A Characterization and Analysis of PTX Kernels

Andrew Kerr*, Gregory Diamos, and Sudhakar Yalamanchili

School of Electrical and Computer Engineering
Georgia Institute of Technology

October 5, 2009

IEEE International Symposium on Workload Characterization 2009

Introduction

- Workload Characterization Goals
- NVIDIA's Parallel Thread Execution (PTX) ISA
  - CUDA Programming Language
- Ocelot Infrastructure
- Application Workloads
- Metrics and Workload Characteristics
- Summary

Workload Characterization Goals

Understand

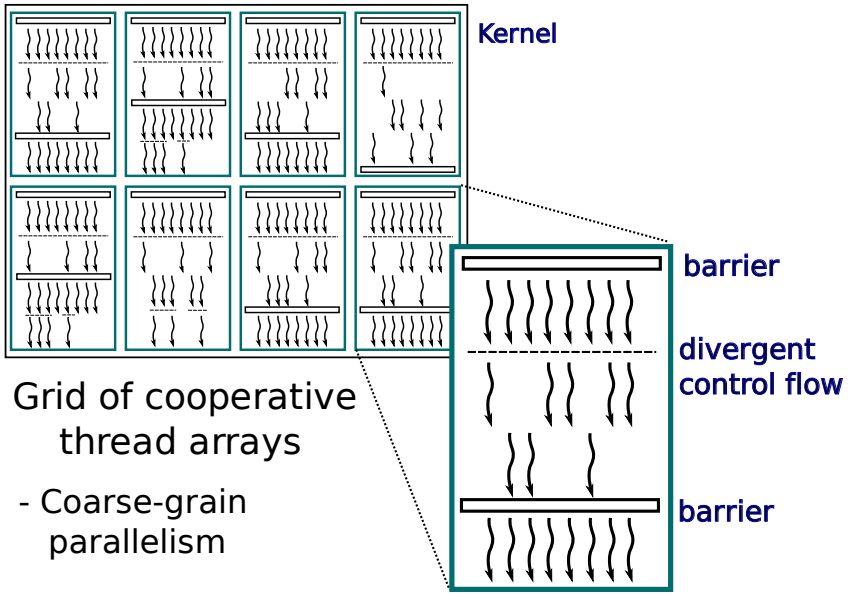
- Control flow behavior of SIMD kernels
- Memory demand
- Available parallelism within and across SIMD kernels

To provide insights for

- Compiler optimizations
- Application restructuring
- Architectural optimizations
- Dynamic optimizations

Parallel Thread Execution (PTX) Model

PTX Thread Hierarchy



Grid of cooperative thread arrays

- Coarse-grain parallelism

Cooperative Thread Array

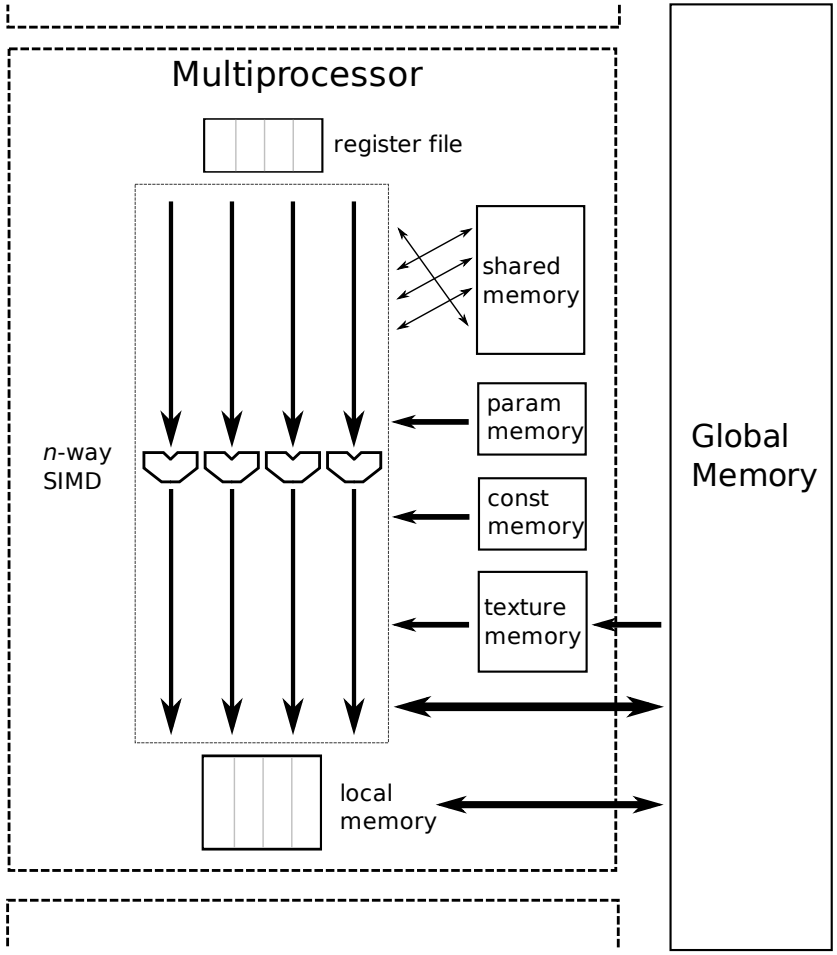
- Fine-grain parallelism

PTX Virtual ISA

- RISC Instruction Set

- Defined by NVIDIA - target of CUDA compiler

Multiprocessor Architecture



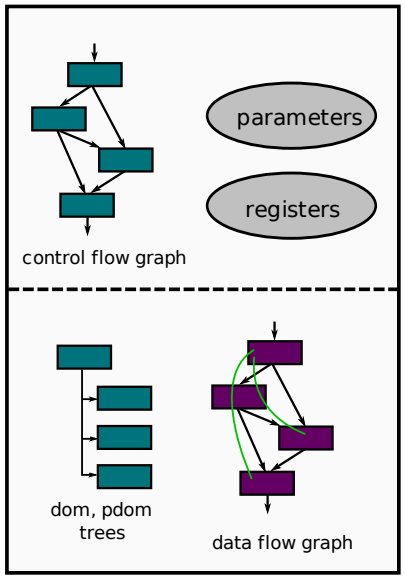
Ocelot

PTX Kernel

```
L_BB_1:  add.s64 %rd2, %rd1, 1
         mul.s64 %rd3, %rd2, 4
         mov.s64 %rd4, 256
         setp.lt.s64 %p1, %rd3, %rd4
         @%p1 bra L_BB_3
L_BB_2:  abs.f64 %fd1, %fd1
         mov.s64 %rd5, 64
         setp.lt.s64 %p2, %rd3, %rd5
         @%p2 bra L_BB_4
L_BB_3:  sin.f64 %fd2, %fd1
         st.f64 %fd2, [%rd0 + 4]
L_BB_4:  reconverge L_BB_2
         reconverge L_BB_1
L_BB_5:  exit
```

Ocelot - PTX Translator

Kernel Internal Representation



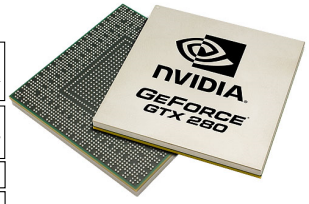
PTX Emulation

```
L_BB_1:  add.s64 %rd2, %rd1, 1
         mul.s64 %rd3, %rd2, 4
         mov.s64 %rd4, 256
         setp.lt.s64 %p1, %rd3, %rd4
         @%p1 bra L_BB_3
L_BB_2:  abs.f64 %fd1, %fd1
         mov.s64 %rd5, 64
         setp.lt.s64 %p2, %rd3, %rd5
         @%p2 bra L_BB_4
L_BB_3:  sin.f64 %fd2, %fd1
         st.f64 %fd2, [%rd0 + 4]
L_BB_4:  reconverge L_BB_2
         reconverge L_BB_1
L_BB_5:  exit
```

x86

GPU

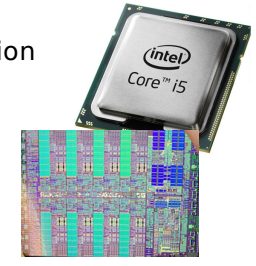
```
L_BB_1:  add.s64 %rd2, %rd1, 1
         mul.s64 %rd3, %rd2, 4
         mov.s64 %rd4, 256
         setp.lt.s64 %p1, %rd3, %rd4
         @%p1 bra L_BB_3
L_BB_2:  abs.f64 %fd1, %fd1
         mov.s64 %rd5, 64
         setp.lt.s64 %p2, %rd3, %rd5
         @%p2 bra L_BB_4
L_BB_3:  sin.f64 %fd2, %fd1
         st.f64 %fd2, [%rd0 + 4]
L_BB_4:  reconverge L_BB_2
         reconverge L_BB_1
L_BB_5:  exit
```



NVIDIA GPU

LLVM Translation

```
L_BB_1:  add.s64 %rd2, %rd1, 1
         mul.s64 %rd3, %rd2, 4
         mov.s64 %rd4, 256
         setp.lt.s64 %p1, %rd3, %rd4
         @%p1 bra L_BB_3
L_BB_2:  abs.f64 %fd1, %fd1
         mov.s64 %rd5, 64
         setp.lt.s64 %p2, %rd3, %rd5
         @%p2 bra L_BB_4
L_BB_3:  sin.f64 %fd2, %fd1
         st.f64 %fd2, [%rd0 + 4]
L_BB_4:  reconverge L_BB_2
         reconverge L_BB_1
L_BB_5:  exit
```



x86 Multicore, Cell, OpenCL

PTX 1.4 compliant

Google Code project:
GPU Ocelot

CUDA SDK: Basic Characteristics

Applications	Kernels	CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bicubic Texture	27	256	1024	222,208	5120	3
Binomial Options	1	256	4	725,280	68,160	8
Black-Scholes Options	1	128	480	3,735,550	94230	4
Box Filter	3	32	16	1,273,808	17,568	4
DCT	9	70.01	2,446	1,898,752	25,600	3
Haar wavelets	2	479.99	2.5	1,912	84	5
DXT Compression	1	64	64	673,676	28,800	8
Eigen Values	3	256	4.33	9,163,154	834,084	13
Fast Walsh Transform	11	389.94	36.8	32,752	1216	4
Fluids	4	36.79	32.6	151,654	3,380	5
Image Denoising	8	64	25	4,632,200	149,400	6
Mandelbrot	2	256	40	6,136,566	614,210	26
Mersenne twister	2	128	32	1,552,704	47,072	7
Monte Carlo Options	2	243.54	96	1,173,898	76,512	8
Threaded Monte Carlo	4	243.54	96	1,173,898	76,512	8
Nbody	1	256	4	82,784	1,064	5
Ocean	4	64	488.25	390,786	17,061	7
Particles	16	86.79	29.75	277,234	26,832	16
Quasirandom	2	278.11	128	3,219,609	391,637	8
Recursive Gaussian	2	78.18	516	3,436,672	41,088	8
Sobel Filter	12	153.68	426.66	2,157,884	101,140	6
Volume Render	1	256	1,024	2,874,424	139,061	5

Table: CUDA SDK Application Statistics

Applications: Basic Characteristics

Benchmarks	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
CP	10	128	256	430,261,760	10,245,120	3
MRI-FHD	7	256	110.571	9,272,268	198,150	5
MRI-Q	4	256	97.5	7,289,604	393,990	5
PNS	112	256	17.85	683,056,349	33,253,961	11
RPES	71	64	64,768.7	1,395,694,886	95,217,761	13
SAD	3	61.42	594	4,690,521	87,813	7
TPACF	1	256	201	1,582,900,869	230,942,677	18

Table: Parboil Application Statistics

Workloads	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
SDK	145	217.64	457.25	55,884,066	3,504,904	26
RIAA	10	64	16	322,952,484	23,413,125	16
RDM	2237	174.558	63.0595	46,448,530	4,082,425	6
Parboil	208	177.238	9,435.09	4,113,166,257	370,339,472	11

Table: Aggregate Workload Statistics

Metrics

Control flow

- Branch Divergence
- Activity Factor

Global memory and data flow

- Memory Intensity
- Memory Efficiency
- Interthread Data Flow

Parallelism

- MIMD Parallelism
- SIMD Parallelism

Analysis Methodology

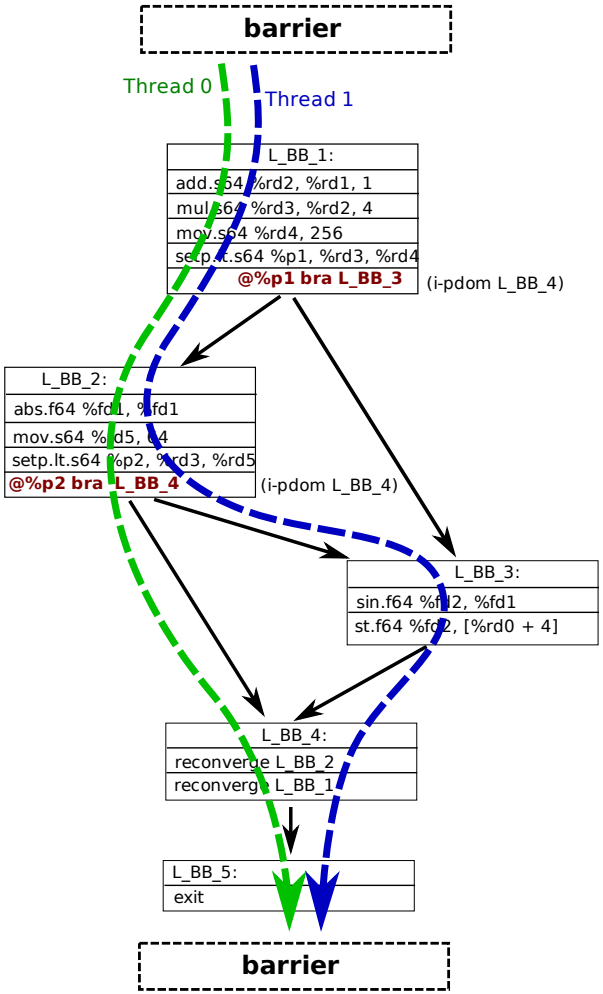
Ocelot serializes execution of CTAs

- Each instruction executed for active threads
- Warp size is equal to CTA size
- Divergent control flow splits active context

Metrics averaged over all dynamic instructions for all kernels in an application

- PC
- Activity mask
- Memory references

Branch Divergence



Divergent?

no

yes

Fraction of branches that are divergent

Branch Divergence

$$BD = \frac{\#divergent\ branches}{\#branches}$$

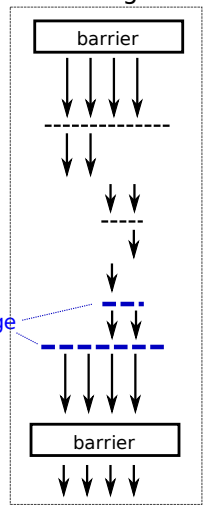
Computed on dynamic instruction stream

Post Dominator versus Barrier Reconvergence

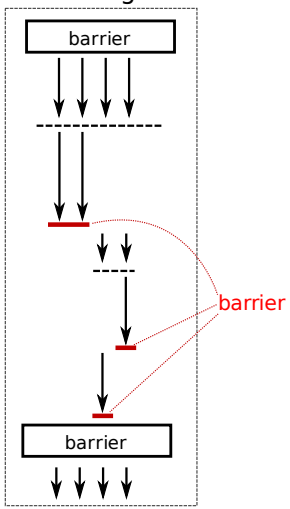
Pseudocode

```
barrier;

s0;

if ( cond_0 ) {
    s1;
    if ( cond_1 ) {
        s2;
    }
    else {
        s3;
    }
    s4;
}
else {
    s5;
}

s6;
barrier;
s7;
```

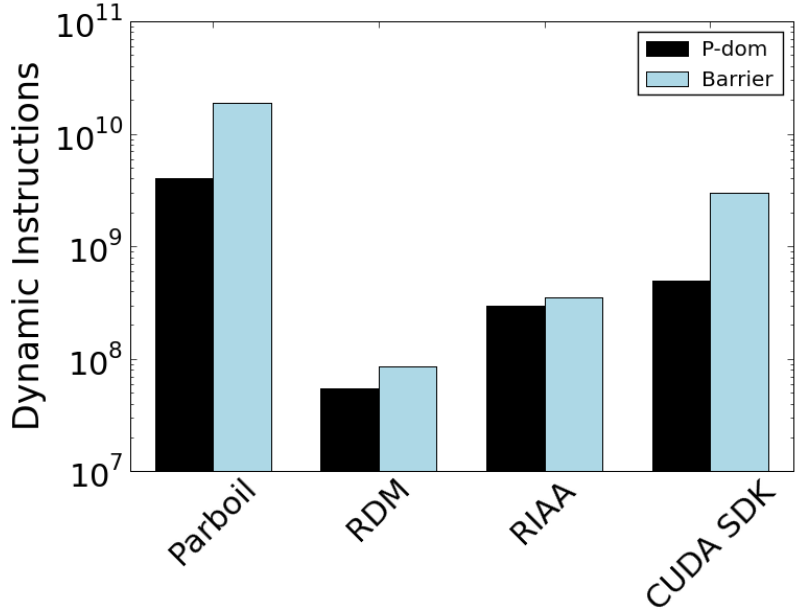
Post-dominator Reconvergence [1]

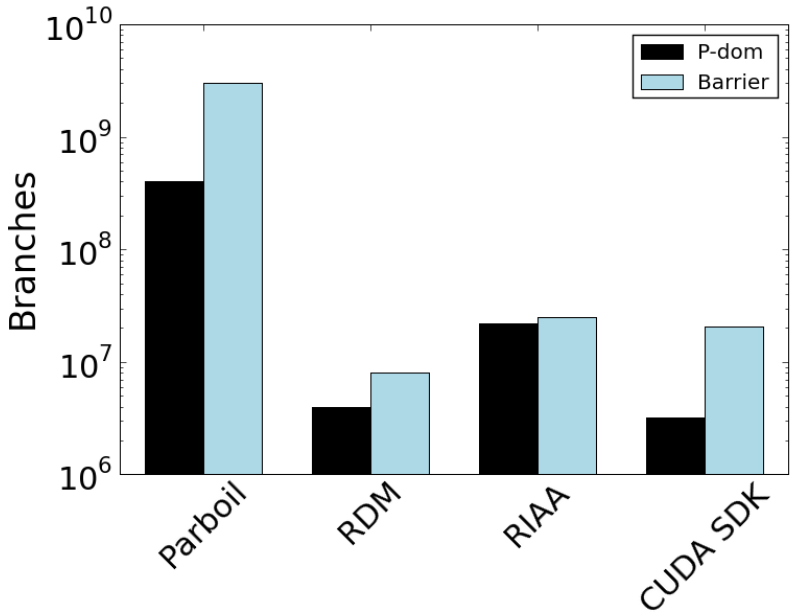


Barrier Reconvergence

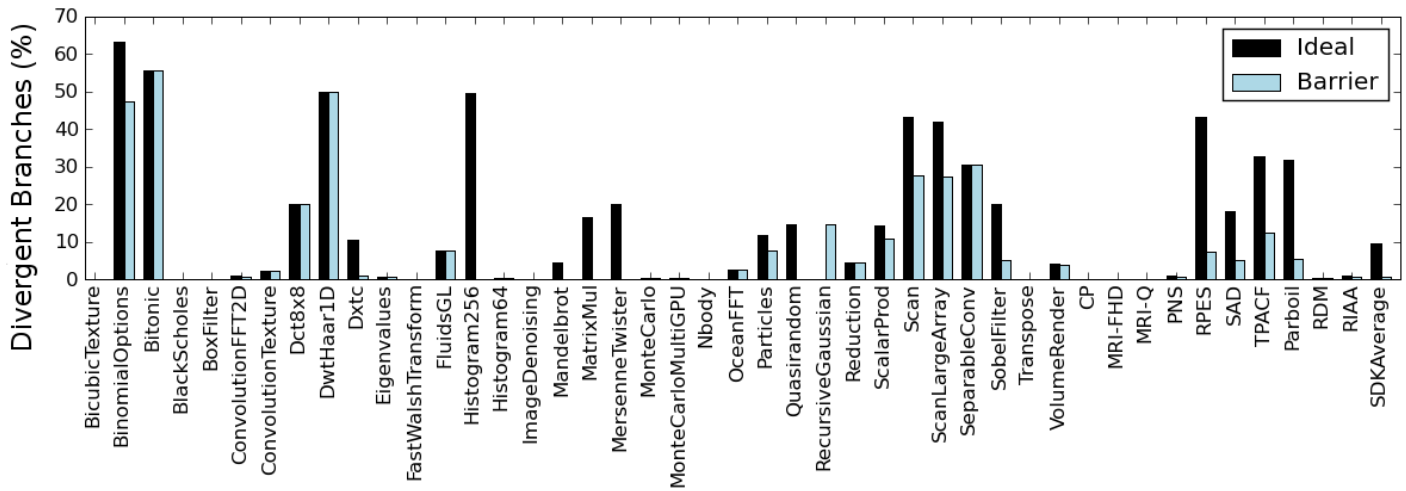


[1] Fung, et al "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow" *IEEE Micro 2007*



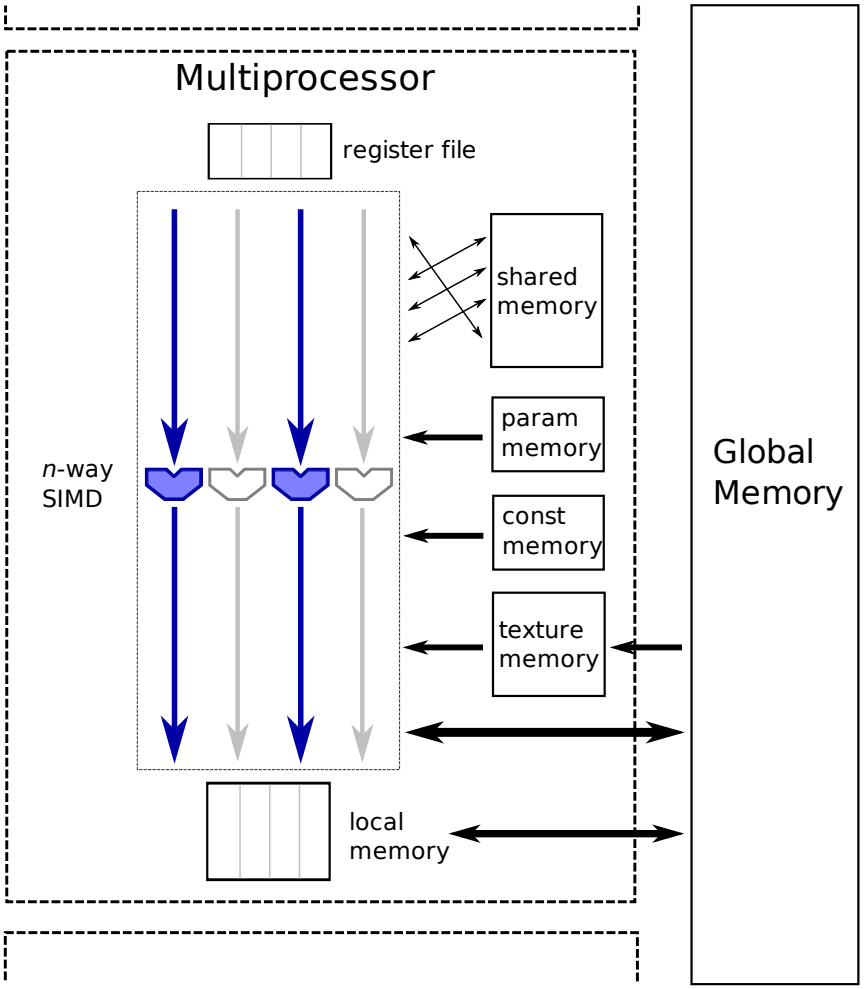


Branch Divergence Results



- Branches correlated (in time within the same thread) result in differences in ideal-vs-barrier reconvergence
- Frequent handling of special cases results in high overall divergent control flow
- Recommendation:
  - Correlation of branches suggests restructuring of threads to reduce divergence
  - If warp split costs are high, use barrier synchronization reconvergence method

Activity Factor



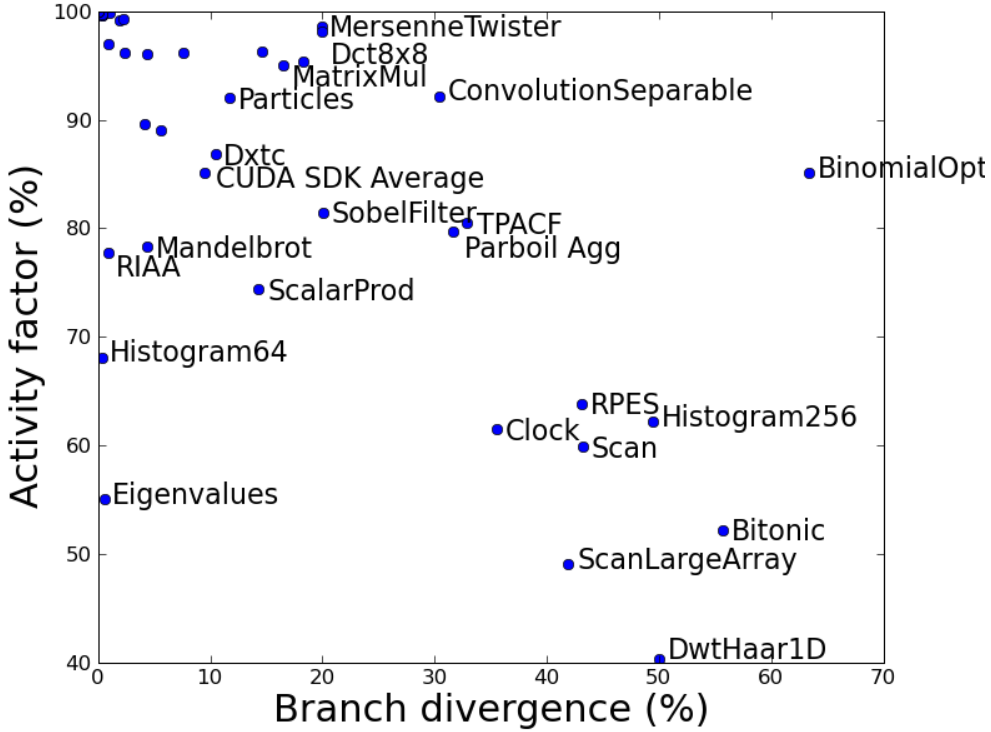
Average number of active SIMD ways

Activity Factor

$$AF = \frac{1}{N} \sum_{i=1}^{N} \frac{active(i)}{CTA(i)}$$

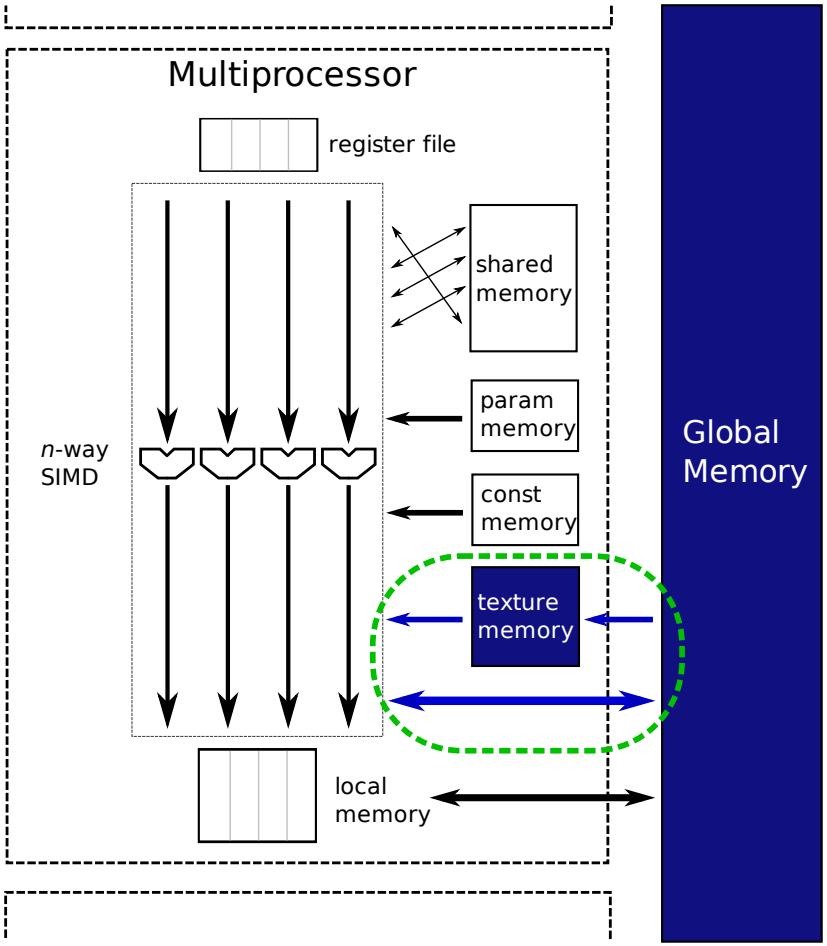
$active(i)$: active threads executing dyn. instruction i
$CTA(i)$: threads in CTA executing i
N: number of dynamic instructions

Activity Factor Results



- Recommendation:
  - Compiler use of predication to reduce control flow for short divergent paths
  - Placement of bar.sync earlier to increase AF
  - Hardware support for p-dom reconvergence

Memory Intensity



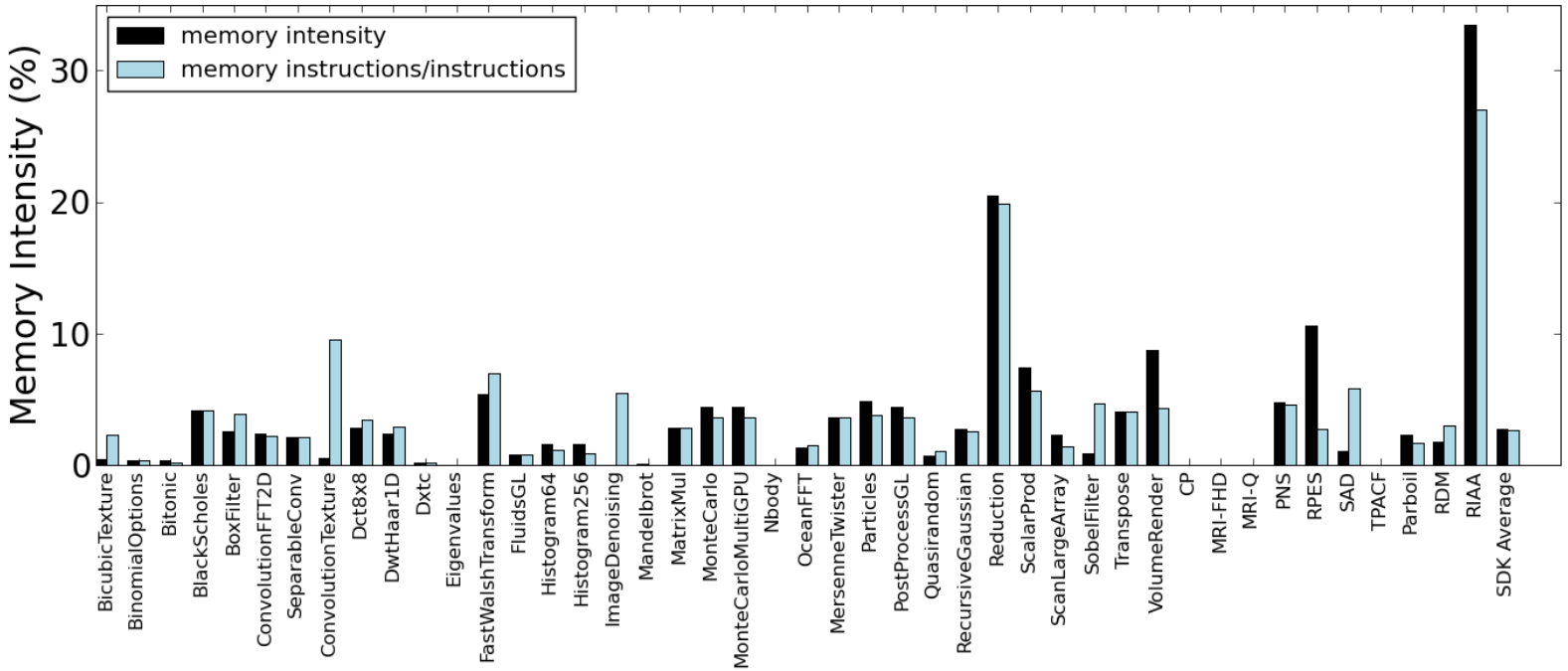
Fraction of loads or stores to global memory per dynamic instruction

Memory Intensity

$$I_M = \times \frac{\sum_{i=1}^{kernels} A_f M_i}{\sum_{i=1}^{kernels} D_i}$$

A_f: activity factor
M_i: global memory instructions
D_i: dynamic instructions
Texture samples counted as global memory accesses

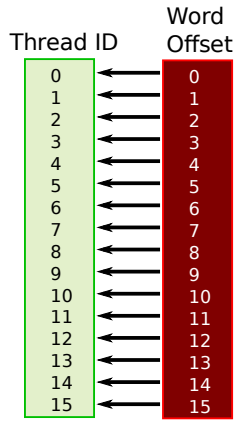
Memory Intensity Results



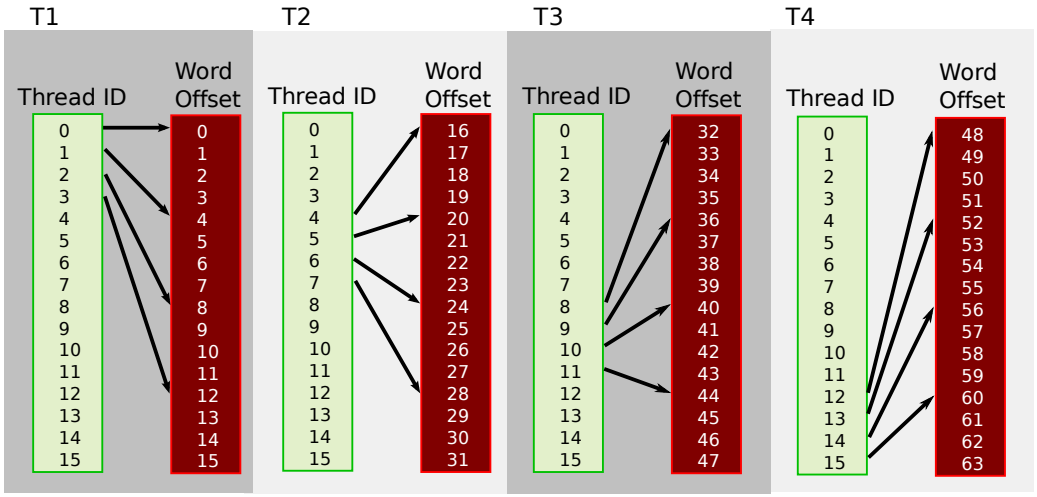
- CUDA SDK, RDM, Parboil have low average memory intensities (3.5%)
  - Efficient applications strive to be compute bound
  - Statistic ignores shared and local memory operations
  - Memory intensity not same as bandwidth
- RIAA application has relatively high memory intensity
  - Consequence of application: large hash table, pointer chasing

Memory Efficiency

```
// CUDA - gather-scatter        // PTX - gather-scatter

a = A[threadIdx.x];             mov.u16 %r0, %tidx
                                add.u64 %rd1, %r0, %rd0
                                ld.global.f32 %f0, [%rd1+0]

__syncthreads();                bar.sync 0

A[4 * threadIdx.x] = a;         mul.u32.lo %r1, %r0, 4
                                add.u64 %rd2, %r1, %rd0
                                st.global.f32 [%rd2+0], %f0
```



Coalesced gather - 1 transaction



Uncoalesced scatter - 4 serialized transactions

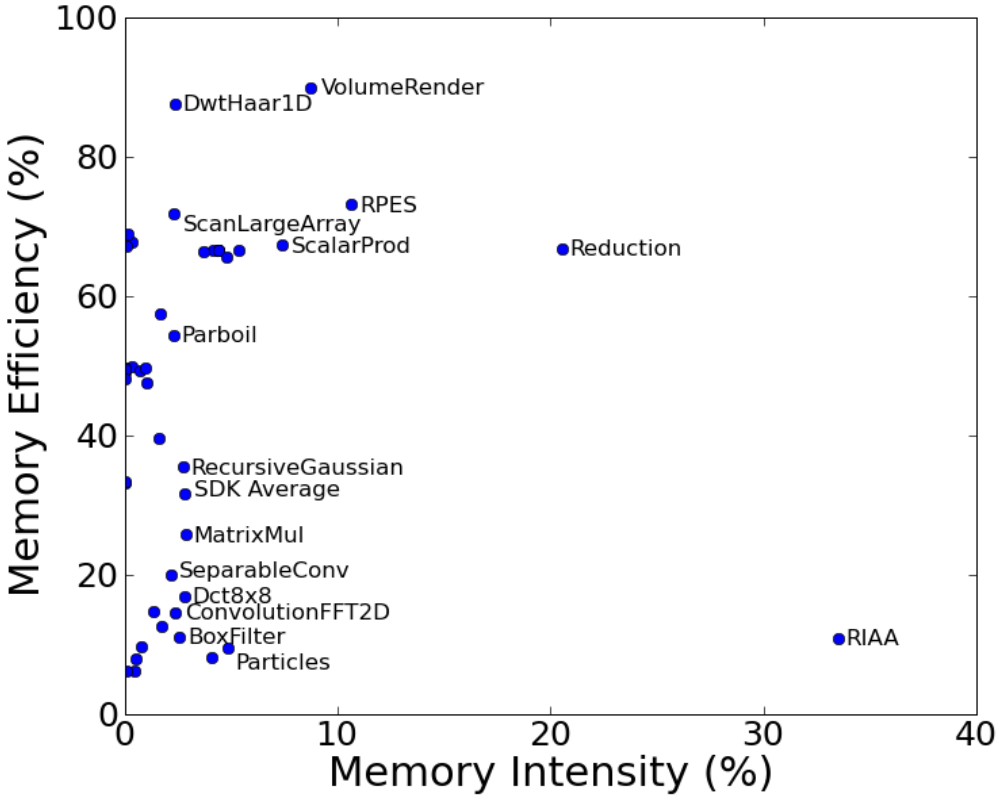
Average number of transactions needed to satisfy a load or store to global memory

Memory Efficiency

$$E_M = \sum_{i=1}^{kernels} \sum_{j=1}^{CTAs} \frac{2W_{i,j}}{T_{i,j}}$$

$W_{i,j}$: warps issuing memory instructions
$T_{i,j}$: transactions required

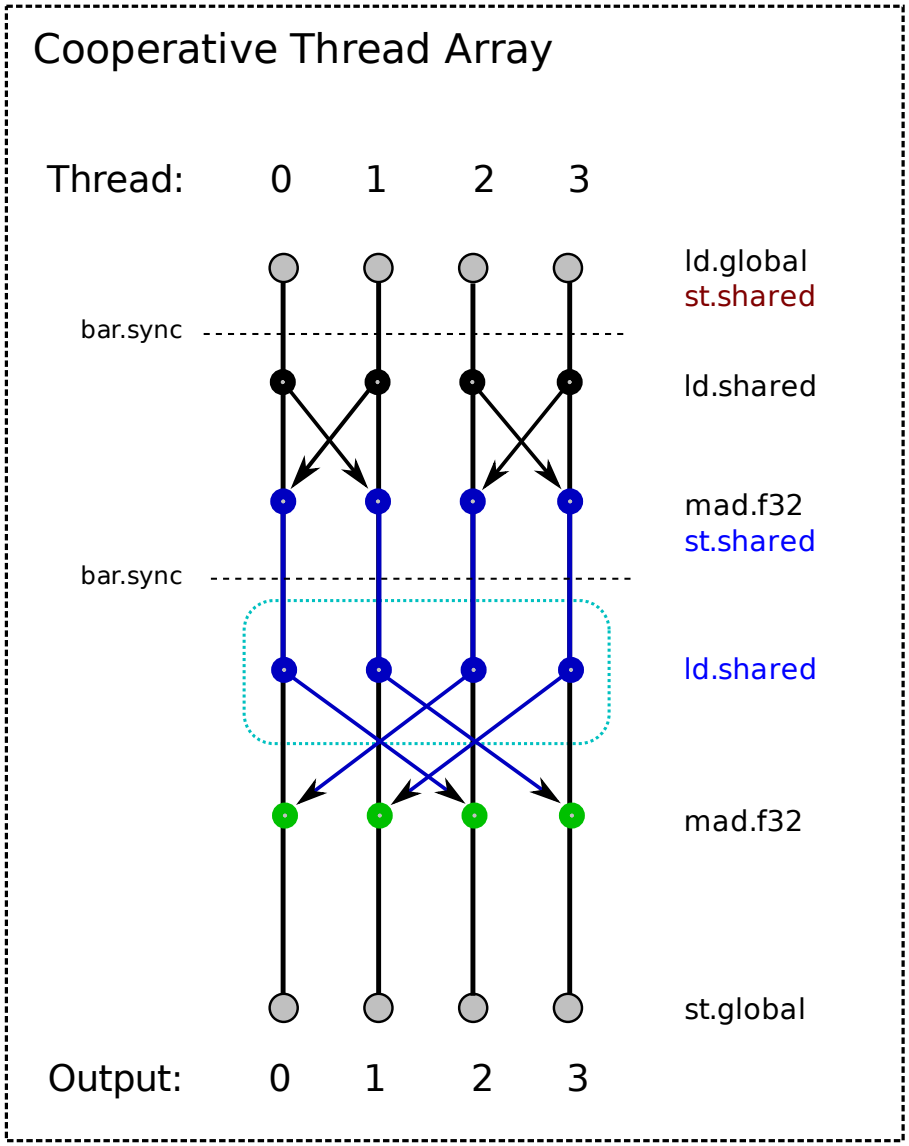
Memory Efficiency Results



Recommendation:

- Opportunity for compiler, hardware, runtime to trade off Activity Factor and Memory Efficiency

Interthread Data Flow



Intensity of producer-consumer relationships within a CTA

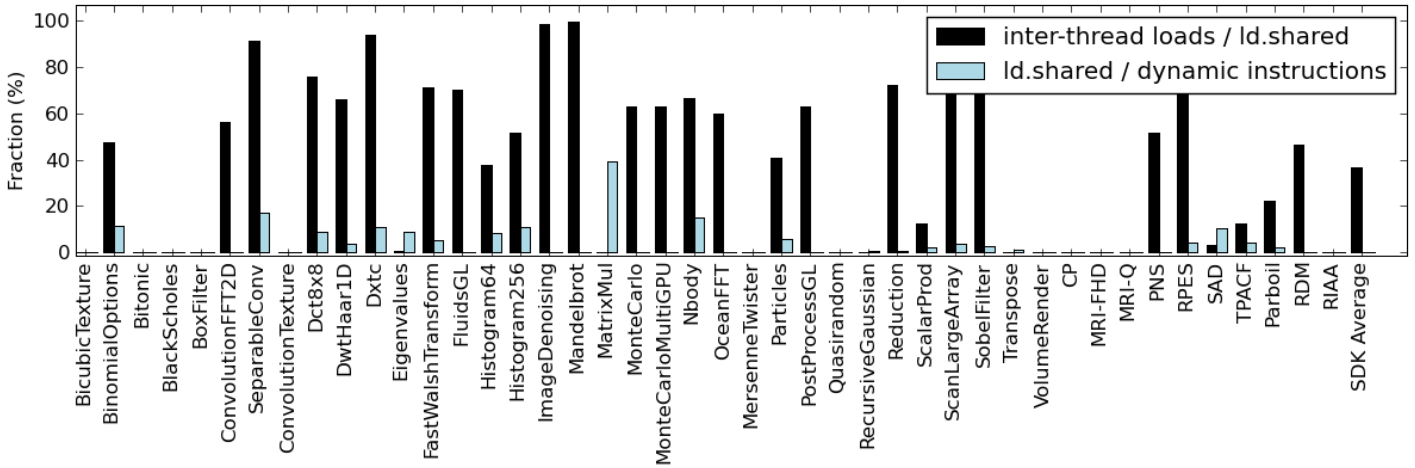
- ignore st.shared if value to store was loaded from global memory
- otherwise, st.shared annotates words in shared memory with writer's thread ID
- ld.shared compares thread ID with annotated thread ID
- count number of ld.shared with annotation != thread ID

Interthread Data Flow

$$IDF = \frac{X_i}{S_i}$$

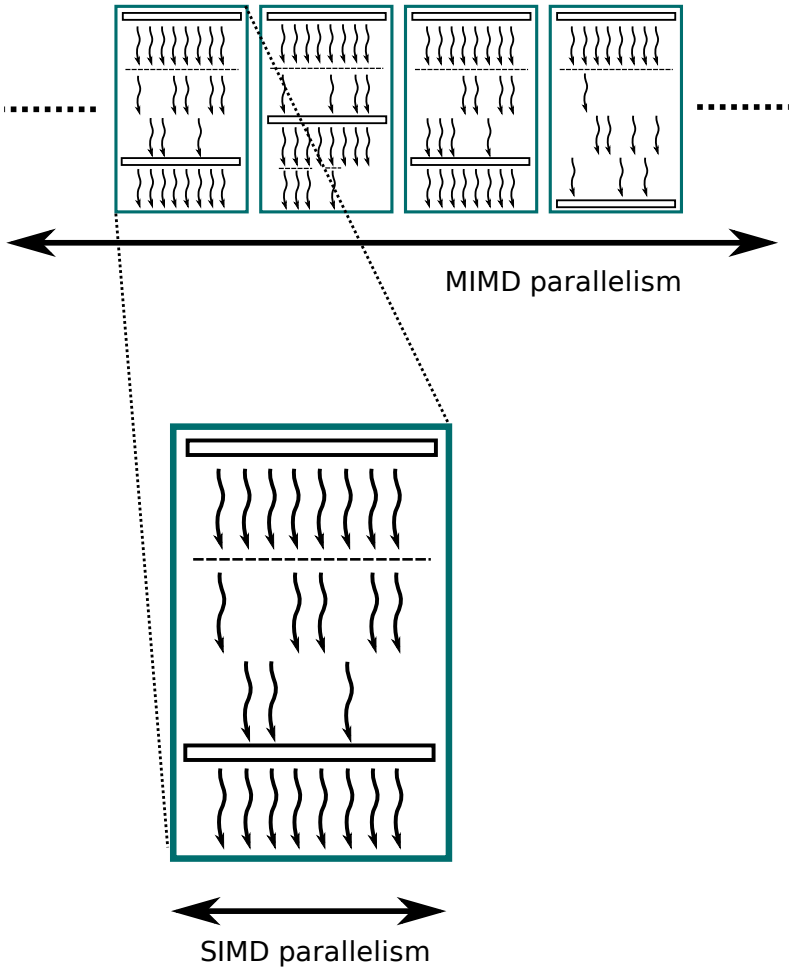
X_i: words loaded by inter-thread ld.shared
S_i: ld.shared instructions

Interthread Data Flow Results



- Shared memory used as: a cache, and as producer-consumer conduit
- Data dependencies between threads inform scheduling decisions and thread placement
- Recommendation:
  - Improve efficiency of data sharing among threads
  - Support smaller synchronization domains

Parallelism Scaling



Average speedup of MIMD/SIMD machine with infinite parallelism

MIMD Parallelism

$$MIMD_{kernel} = \frac{\sum_{i=1}^{ctas} D_i}{\max_{i=1}^{ctas}(D_i)}$$

$$MIMD_{application} = \frac{\sum_{i=1}^{kernels} D_i * MIMD_{kernel\_i}}{\sum_{i=1}^{kernels} D_i}$$

D_i: dynamic instructions
A_f: activity factor

SIMD Parallelism

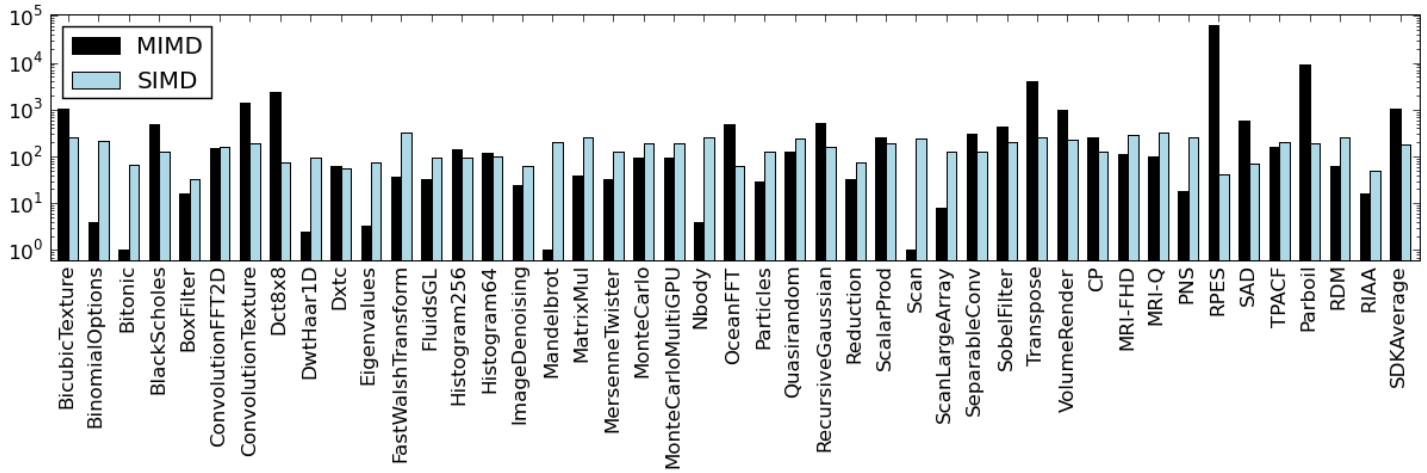
$$SIMD_{kernel} = \frac{\sum_{i=1}^{ctas} A_f * D_i}{\sum_{i=1}^{ctas} D_i}$$

$$SIMD_{application} = \frac{\sum_{i=1}^{kernels} D_i * SIMD_{kernel\_i}}{\sum_{i=1}^{kernels} D_i}$$

D_i: dynamic instructions
A_f: activity factor

Parallelism Results

* semi-log plot warning



- Applications should express as much possible parallelism to enable performance scaling
- Recommendation:
  - Efficiently mapping parallel code to collections of serial processors is crucial
  - Overheads: redundancy, context switching, locality of memory references

Related Work

GPGPU-Sim

- Derived from SimpleScalar to support GPU constructs
- Extended to include PTX as an instruction set
- Assesses impact of architectural parameters

Barra

- Virtual machine for SASS - native GPU instruction set
- Captures calls to CUDA driver API
- Results are detailed but specific to particular architecture implementation

Future Work

PTX to PTX

- Ocelot's PTX internal representation to produce executable PTX kernels
- Optimizations and transformations

PTX to LLVM to Multicore

- Translate PTX to Low-Level Virtual Machine
- Leverage existing optimization passes and code generators
- Target many existing multicore ISAs

Summary

- Characteristics of PTX applications motivate compiler optimizations, adaptive runtimes, architectural support
  - Thread restructuring reduces divergent control flow
  - Reconvergence methods tradeoff warp splitting with activity factor
  - Balance activity factor with memory efficiency
  - Data dependencies among threads suggest smaller synchronization domains
  - Data parallel kernels must be serialized efficiently
- Ocelot provides a unique approach to observing characteristics independently of particular architectures

Acknowledgements

The authors gratefully acknowledge the generous support of this work by LogicBlox Inc., IBM Corp., and NVIDIA Corp. both through research grants, fellowships, as well as technical interactions, and equipment grants from Intel Corp. and NVIDIA Corp.