Cooperative Kernels: Safe Multitasking for Blocking Algorithms on GPUs

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 the semantics of the cooperative kernels programming model and our prototype implementation. We evaluate the approach by porting a set of irregular work stealing and graph algorithms to our programming model and examining their performance under multitasking across three GPUs. Our implementation exploits no vendor-specific hardware, driver or compiler support, so our encouraging results provide a lower-bound on the efficiency with which cooperative kernels can be implemented in practice.

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 be determined by the computation itself. There is growing interest in accelerating such algorithms on GPUs, particularly the ones that traverse and manipulate linked data structures [6–8, 10, 12, 13, 18, 21, 22, 24, 27, 29, 32, 34, 39, 41].

Irregular algorithms usually require *blocking synchronization* between workgroups, to balance load and communicate intermediate results. For example, many graph algorithms employ a level-by-level strategy, requiring a global barrier between each level, and work stealing algorithms require each workgroup to maintain a queue, which is typically protected by a mutex to enable stealing by other workgroups.

To operate correctly, a blocking concurrent algorithm requires *fair* scheduling of workgroups. Without fairness, a workgroup may fail to make progress, leading to starvation. 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 GPU schedulers are *not* fair. The current GPU programming models—OpenCL [20], CUDA [26] and HSA [16]—specify almost no guarantees regarding scheduling of workgroups, and current GPU scheduler 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 [12, 34] 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 the occupancy-bound execution model is widely implemented by today's GPU devices and drivers [6, 12, 27, 34]. It is evidently a simple model for device vendors to implement efficiently.

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 [12]. This works because *today*'s GPUs seem to provide the occupancy-bound execution model.

Resistance to guaranteeing occupancy-bound execution Despite its practical prevalence, none of the current GPU programming models actually mandate occupancy-bound execution. Further, there are clear reasons why this model is undesirable. First, the execution model does not enable multitasking, because a workgroup effectively *owns* a com-

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

pute unit until the workgroup has completed execution. The GPU cannot be used meanwhile for other tasks, such as graphics rendering. Second, *energy throttling* is an important concern for battery-powered devices [40]. In the future, it will be desirable for a mobile GPU driver to power down some compute units if the battery level is low. Implementing energy throttling would require being able to suspend the execution of a workgroup, powering down the compute unit to which the workgroup was bound, something that the occupancy-bound execution model does not allow.

Our assessment, informed by discussions with a number of industrial practitioners who have been involved in the OpenCL and/or HSA standardisation efforts (including [15, 30] and various engineers from GPU vendors), is that GPU vendors (1) will not commit to the occupancy-bound execution model they currently implement, for the above reasons, yet (2) will not guarantee fair scheduling due to the runtime cost of preempting workgroups. Vendors instead wish to retain the essence of the simple occupancy-bound model, supporting preemption only in key special cases.

Our proposal: cooperative kernels To summarise: blocking algorithms demand fair scheduling of workgroups, but for good reasons GPU vendors will not commit even 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 offer_fork, placed by the programmer.

At a point 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. At a point where the cooperative kernel could benefit from having additional workgroups join the computation, a workgroup can execute offer_fork to indicates 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 offer_fork program point.

The use of offer_fork and offer_kill creates a contract between the scheduler and the (programmer of 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 offer_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, or allowing compute units to be powered down quickly when required. The scheduler should respond to offer_kill and offer_fork calls in a manner that avoids underutilisation of hardware resources by the cooperative kernel, e.g. the scheduler should not kill workgroups unless they are required for another purpose, and should facilitate forking of additional workgroups when possible.

The cooperative kernels programming 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: these can be programmed exactly as they are now.
3. The model is backwards-compatible: if offer_kill and offer_fork are ignored, 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 that current GPUs provide: our prototype implementation requires no additional hardware or driver support. Thus the model should be easy for vendors to implement directly.
5. Our experiments with a range of applications using three GPUs show that the model can enable efficient multitasking of cooperative and non-cooperative tasks.

Placing the new primitives requires manual effort, but our experience porting a representative set of GPU-accelerated irregular algorithms to use cooperative kernels (Sec. 5.1) suggests that this is straightforward in practice.

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 OpenCL 2.0 (Sec. 4); and *experiments* assessing the overhead and responsiveness of the cooperative kernels approach over a set of irregular algorithms across three GPUs (Sec. 5).

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 Motivating Examples

We outline the industry-standard 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 its unique id, to access distinct data or follow different control flow paths to other threads.

The threads of a kernel are divided into *workgroups*, each consisting of a fixed number of threads. Functions get_local_id and get_group_id return a thread's local id within its workgroup and the id of the workgroup.[2] The number of threads per workgroup and number of workgroups are obtained via get_local_size and get_num_groups. A thread's local and global id are related by get_global_id = get_group_id × get_local_size + get_local_id. The total number of threads executing the kernel is given by get_global_size = get_local_size × get_num_groups.

Execution of the threads in a workgroup can be synchronised via a workgroup barrier, which also ensures memory consistency. This barrier is a *workgroup-level* function: it can only appear in conditional code if all threads in a workgroup that reach the barrier agree on the guards of all enclosing conditional statements. 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 threads and the host application concurrently. *Global* memory is shared among all threads executing a kernel. 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.

Communication within a workgroup can be achieved using a workgroup barrier. Finer-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 specifically makes no guarantees about fair scheduling between workgroups executing the same kernel, stating [20, p. 31]: *"A conforming implementation may choose to serialize the workgroups . . . There is no safe and portable way to synchronize across the independent execution of workgroups since once in the work-pool, they can execute in any order."* CUDA similarly provides no guarantees [26]. HSA provides limited, one-way guarantees, stating [16, 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 example Work stealing enables dynamic balancing of tasks across several 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. In the context of GPUs, each workgroup has an associated queue from which it obtains tasks to process, and to which it stores new tasks. If a workgroup's queue is empty, the workgroup tries to *steal* a task from another queue. Previous work covers work stealing feasibility and efficiency on GPUs [7, 39].

[2] For simplicity we assume 1D kernels, which captures all the irregular algorithms we know of, and write e.g. get_global_id for get_global_id(0).

```
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    }
9  }
```

Figure 1: 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 < nodes.size; i += stride) {
10       process_node(g, in_nodes[i], out_nodes, level);
11     }
12     swap(&in_nodes, &out_nodes);
13     global_barrier();
14     reset(out_nodes);
15     level++;
16     global_barrier();
17   }
18 }
```

Figure 2: An OpenCL graph traversal algorithm, using a global barrier for synchronisation

Figure 1 shows a simplified version of a work stealing kernel. Each thread receives a pointer to the task queues, in global memory, initialized by the host to contain the initial tasks. Each thread identifies its workgroup via get_group_id (line 2), which is used as a queue id to access the relevant task queue. The pop_or_steal function (line 4) is called by all threads of a workgroup. Inside pop_or_steal only the thread with a local id equal to 0, the *master thread*, tries to pop a task from the workgroup's queue, or to steal a task from a different queue. The master thread uses local memory inside pop_or_steal to ensure that the returned value is valid for all threads of its workgroup. If a task was obtained, then all the workgroup threads participate in the task processing (line 6). This may lead to the creation of further tasks, which the master thread pushes to the workgroup's queue.

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. The kernel presents two opportunities for spin-waiting: spinning to obtain a mutex, and spinning in the main kernel loop to obtain a task. The algorithm thus depends on a fair scheduler, which is not guaranteed by current GPU programming models.

Graph traversal example Figure 2 shows the skeleton of a frontier-based graph traversal algorithm; such algorithms have been shown to execute efficiently on GPUs [6, 27]. 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 indicates the frontier level currently being processed, and in_nodes and out_nodes point to distinct node 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. Each thread processes nodes with the following sequence of indices: tid+stride*0, tid+stride*1, tid+stride*2, etc., where tid is a thread's global id. A thread processes a node via process_node, which may (a) push nodes to process in the next frontier to out_nodes and (b) use the current frontier, level, in the computation. After processing the frontier, the threads swap their node array pointers (line 12).

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 13). After all threads reach this point, the output node array is reset (line 14) and the level is incremented. The threads use another global barrier to wait until the output node is reset (line 16), 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 [34, 44], based on CPU barrier designs [14, ch. 17]. These barriers employ spinning to ensure threads wait at the barrier until all threads have arrived, which requires fair scheduling between workgroups for the barrier to operate correctly.

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 language standards or vendor implementations, and is unlikely to be respected in the future. In Sec. 3.2 we show how the work stealing and graph traversal examples of Figs. 1 and 2 can be updated to use our cooperative kernels programming model to resolve the scheduling issue.

3. Cooperative Kernels

We present our cooperative kernels programming model as an extension to OpenCL. However, the cooperative kernels concept is more general, and could be applied to extend other GPU programming models, such as CUDA and HSA. We describe the semantics of the model (Sec. 3.1), use our motivating examples to discuss programmability (Sec. 3.2), outline important nonfunctional properties that the model requires to work successfully (Sec. 3.3) and explain that the model is backwards-compatible (Sec. 3.4). In Appendices A and B (supplementary material) we present a formal semantics for cooperative kernels atop an abstract GPU programming model, and discuss several cases where we might have taken different and also reasonable semantic decisions.

3.1 Semantics of Cooperative Kernels

As with a regular OpenCL kernel (see Sec. 2.1), a cooperative kernel is launched by the host application. The host application passes parameters to the kernel and specifies a desired number of workgroups, each consisting of a specified number of threads. 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 offer_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 spawns new workgroups by calling offer_fork, the workgroup should transmit its current value for v to the threads of the new workgroups, to serve as an initial value for v. We detail the semantics for this when we describe offer_fork below.

Active workroups 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.

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 were requested at kernel launch time (see Sec. 2.1).

Fair scheduling *is* guaranteed between active workgroups. That is, if some thread in an active workgroup is enabled then eventually some thread in the active workgroup 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 (see Sec. 2.1).

Suppose a workgroup with id m executes offer_kill. If the workgroup has the highest largest id of all active workgroups then it can be killed by the scheduler, except that workgroup 0 can never be killed (to ensure that the kernel computation does not terminate early). 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.

Semantics for offer_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 offer_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 $m \leq M$ executes offer_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 offer_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 offer_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.

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

Global barriers Because workgroups of a cooperative kernel are fairly scheduled, so a global barrier primitive can be provided, and 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. [44], 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 [34]. However, implementing such a barrier is involved, hence why we include this function as a primitive.

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 the following sequence of calls:

```
global_barrier();
if(get_group_id() == 0) offer_fork();
global_barrier();
offer_kill();
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 from a programmer's perspective, M changes atomically. 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 significantly 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, offer_fork and resizing_global_barrier. 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 offer_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.

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. 1, we add offer_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 offer_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 3 shows a cooperative version of the graph traversal kernel of Fig. 2 from Sec. 2.2. On lines 13 and 16, 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 offer_fork directly: the value of stride obtained from get_global_size at line 8 is stable until the next resizing barrier at line 13, and in particular can be safely used inside the for loop.

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

```
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 < nodes.size; i += stride) {
10       process_node(g, in_nodes[i], out_nodes, level);
11     }
12     swap(&in_nodes, &out_nodes);
13     resizing_global_barrier();
14     reset(out_nodes);
15     level++;
16     resizing_global_barrier();
17   }
18 }
```

Figure 3: Cooperative version of the graph traversal kernel of Fig. 2, using a resizing barrier and transmit annotations

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 offer_fork, and few or no transmit annotations are required. Figure 3 is simpler than, but 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 envelope of behaviour 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 provide the scheduler with an opportunity to 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 can be 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 offer_fork calls if compute units are available. Our implementation (Sec. 4) is as generous as possible to the cooperative kernel in this regard.

3.4 Backwards Compatibility

Our new model has no effect on the manner in which traditional GPU workloads are programmed, such as graphics shaders in OpenGL, Vulkan and DirectX, and regular data-parallel computational kernels. Furthermore, if our cooperative language extensions are ignored—offer_kill and offer_fork are treated as no-ops, transmit annotations are ignored, and global_barrier and resizing_global_barrier calls are redirected to an existing global barrier implementation (so that no resizing occurs in the latter case)—then a cooperative kernel will behave just like its non-cooperative counterpart would behave. A developer for whom today's occupancy-bound execution model suffices can upgrade their kernel to a cooperative form and retain backwards-compatibility with today's GPU platforms by pre-processing away our language extensions.

4. Prototype Implementation

Our vision is that support for cooperative kernels will be integrated in the runtimes of future GPU implementations of OpenCL and related programming models, with driver and compiler 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 scheduler, 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 assume all variables declared at the root lexical scope of a cooperative kernel are implicitly transmit-annotated; this allows our implementation to piggy-back on top of Clang (see below) without requiring front-end changes. Finally, we require that offer_kill, offer_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 do not deem the engineering effort associated with lifting these restrictions to be worthwhile.

Merging kernels We have built a tool, KERNELMERGE, that takes a cooperative kernel and a non-cooperative kernel and merges them to produce a megakernel with an embedded GPU scheduler. We implemented KERNELMERGE using the Clang LibTooling framework [9], leveraging recent Clang support for OpenCL 2.0 [37].

4.2 Scheduler design

To enable multitasking through cooperative kernels, the runtime (in our case, the megakernel) must track the state of all 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 offer_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 offer_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 offer_fork. Thus, the scheduler context needs a communication channel for each cooperative 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, the inputs to the kernel, the number of workgroups that should execute the kernel, and a flag indicating whether the kernel is cooperative. In our implementation, 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 make decisions about 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 offer_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 offer_fork command. When a workgroup w of K_c subsequently executes offer_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 offer_fork call. This involves selecting workgroups whose status is *available*, as well as copying the values of transmit-annotated variables to the new workgroups.

Accounting for concurrency Concurrent calls to offer_fork must be protected by a mutex to avoid racing on the available workgroups. The same conflict over available workgroups can occur at a new kernel launch between the scheduler and a workgroup calling offer_fork. Thus, these activities are protected by a mutex, called the *assignment mutex* provided in the scheduler context.

The assignment mutex additionally protects concurrent calls to offer_fork and offer_kill. On calling offer_kill, a workgroup is only permitted to exit if the scheduler workgroup currently holds the assignment mutex; that is, the scheduler is trying to obtain work groups to send to a task. Conversely, on calling offer_fork a workgroup can only spawn new workgroups if the scheduler workgroup does not hold the assignment mutex. Thus the assignment mutex, only ever acquired by the scheduling workgroup, *indirectly* provides mutual exclusion between a pairs of workgroups executing offer_kill and offer_fork concurrently.

Concurrent calls to offer_kill are acceptable because only the workgroup with the highest id may exit, automatically ensuring mutual exclusion for exit-related state changes.

An efficient resizing barrier In Sec. 3.1, we defined the semantics of a resizing barrier in terms of a series of calls to other primitives. It is possible, however to implement the resizing with only one call to a global barrier with offer_fork and offer_kill baked into the implementation.

We consider barriers that use the master/slave model (e.g. [44]): one workgroup (the master) collects signals from the other workgroups (the 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 the slaves, it replies with a signal saying that the slaves may leave.

A first-attempt at incorporating offer_fork and offer_kill into such a barrier is straight forward. 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 spinning, waiting to be released. The master executes offer_fork, and the statement immediately following this offer_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 offer_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.

We can greatly reduce the gather time by providing a new query function for a workgroup executing a cooperative kernel, which returns the number of workgroups that the scheduler needs to obtained from the cooperative kernel. The programmer cannot call query directly; it is only invoked from within the resizing barrier implementation. A resizing barrier can now be implemented as follows: (1) the master waits for all slaves to arrive; (2) the master calls offer_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.

App.	barriers	kill	fork	transmit	LoC	inputs
color	2 / 2	0	0	4	55	2
mis	3 / 3	0	0	0	71	2
bc	3 / 6	0	0	3	150	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

Table 1: Blocking GPU applications investigated

Chip	HD520	HD5500	Iris
# CU	24	24	47
Driver	20.19.15.4501	10.18.15.4281	20.19.15.4463
Host CPU	i3-6100U	i7-5600U	i3-5157U

Table 2: The Intel GPU platforms used in our evaluation

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 multitasked with non-cooperative workloads (Sec. 5.4).

5.1 Applications and GPUs

Table 1 gives an overview of the 8 irregular algorithms that we ported to cooperative kernels. Among them, 6 are graph algorithms, based on the Pannotia [8] and Lonestar [6] GPU application suites, using global barriers. We indicate how many of the original number of barriers are changed to resizing barriers, and how many variables need to be transmitted. In all cases all barriers were converted, except for bc which contains barriers deeper in the call stack, a case not supported by our prototype although these barriers could in principle be converted. The remaining 2 algorithms are work stealing applications: each required the addition of offer_fork and offer_kill at the start of the main loop, and no variables needed to be transmitted. For all these examples the porting to cooperative kernels was similarly straightforward to the porting illustrated in Sec. 3.2 for our running examples. Most graph come with 2 different data sets as input, leading to 13 application/input pairs in total.

From the Lonestar suite we omit two applications that use global barriers: dmr has been shown to be unstable across GPU vendors [33], and mst is trivially short-running so that it is not interesting for our multitasking approach.

Our prototype implementation (Sec. 4) requires two optional features of OpenCL 2.0: SVM fine-grained buffers and SVM atomics. This rules out evaluation on Nvidia and ARM Mali GPUs at present, since these vendors do not yet support OpenCL 2.0, equivalent functionality is not provided by Nvidia's CUDA. Intel drivers meet our requirements on Windows, but not on Linux (we confirmed this by trying the upstream driver that requires a patched Linux kernel). AMD support SVM atomics only their Kaveri family of chips and only on Linux, requiring to compile the latest driver.

Among the GPUs available to us, four met our requirements: Intel HD 520, HD 5500 and Iris 6100, and AMD Radeon R7. However, AMD's Linux drivers for OpenCL suffer from a known defect whereby long-running kernels lead to defunct processes that the OS cannot kill [5]. We observed this on our AMD platform with the latest driver, and simultaneously obtained alarming results on this platform when using SVM atomics: modifying a kernel to include a simple host/device handshake through SVM slowed down kernel execution by an order of magnitude in some cases. This behaviour, combined with the defunct processes issue, lead us to suspect that the support we require for cooperative kernels is not yet mature enough within AMD's drivers, which we plan to discuss with engineers at AMD.

We thus present experimental results for the Intel GPUs, whose characteristics are summarised in Tab. 2.

5.2 Sizing Non-Cooperative Kernels

Enabling rendering of smooth graphics in parallel with execution of 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 use case directly.

We devised the following procedure to determine non-cooperative 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 and the period at which the OS requires the GPU for display rendering, 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 [1] running in a recent version of Firefox.

Chip	HD520	HD5500	Iris
overall geo. mean	1.20	1.14	1.08
overall max	1.75*	1.45†	1.37‡
barrier geo. mean	1.18	1.16	1.07
barrier max	1.75*	1.45†	1.37‡
wk.steal. geo. mean	1.42	1.10	1.12
wk.steal. max	1.42◇	1.18◇	1.23◇

*color ecology, †bc 1k_128k, ‡sssp usa_ny, ◇octree

Table 3: Execution time slowdown associated with cooperative scheduling mode without multitasking

Our results are the following: light workloads needs to run 3ms every 70ms, medium ones 3ms every 40ms, and heavy ones 10ms every 40ms. For the medium and heavy workloads, the 40ms period corresponds to graphics rendering occurring 25 times per second, which coincides with the human persistence of vision. The 3ms execution duration of both light and medium configurations indicates that, as expected, GPU computation is cheaper for basic display rendering compared with rendering that uses complex shaders. We use these results to guide our evaluation of the responsiveness of cooperative scheduling in Sec. 5.4.

5.3 The Overhead of Moving to 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 for a cooperative kernel with that of the non-cooperative task to be co-scheduled. This blow-up in code size can have a negative impact on compilation, e.g. due to higher register pressure, reducing the occupancy for the merged kernel—the maximum number of workgroups that the GPU driver will schedule. This is an artifact of our proof-of-concept implementation strategy, and would not be a problem if our cooperative scheduling 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 For each GPU, Tab. 3 shows the geometric mean and maximum slowdown across all applications and inputs, with averages and maxima computed over 10 runs per benchmark. For the maximum slowdowns, we indicate which application and input was responsible. The slowdown is below 2 even in the worst case, and closer to 1 on average. We consider these results as 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 data set, we launch the kernel asynchronously and then repeatedly schedule a non-cooperative kernel that aims to simulate the intensity of a graphics rendering workload, according to the three classes of workload discussed in Sec. 5.2. In practice, we use a matrix multiplication kernel as the non-cooperative workload, with matrix dimensions tailored for each GPU 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 workgroup slots is requested. Moreover, for the graph algorithms we try both the regular and query barrier implementations.

Our experiments span 13 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 288 configurations. We run each configuration 10 times on each GPU, 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 enough workgroups to launch the non-cooperative kernel, and the non-cooperative kernel execution time. We sanity-check whether all kernels perform their computations correctly by comparing their computed result with expected reference results.

To work around compiler crash and compiler hang bugs in Intel's current OpenCL drivers, we had to apply some semantics-preserving changes to the code generated by KERNELMERGE for a few applications configurations. Driver bugs also meant that we could not obtain results for some configurations where the non-cooperative kernel asks for all-but-one workgroups (namely bc, bfs and l-sssp on all chips, and color on Iris), and could not get the game application to run at all on HD520; these configurations led to severe failures such as machine freezes and blue screens.

Responsiveness Figure 4 reports, on three configurations, the average gather and execution times for the non-cooperative kernel w.r.t. 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 Appendix C (supplementary material).

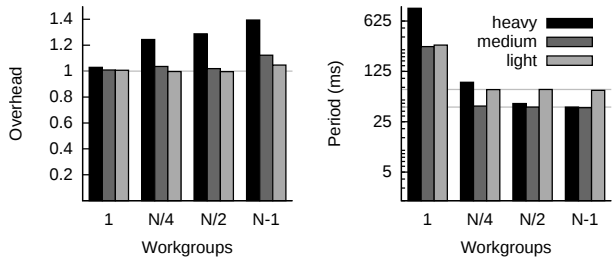

Figure 5: Performance impact of multitasking cooperative and non-cooperative workloads, and the period with which non-cooperative tasks execute

Let us first focus on the left graph, which illustrates a work stealing example executing on the Iris GPU. 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 only needs to collect one workgroup. As more workgroups are allocated to the non-cooperative kernel, its execution time benefits from more computational resources: here the non-cooperative kernel becomes fast enough 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 on the other two GPUs. 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.[3] The gather time is higher on the right figure: the rmat22 input graph is rather wide than deep, so the graph algorithm reaches resizing barriers less often than for the ecology 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 on both the middle and right figure, the gather time of the query barrier is almost stable w.r.t. the number of workgroups that needs to be collected.

Performance Figure 5 reports, for the Iris GPU, 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 (geometric mean over all applications and inputs). We also show the period at which non-cooperative kernels can be scheduled (arithmetic mean over all applications and inputs). The results for the other GPUs are similar. We show results for the three workloads listed in Sec. 5.2. The two gray horizontal lines in the period graph correspond to the period goals of the workloads: the higher (resp. lower) line corresponds to a period of 70ms (resp. 40ms) 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, a quarter of the workgroups are not sufficient to meet the goal for the heavy workload (giving a mean period of 88 ms). If half of the workgroups are allocated to the non-cooperative kernel, the heavy workload averages $\sim$10% over its goal period (mean of 44 ms). If all-but-one are allocated, the heavy workload reaches its goal period. Yet, as expected, allocating more non-cooperative workgroups increases the execution time of the cooperative kernel.

Still, the heavy workloads meet their period (by allocating all-but-one workgroup for non-cooperative kernels), incurring a slow down of less than 1.4$\times$ on average. Light and medium workloads meet their period with only a negligible overhead of the cooperative kernel (less than a 1.03$\times$ slowdown on average).

Overall we are very encouraged by these experimental findings, especially since they provide a lower bound on potential performance of our cooperative kernels model. Implementing the model natively, with device-specific runtime support, could only improve performance and responsiveness compared with that of our megakernel-based prototype.

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 [6–8, 10, 12, 13, 18, 21, 22, 24, 27, 29, 32, 34, 39, 41]. 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 under a fair scheduler in a manner that, with appropriate care, facilitates responsive multitasking.

[3] The right figure has only 3 points per line: recall that for bfs there are no runs with "all-but-one" non-cooperative workgroups.

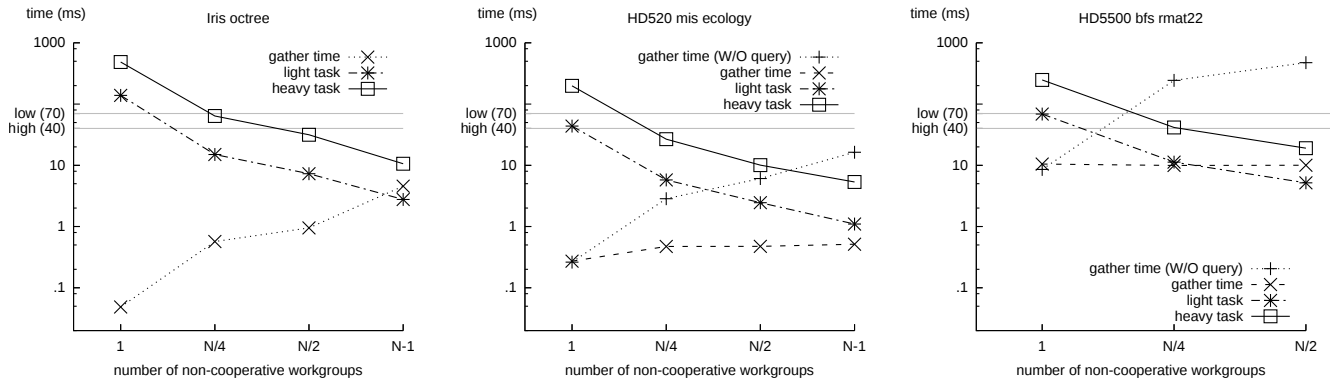

Figure 4: Example gather time and non-cooperative timing results

Preemptive multitasking on GPUs 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 [38]. 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 cancelled and re-scheduled from scratch if it has not yet committed side-effects [28]. Both approaches have been evaluated using simulators, over sets of regular GPU kernels. Very recent Nvidia GPUs targeting data centers now support preemption, though it is not clear whether they guarantee fair scheduling [25].

The advantages of preemption are that it does not require programmer effort, and guards against a rogue long-running kernel making the GPU unavailable for other tasks. The main drawback is that *every* kernel takes a potential performance hit. If implemented natively, our solution does not impact performance on regular kernels that are not scheduled with cooperative kernels. Furthermore, careful placement of our new primitives can exploit problem structure, enabling more efficient multitasking than application-oblivious preemption. There is scope for combining the approaches: if a cooperative kernel is insufficiently cooperative, i.e. it does not invoke offer_kill enough, then hardware support for preemption could be invoked to prevent the kernel from denying other tasks access to the GPU indefinitely.

Optimizing GPU scheduling A number of works have considered how best to schedule dynamic workloads on GPUs [11, 19, 31, 35, 36, 43]. Among these, the Whippletree approach employs a persistent megakernel to schedule multiple, dynamic tasks in a manner that aims to best utilise GPU resources [36], and the TimeGraph approach similarly aims to optimise scheduling of competing workloads, in the context of OpenGL [19]. None of these approaches tackle the problem of *fair* scheduling on GPUs, thus they do not aid in safe deployment of blocking irregular algorithms.

CUDA and OpenCL provide the facility for a kernel to spawn further kernels [20, 26]. 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 [23]. However, kernels that uses dynamic parallelism are still prone to unfair scheduling of workgroups, so this feature does not help solve the problem of safely deploying blocking algorithms on GPUs.

Cooperative multitasking Cooperative multitasking was offered in older operating systems (e.g. Windows before the 95 edition) and is still used by some operating systems, such as RISC OS [2]. Many programming languages offer cooperative multitasking via variations of co-routines, e.g. co-routine objects in Python and Lua, and Ruby's *fiber*.

In the context of GPUs, cooperative multitasking has been used to refer to the standard programming model, where the onus is on the GPU kernel to complete execution within a reasonable time budget [3, 17]. This is in contrast to our cooperative kernels, which specifically aim to support the needs of long-running GPU tasks.

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 over three Intel GPUs shows that the model is a good fit for the needs of current GPU-accelerated irregular algorithms. The performance that could be gained through a native implementation with dedicated 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 [27, 42].

References

[1] Chrome Experiments. https://www.chromeexperiments.com.

[2] RISC OS cooperative mutlitasking. http://www.riscos.info/index.php/Preemptive_multitasking.

[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *HPCA*, pages 1–12, 2012.

[4] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591. ACM, 2015.

[5] AMD. AMD APP SDK ver. 2.8 Release Notes. http://developer.amd.com/download/AMD_APP_SDK_Release_Notes_Developer.pdf.

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

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

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

[9] T. Clang Team. Clang 3.8 LibTooling, 2015. http://clang.llvm.org/docs/LibTooling.html.

[10] 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.

[11] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with gpus. *Real-Time Systems*, 48(1): 34–74, 2012.

[12] 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.

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

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

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

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

[17] F. Ino, A. Ogita, K. Oita, and K. Hagihara. Cooperative multitasking for gpu-accelerated grid systems. *Concurrency and Computation: Practice and Experience*, 24(1), 2012.

[18] 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.

[19] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

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

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

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

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

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

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

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

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

[28] 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.

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

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

[31] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP*, pages 233–248, 2011.

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

[33] T. Sorensen and A. F. Donaldson. The hitchhiker's guide to cross-platform OpenCL application development. In *IWOCL*, pages 2:1–2:12, 2016.

[34] 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.

[35] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg. Softshell: dynamic scheduling on gpus. *ACM Trans. Graph.*, 31(6):161, 2012.

[36] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg. Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.*, 33(6): 228:1–228:11, 2014.

[37] A. Stulova. A journey of OpenCL 2.0 development in Clang. In *European LLVM Developers' Meeting*, 2016. http://llvm.org/devmtg/2016-03/Presentations/AnastasiaStulova_OpenCL20_EuroLLVM2016.pdf.

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

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

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

[41] 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.

[42] 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.

[43] B. Wu, G. Chen, D. Li, X. Shen, and J. S. Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*, pages 119–130, 2015.

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