cache-coherence

November 28, 2024

0.1 Author

Ivan Monaco (ivancmonaco@gmail.com)

1 Results

1.1 Hardware Results

Lets first load up and build tables for all our results.

All tests were ran in a x86 native 12-core processor running Ubuntu 22.x.y. They were ran 100 times and the average was taken.

The following table shows which tests was run (Thread configuration + number of cores) and how long does it take to complete, in ascending order (lower is better).

[103]:
```
import pandas as pd
import os
import glob

results_path = f"{os.getcwd()}/results"

results = []

files = os.listdir(f"{results_path}/native")

for file in files:

    df = pd.read_csv(f"{results_path}/native/{file}", delimiter=" ",␣
 ↪header=None, names=["Label", "Value", "Unit"])

    df["NumericValue"] = df["Value"].astype(float)
    average_time = df["NumericValue"].mean()

    name = file.split("/")[-1].split(".")[0]
    cores = name.split("-")[-1]
    name = " ".join(name.split("-")[:-2]).capitalize()
```

```
    results.append({"Test": name, "Cores": cores , "Average Time (ms)":␣
 ↪average_time})

native_df = pd.DataFrame(results).sort_values(by="Average Time (ms)")
native_df = native_df.reset_index()
native_df = native_df.drop(['index'],axis=1)

native_df
```

[103]:

	Test	Cores	Average Time (ms)
0	Block race opt	4	0.220384
1	All opt	4	0.226861
2	Block race opt	2	0.229220
3	All opt	2	0.244881
4	Naive	1	0.323143
5	Res race opt	1	0.326467
6	Block race opt	1	0.329142
7	Chunking res race opt	1	0.332100
8	Chunking	1	0.332399
9	All opt	1	0.337583
10	Block race opt	8	0.341541
11	All opt	8	0.356096
12	All opt	12	0.406281
13	Block race opt	12	0.414171
14	Chunking res race opt	2	0.580947
15	Res race opt	2	0.587572
16	Naive	2	0.595884
17	Chunking	2	0.610594
18	Chunking	4	0.630850
19	Chunking res race opt	4	0.633819
20	Chunking res race opt	12	0.639322
21	Naive	4	0.645525
22	Chunking	8	0.646897
23	Naive	8	0.648690
24	Naive	12	0.649543
25	Res race opt	4	0.652155
26	Chunking	12	0.655746
27	Chunking res race opt	8	0.662393
28	Res race opt	12	0.668676
29	Res race opt	8	0.676898

Now, with this data we can try to answer the first 3 questions

1.1.1 Question 1

For algorithm 1, does increasing the number of threads improve performance or hurt performance? Use data to back up your answer. Lets filter out only the first and sixth algorithm (Naive and All optimizations)

[104]:
```
q_1 = native_df.query("Test == 'Naive' or Test == 'All opt'")
q_1
```

[104]:

	Test	Cores	Average Time (ms)
1	All opt	4	0.226861
3	All opt	2	0.244881
4	Naive	1	0.323143
9	All opt	1	0.337583
11	All opt	8	0.356096
12	All opt	12	0.406281
16	Naive	2	0.595884
21	Naive	4	0.645525
23	Naive	8	0.648690
24	Naive	12	0.649543

Being the algorithm 1 , the 'Naive' implementation, we can see that the test with just 1 thread did pretty much better than the threaded implementation.

Naive algorithm took double the time to complete comparing runs with 1 and with 12 cores. With this data we can assert that the thread count increase, actually hurt algorithm 1 performance.

1.1.2 Question 2

(a) For algorithm 6, does increasing the number of threads improve performance or hurt performance? Use data to back up your answer. In the other hand, Algorithm 6 (all optimizations enabled), did much better when increasing the thread count, but up to 4, then the trend reverted, and behaved like the naive algorithm.

(b) What is the speedup when you use 2, 4, 8, and 16 threads (only answer with up to the number of cores on your system).

- 1 thread : 100% performance.
- 2 threads : 128% performance.
- 4 threads : 133% performance.
- 8 threads : 95% performance.
- 12 threads: 83% performance.

1.1.3 Question 3

(a) Using the data for all 6 algorithms, what is the most important optimization, chunking the array, using different result addresses, or putting padding between the result addresses? Assuming based on (2) that the ideal amount of threads for this experiment is 4, lets filter out the data and see how the implementations with 4 threads compare to each other

[105]:
```
q_3 = native_df.query("Cores == '4'")
q_3
```

[105]:

	Test	Cores	Average Time (ms)
0	Block race opt	4	0.220384

```
1                 All opt     4           0.226861
18               Chunking     4           0.630850
19  Chunking res race opt     4           0.633819
21                  Naive     4           0.645525
25          Res race opt      4           0.652155
```

We can see that up to the top of the table there is the block race optimization, which is in fact the fastest global algorithm with the 4 threads configuration. This implementation, comparing it with the naive implementation using also 4 threads, is about 3 times faster.

(b) Speculate how the hardware implementation is causing this result. What is it about the hardware that causes this optimization to be most important? The cache line width is responsible. In this case we are using blocks of 64B exclusively for each thread , in an independent form, reducing caches writes to memory.

1.2 Gem5 Simulated Results

For the gem5 experiments the same tests were ran in a 1,2,4,8,16 threads configuration, with the same number of elements (32768), and a default xBar latency of 10.

1.2.1 Question 4

(a) What is the speedup of algorithm 1 and speedup of algorithm 6 on 16 cores as estimated by gem5? Lets load some data related to the experiments!

[106]:
```
gem5_sim_seconds_df = pd.read_csv(f"{results_path}/gem5/stats/simSeconds.txt",␣
 ↪delimiter=",")
gem5_sim_seconds_df = gem5_sim_seconds_df.sort_values(by="Time Taken (s)",␣
 ↪ascending=True)
q_5 = gem5_sim_seconds_df.query("(Test == 'All opt' or Test == 'Naive') and␣
 ↪(Cores == 1 or Cores == 16)")
q_5
```

[106]:
```
       Test  Cores  Time Taken (s)
1   All opt     16        0.000111
20    Naive      1        0.000838
0   All opt      1        0.000839
21    Naive     16        0.000917
```

We can clearly see that the difference is abismal for the sixth algorithm: 656% speedup!

For the naive algorithm it actually hurted its performance.

(b) How does this compare to what you saw on the real system? This behavior matches what we saw in the real world.

The difference relies on the abismal difference on the x16 cores with the all optimizations enabled (we saw an increase, but not that much!)

1.2.2 Question 5

Which optimization (chunking the array, using different result addresses, or putting padding between the result addresses) has the biggest impact on the hit ratio? Lets load the data. We are going to use the L1D demand hit stat (greater is better) for the first core of each implementation (we are using 16 cores for each experiment)

[107]:
```
gem5_cache_hit_ratio_df = pd.read_csv(f"{results_path}/gem5/stats/cacheHitRatio.
↪txt", delimiter=",")
gem5_cache_hit_ratio_df["Cache Hit"] = gem5_cache_hit_ratio_df["Cache Hit"].
↪astype(float)
gem5_cache_hit_ratio_df = gem5_cache_hit_ratio_df.sort_values(by='Cache Hit',␣
↪ascending=False)
```

[108]:
```
q_5 = gem5_cache_hit_ratio_df
q_5
```

[108]:

	Test	Cache Hit
0	All opt	35942.0
3	Chunking res race opt	35721.0
2	Chunking	31936.0
1	Block race opt	29863.0
5	Res race opt	27570.0
4	Naive	25823.0

The hit ratio is affected by all of them, as the all optimizations experiment had a better results than naive (we can even see that the all opt test rank the better, as each optimization contributed)

However, individually, the chunking combined with the address change wins the battle.

1.2.3 Question 6

Which optimization (chunking the array, using different result addresses, or putting padding between the result addresses) has the biggest impact on the read sharing? This question is answered comparing all the algorithms with 16 cores as the thread configuration, and checking the sum of the read sharing stats of each core.

[109]:
```
gem5_read_sharing_df = pd.read_csv(f"{results_path}/gem5/stats/readSharing.
↪txt", delimiter=",")
gem5_read_sharing_df["Read Sharing"] = gem5_read_sharing_df["Read Sharing"].
↪astype(float)
gem5_read_sharing_df = gem5_read_sharing_df.sort_values(by='Read Sharing',␣
↪ascending=True)
q_6 = gem5_read_sharing_df
q_6
```

[109]:

	Test	Read Sharing
2	Chunking	194.0
3	Chunking res race opt	199.0

```
0                All opt         223.0
1            Block race        1193.0
4                 Naive        1970.0
5          Res race opt        1974.0
```

We can see that the biggest impact is made by the Chunking optimization.

1.2.4 Question 7

Which optimization (chunking the array, using different result addresses, or putting padding between the result addresses) has the biggest impact on the write sharing? This question is answered comparing all the algorithms with 16 cores as the thread configuration, and checking the sum of the write sharing stats of each core.

[110]:
```
gem5_write_sharing_df = pd.read_csv(f"{results_path}/gem5/stats/writeSharing.
↪txt", delimiter=",")
gem5_write_sharing_df["Write Sharing"] = gem5_write_sharing_df["Write Sharing"].
↪astype(float)
gem5_write_sharing_df = gem5_write_sharing_df.sort_values(by='Write Sharing',␣
↪ascending=True)
q_7 = gem5_write_sharing_df
q_7
```

[110]:
```
                   Test  Write Sharing
1            Block race          152.0
0               All opt          186.0
3  Chunking res race opt        32735.0
5          Res race opt        32774.0
2              Chunking        32820.0
4                 Naive        32864.0
```

We can see that the biggest impact is made by the Block race optimization (by far, and the only one that made something).

1.2.5 Question 8

(a) Out of the three characteristics we have looked at, the L1 hit ratio, the read sharing, or the write sharing, which is most important for determining performance? Looking at the global table and seeing that the Block race optimization performed the best, and seeing the brutal difference in the write sharing table, I can conclude that the write sharing stat is the most important characteristic to look when determining performance. This makes sense, as every time a core write to a shared cache line, all the threads halt to let this cache coherence take place.

(b) Using data from the gem5 simulations, now answer what hardware characteristic causes the most important optimization to be the most important. As we just said, a good cache coherence in your cache hierarchy is the best optimization to gain performance.

1.2.6 Question 9

As you increase the cache-to-cache latency, how does it affect the importance of the different optimizations? We ran naive and all opt tests, with same number of cores, with x bar latency of 1, 10, and 25.

[111]:
```
gem5_x_bar_latency_df = pd.read_csv(f"{results_path}/gem5/stats/simSecondsXBar.
↪txt", delimiter=",")
gem5_x_bar_latency_df["Time Taken (s)"] = gem5_x_bar_latency_df["Time Taken␣
↪(s)"].astype(float)
gem5_x_bar_latency_df = gem5_x_bar_latency_df.sort_values(by='Time Taken (s)',␣
↪ascending=True)
```

[112]:
```
q_9 = gem5_x_bar_latency_df
q_9
```

[112]:

	Test	X Bar Latency	Time Taken (s)
3	All opt	1	0.000226
4	All opt	10	0.000236
5	All opt	25	0.000251
0	Naive	1	0.000347
1	Naive	10	0.000818
2	Naive	25	0.000958

We can see that the X Bar latency has a huge impact on the performance, outweighting the importance of the optimizations.

If we reduce to a minimum of 1 this latency, the 2 algorithms are not that far from each other, even after all our optimization work!