A Detailed GPU Cache Model Based on Reuse Distance Theory

Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal
Eindhoven University of Technology (Netherlands)

Henri Bal
Vrije Universiteit Amsterdam (Netherlands)

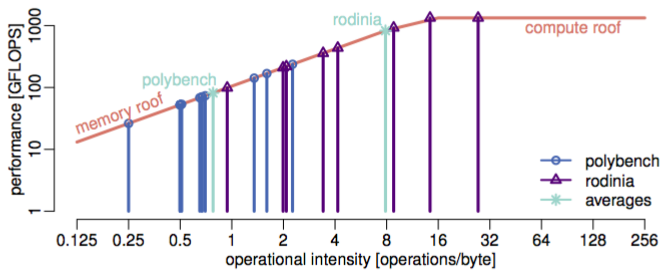
TU/e Technische Universiteit Eindhoven University of Technology

VU UNIVERSITY AMSTERDAM

Why caches for GPUs?

Isn't the GPU hiding memory latency through parallelism?
Why bother with caches at all?

- Lots of GPU programs are memory bandwidth bound (e.g. 18 out 31 for Parboil)
- 25% hits in the cache → 25% 'extra' off-chip memory bandwidth → up to 25% improved performance



Why caches for GPUs?

Isn't the GPU hiding memory latency through parallelism?
Why bother with caches at all?

- Lots of GPU programs are memory bandwidth bound (e.g. 18 out 31 for Parboil)
- 25% hits in the cache → 25% 'extra' off-chip memory bandwidth → up to 25% improved performance

This work focuses on the L1 data-caches only:

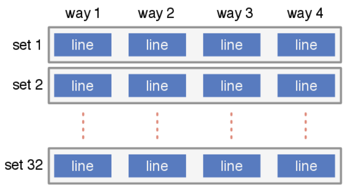
- Finding the order of requests to the L1 is the main challenge
- Existing multi-core CPU models can be re-used to get a L2 model

Modelling NVIDIA GPUs: L1 caches only reads

A cache model for GPUs

A (proper) GPU cache model does not exist yet. Why?

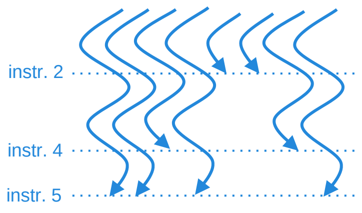
✓ Normal cache structure (lines, sets, ways)
✓ Typical hierarchy (per core L1, shared L2)

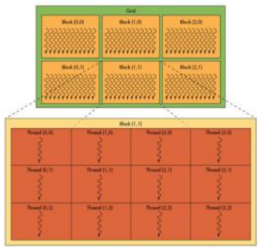


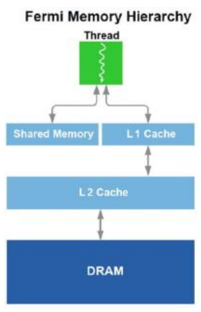
But how to find the **order** of requests?

X Hierarchy of threads, warps, threadblocks

X A single thread processes loads/stores in-order, but multiple threads can diverge w.r.t. each other







But what can it be used for?

A cache model can give:

1. A prediction of the amount of misses
2. Insight into the types of misses (e.g. compulsory, capacity, conflict)

Examples of using the cache model:

- A GPU programmer can identify the amount and types of cache misses, guiding him through the optimisation space
- An optimising compiler (e.g. PPCG) can apply loop-tiling based on a feedback-loop with a cache model
- A processor architect can perform design space exploration based on the cache model's parameters (e.g. associativity)

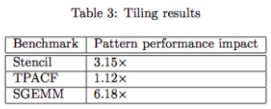
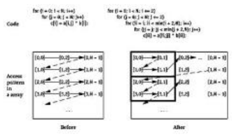
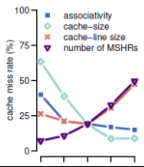


Table 3: Tiling results

Benchmark	Pattern performance impact
Stencil	3.15×
TPACF	1.12×
SGEMM	6.18×

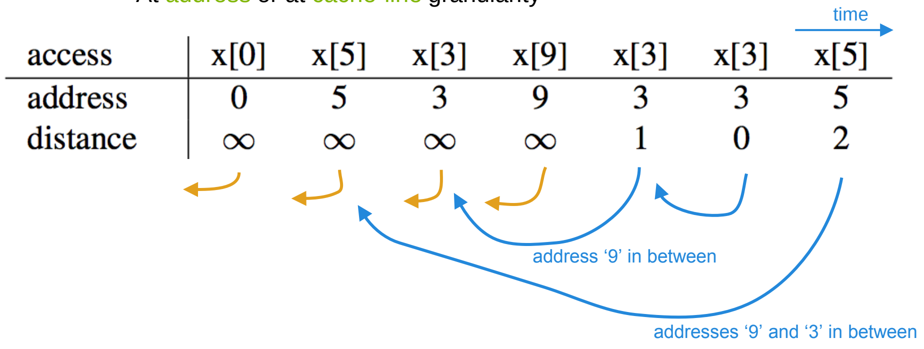




Background: reuse distance theory

Example of reuse distance theory:

- For sequential processors
- At address or at cache-line granularity



access	x[0]	x[5]	x[3]	x[9]	x[3]	x[3]	x[5]
address	0	5	3	9	3	3	5
distance	∞	∞	∞	∞	1	0	2

Background: reuse distance theory

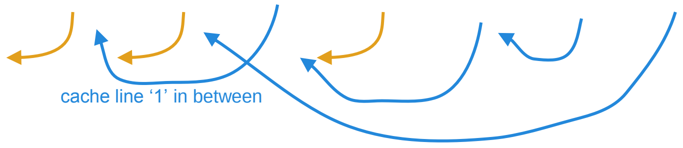
Example of reuse distance theory:

- For sequential processors
- At address or at cache-line (e.g. 4 items) granularity

time →

access	x[0]	x[5]	x[3]	x[9]	x[3]	x[3]	x[5]
address	0	5	3	9	3	3	5
distance	∞	∞	∞	∞	1	0	2
cache-line	0	1	0	2	0	0	1
distance	∞	∞	1	∞	1	0	2

(integer divide by 4)



Background: reuse distance theory

Example of reuse distance theory:

- For sequential processors
- At address or at cache-line (e.g. 4 items) granularity

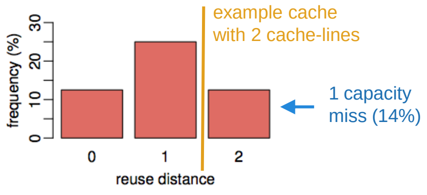
time →

access	x[0]	x[5]	x[3]	x[9]	x[3]	x[3]	x[5]
address	0	5	3	9	3	3	5
distance	∞	∞	∞	∞	1	0	2
cache-line	0	1	0	2	0	0	1
distance	∞	∞	1	∞	1	0	2

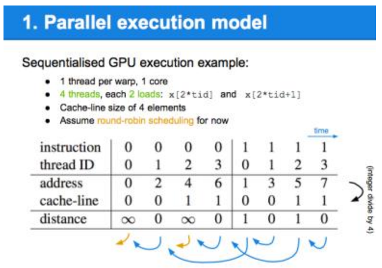
distance	0	1	2	∞
frequency	1	2	1	3

(at cache-line granularity)

3 compulsory misses (42%)



Extensions to reuse distance theory



1. Parallel execution model

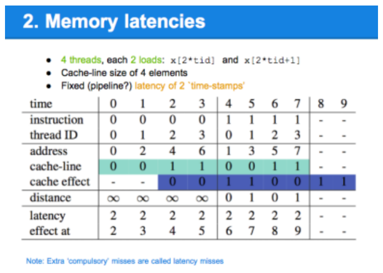
Sequentialised GPU execution example:

- 1 thread per warp, 1 core
- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Assume round-robin scheduling for now

time →

instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
distance	∞	0	∞	0	1	0	1	0

(integer divide by 4)

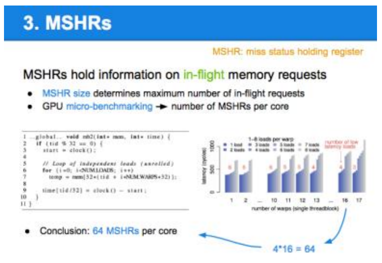


2. Memory latencies

- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Fixed (pipeline?) latency of 2 'time-stamps'

time	0	1	2	3	4	5	6	7	8	9
instruction	0	0	0	0	1	1	1	1	-	-
thread ID	0	1	2	3	0	1	2	3	-	-
address	0	2	4	6	1	3	5	7	-	-
cache-line	0	0	1	1	0	0	1	1	-	-
cache effect	-	-	0	0	1	1	0	0	1	1
distance	∞	∞	∞	∞	0	1	0	1	-	-
latency	2	2	2	2	2	2	2	2	-	-
effect at	2	3	4	5	6	7	8	9	-	-

Note: Extra 'compulsory' misses are called latency misses



3. MSHRs

MSHR: miss status holding register

MSHRs hold information on in-flight memory requests

- MSHR size determines maximum number of in-flight requests
- GPU micro-benchmarking → number of MSHRs per core

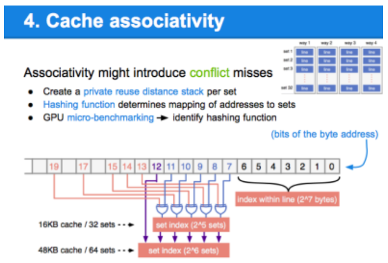
```
__global__ void mb2(int* mem, int* time) {
  if (tid % 32 == 0) {
    start = clock();

    // Loop of independent loads (unrolled)
    for (i=0; i<NUM_LOADS; i++)
      temp = mem[32*(tid + i*NUM_WARPS*32)];

    time[tid/32] = clock() - start;
  }
}
```

- Conclusion: 64 MSHRs per core

4*16 = 64



4. Cache associativity

Associativity might introduce conflict misses

- Create a private reuse distance stack per set
- Hashing function determines mapping of addresses to sets
- GPU micro-benchmarking → identify hashing function

1. Parallel execution model

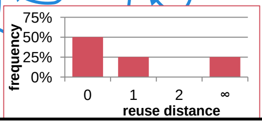
Sequentialised GPU execution example:

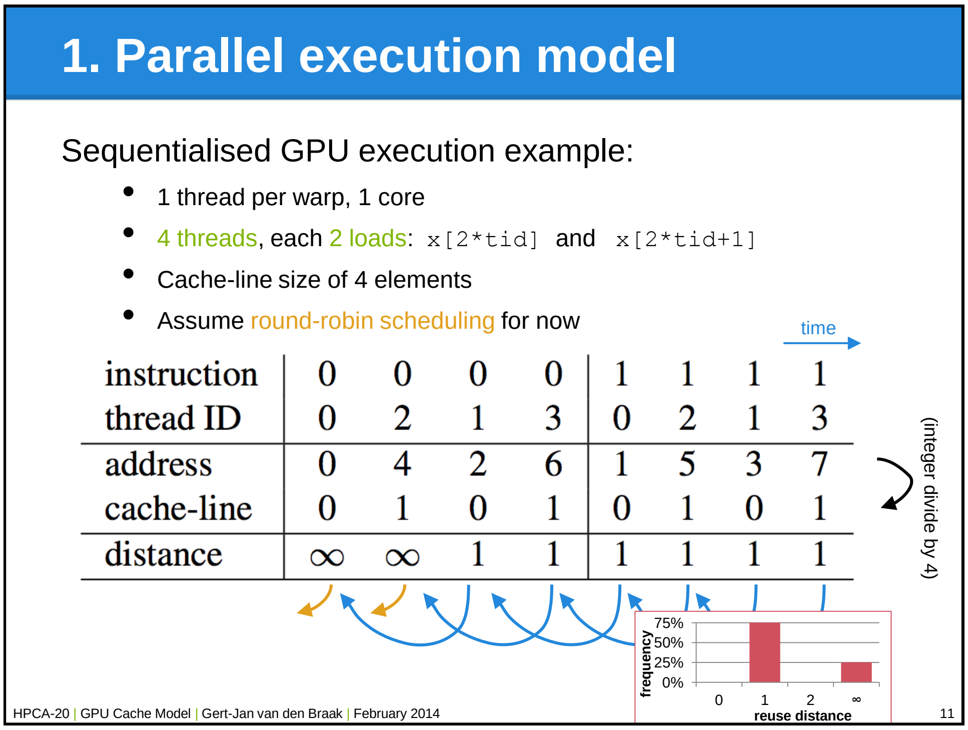
- 1 thread per warp, 1 core
- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Assume round-robin scheduling for now

time →

instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
distance	∞	0	∞	0	1	0	1	0

(integer divide by 4)





1. Parallel execution model

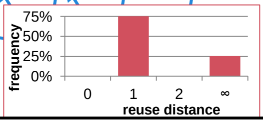
Sequentialised GPU execution example:

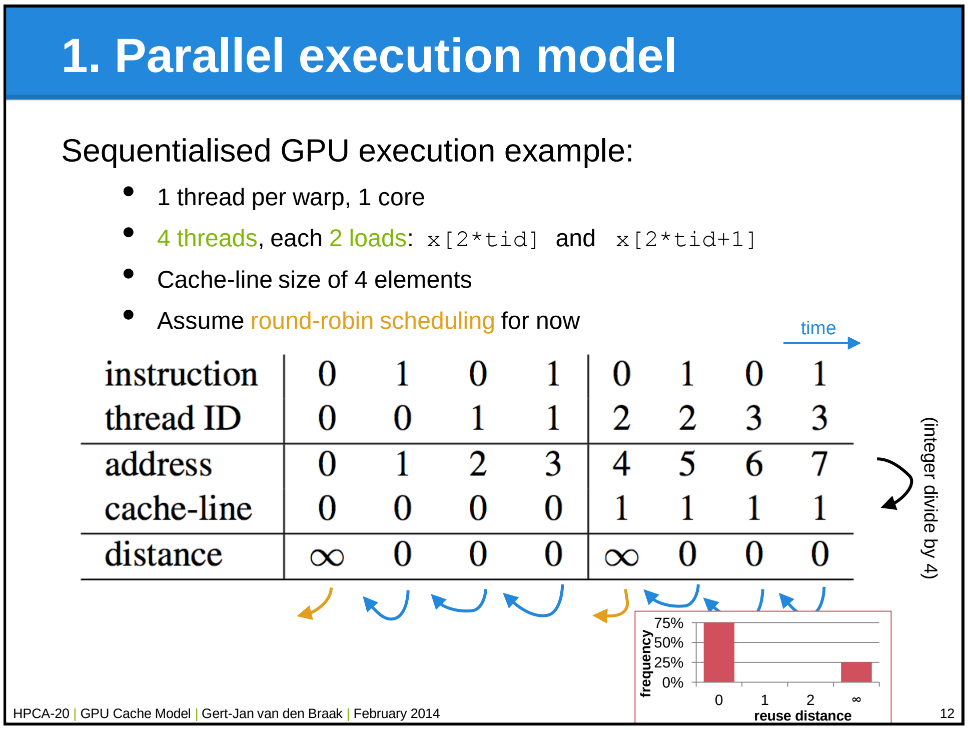
- 1 thread per warp, 1 core
- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Assume round-robin scheduling for now

time →

instruction	0	0	0	0	1	1	1	1
thread ID	0	2	1	3	0	2	1	3
address	0	4	2	6	1	5	3	7
cache-line	0	1	0	1	0	1	0	1
distance	∞	∞	1	1	1	1	1	1

(integer divide by 4)





1. Parallel execution model

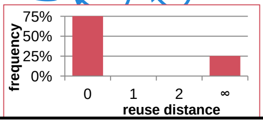
Sequentialised GPU execution example:

- 1 thread per warp, 1 core
- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Assume round-robin scheduling for now

time →

instruction	0	1	0	1	0	1	0	1
thread ID	0	0	1	1	2	2	3	3
address	0	1	2	3	4	5	6	7
cache-line	0	0	0	0	1	1	1	1
distance	∞	0	0	0	∞	0	0	0

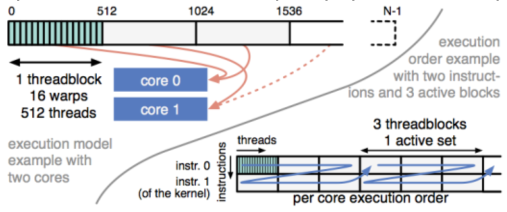
(integer divide by 4)



1. Parallel execution model

And how to handle warps, threadblocks, sets of active threads, multiple cores/SMs, etc?

- Implemented in the model (see paper for details)



But is this the correct order?

2. What about memory latencies and thread divergence?
3. And isn't there a maximum number of outstanding requests?
4. And did we handle cache associativity yet?

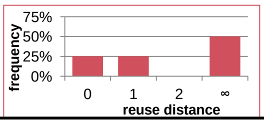
2. Memory latencies

- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Fixed latency of 2 `time-stamps'

time	0	1	2	3	4	5	6	7	8	9
instruction	0	0	0	0	1	1	1	1	-	-
thread ID	0	1	2	3	0	1	2	3	-	-
address	0	2	4	6	1	3	5	7	-	-
cache-line	0	0	1	1	0	0	1	1	-	-
cache effect	-	-	0	0	1	1	0	0	1	1
distance	∞	∞	∞	∞	0	1	0	1	-	-
latency	2	2	2	2	2	2	2	2	-	-
effect at	2	3	4	5	6	7	8	9	-	-

as before

Note: Extra 'compulsory' misses are called latency misses

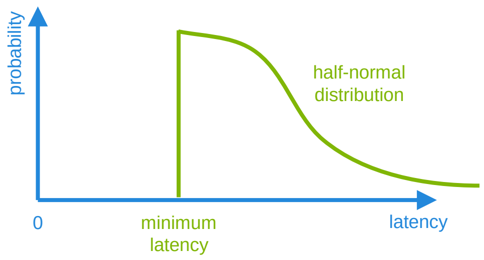


2. Memory latencies

Adding memory latencies changes reuse distances...

- ... and thus the cache miss rate
- But are the latencies fixed?
- And what values do they have?
- Note: ‘time-stamps’ not real time

Use different values for hit / miss latencies



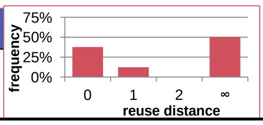
Most hit/miss behaviour (the ‘trend’) is already captured by:

- Introducing miss latencies
- Introducing a distribution

2. Memory latencies

- 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Variable latency of 2 (misses) and 0 (hits)

time	0	1	2	3	4	5	6	7
instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
cache effect	-	-	0	0	1 0	1 0	1	1
distance	∞	∞	∞	∞	0	0	1	0
hit/miss	m	m	m	m	h	h	h	h
latency	2	2	2	2	0	0	0	0
effect at	2	3	4	5	4			



3. MSHRs

MSHR: miss status holding register

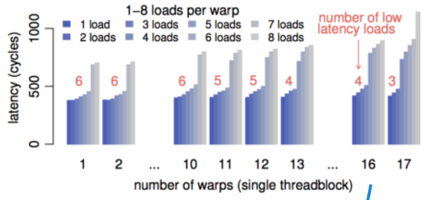
MSHRs hold information on in-flight memory requests

- MSHR size determines maximum number of in-flight requests
- GPU micro-benchmarking → number of MSHRs per core

```
__global__ void mb2(int* mem, int* time) {
  if (tid % 32 == 0) {
    start = clock();

    // Loop of independent loads (unrolled)
    for (i=0; i<NUM_LOADS; i++)
      temp = mem[32*(tid + i*NUM_WARPS*32)];

    time[tid/32] = clock() - start;
  }
}
```



- Conclusion: 64 MSHRs per core

4*16 = 64

3. MSHRs

- 2 out of the 4 threads, each 2 loads: `x[2*tid]` and `x[2*tid+1]`
- Cache-line size of 4 elements
- Only 1 MSHR

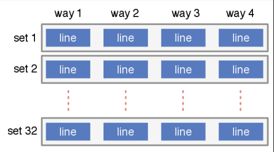
postponed

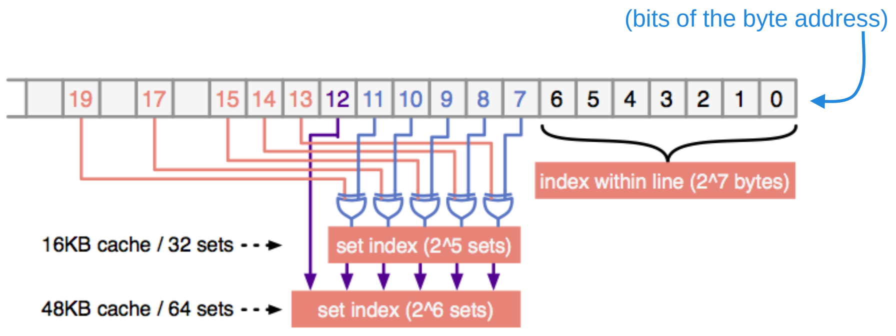
time	0	1	2	3	4	5	6
instruction	0	0	1	0	1	-	-
thread ID	0	2	0	2	2	-	-
address	0	4	1	4	5	-	-
cache-line	0	1	0	1	1	-	-
cache effect	-	-	0 0	-	-	1	1
distance	∞	∞	0	∞	∞	-	-
MSHRs used	0	1	0	0	1	-	-
status	miss	cancel	hit	miss	miss	-	-
MSHRs used	1	-	0	1	1	-	-
latency	2	-	0	2	2	-	-
effect at	2	-	2	5	6	-	-

4. Cache associativity

Associativity might introduce conflict misses

- Create a private reuse distance stack per set
- Hashing function determines mapping of addresses to sets
- GPU micro-benchmarking → identify hashing function

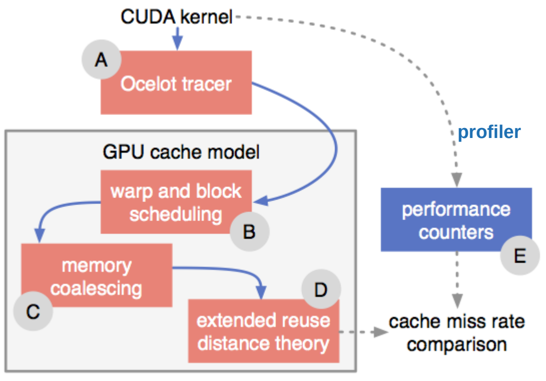




Implementation

Model (source-code) available at:

http://github.com/cnugteren/gpu-cache-model



Experimental set-up

Two entire CUDA benchmark suites:

- Parboil
- PolyBench/GPU

NVIDIA GeForce GTX470 GPU with two configurations:

- 16KB L1 caches (results in presentation)
- 48KB L1 caches

Four types of misses identified:

- Compulsory (cold misses)
- Capacity (cache size not finite)
- Associativity (set conflicts)
- Latency (outstanding requests)

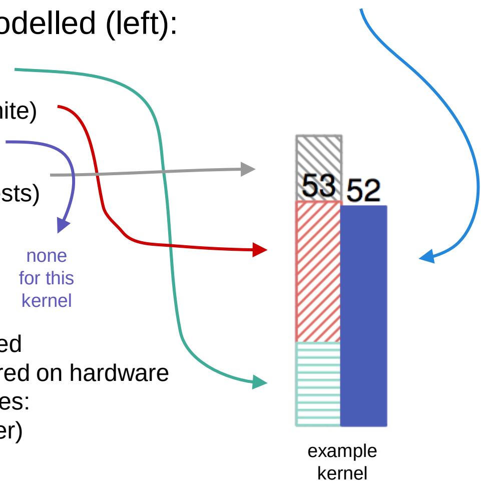
Compared against hardware counters using the profiler

Verification results: example

Compared with hardware counters using the profiler (right)

Four types of misses modelled (left):

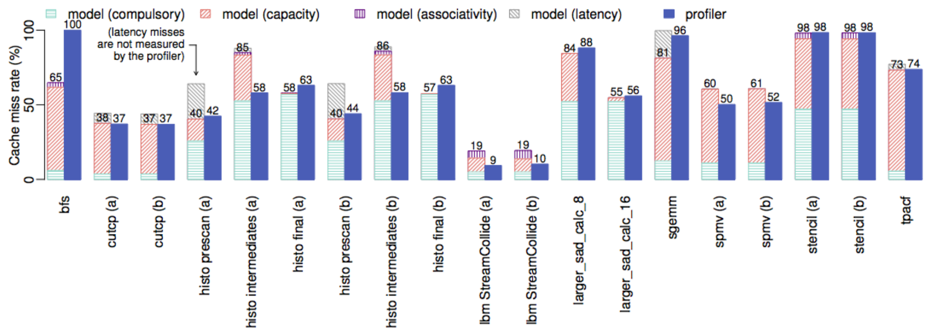
- Compulsory (cold misses)
- Capacity (cache size not finite)
- Associativity (set conflicts)
- Latency (outstanding requests)



Black number:

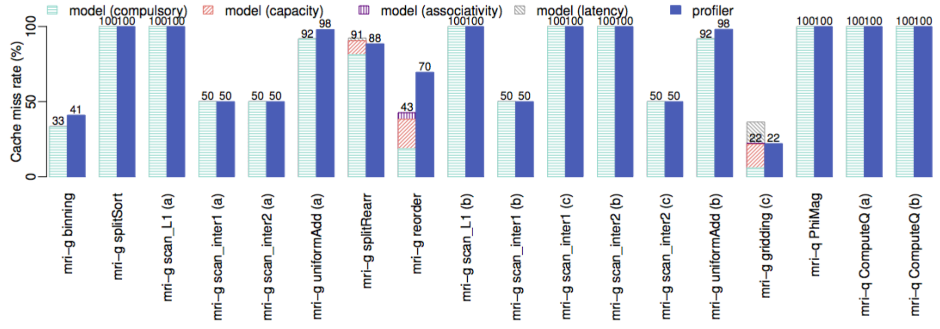
- 53% cache misses predicted
- 52% cache misses measured on hardware
- (not including latency misses: not measured by the profiler)

Verification results (1/3)



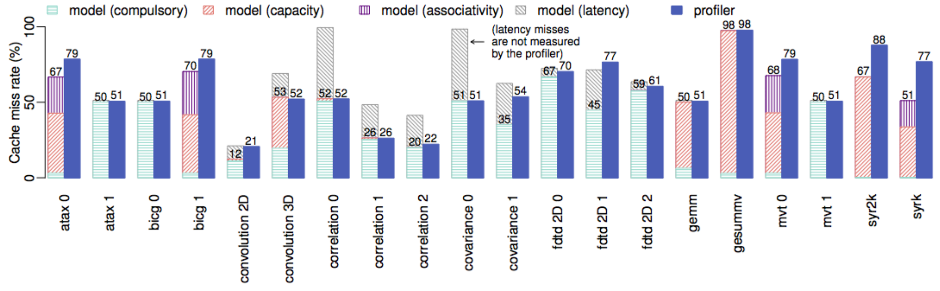
Note: matching numbers → good accuracy of the cache model

Verification results (2/3)



Note: matching numbers → good accuracy of the cache model

Verification results (3/3)

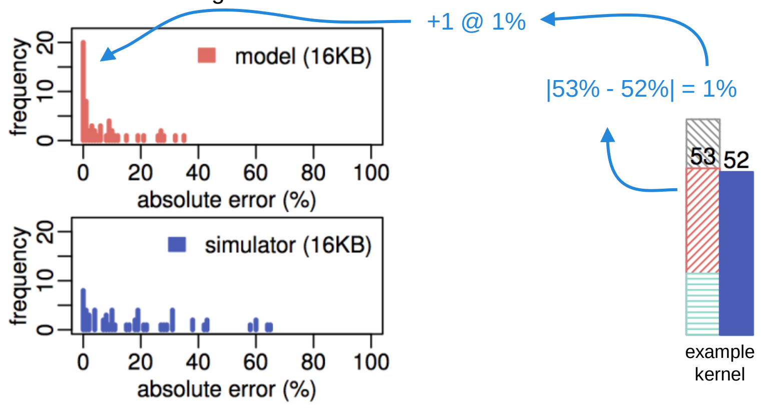


Note: matching numbers → good accuracy of the cache model

Are these results 'good'?

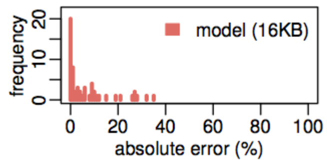
Compared with the GPGPU-Sim simulator

- Lower running time: from hours to minutes/seconds
- Arithmetic mean absolute error: 6.4% (model) versus 18.1% (simulator)
- Visualised as a histogram:



Did we really need so much detail?

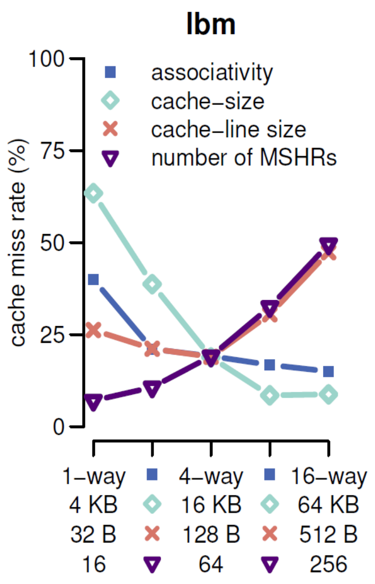
	arithmetic mean absolute error
Full model:	6.4% error
No associativity modelling:	9.6% error
No latency modelling:	12.1% error
No MSHR modelling:	7.1% error



Design space exploration

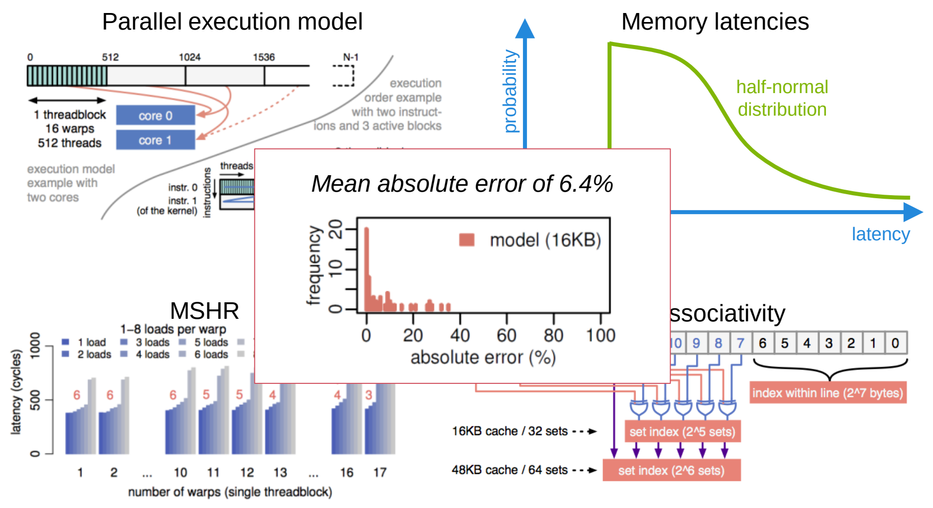
Cache parameters:

- Associativity
  1-way → 16 way
- Cache size
  4KB → 64KB
- Cache line size
  32B → 512B
- # MSHR
  16 → 256



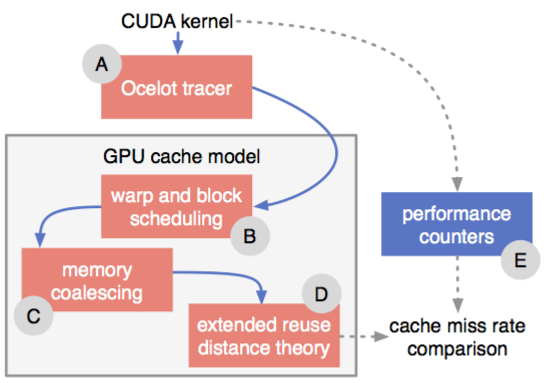
Summary

GPU cache model based on reuse distance theory



HPCA-20 | GPU Cache Model | Gert-Jan van den Braak | February 2014 29

Questions



CUDA kernel
A
Ocelot tracer
GPU cache model
warp and block scheduling
B
memory coalescing
C
D
extended reuse distance theory
performance counters
E
cache miss rate comparison

HPCA-20 | GPU Cache Model | Gert-Jan van den Braak | February 2014 30