Michael LeMay
1/21/2019

Project Description

My implementation was generally the same as the described implementation, with a few changes made to the structure in order to improve performance in python. For a more detailed treatment, please see the README.txt file included in the project submission.

Correctness

To check correctness, I first did as 2.1 described, and ran my emulator with a variety of inputs. I got the correct answers to the matrix multiply and to the daxpy.

They are, respectively:

30	36	42
84	108	132
138	180	222

and

0	7	14
21	28	35
42	49	56

To further check my emulator for correctness, I used numpy to compute the correct answer every time I ran my code. I then did an element-wise check to ascertain whether the two results were equal, and reported if my manually computed answer matched the actual result generated in numpy. Over the course of all the tests I ran, I ensured that my implementation worked. I also randomly generated my X and Y, using integer values to prevent any precision issues. So I am very confident that my implementation works in the sense that the algorithms are right and the right numbers are input and output (even if it is certainly possible my cache

does not work quite properly). For all the data below, the results are compared with the numpy output and they were all correct numerically.

Associativity

Below is the evidence table I generated. There is a pretty clear relation when it comes to cache associativity. Increasing the cache associativity decreases the miss rate overall. This is because each set has more blocks, so there is a reduced chance of a conflict since there will generally be fewer conflict misses. However, there is a tradeoff in terms of time, because it takes time to linearly search a cache.

Because of this, it makes sense to use a cache with some associativity. However, as the associativity continues to increase, diminishing marginal returns on the miss rate decrease and increases searching costs would become an issue. In my data specifically, I got good results around where Intel decided to draw the line for matrix multiplication.

In my data, you can also see a clear relationship between associativity and read miss rate and write miss rate. At low associativity, blocks are evicted more frequently in conflict misses, particularly for writes, which take place sequentially, unlike the A and B accesses. At the size we're dealing with, each write will take place after multiple reads (since a row and a column of data need to be read to produce a write). At least two of these reads will involve blocks that map to the same location we are writing to, since I'm iterating over a whole row and a whole column. So you will never have a C block in memory when you go to write, when the associativity is 1 or 2.

As the associativity continues to increase, there is a huge drop in the number of write misses. This is what I would expect, for the same reason outlined above.

The researchers at Intel have a tradeoff to make between miss rate, which is generally reduced by increased associativity since the number of conflict misses falls, and performance, which is hampered by having to iterate over multiple potential spots searching for a block. While I calculated my data in parallel with the other tables, so this information should be taken with a grain of salt, I saw a large increase in runtime between an associativity of 1 and fully associative. The runtime went from 908 seconds to 12551 seconds. Some of that huge increase is due to context switching because of other tests, but that gives a good idea of the kind of tradeoffs the designers were working with.

Given this, they no doubt ran similar simulations with some kind of cost model relating cache accuracy and runtime, and picked 8-way set associative because it maximized their chosen cost model.

Cache	Instructio	Read	Read	Read	Write	Write	Write

Associativity	ns	Hits	Misses	Miss %	Hits	Misses	Miss %
1	449280000	212364900	1227510 0	0.0546434294871 79485	0	3456000	1.0
2	449280000	206585790	1805421 0	0.0803695245726 4958	0	3456000	1.0
4	449280000	203331210	2130879 0	0.0948575053418 8034	932400	2523600	0.7302083333333 333
8	449280000	198872535	2576746 5	0.1147055956196 5812	1831800	1624200	0.4699652777777 778
16	449280000	187377480	3726252 0	0.1658766025641 0255	2015205	1440795	0.4168967013888 889
1024 (fully associative)	449280000	209952000	1468800 0	0.0653846153846 1539	3348000	108000	0.03125

Table 1: Associativity table

Memory Block Size

At first, increasing block size decreases the miss rate. But that eventually goes away. Eventually, as the block size increases more and more, there are fewer blocks available in the cache. Then cache's miss rate starts to increase again. So the trend is clearly U shaped. This makes sense, given that extremely small blocks lead to a lot of tiny blocks that aren't taking advantage of spatial locality. As you increase the block size, you take more and more advantage of spatial locality. This decreases your miss rate. But eventually you reach diminishing marginal returns and are pretty much saturated in terms of spatial locality you can take advantage of. Then the issue of the increased block size becomes your cache is only storing a few blocks. This makes it hard to take advantage of temporal locality and increases your miss rate. So this tradeoff, between temporal and spatial locality, is captured by the block size. The best performance in terms of miss rate occurs in the middle, between these two extremes.

Also just to notes: The perfect write miss percentage is to be expected in the operation I'm doing, since my only writes are when I update the value of C[i][j] = A dot B transpose. Since A dot B transpose involves a full row and column of operations and the default associativity is just two, the block storing C[i][j] is never in the cache after the dot product for row i and column j is finished computing.

Block Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
8	449280000	187262010	3737799 0	0.166390625	0	3456000	1.0
16	449280000	197699370	2694063 0	0.1199280181623 9317	0	3456000	1.0
32	449280000	203220450	2141955 0	0.0953505608974 359	0	3456000	1.0
64	449280000	206585790	1805421 0	0.0803695245726 4958	0	3456000	1.0
128	449280000	209478060	1516194 0	0.0674943910256 4103	0	3456000	1.0
256	449280000	210579270	1406073 0	0.0625922809829 0598	0	3456000	1.0
512	449280000	113778060	1108619 40	0.4935093482905 983	0	3456000	1.0
1024	449280000	113619960	1110200 40	0.4942131410256 41	0	3456000	1.0

Table 2: Memory block size table

Total Cache Size

Increasing the cache size also reduces the number of misses for the cache, particularly once the cache becomes large enough to hold the entire matrices in cache memory. This

makes sense, since having more space to store stuff in the cache means there will be less capacity misses. The obvious tradeoff is cost and the physical engineering of the increased cache memory.

In my data, I do not get anywhere near a 0.005 (0.5%) threshold the Skylake with the other cache settings on default. I instead get a 0.5 threshold.

In order to achieve the desired accuracy, I will clearly need to adjust the other cache settings discussed so far (block size and associativity). I guessed, based on prior results, I would need to increase the associativity. So I ran a series of tests to determine how much of an increase would be needed. I found that a fully associative cache does get down to the desired threshold. So I would propose implementing a fully associative cache, or at least a highly associative cache, to meet this target. I also got good results with a block size of 32 instead of 64 and a associativity of 64.

Both datapoints point to the fact that removing conflict misses seems to be the answer to decreasing the hit rate. Of course, there is a corresponding performance dropoff in terms of runtime. But the overall solution to that is probably two level caching, one that emphasizes speed (the cache closer to the processor) and one that emphasizes low miss rate (further from processor).

Cache Size (Bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
4096	449280000	109512000	115128000	0.5125	0	3456000	1.0
8192	449280000	109512000	115128000	0.5125	0	3456000	1.0
16384	449280000	109512000	115128000	0.5125	0	3456000	1.0
32768	449280000	109688400	114951600	0.5117147435897436	0	3456000	1.0
65536	449280000	206585790	18054210	0.0803695245726 4958	0	3456000	1.0
131072	449280000	222907245	1732755	0.0077134748931623935	3225600	230400	0.0666666666666 6667

262144	449280000	223321204	1318796	0.0058707086894586894	3341312	114688	0.0331851851851851 85186
524288	449280000	223698840	941160	0.0041896367521 36752	3456000	0	0.0

Table 3: Total cache size table

Problem Size and Cache Thrashing

Tables are after my answers.

1. It is clear in my data that, for the regular mxm algorithm without blocking, changes to the problem size do have a large effect on the results of the problem. For 480 and 488, I saw a huge performance improvement associated with the switch to the blocked algorithm. However, the magnitude of the improvement was not precisely the same. And for 512, there was virtually no performance gain for the 2 and 8 set associative blocked algorithm. There was a large performance gain for the fully associative cache, when switching to fully blocked. The reason for this variety of results (particularly in the 512 2 and 8 set associative caches) is the effect of set size and cache mapping. There are 128 blocks for all of the caches. With a 2 way set associative cache, we have 64 different sets, each with two blocks and with an 8 way set associative cache, we have 16 different sets with 8 blocks. The algorithm guarantees efficient block usage, but if the numbers work out badly, each block could be mapped into the same few locations in the cache. In other words, efficiently using each block is only part of the problem. Another part is making sure that you are using the entire cache efficiently, instead of constantly replacing blocks in the same few spots in memory. The fully associative cache performance for 512 x 512 is proof that this is extremely important.
2. In my data, the blocked matrix multiply algorithm does not outperform the regular algorithm in table 4. There is a slight reduction in the miss rate, from .5 to .49. I would say that this is quite unsuccessful, given how much less straightforward the algorithm is. This poor performance on 2 set associative caches is clearly occurring because of conflict misses, since increasing to full associativity gets great performance. I discussed why I think this is happening above, but just to reiterate, the issue is that in the blocking algorithm, I am not taking advantage of the full cache, and instead have different blocks that map to the same set evicting each other. Switching to fully associative prevents this mapping issue from causing performance.
3. As I discussed above, I do see a good increase in performance in terms of cache hit rate for fully associative caches. This is because increased cache associativity reduces conflict misses, which can cause a lot of performance issues. However, the main reason that hardware designers don't just increase associativity to increase cache performance

is that hit rate is only one aspect of cache performance. The other is runtime. Fully associative caches are significantly slower since blocks must be searched for iteratively. This would result in O(n) searching for each block, every time a memory access is made. A direct mapped cache, in contrast, is O(1). This has a pretty big performance hit in terms of runtime, again as explored above, particularly given this search must happen every memory access. The end result is that a fully associative cache is faster because it misses less but also slower because it takes longer to hit. Hardware designers have to decide between accuracy and search time.

4. The first suggested optimization I have is to implement a blocked matrix multiply algorithm. The data is clear that this will produce even at worst the same result as the regular algorithm, and could produce a significantly better result even without any further experimentation or effort. The second suggestion I have is to then adjust the blocked matrix multiply algorithm so that you minimize the number of conflict misses. This could be done either by writing out what the cache will do with a given blocking factor, then solving for a good blocking factor, or by some kind of hyperparameter optimization combined with a timer. I would personally err towards the second, it will give you real feedback and ensure you didn't miss anything. For example, while I get poor performance with the blocked matrix multiply on 512 by 512 with a block size of 32, I get significantly improved performance over regular matrix multiply with an arbitrarily chosen block size of seven. So, if performance is a big concern for this software, repeated testing while varying the blocking factor is the second recommendation I'd make to improve performance.

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-	442598400	106737210	114446790	0.517427978515625	0	230400	1.0
480	Blocked	32	449280000	206585790	18054210	0.080369524572649 58	0	345600	1.0
488	Regular	-	465095232	101687488	13074 1056	0.5625	0	23814 4	1.0
488	Blocked	8	493910656	243211586	3743742	0.015159591940450056	14498624	28160	0.001938488243509 3686

512	Regular	-	537133056	133891072	134544384	0.501216888427744	0	262144	1.0
512	Blocked	32	545259520	137084928	135544832	0.497175480769208	0	4194304	1.0

Table 4: Matrix multiply problem size table - Associativity = 2

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-	442598400	106735800	114448200	0.517434353298611	0	230400	1.0
480	Blocked	32	449280000	198872535	25767465	0.114705595619658 12	1831800	1624200	0.469965277777778
488	Regular	-	465095232	101687488	130741056	0.5625	0	238144	1.0
488	Blocked	8	493910656	243293864	366164	0.014826422372227 58	14526784	0	0.0
512	Regular	-	537133056	133012480	135422976	0.504489898681640 6	0	262144	1.0
512	Blocked	32	545259520	136249344	136380416	0.500240384615384 7	0	4194304	1.0

Table 5: Matrix multiply problem size table - Associativity = 8

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-	442598400	96768000	124416000	0.5625	0	230400	1.0
480	Blocked	32	449280000	209952000	14688000	0.065384615384615 39	3348000	108000	0.03125
488	Regular	-	465095232	101687488	130741056	0.5625	0	238144	1.0
488	Blocked	8	493910656	243111535	3843793	0.015564729990356798	14526784	0	0.0
512	Regular	-	537133056	117440512	150994944	0.5625	0	262144	1.0
512	Blocked	32	545259520	254803968	17825792	0.065384615384615 39	4063232	131072	0.03125

Table 6: Matrix multiply problem size table - Fully associative

Replacement Policy

I got roughly similar performance for all three of my replacement policies. The random replacement policy had a lower read miss rate and a higher write miss rate. The FIFO and LRU algorithms had roughly similar performance, in my data at least.

I would say I got the best results for Random. Given that Random had pretty similar performance and almost no implementation or performance cost (no queue to maintain), I would recommend using random when running this particular instance.

Replacement Policy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %

Random	449280000	209788731	14851269	0.0661114182692 3077	1112293	2343707	0.6781559606481481
FIFO	449280000	206409810	18230190	0.0811529113247 8633	0	3456000	1.0
LRU	449280000	206585790	18054210	0.0803695245726 4958	0	3456000	1.0