Improving Application Concurrency on GPUs by Managing Implicit and Explicit Synchronizations

Michael Butler, University of Missouri

Abstract

GPUs have progressively become part of shared computing environments, such as HPC servers and clusters. Commonly used GPU software stacks (e.g., CUDA and OpenCL), however, are designed for the dedicated use of GPUs by a single application, possibly leading to resource underutilization. In recent years, several node-level runtime components have been proposed to allow the efficient sharing of GPUs among concurrent applications; however, they are limited by synchronizations embedded in the applications or implicitly introduced by the GPU software stack.

In this work, we analyze the effect of explicit and implicit synchronizations on application concurrency and GPU utilization, design runtime mechanisms to bypass these synchronizations, and integrate these mechanisms into a GPU virtualization runtime named Sync-Free GPU (SF-GPU). The resultant runtime removes unnecessary blockages caused by multitenancy, ensuring any two applications running on the same device experience limited to no interference. Finally, we evaluate the impact of our proposed mechanisms.

Introduction

GPU applications

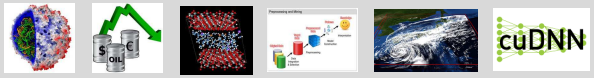
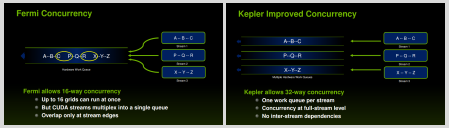


image processing | option pricing | molecular dynamics | data mining | weather prediction | deep neural networks

Streams and Concurrency

- Streams enable instruction concurrency
- Overlap kernel executions and memory transfers
- Streams = SW, work queues = HW



http://on-demand.gputechconf.com/gtc/2012/presentations/S0642-GTC2012-Inside-Kepler.pdf

Objectives

- Maximize GPU utilization and application throughput
  - Eliminate unnecessary waiting times by managing implicit and explicit synchronizations
- Improve execution of single application automatically
- Develop and describe tools that can be used both as a runtime system for virtualized environments and as a library to have dedicated code run faster

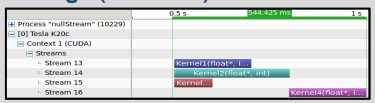
Implicit Synchronizations and Serializations

Notation	Explanation
K	GPU Kernel
M_A	Memcpy Device to Host
M_B	Memcpy Host to Device
S	Call that causes implicit or explicit synch

Default Stream Blockage (all GPUs)

Instruction order:
K1, K2, K3, S0, K4

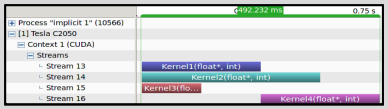
Any call to the default stream will synchronize the entire device.



Single Hardware Work Queue Blockage (pre-Kepler GPUs)

Instruction order:
K1, K2, K3, S1, K4

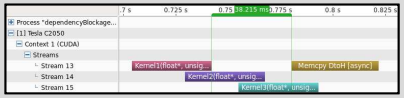
For single work queue, synchronizations will block subsequent calls.



Pending Kernel Call Blockage (pre-Kepler GPUs)

Instruction order:
K1, K2, K3, M_A1

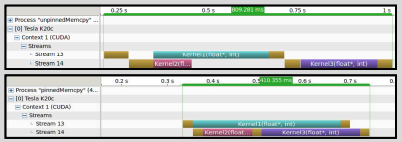
An instruction is implicitly blocked if there is a dependency check and a prior pending kernel call.



Unpinned Memory Serializations (all GPUs)

Explanation:
- Same instruction order
- Top uses unpinned (pageable) memory
- Bottom uses pinned (page-locked) memory

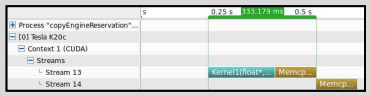
Unpinned memory always synchronous with host, serializes calling thread.



Copy Engine Reservation Blockage (Kepler and post-Kepler GPUs)

Instruction order:
K1, M_B1, M_B2

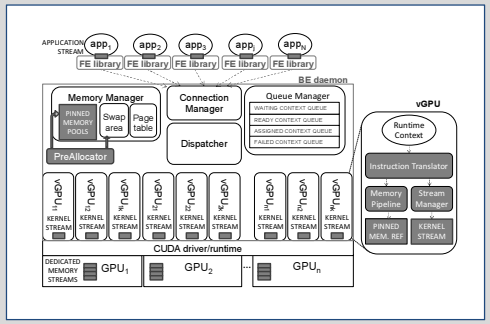
Unpinned memcpy instructions can reserve a copy engine and still be blocked by an implicit synchronization, also blocking memcpy from other threads.



In the above example, M_B1 is issued by one thread, M_B2 by another

Runtime design

Runtime architecture



1) Instruction Translator

What it does
- Control instructions are either ignored or handled by the preallocator
- Memory transfer instructions are redirected to use a dedicated memory stream (DtoH, HtoD, DtoD)
- Kernel sent to dedicated vGPU stream to increase parallelism

Why it is necessary
- Removes dependency checks which cause synchronizations
- Increases parallelism of runtime

2) Preallocator

What it does
- Allocates all necessary resources (memory, streams, events, etc.) when the runtime starts

Why it is necessary
- Prevents synchronizations from unavoidable calls to default stream (cudaStreamCreate, cudaFree, etc.)

3) Memory Transfer System

What it does
- Pipelines memory transfers to and from the device through pinned memory pools

Why it is necessary
- Automatically increases memory transfer bandwidth
- Allows for immediate kernel polling
- Avoids memory serializations by using pinned (page-locked) memory

4) Stream Manager

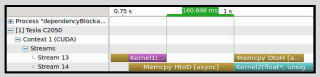
What it does
- Detects when a kernel has finished executing without causing an implicit synchronization

How it works
- Inserts code into Kernel which signals when all blocks have finished executing
- Polls this signal to determine when kernel has finished executing

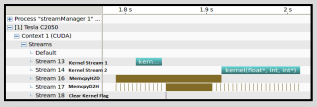
Why it is necessary
- Removes implicit synchronization
- Preserves data dependencies

Experimental Results

Stream Manager at Work



application experiencing a blockage due to an implicit synchronization



same application running without a blockage using the stream manager

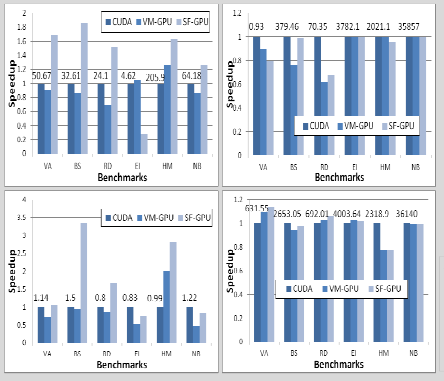
Hardware Setup

Attributes	Values
Type	12-core Intel Xeon E5@2.10GHz, 15MB Cache, 64GB
GPUs	1x Nvidia Tesla K20c (Kepler)
	1x Tesla C2050 (Fermi)
OS/CUDA	CentOS 6.4/CUDA 6.0

Test Cases

Applications	Description
CUDA SDK	
Vector Add (VA)	10M-element of vector addition
BlackScholes (BS)	Processing of 4M financial options
Eigen Value (EI)	Perform bisection algorithm
Reduction (RD)	Compute the sum of 16M elements
Developed Scientific Applications	
N-body (NB)	Simulation of interactions between 50K particles
Himeno (HM)	Stencil computation on 65x65x129 pressure grid

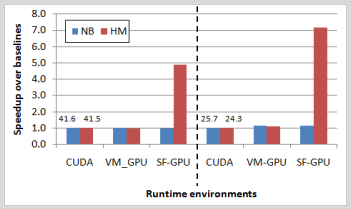
Single Application Performance



Single Application Speedup:
- TL: memory transfer
- TR: kernel execution
- BL: resource management
- BR: overall execution

Single application run in isolation

Multitenancy Performance



Multitenancy Speedup:
- Left: Tesla C2050 (Fermi)
- Right: Tesla K20 (Kepler)

NB and HM are run simultaneously

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.

Acknowledgements

This work has been supported by NSF award CNS-1216756 and by equipment donations from Nvidia Corporation.