CITS 5507 Project

This is Jared Healy (23398223)'s CITS5507 Project write-up. There wasn't a suggested word limit, so I won't go to crazy, but I'll try to explain to you my thought processes for the decisions that I made.

Description of How I have Parallelised the System

The way that the algorithm has been parallelised is through the use of the pragma directive to parallelise the for loop in `convolve.c`. The iteration has been left in 2 dimensions and not collapsed intentionally, to ensure that no false sharing occurs, as the memory that each thread will be working on will not be close in memory. This could theoretically lead to performance to very tall or very wide arrays, but those cases are not common use cases (when I say tall or wide I mean like 5 or 6 on the short dimension - not very practical). Another consideration was the schedule method to use. I tested the file on schedule static as well as guided and dynamic and found that the dynamic allocation performed best at large matrix sizes.

Description of How I am Representing My Arrays

The way that all the arrays are being represented in memory is as a `Matrix` struct, which is defined as follows:

```
typedef struct matrix {
    long height;
    long width;
    float *array;
} Matrix;
```

From this example, you can see that the arrays are stored as flat arrays with a height and width stored. To accommodate for the boundary access and provide the padding, I have a function called `accessMatrixOrZero`, which ensures that the boundaries of the array are respected in both dimensions.

Have I considered the cache when developing my algorithm

While the cache was admittedly not my first issue when designing the algorithm, it was one of the main areas that I worked to optimise. Almost every kernel access is linear in memory, making use of the flat array to ensure that it is accessed in order. The feature map is less cache-optimised, but it will be accessed in order for each row of the kernel (that is, if the kernel is m * n, then each m accesses will be in order before the array has to move to the next "row"). Having said that, I can't think of any way to improve this strategy; given that the algorithm needs to access the data in a grid, the nature of a one dimensional array means that there needs to be a jump between rows, and there was no data structure that would improve the cache hits. (My understanding about 2D

arrays is that they end up the same as 1d arrays in memory anyway, and no other data structure really made sense.)

Performance Metrics and analysis of solution

To test the performance of my algorithm, I wrote 5 competing algorithms, all of which could be run by a command flag (see the README). However, when I first did this, I ran into a very strange issue: No matter what strategy I used, I seemed to get the same results. This wasn't happening on my pc when I tested locally, but only on Kaya, which was the most frustrating part. Obviously, this couldn't be right, but it seemed that everything I tried didn't change the speed that each strategy ran on. However, I found a paper on github that analysed an algorithm doing matrix convolutions. I'll admit that most of this paper I didn't understand, but it had a graph on page 4 that showed a massive drop after 8 cpus were used, and I was using 10. After changing my cpus to 8, I discovered the speedup that I expected. I then ran 2 stress tests on each algorithm, first with a 10k by 10k feature map with a 5 by 5 kernel and then with a 50k by 50k feature map and a 20 by 20 kernel. The results are in the table below.

Test	Static	Dynamic	Guided	Static_Collapse	Linear
10k*10k	1.378844	2.889291	1.357622	1.288325	9.866833
50k*50k	551.771156	520.999243	521.617569	561.596240	4471.680480
10k vs Linear	7.155	3.415	7.268	7.659	1
50k vs Linear	8.104	8.583	8.573	7.962	1

Explaining each algorithm

Static The static algorithm that I used is simply just a naïve implementation, with a few measures to try and avoid false sharing. The outer for loop is parallelised using the `omp parallel for` directive, then specified to use static (which is the default behaviour), but the inner loop is not parallelised, which is to help avoid false sharing. Another way that false sharing is prevented is by putting the matrix y loop outside the matrix x loop, meaning that the array access should be seperated by the width of the array, preventing different threads accessing the same data. The static solution

Dynamic The dynamic allocator also showed a large speed-up, maxing out ay 8x the linear approach. It is almost identical to the static implementation, with the only difference being the `pragma` directive specifying that the for loop is to use dynamic. The dynamic allocator underperformed at lower matrix size, but performed marginally better at the larger matrix size, which makes sense as the dynamic allocation adds scheduling overhead to the runtime, which is felt more at smaller loads, and then helps to improve performance at higher loads.

Guided The guided allocator was once again a carbon copy of the other two, using the `guided` allocator instead. It showed the best performance over both tests, being able to almost match the dynamic solution in the second test and the static_collapse allocator in the first. This can be understood by the fact that the guided allocator has been optimised by the OMP library to best attempt to provide the performance of dynamic without as much of the overhead.

Static Collapse The other method that I tried was using the static allocator with collapse. The reason that I had steered away from collapse earlier is over false sharing concerns, but the best way to know if that would be an issue would be to run test it. After testing, it seemed that false sharing was not an issue (as the speed-up was still close to the 8x speed up expected), but that it was not as fast as any other strategy on the larger data set. I'm not exactly sure what lead this strategy to not scale as well, but it was interesting to try nonetheless.