Lab 4 Report: Cache Line Profiling

Indranil Dutta, Kapil Wanaskar, Prof. Haonan Wang
Computer Engineering Department
San José State University
San Jose, CA 95112
Email: {indranil.dutta, kapil.wanaskar, haonan.wang}@sjsu.edu

I. ABSTRACT

In Lab 4, we explored the application of the pointer-chasing method to empirically determine the cache line sizes for L1 and L2 caches. This report details the multi-step process involving CUDA programming and Python scripting to profile cache line sizes and analyze the impact of cache configurations on memory access patterns.

II. INTRODUCTION

In Lab 4 of our Computer Engineering course, we engaged in a practical exploration of cache memory performance using a pointer-chasing method. The aim was to gain empirical insights into the cache line sizes of both L1 and L2 caches on a GPU. Through the development and execution of a series of CUDA and Python scripts, we profiled memory access times under different cache configurations. This exercise not only reinforced theoretical concepts of cache operation but also provided a hands-on experience in performance analysis and optimization in GPU computing.

III. CACHE LINE PROFILER IMPLEMENTATION (`CACHE_LINE_PROFILER_3.CU`)

The CUDA kernel `P_chasing2` was implemented to measure memory access times across different strides, which is key to understanding cache behavior. This kernel function was designed to prevent compiler optimizations that could affect accurate timing by using volatile pointers. The CUDA kernel was executed with a varying range of stride values to induce cache hits and misses, providing a detailed timing profile for each memory access.

A. *Key Points*

- A CUDA kernel (`P_chasing2`) was implemented to generate memory accesses at different strides and measure the time taken for each.
- The function `init_cpu_data` initialized the array `A` with stride values to set up the memory access pattern.
- The kernel was launched with a range of stride values to cover the potential cache line sizes and gather comprehensive timing data.

IV. DATA RECORDING SCRIPT (`CACHE_LINE_PROFILER_RECORD_3.PY`)

After executing the CUDA kernel, the `cache_line_profiler_record_3.py` script was employed to parse the standard output and record the timing data into a CSV file, `cache_line_profiling_data.csv`. This file served as a structured dataset for further analysis.

A. *Key Points*

- The Python script automated the process of capturing profiling output and structuring it into CSV format.
- The resulting CSV file was used to store the profiling data, which includes cache configuration, stride, iteration number, and time in cycles.

V. AVERAGE TIMES PER STRIDE CALCULATION (`AVERAGE_TIMES_PER_STRIDE.PY`)

The `average_times_per_stride.py` script computed the average time per stride from the profiling data. This computation was crucial to identify patterns in timing that correlate with cache line sizes. The averages were recorded in another CSV file, `average_times_per_stride.csv`, facilitating a simplified analysis.

A. *Visualization and Analysis of Profiling Data*

The two images are graphical representations of the data collected during cache line profiling exercises in Lab 4. They are intended to be included in the lab report to visually convey the findings from the conducted experiments.

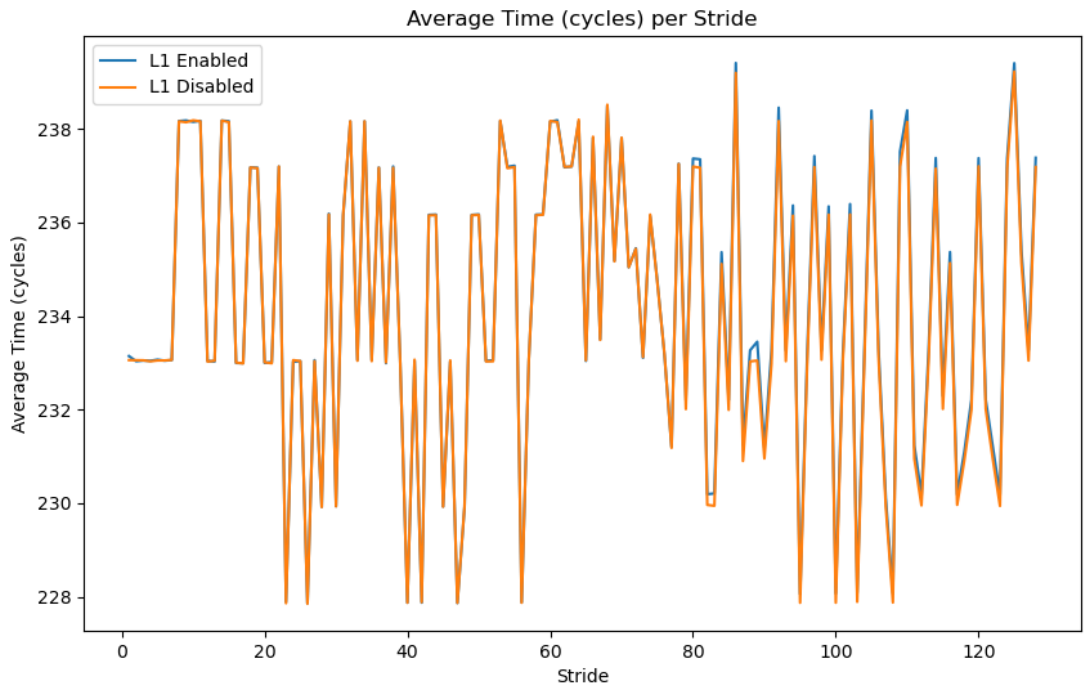

Fig. 1. Average Time (cycles) per Stride

In Figure 1, titled "Average Time (cycles) per Stride," a line plot illustrates the average number of clock cycles (y-axis) it takes to access memory at different stride lengths (x-axis). The two lines represent the conditions where the L1 cache is enabled (blue) and disabled (orange).

Observations from the Figure:

- The blue line (L1 Enabled) and the orange line (L1 Disabled) display a pattern of variability in average access times as the stride increases.
- Notable peaks and troughs suggest that certain stride lengths align with efficient memory access patterns, likely due to cache hits, while others result in increased access times, indicating cache misses.
- The comparison between L1 Enabled and Disabled conditions may reveal the efficiency of the cache. For instance, a lower average time for L1 Enabled across strides suggests that enabling L1 cache generally results in faster memory access.

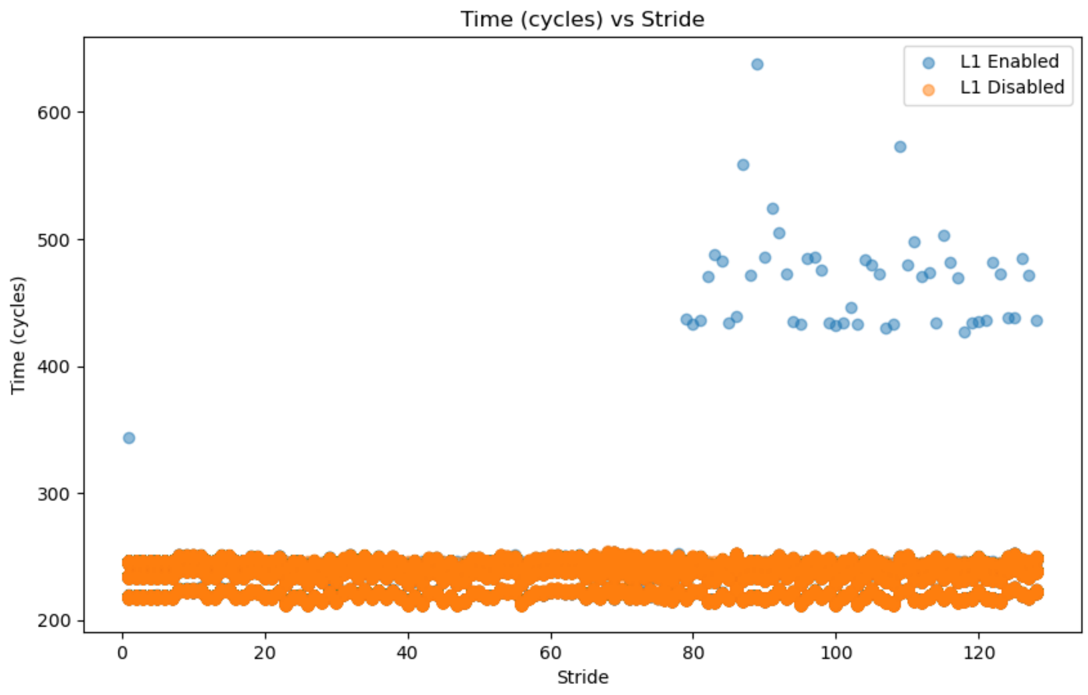

Fig. 2. Time (cycles) vs Stride Scatter Plot

The second figure, titled "Time (cycles) vs Stride Scatter Plot," shows a scatter plot where individual data points reflect the time taken (in clock cycles) to access memory at each stride (x-axis). Similar to the first figure, the blue dots correspond to the L1 cache enabled condition, and the orange dots represent the L1 cache disabled condition.

Observations from the Figure:

- There is a dense clustering of blue dots at lower cycle counts, indicating that when the L1 cache is enabled, memory access times are consistently lower.
- Conversely, the orange dots show a broader spread in cycle counts, reflecting the increased variability and generally higher access times when the L1 cache is disabled.
- There are sporadic spikes in access time for the L1 Enabled condition (blue dots), which may indicate instances where the cache does not effectively serve the memory access pattern, possibly due to cache line misses.

Together, these figures are used to assess the impact of cache line size on memory access performance and to understand the behavior of the L1 and L2 caches under various conditions. They serve as a visual aid to support the quantitative analysis conducted in the lab.

VI. Deduction of Cache Line Sizes (`deduce_cache_line_sizes.py`)

Finally, the `deduce_cache_line_sizes.py` script was utilized to analyze the average timing data and deduce the cache line sizes. By identifying significant increases in average access times at certain strides, the script inferred the cache line boundaries and recorded these deduced sizes in `deduced_cache_line_sizes.csv`.

A. Key Points

- A heuristic approach was taken to identify significant jumps in average access time, suggesting cache line boundaries.
- The inferred cache line sizes for both L1 and L2 caches were saved in a CSV file for reporting and validation purposes.

VII. Conclusion

Through this lab, we have effectively profiled and compared the cache line sizes of the L1 and L2 caches, enhancing our understanding of cache architectures and their impact on memory access patterns. The hands-on experience solidified the theoretical knowledge presented in the lectures, providing valuable insights into GPU programming and performance optimization.

The intricate interplay between cache configurations and memory access efficiency underscores the importance of tailored data structures and algorithms that align with the underlying hardware architecture. The visualization of memory access times in relation to stride lengths has not only corroborated the theoretical models of caching but also highlighted the practical considerations one must make during the software development process.

Moreover, the application of pointer-chasing techniques, accompanied by meticulous data recording and analysis, has laid a solid foundation for predictive modeling of cache behavior. As we move forward, the methodologies and findings from this lab will inform our approach to optimizing memory-intensive applications, particularly in the realm of high-performance computing where cache efficiency translates directly to computational throughput and energy savings.

Ultimately, the knowledge gained through this exercise is a testament to the synergistic relationship between computer architecture and software design. It is a compelling reminder of the continuous need for engineers to bridge the gap between theoretical principles and their application in real-world scenarios.