Uppsala Universitet

High Performance
and Parallel Computing
7.5 hp

Conjugate Gradient Method

Adam Orucu

May 28, 2021

1 Introduction

The aim of the project is to implement an algorithm and then investigate different optimisation methods that could apply for the chosen problem.

1.1 Problem Description

The problem I've aimed to solve and later optimise is the algorithm for conjugate gradient method. This method is used to find a numerical solution for systems of linear equations, with a positive-definite matrix. As it is often implemented as an iterative algorithm, it can be used to solve optimisation problems. This method is often compared to gradient descent as it can reach the optimal point in fewer amount of steps.

Pseudo-code of the iterative method is presented below.

Algorithm 1 Conjugate Gradient Method[1]

$r_0 := b - A * x_0$
if r_{k+1} is small **then**
    return x_0
end if
$p_0 := r_0$
for $k = 1, 2, \ldots, N$ **do**
    $\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} := x_k + \alpha_k * p_k$
    $r_{k+1} := r_k - \alpha_k * A p_k$
    if r_{k+1} is small **then**
        break loop
    end if
    $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} := r_{k+1} + \beta_k * p_k$
end for
return x_{k+1}

2 Solution

This section will describe the algorithm that was implemented before moving on to the tested and applied optimisation techniques. Whilst presenting the structure of the program it will also mention some of the conscious decisions that were made to obtain a more efficient program.

The program was constructed such that it will be easy to test for different data and matrix sizes. For this reason, it contains functions to read and write data from files. After storing the retrieved information for the matrix and vector values it calls the `conjugate` function, which aims to calculate the corresponding x vector of values. During the testing, the time loading and writing the data was disregarded and only the CG algorithm or its inner components were measured.

The structure of the implemented algorithm is similar to the one previously presented as pseudo-code with some small differences. Firstly, the first guess for x is always set to $\mathbf{1}$. Then memory for r, p and Ap is assigned. First two, used for the same purpose as previously shown while Ap representing the product of A and p since it is needed multiple times and to avoid the need of recalculating it unnecessarily. Next, in order to calculate r and later Ap, a function was implemented to calculate matrix-vector multiplications, which will be described later. Another place worth mentioning, that differs from the pseudo-code is the calculation of dot products such as $r^T r$. In order to not waste resources, these values are calculated by iterating over the elements of the vector instead. Last part worth mentioning, regards lines 8 and 9 of the pseudo-code. Since these two calculations are independent of each other and need the same number of iterations to calculate they have been "fused" into a single loop.

The generic matrix-vector multiplication function `mult`, used many times in this algorithm can be viewed below.

Listing 1: Matrix-vector multiplication

```
void mult(const int size, const double* __restrict m,
    const double* __restrict v, double* __restrict d) {
    double temp;
    for (int i=0; i<size; i++) {
        temp = 0.0;
        for (int j=0; j<size; j++) {
            temp += m[i*size + j] * v[j];
        }
    d[i] = temp;
    }
}
```

3 Optimisation and Results

The tests carried out throughout the optimisation process include both checking if the results are still correct (with a decided margin of error) and the time it took different parts of the program to finish their tasks. For this project only strong scaling was tested since it is difficult to calculate how much work will actually be needed to solve a problem with different matrix size and values - which would be needed for weak scaling. All he tests discussed in the report were carried out on `Ubuntu 20.04` with `Intel Core i7-10710U` and `gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0`. However the program was also run on the university computers (`Ubuntu 18.04.5 LTS, Intel Xeon CPU E5520, gcc 7.5.0`) to check for portability. The values presented in the report are the lowest values obtained after multiple runs of the same setup.

Before doing any optimisation some analysis was done in order to identify the parts of the program that take the most amount of time and can be prioritized during the optimisation process.

Timing analysis showed that almost all of the time was spent doing computation for the matrix-vector multiplication by `mult`. For a matrix of size 2000 by 2000; of the total 3.87 seconds, 3.86 was spent in `mult`. This suggests, that this function is the one that takes most time and should be first to optimize.

3.1 Roofline Model

While trying to figure out ways to optimise a calculation it is useful to plot a Roofline Model[4] and see if it is bound by the memory or the CPU speed. In this case, the program was run on an `Intel Core i7-10710U @ 1.10GHz`[2] which is able to obtain approximately 64 GFLOP/second for single precision and 32 GFLOP/second for double precision floating points and has a memory bandwidth of approximately 45.8 GB/s. With this information one can obtain the roofline model presented in figure 1.

3.2 Matrix-Vector Multiplication

Regarding the implemented matrix-vector multiplication function, one can see that it has 1/8 operational intensity, as it accesses the memory of two double precision floating points and makes two arithmetic operations. Hence, $I = \frac{2}{2*8} = \frac{1}{8}$. This value has been displayed in figure 1 and clearly can be seen that this makes the operation memory bound.

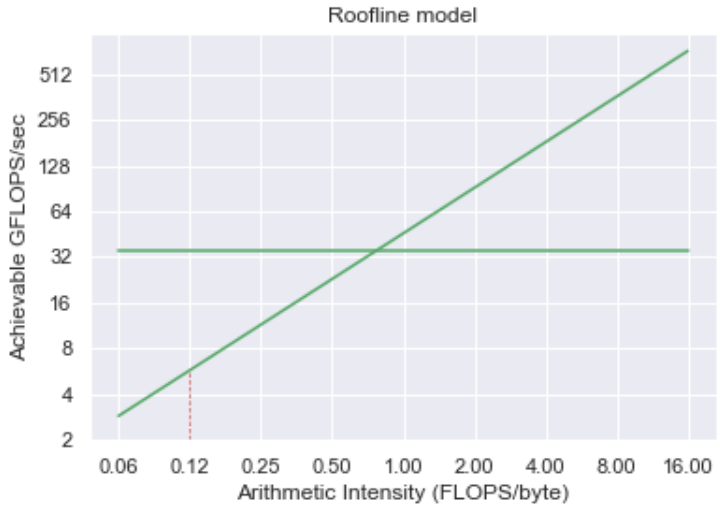

Figure 1: Roofline model for Intel Core i7-10710U while using 1 core with SIMD, I=1/8 shown with the dashed line

Since the bottleneck of the program is the speed at which the CPU is able to read and write the data, my first idea to optimise the calculation of this multiplication was to consider if the better use of CPU cache could improve the speed of the memory access and there for the operation.

In order to examine this I called `cachegrind`, which showed that the CPU has 32768 B of L1 cache among other things. In the `mult` function if the entire vector is not stored in the cache, then doing so might improve the memory access speed and therefore the computation time. If we assume a vector of size 4000 with double precision floating point values this would require $4000 * 8 = 32000$ bytes of space in the cache. Although this space seems to be available in the cache, the vector might become to big for the bigger vectors. For this reason an alternative `mult` function was created to see if this would improve the program. In this function the vector is split into parts and the multiplication is done on part of the matrix, and so on for every section on the split vector. The results are added on top of the output vector, the implementation can be viewed in listing 2 Unfortunately, this implementation made the function run a lot slower, which is probably due to the fact that now the output vector needs to be read and written to a lot more times.

Listing 2: Modified `mult` function

```
void mult2(const int size, const double* __restrict m,
const double* __restrict v, double* __restrict d) {
```

```
    for (int z=0; z<size; z++)
        d[z] = 0.0;
    int split = 2;
    int s = size /split;
    for (int a=0; a<split; a++){
        for (int i=0; i<size; i++) {
            double temp = 0.0;
            for (int j=a*s; j<a*s+s; j++) {
                temp += m[i*size + j] * v[j];
            }
            d[i] += temp;
        }
    }
}
```

Next improvement to enable better use of resources was using OpenMP to parallelize the calculations. With best place to distribute to different threads for the `mult` function being the dot product for each row of the matrix. Making this part parallel instead of every product inside the inner loop means that there is no need to add a critical section which would make the threads have to wait for each other. The updated multiplication can be seen in the below code snippet.

Listing 3: `mult` using OpenMP

```
#pragma omp parallel for
for (int i=0; i<size; i++) {
    double temp = 0.0;
    for (int j=0; j<size; j++) {
        temp += m[i*size + j] * v[j];
    }
d[i] = temp;
}
```

The results of the carried out tests using this version of the function are displayed in the table below. The time was calculated only for the `mult` function. The wall time seems to be going down with the number of threads growing for up to 6 threads. This good and expected as the used CPU contains 6 cores, and higher number of threads are not able to work in parallel.

Since the most time consuming operation in program is matrix-vector multiplication which is a popular operation in many algorithms, there are highly optimised versions of this operation already implemented in external pack-

Table 1: Wall time for the `mult` function with different number of threads using OpenMP

Size	1 thread	2 threads	4 threads	6 threads	8 threads
1000	0.43s	0.21s	0.1s	0.08s	0.1s
2000	4.6s	1.95s	1.58s	1.37s	1.58s
4000	19.78s	10.76s	7.07s	6.02s	8.84s
5000	38.34s	22.06s	14.53s	15.52s	17.18s

ages. Therefore LAPACK's `dgemv()`[3] was also used and tested to see the performance difference by it and the `mult` function. The results obtained by this implementation can be seen in table 2.

Table 2: Wall times for best obtained for `mult` function and `dgmev`

Size	dgemv, O0	dgemv, O3	6 threads, O0	6 threads, O3
1000	0.01s	0.01s	0.08s	0.03s
2000	0.42s	0.46s	1.37s	0.42s
4000	2.44s	2.46s	6.02s	2.25s
5000	5.16s	5.16s	15.52s	4.74s

As seen from the table optimization flags don't seem to improve the execution time of the BLAS function. Moreover this means that the original `mult` function using 6 threads and an O3 optimisation flag is able to complete the computation slightly faster.

3.3 Rest of the Calculations

Since the rest of the program requires much less computation time compared to the matrix-vector, its optimisation was left to end however some methods still were tested.

The rest of the program which is the remaining part of the loop inside the `conjugate` function, the operational intensity was calculated to be approximately 1/8, which unfortunately means that this part is also memory bound.

OpenMP parallelization was tested since it could be a good application for several loops in this part of the program, as they don't have a critical section. Unfortunately, since the actual computation part is very little the change either didn't make a perceivable change or actually made it worse as it had to account for the setting up of the threads.

3.4 Result

At the end all the tested improvements were combined to achieve best possible performance. Which was basically using my-own matrix-vector multiplication function, with 6 threads using OpenMP and the `Ofast` optimisation flag. The wall time compared to the not optimised version can be seen below.

Table 3: Wall times for not optimised code and fully optimised and parallelised

Size	**No opt. or paral.**	**6 thread, Ofast**
1000	0.43s	0.02s
2000	4.79s	0.35s
4000	17.97s	1.87s
5000	38.1s	4.23s

4 Conclusion

It was surprising and enjoyable to see that my=own but multi-threaded version of the matrix-vector multiplication performed better than the BLAS function. I also wanted to test if SIMD wouldn't have improve the results but weren't able to implement a working vectorized `mult` function.

References

[1] Conjugate gradient method - wikipedia. https://en.wikipedia.org/wiki/Conjugate_gradient_method#The_resulting_algorithm. Accessed: 2021-05-26.

[2] Intel core i7-10710u processor product specifications. https://ark.intel.com/content/www/us/en/ark/products/196448/intel-core-i7-10710u-processor-12m-cache-up-to-4-70-ghz.html. Accessed: 2021-05-27.

[3] Lapack: dgemv. http://www.netlib.org/lapack/explore-html/d7/d15/group__double__blas_level2_gadd421a107a488d524859b4a64c1901a9.html#gadd421a107a488d524859b4a64c1901a9. Accessed: 2021-05-27.

[4] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.