CS257 Advanced Computer Architecture Coursework Assignment: ACACGS

Neema Raiyat
2001540

March 2022

Contents

1 conjugateGradient.c

conjugateGradient.c is the main file where all the computation occurs. It utilises functions defined in other files, namely *waxpby.c*, *sparsemv.c*, *ddot.c*. To optimize *conjugateGradient.c*, the main `for` loop had the first iteration peeled from it. The first iteration seeks to check whether `k == 1`. Peeling this loop makes the need for the `if` statement in the `for` loop redundant and hence reduces overhead since the condition no longer needs to be checked in the proceeding iterations of the loop. It also does not affect floating-point correctness. Furthermore, pointers `*r`, `*p` and `*Ap` were changed so that they are now allocated memory using *Intel intrinsics*: `_mm_malloc()` [1] with an alignment of 64 (gathered by `cat /proc/cpuinfo` and the use of double precision floating-points (8 bytes)). This is so that functions that utilize these pointers can now use aligned intrinsic operations, for example: `_mm256_load_pd()` as opposed to `_mm256_loadu_pd()` which seeks to load **unaligned** memory into vector registers (which is slower than already aligned memory).

2 waxpby.c

The first optimization made to this file was to remove the overarching `if` statement which would check whether `alpha` is 1, and then whether `beta` is 1 etc. This is because only the first block in the condition would be executed as `waxpby()` is always called with `alpha` equal to 1. Therefore, we can also remove the argument alpha from the `waxpby()` function. Moreover, intrinsics were used to vectorise the `for` loop. **AVX2** intrinsics were used so that 256-bit vector registers could be used as opposed to the 128-bit vector registers in **SSE4.2**. This is because 256-bit registers can operate on 4 double precision floating-point numbers concurrently whereas 128-bit can only operate on 2. Since all pointers passed to *waxpby.c* have been made to be aligned in *conjugateGradient.c* and *generate_matrix.c*, vector operations used can be quicker as opposed to their unaligned counterparts (as previously explained). Since 256-bit vector registers operate on 4 double-precision floats simultaneously, the `for` loop in *waxpby.c* must be unrolled by a factor of 4. In addition, **OpenMP** [2] was used to parallelize the `for` loop and hence optimize the function further: `#pragma omp parallel for`. `schedule(static, 4)` was also used so that iterations are divided into chunks and are assigned to threads at compile time, rather than at runtime (which would be achieved by `schedule(dynamic, 4)`) and therefore reducing overhead during runtime. Since each thread will be accessing the same memory location pointed to by `*w` then the result will be dependant on the order of threads that have accessed it and hence lead to a race condition. This will negatively affect floating-point correctness. However, testing has shown that the variation in results is very minor, but the performance gain is significant.

3 sparsemv.c

The first optimization was to unroll the inner `for` loop. This does not affect floating-point correctness. Secondly, **OpenMP** was used to parallelize the outer `for` loop. Multiple threads will be attempting to access the same memory locations, e.g.`*A`, ultimately leading to race conditions. Despite this negatively affecting floating-point correctness, the performance gains were far more significant than the loss of correctness. The following compiler hint was given: `private(j, sum)`. This is because these variables are not needed outside of the outer `for` loop, and their initial values are set to 0 anyway, so `firstprivate` is not needed. Loop interchange was not used since the matrix was already being traversed column-wise. Loop interchanging would result in row-traversal, which in *C*, involves more cache-line read and writes (unlike in *FORTRAN*) and hence be slower (but not affect floating-point correctness). A further optimization that could be effective (and not harm floating-point correctness) would be to use loop blocking. This would improve the spatial and temporal locality of the memory used since the cache-reuse of a 'tile' of memory is improved.

4 ddot.c

In this function, a single value is being updated (and therefore shared) across all threads. Rather than using `#pragma omp critical` or `#pragma omp atomic` to prevent race conditions, reductions were used. Reductions enable us to declare both the variable and operation to apply, and hence handle potential race conditions by handling the creation of private variables in each thread which then finally update a shared value.

5 Further Optimizations

Since **Intel(R) Core(TM) i5-8500 CPU** has 6 cores (and does not support hyper-threading), the number of threads used was set to 6, any more would harm performance as there would be contention between threads for a single logical core and hence cause thrashing in the cache. Additionally, the program is compiled using the `-O3` compilation flag for maximum optimization. This does not affect the correctness of the program.

References

[1] Intel Corporation. Intel intrinsics guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=6192,4260,6846,4497`, 2021. Accessed March 6, 2022.

[2] OpenMP. Openmp3.1-ccard. `https://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf`, 2011. Accessed March 6, 2022.