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?

✗ Hierarchy of threads, warps, threadblocks
✗ 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)

Background: reuse distance theory

Example of reuse distance theory:

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>distance</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Tiling results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Pattern performance impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil</td>
<td>3.15x</td>
</tr>
<tr>
<td>TPACF</td>
<td>1.12x</td>
</tr>
<tr>
<td>KOEVA</td>
<td>0.18x</td>
</tr>
</tbody>
</table>
Background: reuse distance theory

Example of reuse distance theory:

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>distance</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>cache-line distance</td>
<td>∞</td>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Example cache with 2 cache-lines:

- 3 compulsory misses (42%)
- 1 capacity miss (14%)
1. Parallel execution model

Sequentialised GPU execution example:

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

<table>
<thead>
<tr>
<th>instruction</th>
<th>0 0 0 0</th>
<th>1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread ID</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>address</td>
<td>0 2 4 6</td>
<td>1 3 5 7</td>
</tr>
<tr>
<td>cache-line</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>distance</td>
<td>(\infty) 0 (\infty) 0 1 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

2. Memory latencies

- 4 threads, each 2 loads: \(x[2*\text{tid}]\) and \(x[2*\text{tid}+1]\)
- Cache-line size of 4 elements
- Fixed (programmable) latency of 2 time-steps

<table>
<thead>
<tr>
<th>instruction</th>
<th>0 0 0 0 1 1 1 1</th>
<th>0 0 0 1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread ID</td>
<td>0 1 2 3 0 1 2 3</td>
<td>0 1 2 3 0 1 2 3</td>
</tr>
<tr>
<td>address</td>
<td>0 2 4 6 1 3 5 7</td>
<td>0 2 4 6 1 3 5 7</td>
</tr>
<tr>
<td>cache-line</td>
<td>0 0 1 1 0 0 1 1</td>
<td>0 0 1 1 0 0 1 1</td>
</tr>
<tr>
<td>distance</td>
<td>0 1 0 1 1 1 1 1</td>
<td>0 1 0 1 1 1 1 1</td>
</tr>
<tr>
<td>latency</td>
<td>2 2 2 2 2 2 2 2</td>
<td>2 2 2 2 2 2 2 2</td>
</tr>
<tr>
<td>effect</td>
<td>2 3 4 3 6 7 8 9</td>
<td>2 3 4 3 6 7 8 9</td>
</tr>
</tbody>
</table>

3. MSHRs

MSHRs hold information on in-flight memory requests
- MSHR size determines maximum number of in-flight requests
- GPU microbenchmarking = number of MSHRs per core
- Conduction: 64 MSHRs per core

4. Cache associativity

Associativity might introduce conflict misses
- Create a private reuse distance cache per set
- Mapping function determines mapping of addresses to sets
- GPU microbenchmarking = Identify mapping function

Note: Cache terminology is used for clarity.
1. Parallel execution model

Sequentialised GPU execution example:

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ID</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Address</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Cache-line</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Distance</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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: \( x[2*\text{tid}] \) and \( x[2*\text{tid}+1] \)

- Cache-line size of 4 elements

- Fixed latency of 2 ‘time-stamps’

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>thread</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>address</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cache-line</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cache effect</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>distance</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>latency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>effect at</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>thread ID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>address</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>cache-line</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cache effect</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>distance</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hit/miss</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>latency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>effect at</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. MSHRs

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

```
1..global void mbl(int mem, int time) {
2   if (tid % 32 == 0) {
3       start = clock();
4       // Loop of independent loads (unrolled)
5       for (i=0; i<NUMLOADS; i++)
6       temp = mem[32*(tid + i*NUM_warps*32)];
7       time[tid/32] = clock() - start;
10 }  
11 }
```

- Conclusion: 64 MSHRs per core

---

### 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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>thread ID</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>address</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cache-line</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cache effect</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>distance</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MSHRs used</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>status</td>
<td></td>
<td>miss</td>
<td>cancel</td>
<td>hit</td>
<td>miss</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MSHRs used</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>latency</td>
<td>2</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>effect at</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

---
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:

\[ |53\% - 52\%| = 1\% \]

\[ +1 @ 1\% \]
Did we really need so much detail?

- 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

Parallel execution model

Memory latencies

half-normal distribution

Mean absolute error of 6.4%

Questions