Assignment 4

Hamlet Fernandez

February 23, 2021

1 5.11

A modern laptop processor has three levels of caches, with the largest one approximately 16MB. Take a picture with your smartphone and check the size of the file.

1.1 A

How many photos from your smart phone would be required to fill the cache?

My picture was about .758 MB therefore to fill a 16MB cache, $\frac{16}{.758} \approx 21$ or 21 photos.

1.2 B

Chances are your estimate is size is based on compressed version of the photo. This is convenient for storage, but often difficult for modifying it. The uncompressed version might be 5 times larger; how many uncompressed photos from your smart phone would be required to fill the cache?

The uncompressed version wwould have a size of $5 * .758 = 3.79$ and $\frac{16}{3.79} \approx 4$ or 4 photos.

2 5.13

A modern smartphone such as the iPhone 12 has a display with 2,532x1,170 resolution of full color (perhaps 3 bytes per pixel).

2.1 A

How much data is required to represent the full image on the screen?

Data for the full image would be 3 bytes per pixel or $3*2532*1170 = 8,887,320$ bytes. This is roughly 8.8 MB.

2.2 B

Clicking on a web page often produces nearly a full new image (and more if you scroll!). Given for a 1.3MB average page, how much data is presented?

Data presented is the sum of download size and resolution: $1.3 + 8.8 = 10.1$ MB.

2.3 C

When using multiple applications, users often "flip back and forth", how much data has to be moved by the computer for each screen "flip"?

With a minimum of 8.8MB for the screen and 1.3 MB per page the computer has to move $8.8 + 1.3 * numFlips$ where numFlips is the number of pages flipped to. SO, for a flip from 1 page to another $numFlips = 2$ therefore we have $8.8 + 1.3 * 2 = 11.4$ MB.

3 5.16

In modern processors, the caches are large, as much as 8-16MB. In this question, we will explore what happens to programs that use only a small amount of data. (Hint: this is a good way to make your program go fast!) Assume you have a processor with the three level cache hierarchy describe in Section 6.2, and you want to execute a program that sums the values (long int's) in an array with the following numbers of elements. Considering only the memory references, how long with the program take to execute for an array of the following sizes:

3.1 A

1,024 elements. The first time? two times? Explain why

We know long ints are 4 bytes in size therefore for 1,024 elements there needs to be $1024 * 8 = 8192$ bytes of memory accessed. We know that the L1 cache holds 32 KiB of data or 32,768 Bytes so all the memory needed for 1,024 elements can fit within the L1 cache. Therefore, 1024 takes 4 cycles to process the first time, and 4 cycles again the second time since the cache has been warmed up and there are no cold misses.

3.2 B

8,192 elements. The first time? two times? Explain why

In a similar fashion to part a, we check how many caches the elements need to use. $8 * 8192 = 65536$ bytes is greater than the 32768 bytes of the L1 cache but less than the 1,048,576 bytes of the L2 cache. Therefore, 8192 elements takes $4 + 14$ cycles to process, or 18 cycles.

3.3 C

65,536 elements. The first time? two times? Explain why.

For 65,536 elements, the data required does not just fit in the L1 cache, but also requires the L2 cache. 4 bytes per element means we need $8*65536 = 524288$ bytes of memory which is greater than the 32768 bytes held in the L1 cache but less than the $1,048,576$ bytes of the L2 cache. Therefore, 65536 also takes $4+14$ cycles to process or 18 cycles.

3.4 D

524,288 elements. The first time? two times? Explain why

Similar to part C, the memory required to sum 524288 elements does not fit solely in the L1 cache but also does not fit in just L1 and L2. Therefore we need L3. The memory required is $8 * 524288 = 4194304$ Bytes. This value is greater than the number of bytes for L2 but less than those in L3 which means we can processs 524,288 elements in $4 + 14 + 60 = 78$ cycles.

3.5 E

4,194,304 elements. The first time? two times? Explain why.

Similar to part D, 4,194,304 needs all 3 cache levels beacuse the memory required is $8 * 4194304 = 33554432$ which is greater than L2 memory 1048576 bytes and less than L3 memory 39,845,888 Bytes (38 MiB). Thus we again have a latency of 78 cycles.

4 5.25

Take the matrix multiply example from Figure 4.19. We will analyze reuse and cache performance for a variety of matrix sizes. For all parts of this problem, consider only the memory references for the matrix access.

4.1 A

Calculate the reuse distance for matrix sizes (N values) of 16, 64, and 128. You can assume that the values in the A[],B[], and Result[] matrices are 64-bit quantities. The reuse distance on the x-axis should have units of memory references, words, or bytes, but make sure it is labelled clearly. How does the size of the matrix affect the reuse distance graph?

For a matrix multiply program that does not tune for reuse distance with tiling, (tiling with k = 1), the reuse distances of matrix multiply are equivalent to the size of 1 row or N. This means that for the first access of data in a matrix of size N the reuse distance is N but for the second access, it is N^2, for the third it is N^3, and etc. Thus, the following:

N	1st reuse distance	2nd reuse distance
16	16	256
64	64	4096
128	128	16384

The size of the matrix increases reuse distance because addresses are only reaccessed every N memory references. This is due to the fact that without *kxk* tiling, the program, like the one in Figure 4.19, is looping through the matrices data point by data point therefore the distance between A[1][1] and A[2][1] (different row same column) is N.

4.2 B

Consider the performance of a cache hierarchy with L1: latency = 4 cycles; L2: latency = 12 cycles; main memory = 300 cycles. For each size, calculate AMAT, estimated from the reuse distance profile. How does the matrix size affect the AMAT for the memory hierarchy?

$N = 16$ Since the arrays A[], B[], and Result[] are 64-bit quantities we need to multiply the each data point they store by 8 bytes to get the quantity of memory they require. For N = 16, the number of data points is N^2 for a 2D matrix. Therefore, for matrix size N = 16, it requires $16^2 * 8 = 2048$ bytes which fits solely in the L1 cache. To calculate the AMAT we want to use the values for filter latency, hit rate, and fraction of memory operations from the previous cache which in this case is 1. Therefore, for a level 1 cache with a filter latency of 1 nano second, a hit rate of 97%, and accepting all the memory accesses, the AMAT is $1 * .97 * 1 = .97$ nanoseconds per memory reference. Thus for N = 16, the total time to access all the values is $.97 * 256 = 248.32$ nanoseconds.

$N = 64$ Similar to $N = 16$ we find the amount of memory required which for N = 64 is $64^2 * 8 = 32768$ bytes. These calculations fit inside the L1 cache. We can use the same AMAT value from N=16, or .97 nanoseconds per memory

reference. This means for $64^2 = 4096$ memory references the program takes $.97 * 4096 = 3973.12$ nanoseconds.

$N = 128$ Lastly, $128^2 * 8$ does not fit in the L1 cache but requires the L2 cache. This means that while 4096 elements take 3973.12 nanoseconds to process, the other $12288(128^2 - 4096)$ are processed by the L2 cache and therefore are processed slower. For the L2 cache, the AMAT per memory reference is $3.5 * .77 * \frac{4096}{131072} = .67$ nanoseconds. Thus, we can find the latency for the remaining operations of $.67 * 12288 = 8232.96$ nanoseconds. We sum the latencies together and get $3973.12 + 8232.96 = 12206.08$ nanoseconds.

As the matrix size increases AMAT also increases but not linearly because the different levels of caches have exponentially increasing latencies. This is consistent with the effect of increasing matrix sizes on reuse distances because there is more data to access in between reaccesses of the same address.

4.3 C

Matrix multiplication is considered a very friendly application for caches. Can you rewrite the application to give even better cache results (and reuse distance graph) than we saw aove in the earlier parts of this question?

We can use the example given during Lecture 12, slide 29 titled "Tuning Reuse Distance". With the matrix multiply example of figure 4.19 there have been memory accesses at every point and calculates in rows of size N, which means the reuse distance has also been size N (k = N). However, if we were to implement the tiling program as seen below, we get to reduce the reuse distance by breaking up the data into tiles that can be calculated in parallel. For example, for our matrix of size $N = 16$, if our tile size, k was 2 then our reuse distance for the first set of accesses would be $2 * 2 =$ instead of 16.

```
//Taken from Lecture 12, Slide 29
// Tile with blocks of size k
//
for (i = 0; i < n; i+=k) {
    for (j = 0; j < n; j+=kj) {
        Block_clear(C,i,j,k);
        mmult(k,A(i,j),B(i,j),C(i,j)
    }
}
```

5 Additional Problem

xtend the idea of optimizing reuse distance in an application forbetter dynamic locality. Consider two sorting algorithms, the bubble sort discussed inChapter 2,

and a Hoare's Quicksort (a recursive sorting program) that you may have seen inan algorithms class, or can quickly find information on the web.

5.1 A

onsider reuse distances for reference to the array being sorted (let's call it A[] forconvenience. For bubblesort, plot the reuse distance for references for an array of sizeN=1024. For simplicity, you can assume that the algorithm always swaps items when there is a conditional test.

5.2 B

onsider reuse distances for reference to the array being sorted (let's call it A[] forconvenience. For the recursive quicksort, plot the reuse distance for references for an array of size N=1024. For simplicity, you can assume that the algorithm always swapsitems when there is a conditional.

5.3 C

These two sorting algorithms compute the same result (a sorted array). How much of a difference does the choice of algorithm make for the reuse distance profile?

5.4 D

ick some larger, more realistic sizes of arrays, say 1 million entries? Mapping these datasets onto a realistic memory hierarchy such as the Skylake processor, use the reuse distance model you created to estimate the difference in average memory access time(and therefore performance) in sorting a 1M element array.

6 7.1

While single-thread performance increased rapidly under Dennard scaling through 2006 (52% per year as shown in Figure 7.1), the rate of increase dropped to only 21% per year after 2006.

6.1 A

Calculate the "lost" single-thread performance due to the end of Dennard scaling from 2006 through today. If Dennard scaling had continued, how much faster would single thread performance be today?

With a 52% increase a year since 2006 we would have had a 1.52 time increase every year for 15 years since 2006. This means we would have had a $1.52^15 = 534.18$ time increase if Dennard's scaling continued. However, without Dennard's scaling we now only have a $1.21^15 = 17.5$ time increase. We lost about 97% ($\frac{17.5}{534.18} = 3\%$) of the speed of our progression.

6.2 B

Instead of faster single-thread performance, Moore's Law has yielded parallelism over the past 15 years. Supposing that the number of cores doubled every 3 years since 2006, starting with 1, how many cores would we have today? What parallel efficiency would we have to achieve across those cores to deliver the same total compute performance of a single thread in (a)?

Doubling the amount of cores every 3 years for 15 years results in $1 * 2^5 = 32$ number of cores. To acheive the same compute performance of a single thread, parallel efficiency would have to keep up with the 17.5 (per core) time performance increase since the end of Dennard's scaling. Therefore, we would need a $17.5 * 32 = 560$ time increase since 2006 to stay consistent.

6.3 C

Discuss how likely it is that a set of applications can achieve greater or lesser parallel efficiency than the threshold in (b). Give an example for each of one that is less, equal, and greater than the threshold and give a plausible explanation why

It is unlikely because the implementation of parallel effieincy with multiple cores creates a new cost, a cost for the communication required between cores. Parallel efficiency complicates the improvements reaped from single thread performance but does not necessarily optimize or remain consistent with the linear speedup of multi-core parallel computing. However, the ideal situation, where parallel computing actually improves performance at a rate faster than single thread computing is if each core served a unique purpose and part of computing process and if communication in between was negligible. Currently, two CPU's each function as their own CPU which does increase performance compared to a single thread CPU because the cores can do more work but each core does not indivdually work faster. However, if a single thread processor were to split up actions into other processors, then the work that can be done individually is lowered, speed is increased, and the benefits of parallel effiency can be reaped. An example of a set of applications where parallel effieicny is equal is simply where each of the 32 CPU's calculated in part B function as fast as a single thread processor. The 32 CPU's can do more work and can compute faster because they can split up data and process them in parallel but each CPU is just as fast at processing than other. Lastly, multi-core could slow down performance is the communication between them took too long. If it takes too long to

copy data over between cores, so long that the processes don't occur in parallel, then the performance benefits of parallel efficiency are ignored.

7 7.4

Using the OpenMP pragmas used in Section 7.3.1, or you can find the full standards at www.openmp.org, write a dot product program for two arrays A[16384] and B[16384].

```
Result[16384]; //result array
n = 16384;

#pragma omp parallel for
for (i = 0; i < n; i++) {
    Result[i] = A[i] * B[i];
}
```

7.1 A

Explain how many elements of of the A[] and B[] arrays will be computed by each thread on an 8-core machine. A 32-core machine?

For each core on an 8 core machine, $\frac{16348}{8} = 2048$ elements will be computed. For each core on a 32-core machine, $\frac{16348}{32} = 510.875$ elements which means 31 cores can compute 510 elements each and 1 core must compute $28 + 32 = 60$ elements.

7.2 B

Look at the worksharing-loop construct, can you change the number of elements that will be computed by each thread. Explain when and when not doing this would increase performance.

The number of elements that can be computed by each thread can be changed with using more OpenMP pragmas or reorganizing the how many computations each core does. For example, in the previous problem, a dot product of 16348 elements across 32 cores can be done either with 31 cores computing 510 elements and 1 core computing 60 or 31 cores computing 527 elements and 1 core computing 11 elements. The difference between the parallelisms is that for the former scenario, 31 cores would have to wait for the last core to finish while in the latter 1 core would have to wait for 31 cores. The latter would not be more ideal because while less cores are waiting idly, less cores have time to do work. In other words, while the 31 cores are waiting for the last to finish in scenario A, they can be doing other work. Lastly, adding more pragmas between loops can help add parallelism but this exponentially strains the processor and could

reduce performance by the number of computations exceeding the number of threads allowed.

This can also be done by swithcing the construct among omp, sections, single, and master. These constructs help control the amount of computations any thread executes. For example, *single* forces a sole trhead to compute the code block while *omp* allows all threads to split up the code and execute in parallel. Switching controls from *omp* would increase performance if the code within the loop body was not truly independent of each other. For example the construct *sections* would work better for code where previous sums were needed later therefore emphasizing a sequence order of execution. Switching from *omp* would not be beneficial in cases when parallelism optimizing the execution of the program, which occurs in many code blocks that are "embrassingly parallel" much like the dot product program above.

7.3 C

A common element critical to efficiency is how OpenMP parallel loops do reductions. Using the dot product example, explain why this is, and how much difference it could make in parallel performance.

Reductions help bring the results of code running in parallel together. With a program that computes the dot product at each i-iteration, a reduction operator would keep track of the value of the dot product at each iteration but could sum the values of previous iterations, allowing for the dot products to be summed at the end at the same time as the dot products are computed. OpenMP parallel loop reductions effectively implement parallelism for functions that want to do other operations on loops, like taking the sum of all the dot products. By keeping a local variable that slowly builds towards a global variable, the program's progress can be tracked which creates another parameter that could allow for parallelism. For example, while the reduction operator global variable is less than 500 sum the dot products between 0 and 8174. This breaks up the sums into two groups and these groups are indeed independent and can be parallelized further.

8 Extra Credit

The reuse distance graphs of figure 5.30 were really helpful and as I worked through them for the pset I begun to understand things a bit more. However, I think I'd also appreciate a worked example of reuse distance using matrices and different k values. Once I understood tiling and how reaccessed addresses appear at after k addresses, I could visualize reuse distance as a concept a lot more. A website that helped me visualize it is here