Programming Assignment II

Simulator-mein

CS683: Advanced Computer Architecture, Autumn 2025
Computer Science and Engineering
Indian Institute of Technology, Bombay
CASPER group: https://casper-iitb.github.io/

** Disclaimer: The memes included in this document are intended solely for fun learning. They are not meant to offend, mislead, or be taken as factual information. Please enjoy them in the spirit of fun learning.*

You are still in CS683, kudos to you! Now moving on to programming assignment ||, Prof. Biswa has covered key microarchitecture concepts, including cache hierarchy, cache management policies, and prefetching mechanisms. Let's proceed from theory to actually implementing and optimising these mechanisms.

In this assignment, you will use the ChampSim simulator to implement/ optimise these techniques and validate their effectiveness by using the various traces of benchmarks provided below.

PA2 GitHub repository: *ChampSim_PA2*
Trace files: Traces (Use IITB Account)

ChampSim tutorial: Video

A friendly reminder: If you think copying the assignment is a clever shortcut, think again. It's not only easy to spot, but also a great way to miss out on the chance to actually learn something. Why not impress us with your own work?

NOTE:

- You need a machine with an AMD or Intel processor.
- Please read the README carefully. If anyone raises a doubt that has already been mentioned in the README, **two points will be deducted.**

The assignment is divided into three major tasks. The task structure points are shown below:

Main Tasks		
Task 1	Offset-based data prefetching	6 points
Task 2	Exclusive cache hierarchy	6 points
Task 3	Data prefetching for exclusive hierarchy	3 points
Total		15 points

Task 1

Offset-based Data Prefetching

Data prefetching is a technique used in modern processors to hide memory access latency by predicting and loading data into the cache before the CPU requests it. This helps avoid cache misses, which are expensive in terms of performance. You have studied various prefetchers, including next-line, IP-stride, and stream prefetchers. But various applications lack IP-based reuse or streaming behaviour. Therefore, the architecture community developed offset-based prefetchers, which have been implemented in most modern processors.

These prefetchers analyse global memory access patterns, i.e., they observe all cache accesses and learn the offsets (the difference between consecutive memory addresses). Typically, offset-based prefetchers reside at the L2 or L3 cache, as they do not require IP addresses for learning. These prefetchers typically prefetch data within an OS page, as they only observe physical addresses at the L2 or L3 level. For a better understanding of offset prefetchers, refer to Best Offset Prefetcher(BOP).

Your task is to implement a **region-based offset prefetcher at the L2 cache**:

- Where the region corresponds to an **OS page**.
- The design should utilise a table that is indexed by a **signature** that uniquely represents each page. (Hint: Think of IP-stride prefetcher)
- The table should store the **offsets** observed within that page. Each table entry should maintain up to **five offsets**, tracking their frequencies to prioritise which offsets should be used for prefetching.
- Ensure prefetch requests remain within the same page region.
- Implement an **accuracy checker as a part of your prefetcher** that evaluates the usefulness of prefetches and determines which offset(s) to fetch and to what level of aggressiveness in terms of prefetch degree.
- The overall objective is to design a prefetcher that dynamically adapts to the memory access patterns within each page, balancing coverage, accuracy, and overhead while avoiding useless prefetches.

Steps to follow:

➔ Implement an L2-based offset prefetcher.
Look at *prefetchers/offset_prefetcher.l2c_pref* file.

➔ Hint: We have provided the IP-stride prefetcher file in the prefetchers directory for your reference.

➔ Follow the README in the ChampSim repository to build and execute the binary with the provided traces.

- ◆ Execute for: warmup_instructions=25M, simulation_instructions=25M.

➔ Note: Update *cache_hierarchies/non_inclusive_cache.cc* for updates in cache if required, not *src/cache.cc*.

➔ Vary the table sizes: 32, 64, 128 entries.

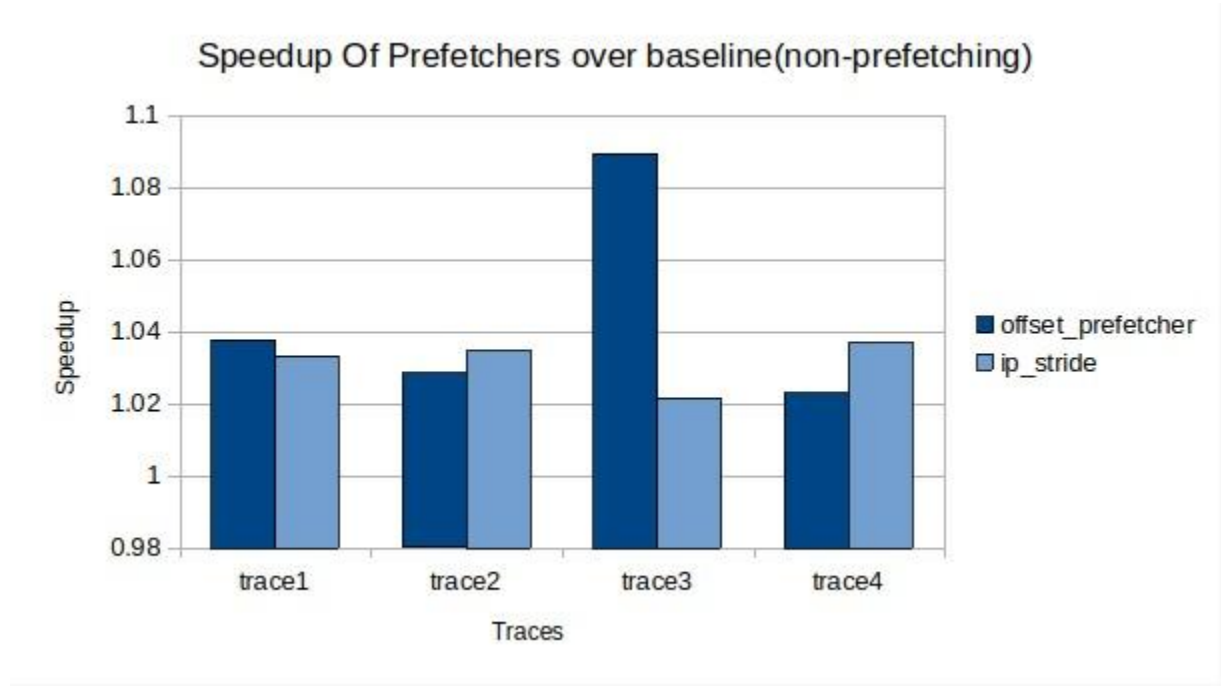
➔ Report the IPC, speedup, and L2 MPKI values in tabular form and plot the overall speedup.

Note:

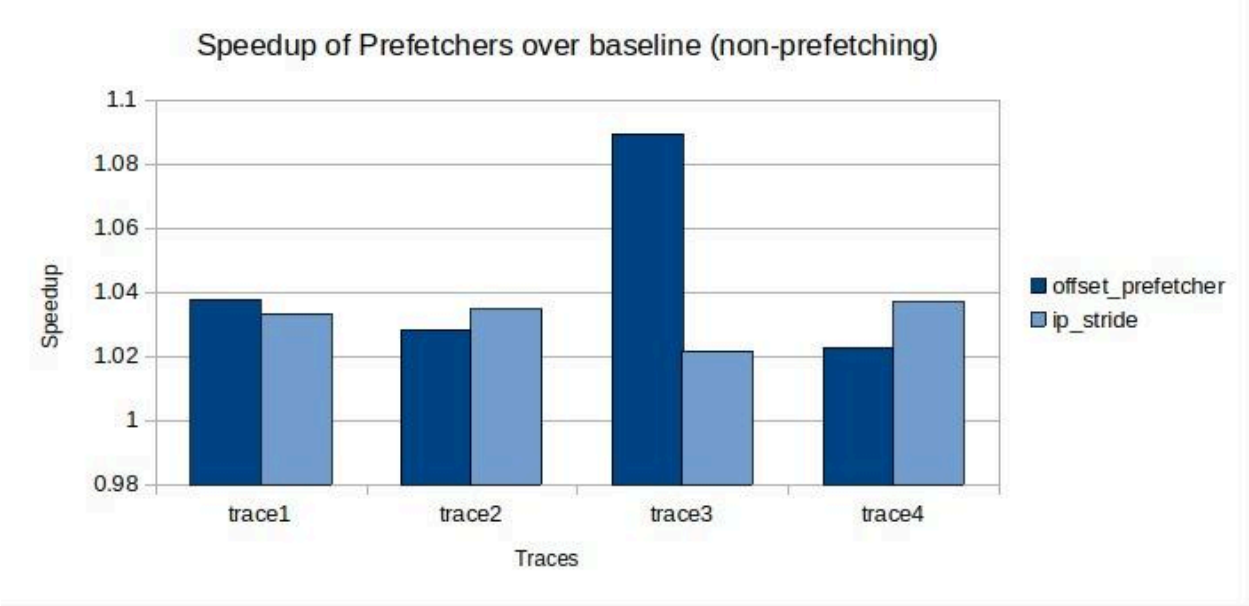
Speedup = $\frac{IPC\ with\ Prefetcher}{IPC\ of\ Baseline(no\ prefetching)}$

IP_Stride				
Traces	**IPC**	**Baseline over Non Inclusive with no prefetching**	**L2 MPKI**	**Speedup over Baseline**
trace1	0.54648	0.529026	40.3407	1.032992707
trace2	0.478241	0.462268	47.7267	1.034553549
trace3	0.852268	0.834309	22.1965	1.021525598
trace4	0.47923	0.462276	47.5704	1.03667506

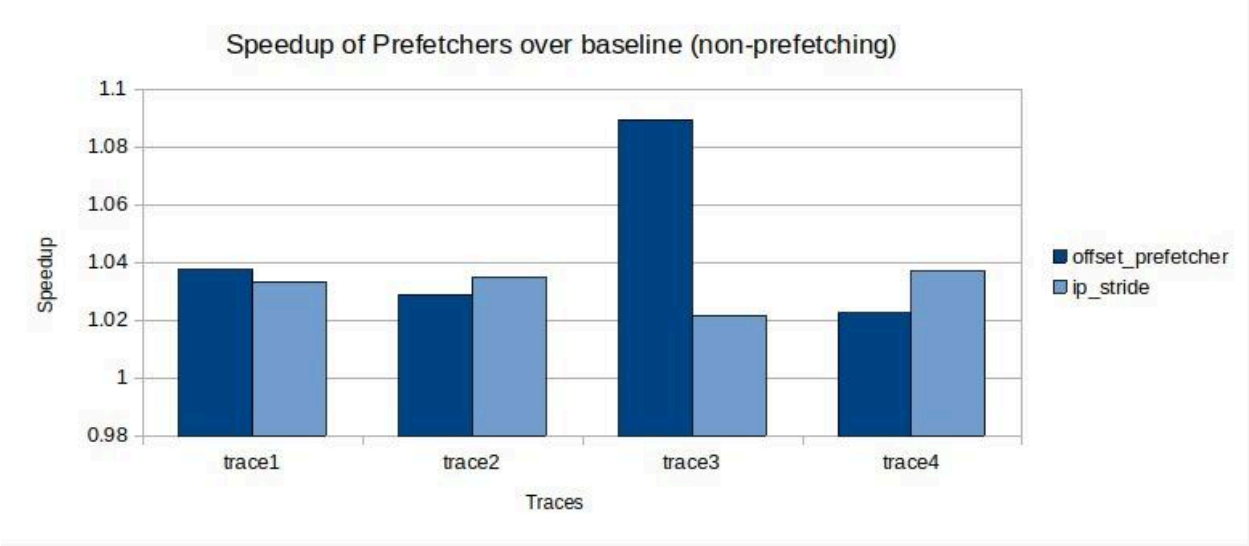
Offset Prefetcher at L2D				
Table Size : 256				
Traces	**IPC**	**Baseline over Non Inclusive with no prefetching**	**L2 MPKI**	**Speedup over Baseline**
trace1	0.54888	0.529026	40.6145	1.037529346
trace2	0.475348	0.462268	47.7591	1.028295275
trace3	0.908659	0.834309	22.0977	1.089115663
trace4	0.472857	0.462276	47.667	1.022888924



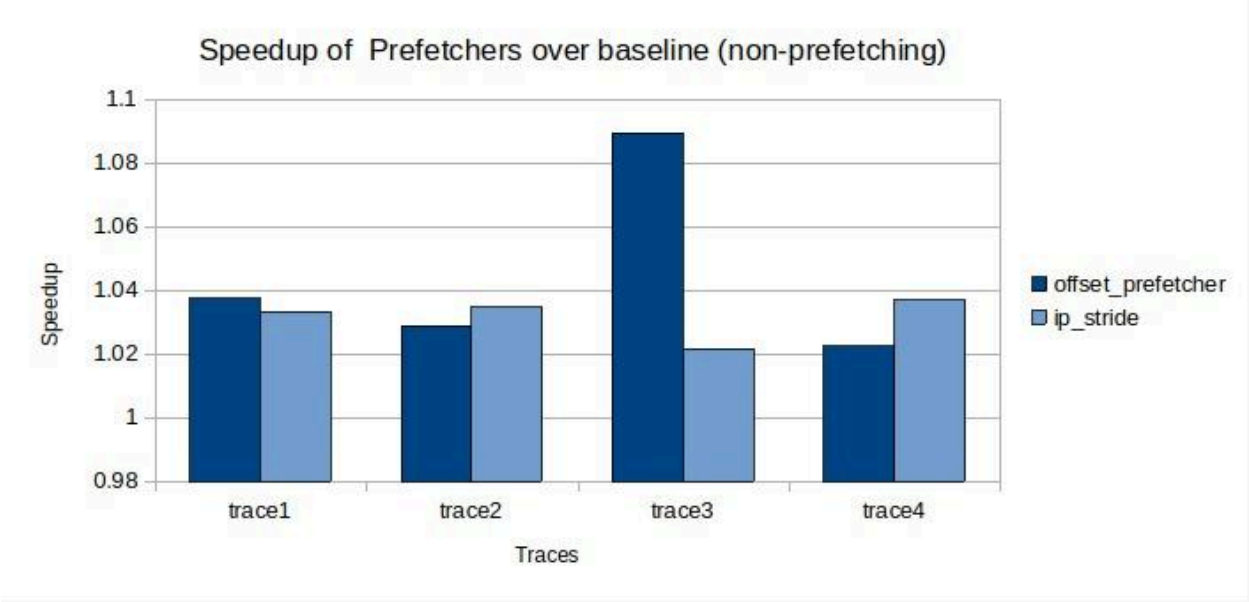
Offset Prefetcher at L2D				
Table Size : 128				
Traces	**IPC**	**Baseline over Non Inclusive with no prefetching**	**L2 MPKI**	**Speedup over Baseline**
trace1	0.54888	0.529026	40.6145	1.037529346
trace2	0.475182	0.462268	47.7591	1.027936176
trace3	0.908659	0.834309	22.0977	1.089115663
trace4	0.472657	0.462276	47.6637	1.022456282



Offset Prefetcher at L2D				
Table Size : 64				
Traces	**IPC**	**Baseline over Non Inclusive with no prefetching**	**L2 MPKI**	**Speedup over Baseline**
trace1	0.548741	0.529026	40.6091	1.037266599
trace2	0.475462	0.462268	47.7569	1.028541885
trace3	0.908659	0.834309	22.0977	1.089115663
trace4	0.472594	0.462276	47.6648	1.022319999



Offset Prefetcher at L2D				
Table Size : 32				
Traces	**IPC**	**Baseline over Non Inclusive with no prefetching**	**L2 MPKI**	**Speedup over Baseline**
trace1	0.548741	0.529026	40.6091	1.037266599
trace2	0.475462	0.462268	47.7569	1.028541885
trace3	0.908659	0.834309	22.0977	1.089115663
trace4	0.472594	0.462276	47.6648	1.022319999



Explanation:

The prefetcher mainly works using a few core data structures that help it track and predict memory accesses. The PageTrackerTable is a fixed-size array that acts as the main memory for the prefetcher. When it becomes full, a Least Recently Used (LRU) policy is used to remove old entries. Each entry in this table is a PageAccessTracker, which stores all the important information of one page. It has a page_address_tag to identify the page, a last_address to store the last accessed address, and an offsets[5] array that keeps the recent offsets and how many times they occur. Two counters, prefetches_issued and prefetches_useful, are also maintained to track how the prefetcher is performing. The OffsetFrequency structure is simple — it connects an offset value (the stride) with how often it appears.

In the operational logic, every time there is an access to the L2 cache, the prefetcher runs. It first finds which page the address belongs to and checks if that page is already being tracked. If not, called a Tracker Miss, one old entry is replaced using the LRU policy and starts tracking the new page. No prefetch is done at this time. If a Tracker Hit occurs, meaning the page is already being tracked, then the main prefetch logic starts. The offset between the current and last address is calculated. If it is not zero, the offset table is updated. The prefetcher then checks its accuracy by calculating useful / issued and sets the prefetch degree (like 1, 2, or 4) depending on that. This adjustment happens only after a small warm-up phase (around 20 issued prefetches). Then the best_offset (the one with the highest frequency) is used to create prefetch addresses using

prefetch_address = current_address + (best_offset * i) while making sure it doesn’t cross the page boundary.

In the feedback mechanism, every prefetch made is marked with a tag called PREFETCH_METADATA_TAG. The cache_fill function checks all the data filled into the L2 cache. A prefetch is counted as useful only when a real CPU request (!prefetch) hits a cache line that was brought by a tagged prefetch. When that happens, the prefetches_useful counter of that page tracker is increased. This feedback helps the prefetcher understand which prefetches were actually useful and improves its accuracy for future accesses.

Task 2

Exclusive Cache Hierarchy

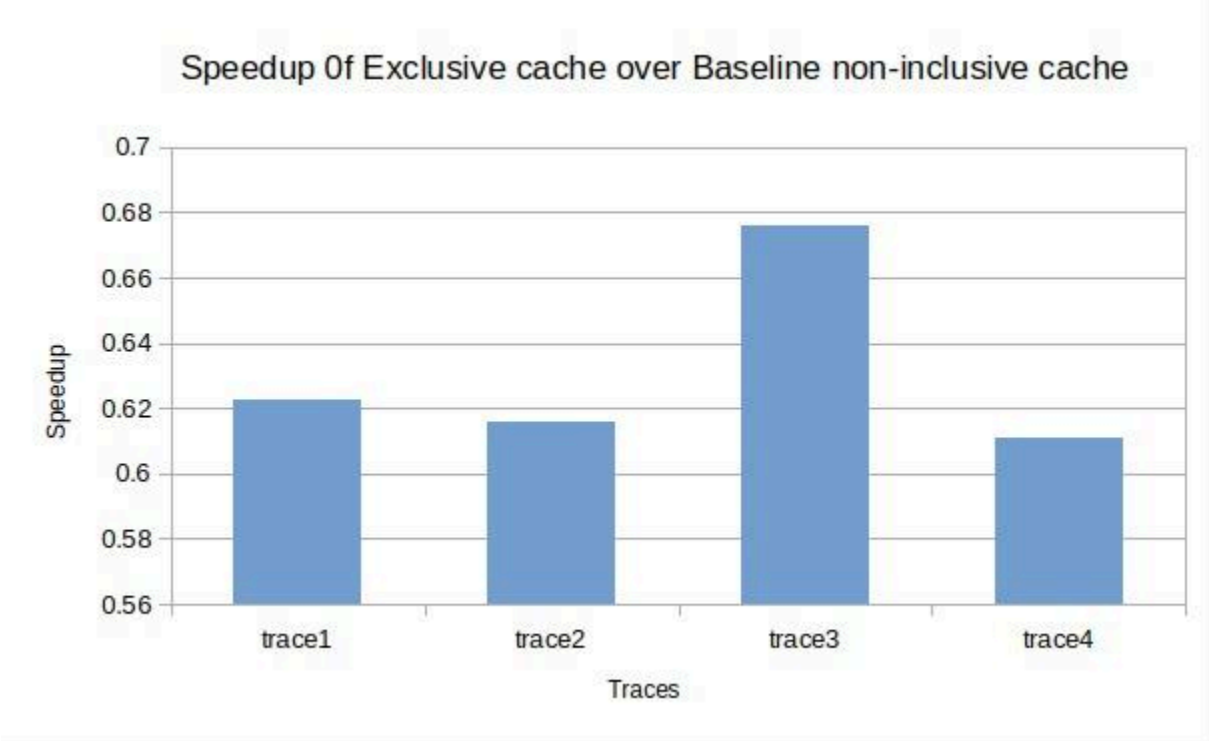
An **exclusive cache hierarchy** is a design where each cache level stores mutually exclusive data, i.e. a cache block having a particular address resides in only one level of the cache hierarchy. This avoids duplication across levels, effectively increasing the total usable cache size. When a cache block is evicted from a lower level (e.g. L1), it's moved to the next higher level(e.g. L2). However, when data moves from a higher level to a lower level, the cache block is removed from the higher level. While it effectively allows us to store more cache lines, it requires more complex management policies compared to non-inclusive hierarchies.
Your task is to implement an exclusive cache hierarchy in the ChampSim simulator (non-inclusive by default) and argue why an exclusive hierarchy is superior to or not than the traditional non-inclusive hierarchy policy.

Steps to follow:

- ➔ Implement an exclusive cache hierarchy across all three cache levels.
  - ◆ Refer and update *cache_hierarchies/exclusive_cache.cc* and replacement policy(LRU) if required.
- ➔ Follow the README of the provided ChampSim repository to build and execute the binary with the provided traces.
  - ◆ Use *build_champsim_exclusive.sh* to build.
  - ◆ Execute for: warmup_instructions=25M, simulation_instructions=25M
- ➔ Report the IPC, MPKI (L1D, L2, LLC), and speedup (over non-inclusive) values in a tabular format, along with the corresponding speedup plot.

Non-inclusive no Prefetching				
Traces	**IPC**	**L1 MPKI**	**L2 MPKI**	**LLC MPKI**
trace1	0.529026	66.0504	40.3453	40.1097
trace2	0.462268	78.022	47.6893	47.38
trace3	0.834309	35.7547	22.197	21.7126
trace4	0.462276	77.7236	47.562	47.1961

Exclusive					Speedup of Exclusive cache over baseline non-inclusive cache
Traces	IPC	L1 MPKI	L2 MPKI	LLC MPKI	
trace1	0.329315	66.0503	112.561	128.452	0.622493034
trace2	0.284645	78.022	133.415	150.205	0.615757526
trace3	0.563952	35.7547	61.1847	68.1966	0.675950997
trace4	0.282452	77.7243	132.837	151.03	0.611002951



Explanation:

handle_fill() function is being used to fill the data from MSHR fill entry to the cache.We invalidated the address from the entire cache hierarchy before filling(calling fill_cache()). This way we ensure that after the block is filled to cache its not present on any other level thus maintaining exclusivity.

handle_writeback() function is being used to write the data which was dirty or has been evicted to next level.We used the same logic as above, whenever the victim (may or may not be dirty) is filled at level after it, before filling(calling fill_cache()) we invalidate that address from entire cache hierarchy. Thus we ensure that after a block is placed into any level of cache , it's always present in only one level of cache.

Also we made changes so that whenever a block is evicted, it is added into the WB queue, not only when it's dirty but also when it's not dirty.So evicted blocks go to the next level.

Task 3

Data Prefetching for Exclusive Cache Hierarchy

In this task, you need to extend your L2-based offset prefetcher to work with an exclusive cache hierarchy. As stated in task 2, in an exclusive cache hierarchy, the higher-level cache acts as a victim cache for lines evicted from the lower-level cache. As a result, the accesses observed by your L2 prefetcher will include load requests raised due to L1D misses, as well as the writeback of all evicted lines (both dirty and clean). Update your L2 prefetcher to observe and learn from these additional accesses due to writeback lines.

The changes mentioned above are not a complete set of changes that should be made to work with an exclusive hierarchy. You should further optimise this design.

Steps to follow:

- ➔ Implement an L2-based offset prefetcher
  prefetchers/offset_prefetcher_exclusive.l2c_pref file.
  Make changes in cache_hierarchies/exclusive_cache.cc to observe writeback requests for prefetcher training.
- ➔ Follow the README in the ChampSim Repository to build and execute the binary with the provided traces.
  - ◆ Execute for: warmup_instructions=25M, simulation_instructions=25M.
- ➔ Experiment for varying table sizes: 32, 64, 128 entries.

➔ Report the IPC, speedup, and L2 MPKI values in tabular form and plot the overall speedup.

➔ Note: Plot the two different speedup graphs with two different baselines: one with a non-inclusive cache (without prefetching) and the other with an exclusive cache (without prefetching).

Submission Instructions

- You have to push all the required deliverables to the GitHub repository with proper commit history. Unclear/last-minute commits may result in loss of points.
- **Keep your repository private.**
- You should also submit a single tar.gz file containing your repo (please don't include trace files) with the name ***<TeamName_LDAP1_LDAP2…>_pa2.tar.gz*** on Moodle.
- The file should contain the PA2 repository code base, all tasks' implementations, generated results and the Summary.pdf(containing data, plots and explanations of the logic used for each task).

Late submission policy

- 20% points penalty per day (applies to late submissions on both GitHub & Moodle)
- 0 marks after 5 days

students asking for zero penalty for late submission due to XYZ reason

Deadline

Deadline: 9th October, 5 PM

VIVA

Viva: Will be informed on Piazza.