CS 3580 - Advanced Topics in Parallel Computing

Analyzing CUDA Workloads Using a Detailed GPU Simulator

Mohammad Hasanzadeh Mofrad
University of Pittsburgh
November 14, 2017

Article information

Title: Analyzing CUDA workloads using a detailed GPU simulator

Authors: Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt

Affiliation: University of British Columbia (**UBC**), Vancouver, Canada

Conference: 2009 IEEE International Symposium on Performance Analysis of Systems and Software (**ISPASS 2009**)

Citation (as of November 2017): **1031**

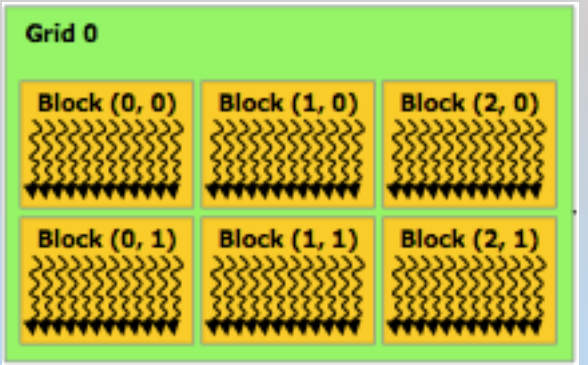
https://scholar.google.com/citations?user=-TSkCDMAAAAJ&hl=en&oi=sra

GPGPU-Sim in a nutshell

- Motivations
  - Presenting a novel microarchitecture performance simulator for NVIDIA GPUs (**GPGPU-Sim**[1]) characterizing following design decisions:
    - Interconnect topology of caches
    - Memory controller design
    - Parallel workload distribution
    - Memory request coalescing
- Outcomes
  - Performance is more sensitive to interconnect bisection bandwidth rather than latency
  - Running fewer number of threads might improve performance by reducing contention
- Our objective in this presentation is
  - Learn what GPGPU-Sim simulates and describe how it works

CUDA programming model

- Kernel launch = Grid of blocks of threads
- Kernel <<<# blocks, # threads>>>
  - `blockDim`, `threadIdx`, `blockIdx`
  - Example thread identifier x = `blockDim.x * blockIdx.x + threadIdx.x`
- Declarations
  - **`__global__`** runs on GPU, callable from CPU
  - **`__device__`** only callable from GPU
- Transfer data from CPU (host) to GPU (device)
  - **`malloc`** for CPUs, and **`cudaMalloc`** for GPUs
  - Copy input data from host to device
  - **`cudaMemcpy`** (Device destination address, Host source address, size, **`cudaMemcpyHostToDevice`**)
  - Copy the results back from device to host
  - **`cudaMemcpy`** (Host Destination address, Device source address, size, **`cudaMemcpyDeviceToHost`**)
  - **`Free`** for CPUs, and **`cudaFree`** for GPUs



https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/
GPGPU-Sim Tutorial (MICRO 2012)

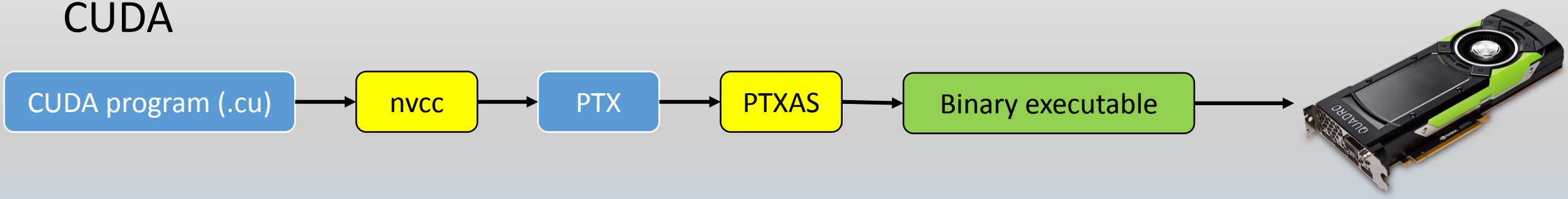
CUDA programming model (continue)

- In CUDA programming model, the **GPU is threated as a co-processor** onto which an application is running on a CPU can launch a massively parallel kernel.
- Revisiting the CUDA kernel
  - The CUDA **kernel** is compromised of a **grid** of threads
  - Within a **grid**, threads are groups into **thread blocks**
  - Within a **block** threads have access to common **fast memory** e.g. shared memory or L1 cache and can perform **barrier synchronization**
  - Each **thread** is given an **unique identifier** which can be used to help divide up work among the threads.

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

What GPGPU-Sim simulates

- GPGPU-Sim simulates Parallel Thread eXecution (PTX) Instruction Set Architecture (ISA) which is a pseudo-assembly language used in NVIDIA's CUDA

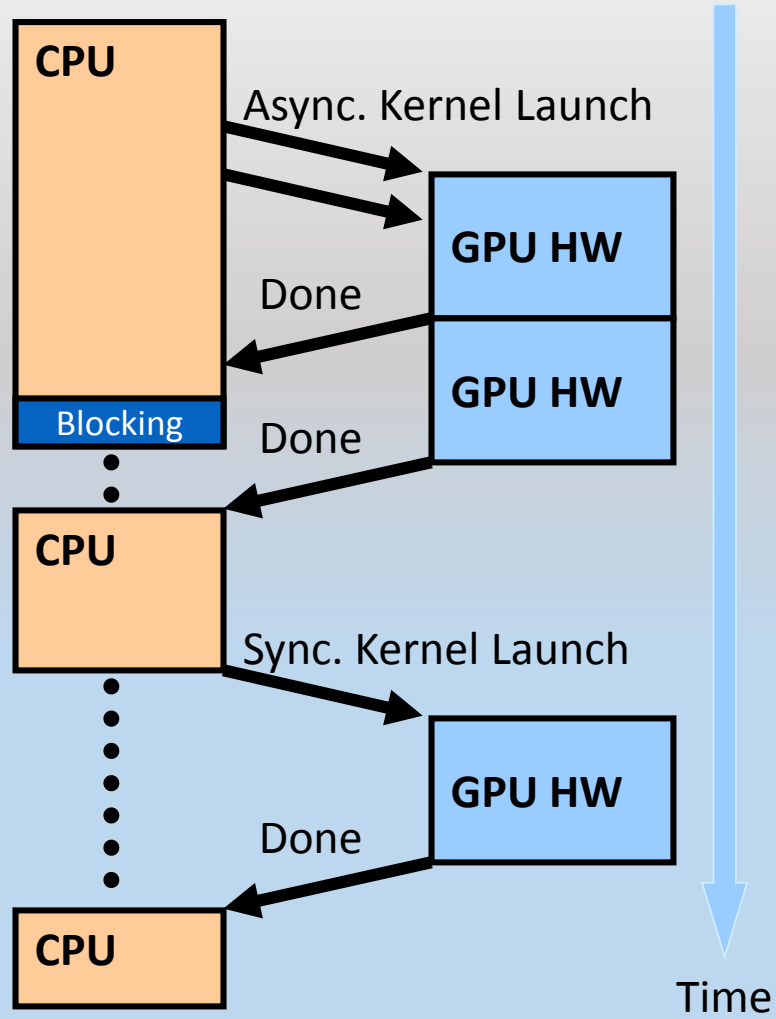


- GPGPU-Sim also simulates SASS the Native ISA for NVIDIA GPUs which is another kind of low-level assembly language that is native to NVIDIA GPU hardware
- GPGPU-Sim uses PTXPlus which can represent both PTX and SASS instructions.

https://en.wikipedia.org/wiki/Parallel_Thread_Execution
PTX ISA Application Guide Version 6.0 NVIDIA, 2017-09-xx
A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

GPGPU-Sim timing model

- GPGPU-Sim simulates timing for
  - CUDA cores
  - Caches
  - interconnection network
  - Memory
- but not for
  - CPU
  - PCIe
  - Graphics specific hardware like rasterizer
- For each kernel GPGPU-Sim
  - Counts the # cycles spent running the kernel
  - Excludes PCIe bus latency
- **CPU may run concurrently with asynchronous kernels**



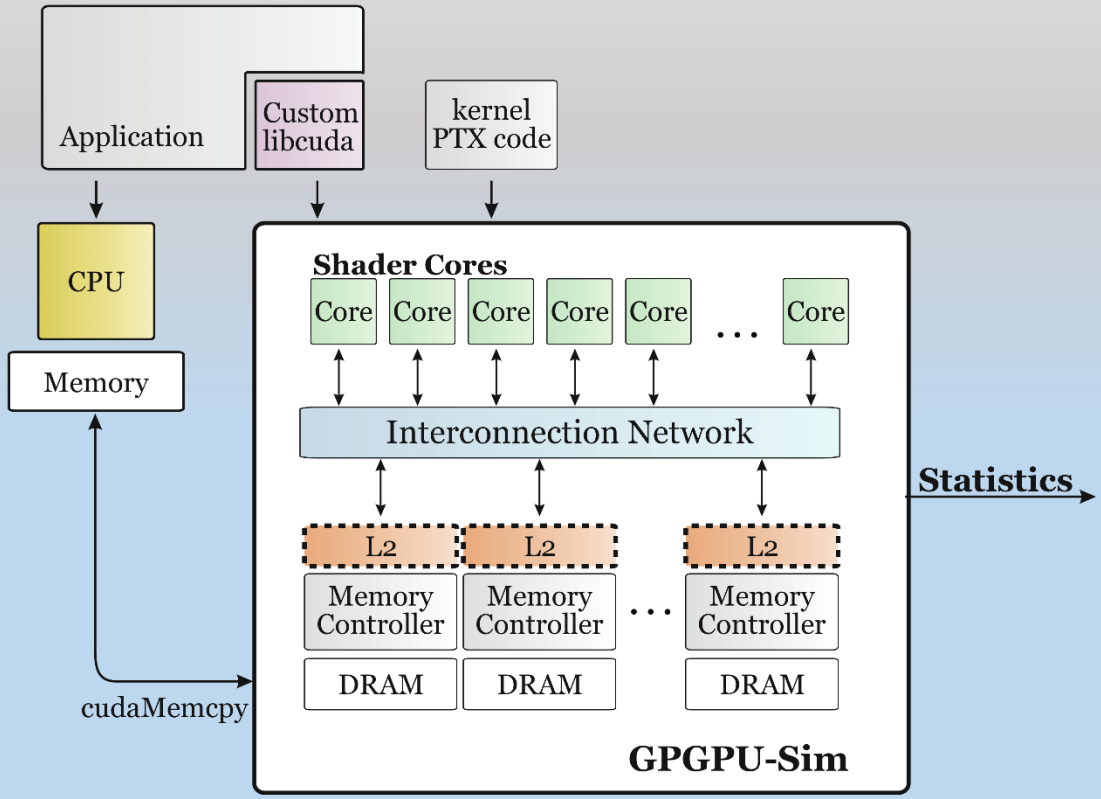
Comparing thread hierarchy and GPU computing power hierarchy

- A CUDA kernel is equal to
  - A grid of blocks of threads
  - Each thread block can contain up to 1024 threads
- A Maxwell GPU microarchitecture consist of
  - Four clusters of 16 Streaming Multiprocessors (SMs)
  - Each SM is partitioned into four 32-CUDA cores (Streaming Processors (SPs)) processing blocks each with one warp scheduler[1]
  - For example, for an NVIDIA GeForce GTX 980
    - The warp size is 32 which is aligned to the number of CUDA cores
    - Each SM has 4 * 32 = 128 cores
    - Each SM can launch 128 threads in parallel
- **In CUDA programming language, each thread block is dispatched on a SM as a unit of work. Then, SM warp schedulers schedule and run partitions of 32 threads on CUDA cores.**

1. “Nvidia geforce gtx 980: Featuring maxwell, the most advanced gpu ever made,” White paper, NVIDIA Corporation, 2014

GPGPU-Sim microarchitecture

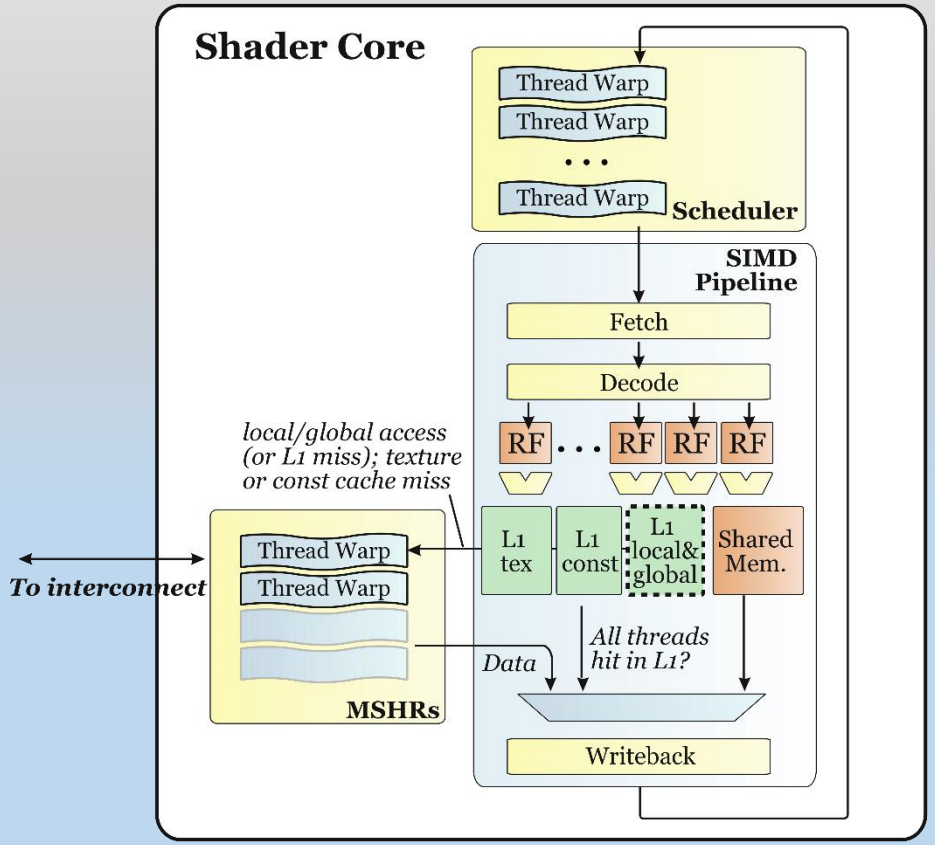
- A set of **shader cores** connected to a set of memory controllers via an interconnection network
- Each shader core represents a **CUDA core (SM)**
- Threads are distributed to shader cores at the granularity of thread blocks
- Registers, shared memory space, and thread slots are shared per thread block
- **Multiple thread blocks can be assigned to a single CUDA core, sharing a common pipeline**



A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

GPGPU-Sim microarchitecture (continue)

- A shader core has a *SIMD width of 8* and uses a *24-stage in-order pipeline* without forwarding
- Thus, at least 192 (8 * 24) active threads are needed to avoid stalling
- A shader core has 6 logical pipeline stages:
  - Fetch, decode, execute, memory1, memory2, writeback
- A set of 32 threads constitutes a warp
- All 32 threads in a given warp execute the same instruction with different data values over 4 consecutive clock cycles in all pipelines
- Immediate post-dominator reconvergence mechanism is used to handle branch divergence

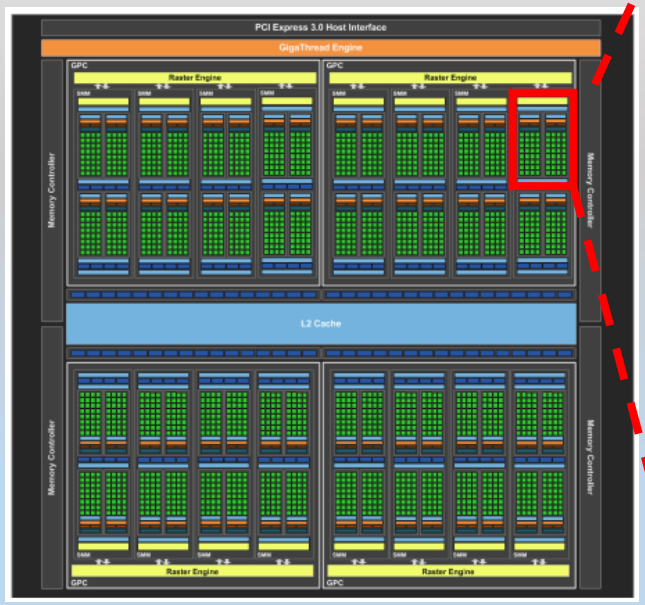


A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

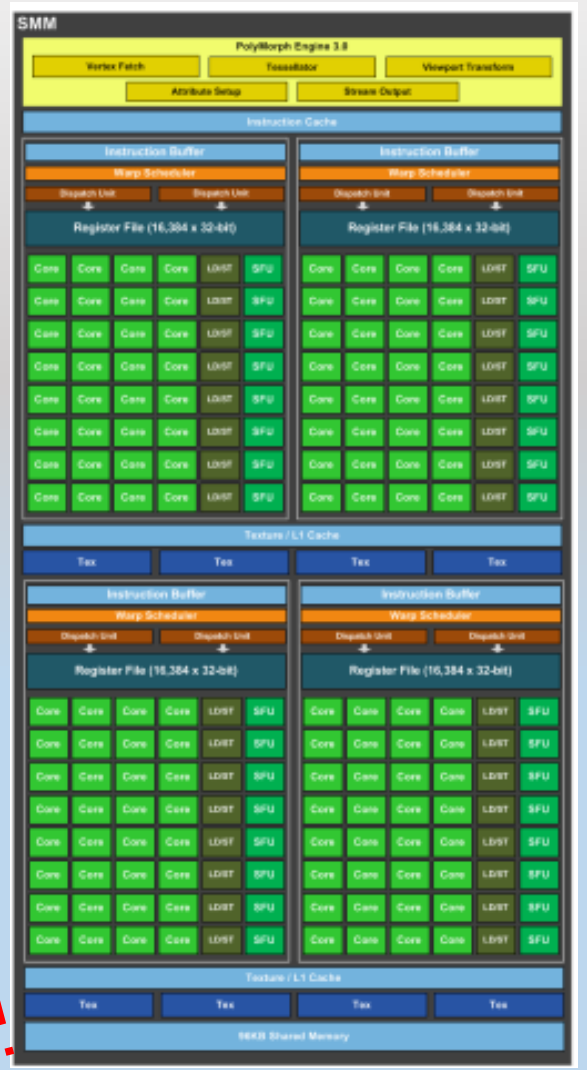
Revisiting NVIDIA GeForce GTX 980 (Maxwell) Streaming Multiprocessor (SM)

- 4 Graphics Processing Cluster (GPC)[1]
- 16 Maxwell SM per GPC
- 4 32-CUDA cores processing blocks per SM
  - 128 CUDA cores per SM
  - 2048 CUDA cores in total
- 4 warp schedulers per SM each capable of launching two instructions per cycle.
- The warp size is 32 equal to #cores per processing block
- In Maxwell microachitecture SMs both have 4 warp schedulers. On each cycle each warp scheduler picks an eligible warp (aka a warp that is not stalled) and issue 1 or 2 independent instructions from the warp.[2]

4 GPC of GTX 980



A single SM



1. “Nvidia geforce gtx 980: Featuring maxwell, the most advanced gpu ever made,” White paper, NVIDIA Corporation, 2014

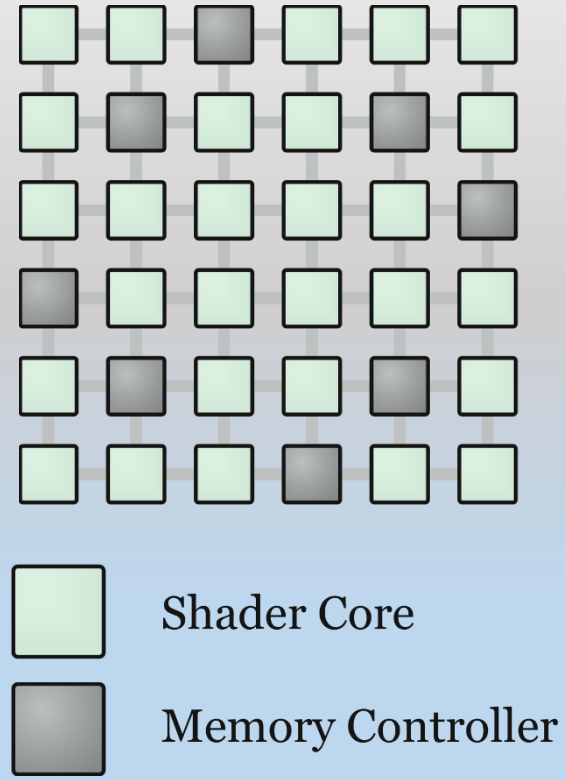
2 https://devtalk.nvidia.com/default/topic/743377/cuda-programming-and-performance/understanding-cuda-scheduling/post/4221610/

GPGPU-Sim memory unit

- **Constant cache**: A read-only cache for constant memory
  - A warp can access one constant cache location in a single memory unit cycle
- **Texture cache**: A read-only cache with FIFO retirement
- **Shared memory**: A 16 KB low latency highly-banked per-core
  - Threads within a block can cooperate via shared memory
  - As fast as register files in absence of bank conflicts
  - Each bank serves one address per cycle. Multiple accesses to a bank in a single cycle causes a bank conflict
- **Global memory**: Off-chip DRAM memory. Accesses must go through interconnect
  - Global texture memory with a per-core texture cache
  - Global constant memory with a per-core constant cache

GPGPU-Sim interconnection network

- The on-chip interconnection network can be designed in various ways based on cost and performance.
  - **Cost** is determined by complexity and number of routers as well as density and length of wires
  - **Performance** depends on latency, bandwidth, and path diversity of the network.
- In order to access the global memory, memory requests must be sent via an interconnection network to a corresponding memory controller which are physically distributed over the chip.
- The memory controllers are laid out in a **6x6 mesh configuration**



A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.

GPGPU-Sim thread scheduling

- Thread scheduling is done inside the shader core.
- Every **4 cycles**, warps ready for execution are selected by the warp scheduler and scheduled in a **round robin** fashion.
- If a thread inside a warp faces a **long latency operation**, all threads in the warp are taken out of the scheduling pool until the long latency operation is over.
- The ready warps are sent to the pipeline for execution in a round robin order.
- Many threads running on each shader core thus allow a shader core to tolerate long latency operations without reducing throughput.

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

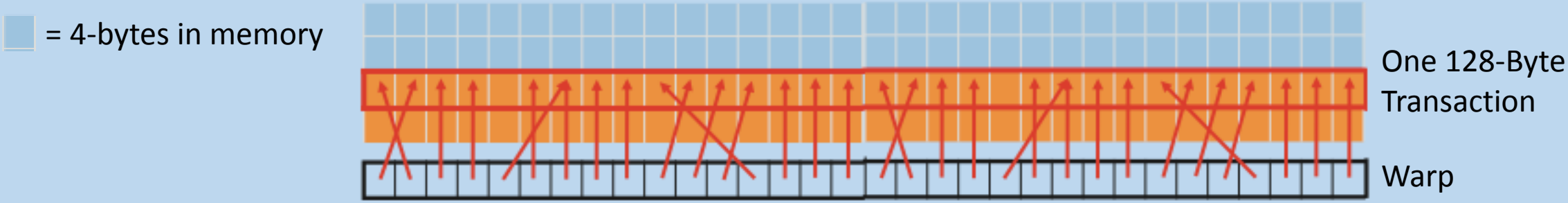
GPGPU-Sim thread block distribution

- **Interleaving the execution of warps** allows GPUs to tolerate memory access latencies.
- **Running multiple smaller thread blocks on a shader core is better than using a single larger thread block:**
  1. Threads from one thread block can make progress while threads from another thread block are waiting at a **barrier synchronization** point.
  2. Larger thread blocks require **more register** and increase shared memory usage.
  3. Running more threads blocks on a shader core provides **additional memory latency** tolerance.
- However, If a memory-intensive kernel completely fills up all thread block slots, it may increase contention in the interconnection network.
- **Breadth-first strategy** is used to distribute thread blocks among shader cores, selecting a core with the minimum number of running thread blocks.

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

GPGPU-Sim memory access coalescing (intra-warp)

- Objective: Combining memory accesses made by threads in a warp into fewer transactions
- Example: If threads in a warp are accessing consecutive 4-byte sized locations in memory
  - Send one 128–byte request to DRAM (coalescing)
  - Instead of 32 4-byte requests
- This reduces the number of transactions between CUDA cores and DRAM
  - Less work for interconnect, memory partition, and DRAM.



GPGPU-Sim read memory access coalescing (inter-warp)

- Cache accesses are tracked using the **Miss Status Holding Registers (MSHR)**
- MSHRs keep track of outstanding memory requests, merges simultaneous requests for the same cache block
- In GPGPU-Sim each cache has its own set of MSHR entries
  - Each MSHR entry contains one or more request to the same memory address
  - The number of MSHR entries are configurable
  - Memory unit stalls if cache runs out of MSHR entries
- The inter-warp memory coalescing consolidates read memory requests from later warps that require access to data for which a memory request is already in progress due to another warp running on the same shader core.

A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.

GPGPU-Sim caching

- While coalescing memory requests captures **spatial locality** among threads, memory bandwidth requirements may be further reduced with caching if an application contains **temporal locality** or spatial locality within the access pattern of individual threads.

A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.

Software dependencies

- Linux machine (Preferably Ubuntu 11.04 64 bit)
- CUDA toolkit 4.2, and NVIDIA GPU Computing SDK 4.2
- gcc 4.{4,5}.x, g++ 4.{4,5}.x, and gfortran 4.{4,5}.x
- Development tools (especially build-essential package), OpenMPI, Boost library, and etc.
- Can be cloned from:
  - git clone git://dev.ece.ubc.ca/gpgpu-sim
  - Git clone https://github.com/gpgpu-sim/gpgpu-sim_distribution

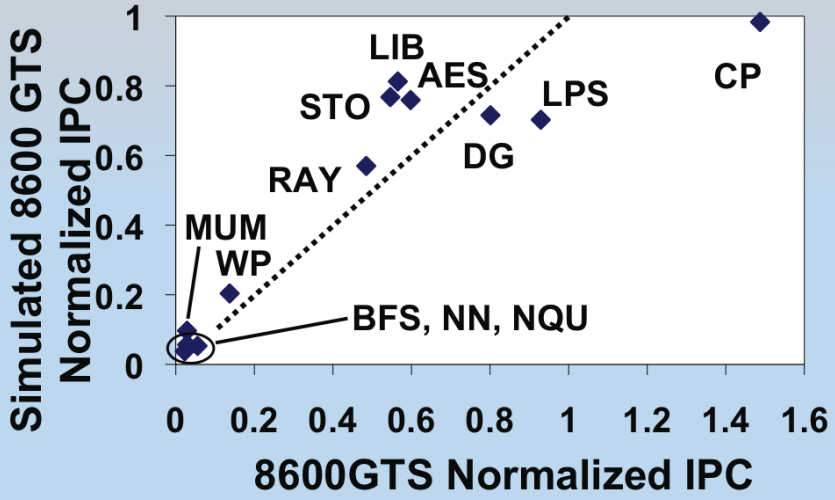
Simulation setup for baseline GPU configuration

Number of shader cores	28
Warp Size	32
SIMD pipeline width	8
# of Threads/CTAs/Registers per Core	1024 / 8 /16384
Shared Memory / Core	16KB (16 banks)
Constant Cache / Core	8KB (2-way set assoc. 64B lines LRU)
Texture Cache / Core	64KB (2-way set assoc. 64B lines LRU)
Memory Channels	8
BW / Memory Module	8 Byte/Cycle
DRAM request queue size	32
Memory Controller	Out of order (FR-FCFS)
Branch Divergence handling method	Immediate Post Dominator
Warp Scheduling Policy	Round Robin among ready Warps

A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.

Benchmarks

- GPGPU-Sim is a academic **cycle-level** simulator for modeling a modern GPU running **non-graphics** workloads
- **14 benchmarks** written in CUDA. Needed software tuning or changes in hardware design
- Less than 50x reported speedups



Advanced Encryption Standard (AES)	Breadth First Search (BFS)	Coulombic Potential (CP)	gpuDG (DG)	3D Laplace Solver (LPS)
LIBOR Monte Carlo (LIB)	MUMmerGPU (MUM)	Neural Network Digit Recognition (NN)	N-Queens Solver (NQU)	Ray Tracing (RAY)
StoreGPU (STO)	Weather Prediction (WP)	Black-Scholes option pricing (BLK)	Fast Walsh Transform (FWT)	

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

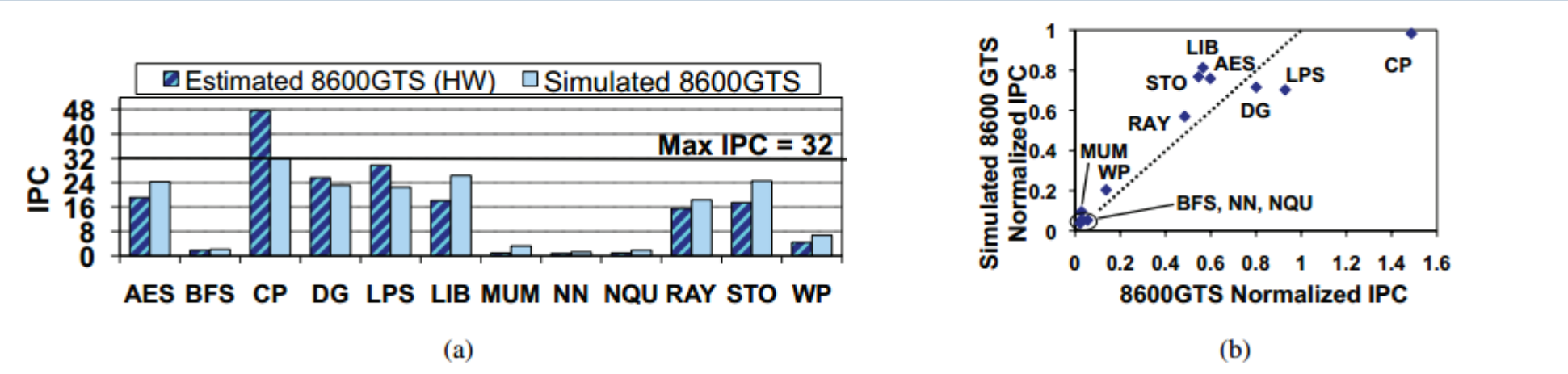
Benchmarks (continue)

Benchmark	Abbr.	Claimed Speedup
AES Cryptography	AES	12x
Breadth First Search	BFS	2x-3x
Coulombic Potential	CP	647x
gpuDG	DG	50x
3D Laplace Solver	LPS	50x
LIBOR Monte Carlo	LIB	50x
MUMmerGPU	MUM	3.5x-10x
Neural Network	NN	10x
N-Queens Solver	NQU	2.5x
Ray Tracing	RAY	16x
StoreGPU	STO	9x
Weather Prediction	WP	20x

A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.

Comparison between GPGPU-Sim and NVIDIA GeForce 8600 GTS

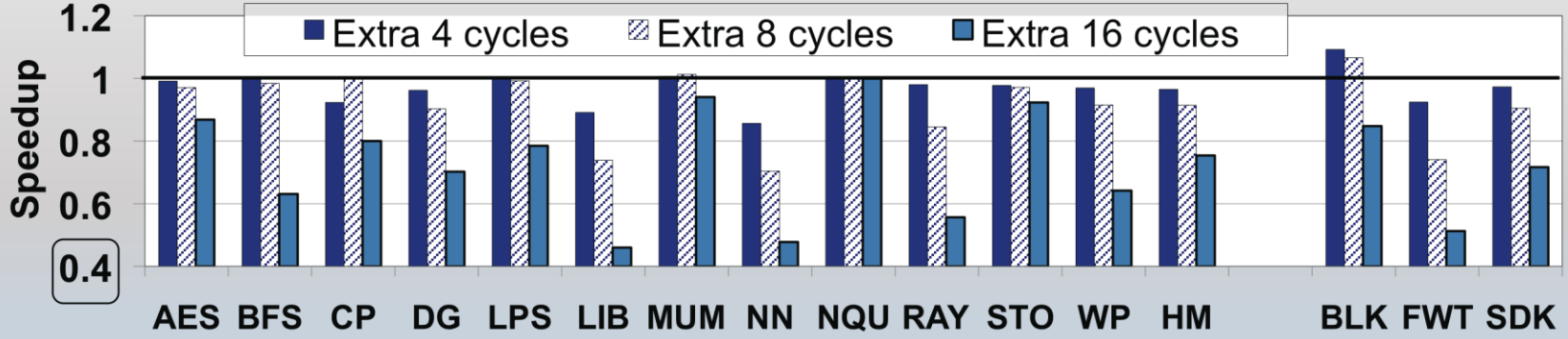
- Maximum GPGU-Sim IPC = 224 (28 cores * 8-wide pipeline)
- NVIDIA GeForce 8600 GTS IPC =
  - # PTX instructions / (Runtime * Core frequency)
- 0.899 Correlation coefficient



A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

Interconnection network latency sensitivity

- Results for various mesh network latencies, adding extra pipeline latency of 4, 8, or 16 cycles

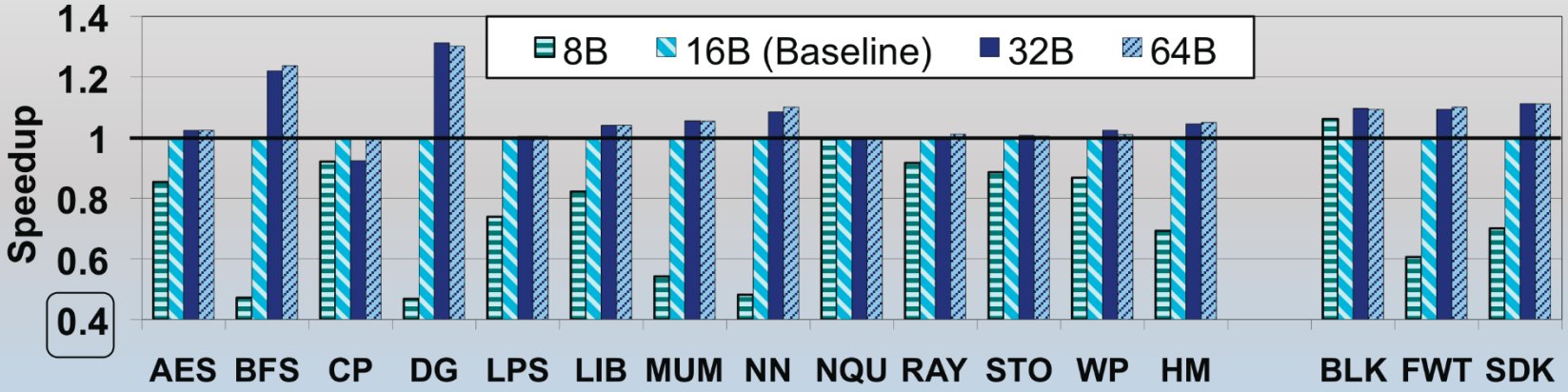


- Slight increase in **interconnection latency has no severe effect of overall performance**
  - No need to overdesign interconnection to decrease latency

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

Interconnection network bandwidth sensitivity

- Results for modifying mesh interconnect bandwidth by varying the channel bandwidth from 8 bytes to 64 bytes.

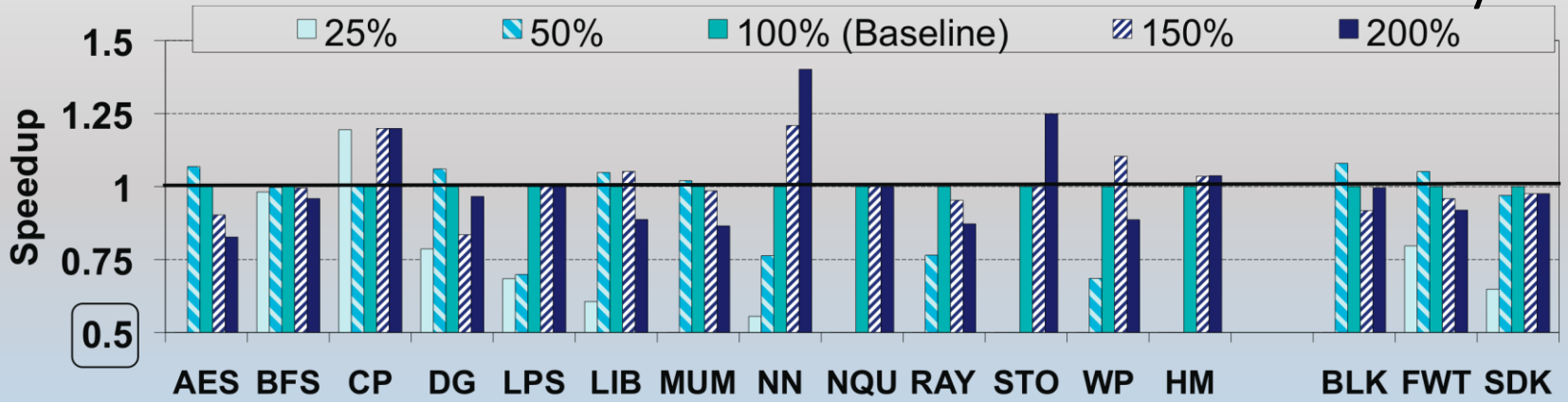


- Low Bandwidth decreases performance a lot (8B)
- Overall, **performance is more sensitive to interconnect bandwidth** than to latency
- In other words, restricting the channel bandwidth causes the interconnect to become a bottleneck

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

Effects of varying number of thread blocks

- Results for exploring the effects of varying the resources that limit the number of threads including: the amount of registers, shared memory and threads.
- These limitations limit the number of thread blocks can run concurrently on a core.



- Most benchmarks do not benefit substantially
- **Some benchmarks even perform better with fewer concurrent threads** (e.g. AES)
  - Less contention in DRAM
- Given the widely-varying workload-dependent behavior, always scheduling the maximal number of thread blocks supported by a core is not always the best scheduling policy.

A. Bakhoda, et al. "Analyzing CUDA workloads using a detailed GPU simulator." IEEE ISPASS 2009.

Summary

- GPGPU-Sim is a GPU simulator capable of simulating CUDA workloads.
- Various aspects of the GPGPU-Sim are studied including:
  - Branch divergence
  - Interconnect topology
  - Interconnect latency and bandwidth
  - DRAM utilization and efficiency
  - Cache effect
  - Thread block size
  - Memory request coalescing
- Running fewer number of thread blocks can improve performance (less DRAM contention)

A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.

Key references

- A. Bakhoda, et al. “Analyzing CUDA workloads using a detailed GPU simulator.“ IEEE ISPASS 2009.
- http://www.gpgpu-sim.org/micro2012-tutorial/