Project: Conjugated Gradient Solver

ShiLin WANG shi-lin.wang@epfl.ch SCIPER: 294925
https://c4science.ch/diffusion/9010/

Abstract

As a powerful solver for searching minimum of quadratic function or equivalently for solving Ax=b, conjugated gradient is normally limited by the size of matrix. In this project, I aim to circumvent such problem by parallelism. First, I will discuss about the theoretical aspect. Then a parallel implementation by MPI and CUDA will be conducted and the results will also be shown.

1 Scientific Background

1.1 Introduction

Consider minimizing a quadratic function $f(x) = \frac{1}{2}x^T Ax - b^T x + c$, where A is a positive-definite matrix and b is a vector, plenty of solvers have been designed. One classic and well-known method is steepest descent solver, which searches minimum in the opposite direction of residual $r = -f'(x) = b - Ax$. With a proper step size $\gamma = \frac{r^T r}{r^T A r}$, an optimum performance can therefore be reached. However, steepest descent solver is not an efficient method because there are always some repeating searching directions which leads to a zigzag searching path. One might think of a way to search the minimum without going through the same searching direction again and again. In other words, a set of orthogonal (or A-orthogonal) searching directions $d = \{d^0, d^1, \ldots, d^n\}$ will be picked so that one takes exactly one step for each searching direction. This is the principle of conjugated gradient solver. A very common comparison between steepest descent solver and conjugated gradient solver is shown as Figure-1.

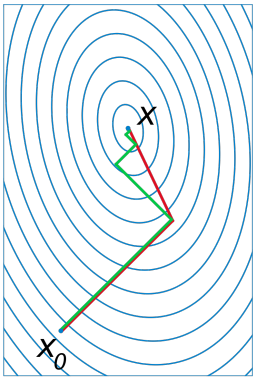

Figure 1: (Picture from Wikipedia) Searching paths for steepest descent solver (green line) and conjugate gradient solver (red line), where it starts at x_0 and stops at local minimum x. By applying orthogonal directional searching directions, conjugate gradient method is able to take least steps in searching while steepest descent method will have a zigzag searching path, which takes additional steps on the same searching direction.

1.2 Algorithm of Conjugated Gradient Solver

The algorithm of conjugated gradient method is as follow. First, the program begins with an arbitrary starting point x_0, where usually $x_0 = 0$. Then it calculates the first residual $r_0 = b - Ax_0$ and sets the initial searching direction as $p_0 = r_0 = b - Ax_0$. Based on these parameters, starting from $k = 0$, a loop is repeating until (1) the norm of residual r_{k+1} is less or equal to the defined tolerance ϵ or (2) until k equals to the defined steps n. During the loop, the searching direction p_{k+1} is updated to be the vector which is A-conjugated with the previous searching direction p_k. The searching step β is calculated based on r_k with similar manner as the optimum steepest descent searching step γ (as described in section-1.1). Below I summarize the algorithm in pseudo-code,

- x_0
- $p_0 = r_0 = b - Ax_0$
- For $k = 0, 1, 2, \ldots n$:
  1. $x_{k+1} = x_k + \alpha_k\ p_k$ where $\alpha_k = (r_k^T r_k)/(p_k^T A p_k)$
  2. $r_{k+1} = b - Ax_{k+1} = r_k - Ap_k$
  3. if $r_{k+1} \leq \epsilon$ then break
  4. $p_{k+1} = r_{k+1} + \beta_k p_k$ where $\beta_k = (r_{k+1}^T r_{k+1})/(r_k^T r_k)$
  5. $k = k + 1$
- End

1.3 Theoretical Analysis of Algorithm

1.3.1 Time Complexity

Based on section-1.2, the total number of floating point operations (Flops) can be calculated. First, before looping, there is one matrix-times-vector and one vector addition. Second, works involved in each iteration are one matrix-times-vector, two divisions and three inner products and three vector-plus-scalar-times-vector. The number of flops are listed in Table-1.

Operation	Representation	Flops
Matrix-times-vector	$c = A \times b$	n^2
Vector-plus-scalar-times-vector	$c = b + \alpha \times c$	$2n$
inner product	$\alpha = b \cdot c$	n
Division	$\alpha = \alpha/\beta$	1
Assuming A(n×n), b(n), c(n), α(scalar), β(scalar)		

Table 1: Summary of operation vs. approximate number of flops

The matrix size of A is n^2. Therefore, the worst case for iteration is n steps. Usually residual converges earlier before n steps. Given κ the number of total iterations, the whole program has the total number of flops to be,

$$Flops = (n + n^2) + \kappa\{(n^2) + 3(2n) + 3(n) + 2(1)\} \tag{1}$$

An asymptotic representation will therefore be as,

$$Flops = (\kappa + 1)(n^2) \tag{2}$$

which also shows the time complexity of such program,

$$\mathcal{O}(\kappa n^2) \tag{3}$$

Such time complexity is very general case since I take A as any arbitrary matrix. However, for a more careful analysis of quadratic function minimum problem, since A is positive definite and symmetric, research has shown that the complexity can be reduced to

$$\mathcal{O}(m\sqrt{\kappa}) \tag{4}$$

where m is the number of non-zero elements in matrix A and κ is normally depending on the dimension and boundary conditions of problem.

1.3.2 Space Complexity

After calculating the time complexity of program to be $\mathcal{O}(\kappa n^2)$, or $\mathcal{O}(\sqrt{\kappa}m)$. An approximation of total required memory can be calculated. There are two integers (n, k), two double-type scalar (α, β), eight double-type vectors with size n (A_p, b, x_k, x_{k+1}, r_k, r_{k+1}, p_k, p_{k+1}) and one double-type matrix with size n^2 (A). There are no further requirement throughout the computation. Consequently, the memory will be roughly $4(2) + 8(n^2 + 8n + 2)$ bytes.

Finally, it is important to estimate the space complexity. As one can see in last paragraph and section-1.2, throughout the program, the required memory space is dominated by the storage of matrix elements with size of n^2. Thus the space complexity will be,

$$\mathcal{O}(n^2) \tag{5}$$

1.3.3 Arithmetic Intensity

It is know that the arithmetic intensity (AI) is defined as

$$AI = \frac{W}{Q} \tag{6}$$

where W is the amount of work, which is expressed as Flops. Q is the amount of memory access for read and write. From section-1.3.1, it is shown that the most dominant step is where matrix-vector-multiplication happens. Therefore, to make the estimation of AI simpler, I zoom-in to one representative step as follow,

$$r_{k+1} = r_k - A \times p_k \tag{7}$$

As for W, the total number of Flops for this step can be calculated to be $2n^2$. On the other hand, Q can be seen that there are three readings from main memory to cache. They are two vectors (p_k and r_k with size n) and one matrix (A with size n^2). There is also one writing from cache to main memory, that is one vector (r_{k+1} with size n). Thus Q can be calculated as $n^2 + 3n$. From section-1.3.1, Finally, an estimate of AI will be,

$$AI \approx (2n^2)/(n^2 + 3n) \text{ flops/bytes} \tag{8}$$

2 Implementation

2.1 Roofline model

In order to have an in-depth estimate of performance, the roofline model is used in this section. Roofline model is an intuitive way to assess the performance of program running on certain processor. It consists of two parameters, performance P (Gflops/sec) and arithmetic intensity AI (flops/bytes). Thier relationship is defined by the peak performance of processor π and the peak band width β of the processor as follow,

$$P(AI) = min \begin{cases} \pi \\ AI \times \beta \end{cases} \tag{9}$$

If AI for a given program is bigger than the threshold AI (AI^*), where $AI^* \times \beta = \pi$. One can suspect that such program spends most of time on CPU for calculation. On the other hand, if AI is less than AI^*, the program is then spending most of the time transferring the data. With such understanding, one is able to optimize their program to boost the performance.

For the serial version, I run the calculation on the **debug** node of **EPFL** cluster **FIDIS** with single core. For each node, there are two processors of **Xeon E5-2690 v4 processors** each with 14 cores and integrated memory controller **DDR4-2400**. The peak band width is 71.53 Gbytes/sec. The peak performance is calculated as,

$$\begin{aligned} \text{Peak P} &\approx \text{Number of FP ports} \times \\ &\quad \text{Flops/cycle} \times \\ &\quad \text{Vector length} \times \\ &\quad \text{Frequency} \times \\ &\quad \text{Number of cores} \\ &= 2 \times 16 \times 32 \times 2.6 \times 1 \quad \text{Gflops/sec} \end{aligned}$$

The roofline plot is shown as Figure-2. Based on section-1.3.3, the AI of conjugated gradient solver is estimated. For the value of n, the example is taken from file **lap2D_5pt_n100.mtx**, which is matrix of size 10000×10000. Therefore, AI can be calculated as,

$$AI = (2 \times 10000^2)/(10000^2 + 3 \times 10000) \approx 1.9994 \approx 2 \quad \text{flops/bytes} \tag{10}$$

As one can notice, the obtained AI of conjugated gradient method locates at the far left of the roofline plot. This indicates the program is mainly bounded by the bandwidth. Therefore, a further study has to be done to reduce the time or the amount of transferring data so that the relative time for calculation and data transfer can be balanced.

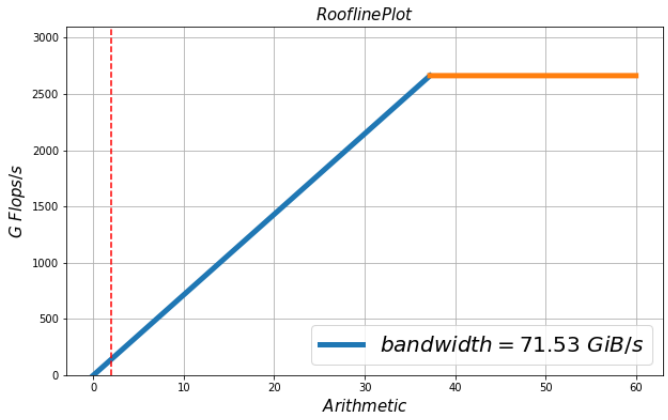

Figure 2: Roofline plot of single core, where red dashed line indicates AI of conjugated gradient solver.

2.2 Profiling

In order to increase the AI so that the calculation wont be bounded by bandwidth, one needs a way to assess the relative workload throughout steps. Thus, a profiling of serial version of conjugated gradient solver is done by performace analyzaing tool **perf**. The executable **perf** does the sequential profiling and the statistical analysis of the execution using hardware counters present in the CPU. Although the results may vary because it relies on statistical counting, such tool is still good enough to provide an overview of relative time spent on each step in the program.

The result shows that conjugated gradient solver spends approximately 99.75% of execution counts on function **cblas_dgemv**, which does the multiplication of a matrix by a vector and the vector addition with double precision. This is as my expectation because the time complexity is dominated by matrix-time-vector operation as shown in section-1.3.1. Consequently, to decrease the calculation time, one can either increase cache size or decrease the amount of transferring data during the calculation of **cblas_dgemv**.

2.3 Parallelization

Based on previous section, a way to decrease the amount of transferring data during the calculation of **cblas_dgemv** is by parallelism. By distributing independent calculation among nodes, less data is required and less data needs to be stored at main memory so that the program can be less bounded by bandwidth. In the following, the discussion is focusing on the speed-up and how to actually implement the parallelism into the code.

2.3.1 Speed-up

Given a fixed problem size, Amdahl's law is usually used to estimate the upperbound of speed-up S after parallelization. It has the following relationship,

$$S(p) \leq \frac{1}{f + (1-f)/p} \tag{11}$$

where p represents number of processors and f indicates the relative portion of non-parallelized part. The efficiency E is defined as,

$$E(p) = \frac{S(p)}{p} \tag{12}$$

In order to estimate the speed-up, the relative portion of non-parallelized part f in the code has to be analyzed first. Since I choose to parallelize **cblas_dgemv**, the parallelized portion is therefore proportional to the number of Flops in this step. Based on section-1.3.1, it is shown that the parallelized portion (1-f) is therefore,

$$(1-f) = \frac{\text{Flops}(\textbf{cublas\_Dgemv})}{\text{total Flops}} \tag{13}$$

Therefore, consider n=10000 for example file **lap2D_5pt_n100.mtx**, f and (1-f) are calculated as below,

$$\begin{aligned}(1-f) \approx \frac{n^2+2n}{n^2+9n+2} &= \frac{10000^2+2\times 10000}{10000^2+9\times 10000+2} \approx 99.93\% \\ f &= 0.07\%\end{aligned}$$

It can be seen that although the estimation is 99.93%, the measurement is actually 99.75% for **cblas_dgemv**. It may be the reason that theoretical result doesn't include the time for reading matrix and the time for printing output file. Nevertheless, the speed-up can be calculated as,

$$S(p) = \frac{1}{0.0007+0.9993/p}$$

Issue for Amdahl's law usually arises when the sequential part has lower complexity than parallelized part does. It is because Amdahl's law assumes a fixed problem size. When the problem size goes too high, the sequential part will become insignificant if it has a much lower complexity. To take the problem size into consideration, here I use Gustafson's law for estimation. Its relation is shown as follow,

$$S(p) \leq \frac{t_s}{t_p} = \frac{s+a\times n}{s+a\times \frac{n}{p}} \tag{14}$$

where t_s, t_p is time for sequential and parallelized program, s and a are portions for sequential and parallelized parts, and p is the number of processors. Based on the relative portions obtained previously, the speed-up can be calculated as,

$$S(p) = \frac{0.0007+0.9993\times 10000}{0.0007+0.9993\times \frac{10000}{p}} \tag{15}$$

Result is plot as Figure-3. It can be seen that Amdahl's speed-up converges when p approaches 1500 while Gustafson's speed-up keeps linear. Actually, the speed-up should be lower since I don't include the matrix reading and output into serial portion. Moreover, it should also include the latency for inter communication between processors after each iteration.

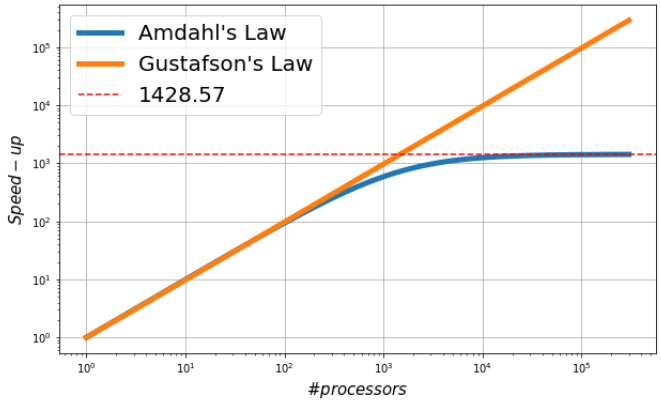

Figure 3: Speed-up vs. number processor

In section-2.3.1, Although Figure-3 is ideal case, it still shows that the program can be substantially speed-up by increasing the number of processors. As such, next is to think about how to parallelize **cblas_dgemv**. In the following section, I will illustrate my parallel strategy for MPI and CUDA.

2.3.2 Parallel strategy for MPI

Consider a matrix-vector multiplication $y = A \times p$ as below,

$$y = A \times p = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} \tag{16}$$

Equivalently, one can respectively tear A and p apart into three parts in a columnar fashion, do the multiplication and sum them,

$$\sum_{i=1}^{3} A_i \times p_i = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} \times \begin{bmatrix} p_1 \end{bmatrix} + \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix} \times \begin{bmatrix} p_2 \end{bmatrix} + \begin{bmatrix} a_{13} \\ a_{23} \\ a_{33} \end{bmatrix} \times \begin{bmatrix} p_3 \end{bmatrix} \tag{17}$$

One can therefore distribute these works to three processors instead of just using one processor doing all the work. With this simple illustration, such concept can be generalized to an arbitrary size of column-major matrix-vector multiplication problem with size (n^2 and n). Given m processors where n/m is an integers, the general concept is as follow. First, program divides both matrix and vector into m pieces. Each processor takes each piece and do the matrix-vector-multiplication. Finally, resulting vectors are summed up. A graphical representation is shown as Figure-4. The mathematical representation can be expressed as below where m is the total number of processors.

$$y = \sum_{i=1}^{m} A_i \times p_i = \sum_{i=1}^{m} \begin{bmatrix} a_{1\ (i)(n/m)} \cdots a_{1\ (i+1)(n/m)} \\ \vdots \\ a_{n\ (i)(n/m)} \cdots a_{n\ (i+1)(n/m)} \end{bmatrix} \times \begin{bmatrix} p_{1\ (i)(n/m)} \\ \vdots \\ p_{1\ (i+1)(n/m)} \end{bmatrix} \tag{18}$$

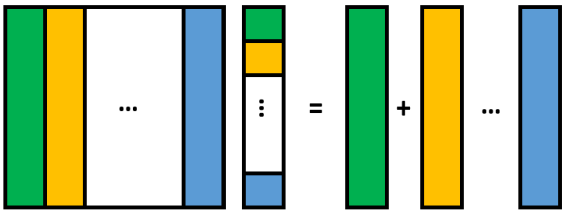

Figure 4: Graphical representation of parallel strategy for MPI. Both matrix and vector are divided into m smaller pieces. Each processor takes pieces of smaller matrix and vector then do the matrix-vector multiplication. Finally, the resulting vectors are summed up. The color is to indicate pieces taken by different processors.

2.3.3 Parallel strategy for CUDA

In section-2.3.2, the maximum processors to be used is limited to the width of matrix (n). For CUDA implementation, the parallelization is massive due to a large number of processors of GPU (threads). For example, a few thousands of threads or even more are easy to be launched for parallelization. Therefore, one can aggressively distribute works to more threads. It should be noted that **NVIDIA** has provided an parallelized function for **cblas_dgemv** called **cublas_Dgemv**. However, the number of threads is already optimized for given matrix and vector size. Therefore, I will use it as benchmark. In the following, I propose a naive parallel implementation which does the same as **cblas_dgemv**.

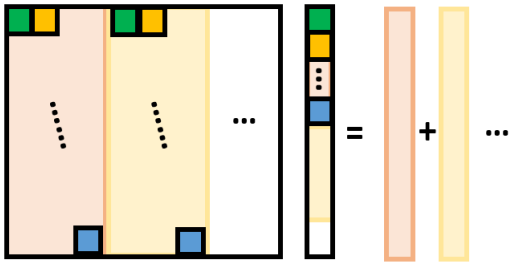

Figure 5: Graphical representation of parallel strategy for CUDA. The light orange region shows the first step where m threads (shown as dark green, yellow to blue) take its own elemental multiplication. Light yellow region shows the second step. Resulting vectors from each step are then summed-up.

Since one is able to launch a large number of threads using CUDA, instead of distributing matrix and vector in a columnar fashion like what is done for MPI, I simply call one thread for each elemental multiplication. For example, considering equation-16, instead of splitting works into three parts, one can distribute all the calculation to 9 threads. Thus, it can be generalized to an arbitrary size of column-major matrix-vector multiplication problem with size of (n^2 and n). Figure-5 illustrates how it works. Given m threads, m elemental multiplication will first be done by each thread. Results are then summed-up according to the index. Finally, threads will move on or stop until all the calculation is done.

3 Result

Based on strategies from section-2.3.2, MPI-based and CUDA-based version of conjugated gradient methods are implemented. The codes are provided in the URL shown at the beginning of this report. Series of tests for strong scaling and weak scaling will be conducted for both implementation and shown in the following.

3.1 Implementation of MPI

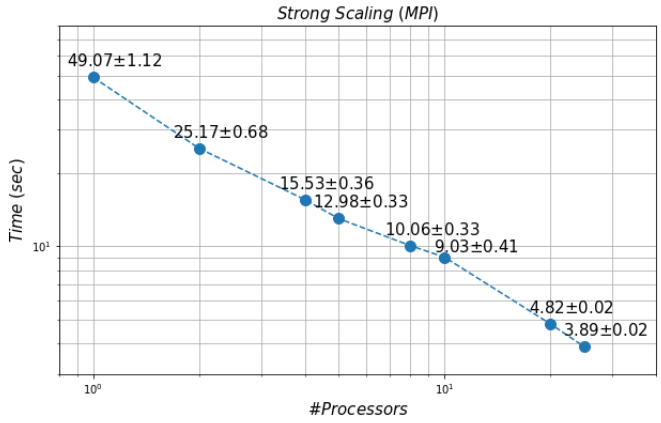

Figure 6: Strong scaling result of MPI

Strong scaling is to assess the calculation time given different number of processors at a fixed problem size. The problem size is $n = 10000$ and the number of processors are $p = \{1, 2, 4, 5, 8, 10, 20, 25\}$ respectively. For each number of processor, 50 independent samples are done. The standard error is already shown as error bars while it is too small to be viewed. As can be told from logarithmic diagram of Figure-6, the calculation time scales linearly with the number of processors. It is because conjugated gradient method is bounded by bandwidth. When works are distributed by more and more processors, less and less data will be transferred from main memory to the cache.

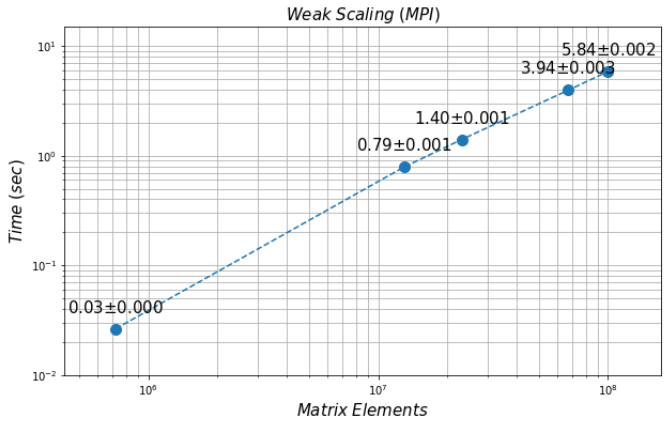

Figure 7: Weak scaling result of MPI

Weak scaling is to assess the calculation time given different problem size at a fixed number of processors. The number of processors is $p = 16$ and the problem sizes are $n = \{848, 3600, 4800, 8192, 10000\}$ respectively. Similarly, for each number of processor, 50 independent samples are done. The standard error is also shown as error bars while it is too small to be viewed. From Figure-7, the calculation time scales linearly with the problem size. It results from a bigger matrix and so a bigger amount of data to be transferred from main memory.

3.2 Implementation of CUDA

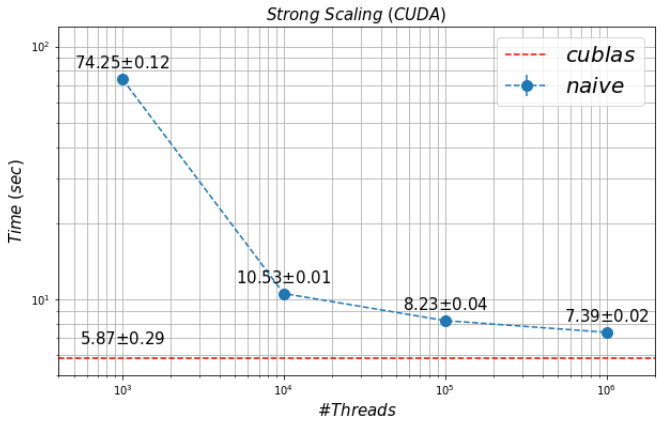

Figure 8: Strong scaling result of CUDA

For the strong scaling of the implementation of CUDA, I choose $n = 10000$ and the number of threads $p = \{10^3, 10^4, 10^5, 10^6\}$ respectively. As mentioned in section-2.3.3, **NVIDIA** has provided an optimized function called **cublas_Dgemv** whose threads cannot be freely tuned. Therefore, I use it as benchmark to compare with my naive implementation. Result is shown as Figure-8. It can be noticed that calculation time decreases as the number of threads increases. However, they don't scale linearly. It is because that the cache size of GPU is smaller than the cache size of CPU. When the number of threads is not high enough, most data has to be fetched from shared memory and so the whole procedure is slowed down. Thus one can also find the calculation time for $p = 10^3$ is even longer than serial version. Finally, when number of threads gradually approach the number of matrix elements, my naive implementation converges to the **cublas_Dgemv**.

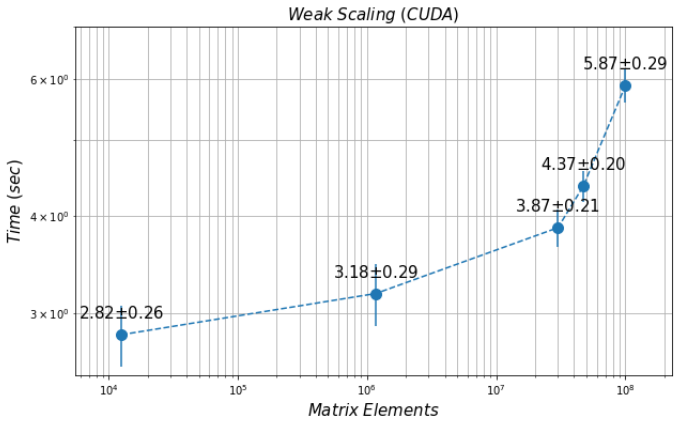

Figure 9: Weak scaling result of CUDA

For the strong scaling of the implementation of CUDA, I use **cublas_Dgemv** and the problem size $n = \{848, 3600, 4800, 8192, 10000\}$ respectively. It can be seen that the calculation time increases as the problem size increases. Similarly, it results from that the program has an increasing amount of data to be transferred while only has a small cache.

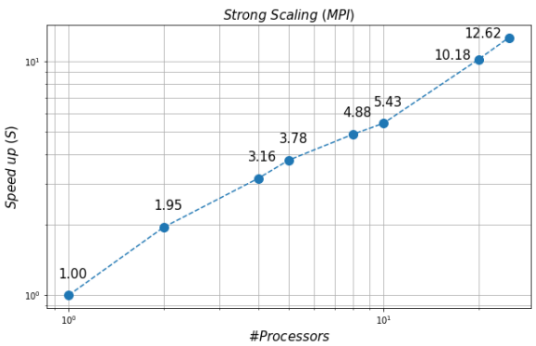

Figure 10: Speed-up of MPI

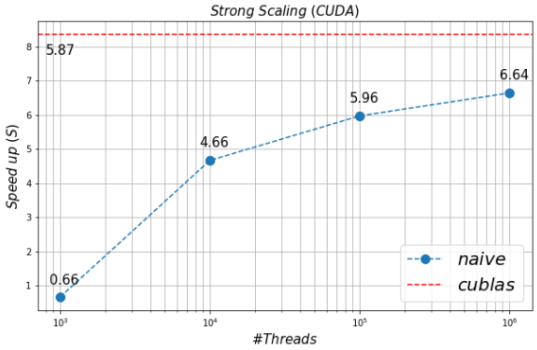

Figure 11: Speed-up of CUDA

Finally, the speed-up for implementation of MPI and CUDA are presented as Figure-10 and 11. Compared with ideal prediction from section-2.3.1, the speed-up is way more lower. Again, since it doesn't include the reading of matrix and output into speed-up calculation. Also, in reality, the cache size is limited and there is always latency no matter how close it is between the memory and processor.

3.3 Resource budget proposal

In this section, a resource budget for a larger scale conjugated gradient project will be proposed. For the simplicity, I select MPI for proposal because it will be easier for extrapolation. Based on section-2.3.2 and 2.3.3, it can be seen that the implementation of MPI scales linearly for both strong scaling and weal scaling. Their fitting functions are as below,

$$\text{Strong scaling (n=10000):} \quad log_{10}\, t = -0.753\, log_{10}\, p + 1.662 \tag{19}$$

$$\text{Weak scaling (p=16):} \quad log_{10}\, t = 1.097\, log_{10}\, n^2 - 7.966 \tag{20}$$

Here I choose an arbitrary goal as finishing the conjugated gradient method program for a larger matrix, say $n = 10^5$, within 5 minutes. The total number is 100 independent samples. Starting from equation-20, I can calculate the number of processors p required as

$$t = 10^{\left(1.097\ log_{10}\ 10^{10} - 7.966\right)} \approx 1003.456\text{sec} \tag{21}$$

Although it may not be the case, assuming the relation of speed-up vs. log of the number of processors still holds for $n = 10^5$, I can estimate the minimum p,

$$log_{10}\left(\frac{t_1}{t_2}\right) = log_{10}\left(\frac{p_1}{p_2}\right)^{-0.753} \tag{22}$$

$$log_{10}\left(\frac{360}{1003.456}\right) = log_{10}\left(\frac{p}{16}\right)^{-0.753} \tag{23}$$

$$p \approx 80 \tag{24}$$

The memory required during the calculation can also be obtained from section-1.3.2 as,

$$4(2) + 8(10^{10} + 8 \times 10^6 + 2) \approx 80 \text{ GBytes} \tag{25}$$

Assuming the output file contains the time and the position of minimum point. Each number is expressed in 10 digit characters. The disk space required would roughly be,

$$10 \times (10^5 + 1) + 1 \times (10^5) \approx 1.2 \text{ MBytes} \tag{26}$$

Finally, the budget is listed as below,

Total number of requested cores	80
Minimum total memory	100 GBytes
Maximum total memory	100 GBytes
Temporary disk space for a single run	1.2 MBytes
Permanent disk space for the entire project	120 MBytes
Library requirements	gcc, openblas, mvapich2
Code publicly available	YES
Architectures where code ran	Xeon E5 v4 processor