Efficient Implementation of GPGPU Synchronization Primitives on CPUs

Jayanth Gummaraju Ben Sander Laurent Morichetti Benedict Gaster Lee Howes

Compute Research Group, Advanced Micro Devices, Sunnyvale, CA-94085, USA
firstname.lastname@amd.com

ABSTRACT

The GPGPU model represents a style of execution where thousands of threads execute in a data-parallel fashion, with a large subset (typically 10s to 100s) needing frequent synchronization. As the GPGPU model evolves to target both GPUs and CPUs as acceleration targets, thread synchronization becomes an important problem when running on CPUs. CPUs have little hardware support for synchronization and must be emulated in software, reducing application performance. This paper presents software techniques to implement the GPGPU synchronization primitives on CPUs, while maintaining application debug-ability. Performing limit studies using real hardware, we evaluate the potential performance benefits of an efficient `barrier` *primitive.*

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming

General Terms: Performance.

Keywords: GPGPU, Multicore, Synchronization.

1. INTRODUCTION

Recently, the high performance of GPU architectures has led developers to use graphics hardware for more general-purpose applications i.e., GPGPU model. Many GPGPU programming frameworks have been developed, from streaming models based on Brook [3], ATI CTM [1], and NVIDIA CUDA [5] to the more recent OpenCL^{TM} [2]. These frameworks run control code on the CPU and performance critical data-parallel code on the GPU using GPUs as accelerators.

The onset of multicore CPUs and closely integrated CPUs and GPUs, are driving the GPGPU model to encompass CPUs as acceleration targets in addition to GPUs [7]. The rapidly increasing FLOPS on CPUs complements well the FLOPS on GPUs. A key challenge in extending the GPGPU model to encompass CPUs is to reduce the overhead of thread synchronization. The GPGPU model embodies a style of execution where thousands of threads execute in a data-parallel fashion, with a large subset of threads (typically 10s to 100s) needing to synchronize frequently to manage shared local memories. Unlike GPUs, CPUs have little hardware support for synchronization, which therefore must be emulated in software. Hence, the cost of synchronization could be several orders of magnitude higher for CPUs, reducing application performance.

In this paper, we present new software techniques to implement GPGPU synchronization primitives on CPUs. We show how these primitives can be implemented entirely inside the runtime system. This ensures easy debug-ability of the application kernels and preserves the use of standard, off-the-shelf compilers and debuggers with only minor extensions, unlike other approaches (e.g., mCUDA [7]). Using limit studies, we demonstrate the potential performance benefits of efficiently implementing the `barrier` primitive for several GPGPU applications on real hardware (AMD Phenom^{TM} II).

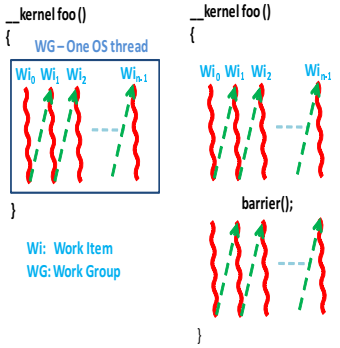

Figure 1: Work-item Scheduling

2. MAPPING THE GPGPU MODEL

Without loss of generality, we use OpenCL terminology [2]. In OpenCL, a control thread executing on the host (e.g., CPU) dispatches kernels onto a compute device (e.g., GPU). The compute device breaks the kernel execution into *work-groups* each comprising of several threads or *work-items*. The work groups are dynamically scheduled on the cores. Once all work groups finish execution, the control thread is notified.

Mapping work-groups comprising hundreds of work-items onto CPUs using conventional methods (e.g., pthreads, OpenMP [6], etc) is inefficient since CPUs have only a few hardware contexts, and context switching between hundreds of work-items is very expensive (several 1000 cycles per context switch). The cache locality between work-items of the same work-group is also lost since the work-items potentially execute on different cores.

Our implementation maps work-items of the same work-group to the same core (Figure 1). A single OS thread per core time-multiplexes work-items. The local memory, used for data shared between work-items, is conceptually mapped to a portion of the L1 data cache, leading to efficient communication between work-items.

CF'10, May 17–19, 2010, Bertinoro, Italy.
ACM 978-1-4503-0044-5/10/05.

```
__kernel foo()
{
      :
      :
      :
   barrier();
      :
      :
      :
}
```

```
WI::barrier() //Inside the runtime system
{
     //save current context
     if (setjmp() != 0)
        return; //if returning from longjmp

     if (nextWIContext == NULL)
        longjmp(); //to createWIContext

     longjmp(); //to nextSavedWIContext
}
```

Figure 2: Barrier Implementation

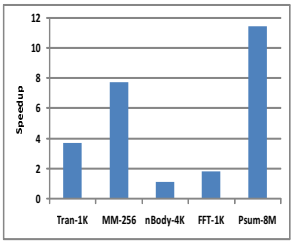

Figure 3: Zero-overhead Barrier.[Transpose: 1K matrix; Matrix multiply: 256x256 matrices;]

3. SYNCHRONIZATION PRIMITIVES

The GPGPU synchronization primitives fall into three main categories: *Atomics*, *Fences*, and *Barriers*.

On GPUs, atomic updates (e.g., atom_add/xchg) are implemented using custom hardware for both local and global memories. On CPUs, atomics to local memory are turned into `nop`s because work-groups execute on a single CPU core, and atomics to global memory are implemented in the runtime using `atomic` prefixes (e.g., `lock` prefix in x86). This prefix ensures that the data-bus/cache line containing the data is exclusively available before performing the update.

Memory fences on the GPU are typically implemented using `ACK` instruction (e.g., `WAIT_ACK` in ATI ISA) which waits for all the outstanding writes to the specified memory to be committed. On the CPUs, as with atomics, the fences to local memory are turned into `nop`s permissible because of the CPU memory consistency model. Global memory fences are implemented in the runtime using a `fence` instruction (e.g., `mfence` in x86) which is equivalent to the GPU `ACK` instruction.

The `barrier` primitive mandates all work-items of the same work-group to reach this point before executing further. GPUs have a `BARRIER` instruction with custom hardware support that typically takes just a few cycles [4] to execute (a few 10s of cycles for large work-groups). Threads waiting on the `barrier` can be temporarily suspended and easily replaced with another set of active threads, effectively hiding even this overhead. In contrast, `barrier` on CPUs needs to be emulated in software consuming several thousand cycles.

We implement the `barrier` primitive on the CPUs by transforming the problem into one of reducing the context switching overhead. Figure 2 shows the pseudo-code for our `barrier` implementation inside the runtime. When a work-item executes *barrier*(), the runtime intercepts the call and executes *setjmp*() to save the current work-item context. Depending on whether or not the next work-item context already exists, the control jumps to a saved context or to a newly created context using *longjmp*(). Once all the work-items execute *barrier*(), the control jumps back to the saved context of the first work-item, and so on (Figure 1). For good performance, we are investigating custom implementations of *setjmp*()/*longjmp*().

The memory layout of the work-item stacks, which are saved and restored upon context switch, also plays a critical role in the execution time of `barrier`. The runtime allocates a contiguous region of memory and splices it into contiguous chunks for each work-item stack. The location of these stacks directly affect their cache placement and TLB utilization. We are currently investigating techniques for efficiently utilizing caches and TLB.

4. PRELIMINARY EVALUATION

We performed limit studies to evaluate the potential benefits of efficient `barrier` implementation. We turned the `barrier` into a `nop`, eliminating its overhead. We ran experiments on 3.2GHz AMD Phenom II Quad-core system with 64-bit Ubuntu Linux. Each core has a 2-way 64K L1-D cache and 16-way 512K L2 cache.

Figure 3 shows the potential speedup for a few application kernels over a baseline implementation using standard `setjmp`/`longjmp`. The average speedup across all applications is 5.1X. Prefixsum can be improved by more than 11X if barrier overhead is eliminated. We see little benefit for compute-intensive applications that call `barrier` infrequently (e.g., nbody). Overall, these results show that several applications can be speeded up significantly with efficient implementation of `barrier`.

5. CONCLUSION

Synchronization in GPGPU applications is an important problem especially while executing on the CPUs. While `atomic` and `memory fence` operations have hardware support on CPUs analogous to the GPUs, there is little hardware support for implementing `barrier` on the CPUs. We presented software techniques to implement the `barrier` primitive, while maintaining the debug-ability of the application kernels. Our limit studies show that GPGPU applications can be speeded up significantly by reducing `barrier` primitive overhead.

6. REFERENCES

[1] ATI CTM. `ati.amd.com/companyinfo/researcher/documents/ATI\_CTM\_Guide.pdf`.

[2] OpenCL. www.khronos.org/opencl/.

[3] Buck, I et al. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[4] D. Cedarman and P. Tsigas. On dynamic load balancing on graphics processors. In *Graphics Hardware*, 2008.

[5] NVIDIA Corporation. Cuda programming guide 2.0, 2008.

[6] OpenMP Architecture Review Board. *OpenMP Application Program Interface 3.0*, 2007.

[7] Stratton, J.A. et al. M-CUDA: An efficient implementation of CUDA kernels on multicores. *Int'l Workshop on Languages and Compilers for Parallel Computing*, 2008.