GPUpIO: The Case for I/O-Driven Preemption on GPUs

Lior Zeno   Avi Mendelson   Mark Silberstein

Technion – Israel Institute of Technology
{liorz@tx, avi.mendelson@tce, mark@ee}.technion.ac.il

ABSTRACT

As GPUs become general purpose, they are outgrowing the co-processor model and require convenient I/O abstractions such as files and network sockets. Recent studies have shown the benefits of native GPU I/O layers, in terms of both programmability and performance. However, due to lack of hardware support, the GPU threads performing I/O calls are forced to busy-wait for the completion of I/O operations, resulting in underutilized hardware, higher power consumption, and reduced system throughput.

We argue that I/O-driven *preemption* improves the performance of existing solutions, despite many challenging system characteristics such as a large kernel state. We analyze the benefits of adding preemption support using a simple system performance model, and, encouraged by the results, explore the design of a software-based preemption mechanism for GPUs. In our prototype, **GPUpIO**, we implement a source-to-source compiler for state checkpoint and restoration, and a runtime library for scheduling preempted thread-blocks, which together enable I/O-driven preemption for GPUs.

We evaluate our prototype across a variety of system parameters and workloads to determine when preemption is worthwhile. We show that in some workloads the I/O-driven preemption approach may indeed double the effective system throughput by completely hiding the I/O latency behind computations. However, we also observe that the software-only solution is currently limited, not only due to its overheads, but also because it does not have sufficient control of the hardware scheduler queue and therefore may lead to starvation of I/O kernels. We then discuss a new hardware feature that, if added, may render a general I/O-driven preemption mechanism on GPUs practical.

CCS Concepts

•**Software and its engineering** → **Scheduling;** Compilers; •**Computer systems organization** → *Single instruction, multiple data;*

Keywords

Accelerators, file systems, GPGPUs, operating systems design, source-to-source compiliation

1. INTRODUCTION

Over the last few years GPUs have made a gigantic leap forward toward efficient acceleration of general purpose computing applications. In line with the hardware enhancements, recent work shows that I/O abstractions such as file and network I/O available directly from GPU code provide significant programmability and performance benefits. Coupled with direct access to I/O devices, these abstractions make it possible to build efficient I/O intensive applications that access network and storage directly from GPU code.

For example, GPUnet [3] provides a CPU-like socket API that makes sending data over the network from the GPU as simple as sending data from the CPU. Such a native I/O exhibits three important programmability and performance advantages. First, it eliminates the need to exit the GPU kernel in order to send or receive new data, thereby saving the kernel invocation and synchronization overheads. Second, the underlying system seamlessly takes advantage of advanced hardware capabilities, such as peer-to-peer DMA, that enable direct access to the I/O hardware, bypassing the CPU for both data transfer and control. Last, pending I/O operations that block a few threads of the GPU kernel are overlapped with computations performed by other threads, enabling I/O pipelining.

Along with these benefits, the use of GPU native I/O operations has one important limitation, in particular in the case of slower I/O devices like disks. In the absence of hardware support for interrupt delivery to GPU application code, a GPU kernel must spin on an I/O control register (maintained by the I/O device or by the CPU) in order to block while waiting for the completion of an I/O operation. As a result, the spinning threads waste GPU compute resources, resulting in lower effective GPU utilization and lower power efficiency, in particular when there are other GPU kernels waiting for execution. GPU underutilization due to I/O is also a problem in GPUs that support page faults, as is the case for existing hybrid GPU-CPU systems and expected in future discrete GPUs.

In CPU systems the problem was solved long ago with the interrupt-driven I/O design: the blocked process is preempted – removed from the CPU run queue – and the OS reschedules it after receiving an interrupt notifying of the I/O request completion. Unfortunately, GPUs do not deliver interrupts to GPU code, and the GPU hardware scheduler provides no public APIs for controlling its queue. Thus, none of the existing GPU libraries implements I/O-driven preemption.

In this paper we explore the idea of adding an I/O-driven preemption mechanism to GPUs. To understand the range of system parameters for which preempting the execution may improve the system throughput, we construct a simple performance model parameterized by the workload compute intensity ($\frac{compute}{IO}$) and the preemption overhead. We use this model to show that the preemption improves the system throughput for I/O bound GPU work-

loads, even when the preemption overhead is as high as one-half of the I/O time. This result motivates our further efforts to prototype a complete preemption mechanism in software, GPUpIO, on NVIDIA GPUs. We use GPUpIO to evaluate the overheads and analyze the system tradeoffs of using preemption in a set of I/O intensive workloads that perform disk I/O from GPU code.

The basic idea in GPUpIO is quite similar to the CPU preemption mechanisms: instead of waiting in a spinlock on the GPU, the threads blocked on an I/O operation are evicted from the GPU and then re-invoked when the I/O operation completes. Our design is built on the following ideas:

Preemption granularity We perform I/O preemption at the granularity of a single threadblock, terminating and later restarting all the threads in it. There are several reasons for this decision. First, evicting a few threads of a warp, or a few warps of a whole threadblock, would prevent warps from other threadblocks from using the same core, because the hardware scheduler operates at a threadblock granularity. Thus, GPU resources would still remain underutilized. Second, current GPU I/O libraries follow a threadblock-level API design. Thus, calls such as read/write/send/recv must be collaboratively and synchronously invoked by all the threads in a threadblock. Consequently, the blocking I/O calls block all the threads in the entire threadblock (even through only one thread is spinning). Finally, breaking a threadblock into parts greatly complicates the checkpoint-restart logic (described below), and makes it hard to maintain correctness of threadblock-wise synchronization operations.

Checkpoint-restore We implement a checkpoint library that saves the active state of a threadblock in global memory and then restores it when the threadblock is reinvoked later. In order to capture the active state of a threadblock, we build a CUDA source-to-source compiler that analyzes the original code and instruments it with the checkpoint and restart logic.

Preemption-restart The threadblock preemption is triggered by a GPU I/O library when the GPU issues an I/O call that is expected to block. The I/O call checkpoints the threadblock state, after which all the threads in a threadblock terminate. The CPU runtime may notify the GPU that the I/O call is expected to return shortly, which leads the GPU to choose spinning. The CPU runtime handles the I/O call and reinvokes the threadblock in a *separate* kernel in another CUDA stream. When re-invoked, the threadblock recovers its entire state. In particular, we replace the values of the hardware constants, such as `blockIdx` and `gridDim`, to match the values of the original kernel.

To evaluate the preemption mechanism, we integrate it with a simple I/O library that allows the GPU to read blocks from files. In our experiments we run several sets of mixed workloads containing different numbers of I/O bound and compute bound kernels. For the I/O-bound kernel we use a matrix product kernel that reads its inputs from hard disk. We choose the compute-intensive workloads from the Rodinia[1] benchmark suite.

Our results provide a number of interesting insights. First, the total system throughput is almost doubled, achieving optimal utilization with full I/O-compute overlap in some cases, despite the preemption overheads, as predicted by the model. Second, the preemption overhead is dominated by the threadblock restart time, whereas the overhead of the state checkpoint and restore is in the order of few microseconds and practically negligible. Third, we expose a limitation of the software-only I/O-driven preemption approach presented here: I/O intensive kernels are effectively starved by the compute kernels, which cannot be preempted.

The main contributions of this work are as follows:

- Analysis of the expected performance benefits of the I/O-driven preemption.

- A complete prototype that implements a preemption mechanism on commodity NVIDIA GPUs.

- Experiments that show significant throughput benefits of preemption on the one hand, but also highlight the need for hardware-assisted preemption mechanisms to avoid the starvation of I/O intensive kernels. We discuss one possible hardware assisted design in §7.

This paper begins with an overview of I/O-driven preemption in CPU OS and the challenges of adopting these methods for GPUs (§2). We then present an analysis for the performance gains of preemption (§3). We then discuss the design of a software-based solution (§4), our implementation (§5), and experimental results (§6). Finally, we discuss the limitations of a software-based solution and propose a hardware extension to alleviate the overheads (§7), survey related work (§8), and conclude (§9).

2. BACKGROUND

We provide an overview of I/O driven preemption on CPUs and the GPU software/hardware model. We use NVIDIA CUDA [6] since we implement GPUpIO on NVIDIA GPUs; however, most other GPUs that support the OpenCL standard [12] follow the same semantics and concepts.

I/O-driven preemption in CPU OS.

A classic interrupt-based I/O design enables efficient use of CPU resources while a process is blocked waiting for completion of an I/O operation. The process is preempted and removed from the OS run queue, making the CPU available for other processes, thereby overlapping the I/O with computations and improving CPU utilization. The process is resumed after completion of the I/O call, usually by an I/O interrupt handler.

GPU execution and scheduling.

GPUs are massively parallel processors that expose programmers to hierarchically structured hardware parallelism [4]. They comprise multiple cores – streaming multiprocessors (SM) – which have access to shared global memory. Multiple threads are grouped into a threadblock running on an SM, and multiple threadblocks form a GPU kernel. All threads in a threadblock share a limited, on-die shared memory, and are executed on the same SM.

If the number of threadblocks in a kernel exceeds the number of SMs on the GPU, the threadblocks are enqueued into a hardware queue from the CPU, and the GPU hardware scheduler dynamically schedules them on the SMs. Once a threadblock has been invoked on an SM, it cannot be preempted and occupies that SM until all its threads terminate.

The lack of control over threadblock scheduling, and the lack of support for interrupts, make it hard to build an efficient native I/O layer for GPUs. In light of the GPU hardware constraints, in this paper we explore the design and prototype of an I/O-driven preemption mechanism for GPUs.

3. PERFORMANCE MODEL

In this section we analyze the expected impact of preemption on system throughput. We do so by constructing a simple performance model for the system with and without preemption and comparing their throughput.

Consider a GPU with one SM that can run only one threadblock at each point in time. We denote I to be I/O operation time per threadblock, $C > 0$ to be the computation time per threadblock, α the preemption overhead per threadblock, and B the number of
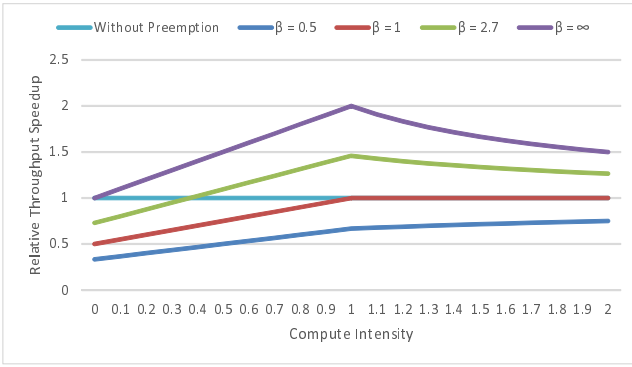
Figure 1: The relative throughput of a system with preemption over blocking I/O as a function of compute-to-IO ratio and different β. Smaller β correspond to higher overhead relative to the I/O time.

Figure 2: Preemption becomes unnecessary for compute-intensive workloads

threadblocks to run. The total execution time of an IO-bound kernel without preemption is:

$$B \times (I + C),$$

while the total execution time of the same kernel with preemption is:

$$B \times (\max\{I, C\} + \alpha) + C \approx B \times (\max\{I, C\} + \alpha).$$

The computation time per threadblock is not dominant for large enough values of B, and therefore we ignore it. Thus, the expected speedup due to preemption is:

$$\frac{I + C}{\max\{I, C\} + \alpha}.$$

This model is simplistic because it assumes the worst-case scenario where there is no overlap between computations and I/O in the system without preemption. Nonetheless, it allows us to intuit when preemption is worthwhile.

We evaluate the model for different compute-to-IO ratios, and for different $\beta \triangleq \frac{I}{\alpha}$. $\beta < 1$ indicates that the preemption overhead is larger than the I/O time. A larger β means lower overhead relative to I/O time. In addition, given a compute-to-IO ratio, β indicates the relationship between the compute time and preemption overhead.

The graph in Figure 1 shows the relative throughput of a system with preemption as a function of compute intensity (compute/IO). To obtain this graph, we vary the computation time for several fixed values of β.

The maximum is achieved at $\beta = \infty$ and compute intensity of one. This makes sense since we have a pipeline of two stages, and the best performance is guaranteed when these stages are completely balanced, under zero preemption overhead. The more balanced the pipeline, the lower the sensitivity of the performance to the negative impact of preemption overhead. However, for purely I/O bound workloads, preemption is less beneficial in general, and in fact slows down the system significantly.

For the workloads dominated by compute-intensive kernels with higher compute-to-IO ratio, the benefits of preemption decrease. We see this clearly in Figure 2.

We conclude that a preemption mechanism implemented purely in software might still improve the system throughput for many workloads despite the relatively high expected overhead.
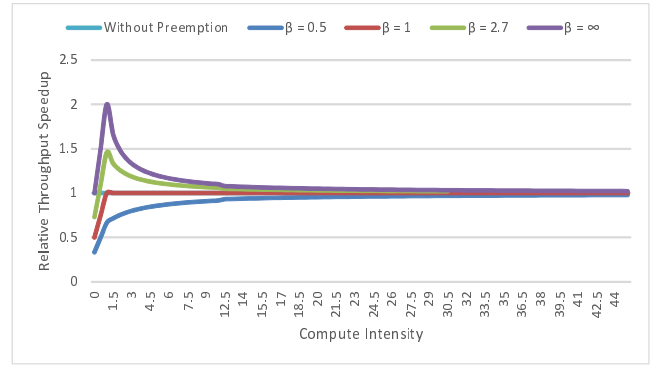
4. DESIGN

The design of the preemption mechanism involves two key components: a mechanism for kernel execution state checkpoint-restore, and a mechanism for preempting-restarting a threadblock. We now describe the main design considerations.

4.1 Checkpoint-Restore mechanism

Checkpoint-Restore (CR) is a mechanism that makes it possible to halt the execution of a running threadblock, checkpoint and seamlessly restore its state (registers, program counter, shared memory), and restart the execution where it left off at a later time.

Several main design challenges must be overcome to implement the checkpoint-restart functionality in the general case:

1. Determining the active state per thread.
2. Terminating threadblocks at an arbitrary point, difficult due to control flow divergence.
3. Limited amount of GPU RAM.

We address the first challenge by implementing a source-to-source compiler that analyzes the original code, determines the active state at the checkpoint, and instruments the code with the necessary save and restore procedures, as we describe in Section 5.1.

The second challenge is less critical for implementing I/O-driven preemption. This is because the existing GPU I/O libraries assume that an I/O call is invoked jointly and concurrently by all the threads in a threadblock at the same point in the code. Thus, at the point of an I/O call, it is guaranteed that there will be no divergence.

We defer the solution to the last challenge for future work. We do not expect this problem to be serious in practice, since in our experience the per-threadblock state is relatively small, and is much smaller than the theoretical maximum.

4.2 Preemption-Restart mechanism

A preemption-restart mechanism allows the system to preempt a certain threadblock and then restart it later. The preempt-restart uses the checkpoint-restore mechanism internally.

The preemption - restart sequence is as follows: (1) a running threadblock decides to preempt in coordination with the CPU; (2) the state checkpoint is created; (3) the threadblock terminates; (4) the threadblock is restarted later by the CPU runtime; (5) the execution state is restored, so the threadblock continues execution where it left off.

Designing this mechanism in the general case in existing GPUs is not trivial because:

1. GPUs provide no interrupt mechanism for the CPU for out-of-bound communication with a running GPU kernel. Existing

interrupt handlers may only terminate all GPU threads.

2. GPU schedulers do not provide control over their internal hardware queues, so a threadblock cannot yield and then return without terminating

The first challenge is irrelevant for the I/O-driven preemption mechanism we consider here, because the preemption is synchronous. Namely, the threadblock yields the GPU resources on its own, rather than preempted by a CPU.

To solve the second problem we approximate the ideal solution as follows: when a threadblock terminates, it stores the hardware constants like `gridDim` and `blockIdx`. When the threadblock is restarted, it is invoked as a *separate* single-threadblock kernel in a separate GPU stream. When the block is restored, its state, including the hardware constants, is restored as well.

Note that this solution does not fully overcome the problem. Without the ability to enqueue the restarting threadblock in the top of the hardware run queue, this threadblock may be delayed or even starved by other kernels. We elaborate on this limitation in Section 6.

5. IMPLEMENTATION

In this section we describe the implementation, optimizations, and limitations of GPUpIO. We implement a compiler and a run-time library (scheduler). The compiler enables checkpoint-restore by generating the code that saves and restores the active state at run-time. The run-time library implements the threadblock preemption-restart mechanism

5.1 Source-to-Source checkpoint-restart compiler

The compiler adds a block of code that saves the threadblock's state and halts execution. In addition, the compiler adds a restore block that restores the threadblock's state and continues the execution where it left off.

In order to allow a threadblock preemption, we introduce a `#pragma preempt` directive that indicates to the compiler where to generate the code for checkpoint and restore.

We implement a Source-to-Source compiler with the extended CUDA language front-end and a CUDA back-end. We base our implementation on *yaCF*, which is a front-end compiler for ac-cULL [8] that implements the OpenACC standard. *yaCF* follows the traditional approach of layer-oriented compiler design. A source file is parsed in the front-end into the intermediate representation (IR), transformations and optimizations are applied in the middle-end, and code is generated by the back-end. *yaCF* wraps *pycparser*, a C99-compliant Python front-end that uses an abstract syntax tree (AST) as an IR. We extend the grammar to support CUDA. We extend the back-end to generate CUDA code for the new pragmas, and finally we extend the middle-end to analyze the code and make the necessary transformations.

The compiler performs live variable analysis to find live variables that are needed after the restore point. These variables are checkpointed in a state struct, which includes all live variables and the instruction pointer.

The following list summarizes the operations performed by the compiler:

1. Parses the input code into an AST.
2. Locates the preemption points and performs a live variable analysis for checkpoint and restore. This includes a simple control flow analysis heuristic for avoiding preemption points in divergent areas, and shared memory checkpoint and restoration as defined by the layout of the shared memory.

3. Modifies the kernel by generating the appropriate checkpoint and restore blocks.
4. Declares a state struct that will be passed to the kernel as an argument and will hold the state of the kernel when evicted from the GPU.
5. Extends the kernel argument list with this struct
6. Generates a set of functions: allocate, free and copy for this memory area.

We consider a variable as live if and only if it is read after the restore point and it is not a constant. There is no reason to checkpoint variables that will be later overwritten.

We define the checkpoint cost function, denoted as COST(P), as the tuple (T, B), which is the state size (in bytes) that has to be stored for preempting P. T is the number of bytes per thread and B is the number of bytes per threadblock. The smallest preemption state is to store 4 bytes of the instruction pointer per threadblock. We note that our compiler may not be able to identify the state with the smallest possible preemption cost because we do not have access to the generated PTX code and perform transformations at the level of an AST.

The compiler declares a new data structure – state struct – that holds each threadblock's state in a struct of arrays (SOA), in order to make sure that the accesses to that struct at restore time are coalesced. Each field in this struct represents an array to store the contents of a register, shared memory, or the instruction pointer.

Since the state struct size is unknown statically, the compiler also generates all the memory management calls.

Checkpoint and restore.

The compiler splits the code into $n + 1$ sections, where n is the number of preemption points in the kernel. For each section it analyzes the state needed for the rest of the computation and creates the struct with the union of the states at different checkpoint points. The compiler injects a label at the beginning of each section, and an epilogue block that checkpoints the data to the input state struct and halts execution. The compiler also injects a barrier `__syncthreads` call in order to ensure that the whole threadblock is exiting. Therefore, placing a preemption point in a place where the behavior of `__syncthreads` is undefined, e.g., a potentially divergent if statement, is not allowed, and the compiler delays the preemption point to the nearest convergence point.

To enable only one threadblock of the kernel to be restarted, the compiler also extends the kernel's argument with an input of the original kernel's configuration parameters (the grid dimensions and the block dimensions).

Finally, the compiler injects a switch case in the beginning of the kernel. This switch case allows the restart logic to restore the correct state and jump to the correct preemption point.

Figure 3 illustrates the transformed kernel's layout. It shows a simple GPU kernel that invokes a read from disk followed by computation, and its transformed layout. Each threadblock invokes gread, which is a native GPU I/O function, similar to the read system call, and then performs some computation on the data. The threadblock preempts in cases when the data is not yet residing in the GPU global memory. We simplify the compiled version for the sake of brevity, and only visualize the main layout generated by our compiler.

5.2 The run-time library

The run-time library is linked to a user's GPU kernel. It performs kernel scheduling, restarting the preempted threadblocks when needed (e.g., upon the completion of the I/O call), and manages the memory for storing the state of the preempted threadblocks.

```
compute_from_disk(int fd)
{
    __shared__ char buff[BUFF_LEN];

    if (!gread(fd, buff, BUFF_LEN)) {
        // preempt if the data is not ready
        #pragma preempt
    }

    __syncthreads();

    compute(buff, BUFF_LEN);
}
```

```
compute_from_disk(int fd, struct backup back, ...)
{
    // restore execution configuration
    __shared__ char buff[BUFF_LEN];

    switch (back.checkpoint[TB]) {
        case 0:
            goto begin;
        case 1:
            // restore state
            goto pragma_1;
        default:
            return;
    }

begin:
    if (!gread(fd, buff, BUFF_LEN)) {
        // checkpoint the threadblock's state
        back.checkpoint[TB] = 1;
    }

    __syncthreads();

    if (back.checkpoint[TB] == 1) {
        // checkpoint private registers
        return;
    }

pragma_1:
    compute(buff, BUFF_LEN);
}
```

Figure 3: The layout of a transformed kernel. The left side is the original kernel, and the right side shows the final code generated by GPUpIO.

The memory management is done by the allocate, free and copy functions that are created by the compiler for each preemption point. The run-time library is aware of these generated functions and calls them when necessary.

The user invokes its kernels via the scheduler, and the GPU I/O runtime uses the scheduler to restart individual threadlbocks.

5.3 Limitations

Our current prototype has a number of limitations.

- pycparser does not support C++, and therefore our parser fails to parse templated kernels, or kernels using templated CUDA functions such as `tex2D`.
- The scheduler invokes the preempted threadblocks one by one instead of aggregating them in a single kernel invocation, potentially reducing the overhead. On the other hand, we found that launching many small kernels might actually improve the system's throughput by allowing multiple kernels to overlap.

6. EVALUATION

We evaluate our implementation on a machine featuring a 12-core Intel Xeon E5-2620 CPU, NVIDIA Tesla K40m GPU with 12GB of GDDR5 memory, and a regular HDD Seagate Barracuda ST1000DM003 7200 RPM. We run Ubuntu Linux kernel 3.13.0-32, with CUDA SDK 7.0, GPU driver 352.39. We evaluate the system performance with a set of microbenchmarks. For every data point we report the median of 10 trials.

6.1 Overhead of the state checkpoint-restore

We define the checkpoint time as the time between the beginning of the preemption until the threadblock terminates. Similarly, the restore time is the time between the point when the block is restarted on GPU, until its state is fully restored. We measure the time using `clock64()` intrinsic.

We run the HotSpot application from the Rodinia [1] benchmark suite (256 threads per threadblock, and 1849 threadblocks in total, achieving full occupancy), and measure the checkpoint and restore times for different state sizes. We change the checkpoint location in the code to vary the state size. We measure the checkpoint and restore times for all the threadblocks and report the average.

Figure 4 and Figure 5 show the checkpoint and restore times when the state is stored in registers alone, and when in addition a threadblock allocates 2KB of shared memory. Observe that the total time for the state checkpoint and restore is in the order of few μsec. Observe also that our checkpoint library imposes zero overhead if it is not used (state size zero in Figure 4).

6.2 End-to-end benchmarks

Matrix product from files.

We implement a matrix product kernel AB^T, which reads the inputs from disk, in row-major data layout, via a native GPU I/O layer specifically tailored for matrix product logic. We note that the effective checkpoint state for this code is only 4 bytes used for the metadata by the checkpoint layer. The data is read in the beginning
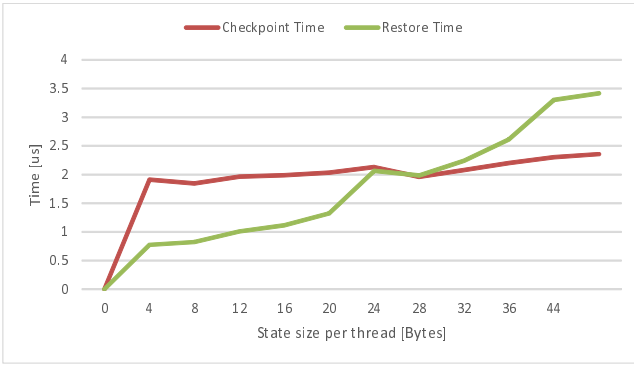
Figure 4: State checkpoint/restore cost per threadblock, registers only.
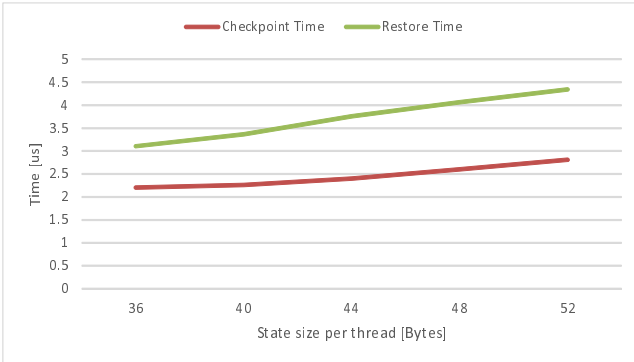
Figure 5: State checkpoint/restore cost per threadblock, registers plus 2KB of shared memory.

of each threadblock, and therefore the size of the active state is zero.

The GPU I/O layer transfers all the I/O requests to the CPU via CPU-GPU shared memory. The CPU reads the requested matrix block from the file and copies it into GPU memory. We implement a simple cache of read input blocks to avoid disk access if the GPU requests a block that has already been transferred into GPU memory. The CPU also keeps track of all the pending disk requests and eliminates duplicate requests generated concurrently by different threadblocks. Finally, the I/O library implements the threadblock preemption mechanism introduced in this paper.

Mixed workload.
We run a mixed workload comprising compute-intensive and I/O-intensive kernels. We choose SRAD and PathFinder from the Rodinia benchmark suite to represent compute-intensive applications that perform no file I/O. The SRAD application invokes four kernels, each running for 100ms on average. The PathFinder application invokes many short kernels for a total of 400ms. We use the matrix product that reads its inputs from files as an I/O intensive application. Each matrix is a squared matrix of order 1K with float elements, and its execution (including I/O) fully occupies the device for 80ms, where the compute intensity is about 6%.

In the experiments we vary the number of I/O-intensive kernels from 1 to 10, each reading from a different file, while keeping the number of compute-intensive kernels the same. This allows us to change the relative amount of I/O in the workload. Note that the execution times are tailored to achieve full overlap between the com-
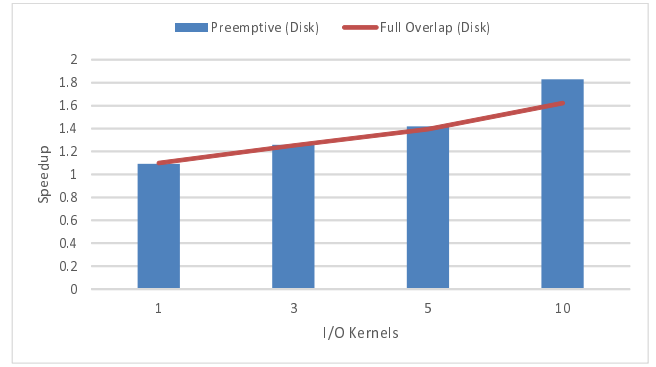
Figure 6: Relative throughput for mixed workload with disk. Higher is better.

putations and I/O. In each experiment we invoke all the kernels at once by queuing them into different CUDA streams. We flush the OS page cache before each trial to enforce reads from disk.

Throughput with disk I/O.
We measure the improvement in system throughput by comparing the total runtime for all the kernels together, with and without preemption. The results in Figure 6 show that the throughput reaches the upper bound (assuming full overlap of compute and I/O) for 5 I/O kernels. We set the full overlap trend to be an approximation for the ideal case where the I/O is completely hidden by computation. However, as can be seen in the results, an overlap of the restarted matrix product threadblocks and the PathFinder kernels can even further improve the system's throughput.


Figure 7: A profiler snapshot of 5 concurrent IO kernels reading from disk using spinning and the two compute kernels. The horizontal axis is time and the vertical axis shows compute and DMA transactions. The first row (with brown lines) shows the asynchronous memory copies. The other rows show kernel computations. If there is an overlap between two kernels, the profiler inserts a new row with that kernel.

Figure 8: A profiler snapshot of 5 concurrent IO kernels reading from disk with preemptive IO and the compute kernels. Long lines are compute kernels. Note that the horizontal axis scales here and in Figure 7 are different, but the actual kernel execution time is the same.

To illustrate the effect of preemption on the kernel execution in the GPU, Figure 7 and Figure 8 show the screen shots from the NVIDIA visual profiler for the GPU occupancy for 5 I/O and the compute kernels without and with preemption respectively. Without preemption the kernels are serialized. Preemption enables full
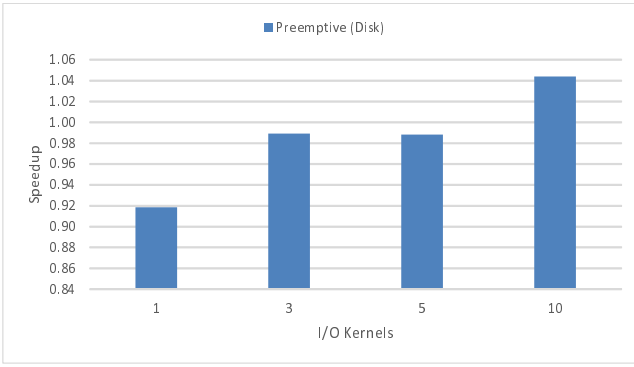
Figure 9: Relative throughput for I/O bound workload. Higher is better.

Figure 10: Relative throughput for mixed workload with RAMFS. Higher is better.

Figure 11: Relative throughput for IO bound workload using RAMFS. Higher is better.

overlap between I/O and computations, and shortens the waiting time of compute kernels (long lines at the beginning).

IO-bound workload.

We analyze the benefits of preemption for I/O-bound workloads alone. In the experiments we invoke one or more I/O-intensive matrix product kernels, but omit compute-intensive workloads from the mix.

As Figure 9 shows, preemption overhead results in performance degradation for a single kernel. We found that all of the 1024 threadblocks invoked by the kernel are preempted, and then re-enqueued one after the other in different kernels. The computations alone without I/O require about 5msec, and the overhead for re-execution is about 20-50μsec per threadblock; hence the overhead is comparable with the compute time. The performance degradation is correctly predicted by our model, which shows that with low compute intensity and high overheads preemption is undesirable.

Note that when the number of I/O kernels grows, performance improves slightly. One reason is that in different kernels the overlap is not only between I/O in one kernel and computations in the other, but also between computations in both kernels, resulting in better GPU utilization. Another reason is the reduction in total I/O time, most likely as a result of a better access pattern to the disk.
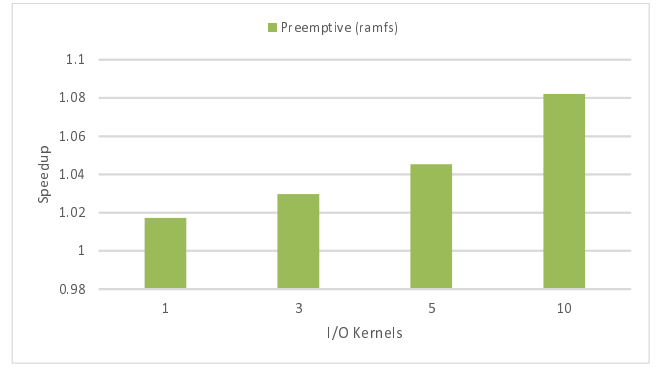
Throughput with RAMFS.

We consider the impact of I/O time on the throughput, with and without preemption. In this experiment we run the same workloads but use the RAMFS in-memory file system to store the input. Figure 10 shows the results for the mixed workload, and Figure 11 shows the performance for the I/O workload alone. As expected, the speedup due to preemption is quite modest in the first case, because fewer overlap opportunities are available. Also as expected, in the I/O-bound workload the preemption overhead dominates both I/O and compute time, which leads to performance degradation.

Kernel runtime analysis.

We evaluate the impact of preemption on individual kernels, specifically, on the total time a kernel spends in the system both waiting and executing. All the kernels are enqueued at once, with the I/O kernels in the top of the queue. As before, we invoke the experiment with and without preemption, but now measure the completion time for each kernel and compute the average for I/O-intensive and for compute-intensive kernels separately.

Figure 12 and Figure 13 show the results. The total time in the
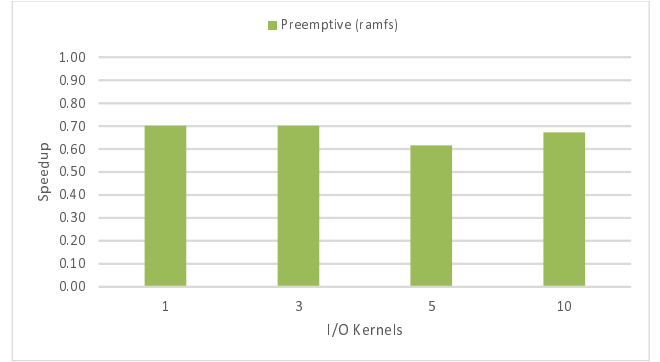
system for an I/O kernel with preemption increases dramatically, whereas the compute kernels are processed much faster.

This experiment clearly shows that I/O kernels get *starved* by the compute kernels. Initially, a preempted threadblock is replaced by a threadblock of a compute kernel. However, since the current system restarts it later as a single-threadblock kernel, the threadblock is effectively placed in the hardware runtime queue *after* all the threadblocks of the compute kernel. Therefore, each preempted threadblock is delayed substantially.

To highlight this effect further, we compare the average restart delay – the time a preempted threadblock spends outside the GPU (i.e., the time from when the preemption is triggered until the threadblock is restarted), for the mixed workload and the pure I/O-bound workload with I/O kernels alone. The results are in Figure 14 and Figure 15 respectively. We see that the average restart delay in Figure 14 is higher by about 300μsec regardless of the number of I/O kernels invoked, showing that all preempted kernels get delayed by the same time as compute kernels.

The starvation can be eliminated if the GPU allowed re-enqueuing threadblocks to the top of the execution queue. However, we are not aware of any such capability.

7. DISCUSSION

We believe that native GPU I/O support is an essential part of the GPU general-purpose acceleration portfolio. However, without proper support for I/O-driven preemption, the native GPU I/O may be too wasteful in terms of power and compute resources. In this paper we show that such support is beneficial in many cases,
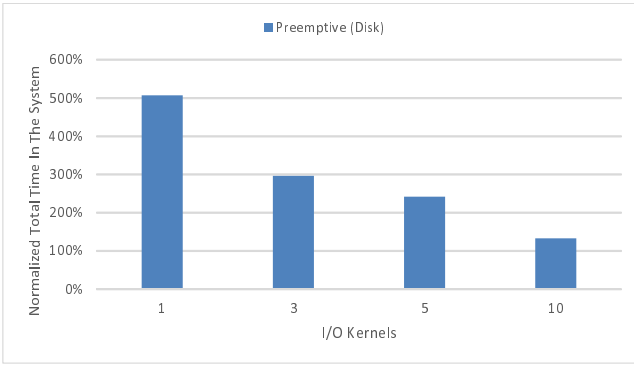
Figure 12: The relative average total time in the system per IO kernel in mixed workload. Lower is better.
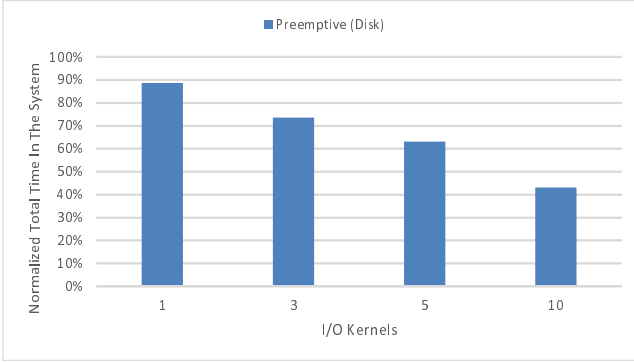
Figure 13: The relative average total time in the system per compute kernel in mixed workload. Lower is better.

Figure 14: The average restart delay per I/O threadblock in a mixed workload. Lower is better.

Figure 15: The average restart delay per I/O threadblock in a I/O workload. Lower is better.

describe the basic building blocks for designing the preemption mechanism, and build a working prototype to analyze preemption in a practical setting.
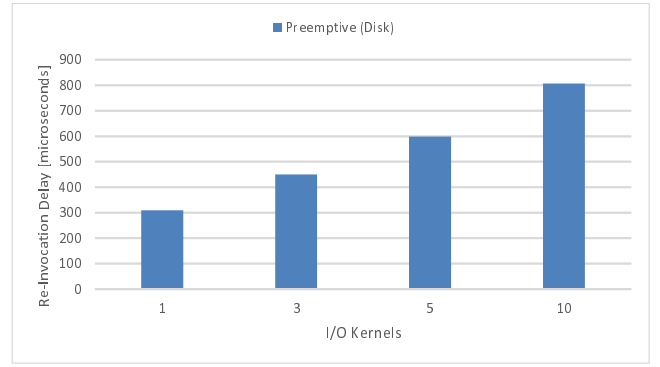
The limitations of the software-based solution are, however, apparent: high re-invocation overhead and I/O kernel starvation. Below we discuss a new hardware feature that may help alleviate these problems.
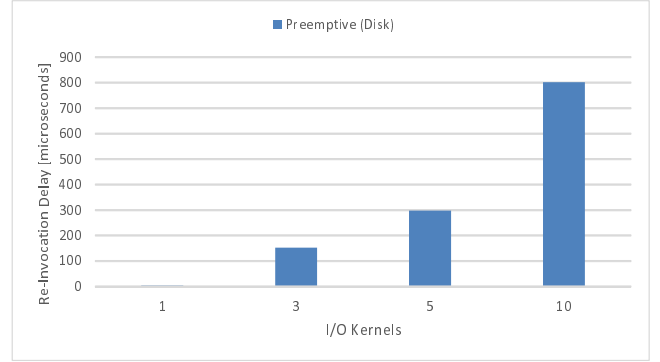
Hardware yield instruction.

As we observe in Section 6, I/O-driven preemption is beneficial for workloads with enough I/O to provide the opportunity for overlapping I/O with computations. But if the workload is dominated by compute-bound kernels, the preemption results in starvation of the preempted threadblocks. In the most general case, overcoming this problem requires adding a complete on-demand preemption capability to the GPU, which might be too intrusive. Here we propose a new GPU hardware feature that may help improve this problem without major changes to the architecture.

We propose a new hardware instruction called `yield(void *addr)`. This instruction behaves similarly to the x86 mwait and monitor pair. Specifically, when a threadblock calls yield, it terminates and is removed from the execution queue. Upon write to addr, the threadblock is enqueued into the *head* of the hardware queue. The thread state checkpoint/restore can be done either by hardware or by software.

The `yield` instruction ensures that the check of addr and the preemption decision are performed *atomically* to avoid deadlock. The `yield` instruction has no effect if the scheduling queue is empty. `yield(NULL)` may evict the threadblock and enqueue it again in the ready queue, which implements a kind of spinning.

The `yield` instruction can be useful beyond preemption support, for example for implementing GPU producer-consumer, conditional variables, and others, and it might not require significant changes in the existing hardware. In fact, we speculate that NVIDIA GPUs already provide some form of preemption support to allow nested parallelism in CUDA. A parent threadblock might be descheduled in favor of the child kernel it invokes. Our attempts to leverage the existing nested parallelism mechanisms for our preemption have not been successful so far.

8. RELATED WORK

To the best of our knowledge, GPUpIO is the first software-based solution that allows preemptive I/O on GPUs. This work has a bearing on many areas, from I/O support for GPUs, to compilation methods, GPU architectures, and programming techniques.

Preemption for GPUs. Chimera [7] proposes a set of hardware extensions such as SM flushing, a GPU-specific preemption that is enhanced to exploit the semantics of thread blocks in the GPU programming model and the concept of idempotence to achieve low preemption latency. However, Chimera focuses on preemption for collaborative multitasking purposes.

OS I/O support for GPUs. GPUnet[3] and GPUfs[11] provide, respectively, a native networking abstraction and file I/O API for GPU applications, providing both programmability and performance benefits. However, they share the same major limitation: they spin on an I/O control register while waiting for the I/O operation to

complete.

Scheduling for heterogeneous processors. TimeGraph[2] provides better isolation and prioritization capabilities for real-time tasks, by applying different scheduling policies on command groups issued by GPU accelerated applications. PTask[9] proposes a dataflow programming model that is leveraged by the OS scheduler to provide fairness and performance isolation for heterogeneous processors.

Checkpoint Restore for GPUs. CheCUDA[13] and NVCR[5] provide a checkpoint-restart mechanism for CUDA applications in the context of fault-tolerance and CUDA application migration. However, these works do not deal with the complex memory hierarchy of the GPU and allow checkpointing only in between kernel invocations. Consequently, these implementations cannot be used for preemptive I/O.

Compiler support for GPUs. Dandelion[10] is a language and system support for data-parallel applications on heterogeneous architectures, which effectively offloads portions of data-parallel code to available computing resources.

9. CONCLUSION

In this paper, we addressed the problem of preemptive I/O on GPUs. Due to the lack of hardware support on today's GPUs, native GPU I/O layer implementations are forced to actively spin on the GPU, which results in lower overall throughput and higher energy consumption. We motivated our work with a simple, but enlightening performance model, showing the promise of I/O driven preemption. We proposed a full software-based design composed of two major key mechanisms: checkpoint-restore and preemption-restart. We implemented a prototype, GPUpIO, and extensively evaluated our solution on several workloads. We showed that our approach achieves optimal GPU utilization with full I/O-compute overlap in some cases, despite the challenging hardware characteristics. We also recognize the high overheads imposed by the preemption, in our experiments with RAMFS.

Finally, we showed that due to the limited interface given to the internal GPU scheduling queues, a software-based solution results in starvation of I/O kernels. We proposed a hardware mechanism, `yield`, that may alleviate this problem.

10. REFERENCES

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[2] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[3] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, Broomfield, CO, Oct. 2014. USENIX Association.

[4] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[5] A. Nukada, H. Takizawa, and S. Matsuoka. NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA.
In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 104–113, May 2011.

[6] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[7] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[8] R. Reyes, I. Lopez, J. Fumero, and F. de Sande. accULL: An User-directed Approach to Heterogeneous Programming. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 654–661, July 2012.

[9] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[10] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[11] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13. ACM, 2013.

[12] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[13] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 408–413, Dec 2009.