Cooperative Kernels: GPU Multitasking for Blocking Algorithms

Tyler Sorensen
Imperial College London
London, UK
t.sorensen15@imperial.ac.uk

Hugues Evrard
Imperial College London
London, UK
h.evrard@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
London, UK
alastair.donaldson@imperial.ac.uk

ABSTRACT

There is growing interest in accelerating irregular data-parallel algorithms on GPUs. These algorithms are typically *blocking*, so they require fair scheduling. But GPU programming models (e.g. OpenCL) do not mandate fair scheduling, and GPU schedulers are unfair in practice. Current approaches avoid this issue by exploiting scheduling quirks of today's GPUs in a manner that does not allow the GPU to be shared with other workloads (such as graphics rendering tasks). We propose *cooperative kernels*, an extension to the traditional GPU programming model geared towards writing blocking algorithms. Workgroups of a cooperative kernel *are* fairly scheduled, and multitasking is supported via a small set of language extensions through which the kernel and scheduler cooperate. We describe a prototype implementation of a cooperative kernel framework implemented in OpenCL 2.0 and evaluate our approach by porting a set of blocking GPU applications to cooperative kernels and examining their performance under multitasking.

CCS CONCEPTS

• **Software and its engineering** → **Multiprocessing / multi-programming / multitasking**; Semantics; • **Computing methodologies** → *Graphics processors*;

KEYWORDS

GPU, cooperative multitasking, irregular parallelism

1 INTRODUCTION

The Needs of Irregular Data-parallel Algorithms. Many interesting data-parallel algorithms are *irregular*: the amount of work to be processed is unknown ahead of time and may change dynamically in a workload-dependent manner. There is growing interest in accelerating such algorithms on GPUs [1–4, 6, 7, 11, 13, 14, 16, 19, 21, 25, 26, 30, 32]. Irregular algorithms usually require *blocking synchronisation* between workgroups, e.g. many graph algorithms use a level-by-level strategy, with a global barrier between levels; work stealing algorithms require each workgroup to maintain a queue, typically mutex-protected, to enable stealing by other workgroups.

To avoid starvation, a blocking concurrent algorithm requires *fair* scheduling of workgroups. For example, if one workgroup holds a mutex, an unfair scheduler may cause another workgroup to spin-wait forever for the mutex to be released. Similarly, an unfair scheduler can cause a workgroup to spin-wait indefinitely at a global barrier so that other workgroups do not reach the barrier.

A Degree of Fairness: Occupancy-bound Execution. The current GPU programming models—OpenCL [12], CUDA [18] and HSA [10], specify almost no guarantees regarding scheduling of
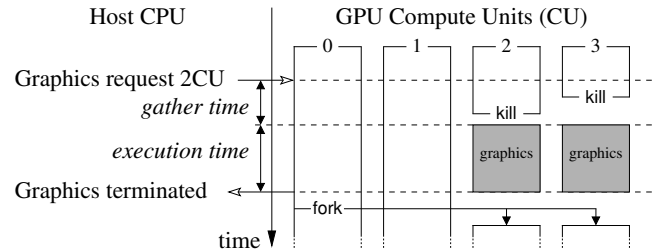

Figure 1: Cooperative kernels can flexibly resize to let other tasks, e.g. graphics, run concurrently

workgroups, and current GPU schedulers are unfair in practice. Roughly speaking, each workgroup executing a GPU kernel is mapped to a hardware *compute unit*.[1] The simplest way for a GPU driver to handle more workgroups being launched than there are compute units is via an *occupancy-bound* execution model [6, 26] where, once a workgroup has commenced execution on a compute unit (it has become *occupant*), the workgroup has exclusive access to the compute unit until it finishes execution. Experiments suggest that this model is widely employed by today's GPUs [1, 6, 19, 26].

The occupancy-bound execution model does not guarantee fair scheduling between workgroups: if all compute units are occupied then a not-yet-occupant workgroup will not be scheduled until some occupant workgroup completes execution. Yet the execution model *does* provide fair scheduling between *occupant* workgroups, which are bound to separate compute units that operate in parallel. Current GPU implementations of blocking algorithms assume the occupancy-bound execution model, which they exploit by launching no more workgroups than there are available compute units [6].

Resistance to Occupancy-bound Execution. Despite its practical prevalence, none of the current GPU programming models actually mandate occupancy-bound execution. Further, there are reasons why this model is undesirable. First, the execution model does not enable multitasking, since a workgroup effectively *owns* a compute unit until the workgroup has completed execution. The GPU cannot be used meanwhile for other tasks (e.g. rendering). Second, *energy throttling* is an important concern for battery-powered devices [31]. In the future, it will be desirable for a mobile GPU driver to power down some compute units, suspending execution of associated occupant workgroups, if the battery level is low.

Our assessment, informed by discussions with a number of industrial practitioners who have been involved in the OpenCL and/or HSA standardisation efforts (including [9, 22]), is that GPU vendors (1) will not commit to the occupancy-bound execution model

---

[1]In practice, depending on the kernel, multiple workgroups might map to the same compute unit; we ignore this in our current discussion.

they currently implement, for the above reasons, yet (2) will not guarantee fair scheduling using preemption. This is due to the high runtime cost of preempting workgroups, which requires managing thread local state (e.g. registers, program location) for all workgroup threads (up to 1024 on Nvidia GPUs), as well as *shared memory*, the workgroup local cache (up to 64 KB on Nvidia GPUs). Vendors instead wish to retain the essence of the simple occupancy-bound model, supporting preemption only in key special cases.

For example, preemption is supported by Nvidia's Pascal architecture [17], but on a GTX Titan X (Pascal) we still observe starvation: a global barrier executes successfully with 56 workgroups, but deadlocks with 57 workgroups, indicating unfair scheduling.

Our Proposal: Cooperative Kernels. To summarise: blocking algorithms demand fair scheduling, but for good reasons GPU vendors will not commit to the guarantees of the occupancy-bound execution model. We propose *cooperative kernels*, an extension to the GPU programming model that aims to resolve this impasse.

A kernel that requires fair scheduling is identified as *cooperative*, and written using two additional language primitives, offer_kill and request_fork, placed by the programmer. Where the cooperative kernel could proceed with fewer workgroups, a workgroup can execute offer_kill, offering to sacrifice itself to the scheduler. This indicates that the workgroup would ideally continue executing, but that the scheduler may preempt the workgroup; the cooperative kernel must be prepared to deal with either scenario. Where the cooperative kernel could use additional resources, a workgroup can execute request_fork to indicate that the kernel is prepared to proceed with the existing set of workgroups, but is able to benefit from one or more additional workgroups commencing execution directly after the request_fork program point.

The use of request_fork and offer_kill creates a contract between the scheduler and the cooperative kernel. Functionally, the scheduler must guarantee that the workgroups executing a cooperative kernel are fairly scheduled, while the cooperative kernel must be robust to workgroups leaving and joining the computation in response to offer_kill and request_fork. Non-functionally, a cooperative kernel must ensure that offer_kill is executed frequently enough such that the scheduler can accommodate soft-real time constraints, e.g. allowing a smooth frame-rate for graphics. In return, the scheduler should allow the cooperative kernel to utilise hardware resources where possible, killing workgroups only when demanded by other tasks, and forking additional workgroups when possible.

Cooperative kernels allow for *cooperative multitasking* (see Sec. 6), used historically when preemption was not available or too costly. Our approach avoids the cost of arbitrary preemption as the state of a workgroup killed via offer_kill does not have to be saved. Previous cooperative multitasking systems have provided *yield* semantics, where a processing unit would temporarily give up its hardware resource. We deviate from this design as, in the case of a global barrier, adopting yield would force the cooperative kernel to block *completely* when a single workgroup yields, stalling the kernel until the given workgroup resumes. Instead, our offer_kill allows a kernel to make progress with a smaller number of workgroups, with workgroups potentially joining again later via request_fork.

Figure 1 illustrates sharing of GPU compute units between a cooperative kernel and a graphics task. Workgroups 2 and 3 of

the cooperative kernel are killed at an offer_kill to make room for a graphics task. The workgroups are subsequently restored to the cooperative kernel when workgroup 0 calls request_fork. The *gather time* is the time between resources being requested and the application surrendering them via offer_kill. To satisfy soft-real time constraints, this time should be low; our experimental study (Sec. 5.4) shows that, in practice, the gather-time for our applications is acceptable for a range of graphics workloads.

The cooperative kernels model has several appealing properties:

(1) By providing fair scheduling between workgroups, cooperative kernels meet the needs of blocking algorithms, including irregular data-parallel algorithms.

(2) The model has no impact on the development of regular (non-cooperative) compute and graphics kernels.

(3) The model is backwards-compatible: offer_kill and request_fork may be ignored, and a cooperative kernel will behave exactly as a regular kernel does on current GPUs.

(4) Cooperative kernels can be implemented over the occupancy-bound execution model provided by current GPUs: our prototype implementation uses no special hardware/driver support.

(5) If hardware support for preemption *is* available, it can be leveraged to implement cooperative kernels efficiently, and cooperative kernels can avoid unnecessary preemptions by allowing the programmer to communicate "smart" preemption points.

Placing the primitives manually is straightforward for the representative set of GPU-accelerated irregular algorithms we have ported so far. Our experiments show that the model can enable efficient multitasking of cooperative and non-cooperative tasks.

In summary, our main contributions are: *cooperative kernels*, an extended GPU programming model that supports the scheduling requirements of blocking algorithms (Sec. 3); a *prototype implementation* of cooperative kernels on top of OpenCL 2.0 (Sec. 4); and *experiments* assessing the overhead and responsiveness of the cooperative kernels approach over a set of irregular algorithms (Sec. 5), including a best-effort comparison with the efficiency afforded by hardware-supported preemption available on Nvidia GPUs.

We begin by providing background on OpenCL via two motivating examples (Sec. 2). At the end we discuss related work (Sec. 6) and avenues for future work (Sec. 7).

2 BACKGROUND AND EXAMPLES

We outline the OpenCL programming model on which we base cooperative kernels (Sec. 2.1), and illustrate OpenCL and the scheduling requirements of irregular algorithms using two examples: a work stealing queue and frontier-based graph traversal (Sec. 2.2).

2.1 OpenCL Background

An OpenCL program is divided into *host* and *device* components. A host application runs on the CPU and launches one or more *kernels* that run on accelerator devices—GPUs in the context of this paper. A kernel is written in OpenCL C, based on C99. All threads executing a kernel start at the same entry function with identical arguments. A thread can call get_global_id to obtain a unique id, to access distinct data or follow different control flow paths.

The threads of a kernel are divided into *workgroups*. Functions get_local_id and get_group_id return a thread's local id within its

```
1  kernel work_stealing(global Task * queues) {
2    int queue_id = get_group_id();
3    while (more_work(queues)) {
4      Task * t = pop_or_steal(queues, queue_id);
5      if (t)
6        process_task(t, queues, queue_id);
7    }
8  }
```

Figure 2: An excerpt of a work stealing algorithm in OpenCL

```
1  kernel graph_app(global graph * g,
2        global nodes * n0, global nodes * n1) {
3    int level = 0;
4    global nodes * in_nodes = n0;
5    global nodes * out_nodes = n1;
6    int tid = get_global_id();
7    int stride = get_global_size();
8    while(in_nodes.size > 0) {
9      for (int i = tid; i < in_nodes.size; i += stride)
10       process_node(g, in_nodes[i], out_nodes, level);
11     swap(&in_nodes, &out_nodes);
12     global_barrier();
13     reset(out_nodes);
14     level++;
15     global_barrier();
16   }
17 }
```

Figure 3: An OpenCL graph traversal algorithm

workgroup and the workgroup id. The number of threads per workgroup and number of workgroups are obtained via get_local_size and get_num_groups. Execution of the threads in a workgroup can be synchronised via a workgroup barrier. A *global* barrier (synchronising all threads of a kernel) is *not* provided as a primitive.

Memory Spaces and Memory Model. A kernel has access to four memory spaces. *Shared virtual memory* (SVM) is accessible to all device threads and the host concurrently. *Global* memory is shared among all device threads. Each workgroup has a portion of *local* memory for fast intra-workgroup communication. Every thread has a portion of very fast *private* memory for function-local variables.

Fine-grained communication within a workgroup, as well as inter-workgroup communication and communication with the host while the kernel is running, is enabled by a set of atomic data types and operations. In particular, fine-grained host/device communication is via atomic operations on SVM.

Execution Model. OpenCL [12, p. 31] and CUDA [18] specifically make no guarantees about fair scheduling between workgroups executing the same kernel. HSA provides limited, one-way guarantees, stating [10, p. 46]: *"Work-group A can wait for values written by work-group B without deadlock provided ... (if) A comes after B in work-group flattened ID order"*. This is not sufficient to support blocking algorithms that use mutexes and inter-workgroup barriers, both of which require *symmetric* communication between threads.

2.2 Motivating Examples

Work Stealing. Work stealing enables dynamic balancing of tasks across processing units. It is useful when the number of tasks to be processed is dynamic, due to one task creating an arbitrary number of new tasks. Work stealing has been explored in the context of GPUs [2, 30]. Each workgroup has a queue from which it obtains tasks to process, and to which it stores new tasks. If its queue is empty, a workgroup tries to *steal* a task from another queue.

Figure 2 illustrates a work stealing kernel. Each thread receives a pointer to the task queues, in global memory, initialised by the host to contain initial tasks. A thread uses its workgroup id (line 2) as a queue id to access the relevant task queue. The pop_or_steal function (line 4) pops a task from the workgroup's queue or tries to steal a task from other queues. Although not depicted here, concurrent accesses to queues inside more_work and pop_or_steal are guarded by a mutex per queue, implemented using atomic compare and swap operations on global memory.

If a task is obtained, then the workgroup processes it (line 6), which may lead to new tasks being created and pushed to the

workgroup's queue. The kernel presents two opportunities for spin-waiting: spinning to obtain a mutex, and spinning in the main kernel loop to obtain a task. Without fair scheduling, threads waiting for the mutex might spin indefinitely, causing the application to hang.

Graph Traversal. Figure 3 illustrates a frontier-based graph traversal algorithm; such algorithms have been shown to execute efficiently on GPUs [1, 19]. The kernel is given three arguments in global memory: a graph structure, and two arrays of graph nodes. Initially, n0 contains the starting nodes to process. Private variable level records the current frontier level, and in_nodes and out_nodes point to distinct arrays recording the nodes to be processed during the current and next frontier, respectively.

The application iterates as long as the current frontier contains nodes to process (line 8). At each frontier, the nodes to be processed are evenly distributed between threads through *stride* based processing. In this case, the stride is the total number of threads, obtained via get_global_size. A thread calls process_node to process a node given the current level, with nodes to be processed during the next frontier being pushed to out_nodes. After processing the frontier, the threads swap their node array pointers (line 11).

At this point, the GPU threads must wait for all other threads to finish processing the frontier. To achieve this, we use a global barrier construct (line 12). After all threads reach this point, the output node array is reset (line 13) and the level is incremented. The threads use another global barrier to wait until the output node is reset (line 15), after which they continue to the next frontier.

The global barrier used in this application is not provided as a GPU primitive, though previous works have shown that such a global barrier can be implemented [26, 34], based on CPU barrier designs [8, ch. 17]. These barriers employ spinning to ensure each thread waits at the barrier until all threads have arrived, thus fair scheduling between workgroups is required for the barrier to operate correctly. Without fair scheduling, the barrier threads may wait indefinitely at the barrier, causing the application to hang.

The mutexes and barriers used by these two examples appear to run reliably on current GPUs for kernels that are executed with no more workgroups than there are compute units. This is due to the fairness of the occupancy-bound execution model that current

GPUs have been shown, experimentally, to provide. But, as discussed in Sec. 1, this model is not endorsed by language standards or vendor implementations, and may not be respected in the future.

In Sec. 3.2 we show how the work stealing and graph traversal examples of Figs. 2 and 3 can be updated to use our cooperative kernels programming model to resolve the scheduling issue.

3 COOPERATIVE KERNELS

We describe the semantics of our cooperative kernels programming model (Sec. 3.1), use our motivating examples to discuss programmability (Sec. 3.2) and outline nonfunctional properties required by the model (Sec. 3.3). In the extended version of this paper we present a formal semantics, and discuss alternative semantic choices [27].

3.1 Semantics of Cooperative Kernels

As with a regular OpenCL kernel, a cooperative kernel is launched by the host application, passing parameters to the kernel and specifying a desired number of threads and workgroups. Unlike in a regular kernel, the parameters to a cooperative kernel are immutable (though pointer parameters can refer to mutable data).

Cooperative kernels are written using the following extensions: transmit, a qualifier on the variables of a thread; offer_kill and request_fork, the key functions that enable cooperative scheduling; and global_barrier and resizing_global_barrier primitives for inter-workgroup synchronisation.

Transmitted Variables. A variable declared in the root scope of the cooperative kernel can optionally be annotated with a new transmit qualifier. Annotating a variable v with transmit means that when a workgroup uses request_fork to spawn new workgroups, the workgroup should transmit its current value for v to the threads of the new workgroups. We detail the semantics for this when we describe request_fork below.

Active Workgroups. If the host application launches a cooperative kernel requesting N workgroups, this indicates that the kernel should be executed with a *maximum* of N workgroups, and that as many workgroups as possible, up to this limit, are desired. However, the scheduler may initially schedule fewer than N workgroups, and as explained below the number of workgroups that execute the cooperative kernel can change during the lifetime of the kernel.

The number of *active workgroups*—workgroups executing the kernel—is denoted M. Active workgroups have consecutive ids in the range $[0, M-1]$. Initially, at least one workgroup is active; if necessary the scheduler must postpone the kernel until some compute unit becomes available. For example, in Fig. 1: at the beginning of the execution $M = 4$; while the graphics task is executing $M = 2$; after the fork $M = 4$ again.

When executed by a cooperative kernel, get_num_groups returns M, the *current* number of active workgroups. This is in contrast to get_num_groups for regular kernels, which returns the fixed number of workgroups that execute the kernel (see Sec. 2.1).

Fair scheduling *is* guaranteed between active workgroups; i.e. if some thread in an active workgroup is enabled, then eventually this thread is guaranteed to execute an instruction.

Semantics for offer_kill. The offer_kill primitive allows the cooperative kernel to return compute units to the scheduler by offering

to sacrifice workgroups. The idea is as follows: allowing the scheduler to arbitrarily and abruptly terminate execution of workgroups might be drastic, yet the kernel may contain specific program points at which a workgroup could *gracefully* leave the computation.

Similar to the OpenCL workgroup barrier primitive, offer_kill, is a workgroup-level function—it must be encountered uniformly by all threads in a workgroup.

Suppose a workgroup with id m executes offer_kill. If the workgroup has the largest id among active workgroups then it can be killed by the scheduler, except that workgroup 0 can never be killed (to avoid early termination of the kernel). More formally, if $m < M - 1$ or $M = 1$ then offer_kill is a no-op. If instead $M > 1$ and $m = M - 1$, the scheduler can choose to ignore the offer, so that offer_kill executes as a no-op, or accept the offer, so that execution of the workgroup ceases and the number of active workgroups M is atomically decremented by one. Figure 1 illustrates this, showing that workgroup 3 is killed before workgroup 2.

Semantics for request_fork. Recall that a desired limit of N workgroups was specified when the cooperative kernel was launched, but that the number of active workgroups, M, may be smaller than N, either because (due to competing workloads) the scheduler did not provide N workgroups initially, or because the kernel has given up some workgroups via offer_kill calls. Through the request_fork primitive (also a workgroup-level function), the kernel and scheduler can collaborate to allow new workgroups to join the computation at an appropriate point and with appropriate state.

Suppose a workgroup with id $m \le M$ executes request_fork. Then the following occurs: an integer $k \in [0, N - M]$ is chosen by the scheduler; k new workgroups are spawned with consecutive ids in the range $[M, M + k - 1]$; the active workgroup count M is atomically incremented by k.

The k new workgroups commence execution at the program point immediately following the request_fork call. The variables that describe the state of a thread are all uninitialised for the threads in the new workgroups; reading from these variables without first initialising them is an undefined behaviour. There are two exceptions to this: (1) because the parameters to a cooperative kernel are immutable, the new threads have access to these parameters as part of their local state and can safely read from them; (2) for each variable v annotated with transmit, every new thread's copy of v is initialised to the value that thread 0 in workgroup m held for v at the point of the request_fork call. In effect, thread 0 of the forking workgroup *transmits* the relevant portion of its local state to the threads of the forked workgroups.

Figure 1 illustrates the behaviour of request_fork. After the graphics task finishes executing, workgroup 0 calls request_fork, spawning the two new workgroups with ids 2 and 3. Workgroups 2 and 3 join the computation where workgroup 0 called request_fork.

Notice that $k = 0$ is always a valid choice for the number of workgroups to be spawned by request_fork, and is guaranteed if M is equal to the workgroup limit N.

Global Barriers. Because workgroups of a cooperative kernel are fairly scheduled, a global barrier primitive can be provided. We specify two variants: global_barrier and resizing_global_barrier.

Our global_barrier primitive is a kernel-level function: if it appears in conditional code then it must be reached by *all* threads

executing the cooperative kernel. On reaching a global_barrier, a thread waits until all threads have arrived at the barrier. Once all threads have arrived, the threads may proceed past the barrier with the guarantee that all global memory accesses issued before the barrier have completed. The global_barrier primitive can be implemented by adapting an inter-workgroup barrier design, e.g. [34], to take account of a growing and shrinking number of workgroups, and the atomic operations provided by the OpenCL 2.0 memory model enable a memory-safe implementation [26].

The resizing_global_barrier primitive is also a kernel-level function. It is identical to global_barrier, except that it caters for cooperation with the scheduler: by issuing a resizing_global_barrier the programmer indicates that the cooperative kernel is prepared to proceed after the barrier with more or fewer workgroups.

When all threads have reached resizing_global_barrier, the number of active workgroups, M, is atomically set to a new value, M' say, with $0 < M' \leq N$. If $M' = M$ then the active workgroups remain unchanged. If $M' < M$, workgroups $[M', M-1]$ are killed. If $M' > M$ then $M' - M$ new workgroups join the computation after the barrier, as if they were forked from workgroup 0. In particular, the transmit-annotated local state of thread 0 in workgroup 0 is transmitted to the threads of the new workgroups.

The semantics of resizing_global_barrier can be modelled via calling request_fork and offer_kill, surrounded and separated by calls to a global_barrier. The enclosing global_barrier calls ensure that the change in number of active workgroups from M to M' occurs entirely within the resizing barrier, so that M changes atomically from a programmer's perspective. The middle global_barrier ensures that forking occurs before killing, so that workgroups $[0, \min(M, M') - 1]$ are left intact.

Because resizing_global_barrier can be implemented as above, we do not regard it *conceptually* as a primitive of our model. However, in Sec. 4.2 we show how a resizing barrier can be implemented more efficiently through direct interaction with the scheduler.

3.2 Programming with Cooperative Kernels

A Changing Workgroup Count. Unlike in regular OpenCL, the value returned by get_num_groups is not fixed during the lifetime of a cooperative kernel: it corresponds to the active group count M, which changes as workgroups execute offer_kill, and request_fork. The value returned by get_global_size is similarly subject to change. A cooperative kernel must thus be written in a manner that is robust to changes in the values returned by these functions.

In general, their volatility means that use of these functions should be avoided. However, the situation is more stable if a cooperative kernel does not call offer_kill and request_fork directly, so that only resizing_global_barrier can affect the number of active workgroups. Then, at any point during execution, the threads of a kernel are executing between some pair of resizing barrier calls, which we call a *resizing barrier interval* (considering the kernel entry and exit points conceptually to be special cases of resizing barriers). The active workgroup count is constant within each resizing barrier interval, so that get_num_groups and get_global_size return stable values during such intervals. As we illustrate below for graph traversal, this can be exploited by algorithms that perform strided data processing.

```
1  kernel graph_app(global graph *g,
2        global nodes *n0, global nodes *n1) {
3    transmit int level = 0;
4    transmit global nodes *in_nodes = n0;
5    transmit global nodes *out_nodes = n1;
6    while(in_nodes.size > 0) {
7      int tid = get_global_id();
8      int stride = get_global_size();
9      for (int i = tid; i < in_nodes.size; i += stride)
10       process_node(g, in_nodes[i], out_nodes, level);
11     swap(&in_nodes, &out_nodes);
12     resizing_global_barrier();
13     reset(out_nodes);
14     level++;
15     resizing_global_barrier();
16   }
17 }
```

Figure 4: Cooperative version of the graph traversal kernel of Fig. 3, using a resizing barrier and transmit **annotations**

Adapting Work Stealing. In this example there is no state to transmit since a computation is entirely parameterised by a task, which is retrieved from a queue located in global memory. With respect to Fig. 2, we add request_fork and offer_kill calls at the start of the main loop (below line 3) to let a workgroup offer itself to be killed or forked, respectively, before it processes a task. Note that a workgroup may be killed even if its associated task queue is not empty, since remaining tasks will be stolen by other workgroups. In addition, since request_fork may be the entry point of a workgroup, the queue id must now be computed after it, so we move line 2 to be placed just before line 4. In particular, the queue id cannot be transmitted since we want a newly spawned workgroup to read its own queue and not the one of the forking workgroup.

Adapting Graph Traversal. Figure 4 shows a cooperative version of the graph traversal kernel of Fig. 3 from Sec. 2.2. On lines 12 and 15, we change the original global barriers into a resizing barriers. Several variables are marked to be transmitted in the case of workgroups joining at the resizing barriers (lines 3, 4 and 5): level must be restored so that new workgroups know which frontier they are processing; in_nodes and out_nodes must be restored so that new workgroups know which of the node arrays to use for input and output. Lastly, the static work distribution of the original kernel is no longer valid in a cooperative kernel. This is because the stride (which is based on M) may change after each resizing barrier call. To fix this, we re-distribute the work after each resizing barrier call by recomputing the thread id and stride (lines 7 and 8). This example exploits the fact that the cooperative kernel does not issue offer_kill nor request_fork directly: the value of stride obtained from get_global_size at line 8 is stable until the next resizing barrier at line 12.

Patterns for Irregular Algorithms. In Sec. 5.1 we describe the set of irregular GPU algorithms used in our experiments, which largely captures the irregular blocking algorithms that are available as open source GPU kernels. These all employ either work stealing or operate on graph data structures, and placing our new constructs follows a common, easy-to-follow pattern in each case. The work stealing algorithms have a transactional flavour and require little

or no state to be carried between transactions. The point at which a workgroup is ready to process a new task is a natural place for offer_kill and request_fork, and few or no transmit annotations are required. Figure 4 is representative of most level-by-level graph algorithms. It is typically the case that on completing a level of the graph algorithm, the next level could be processed by more or fewer workgroups, which resizing_global_barrier facilitates. Some level-specific state must be transmitted to new workgroups.

3.3 Non-Functional Requirements

The semantics presented in Sec. 3.1 describe the behaviours that a developer of a cooperative kernel should be prepared for. However, the aim of cooperative kernels is to find a balance that allows *efficient* execution of algorithms that require fair scheduling, and *responsive* multitasking, so that the GPU can be shared between cooperative kernels and other shorter tasks with soft real-time constraints. To achieve this balance, an implementation of the cooperative kernels model, and the programmer of a cooperative kernel, must strive to meet the following non-functional requirements.

The purpose of offer_kill is to let the scheduler destroy a workgroup in order to schedule higher-priority tasks. The scheduler relies on the cooperative kernel to execute offer_kill sufficiently frequently that soft real-time constraints of other workloads can be met. Using our work stealing example: a workgroup offers itself to the scheduler after processing each task. If tasks are sufficiently fast to process then the scheduler will have ample opportunities to de-schedule workgroups. But if tasks are very time-consuming to process then it might be necessary to rewrite the algorithm so that tasks are shorter and more numerous, to achieve a higher rate of calls to offer_kill. Getting this non-functional requirement right is GPU- and application-dependent. In Sec. 5.2 we conduct experiments to understand the response rate that would be required to co-schedule graphics rendering with a cooperative kernel, maintaining a smooth frame rate.

Recall that, on launch, the cooperative kernel requests N workgroups. The scheduler should thus aim to provide N workgroups if other constraints allow it, by accepting an offer_kill only if a compute unit is required for another task, and responding positively to request_fork calls if compute units are available.

4 PROTOTYPE IMPLEMENTATION

Our vision is that cooperative kernel support will be integrated in the runtimes of future GPU implementations of OpenCL, with driver support for our new primitives. To experiment with our ideas on current GPUs, we have developed a prototype that mocks up the required runtime support via a *megakernel*, and exploits the occupancy-bound execution model that these GPUs provide to ensure fair scheduling between workgroups. We emphasise that an aim of cooperative kernels is to *avoid* depending on the occupancy-bound model. Our prototype exploits this model simply to allow us to experiment with current GPUs whose proprietary drivers we cannot change. We describe the megakernel approach (Sec. 4.1) and detail various aspects of the scheduler component of our implementation (Sec. 4.2).

4.1 The Megakernel Mock Up

Instead of multitasking multiple separate kernels, we merge a set of kernels into a megakernel—a single, monolithic kernel. The megakernel is launched with as many workgroups as can be occupant concurrently. One workgroup takes the role of the scheduler,[2] and the scheduling logic is embedded as part of the megakernel. The remaining workgroups act as a pool of workers. A worker repeatedly queries the scheduler to be assigned a task. A task corresponds to executing a cooperative or non-cooperative kernel. In the non-cooperative case, the workgroup executes the relevant kernel function uninterrupted, then awaits further work. In the cooperative case, the workgroup either starts from the kernel entry point or immediately jumps to a designated point within the kernel, depending on whether the workgroup is an initial workgroup of the kernel, or a forked workgroup. In the latter case, the new workgroup also receives a struct containing the values of all relevant transmit-annotated variables.

Simplifying Assumptions. For ease of implementation, our prototype supports multitasking a single cooperative kernel with a single non-cooperative kernel (though the non-cooperative kernel can be invoked many times). We require that offer_kill, request_fork and resizing_global_barrier are called from the entry function of a cooperative kernel. This allows us to use goto and return to direct threads into and out of the kernel. With these restrictions we can experiment with interesting irregular algorithms (see Sec. 5). A non-mock implementation of cooperative kernels would not use the megakernel approach, so we did not deem the engineering effort associated with lifting these restrictions in our prototype to be worthwhile.

4.2 Scheduler Design

To enable multitasking through cooperative kernels, the runtime (in our case, the megakernel) must track the state of workgroups, i.e. whether a workgroup is waiting or computing a kernel; maintain consistent context states for each kernel, e.g. tracking the number of active workgroups; and provide a safe way for these states to be modified in response to request_fork/offer_kill. We discuss these issues, and describe the implementation of an efficient resizing barrier. We describe how the scheduler would handle arbitrary combinations of kernels, though as noted above our current implementation is restricted to the case of two kernels.

Scheduler Contexts. To dynamically manage workgroups executing cooperative kernels, our framework must track the state of each workgroup and provide a channel of communication from the scheduler workgroup to workgroups executing request_fork and offer_kill. To achieve this, we use a *scheduler context* structure, mapping a primitive workgroup id the workgroup's status, which is either *available* or the id of the kernel that the workgroup is currently executing. The scheduler can then send cooperative kernels a *resource message*, commanding workgroups to exit at offer_kill, or spawn additional workgroups at request_fork. Thus, the scheduler context needs a communication channel for each cooperative

---

[2]We note that the scheduler requirements given in Sec. 3 are agnostic to whether the scheduling logic takes place on the CPU or GPU. To avoid expensive communication between GPU and host, we choose to implement the scheduler on the GPU.

kernel. We implement the communication channels using atomic variables in global memory.

Launching Kernels and Managing Workgroups. To launch a kernel, the host sends a data packet to the GPU scheduler consisting of a kernel to execute, kernel inputs, and a flag indicating whether the kernel is cooperative. In our prototype, this host-device communication channel is built using fine-grained SVM atomics.

On receiving a data packet describing a kernel launch K, the GPU scheduler must decide how to schedule K. Suppose K requests N workgroups. The scheduler queries the scheduler context. If there are at least N available workgroups, K can be scheduled immediately. Suppose instead that there are only $N_a < N$ available workgroups, but a cooperative kernel K_c is executing. The scheduler can use K_c's channel in the scheduler context to command K_c to provide $N - N_a$ workgroups via offer_kill. Once N workgroups are available, the scheduler then sends N workgroups from the available workgroups to execute kernel K. If the new kernel K is itself a cooperative kernel, the scheduler would be free to provide K with fewer than N active workgroups initially.

If a cooperative kernel K_c is executing with fewer workgroups than it initially requested, the scheduler may decide make extra workgroups available to K_c, to be obtained next time K_c calls request_fork. To do this, the scheduler asynchronously signals K_c through K_c's channel to indicate the number of workgroups that should join at the next request_fork command. When a workgroup w of K_c subsequently executes request_fork, thread 0 of w updates the kernel and scheduler contexts so that the given number of new workgroups are directed to the program point after the request_fork call. This involves selecting workgroups whose status is *available*, as well as copying the values of transmit-annotated variables to the new workgroups.

An Efficient Resizing Barrier. In Sec. 3.1, we defined the semantics of a resizing barrier in terms of calls to other primitives. It is possible, however, to implement the resizing barrier with only one call to a global barrier with request_fork and offer_kill inside.

We consider barriers that use the master/slave model [34]: one workgroup (master) collects signals from the other workgroups (slaves) indicating that they have arrived at the barrier and are waiting for a reply indicating that they may leave the barrier. Once the master has received a signal from all slaves, it replies with a signal saying that they may leave.

Incorporating request_fork and offer_kill into such a barrier is straightforward. Upon entering the barrier, the slaves first execute offer_kill, possibly exiting. The master then waits for M slaves (the number of active workgroups), which may decrease due to offer_kill calls by the slaves, but will not increase. Once the master observes that M slaves have arrived, it knows that all other workgroups are waiting to be released. The master executes request_fork, and the statement immediately following this request_fork is a conditional that forces newly spawned workgroups to join the slaves in waiting to be released. Finally, the master releases all the slaves: the original slaves and the new slaves that joined at request_fork.

This barrier implementation is sub-optimal because workgroups only execute offer_kill once per barrier call and, depending on order of arrival, it is possible that only one workgroup is killed per barrier call, preventing the scheduler from gathering workgroups quickly.

Table 1: Blocking GPU applications investigated

App.	barriers	kill	fork	transmit	LoC	inputs
color	2 / 2	0	0	4	55	2
mis	3 / 3	0	0	0	71	2
p-sssp	3 / 3	0	0	0	42	1
bfs	2 / 2	0	0	4	185	2
l-sssp	2 / 2	0	0	4	196	2
octree	0 / 0	1	1	0	213	1
game	0 / 0	1	1	0	308	1
■ Pannotia		■ Lonestar GPU			work stealing	

We can reduce the gather time by providing a new query function for cooperative kernels, which returns the number of workgroups that the scheduler needs to obtain from the cooperative kernel. A resizing barrier can now be implemented as follows: (1) the master waits for all slaves to arrive; (2) the master calls request_fork and commands the new workgroups to be slaves; (3) the master calls query, obtaining a value W; (4) the master releases the slaves, broadcasting the value W to them; (5) workgroups with ids larger than $M - W$ spin, calling offer_kill repeatedly until the scheduler claims them—we know from query that the scheduler will eventually do so. We show in Sec. 5.4 that the barrier using query greatly reduces the gather time in practice.

5 APPLICATIONS AND EXPERIMENTS

We discuss our experience porting irregular algorithms to cooperative kernels and describe the GPUs on which we evaluate these applications (Sec. 5.1). For these GPUs, we report on experiments to determine non-cooperative workloads that model the requirements of various graphics rendering tasks (Sec. 5.2). We then examine the overhead associated with moving to cooperative kernels when multitasking is *not* required (Sec. 5.3), as well as the responsiveness and throughput observed when a cooperative kernel is multi-tasked with non-cooperative workloads (Sec. 5.4). Finally, we compare against a performance model of *kernel-level* preemption, which we understand to be what current Nvidia GPUs provide (Sec. 5.5).

5.1 Applications and GPUs

Table 1 gives an overview of the 7 irregular algorithms that we ported to cooperative kernels. Among them, 5 are graph algorithms, based on the Pannotia [3] and Lonestar [1] GPU application suites, using global barriers. We indicate how many of the original number of barriers are changed to resizing barriers (all of them), and how many variables need to be transmitted. The remaining two algorithms are work stealing applications: each required the addition of request_fork and offer_kill at the start of the main loop, and no variables needed to be transmitted (similar to example discussed in Sec. 3.2). Most graph applications come with 2 different data sets as input, leading to 11 application/input pairs in total.

Our prototype implementation (Sec. 4) requires two optional features of OpenCL 2.0: SVM fine-grained buffers and SVM atomics. Out of the GPUs available to us, from ARM, AMD, Nvidia, and Intel, only Intel GPUs provided robust support of these features.

We thus ran our experiments on three Intel GPUs: HD 520, HD 5500 and Iris 6100. The results were similar across the GPUs,

so for conciseness, we report only on the Iris 6100 GPU (driver 20.19.15.4463) with a host CPU i3-5157U. The Iris has a reported 47 compute units. Results for the other Intel GPUs are available in the extended version of this paper [27].

5.2 Sizing Non-cooperative Kernels

Enabling rendering of smooth graphics in parallel with irregular algorithms is an important use case for our approach. Because our prototype implementation is based on a megakernel that takes over the entire GPU (see Sec. 4), we cannot assess this directly.

We devised the following method to determine OpenCL workloads that simulate the computational intensity of various graphics rendering workloads. We designed a synthetic kernel that occupies all workgroups of a GPU for a parameterised time period t, invoked in an infinite loop by a host application. We then searched for a maximum value for t that allowed the synthetic kernel to execute without having an observable impact on graphics rendering. Using the computed value, we ran the host application for X seconds, measuring the time $Y < X$ dedicated to GPU execution during this period and the number of kernel launches n that were issued. We used $X \geq 10$ in all experiments. The values $(X - Y)/n$ and X/n estimate the average time spent using the GPU to render the display between kernel calls (call this E) and the period at which the OS requires the GPU for display rendering (call this P), respectively.

We used this approach to measure the GPU availability required for three types of rendering: *light*, whereby desktop icons are smoothly emphasised under the mouse pointer; *medium*, whereby window dragging over the desktop is smoothly animated; and *heavy*, which requires smooth animation of a WebGL shader in a browser. For *heavy* we used WebGL demos from the Chrome experiments [5].

Our results are the following: $P = 70\,ms$ and $E = 3\,ms$ for light; $P = 40\,ms$, $E = 3\,ms$ for medium; and $P = 40\,ms$, $E = 10\,ms$ for heavy. For medium and heavy, the $40\,ms$ period coincides with the human persistence of vision. The $3\,ms$ execution duration of both light and medium configurations indicates that GPU computation is cheaper for basic display rendering compared with more complex rendering.

5.3 The Overhead of Cooperative Kernels

Experimental Setup. Invoking the cooperative scheduling primitives incurs some overhead even if no killing, forking or resizing actually occurs, because the cooperative kernel still needs to interact with the scheduler to determine this. We assess this overhead by measuring the slowdown in execution time between the original and cooperative versions of a kernel, forcing the scheduler to never modify the number of active workgroups in the cooperative case.

Recall that our mega kernel-based implementation merges the code of a cooperative and a non-cooperative kernel. This can reduce the occupancy for the merged kernel, e.g. due to higher register pressure, This is an artifact of our prototype implementation, and would not be a problem if our approach was implemented inside the GPU driver. We thus launch both the original and cooperative versions of a kernel with the reduced occupancy bound in order to meaningfully compare execution times.

Results. Tab. 2 shows the geometric mean and maximum slowdown across all applications and inputs, with averages and maxima

Table 2: Cooperative kernel slowdown w/o multitasking

overall		barrier		wk.steal.	
mean	max	mean	max	mean	max
1.07	1.23‡	1.06	1.20◦	1.12	1.23‡

‡octree, ◦color G3_circuit

computed over 10 runs per benchmark. For the maximum slowdowns, we indicate which application and input was responsible. The slowdown is below 1.25 even in the worst case, and closer to 1 on average. We consider these results encouraging, especially since the performance of our prototype could clearly be improved upon in a native implementation.

5.4 Multitasking via Cooperative Scheduling

We now assess the responsiveness of multitasking between a long-running cooperative kernel and a series of short, non-cooperative kernel launches, and the performance impact of multitasking on the cooperative kernel.

Experimental Setup. For a given cooperative kernel and its input, we launch the kernel and then repeatedly schedule a non-cooperative kernel that aims to simulate the intensity of one of the three classes of graphics rendering workload discussed in Sec. 5.2. In practice, we use matrix multiplication as the non-cooperative workload, with matrix dimensions tailored to reach the appropriate execution duration. We conduct separate runs where we vary the number of workgroups requested by the non-cooperative kernel, considering the cases where one, a quarter, a half, and all-but-one, of the total number of workgroups are requested. For the graph algorithms we try both regular and query barrier implementations.

Our experiments span 11 pairs of cooperative kernels and inputs, 3 classes of non-cooperative kernel workloads, 4 quantities of workgroups claimed for the non-cooperative kernel and 2 variations of resizing barriers for graph algorithms, leading to 240 configurations. We run each configuration 10 times, in order to report averaged performance numbers. For each run, we record the execution time of the cooperative kernel. For each scheduling of the non-cooperative kernel during the run, we also record the *gather time* needed by the scheduler to collect workgroups to launch the non-cooperative kernel, and the non-cooperative kernel execution time.

Responsiveness. Figure 5 reports, on three configurations, the average gather and execution times for the non-cooperative kernel with respect to the quantity of workgroups allocated to it. A logarithmic scale is used for time since gather times tend to be much smaller than execution times. The horizontal grey lines indicates the desired period for non-cooperative kernels. These graphs show a representative sample of our results; the full set of graphs for all configurations is provided in the extended version of this paper [27].

The left-most graph illustrates a work stealing example. When the non-cooperative kernel is given only one workgroup, its execution is so long that it cannot complete within the period required for a screen refresh. The gather time is very good though, since the scheduler needs to collect only one workgroup. The more workgroups are allocated to the non-cooperative kernels, the faster it can compute: here the non-cooperative kernel becomes fast enough
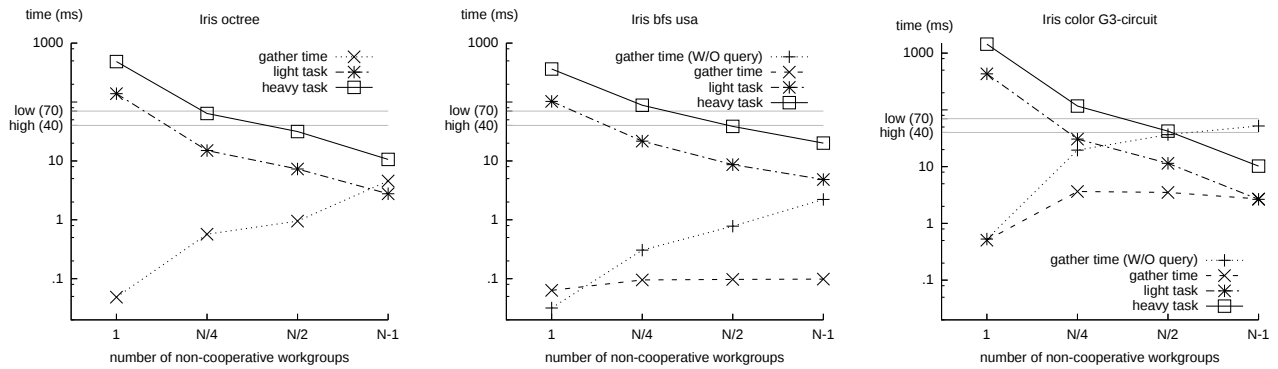
Figure 5: Example gather time and non-cooperative timing results
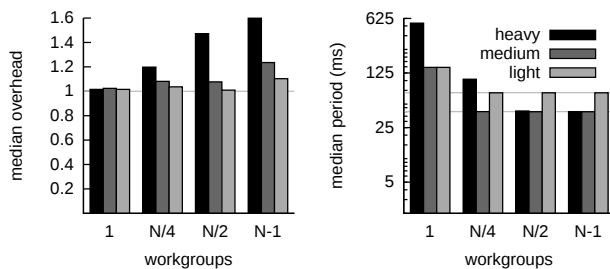

Figure 6: Performance impact of multitasking cooperative and non-cooperative workloads, and the period with which non-cooperative kernels execute

with a quarter (resp. half) of available workgroups for light (resp. heavy) graphics workload. Inversely, the gather time increases since the scheduler must collect more and more workgroups.

The middle and right graphs show results for graph algorithms. These algorithms use barriers, and we experimented with the regular and query barrier implementations described in Sec. 4.2. The execution times for the non-cooperative task are averaged across all runs, including with both types of barrier. We show separately the average gather time associated with each type of barrier. The graphs show a similar trend to the left-most graph: as the number of non-cooperative workgroups grows, the execution time decreases and the gather time increases. The gather time is higher on the rightmost figure as the G3 circuit input graph is rather wide than deep, so the graph algorithm reaches resizing barriers less often than for the USA road input of the middle figure for instance. The scheduler thus has fewer opportunities to collect workgroups and gather time increases. Nonetheless, scheduling responsiveness can benefit from the query barrier: when used, this barrier lets the scheduler collect all needed workgroups as soon as they hit a resizing barrier. As we can see, the gather time of the query barrier is almost stable with respect to the number of workgroups that needs to be collected.

Performance. Figure 6 reports the overhead brought by the scheduling of non-cooperative kernels over the cooperative kernel execution time. This is the slowdown associated with running the

cooperative kernel in the presence of multitasking, vs. running the cooperative kernel in isolation (median over all applications and inputs). We also show the period at which non-cooperative kernels can be scheduled (median over all applications and inputs). Our data included some outliers that occur with benchmarks in which the resizing barrier are not called very frequently and the graphics task requires half or more workgroups. For example, a medium graphics workload for bfs on the rmat input has over an 8× overhead when asking for all but one of the workgroups. As Figure 6 shows, most of our benchmarks are much better behaved than this. In future work is required to examine the problematic benchmarks in more detail, possibly inserting more resizing calls.

We show results for the three workloads listed in Sec. 5.2. The horizontal lines in the period graph correspond to the goals of the workloads: the higher (resp. lower) line corresponds to a period of $70\,ms$ (resp. $40\,ms$) for the light (resp. medium and heavy) workload.

Co-scheduling non-cooperative kernels that request a single workgroup leads to almost no overhead, but the period is far too high to meet the needs of any of our three workloads; e.g. a heavy workload averages a period of $939\,ms$. As more workgroups are dedicated to non-cooperative kernels, they execute quickly enough to be scheduled at the expected period. For the light and medium workloads, a quarter of the workgroups executing the non-cooperative kernel are able to meet their goal period (70 and $40\,ms$ resp.). However, this is not sufficient to meet the goal for the heavy workload (giving a median period of $104\,ms$). If half of the workgroups are allocated to the non-cooperative kernel, the heavy workload achieves its goal period (median of $40\,ms$). Yet, as expected, allocating more non-cooperative workgroups increases the overhead of the cooperative kernel. Still, heavy workloads meet their period by allocating half of the workgroups, incurring a slow down of less than 1.5× (median). Light and medium workloads meet their period with only a small overhead; 1.04× and 1.08× median slowdown respectively.

5.5 Comparison with Kernel-Level Preemption

Nvidia's recent Pascal architecture provides hardware support for instruction-level preemption [17, 24], however, preemption of entire kernels, but not of individual workgroups is supported. Intel GPUs do not provide this feature, and our OpenCL prototype of cooperative kernels cannot run on Nvidia GPUs, making a direct

Table 3: Overhead of kernel level preemption vs cooperative kernels for three graphics workloads

g. workload	kernel-level	cooperative	resources
light	1.04	1.04	$N/4$
medium	1.08	1.08	$N/4$
heavy	1.33	1.47	$N/2$

comparison impossible. We present here a theoretical analysis of the overheads associated with sharing the GPU between graphics and compute tasks via kernel-level preemption.

Suppose a graphics workload is required to be scheduled with period P and duration D, and that a compute kernel requires time C to execute without interruption. If we assume the cost of preemption is negligible (e.g. Nvidia have reported preemption times of 0.1 *ms* for Pascal [24], because of special hardware support), then the overhead associated with switching between compute and graphics every P time steps is $P/(P - D)$.

We compare this task-level preemption overhead model with our experimental results per graphics workload in Tab. 3. We report the overhead of the configuration that allowed us to meet the deadline of the graphics task. Based on the above assumptions, our approach provides similar overhead for low and medium graphics workloads, however, has a higher overhead for the high workload.

Our low performance for heavy workloads is because the graphics task requires half of the workgroups, crippling the cooperative kernel enough that request_fork calls are not issued as frequently. Future work may examine how to insert more resizing calls in these applications to address this. These results suggest that a hybrid preemption scheme may work well. That is, the cooperative approach works well for light and medium tasks; on the other hand, heavy graphics tasks benefit from the coarser grained, kernel-level preemption strategy. However, the preemption strategy requires specialised hardware support in order to be efficient.

6 RELATED WORK

Irregular Algorithms and Persistent kernels. There has been a lot of work on accelerating blocking irregular algorithms using GPUs, and on the *persistent threads* programming style for long-running kernels [1–4, 6, 7, 11, 13, 14, 16, 19, 21, 25, 26, 30, 32]. These approaches rely on the occupancy-bound execution model, flooding available compute units with work, so that the GPU is unavailable for other tasks, and assuming fair scheduling between occupant workgroups, which is unlikely to be guaranteed on future GPU platforms. As our experiments demonstrate, our cooperative kernels model allows blocking algorithms to be upgraded to run in a manner that facilitates responsive multitasking.

GPU Multitasking and Scheduling. Hardware support for preemption has been proposed for Nvidia GPUs, as well as *SM-draining* whereby workgroups occupying a symmetric multiprocessor (SM; a compute unit using our terminology) are allowed to complete until the SM becomes free for other tasks [28]. SM draining is limited the presence of blocking constructs, since it may not be possible to drain a blocked workgroup. A follow-up work adds the notion of SM *flushing*, where a workgroup can be re-scheduled from scratch

if it has not yet committed side-effects [20]. Both approaches have been evaluated using simulators, over sets of regular GPU kernels. Very recent Nvidia GPUs (i.e. the Pascal architecture) support preemption, though, as discussed in Sec. 1 and Sec. 5.5, it is not clear whether they guarantee fairness or allow tasks to share GPU resources at the workgroup level [17].

CUDA and OpenCL provide the facility for a kernel to spawn further kernels [18]. This *dynamic parallelism* can be used to implement a GPU-based scheduler, by having an initial scheduler kernel repeatedly spawn further kernels as required, according to some scheduling policy [15]. However, kernels that uses dynamic parallelism are still prone to unfair scheduling of workgroups, and thus does not help in deploying blocking algorithms on GPUs.

Cooperative Multitasking. Cooperative multitasking was offered in older operating systems (e.g. pre 1995 Windows) and is still used by some operating systems, such as RISC OS [23]. Additionally, cooperative multitasking can be efficiently implemented in today's high-level languages for domains in which preemptive multitasking is either too costly or not supported on legacy systems [29].

7 CONCLUSIONS AND FUTURE WORK

We have proposed *cooperative kernels*, a small set of GPU programming extensions that allow long-running, blocking kernels to be fairly scheduled and to share GPU resources with other workloads. Experimental results using our megakernel-based prototype show that the model is a good fit for current GPU-accelerated irregular algorithms. The performance that could be gained through a native implementation with driver support would be even better. Avenues for future work include seeking additional classes of irregular algorithms to which the model might (be extended to) apply (to), investigating implementing native support in open source drivers, and integrating cooperative kernels into template- and compiler-based programming models for graph algorithms on GPUs [19, 33].

ACKNOWLEDGMENTS

REFERENCES

[1] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IISWC*, pages 141–151. IEEE, 2012.

[2] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *EGGH*, pages 57–64, 2008.

[3] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IISWC*, pages 185–195, 2013.

[4] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *IPDPS*, pages 349–359, 2014.

[5] Google. Chrome Experiments. https://www.chromeexperiments.com.

[6] K. Gupta, J. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *InPar*, pages 1–14, 2012.

[7] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208, 2007.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[9] L. W. Howes. Personal communication. Editor of the OpenCL 2.0 specification. 10 September 2016.

[10] HSA Foundation. HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG), February 2016. http://www.hsafoundation.com/standards/.

[11] R. Kaleem, A. Venkat, S. Pai, M. W. Hall, and K. Pingali. Synchronization trade-offs in GPU implementations of graph algorithms. In *IPDPS*, pages 514–523, 2016.

[12] Khronos Group. The OpenCL specification version: 2.0 (rev. 29), July 2015. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[13] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, pages 107–116, 2012.

[14] D. Merrill, M. Garland, and A. S. Grimshaw. High-performance and scalable GPU graph traversal. *TOPC*, 1(2):14, 2015.

[15] P. Muyan-Özçelik and J. D. Owens. Multitasking real-time embedded GPU computing tasks. In *PMAM*, pages 78–87, 2016.

[16] S. Nobari, T. Cao, P. Karras, and S. Bressan. Scalable parallel minimum spanning forest computation. In *PPoPPP*, pages 205–214, 2012.

[17] NVIDIA. NVIDIA Tesla P100, 2016. Whitepaper WP-08019-001_v01.1.

[18] Nvidia. CUDA C programming guide, version 7.5, July 2016.

[19] S. Pai and K. Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *OOPSLA*, pages 1–19, 2016.

[20] J. J. K. Park, Y. Park, and S. A. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *ASPLOS*, pages 593–606, 2015.

[21] T. Prabhu, S. Ramalingam, M. Might, and M. W. Hall. EigenCFA: accelerating flow analysis with GPUs. In *POPL*, pages 511–522, 2011.

[22] A. Richards. Personal communication. CEO of Codeplay Software Ltd. 2 September 2016.

[23] RISC OS. Preemptive multitasking. http://www.riscos.info/index.php/Preemptive_multitasking.

[24] R. Smith and Anandtech. Preemption improved: Fine-grained preemption for time-critical tasks, 2016. http://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10.

[25] S. Solomon, P. Thulasiraman, and R. K. Thulasiram. Exploiting parallelism in iterative irregular maxflow computations on GPU accelerators. In *HPCC*, pages 297–304, 2010.

[26] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamaric. Portable inter-workgroup barrier synchronisation for GPUs. In *OOPSLA*, pages 39–58, 2016.

[27] T. Sorensen, H. Evrard, and A. F. Donaldson. Cooperative kernels: Gpu multi-tasking for blocking algorithms (extended version). *CoRR*, 2017.

[28] I. Tanasic, I. Gelado, J. Cabezas, A. Ramírez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *ISCA*, pages 193–204, 2014.

[29] M. Tarpenning. Cooperative multitasking in c++. *Dr. Dobb's J.*, 16(4), Apr. 1991.

[30] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *HPG*, pages 29–37, 2010.

[31] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *IEEE Communications Surveys and Tutorials*, 15(1):179–198, 2013.

[32] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *HPG*, pages 167–171, 2009.

[33] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP*, pages 11:1–11:12, 2016.

[34] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12, 2010.