ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

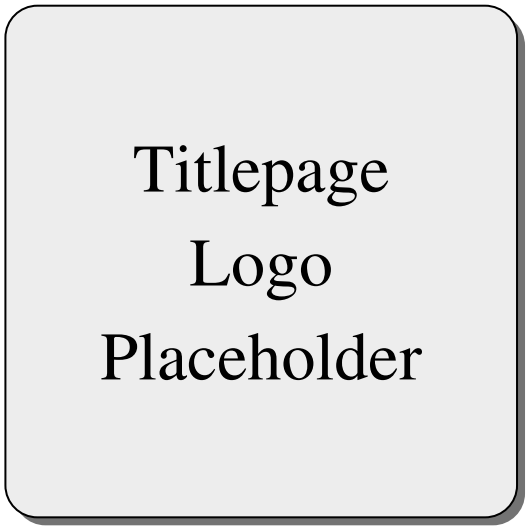
Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Spring 2024

Implementing an OpenMP runtime for MemPool in LLVM

Semester Project



Diego de los Santos Gausí
dgausi@student.ethz.ch

1st of July 2024

Advisors: Samuel Riedel, sriedel@iis.ee.ethz.ch
Sergio Mazzola, smazzola@iis.ee.ethz.ch

Professor: Prof. Dr. L. Benini, lbenini@iis.ee.ethz.ch

Todo list

Abstract

In order to effectively harness MemPool's parallelism, an initial GCC-compatible OpenMP runtime was developed for this architecture, supporting the most commonly used set of features. However, due to increasing reliance on the LLVM compiler infrastructure by the MemPool team, it is becoming crucial to have an OpenMP runtime compatible for it. This project aims to implement an OpenMP runtime that works with LLVM, supporting at least the same set of features as the previous runtime, while maintaining comparable performance and offering the possibility of future additions and improvements.

Add examples for improvements, perhaps cite the dynamic memory remapping thing

Acknowledgments

You rarely work in complete isolation, and this is the place to acknowledge contributions by others.

Declaration of Originality

Download the official declaration of originality[1], print it, and sign it. When printing your thesis, replace this sheet with the physically signed original paper. For the digital version, scan the filled declaration of originality (either before signing or remove your signature[2] from the image) and replace the text inside this file with `\includepdf[scale=0.9]{declaration_of_originality}`.

[1] https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/declaration-originality.pdf

[2] By removing your signature from the digital version of your report, you enable us to share your report with collaborators without having to deal with privacy concerns.

Contents

Chapter 1

Introduction

MemPool [1] is an open-source manycore architecture developed at ETH. It is comprised of up to 1024 [2] 32-bit RISC-V Snitch [3] cores that share a common L1 cache. Unlike on GPUs, each core is individually programmable, therefore enabling the familiar shared memory programming model.

Although MemPool's parallelism can be harnessed by manually managing it using the primitives provided by MemPool's C runtime library, this can become a tedious and error prone task since the programmer has to be knowledgeable about the underlying architecture. Luckily, higher level abstraction layers such as OpenMP [4] allow the programmer to express this parallelism in a clear and concise way using the fork-join paradigm and without needing advanced architectural knowledge.

OpenMP is comprised of a set of compiler directives for both C/C++ and Fortran that enable this higher level abstraction. In order to use OpenMP, one has to use a compatible compiler that is able to roughly do the following:

1. The compiler must be able to parse the directives and compile the surrounding code according to their semantics.
2. The output of the compilation process must match the target platform in order to make use of its parallelism capabilities and primitives.

Both the GNU Compiler Collection (GCC) and LLVM provide an OpenMP runtime library interface that bridges the gap between both points: instead of directly generating a binary from OpenMP annotated code, the compiler generates calls to the OpenMP runtime library. This library has to be aware of how to parallelize the code given the platform, which makes it inherently non-portable. Naturally, popular compilers ship with their own OpenMP runtime library implementation, however, in order to get around the non-portability issue, they target a host running an operating system which is able to provide the necessary parallelism primitives. Since MemPool is designed to

run bare-metal applications, this means that a custom OpenMP runtime library has to be developed.

Previously, the authors of MemPool developed a GCC compatible OpenMP runtime library with support for the most commonly used OpenMP constructs, such as *work sharing*, *critical sections* and *atomics*.[1] However, as the MemPool team increasingly relies on the LLVM compiler infrastructure, it is essential to have a compatible OpenMP runtime library for LLVM as well.

The goal of this project is to implement an OpenMP runtime for LLVM that works on MemPool, supports at least the same set of features as the previous GCC runtime and achieves comparable performance.

The runtime will be implemented in C++ as detailed in Chapter 4. In Chapter 5, the new runtime will be evaluated in terms of performance and correctness against the GCC runtime. Finally, we will summarize the results and discuss possible shortcomings and future work in Chapter 6.

[1] This implementation will be discussed in more detail in Section 3.1.

Chapter 2

Background

2.1. MemPool

As demand for processing of highly parallel workloads grow over time, computing architectures with increasing core/processing element counts become more common. These can range all the way from custom accelerators with domain-specific architectures and programming models, to general-purpose multi-core processors. There is generally a trade-off between ease of programming and performance/efficiency.

One can achieve an interesting middle ground by grouping many relatively simple general purpose cores into a so called *compute cluster*. In practice, however, this approach does not scale well beyond a few tens of cores, which lead to the development of *multi-cluster* architectures comprised of many of such clusters. While this allows for greater core counts, it also introduces additional complexity and memory access overhead between each cluster.

MemPool [1] tries to address this issue by scaling a single cluster to the order of hundreds of cores that share a low-latency software managed L1 cache.

2.1.1. Architecture Overview

At the heart of MemPool is the *Snitch* core, which is a simple, in-order, 32-bit RISC-V core. This core was adapted for this architecture by adding the ability to retire out-of-order load instructions, which, in addition to its previously existing support for issuing multiple outstanding loads and stores through scoreboarding, allows MemPool to completely hide the L1 access latency.

MemPool's architecture is hierarchical and comprised of the following building blocks:

Tile A tile is made up of a number of cores, some shared L1 banks, an L1 instruction cache shared amongst the tile's cores, as well as a set of remote ports to send/receive requests to/from other tiles trying to access the global L1 cache. Finally, it also has an Advanced eXtensible Interface (AXI) port connected to the system bus, which allows the tile to access the global L2 cache or system memory.

Group A group is made up of multiple tiles. Each tile's remote ports are connected with each other both within the same group as well as with other groups through a series of interconnects. Each tile's AXI port is connected to the rest of the system, also supporting Direct Memory Access (DMA).

Cluster Finally, a cluster is made up of multiple groups. Here, an AXI interface is used to connect MemPool to the rest of the system (e.g., L2 cache, system memory, etc.).

Another interesting aspect of MemPool's architecture is the *hybrid memory addressing scheme*. MemPool's shared L1 memory is interleaved at the word level across all banks. This means that accessing any given sequential region of memory would result in many strided accesses to remote tiles. This is not an issue for data shared across all cores, however, it is needlessly costly for local data (e.g., the stack). For this reason, MemPool is able to dedicate *sequential regions* of memory that are interleaved only within the same tile by re-interpreting the memory address, which is entirely done in hardware.

2.2. OpenMP

In her report [5], Qiu provides a comprehensive overview of OpenMP and its most commonly used constructs and clauses. In this work, we will implement all of the ones mentioned in her work, as well as the *teams* construct, described in the following:

The *teams* construct uses the following syntax:

Listing 2.1: teams construct

```
1 #pragma omp teams [clause[ [,] clause] ... ] new-line
2   structured-block
```

with the following clauses:

Listing 2.2: teams clauses

```
1 num_teams(scalar-integer-expression)
2
3 thread_limit(scalar-integer-expression)
4
```

```
 5 default(shared | firstprivate | private | none)
 6
 7 private(list)
 8
 9 firstprivate(list)
10
11 shared(list)
12
13 reduction([default ,] reduction-identifier : list)
14
15 allocate([allocator :] list)
```

The *teams* construct is used to create a league of teams, where each team is a group of threads. The number of teams is less then or equal to the value defined by the *num_teams* clause, and the maximum number of threads in each team is less than or equal to the value defined by the *thread_limit* clause.

The *structured_block* is able to contain other OpenMP constructs such as *parallel*, meaning that this construct essentially allows the creation of independent groups of threads, each of which could work on different tasks. This is particularly useful in the context of MemPool, as it allows for the creation of multiple teams, each of which could be assigned to a different tile or group (or even to different clusters in a multi-cluster system).

Chapter 3

Related Work

3.1. GCC Implementation

As previously mentioned, MemPool already had a working OpenMP runtime library for GCC. In the following, we will briefly discuss its architecture and implementation.

3.1.1. Event Loop

Because the entry point of the application is the same for all cores, it is necessary to distinguish between master and worker cores at runtime. Therefore, every application intended to use the GCC OpenMP runtime has to include the code shown on Listing 3.1 in its main file. Essentially, it distinguishes between two cases:

1. If the current core has ID 0 (i.e., it is the master core), it will continue executing standard OpenMP code (i.e., containing `#pragma omp parallel` directives, etc.).
2. Otherwise, it will enter an infinite loop, where the core first goes to sleep and waits for an interrupt, and then participates in the parallel execution of the program by running its assigned task. Waking up other cores and assigning tasks is done by the master core when it encounters OpenMP constructs that require it.

Listing 3.1: Main Event Loop

```
1 uint32_t core_id = mempool_get_core_id();
2
3 if (core_id == 0) {
4   /* OpenMP Code */
5 } else {
6   while (1) {
```

```
7       mempool_wfi();
8       run_task(core_id);
9     }
10  }
```

Listing 3.2: run_task Implementation

```
1   typedef struct {
2     void (*fn)(void *);
3     void *data;
4     uint32_t nthreads;
5     uint32_t barrier;
6     uint8_t thread_pool[NUM_CORES];
7   } event_t;
8
9   void run_task(uint32_t core_id) {
10    if (event.thread_pool[core_id]) {
11      event.fn(event.data);__
12      atomic_add_fetch(&event.barrier, -1, __ATOMIC_SEQ_CST);
13    }
14  }
```

Listing 3.2 shows the implementation of `run_task` as well as `event_t`. Since this implementation assumes at most a single team (i.e., group of threads) working on a single task, it uses a global variable `event` of type `event_t` to store the task information.

The thread first checks if it is supposed to participate in the parallel execution by checking whether `event.thread_pool[core_id]` is set. If it is, it executes the function pointed to by `event.fn` and decrements the barrier, which makes sure that all threads have finished their work before starting the next task.

Listing 3.3: set_event Implementation

```
1   void set_event(void (*fn)(void *), void *data, uint32_t nthreads) {
2     uint32_t num_cores = mempool_get_core_count();
3     event.fn = fn;
4     event.data = data;
5     if (nthreads == 0) {
6       event.nthreads = num_cores;
7       event.barrier = num_cores;
8     } else {
9       event.nthreads = nthreads;
10      event.barrier = nthreads;
11    }
12
13    for (uint32_t i = 0; i < num_cores; i++) {
```

```
14      event.thread_pool[i] = (i < event.nthreads) ? 1 : 0;
15    }
16  }
```

A task is set when the master thread calls `set_event` (Listing 3.3) after encountering an OpenMP construct that requires it. The function pointer and its arguments are passed through `fn` and `data`, respectively. The number of threads that should participate in the parallel execution is passed through `nthreads`. If `nthreads` is 0, all cores will execute it. This is controlled by the `event.thread_pool` array.

3.1.2. Parallel Regions

OpenMP parallel constructs will be transformed into calls to `GOMP_parallel`, which is responsible for setting the current task, waking up all cores, and waiting for them to finish. Note that this function is only called by the master thread.

Listing 3.4: GOMP_parallel Implementation

```
1  void GOMP_parallel_start(void (*fn)(void *), void *data,
2                           unsigned int num_threads) {
3    set_event(fn, data, num_threads);
4    wake_up_all();
5    mempool_wfi();
6  }
7
8  void GOMP_parallel_end(void) {
9    uint32_t num_cores = mempool_get_core_count();
10   while (event.barrier > 0) {
11     mempool_wait(4 * num_cores);
12   }
13 }
14
15 void GOMP_parallel(void (*fn)(void *), void *data,
16                    unsigned int num_threads,
17                    unsigned int flags) {
18   uint32_t core_id = mempool_get_core_id();
19   gomp_new_work_share();
20   GOMP_parallel_start(fn, data, num_threads);
21   run_task(core_id);
22   GOMP_parallel_end();
23 }
```

3.1.3. Work Sharing Constructs

Work sharing constructs (e.g., `for`, `sections`, etc.) allow for threads in the current team to distribute the work to be done in a given parallel region. A global variable of type `work_t` is used for the bookkeeping related to these constructs and is initialized by calling `gomp_new_work_share` (Listing 3.4, Line 19).

Listing 3.5: work_t Struct

```
1  typedef struct {
2    int end;
3    int next;
4    int chunk_size;
5    int incr;
6
7    omp_lock_t lock;
8
9    // for single construct
10   uint32_t checkfirst;
11   uint32_t completed;
12   void *copyprivate;
13
14   // for critical construct
15   omp_lock_t critical_lock;
16
17   // for atomic construct
18   omp_lock_t atomic_lock;
19 } work_t;
```

As an example for a work sharing construct, Listing 3.6 shows the implementation of the functions required for a dynamically scheduled for loop construct (i.e., `#pragma omp parallel for schedule(dynamic)`).

`GOMP_parallel_loop_dynamic` (Line 60) is very similar to `GOMP_parallel`, but it also initializes the loop bounds and chunk size with `gomp_loop_init` (Line 66).

Each thread calls `GOMP_loop_dynamic_start` (Line 1) once and `GOMP_loop_dynamic_next` (Line 36) continuously afterwards until the loop is finished. The former checks if another thread already initialized the loop bounds and chunk size, and initializes them if that is not the case. Then, it atomically increments the start of the next chunk for the next thread to use and sets the bounds of the current chunk for the current thread. This is done again with every call to `GOMP_loop_dynamic_next`.

Listing 3.6: GOMP Dynamic For Loop Implementation

```
1  int GOMP_loop_dynamic_start(int start, int end, int incr,
2                              int chunk_size, int *istart,
3                              int *iend) {
4    int chunk, left;
5    int ret = 1;
6
7    if (gomp_work_share_start()) { // work returns locked
8      gomp_loop_init(start, end, incr, chunk_size);
9    }
10   gomp_hal_unlock(&works.lock);
11
12   chunk = chunk_size * incr;
13
14   start = __atomic_fetch_add(&works.next, chunk, __ATOMIC_SEQ_CST);
15
16   if (start >= works.end) {
17     ret = 0;
18   }
19
20   if (ret) {
21     left = works.end - start;
22
23     if (chunk > left) {
24       end = works.end;
25     } else {
26       end = start + chunk;
27     }
28   }
29
30   *istart = start;
31   *iend = end;
32
33   return ret;
34 }
35
36 int GOMP_loop_dynamic_next(int *istart, int *iend) {
37   int start, end, chunk, left;
38
39   chunk = works.chunk_size * works.incr;
40   start = __atomic_fetch_add(&works.next, chunk, __ATOMIC_SEQ_CST);
41
42   if (start >= works.end) {
```

```
43      return 0;
44    }
45
46    left = works.end - start;
47
48    if (chunk > left) {
49      end = works.end;
50    } else {
51      end = start + chunk;
52    }
53
54    *istart = start;
55    *iend = end;
56
57    return 1;
58  }
59
60  void GOMP_parallel_loop_dynamic(void (*fn)(void *), void *data,
61                                  unsigned num_threads, long start,
62                                  long end, long incr, long chunk_size) {
63    uint32_t core_id = mempool_get_core_id();
64
65    gomp_new_work_share();
66    gomp_loop_init(start, end, incr, chunk_size);
67
68    GOMP_parallel_start(fn, data, num_threads);
69    run_task(core_id);
70    GOMP_parallel_end();
71  }
72
73  void GOMP_loop_end() {
74    uint32_t core_id = mempool_get_core_id();
75    mempool_barrier_gomp(core_id, event.nthreads);
76  }
```

3.1.4. Barriers

To implement barriers, the runtime makes calls to `mempool_barrier` as implemented in MemPool's runtime library. It makes use of MemPool's interrupt capabilities to wake up all cores when the last core reaches the barrier. Note that cores that are not participating in the parallel execution will also be woken, up but they will immediately go back to sleep because of the check on Line 10 in Listing 3.2.

Listing 3.7: mempool_barrier Implementation

```
1  void mempool_barrier(uint32_t num_cores) {
2    // Increment the barrier counter
3    if ((num_cores - 1) == __atomic_fetch_add(&barrier, 1,
4                                              __ATOMIC_RELAXED)) {
5      __atomic_store_n(&barrier, 0, __ATOMIC_RELAXED);
6      __sync_synchronize(); // Full memory barrier
7      wake_up_all();
8    }
9    // Some threads have not reached the barrier --> Let's wait
10   // Clear the wake-up trigger for the last core reaching the barrier
11   // as well
12   mempool_wfi();
13 }
```

3.2. HERO LLVM Implementation

Chapter 4

Implementation

- Top down explanation of the implementation, including different iterations and why they were discarded
- Minor things like the testing scripts and main function wrapper

This chapter explains your contributions, be it algorithms, architectures, libraries, hardware, or others. As usual, follow a top-down approach and break the chapter into meaningful sections. It may even be necessary to split the chapter into multiple chapters (e.g., to separate architecture from implementation in a hardware design).

Derive a structure for this chapter that is meaningful for your report and discuss it with your advisors.

Chapter 5

Results

- Performance against existing runtime
- Discussing correctness tests

In this chapter, you want to show that your implementation meets the objectives. Start by briefly describing the type of results you have collected and how these results relate to the objectives.

5.1. Evaluation setup

Precisely describe your measurement/evaluation setup in a top-down manner. For example:

- If you used a development platform, which one and how was it configured?
- Which tools did you use?
- What input data set / program / signals / ... did you use? If your data set is very large, you should fully define and describe it in an appendix chapter and refer to that definition here using simple labels.
- If your implementation is configurable, which configuration(s) did you evaluate?

The information given here (with references to external work) should be sufficient to reproduce the setup you used for your measurements.

5.2. Other sections

Structure the rest of this chapter into sections. For example, each section could discuss a different figure of merit or a different part of your implementation. As usual, discuss structure and content in advance with your advisors.

5.3. Comparison to related work

Compare your results to those achieved by others working on similar problems. This can be done in a separate section or directly in the sections above.

5.4. Current limitations

Demonstrating that your implementation meets the objectives is usually done by showing a lower bound on a given figure of merit. Conversely, you should also illuminate the limitations of your implementation by showing upper bounds on these (or other appropriate) figures of merit. Critically examining your ideas and their implementation trying to find their limits is also part of your work!

Chapter 6

Conclusion and Future Work

- Interesting missing features (like nested parallelism and explicit tasks, as well as more clever ways to create teams) and how performance may be improved further.
- This can also be split into two chapters.

Draw your conclusions from the results and summarize your contributions. Point out aspects that need to be investigated further.

The conclusion can be structured inversely to the introduction: Summarize how the *evaluation* backs the *solution*, which solves the *problem*. Describe how your contributions improve the *situation* and what other (potentially newly discovered) problems have to be solved in the future.

Be concise: the conclusion normally fits on a single page and is rarely longer than two pages.

Appendix A

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

TASK ASSIGNMENT FOR A
SEMESTER THESIS AT THE DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

SPRING SEMESTER 2024

Diego de los Santos

Implementing an OpenMP runtime for MemPool in LLVM

March 18, 2024

Advisors: Samuel Riedel, ETZ J69.2, Tel. +41 44 632 65 69, sriedel@iis.ee.ethz.ch
Sergio Mazzola, ETZ J76.2, Tel. +41 44 632 81 49, smazzola@iis.ee.ethz.ch

Professor: Prof. Dr. L. Benini

Handout: March 18, 2024
Due: July 01, 2024

1 Introduction

Striving for high image quality, even on mobile devices, has led to an explosion in the pixel count of smartphone cameras over the last decade [1]. These image sensors, boasting tens of millions of pixels, create a massive amount of data to be processed on a tight power envelope as quickly as possible. Computational photography, computer vision, augmented reality, and machine learning are only a few of the possible applications. Highly specialized image signal processors (ISPs) harness the features of these workloads, which are highly parallelizable, to meet timing and power constraints. Google offers a prominent example: the Pixel Visual Core, employed for image processing in Google smartphones [2].

At ETH, we are developing our own manycore system that can be used as an ISP called MemPool [3]. It boasts 256 area-optimized 32-bit Snitch [4]. Snitch, developed at ETH as well, implements the RISC-V instruction set architecture (ISA), which is an open ISA targeting modularity and scalability [5]. Despite its size, MemPool gives all 256 cores low-latency access to the shared L1 memory, with a maximum latency of only five cycles when no contention occurs. This implements efficient communication among all cores, making MemPool suitable for various workload domains and easy to program.

A common framework for parallel programming is OpenMP [6]. It implements a task-based or fork-join approach to parallelization through annotations in the C code. The compiler will generate routines and runtime calls to distribute the parallel execution across the specific platform correctly. To this end, each system supporting OpenMP needs to implement a corresponding OpenMP Runtime. Specifically, there are two major runtimes, the GOMP and KMP runtime for GCC and LLVM, respectively. MemPool has minimal support for the GOMP runtime. However, since LLVM has become the standard compiler infrastructure for our research group, a KMP runtime to support LLVM is crucial.

Goal The goal of this thesis is to implement the KMP runtime for MemPool. To this end, the student will first identify the most important OpenMP functions for MemPool and then implement the runtime in conjunction with functional tests for verification. For benchmarking, the student will implement and evaluate digital signal processing (DSP) kernels and potentially full applications from the image processing or communications domain.

2 Milestones

The following are the milestones that we expect to achieve throughout the project:

- Become familiar with MemPool and the OpenMP runtime.

- Investigate the most useful features of the OpenMP standard.
- Implement the key features of the KMP runtime for MemPool and functionally verify them.
- Benchmark the newly implemented runtime on several DSP kernels.

2.1 Stretch Goals

Should the above milestones be reached earlier than expected and the student is motivated to do further work, we propose the following stretch goals to aim for:

- Advance the KMP implementation by implementing features beyond the basic set determined during the initial project phase.
- Build a full demonstrator application using OpenMP. The application can be chosen during the project, but examples include an HDR image processing pipeline, a ray tracing algorithm, or a 5G communication pipeline.
- Tune your benchmarks or demonstrator application for Heartstream, a 64-core chip implementing the latest MemPool architecture.

3 Project Realization

3.1 Time Schedule

The time schedule presented in Table 1 is merely a proposition; it is primarily intended as a reference and an estimation of the time required for each required step.

Project phase	**Time estimate**
Familiarization with MemPool and OpenMP to determine key features	2 weeks
Implementation and verification of key features	4 weeks
Evaluation of your work and implementation of DSP kernels in OpenMP	4 weeks
Write report	2 weeks
Prepare presentation	2 weeks

Table 1: Proposed time schedule and investment

3.2 Meetings

Weekly meetings will be held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the student's and the assistants' schedule. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

3.3 Weekly Reports

The student is advised, but not required, to write a weekly report at the end of each week and to send it to his advisors. The idea of the weekly report is to briefly summarize the work, progress and any findings made during the week, to plan the actions for the next week, and to bring up open questions and points. The weekly report is also an important means for the student to get a goal-oriented attitude to work.

3.4 Coding Guidelines

HDL Code Style Adapting a consistent code style is one of the most important steps in order to make your code easy to understand. If signals, processes, and modules are always named consistently, any inconsistency can be detected more easily. Moreover, if a design group shares the same naming and formatting conventions, all members immediately *feel at home* with each other's code. At IIS, we use lowRISC's style guide for SystemVerilog HDL: `https://github.com/lowRISC/style-guides/`.

Software Code Style We generally suggest that you use style guides or code formatters provided by the language's developers or community. For example, we recommend LLVM's or Google's code styles with `clang-format` for C/C++, PEP-8 and `pylint` for Python, and the official style guide with `rustfmt` for Rust.

Version Control Even in the context of a student project, keeping a precise history of changes is *essential* to a maintainable codebase. You may also need to collaborate with others, adopt their changes to existing code, or work on different versions of your code concurrently. For all of these purposes, we heavily use *Git* as a version control system at IIS. If you have no previous experience with Git, we *strongly* advise you to familiarize yourself with the basic Git workflow before you start your project.

3.5 Report

Documentation is an important and often overlooked aspect of engineering. A final report has to be completed within this project.

The common language of engineering is de facto English. Therefore, the final report of the work is preferred to be written in English.

Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Inkscape or any other vector drawing software (for block diagrams) is strongly encouraged by the IIS staff.

If you write the report in LaTeX, we offer an instructive, ready-to-use template, which can be forked from the Git repository at `https://iis-git.ee.ethz.ch/akurth/iisreport`.

Final Report The final report has to be presented at the end of the project and a digital copy needs to be handed in and remain property of the IIS. Note that this task description is part of your report and has to be attached to your final report.

3.6 Presentation

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

4 Deliverables

In order to complete the project successfully, the following deliverables have to be submitted at the end of the work:

- Final report incl. presentation slides
- Source code and documentation for all developed software and hardware
- Testsuites (software) and testbenches (hardware)
- Synthesis and implementation scripts, results, and reports

References

[1] S. Skafisk, *This is How Smartphone Cameras Have Improved Over Time*, 2017 (accessed August 18, 2020). [Online]. Available: https://petapixel.com/2017/06/16/smartphone-cameras-improved-time/

[2] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham, "Pixel Visual Core: Google's fully programmable image, vision and AI processor for mobile devices," in *2018 IEEE Hot Chips 30 Symposium (HC30)*. Cupertino, US: IEEE Technical Committee on Microprocessors and Microcomputers, Aug. 2018.

[3] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "MemPool: A scalable manycore architecture with a low-latency shared L1 memory," *IEEE Transactions on Computers*, vol. 72, no. 12, pp. 3561–3575, 2023.

[4] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Trans. Comput.*, vol. 70, no. 11, pp. 1845–1860, Feb. 2021.

[5] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, A. Waterman, Y. Lee, and D. Patterson, "The RISC-V instruction set manual," 2014. [Online]. Available: https://github.com/riscv/riscv-isa-manual

[6] O. A. R. Board, "OpenMP Application Programming Interface," 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Zurich, March 18, 2024 Prof. Dr. L. Benini

List of Acronyms

AXI . . . . . .Advanced eXtensible Interface

DMA . . . . . .Direct Memory Access

GCC . . . . . .GNU Compiler Collection

List of Figures

List of Tables

Bibliography

[1] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "Mempool: A scalable manycore architecture with a low-latency shared l1 memory," *IEEE Transactions on Computers*, vol. 72, no. 12, p. 3561–3575, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1109/TC.2023.3307796

[2] M. Bertuletti, S. Riedel, Y. Zhang, A. Vanelli-Coralli, and L. Benini, "Fast shared-memory barrier synchronization for a 1024-cores risc-v many-core cluster," 2023.

[3] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, p. 1845–1860, Nov. 2021. [Online]. Available: http://dx.doi.org/10.1109/TC.2020.3027900

[4] *OpenMP: The OpenMP API specification for parallel programming*. [Online]. Available: https://ww.openmp.org

[5] W. Qiu, "Design and verification of llvm openmp runtime library on pulp/hero," 2019.