Concurrent Systems Lab 1
Documentation
Optimising a convolution routine

Sean Durban (14320715) Luke Egan (14320891)

Introduction

The purpose of this lab is to better understand vectorisation, code optimisation and parallelisation by parallelising and optimising an efficient convolution routine. We were given a basic algorithm to start with and we will optimise this basic algorithm taking into account locality of data accesses and the multiple available processor cores of the target machine.

Our target machine is stoker.scss.tcd.ie. This machine has four processors. Each processor has eight out-of-order pipelined, superscalar cores and each core has two-way simultaneous multithreading 64 threads of utilisation.

The width & height of image, kernel order, number of kernels and channels are given as command line arguments. The image is stored as a 3D array of floats. Which is organised by width, height and channel number. The kernel is stored as a 4D array of floats organised by kernel number, channel number ,kernel order and kernel order. Both the image and the kernel are randomly generated by the program with the given parameters.

We will use *'x86intrin.h'* for SSE instructions. This header file is part of gcc, which is used to compile the program with the highest level (03) of compiler optimisations we used the following command:

"gcc -03 -msse4 -fopenmp conv-harness.c"

To test the program we time, the time it takes to run our optimised routine and the basic routine. The difference in times is calculated are printed to the console in terms of the speed markup. The two resulting outputs are also compared and the absolute sum of difference between the correct basic algorithm and our optimised one is calculated and printed too.

Result of optimisation

Test cases:

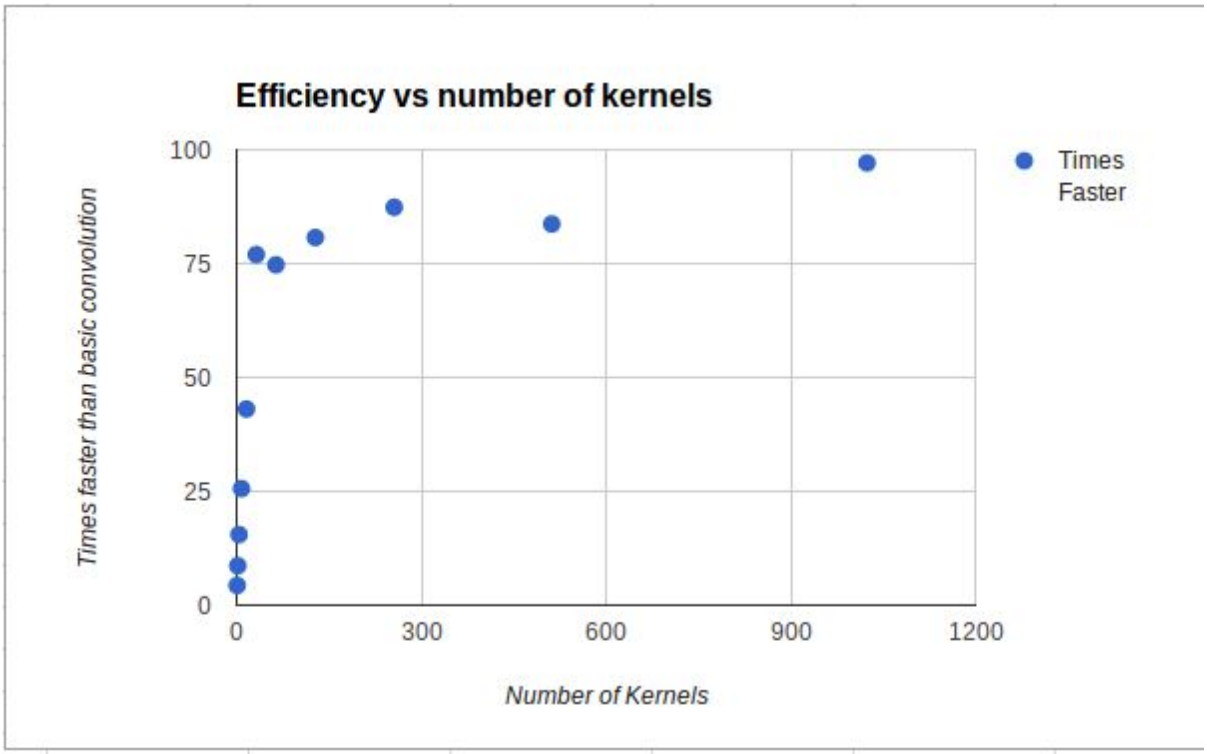
Input	Times Faster	Time (in ms)	Sum Of Differences
128 128 5 32 100	75.101	43943	0.029286
32 32 3 64 1024	101.634	18582	0.017451
255 255 1 63 127	67.004	47004	0.017614
192 192 7 1 512	17.818	117487	0 (no sse)

In our program, the sse depends on the amount of channels being greater than 4 to be useful. This is evident looking at the result of the fourth test case. This means the result is more accurate, as no error is accumulated in the addition of floating point values in sse registers, but this come with the tradeoff of lower speeds.

Investigation to see most efficient amount of kernels:

The amount of threads openmp uses for this computation is dependent on the amount of kernels in the input. This is evident looking at the column “Times Faster” which shows how many times the optimised code is faster than the unoptimised code given to us at the beginning of the project.

Number of kernels	Other Input	Times Faster	Time (in ms)	Sum Of Differences
1	200 200 5 40 %d	4.34	40978	0.000851
2	201 200 5 40 %d	8.669	41261	0.001732
4	202 200 5 40 %d	15.5	39114	0.003452
8	203 200 5 40 %d	25.612	39039	0.00696
16	204 200 5 40 %d	43.070864	39103	0.013863
32	205 200 5 40 %d	76.992	39285	0.027763
64	206 200 5 40 %d	74.743	76636	0.055326
128	207 200 5 40 %d	80.736	129342	0.110913
256	208 200 5 40 %d	87.376	255300	0.222188
512	209 200 5 40 %d	83.7	539358	0.4431
1024	210 200 5 40 %d	97.108	907120	0.887517



The graph above shows the correlation between the number of kernels in the input and the efficiency of the program. Running on a machine with a large amount of cores, increasing the amount of kernels, increases the efficiency of the program, up to a certain point.

Method

Openmp:

First we looked at openmp and which of the loops could run in parallel. We found that the first loop (which goes through each kernel) was a clear candidate. With the following line of code:

#pragma omp parallel for private()

An iteration of the loop is then sent to each available core.This allows the program to apply each kernel to the image in parallel, resulting in a theoretical speedup of the number of kernels - up to the max number of available cores. Some variables have to be kept private for each instance of the program, these are specified in the omp statement.

Depending on the number of kernels the speedup can be minimal, changing the line of code to the for loop underneath (width) then the speedup can be greater than before. So there's a minimum number of kernels where the speedup becomes more significant.

Beyond this we did not run any other parts of the routine in parallel, one reason for this was due to the rest of the routine being run in parallel already. Also running code in parallel has some overhead from setting up the process on another core, loading data etc. so certain parts which could be parallelised are not because there's is no optimisation gained from it.

For loop and Kernel Changes:

After attempting to vectorise the code, we came to the conclusion that it would not be viable in the current form. Therefore we found we would have to change the structure of the image or the kernel. The obvious choice is the kernel seeing as in most cases it is significantly smaller and the changing the structure would affect the time minimally . The kernel is changed once per execution and to allow the last index to be the channel number. i.e the kernel went from:

kernel[m][c][x][y] ----> kernel[m][x][y][c]

Where: m = kernel#, c = channel#, x & y =kernel order (indexes into kernel)

The for loops were also reorganised, the loop now iterates through each channel at a certain kernel order (x,y). Both this and the kernel change were done to ensure all memory reads and writes are contiguously as possible and in turn improving cache coherency of the routine. Now as we iterate through the program, the address of the kernel only needs to be incremented and not calculated. The kernel change also allows the vectorisation discussed in the next section.

The trade off with this change is that we added a constant overhead to the program. Although the max kernel order is 7 ensuring the overhead is minimal in the worst case, it is still additional time nonetheless. The number of channels also affects the time this takes, but given that this optimisation is with the number of channels and a larger number of channels results in a larger speedup, then it is justified.

Vectorisation:

As explained above we changed the kernel structure in order to vectorise efficiently. The program using SSE instructions loads up 4 channel values from the image at a certain

position and loads up 4 channel values from the kernel at the corresponding position. These vectors are then multiplied and the result is added to a rolling sum for the current pixel at (w,h). When the program has applied the kernel and added all the results , then all 4 lanes of the sum are added together and stored in the correct index of the output.

One obvious issue is that this will only work when number of channels is divisible by 4, otherwise the behaviour is undefined or will result in a segmentation fault. To counter these we precalculate the point at which the loop will hit this case, when it does we break out and just load these remaining channels one a time. With the max remaining channels being 3 then the routine should only loop through 3 channels individually in the worst case. One issue in this case is with the loading of the vector, we found that using load_ps here again always resulted in a segmentation fault. Therefore we had to change it to loadu_ps which is slower in most cases, which is shown in the running times. This is the case because the image and kernel are no longer loaded on aligned addresses.

In cases where the number of channels is <4 then no vectorisation will occur. This case is handled by just going through the image as with the basic algorithm. This meant that the restructuring of the kernel was redundant and was removed for this case. Therefore no vectorisation and just openmp is used to optimisation, resulting in a significant slow down vs other inputs. This is clearly shown in the results gathered.

Another issue we encountered was that with larger inputs, the more vectorisation that occurred the more the difference increased and was over the predefined epsilon. To address this we added the sum vector together and added it to a double sum value more often. This lead to an increase in accuracy and the difference being decreased below epsilon. This of course did lead to more instructions and in turn a slow down to the routine. But we figured the minimal slowdown was worth being much more accurate in this case.

One other trade-off we had to make with the vectorisation was to use floats or doubles. With doubles our accuracy would be much improved but our performance would effectively be halved. As we would only be able to load and manipulate 2 values at a time rather than 4 with floats. But the image is currently stored as floats and through testing we found that the sum of difference between our output and the correct output were small enough to justify this decision. We ultimately decided on choosing floats and prioritising speed over accuracy. Also the lab allowed for a margin of error and should the lab had prioritised accuracy over speed then we would have gone with doubles.

Process stalls:

From running perf we realised that the SSE instructions that loaded the values from the image and the kernel were a main cause of a large percentage of front end stalls. We figured this could be improved and in turn the speedup by restructuring the image, although it was defined in the lab specification not to do this.

Register limitations:

With the current configuration of the stoker machine (Intel xeon e7-4820), we make use of 128-bit sse registers. Which allow for 4 floats or 2 doubles. An upgrade to the e7-4820 v2

allows avx operations, which use 256 bit registers (8 floats). This would theoretically double the speed of the program. Newer processors even have 512 bit wide avx registers.