tiny-cache

An implementation of a multicore cache and tiny-isa assembler

Dylan Sandall

12/07/2024 (last updated 11/12/2024)
CPE-523

ABSTRACT

In an effort to better understand highly parallelized data synchronization, I implemented a cache for the tiny-gpu project, an open source GPU. The implemented cache is direct-mapped, and fairly primitive aside from the fact that it must serve multiple consumers concurrently.

In addition to the cache and testbenches, I wrote a small assembler for the modified assembly language tiny-gpu supports. The tiny-gpu ISA is not really intended for production, but is based loosely on RISC-V and pared down for the beginner-friendly project.

“Global” memory and Caches

> *Why Caches?*

As any digital designer worth their salt knows, memory is slow, and for this reason caches are a crucial optimization point for processor design. Layering caches increases the area usage of the design, but can allow for much faster data accesses when nearby, a principle known as *spatial locality.* Because the memory accesses are decoupled from the cache accesses, there are a number of cache control strategies which attempt to increase the chance that the requested data is already present in the most local cache, including direct and set associative.

> *Why are Caches problematic for GPUs? (and multicore CPUs)*

For a single core, it is trivial to manage data synchronization, as you only have a single writer/reader of the data[1]. Parallelization through increasing the core count of a processor adds increased complexity to the cache, as now you must handle transactions with some number of consumers, and some number of data providers. For a 4 core processor, with single channel memory, you have 4 potential sources of data, and no guarantee that they will agree. This is the problem known as *cache coherency*, and is a huge problem for modern digital design. Modern high performance processors typically include 3 layers of cache, and rarely 4 or more, due to the additional area, power, and

To further complicate the issue, and illustrate how there is no clear solution to the problem, there are multiple hardware configurations which can be used. Transferring data from N number of memory channels to M lower level caches can be done using a simple shared bus, crossbar busses, or a dedicated memory transaction arbiter (Yousif et al.).

> *Great minds think in parallel*

This problem of shared memory access is fundamental, and there are numerous ASICs, algorithms, and design patterns which exist to exploit parallelism and avoid the need for shared memory in the first place.

[1] As long as the memory itself is not shared with other hardware units, like interrupts or DMA channels.

“It’s probably fine… just ship it”

> What is most important: Power Consumption, Clock Speed, or Man Hours?

As one might expect, designing a processor is a difficult task. If you disregard the actual design, fabrication, and funding, you still have to go through the process of testing it out before selling to a customer. Validation, which accounts for a huge amount of the effort put into modern chips, is a problem that has only grown in scope as our digital design skills have increased. As chips scale down, core counts and cache layers rise, the need for an efficient software stack and comprehensive testing environment becomes evident.

HIGH LEVEL DESIGN

> An introduction to tiny-gpu

> *“A minimal GPU implementation in Verilog optimized for learning about how GPUs work from the ground up. Built with <15 files of fully documented Verilog, complete documentation on architecture & ISA, working matrix addition/multiplication kernels, and full support for kernel simulation & execution traces” https://github.com/adam-maj/tiny-gpu*

The tiny-gpu, as provided on github, has 2 compute cores,[2] with 4 threads each. It makes use of the Harvard Architecture for memory, and uses memory controllers to interface the fetchers and load store units with the program and data memory. The data memory has 4 8-bit channels, and the program memory has 1 16-bit channel. There is 1 fetcher and 4 load/store units per core.

Despite the 16 bit instructions, the tiny-gpu operates in an 8-bit address space, and it’s operations are the bare minimum. Despite the limitations, it is highly parameterized, and could be adapted to be more suited for testing highly parallel cache designs.

[2] Although the diagram on the readme illustrates 4 cores, which is not the default configuration

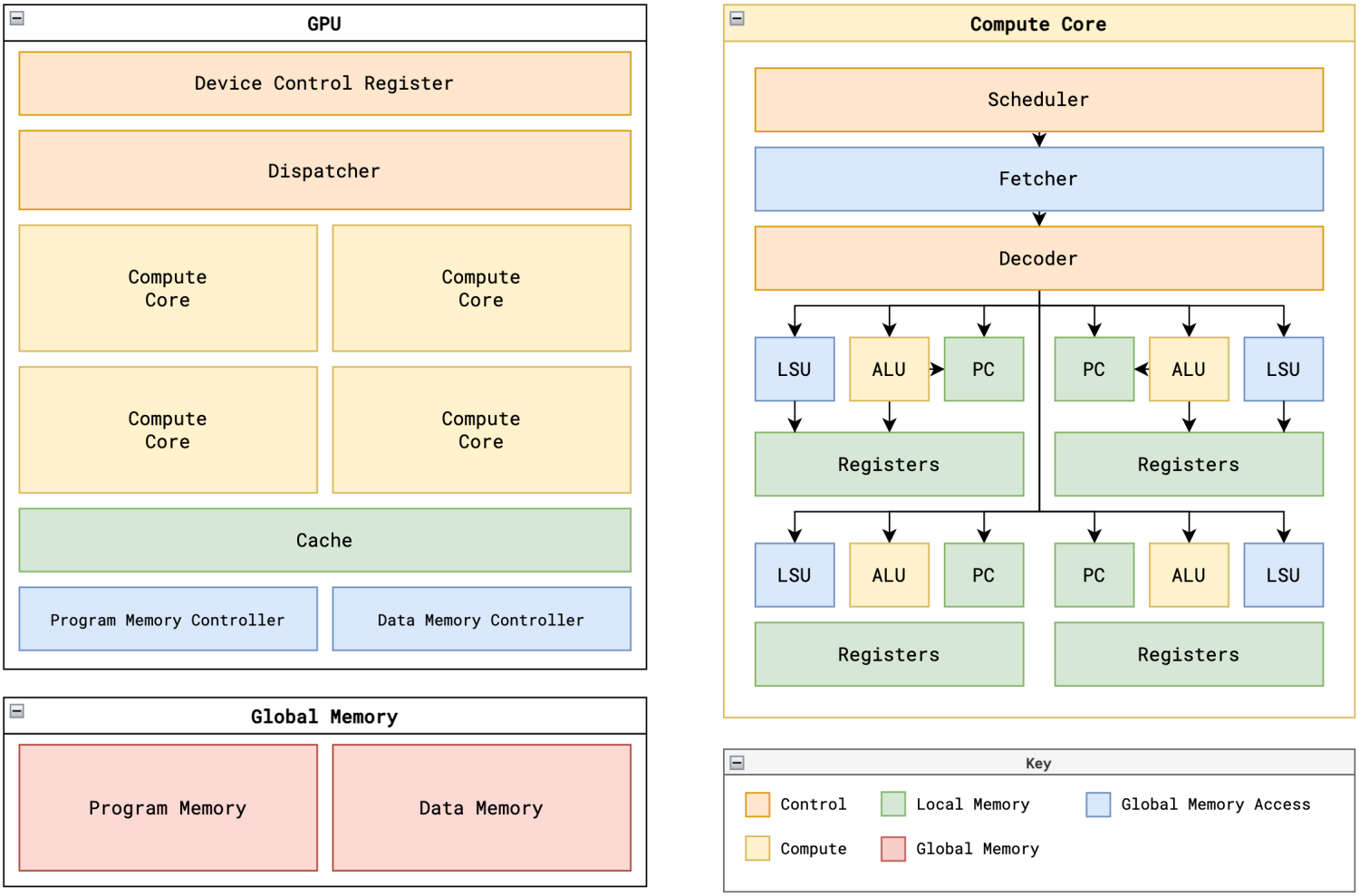

Figure 1: tiny-gpu reference design, configured with 4 cores and 4 threads per core

> *Modifications Introduced*

- Program Cache
  - 16 x 16 bit cache lines (yes, a whopping 256 bits of cache!)
  - 4 bit tag/index, no offset
  - Direct Mapped
- Data Cache / Controller
  - 16 x 16 bit cache lines
  - 3 bit tag, 4 bit index, 1 bit offset
  - Direct Mapped

> *Design Process*

Implementing a cache requires two related but separate tasks, storing the data and managing its correctness, and communicating with the source and consumers. The memory controller included with tiny-gpu handles the second task nicely, and naturally integrates with the cache design. The program memory cache as included with this document is separate from the program memory controller, whereas the data cache is integrated with the controller for better performance.

By first implementing a single input, single output cache for the program memory, I was able to get a basic cache design working. The program memory cache used in this design improves performance only when branches and loops are used, as no prefetching occurs.

Following that, a direct mapped cache was designed based on the program cache, including modifications to allow writes and integrating the memory controller with the cache storage.

> *Validation*

Beyond the intricacies of the cache design, a major hurdle was uncovered - validating the design. Aside from the provided test_matadd and test_matmul programs, there was no way to test code on the tiny-gpu, cache or no cache. This lead me to building a “quick and dirty” assembler for the tiny-gpu ISA. It took me way longer than expected, and was only the first part of validation.

Another issue arose from the memory simulation - The existing tiny-gpu project makes use of a CocoTB module for memory simulation rather than SystemVerilog, so it took a bit of work to make it behave as desired. The benefits of writing testbenches in Python is very attractive, but CocoTB’s simulation is very slow, and most of what it enables could be done with scripting. Overall, I would not recommend CocoTB for any serious project, aside from the rapid prototyping stage.

VERIFICATION/THOUGHTS ON METRICS

Testing was done using CocoTB, icarus-verilog, sv2v. A variety of assembly programs were tested and verified that their resulting data memory was equal. Various memory read/write delay parameters were used to evaluate the performance in different memory configurations.

RESULTS - Figure 1

Test Duration (ns)	tiny-gpu (reference) Execution Time	tiny-gpu + tiny-cache Execution Time	Execution time (%)
Test_matadd 5 cycle memory delay	7075 ns	6775 ns	96 %
Test_matmul 5 cycle memory delay	17050 ns	17675 ns	104 %
Test_load 0 cycle memory delay	15025 ns	17025 ns	113 %
Test_load 20 cycle memory delay	30150 ns	21825 ns	72 %

ANALYSIS

There is considerable room for improvement, but the results are generally positive. An improvement on the design would have cache controller threads lock onto cache lines, not lock onto requesters. Future improvements could include set associativity, or splitting the cache levels even further. Some sort of synchronization primitive could be useful as well, but the most pressing improvement would be in the test and verification environment. While much has been done to set up a test environment, it is still lacking, and severely limits the testing for highly parallel systems, where layered caches become more important.

RESOURCES

Resource #1 - tiny-gpu files

- Majorly changed files include:
  - tiny-cache/test/
  - tiny-cache/src/dmem_cache.sv
  - tiny-cache/src/pmem_cache.sv
  - tiny-cache/src/gpu.sv

Resource #2 - tiny-assembler files

- This was entirely from scratch, referencing the tiny-gpu ISA and test programs.

RELEVANT PAPERS

Indra Vir Singh, et al. “Cache Coherence for GPU Architectures.” *High-Performance Computer Architecture*, 23 Feb. 2013, https://doi.org/10.1109/hpca.2013.6522351. Accessed 1 May 2023.

Loh, Gabriel H. *Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy*. 12 Dec. 2009, https://doi.org/10.1145/1669112.1669139. Accessed 29 Aug. 2023.

Yousif, Mazin S, et al. “Cache Coherence in Multiprocessors: A Survey.” *Advances in Computers*, 1 Jan. 1995, pp. 127–179, https://doi.org/10.1016/s0065-2458(08)60546-x. Accessed 8 Dec. 2024.