queue. When the host program needs to synchronize with commands running in the devices, it calls *clWaitForEvents()* to ensure a particular command and all its prerequisite commands in the chain of dependent tasks have finished, or blocks until all commands in the queue have finished execution by calling *clFinish()*.

When *clFinish()* is called, the runtime is signaled that no new commands will be pushed to the queue which would allow analyzing the contents of all CQs involved in the application to construct an expansive task graph. However, because the command queues can be constructed and sent for execution incrementally with an assumption of asynchronous execution progress for the already queued tasks, it means the task graph used by the runtime must be adaptive and efficiently expanded with new information when new commands are pushed to the runtime. This task is trivial with out-of-order queues and their explicit event-based synchronization that maps directly to task graph edges, but with in-order queues it requires some additional work.

In the proposed task graph based runtime, TG is constructed incrementally whenever new commands are pushed to a CQ. In order to speed up the edge analysis and to enable dynamic construction, the edge predicate checking is performed only between the newly pushed tasks and a set of previous ones. To analyze data dependencies dynamically, a data access bookkeeping structure as shown in Fig. 4 is attached to each MO or PSV. The structure keeps track of the last queued task that modifies the given MO or PSV, and the tasks that read the data and have been enqueued after it. When a new write to the same object is encountered, read-after-write edges are created between the first write and the reads, and between all the reads and the new write. After a new write is encountered, the earlier *last write* and *last read* sets of tasks can be discarded with newly pushed data reading tasks being associated with a new *last write* record.

5.2 Dynamic Task Scheduling for Shared Memory Multicores

In the proposed dynamic task scheduling algorithm for various shared memory multicores, we group the compute units (processor cores) according to their cache hierarchy proximity to "compute groups". Physical cores (or hardware threads) that share the same lowest level cache entity, form a compute group. The purpose of the grouping is to enhance data locality improvements when task parallelism

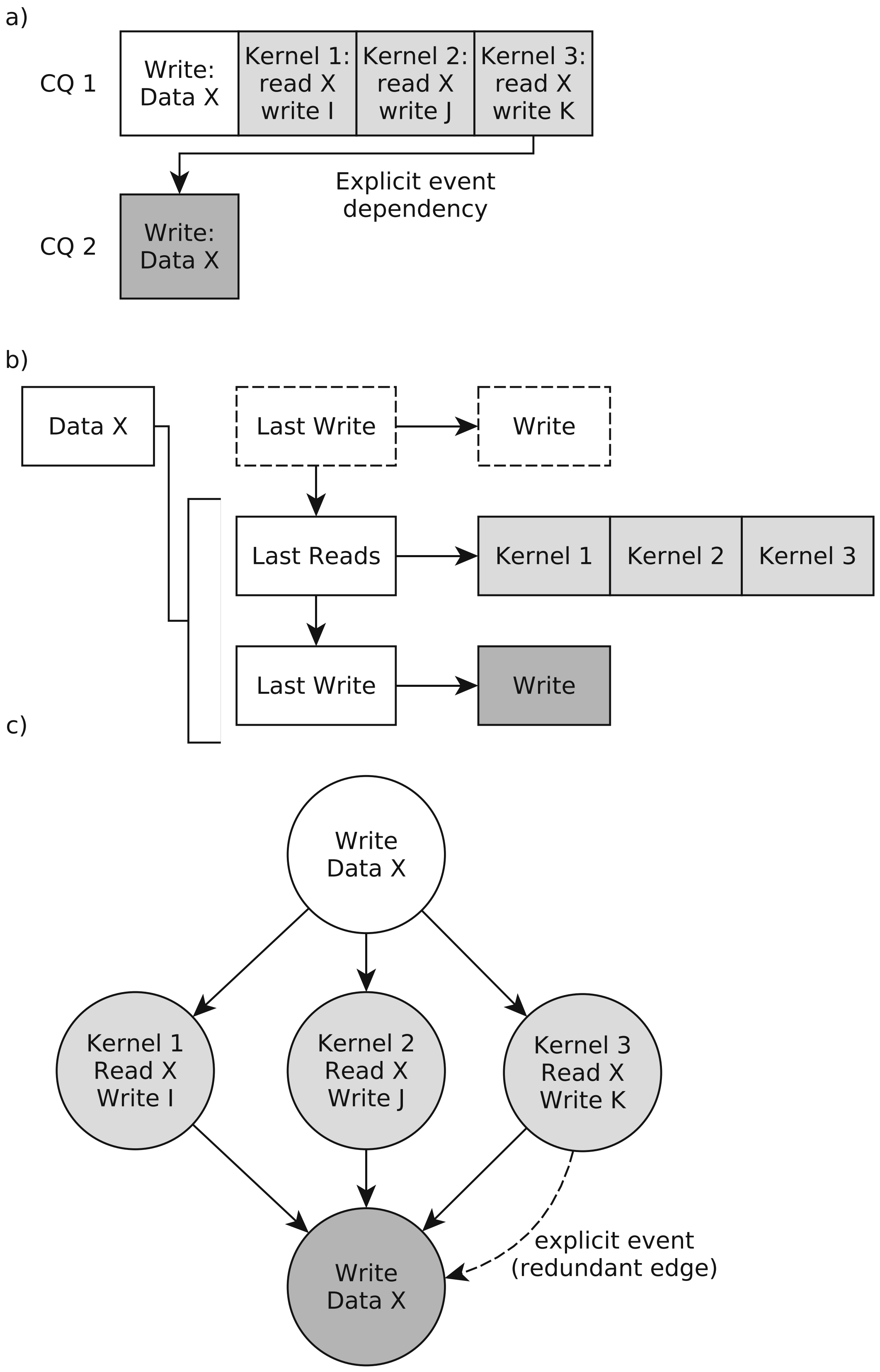

Figure 4 An application with **a** commands enqueued to two in-order command queues with an explicit event synchronization point, **b** a dynamic data access book keeping structure for data X (MO or PSV) when all commands are enqueued, but none have completed yet, and **c** the extracted task graph that exposes parallelism, but preserves the in-order queue semantics.

is available by prioritizing cores in the same group when launching tasks that are dependent on the previous kernels executing in the group. Especially in a "tiled" execution model typical in image or video processing pipelines the data is processed in smaller blocks with multiple successive kernels this type of execution is expected to enhance cache hits as the product of the previous kernel is likely still remaining in the close cache for the next kernels to read.

Each compute group contains a work queue for each physical core and a worker thread for each hardware thread. All of the worker threads are bound to one of the physical cores belonging to the compute node.

The initial work sharing is performed as follows. Enqueued commands are stored to a task list in their order of arrival. When a command queue is flushed, the task list is dismantled by utilizing the event dependency information to compose the task graphs as was described previously. Task graphs are pushed to the compute groups in a round robin fashion as soon as they are extracted. Inside the compute groups, the task graphs are further round robin distributed to the work queues.

During execution, the load is balanced by means of locality aware work stealing: Each worker thread has a priority ordered list of work queues from where they seek for work. When executing commands, workers try to maintain depth first ordering by preferring immediate task graph successors of the commands they execute in order to enhance data locality.

The worker thread execution thus proceeds in the following main steps, which is looped until no commands are available:

1. Go through the work queues in the compute group until a launched task graph or a ready to launch task graph is found.
2. Keep executing commands from the chosen task graph until there are no commands to execute or there are only commands blocked by other events.
3. When own compute group runs out of task graphs, the first worker in the node to make this observation tries to steal one full task graph from another compute group.
4. If stealing was not possible start going through work queues in the next compute group, emphasizing groups with shared lower level cache hierarchies.

6 Case Study

6.1 The Application

We evaluated the current state of OpenCL task parallelism in vendor SDKs and the proposed command scheduling algorithm using an advanced noise reduction that consists of multiple steps that were natural to divide to multiple OpenCL kernels.

The algorithm is called BM3D [4]. Its OpenCL implementation was partitioned into multiple kernels executing individual steps forming a kernel pipeline. The pipeline begins with a *block matcher* kernel which reads in smaller tiles of the input image, and searches for similar tiles from the other parts of the image. *Find matches* kernel does the ordering of the matching blocks from the most similar to the least similar. *Threshold* kernel does the actual noise canceling by using *haar* transforms and thresholding. *Aggregate* returns blocks back to their places in the output image, and finally, *Aggregate division* computes a weighted average of the blocks written to the output.

The kernel pipeline is launched for 30 input frames in parallel in order to produce abundant task-level parallelism for the tested runtimes to exploit.

6.2 Tested Runtimes

We used the following OpenCL implementations for comparing their runtimes' capabilities to exploit kernel level parallelism: The latest release of *Portable Computing Language (pocl-0.14* [6]*)*, pocl 0.14 with the proposed task scheduling runtime, Intel OpenCL runtime v16.1.1 and AMD APP SDK v3.0.

The benchmarked multicore platforms were: A dual-socket Intel Xeon E5-2697 v3, the CPU of AMD A10-7850K (Kaveri), and Intel Core i7. The Xeon platform is a NUMA architecture with two separate 12-core (24 hardware thread) processors each having their own cache hierarchy. This CPU incurs high penalties from bad task assignments due to the multi-socket setup. AMD A10 is a quad core that consists of two dual core processors with their own L2 caches. Intel Core i7 is a quadcore with a single shared L3 cache.

Clearly, the AMD OpenCL implementation is assumed to be optimized on AMD CPUs and the Intel's on their own, it is possible to run them in their competitors' CPUs. Thus, just for the sake of comparison we included numbers of AMD OpenCL SDK on Intel's CPUs and vice versa.

Multiple different ways to use command queues to expose task parallelism were evaluated: The first one is the case simplest to the programmer; all commands are enqueued to a single in-order command queue without explicit kernel-level task parallelism. The second alternative uses a separate in-order command queue for each of the 30 frames, thus explicitly communicating that commands processing each of the frames are mutually independent. The third option is a single out-of-order command queue where all available task parallelism is stated explicitly by forming event dependencies only between those commands that must be ordered. Finally, the fourth alternative uses

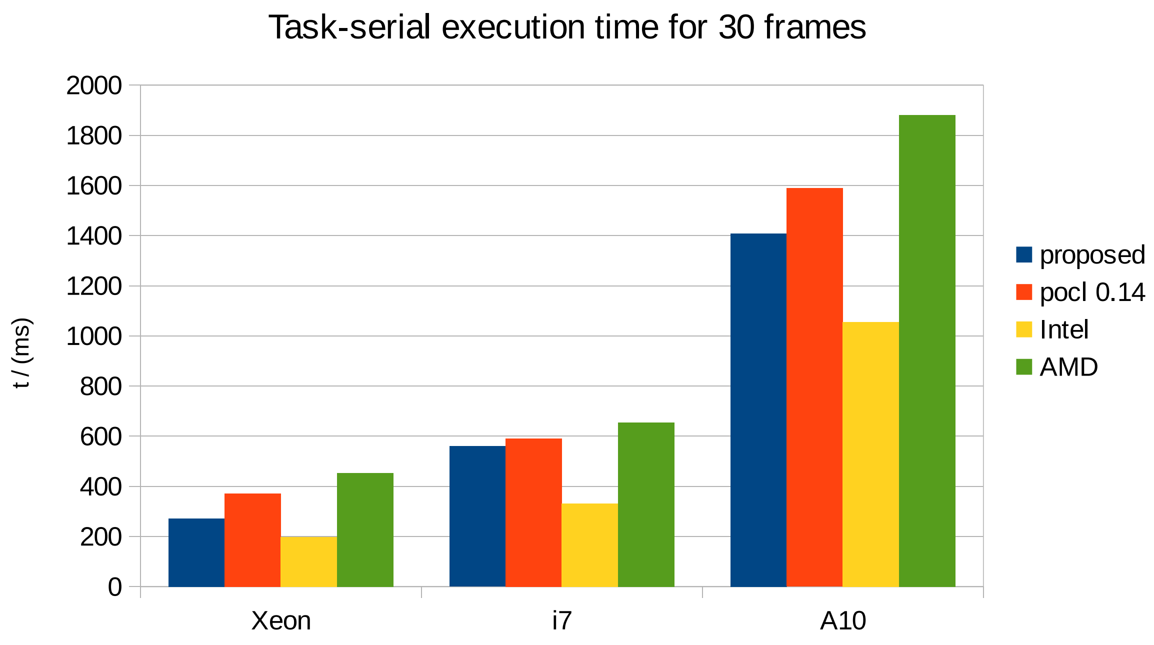

Figure 5 Total execution times for each platform with a single in-order queue.

a separate out-of-order-queue for each frame with event synchronization. The proposed implementation's in-order queue parallelization capabilities were also evaluated with a single in-order queue, assuming OpenCL 2.0 command queue execution semantics which explicitly allow in-order queue parallelization. We found that the other SDKs do not yet support this mode at the time of this writing.

The total execution times are not feasible to compare between the OpenCL platforms because the performance of the kernel compilers varies a lot (e.g., due to executing an Intel SDK on an AMD CPU and vice versa) which affects the cache footprint and the synergy between multiple kernels running at the same time. Regarding the kernel compiler we found that the Intel's performs the best for most of the cases and that pocl 0.14's kernel compilation performance has fallen behind. However, as the focus of this article is in how task parallelization differs with different command queue usage styles, we computed the speedups in comparison to the kernel serial execution in each case. It should be noted that the compared kernel serial execution can still utilize work-group level parallelism across multiple cores and the additional task level parallelism measured here results from concurrent execution of multiple commands. The task-serial execution times of the application on each platform is shown in Fig. 5. This is the baseline we measure the task parallelization speedups from.

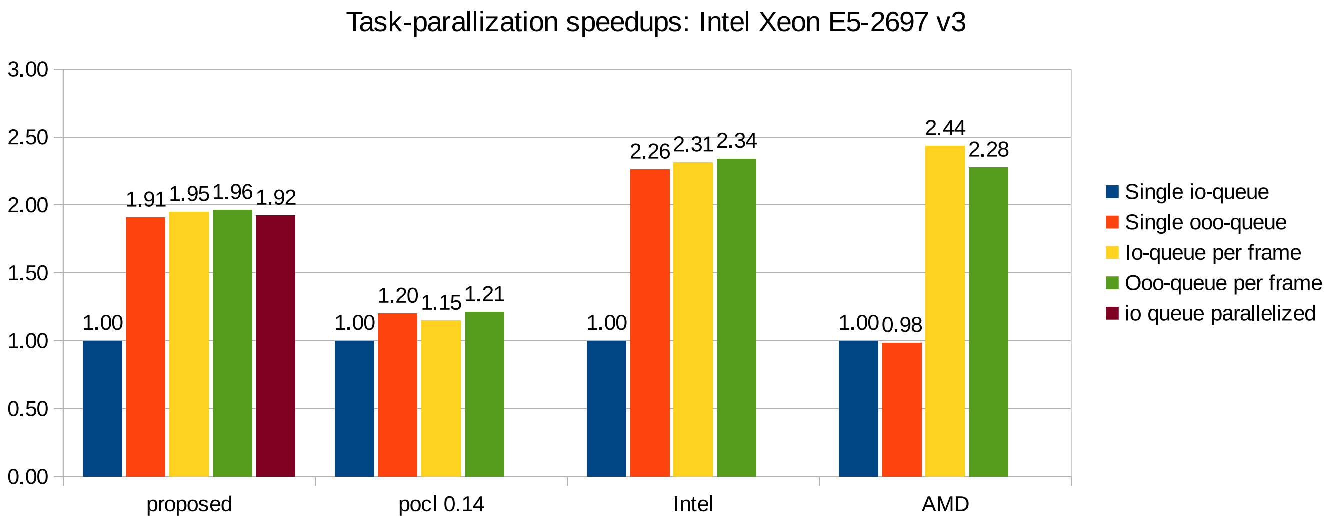

Figure 6 Speedups from utilizing task parallelism on the Intel Xeon CPU.

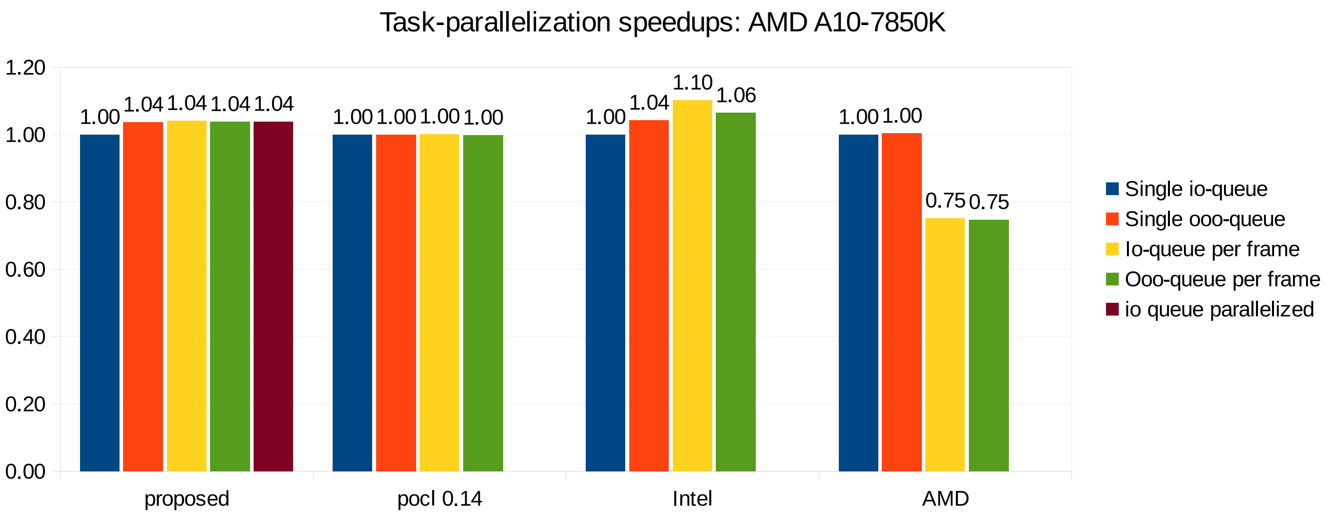

Figure 7 Speedups from utilizing task parallelism on the AMD A10 CPU.

Intel Xeon results are shown in Fig. 6. Intel and AMD both can exploit the abundance of cores in this CPU the best. Proposed runtime gets speedups up to 1.96x, while Intel's SDK up to 2.3x and AMD's up to 2.44x. However, AMD requires multiple command queues to be used in order to execute kernels in parallel. Intel can also extract task parallelism from a single out-of-order queue. The proposed runtime is the only one that can utilize the automatic in-order queue kernel parallelization and can reach the same parallelization performance utilizing only a single in-order queue for all the commands. Similar benefit can be seen with the other CPUs as well. Pocl 0.14 reaches only up to 1.2x speedup in its current state.

AMD A10 results are shown in Fig. 7. The results from *proposed* and Intel are quite similar to Xeon platform, both platforms gain some performance improvement when task parallelism is available. Intel's task parallelization speedup is 4 - 10%. *Proposed* gets a steady speedup of 4%, also with the automatic in-order queue parallelization. Pocl 0.14 gains no performance improvement from kernel level parallelism regardless of how it was described. AMD SDK results are quite unexpected since it seems to suffer a 25% performance *penalty* when multiple queues are used to describe kernel level parallelism. It is likely utilizing some sort of a simple but fair round robin scheduling scheme, executing a command from each queue at the time, which leads to a bad cache foot print as it doesn't exploit the data locality between kernels processing a single frame. The speedups in general are considerably lower than with the Xeon. This is likely due to the individual kernels in the application sufficiently utilizing all the cores via work-group parallelization.

Finally, Intel i7 results are shown in Fig. 8. On this platform, *proposed* gets steady 7% speedups, while Intel's

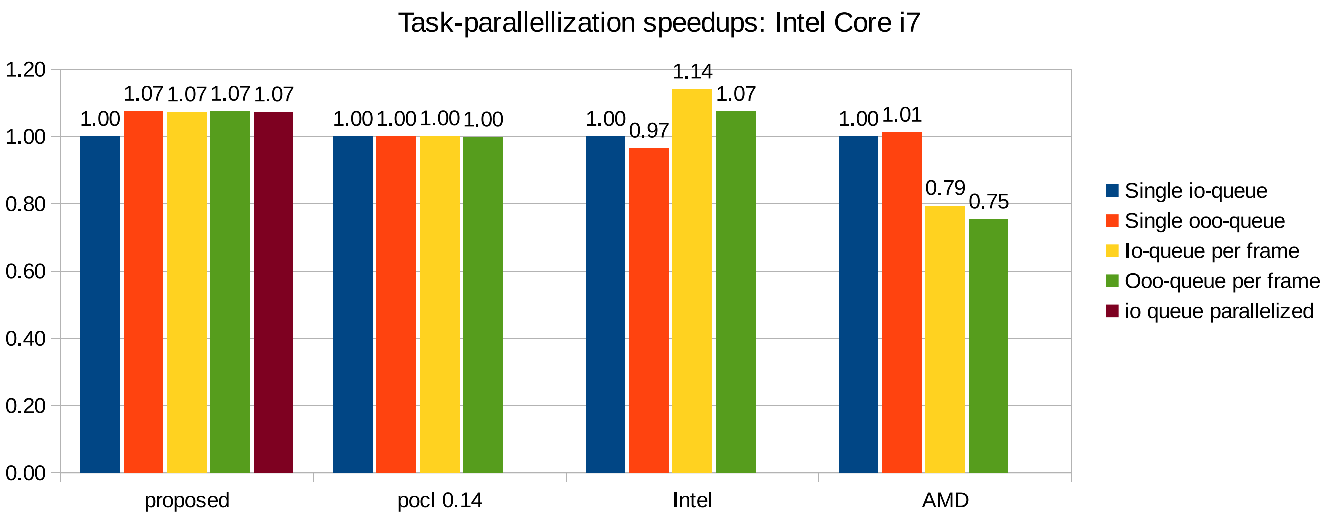

Figure 8 Speedups from utilizing task parallelism on the Intel Core i7.

speedup is capped at 14%. For some reason Intel's single ooo-queue case is 3% slower than the base line. Pocl 0.14 again gains no speedups from kernel level parallelization. AMD SDK does not gain anything from a single out-of-order queue and again suffers over 20% performance penalty in multiple command queue scenarios.

In conclusion, the proposed task scheduler which was added to pocl 0.14, improves its performance of task parallel execution and is the only one of the compared ones currently capable of task parallelizing commands from a single in-order queue efficiently. However, Intel OpenCL SDK has overall fastest implementation on the tested CPUs. It can also efficiently utilize task parallelism described using the different command queue schemes. This is especially visible on the dual Xeon platform. It also performs well even when using rather large number of command queues to isolate the task parallel parts.

AMD SDK, on the other hand, seems not to be optimized for task parallel execution. The dual Xeon platform makes an exception where speedups are on par with Intel's. The results suggest that AMD's SDK is not currently making data locality aware scheduling decisions based on the command queue dependencies, but schedules from command queues "fairly" which had severe impact on the platforms with more limited cache resources of this case study.

7 Related Work

This article presented the ways task-level parallelism can be described and utilized in OpenCL programs, and evaluated how well this aspect of OpenCL applications is taken advantage of by commercial and open source runtimes. To the best of the author's knowledge, there is no article with similar information published.

The article also described runtime techniques involved in extracting task level parallelism from in-order-queues utilizing the explicit data dependency information. The proposed runtime is the only one that can do automatic in-order queue kernel parallelization and reach the same parallelization performance utilizing only a single in-order queue for all the commands.

Of the related evaluated OpenCL runtimes, Intel OpenCL SDK had overall fastest runtime implementation on CPUs, efficiently utilizing parallelism described using the different command queue schemes. Intel's SDK driver offloads the task level scheduling to the Threading Building Blocks, which has advanced algorithms tuned to exploit Intel multicore CPUs.

The other related OpenCL runtime for CPUs, AMD's OpenCL SDK, seems not to be optimized for task parallel execution. Our evaluation suggests that AMD's SDK, unlike the proposed runtime, is not currently making data locality aware scheduling decisions based on the command queue buffer dependencies, but schedules from command queues "fairly" which had severe impact on the platforms with more limited cache resources of this case study.

8 Conclusions

OpenCL enables multiple ways to explicitly describe parallelism across multiple commands (tasks) utilizing out-of-order command queues and event synchronization. In addition, starting from version 2.0 of the standard, implicit extraction of task parallelism from in-order command queues is allowed.

We used a case study of an advanced multi-kernel image denoising application to measure how well the vendor OpenCL CPU implementations can exploit task parallelism when the commands are pushed in different ways to command queues by the host program. In addition, we developed a task scheduling runtime to the open source *pocl* implementation which can exploit task parallelism from the explicit command queue scenarios, and also from in-order queues based on the buffer dependencies.

We found that the tested OpenCL implementations were generally able to exploit task parallelism when independent tasks were pushed to separate in-order or out-of-order command queues, which is the most explicit way to describe task parallelism in OpenCL. Intel's implementation was able to also parallelize multiple independent task graphs pushed to a single out-of-order-queue. None of the vendor SDKs were currently able to parallelize in-order command queues. The speedups measured from the implicit task parallelism that the proposed runtime could utilize were on par with the explicit ones for the case study application.

One important aspect in task graph execution is to utilize the task graph's dependencies to schedule dependent tasks (those that share data) closer to each other both spatially and temporally. The case study indicated that AMD's SDK seems not to do this, but use some kind of round robin scheme as the rather high number (30) of command queues actually resulted in a slow down. The proposed runtime and the Intel's were able to utilize data locality and were not affected inversely by distributing the task graphs to their own command queues.

This article described internals of a key component used in the ALMARVI project's image processing software stack. The project used an OpenCL centric stack that was supported with unified hardware interfaces and higher level programming models. The OpenCL task scheduling runtime described in this article was utilized to accelerate multiple kernel execution of complex image filtering algorithms.

Acknowledgments This work was primarily supported by the ARTEMIS joint undertaking under grant agreement no 621439 (ALMARVI). Some of the authors also received funding from Academy of Finland (funding decision 297548) and Finnish Funding Agency for Technology and Innovation (project ”Parallel Acceleration” funding decision 40115/13).

References

1. Hsa platform system architecture specification, 2015.
2. Bacon, D., Rabbah, R., Shukla, S. (2013). FPGA programming for the masses. *Commun. ACM*, *56*(4), 56–63. https://doi.org/10.1145/2436256.2436271.
3. Czajkowski, T., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D. (2012). From openCL to high-performance hardware on FPGAs. In: Proc. Int. Conf. on field programmable logic and applications, pp. 531–534. https://doi.org/10.1109/FPL.2012.6339272.
4. Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K. (2007). Image denoising by sparse 3-d transform-domain collaborative filtering. *IEEE Transactions on Image Processing*, *16*(8), 2080–2095. https://doi.org/10.1109/TIP.2007.901238.
5. Intel: Intel expands customer choice with first configurable intel® atom™-based processor. Press release, 2015. http://newsroom.intel.com/.
6. Jääskeläinen, P., de La Lama, C.S., Schnetter, E., Raiskila, K., Takala, J., Berg, H. (2015). pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming*, *43*(5), 752–785. https://doi.org/10.1007/s10766-014-0320-y.
7. Kaxiras, S., & Keramidas, G. (2010). SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro*, *30*(5), 54–65. https://doi.org/10.1109/MM.2010.82.
8. Kayi, A., Serres, O., El-Ghazawi, T. (2015). Adaptive cache coherence mechanisms with producer-consumer sharing optimization for chip multiprocessors. *IEEE Transactions on Computers*, *64*(2), 316–328. 10.1109/TC.2013.217.
9. Khronos Group (2012). Beaverton, OR: OpenCL Specification v1.2r19 edn.
10. Khronos Group (2015). Beaverton, OR: OpenCL Specification v2.0 edn.
11. Khronos Group (2015). Beaverton, OR: OpenCL Specification v2.1 edn.
12. Movidius: Software Development Kit (web page). http://www.movidius.com/solutions/software-development-kit.
13. Singh, I., Shriraman, A., Fung, W., O'Connor, M., Aamodt, T. (2013). Cache coherence for gpu architectures. In: 2013 IEEE 19th international symposium on high performance computer architecture (HPCA2013), pp. 578–590. https://doi.org/10.1109/HPCA.2013.6522351.
14. Synopsys: ASIP designer – application-specific processor design made easy. web (2015). https://www.synopsys.com/dw/doc.php/ds/cc/asip-brochure.pdf.
15. Texas Instruments: OpenCL Runtime Documentation v01.01.xx (2015). http://downloads.ti.com/mctools/esd/docs/opencl.
16. Xilinx, Inc.: Xilinx UltraScale Architecture and Product Overview (2015). DS890 (v2.6).
17. Yang, T., & Gerasoulis, A. (1991). A fast static scheduling algorithm for dags on an unbounded number of processors. In: Proceedings of the 1991 ACM/IEEE conference on supercomputing, 1991. Supercomputing '91, pp. 633–642. https://doi.org/10.1145/125826.126138.

Pekka Jääskeläinen received his M.Sc. and D.Sc.(Tech.) degrees from TUT in 2005 and 2012, respectively, and an Adjunct Professorship from University of Oulu in 2017. Currently he leads the Customized Parallel Computing (CPC) group. On top of publication activities, he is a contributor to heterogeneous parallel platform related open source projects acting as the lead developer of TTA-Based Co-design Environment (http://tce.cs.tut.fi) and Portable Computing Language (http://portablecl.org) projects. His current research interests include methods and tools to reduce the engineering effort involved in design and efficient programming of diverse heterogeneous platforms, and hardware and compiler techniques to reduce the energy consumption of the softwarebased control of processors.

Ville Korhonen was part of the Customized Parallel Computing (CPC) group of Tampere University of Technology through years 2013 to 2017. In CPC he focused on researching parallel programming of heterogeneous platforms in the context of OpenCL API and the Portable Computing Language (pocl) project. Currently he has moved to industry and is working as an embedded software developer in Wapice Ltd. Finland.

Matias Koskela received his bachelor's and master's degrees with honors from Tampere University of Technology in 2014 and in 2015, respectively and is now pursuing his doctoral degree. His research interests include optimizations and parallelism in real-time implementations, especially focusing in real-time ray tracing.

Jarmo Takala received his M.Sc.(hons) and D.Sc.(Tech.) degrees from Tampere University of Technology, Tampere, Finland (TUT) in 1987 and 1999, respectively. He joined VTT-Automation, Tampere, Finland in 1992, Nokia Research Center, Finland in 1995. He joined TUT in 1996 and he has been Professor in Computer Engineering at TUT since 2000. During 2007- 2011 he was Associate Editor and Area Editor for IEEE Trans. Signal Process. and in 2012-2013 he was the Chair of IEEE Signal Processing Society's Design and Implementation of Signal Processing Systems Technical Committee.

Karen Egiazarian received M.Sc. in mathematics from Yerevan State University, Armenia, in 1981, the Ph.D. degree in physics and mathematics from Moscow State University, Russia, in 1986, and a Doctor of Technology from Tampere University of Technology (TUT), in 1994. In 2015 he has received the Honorary Doctoral degree from Don State-Technical University (Rostov-Don, Russia). Dr. Egiazarian is a co-founder and CEO of Noiseless Imaging Oy (Ltd), a TUT spin-off company. He is a Professor at Signal Processing Laboratory, TUT, leading the Computational Imaging group, head of Signal Processing Research Community (SPRC) at TUT, and an Adjunct Professor in the Department of Information Technology, University of Jyv?skyl? (Finland). His main research interests are in the field of computational imaging, compressed sensing, efficient signal processing algorithms, image/video restoration and compression. Dr. Egiazarian has published over 700 refereed journal and conference articles, books and patents in these fields. He is an Editorin- Chief of Journal of Electronic Imaging (SPIE) and a Member of the DSP Technical Committee of the IEEE Circuits and Systems Society.

Aram Danielyan received his Ph.D. degree in Signal Processing from Tampere University of Technology in 2013. He has been working as a Senior Imaging Algorithm Engineer in Noiseless Imaging ltd. and poLight ltd. Since 2018 he is with Intel Finland. His professional interests include image and video processing and efficient software implementations of imaging algorithms.

Cristóvão Cruz completed his masters degree in Electronic and Telecommunications Engineering at University of Aveiro in 2014. He is currently working at Noiseless Imaging Oy (Ltd) as an algorithms engineer and is enrolled on the doctoral programme of Computing and Electrical Engineering at Tampere University of Technology. His current research is focused on the design and implementation of image restoration algorithms.

James Price is a Research Associate in the High Performance Computing group at the University of Bristol. His research focuses on improving the programmability of modern, many-core computer architectures. James is currently a Research Software Engineer for the Isambard HPC facility, and is heavily involved in porting, benchmarking and optimizing codes for Arm processors. He has a PhD in Computer Science from the University of Bristol.

Simon McIntosh-Smith is a full Professor of High Performance Computing at the University of Bristol in the UK. He began his career as a microprocessor architect at Inmos and STMicroelectronics in the early 1990s, before co-designing the world's first fully programmable GPU at Pixelfusion in 1999. In 2002 he co-founded ClearSpeed Technology where, as Director of Architecture and Applications, he co-developed the first modern many-core HPC accelerators. He now leads the High Performance Computing Research Group at the University of Bristol, where his research focuses on performance portability and application based fault tolerance. He plays a key role in designing and procuring HPC services at the local, regional and national level, including the UK's national HPC server, Archer. In 2016 he led the successful bid by the GW4 Alliance along with the UK?s Met Office and Cray, to design and build 'Isambard', the world?s first production, ARMv8-based supercomputer.