Improving Application Concurrency on GPUs by Managing Implicit and Explicit Synchronizations

Michael Butler
Department of Electrical and Computer Engineering
University of Missouri – Columbia
michael.butler@mail.missouri.edu

ABSTRACT

Refer to poster for abstract

1. INTRODUCTION

GPUs have been widely used to accelerate a variety of applications from different domains, and are progressively being introduced in shared computing environments. Recent advances in GPU architecture have been increasing the hardware parallelism of these devices to allow for more concurrency and therefore more device utilization. There are many runtime systems which virtualize GPUs and run multiple applications on each; however, all of these systems overlook many types of blockages caused by the GPU software stack, potentially leading to underutilization of the devices. This work focuses on CUDA, but can be ported over to OpenCL.

2. MOTIVATION

Synchronization and serialization points introduced by the software stack can cause device underutilization, and there are five primary causes. First, any primitive that issues to the default stream must wait for the device to be idle before it can begin executing and likewise all other primitives must wait until it has finished before they can begin executing, effectively idling the device. Second, when the device only has a single work queue (ie. pre-Kepler architecture), any time a primitive must wait for a previous primitive to finish executing, all subsequent primitives are blocked, effectively synchronizing the entire device. Third, also for pre-Kepler GPUs, if an instruction incurs a dependency check, it must wait until all kernels that have entered the device have begun executing before continuing, again causing blockages. Fourth, for all GPU architectures, any memory transfer call acting on unpinned (pageable) memory is always synchronous with respect to the host, which serializes calls from a thread even across streams. Finally, in a similar problem only faced by Kepler and post-Kepler architectures, a memory transfer acting on unpinned memory must first reserve a copy engine, even if it cannot run due to an implicit synchronization, preventing any thread ready to use that copy engine from running. These synchronizations and serializations need to be prevented for better device utilization.

3. DESIGN

This runtime system (Figure 1) introduces four new components which together are able to manage explicit and implicit synchronizations. These components include an Instruction Translator, a Preallocator, a Memory Transfer System, and a Stream Manager. At a high level of understanding, these systems work together to ensure that an instruction is not uploaded to the GPU unless it is guaranteed to run without blocking subsequent calls.

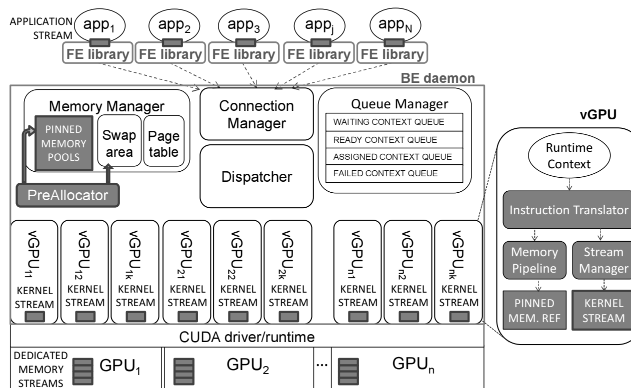

Figure 1: Extended architecture of the VM-GPU [3]: in white the original components, in grey our extensions. The internals of the vGPU design are detailed to the right.

The Instruction Translator receives a request and (when possible) delegates it to a stream dedicated to that request so that there is never an implicit synchronization. These dedicated streams include three memory streams (DeviceToHost, HostToDevice, DeviceToDevice) and separate kernel streams dedicated to each application. The Preallocation system accounts for those instructions which cannot be sent to a dedicated stream—such as cudaFree and cudaStreamCreate—by allocating all necessary resources when the runtime first starts, and it virtually allocates and frees these resources when requested. The Memory Transfer System solves the problem of unpinned memory serializations by using pinned (page-locked) memory. To ensure the system is not overburdened with too much pinned memory, data is pipelined through a small pool of pinned memory, which also serves to increase transfer bandwidth. Finally, the Stream Manager detects when a kernel has finished executing without incurring an implicit synchronization. It does this by inserting code into a kernel at runtime, causing it to increment a counter whenever a block has finished executing. After issuing a kernel, the Stream Manager simply polls this counter by asynchronously copying the memory on a dedicated stream, recognizing the kernel has finished when the number returned equals the number of blocks executed.

4. EXPERIMENTAL RESULTS

In order the gauge the efficiency of the runtime system, four CUDA SDK programs (Vector Addition, Black Scholes, Eigenvalues, and Reduction) and two scientific applications (N-body and Himeno) were used as test cases. Two different types of tests were performed: gauging the speedup of a single application and gauging the speed of two applications running concurrently.

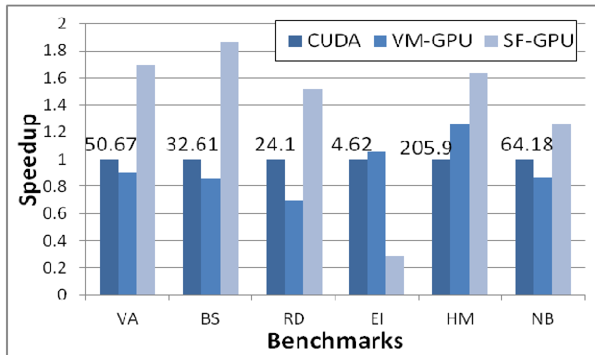

Figure 2: Single application speedup -- memory transfer

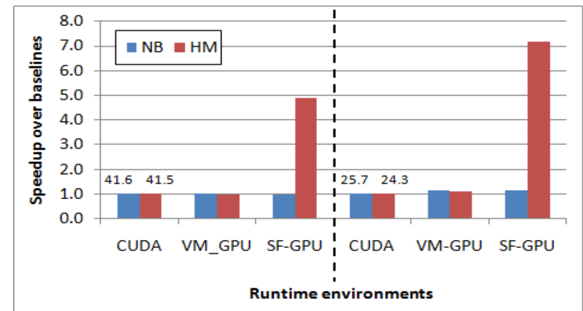

Figure 3: Speedup of HM and NB over the baselines when running in a shared GPU. Left: Tesla C2050 (Fermi), Right: K20 (Kepler)

The former test used only a Fermi GPU, whereas the latter test used first the Fermi GPU, then the Kepler GPU.

4.1 Single Application Evaluation

Due to the pinned memory pool in the Memory Transfer system, the runtime automatically accelerates data movements to and from the device in nearly all cases, achieving typically between 1.5x and 1.9x speedup as seen in Figure 2. However, the system has degraded performance if there are many miniscule data transfers, as with Eigenvalues.

4.2 Multitenancy Evaluation

As seen in Figure 3, by removing implicit synchronizations and by allowing HM kernel launches to progress independently, the runtime achieves a performance improvement as high as 4.8x and 7.1x for the Fermi and Kepler GPUs, respectively.

5. REFERENCES

[1] Nvidia. GPU-Accelerated Applications Catalog: *http://www.nvidia.com/content/tesla/pdf/gpu-apps-catalog-mar14-digital-fnl-hr.pdf*

[2] V. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *Proc. of HPDC '11*, San Jose, CA, USA, June 2011, pp. 217-228.

[3] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A Virtual Memory Based Runtime to Support Multi-tenancy in Clusters with GPUs. In *Proc. of HPDC '12*, Delft, The Netherlands, June 2012, pp. 97-108.

[4] B. Van Werkhoven*, et al.*, "Performance Models for CPU-GPU Data Transfers," in Proc. of CCGrid 2014.

[5] A. M. Aji*, et al.*, "On the efficacy of GPU-integrated MPI for scientific applications," in Proc. of HPDC 2013.