Parallel Multi-Channel Multi-Kernel Convolution

- James Tait, Seamus Woods, Tom Wisniowski

Algorithmic Improvements:

The first improvements we made to the function were simply modifications to the existing code. We noticed that the output was continuously overwritten when placed within the ‘c’ for loop. We moved it out of that for loop so the final output was stored just the once, saving us lots of unnecessary memory accesses.

We also noticed that ‘a’ did not change when ‘m’ did, so we calculated two ‘b’s, and subsequently two ‘sums’, at once, in order to evaluate ‘a’ half as many times. We attempted at one point to move the whole ‘m’ loop to be the innermost one but that ended up not yielding any speed up and made it much more awkward to do two kernels at a time, so we ditched that change.

SIMD SSE - x86 Intrinsics:

The second optimization we added to the algorithm was the __m128d data type (2 64 bit double precision floats). The reason we didn’t use the __m128 data type (4 32 bit precision floats) was due to a loss in accuracy, which prevented us from obtaining the correct answer.

Using this order meant we were able to halve the amount of time a for loop was run. We implemented this data type for the number of kernels and the number of channels. This was particularly good for the number of kernels since it is the outermost loop, meaning the entire code was only being executed half the amount of times.

OpenMp:

The third improvement we added was using the “#pragma omp parallel” command. This allowed us to run parts of our program using multiple threads concurrently, dividing up the work and speeding up the code by at least a factor of 2.

“Parallel For” divided the iterations of a for loop between the threads while using the extension “collapse()” merged several for loops into an iteration space and divided according to the schedule clause. The sequential execution of the iteration in all associated loops determines the order of the iterations in the collapsed iteration space.

Problems we ran into:

We ran into several problems during the assignment, mainly trying to implement the __m128 data type and using openmp. When trying to implement the __m128 data type, we were losing accuracy along the way which resulted in the wrong answer. We messed around with this for quite a while before deciding to go back to using the __m128d data type.

OpenMP threads were also sharing loop variables, which meant parts of the ‘image’ were being skipped etc, and so the output we were getting was totally wrong. Once we added in the ‘private()’ clause, with the right variables inside of it, the algorithm was working properly again.

Soon we noticed that running multiple threads on small inputs slowed the program down rather than speeding it up. Threads come not free, but with overhead like context switches, and locking mechanisms. This is only worth the cost when you actually have more dedicated CPU cores to run code on. After finding this out we’ve put in “if statements” to insure that threading will only be applied when large inputs are being used.

What we learned:

Optimization of code and algorithms is hugely important in many different areas of computer science. However, the effort it requires to change a sequential program into a fully correct concurrent one, is often not worth it, particularly if execution time is not critical to the functionality of the program.

It’s not always worth the effort of threading or parallelizing, since there can be diminishing returns from doing so.

Not to thread something where the amount of work is small, since there is a significant cost in executing OpenMP parallel constructs. We used if statements in our OpenMP calls to avoid this.

Timings:

Input: 256 256 3 1024 1024

Average Time: 44.92 Seconds

Input: 128 128 7 512 512

Average Time: 15.4 Seconds

Input: 128 128 3 512 512

Average Time: 3.88 Seconds

Input: 16 16 1 32 32

Average Time: 2089 Microseconds