Massively Parallel Systems Lab Assignments

Sohan Lal

sohan.lal@tuhh.de

Massively Parallel Systems Group

Technische Universität Hamburg

13. Januar 2023

This document describes the lab assignments of the massively parallel systems: architecture and programming module. We plan to do 3 to 4 assignments. The first assignment consists of designing a cache simulator which we will extend to model a cache coherence protocol.

Each assignment has deliverables. The deliverables have to be submitted via the StudIP. A link will be provided to upload the assignments. The assignments can be performed in groups of 2-3 students. Please submit only one assignment per group.

Deadlines of the assignments are:

Assignment 1: Cache Design - Task 1	**23.11.2022**
Assignment 1: Cache Coherence - Task 2	**14.12.2022**

Submitting the deliverables past the deadline reduces your maximum points for the respective assignment. You can submit the assignments until midnight on the day of the deadline.

The cache coherence assignments are based on the lab assignments of the course *Advances in Computer Architecture*[1] given by Prof. C. R. Jesshope, University of Amsterdam.

Introduction

In the cache coherence part of the lab, you will use SystemC to build a simulator of a L1 data cache and various implementations of a cache coherence protocols and evaluate their performance. The simulator will be driven using trace files which will be provided. For information about cache coherence protocols and memory, please refer to the lecture slides and see Appendix A in file doc/Appendices.pdf.

The framework (part1_student_srcs.zip) for this part of the lab can be downloaded from the StudIP page. Download it to your local home directory, and extract it. This framework contains all documentation, the helper library with supporting functions for managing tracefiles and statistics, the tracefiles, and a Makefile to automatically compile your assignments. Using the Makefile, the contents of each directory under src/ are compiled automatically as a separate target. It already includes directories and the code of the Tutorial, as well as a piece of example code for Task 1.

Trace files are loaded through the functions provided by the helper library (aca2009.h and aca2009.cpp), which is in the provided framework. Detailed documentation for these functions and classes is provided in Appendix C in file doc/Appendices.pdf. The example code for Task 1 also contains all these functions and should make it self-explanatory. Note that, although later on simulations with up to eight processors with caches are required, it is easier if you start with a single processor organization. Then, extend it to support multiple processors. Make sure that the number of processors and caches in your simulation is not statically defined in your code, but uses the number of processors read from the trace file.

Trace files

There are 3 kinds of trace files available to you: random, debug and FFT. All versions exist for 1, 2, 4 and 8 processors. The random (rnd_pX.trf) trace files have the processors read or write memory randomly in a window in memory that is moving. The debug (dbg_pX.trf) trace files are a short version of the random trace files, suited for debugging your simulation. The FFT (fft_16_pX.trf) trace files are the result of the execution of a FFT on a 64K array and can be used, along with the random trace files, to generate results for your reports. All reads and writes are byte-addressed, word-aligned accesses.

Submission and reports

Students can work in groups of 2-3 persons. The deliverables need to be submitted via StudIP and should include:

- the source code (zip format).

[1] http://staff.science.uva.nl/ mwvantol/teaching/ACA2009/

- a brief report (in pdf, with cover page indicating group members and assignment name) at the end of each assignment with your results.

General guidelines:

- Your source code should compile and run without any modifications on the Linux machine (access will be provided), using the provided Makefile.
- The first command line argument your program accepts is the name of the tracefile. (this happens automatically when the init_tracefile function is used).
- Hit/Miss rate statistics should be gathered with the provided functions, and should be printed with the stats_print function after the simulation ends.
- Your simulations should always terminate and exit without any errors.
- Supporting a different number of processors in your simulation should not require recompilation.

1.	Massively Parallel Systems Group **Massively Parallel Systems: Architecture and Programming**	WS 22/23

Assignment start: 03.11.2022 Submission deadline: 23.11.2022

Assignment 1 - SystemC Intro and L1 Cache **25 Points**

In this task you will build a simulator for L1 D-cache. The cache size is 32 KB, 8-way set-associative and cache line size is 32 bytes. The simulator must be driven with the single processor trace files. Assume a memory latency of 100 cycles, and single cycle cache latency. Use least-recently-used, write-back replacement strategy.

Notes:

- Read doc/SystemC_tutorial.pdf file, and practice the examples mentioned there.
- You can use the example code provided for Task 1, which is based on the tutorial, as a guide and modify the Memory module so that instead of RAM, it simulates a set-associative data cache.
- As we only simulate accesses to memory, it is not necessary to simulate any actual contents inside the cache. Furthermore, you do not need to simulate the actual communications with the memory, just the delay.
- We expect that when running your code, it will generate a waveform file, which includes the important signals in your design, for example: clock, address, set number, line number in case of a hit, line number to be replaced in case of a miss, hit/miss, read/write, etc. Please make a SNAPSHOT of the wave file and include it in your report
- In addition to your source code, please also submit a brief report, not more than one page, including your calculations of the tag/index/offset bits, your results that are printed at the end of run, in addition to the total run time, and average memory access time.
- In the report, include your observations about the results such as reasons for variation in hit rates across different benchmarks.

2.	Massively Parallel Systems Group **Massively Parallel Systems: Architecture and Programming**	WS 22/23

Assignment start: 01.12.2022 Submission deadline: 15.12.2022

Assignment 2 - Valid-Invalid Protocol 25 Points

Extending the assignment 1, you are supposed to simulate a multiprocessing system with a shared memory architecture. Your simulation should be able to support multiple processors, all working in parallel. Each of the processors is associated with a local cache, and all cache modules are connected to a single bus. A main memory is also connected to the bus, where all the correct data is supposed to be stored, although, again, the memory and its contents does not have to be simulated. Assume a pipelined memory which can handle a request from each processor in parallel with the same latency.

In order to make cache data coherent within the system, implement a bus snooping protocol. This means that each cache controller connected to the bus receives every memory request on the bus made by other caches, and makes the proper modification on its local cache line state. In this task, you should implement the simplest VALID-INVALID protocol. The cache line state transition diagram for this protocol is shown in Figure 1.

Using your simulator you must perform experiments with different numbers of processors (1 to 8 processors) using the different trace files for each case.

For each experiment your program should print and you should also record (but not limited to) the following data for your report:

1. Per cache, please report the following in tabular format: (use the framework functions)
   #Reads RHit RMiss #Writes WHit WMiss Hitrate

2. Please also add two additional rows for total and average statistics.
3. Per cache, while snooping on the bus, count and report the total number of read/write operations on the bus, in which the desired address is found in the snooping cache (probe read hits and probe write hits).
4. Average memory access time
5. Number of bus reads, bus writes, and total accesses (reads+writes).
6. We expect that when running your code, it will generate a waveform file, which includes the important signals in your design and include a SNAPSHOT in the report.

Hints:

- Refer to the lecture notes for details about the Valid-Invalid Protocol. According to this protocol, your cache is write-through, write-allocate cache.
- Some help on implementing a Bus is provided in Appendix B in file doc/Appendices.pdf. You could use or modify these implementations for your simulation.
- The bus can only serve one request at any certain time. Thus if one cache occupies the bus, the other cache has to wait for the current operation to complete before it can utilize the bus. The cache does not occupy the bus when it waits for the memory to respond. The responses from memory should have priority on the bus.
- The cache must not only be able to serve the request from memory, it also has to observe all the

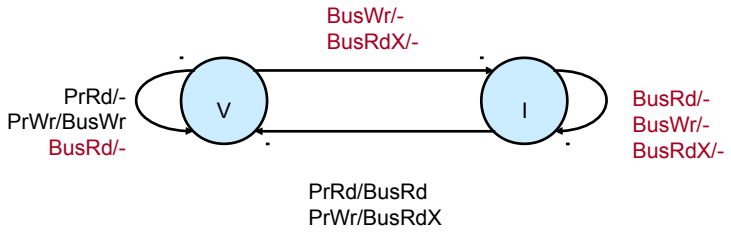

Abbildung 1: VALID – INVALID Cache Line State Transition Diagram

transactions happening on the shared bus, and make proper adjustment on its own corresponding cache line states as described in Figure 1.

- Keep in mind that in Assignment 3, you also need to implement cache to cache transfers for when a CPU reads from or writes to a location for which another cache holds data which was not written back to memory yet.

Pleae keep in mind:

- Late submission will result in deducting your final grade.
- There are some hints about implementing the LRU algorithm in the LRU.pdf file in the "doc"folder.
- Note the relationship between sc_buffer and value_changed_event(), and how it differs when using sc_signal instead.
- Read the general guidelines above.

3.	Massively Parallel Systems Group **Massively Parallel Systems: Architecture and Programming**	WS 22/23

Assignment start: 15.12.2022 Submission deadline: 12.01.2023

Assignment 3 - GPU Password Cracker 25 Points

In this assignment a GPU Password Cracker should be developed. A list of MD5 hashes of lower case letter only passwords is provided and your program should brute force through all possible passwords to map the hashes back to the passwords. No salt is used. A CPU version of the password cracker is provided as md5.c.

- Parallelize the password cracker using CUDA.
  - Allocate memory on the GPU for the hashes and found passwords.
  - Implement a kernel, where each threads calculates the hash of a password and compares it to all provided hashes.
  - If a match was found, store the match in the GPU DRAM in an appropriate data structure.
  - Transfer the results back and output all found passwords on the standard output
- Test your kernel using 3 letter passwords for shorter runtimes.
- Analyze the performance using GPU profiler (optional).
- Report the results for 3-, 4-, and 6-letter passwords and explain the design of your kernel(half a page), focus on design decisions relevant to performance
- Also submit the source code of your CUDA password cracker.
  - Measure GPU execution time and compare to CPU execution time for 6-letter passwords.
  - The lab machine has a NVIDIA GPU and CUDA installed. The nvcc, which is a NVIDIA compiler to compile CUDA programs is installed at */usr/local/cuda/bin/nvcc*. I created one user per group. Please use your group username and password to login.

3.	Massively Parallel Systems Group **Massively Parallel Systems: Architecture and Programming**	WS 22/23

Assignment start: 13.01.2023 Submission deadline: 01.02.2023

Assignment 4 - OpenMP Multithreading 25 Points

In the final lab exercise you will build on the multithreading concepts acquired in the lectures and more specifically you will get hands on experience using OpenMP to parallelize sequential programs. OpenMP is a parallel programming model for shared-memory multi-processor systems. It allows a programmer to specify a portion of code that should be executed in parallel using compiler directives. OpenMP implementation of multithreading uses the fork-join model, whereby a master thread forks a number of slave threads dividing work among them. The threads then run concurrently, with the runtime environment allocating threads to different processors core or hardware threads. In this assignment we provide you with sequential code implementing the LU decomposition algorithm, https://en.wikipedia.org/wiki/LU_decomposition. Your task is to parallelize this sequential program with the help of OpenMP.

The sources for the assignment are available on the course page (StudIP). You can perform the experiments on your local machine but for consistency we will be checking your assignments on an 12th Gen Intel(R) Core(TM) i9-12900K with 16 cores (24 threads) using GCC compiler version 12.1.0 running on Ubuntu1 22.04.

The lud_seq.cpp file contains our sequential baseline implementation. As part of this assignment, you have to perform the following:

- Modify the lud_par.cpp file by parallelizing the sequential code using OpenMP directives.
- Modify the lud_opt.cpp file to produce your best optimized version. You can make any changes to the the algorithm, use any OpenMP directives, number of threads etc.
- Produce a graph by varying the number of OpenMP threads and showing the speedup of your lud_opt.cpp and lud_par.cpp over lud_seq.cpp. Use the 1024.dat input file.
- Write a small report analyzing the produced figure. Also include brief explanations of the modifications you made to both lud_par.cpp and lud_opt.cpp.

You can test your implementation on the lab machine that we will use for grading your submission. The machine is accessible via ssh using: ssh groupnumber@134.28.76.11 through the ssh.rz.tuhh.de tunnel. I created one user per group. Please use your group username and password to login.

Your deliverable should contains the source files (lud_par.cpp, lud_opt.cpp) and your brief report. The assignment will be evaluated by the performance of your implementations and explanation.