MEMORY-AWARE SEMI-STATIC SCHEDULING ALGORITHMS

15-418 Final Project

Joshua Cheng
joshuac2
Rae Wong
yingyeew

Contents

1 Summary

Our project aims to explore multi-core memory-aware scheduling algorithms that take into account both task computation time and memory accesses. While the computation time of tasks is the primary metric of schedulers, memory accesses can contribute significantly to the overall execution time in memory bound programs. We studied different memory-aware scheduling algorithms to take advantage of the memory access pattern and maintain data locality. This project presents a framework to evaluate the effectiveness of different scheduling algorithms on parallel matrix multiplication. We first profiling each task using the Intel Pin tool to record the memory trace, before scheduling them onto cores and evaluating the program execution time. The two main scheduling algorithms discussed in this paper are Deque Model Data Aware heuristic (DMDA) [1] and Hierarchical Fair Packing (HFP) [2], of which DMDA has shown to be more effective. To support our evaluation efforts, we also wrote a multi-core cache simulator with support for MESI cache coherence for greater flexibility in assessing cache behavior. A key application of this project is in semi-static scheduling where we can profile the behavior of tasks and reschedule them over time. The codebase for this project can be found here. [1]

2 Motivation

2.1 Importance of Data Locality

Modern CPUs make use of caches in order to hide the latency of accesses to main memory, by storing relevant or previously accessed data in the cache which can be more quickly accessed by a CPU core than main memory. However, for multicore processors with shared memory, caching behavior can be more complicated. Memory may be loaded into the cache for one core, but not the other. In addition, changes to data from one core may need to be propagated to other cores in order to maintain a coherent memory system. This is done through some of the cache coherence techniques we learnt in class. Cache coherence is also a factor that increases latency on top of cache misses. Hence, understanding the cache behavior is crucial in optimizing execution time.

When we look at the performance of parallel code, it is even more important to take into account memory accesses to avoid having memory operations and communication as the bottleneck. Therefore, a system that is able to take these operations into account when scheduling parallel tasks has the potential to greatly increase the performance of parallel programs.

2.2 Cache hierarchy of the System

The cache hierarchy can vary greatly from processor to processor. For example, many processors we see in consumer laptops and desktops have 3 levels of cache (L1/L2 for each cores and L3 shared between cores). However, certain multi-core low power/embedded processors only have 2 levels of cache (L1 for each core and L2 shared) while other CPUs even have up to L4 cache. In addition, the L1 cache is sometimes split into instruction/data

[1] https://github.com/yingyee0111/memory-aware-scheduling

caches to store instruction memory and data memory separately. Note that for this project, we will not be considering the instruction cache.

The difference in cache architecture implies that a good memory-aware scheduling algorithm would have to be conscious of the cache configuration of the system it is running on, hence motivating us to design our own scheduling algorithm. Specifically, our scheduling algorithm takes into account the cache block size as it affects how much data is fetched at once. For the purpose of evaluation, we will have 64B cache blocks as specified on the GHC machines.

2.3 Real-World Application

Memory-aware scheduling algorithms needs to know the size and memory trace of each task. This can be supplied in two ways: by user annotation or by runtime profiling. We will focus on the latter as the former implies naive static scheduling algorithm would suffice. Our algorithms benefit programs where data patterns are difficult to describe or determine beforehand, but could still be generalized to automate static scheduling.

Obtaining a runtime profile and rescheduling corresponds to semi-static scheduling. When we reschedule in a real-world system, we would need to consider the overhead of scheduling. However, there are ways to minimize the disruption such as by rescheduling at synchronization points and having minimal rescheduling attempts. In addition, less frequent rescheduling attempts gives us more time to profile the individual parallel tasks while they run, lending us a better understanding of their memory accesses and execution time [3]. Hence, we deem the benefits of the semi-static scheduling nature worthwhile despite some potential overhead.

3 Introduction

The scheduling problem is challenging as it is a computationally hard problem [4]. Most scheduling algorithms use heuristics to give an approximately good allocation, often by greedy scheduling, while some enable work stealing [5]. Adding the memory access factor makes the problem more difficult as we are optimizing with 2 different metrics in mind. Our allocation has to achieve an even distribution across cores based on the computation time per task, and also optimize for tasks with memory sharing to be placed on the same core. We have to choose which metric to prioritize and adjust the weight given to each metric. In this section, we will outline the algorithms explored and discuss their theoretical reasoning.

3.1 Profile of Tasks (input to scheduling algorithm)

Our memory-aware scheduling algorithms takes a list of tasks T_i as an input where

$$
\begin{aligned}
T_i = (\\
&id_i: \text{ task id},\\
&time_i: \text{ estimated computation time of task i},\\
&R_i: \text{ a list of memory addresses } r_i^j \text{ read by task i},\\
&W_i: \text{ a list of memory addresses } r_i^j \text{ written by task i}\\
)
\end{aligned}
$$

The way we profile each task for the necessary information is detailed in section 4.

3.2 Scheduling Algorithms

We surveyed existing memory-aware scheduling algorithms and selected two main algorithms to focus on: one prioritizing computation time and the other prioritizing cache affinity.

3.2.1 Greedy scheduling

Before we dive into our memory-aware scheduling algorithms, we will first introduce the traditional greedy scheduling algorithm. In the scenario where we are performing a rescheduling of tasks, all tasks are available for scheduling in any order and we simply optimize for the shortest overall runtime time based on the size (computation time) of each task. However, this problem is still difficult and hence, greedy scheduling is used to achieve a $2-$approximate solution. We sort the tasks in descending order based on their computation time required. Each core then takes a task from this queue whenever it is idle.

Algorithm 1 Greedy Scheduling Algorithm

Require: $T_i = (id_i, time_i)$
  $Deque \leftarrow [\cdots T_i \cdots]$ in descending order of $time_i$
  while $Deque$ and core C_k is idle **do**
    $C_k \leftarrow pop(Deque)$
  end while

The greedy scheduling algorithm will be used as a baseline benchmark. Comparing memory-aware algorithms against this will show any improvement and tradeoff from considering the additional heuristics.

3.2.2 Deque Model Data Aware (DMDA)

The Deque Model Data Aware heuristic is one that prioritizes computation time and it works in a similar fashion as the greedy scheduling algorithm. DMDA incorporates the cost of memory fetches into the size of a task. Instead of solely scheduling based on the computation time, we also account for the time needed to fetch data that has not been cached. Hence, when we attempt to schedule a task, we search for the core C_k that is able to complete the task T_i the earliest after finishing its current tasks and fetching all required memory accesses d_i^j.

Algorithm 2 DMDA Scheduling Algorithm

Require: $T_i = (id_i, time_i, Data_i = [\cdots d_i^j \cdots])$
  $Deque \leftarrow [\cdots T_i \cdots]$ in descending order of $time_i$
  while $Deque$ **do**
    Find C_k such that $Completion_k(T_i)$ is minimal
      where $Completion_k(T_i) = next\_available\_time_k$
        $+ Fetch_k(d_i^j)$ $\forall d_i^j \in D_i, d_i^j \notin C_k$
        $+ time_i$
    $C_k \leftarrow pop(Deque)$
    Add D_i to the memory of C_k
  end while

As seen from the algorithm outline, DMDA heuristic prioritizes completion time which serves as a guideline to avoid introducing load imbalance as a tradeoff for cache affinity. At the same time, it incorporates the cost of memory transfer. This algorithm hinges on the accuracy of the estimation function $Fetch_k(d)$ which measures how much time is needed for data transfers. This is dependent on the architecture of runtime systems and cache hierarchy as discussed in Section 2.2.

When we implement this algorithm, we made sure to model the correct cache configuration. The GHC machines we ran our experiments on has cache with block size 64B and hence, each memory fetch would bring the next 64B into memory of the core. This behaviour is necessary to model as it provides the context for memory-aware algorithms. We want to be able to reuse data in the cache as much as possible.

3.2.3 Hierarchical Fair Packing (HFP)

The Hierarchical Fair Packing algorithm [2] was originally designed for a single core machine where we want to optimize the order of task execution to take advantage of temporal locality. This algorithm thus tends to prioritize cache affinity over the problem of load balancing.

HFP begins by having each task in an individual package and proceed to merge packages based on the amount of data they share $D(P_i) \cap D(P_j)$. When merging packages, we prefer to merge smaller packages before considering a merge with a larger package. HFP also performs two iterations of merges where the first iteration is constrained by the memory bound of the device. In our adapted algorithm, we consider a stricter memory bound of the L1 cache size. Merging also preserves the order of tasks within the package to maintain the locality carried over. In the case of a multi-core machine, we would end up with k packages in the end. Finally, after constructing the k packages, we attempt to balance the load between them with a second phase of reallocation.

Algorithm 3 HFP Scheduling Algorithm (phase 1: forming packages)

Require: $T_i = (id_i, Data_i = [\cdots d_i^j \cdots])$
  Let $P_i \leftarrow [T_i]$
  while $|P| > k$ **do**
    Process P_i's in ascending order of the number of tasks
    Find a package P_j to merge with where $D(P_i) \cap D(P_j)$ is maximal
  ▷ In the first round, do not merge if the total data required exceeds memory bound M
  end while
  Return $P_i \cdots P_k$

Algorithm 4 HFP Scheduling Algorithm (phase 2: load balancing)

Require: $P_i = ([\cdots T_i^j \cdots])$
  while $\max_i P_i > Average\_Size$ **do**
    Transfer tasks from P_{max} to P_{min}
  end while
  Return $P_i \cdots P_k$

3.2.4 HFP with Heterogeneous Tasks

The algorithm described above has the shortcoming of using the number of tasks as a measure of the size of a package. However, many programs have heterogeneous tasks of varying computation time. Hence, we incorporate the varying size of each task as part of the load balancing heuristic in phase 2.

Algorithm 5 HFP Scheduling Algorithm (phase 2: load balancing)

Require: $P_i = ([\cdots T_i^j \cdots])$
  while $\max_i P_i > Average\_Load * slack$ **do**
    ▷ *slack* is present as we might not be able to achieve a perfect load balance
    Transfer tasks from P_{max} to P_{min}
  end while
  Return $P_i \cdots P_k$

3.2.5 Ready ordering

Finally, after we have allocated tasks to each core, we can considering reordering them to minimize cache trashing. The reordering algorithm we implemented is as follows.

Algorithm 6 Ready Ordering

Require: $L_k = ([\cdots T_i \cdots])$ allocated on C_k
  $L'_k \leftarrow []$
  while L_k **do**
    Search for T_i that requires that fewest memory fetches
    $L'_k \leftarrow T_i$
    Adds D_i to the memory of C_k
  end while
  Return L'_k

4 Approach

4.1 Sample Program and Infrastructure

We identified parallel blocked matrix multiplication as a suitable benchmark program as it breaks down nicely with a fixed data sharing pattern. We wrote the matrix multiplication program with multi-threading and consider the work on each thread as a unit of task. To observe runtime differences, we loop over the program for 1000 iterations. Experiments were ran on the GHC machines using 4 and 8 cores.

4.2 Profiling

To obtain a measure of the computation time, we dumped out the assembly instructions of the program and use instruction count as a gauge. The other profiling we need is the data accessed per thread. To track the memory accesses, we used the Intel Pin tool [6] to profile the memory trace. We wrote a custom script for the Intel Pin tool, based on the provided example code with the Pin tool installation.

4.3 Scheduling

Once we have profiled the tasks, we feed the list of tasks in the format specified in Section 3.1 into our scheduling algorithms. The scheduling algorithms function at a higher level and are written in python. The scheduler then outputs an assignment matching each task to a specific core.

4.4 Assessing Effectiveness

To evaluate the effectiveness of our scheduler, we set each thread to run on its scheduled core by calling `pthread_setaffinity_np`. This is done for our memory-aware scheduling algorithms and the greedy scheduling algorithm that we benchmark against. The runtime of the compiled code is then used to judge if we have achieved a better scheduling.

4.5 Cache Simulator

As an additional tool for evaluating the effectiveness of our scheduler on the cache behavior, we developed a multi-core cache simulator. The cache simulator worked on the memory traces we obtained using the Intel Pin tool, while allowing us to schedule specific tasks to run on specific cores. The simulator was designed in such a way that the cache parameters (such as block size and associativity) would be easily parameterizable, though we ran our simulations with cache parameters based on the Intel i7-9700 CPU present in the GHC machines.

The cache coherence protocol used in the simulator was based on the MESI protocol discussed in class, rather than the MESIF protocol that is actually used by Intel processors. In addition, the interconnect behavior involved in bus transactions was abstracted away, since we did not have a real sense of the timing involved in the various memory transactions. This was because we only had memory traces instead of real-time timestamps of memory operations, and these real-time timestamps would likely vary from run-to-run anyways. Thus, we decided to measure overall interconnect traffic rather than try to simulate any amount of interconnect connection.

To abstract away some of the complexity of the cache hierarchy, the cache simulator was designed around the L1 Cache of each core, since L1 cache misses were likely the biggest contributor in terms of performance for each multi-threaded program. This is especially so given that the programs we are profiling were relatively short, due to limitations and the long runtime of our memory tracing using the intel Pin tool.

The performance and accuracy of the multi-core cache simulator was verified using small, hand-traceable memory traces.

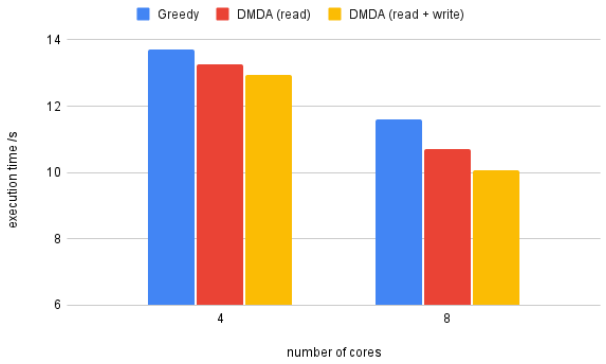
5 Results

5.1 Deque Model Data Aware (DMDA)

As shown in 1, the runtime of our program scheduled by the DMDA algorithm is faster than the program scheduled by greedy algorithm. At 8 core count, we are able to achieve a speedup of 10-15%. This shows that the heuristic of incorporating memory access into our cost function is useful. We also investigated if read or write access plays a bigger role and they turn out to complement each other. This is expected as both read and write fetch data into the cache.

In addition, we ran our cache simulator on the DMDA scheduled program and notice that the cache coherence traffic is slightly lower compared to that of the greedy scheduled program. The full output of the scheduler can be found in the appendix.

Figure 1: Improvement in runtime using DMDA Scheduler



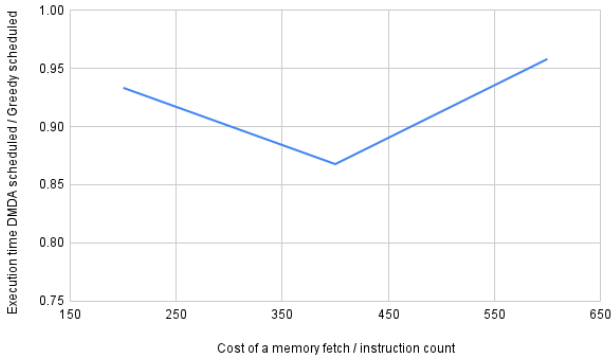
A hyper parameter we experimented with was the memory fetching cost function in DMDA. The cost is measured in terms of instruction counts to have the same unit of measurement as our estimated computation time. We gauged the cost of a cache miss to be around 400 instruction counts as a reasonable heuristic would be 100+ cycles and the system could operate at 2-3 instructions per cycle. As shown in 2, there is an optimal value that accurately models after the actual cost. In our model, we assume a fix cost of fetching any data. To extend the usability of this algorithm, we would need to consider the topology of memory storage and communication to account for variable costs.

5.2 Hierarchical Fair Packing (with Heterogeneous Tasks)

In our evaluation of HFP scheduling algorithm, we used both read and write memory accesses as proven effective by the DMDA scheduler.

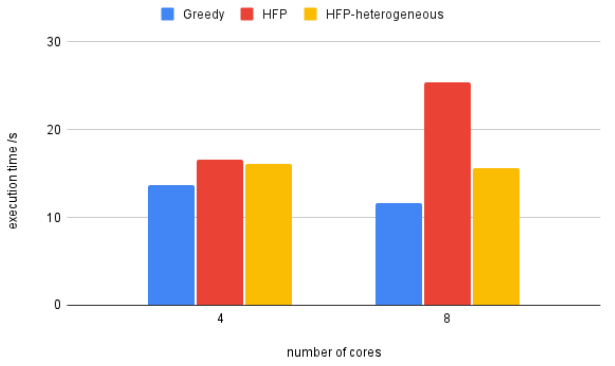
The HFP algorithm unfortunately does not perform well. This is clearly attributed to the tradeoff between prioritizing data locality and load balancing. As seen in 3, the execution time is lower after we implemented a better load balancing pass in the heterogeneous HFP algorithm. This is especially so when 8 cores were used. The results suggests that HFP might be more suitable for programs with homogeneous tasks with a large amount of data sharing.

Figure 2: Memory cost function of the DMDA Scheduler



To verify our hypothesis, we wrote a parallel image convolution program and find that HFP no longer perform worse than the greedy scheduler (4). In an image convolution, each task is homogeneous and shares data uniformly with its neighboring tasks. Hence, HFP might be more suited for such scenarios.

Figure 3: Tradeoff between data locality and load balancing in HFP



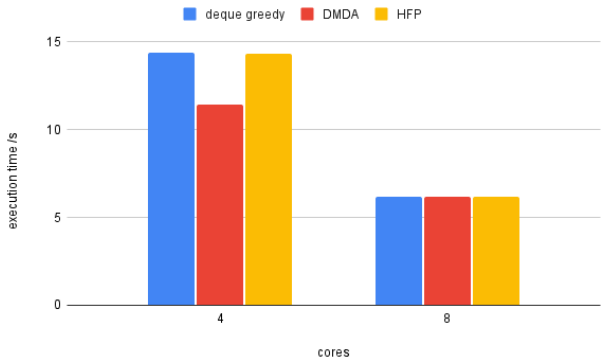
5.3 Ready Ordering

Since we cannot fix the execution order of running threads, we evaluate the effects of task reordering using our cache simulator. We evaluate the ready ordering algorithm by the running it on the output of the greedy scheduler as well as the DMDA scheduler. The reordering is meant to allow data to be loaded incrementally and minimize evictions, however, we did not see any improvement from our cache simulation as attached in the appendix. This suggests that the reordering algorithm could be better. For instance, we could reorder tasks according to the lifetime of its data instead of simply using the size of the data set.

6 Conclusion

This project explored the use of memory access patterns for task scheduling. We conclude that memory-aware scheduling algorithms have the potential to improve execution time,

Figure 4: Scheduler performance on an image convolution program



and are especially useful when data sharing patterns can be captured over time. However, there are also limitations of this approach. Seeking to optimize data locality might be at odds with load balancing, which should be prioritized. The effectiveness of memory-aware scheduling is also highly dependent on the kind of program and system we are running. Hence, such scheduling algorithms might be used under a domain specific context rather than being used as a general guideline.

References

[1] M. Gonthier, L. Marchal, and S. Thibault, "Memory-aware scheduling of tasks sharing data on multiple gpus with dynamic runtime systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 694–704.

[2] ——, "Locality-aware scheduling of independent tasks for runtime systems," in *Euro-Par 2021: Parallel Processing Workshops*, R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, and L. Ricci, Eds. Springer International Publishing, 2022, pp. 25–27.

[3] K. Kaya and C. Aykanat, "Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 883–896, 2006.

[4] J. Ullman, "Np-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, 1975. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000075800080

[5] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 1–12. [Online]. Available: https://doi.org/10.1145/341800.341801

[6] "Pin - a dynamic binary instrumentation tool," https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, accessed: 2023-05-20.

7 Credit

The work distribution between our group is 50-50. Rae has been working on the scheduling algorithms: understanding the theory, implementing them and running experiments. Joshua has been working on the profiling of tasks and cache behavior: collecting memory traces and building the cache simulator. We collectively put together this report.

8 Appendix

8.1 Cache Simulator Output

8.1.1 Greedy scheduled

```
Total Threads: 57
Interconnect Traffic: 1564

**** CORE 0 ****
Memory Reads: 1019
Memory Writes: 502
Cycle Count: 3100
Evictions: 10
Bus Responses: 63

**** CORE 1 ****
Memory Reads: 1042
Memory Writes: 539
Cycle Count: 3286
Evictions: 11
Bus Responses: 16

**** CORE 2 ****
Memory Reads: 77
Memory Writes: 613
Cycle Count: 3857
Evictions: 36
Bus Responses: 42

**** CORE 3 ****
Memory Reads: 77
Memory Writes: 678
Cycle Count: 4232
Evictions: 57
Bus Responses: 30

**** CORE 4 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4057
Evictions: 86
Bus Responses: 34

**** CORE 5 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4029
Evictions: 87
Bus Responses: 29

**** CORE 6 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 3966
Evictions: 83
Bus Responses: 29

**** CORE 7 ****
main
```

8.1.2 DMDA scheduled

```
Total Threads: 57
Interconnect Traffic: 1552

**** CORE 0 ****
Memory Reads: 1231
Memory Writes: 632
Cycle Count: 3757
Evictions: 12
Bus Responses: 65

**** CORE 1 ****
Memory Reads: 1148
Memory Writes: 604
Cycle Count: 3620
Evictions: 13
Bus Responses: 19

**** CORE 2 ****
Memory Reads: 956
Memory Writes: 548
Cycle Count: 3491
Evictions: 19
Bus Responses: 40
```

```
**** CORE 3 ****
Memory Reads: 947
Memory Writes: 548
Cycle Count: 3548
Evictions: 25
Bus Responses: 28

**** CORE 4 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4055
Evictions: 87
Bus Responses: 35

**** CORE 5 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4027
Evictions: 88
Bus Responses: 30

**** CORE 6 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 3958
Evictions: 83
Bus Responses: 29

**** CORE 7 ****
main
```

8.1.3 Greedy scheduled and Ready ordered

```
Total Threads: 57
Interconnect Traffic: 1582

**** CORE 0 ****
Memory Reads: 1019
Memory Writes: 502
Cycle Count: 3223
Evictions: 13
Bus Responses: 94

**** CORE 1 ****
Memory Reads: 1042
Memory Writes: 539
Cycle Count: 3409
Evictions: 16
Bus Responses: 28

**** CORE 2 ****
Memory Reads: 291
Memory Writes: 613
Cycle Count: 3967
Evictions: 45
Bus Responses: 30

**** CORE 3 ****
Memory Reads: 282
Memory Writes: 678
Cycle Count: 4265
Evictions: 64
Bus Responses: 27

**** CORE 4 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4039
Evictions: 89
Bus Responses: 31

**** CORE 5 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4008
Evictions: 90
Bus Responses: 32

**** CORE 6 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 3912
Evictions: 79
Bus Responses: 31

**** CORE 7 ****
Memory Reads: 0
Memory Writes: 0
Cycle Count: 0
Evictions: 0
Bus Responses: 0
```

8.1.4 DMDA scheduled and Ready ordered

```
Total Threads: 57
Interconnect Traffic: 1567

**** CORE 0 ****
Memory Reads: 1231
Memory Writes: 632
Cycle Count: 3862
Evictions: 14
Bus Responses: 85

**** CORE 1 ****
Memory Reads: 1148
Memory Writes: 604
Cycle Count: 3746
Evictions: 17
Bus Responses: 29

**** CORE 2 ****
Memory Reads: 956
Memory Writes: 548
Cycle Count: 3573
Evictions: 26
Bus Responses: 32

**** CORE 3 ****
Memory Reads: 947
Memory Writes: 548
Cycle Count: 3560
Evictions: 28
Bus Responses: 26

**** CORE 4 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4010
Evictions: 86
Bus Responses: 28

**** CORE 5 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 4000
Evictions: 89
Bus Responses: 31

**** CORE 6 ****
Memory Reads: 77
Memory Writes: 715
Cycle Count: 3905
Evictions: 78
Bus Responses: 29

**** CORE 7 ****
main
```