Solving Poisson Equation Using Conjugate Gradient Iteration.

Table of Contents

Abstract

Poisson’s equation is a common partial differential equation that can be applied in many engineering cases. However, due to the large matrix dimension, performance is extremely limited on single node computation. Supercomputers can high-performance computing are introduced to solve the performance issue. Techniques like OpenMP and MPI can improve the performance to a large extent. But in this case, most computation is matrix or vector operations, which both are Single Instruction Multiple (SIMD) Data operations. We can see GPU computation is especially suitable for operations like these.

In this project, I implemented a serial and CUDA program to discretize Poisson’s Equation and solve problem using conjugate gradient iteration method. For storage efficiency, I adopt two different storage schemes (coordinate format and compressed sparse row) to store the sparse matrix.

I conduct extensive evaluations on both serial and parallel program to compare the performance in terms of CPU time and elapsed time. Besides, I compare the storage efficiency for the two schemes I use.

Background

Poisson’s Equation

In mathematics, Poisson's equation is a partial differential equation of elliptic type with broad utility in mechanical engineering and theoretical physics. It is a generalization of Laplace's equation, which is also frequently seen in physics. It arises, for instance, to describe the potential field caused by a given charge or mass density distribution; with the potential field known, one can then calculate gravitational or electrostatic field. The equation is named after the French mathematician, geometer, and physicist Siméon Denis Poisson.[1]

9-point finite difference stencil

In this project, I use 9-point finite difference stencil to discretize the equation. The 9-point grid stencil can be viewed as in Figure 1. Given a square grid in one or two dimensions, the 9-point stencil of a point in the grid is a stencil made up of the point itself together with its eight "neighbors". This method is mainly used to write finite difference approximations to derivatives at grid points.

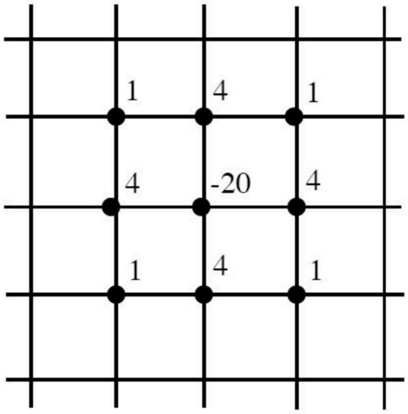


Figure 1. 9-point grid stencil[2]

Conjugate Gradient Iteration

After we discretize the equation using 9-point finite difference stencil, we need to approximate the solution x in the following linear equations.

$$Ax = b$$

A and b is known and A is symmetric and positive definite. We start our iteration by setting a first guess X_0, we can always guess $x0 \ = \ 0$ if we have no reason to guess for anything else. Let R_k denotes the residual at the kth step:

$$r_k = b - AX_k$$

The iteration will continue until it reaches the maximal iteration times or the 2-norm of the residual r, which equals to $\sqrt{r^T r}$ is reduced below some specified stopping tolerance. In this project, the maximal iteration times is 1500 and the stopping tolerance is $10^{-6}\sqrt{b^T b}$.

[1] http://www.wikiwand.com/en/Poisson%27s_equation

[2] http://brunoc69.xtreemhost.com/courses/WS05-06/numerikII/stencil.pdf?i=1

CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.[3] It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). GPU are much more powerful than CPU in dealing with SIMD operations. In our project, the largest matrix dimension can be up to $4{,}194{,}304 \times 4{,}194{,}304$. With thousands of cores in GPU, we can boost the performance significantly.

Sparse Matrix Storage Schemes

Though the dimension of matrix in this project can be up to $4{,}194{,}304$. But most of the elements inside the matrix is zero. We usually call this kind of matrix sparse matrix. There are a lot of ways to compressed the storage to avoid storing zero elements. The two schemes I use in this project is called coordinate format (CF) and compressed sparse row (CSR)[4]. Below Nz represents the number of non-zero elements.

Coordinate Format

The data structure consists of three arrays:

(1) a real array containing all the real values of the nonzero elements of A in any order;
(2) an integer array containing their row indices; and
(3) a second integer array containing their column indices. All three arrays are of length Nz, the number of nonzero elements.

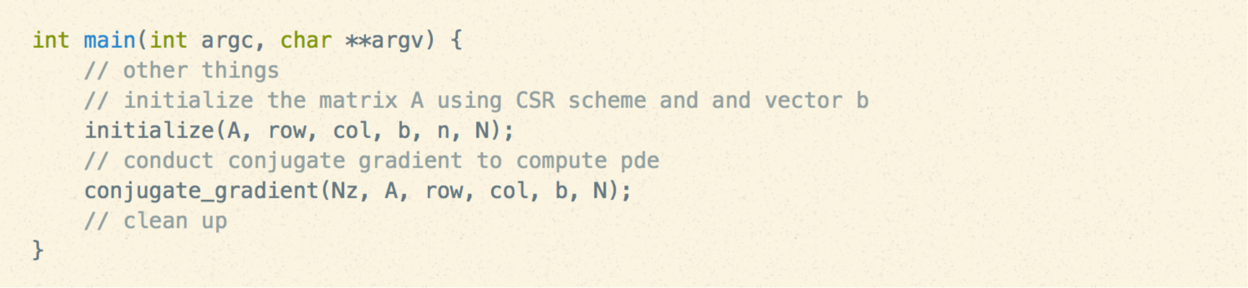
Compressed Sparse Row

(1) A real array AA contains the real values a_{ij} stored row by row, from row 1 to n. The length of AA is Nz.
(2) An integer array JA contains the column indices of the elements a_{ij} as stored in the array AA. The length of JA is Nz.
(3) An integer array IA contains the pointers to the beginning of each row in the arrays AA and JA.

[3] http://www.wikiwand.com/en/CUDA
[4] http://www-users.cs.umn.edu/~saad/toc.pdf

Workload

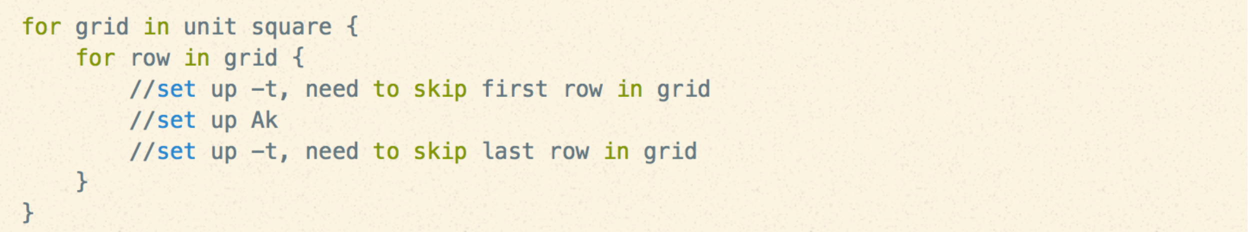
I implemented the serial and CUDA program to solve the problem I stated above. The program logic for both serial and parallel program is quite similar. The main function looks like the following pseudo code.



```
int main(int argc, char **argv) {
    // other things
    // initialize the matrix A using CSR scheme and and vector b
    initialize(A, row, col, b, n, N);
    // conduct conjugate gradient to compute pde
    conjugate_gradient(Nz, A, row, col, b, N);
    // clean up
}
```

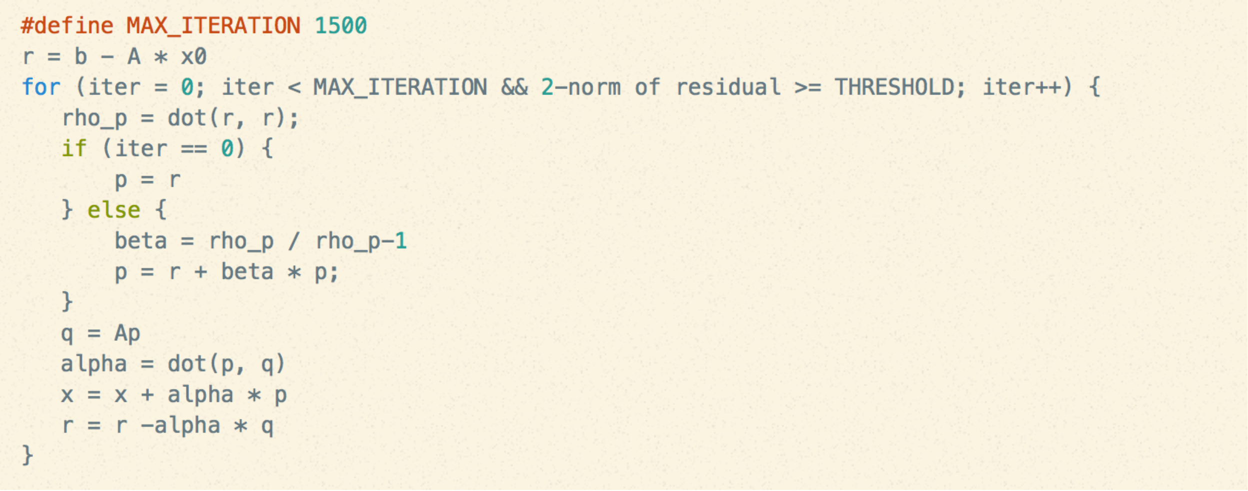
The whole program can be split into two stages. The first stage is to prepare the data for example matrix A and vector b. The second stage is to conduct conjugate gradient iteration.

The first is stage seems easy but turns out to be the error prone. For each grid in the square unit, and for each row inside the grid, we set up $-t, A_k, -t$. But we need to handle the edge case where we are at the first row of the grid. Below is the pseudo code.



```
for grid in unit square {
    for row in grid {
        //set up -t, need to skip first row in grid
        //set up Ak
        //set up -t, need to skip last row in grid
    }
}
```

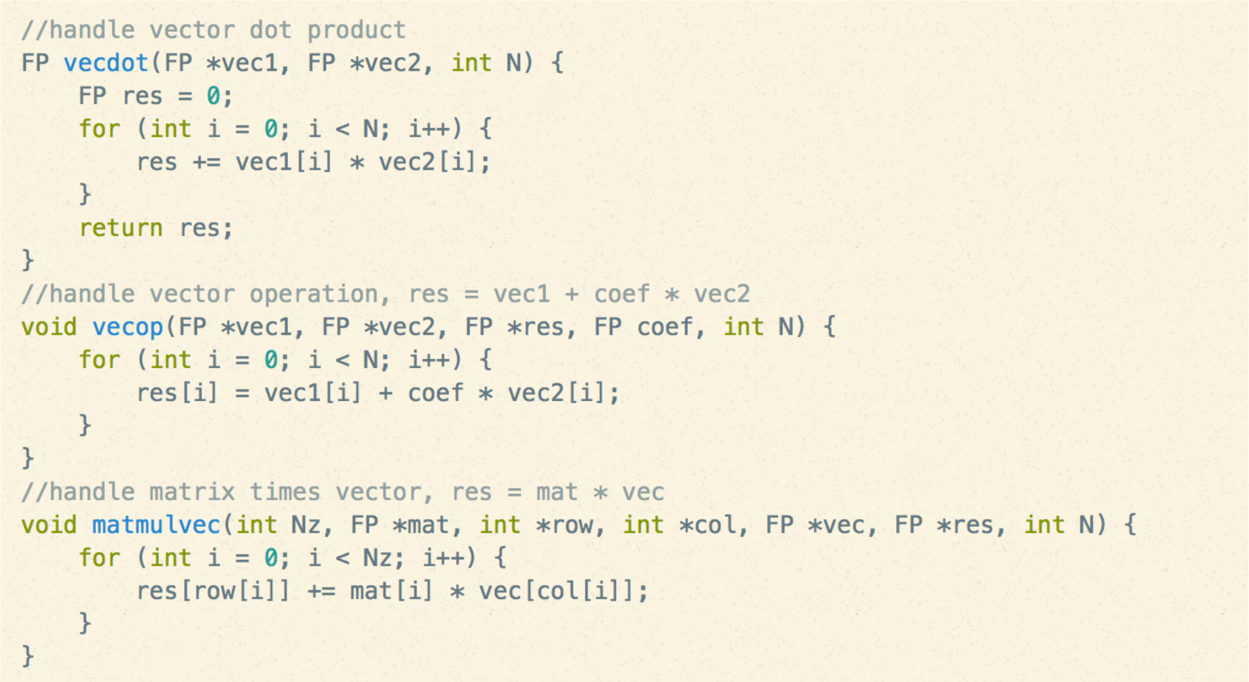
The pseudo code of conjugate gradient iteration is as follow. We need to solve $Ax = b$, where A and b are initialized in the previous stage. We start with a guess for x, call it x_0, we continue our iteration until $\sqrt{r^T r} < 10^{-6}\sqrt{b^T b}$ or iteration times reaches maximal iteration times.



```
#define MAX_ITERATION 1500
r = b - A * x0
for (iter = 0; iter < MAX_ITERATION && 2-norm of residual >= THRESHOLD; iter++) {
   rho_p = dot(r, r);
   if (iter == 0) {
       p = r
   } else {
       beta = rho_p / rho_p-1
       p = r + beta * p;
   }
   q = Ap
   alpha = dot(p, q)
   x = x + alpha * p
   r = r -alpha * q
}
```

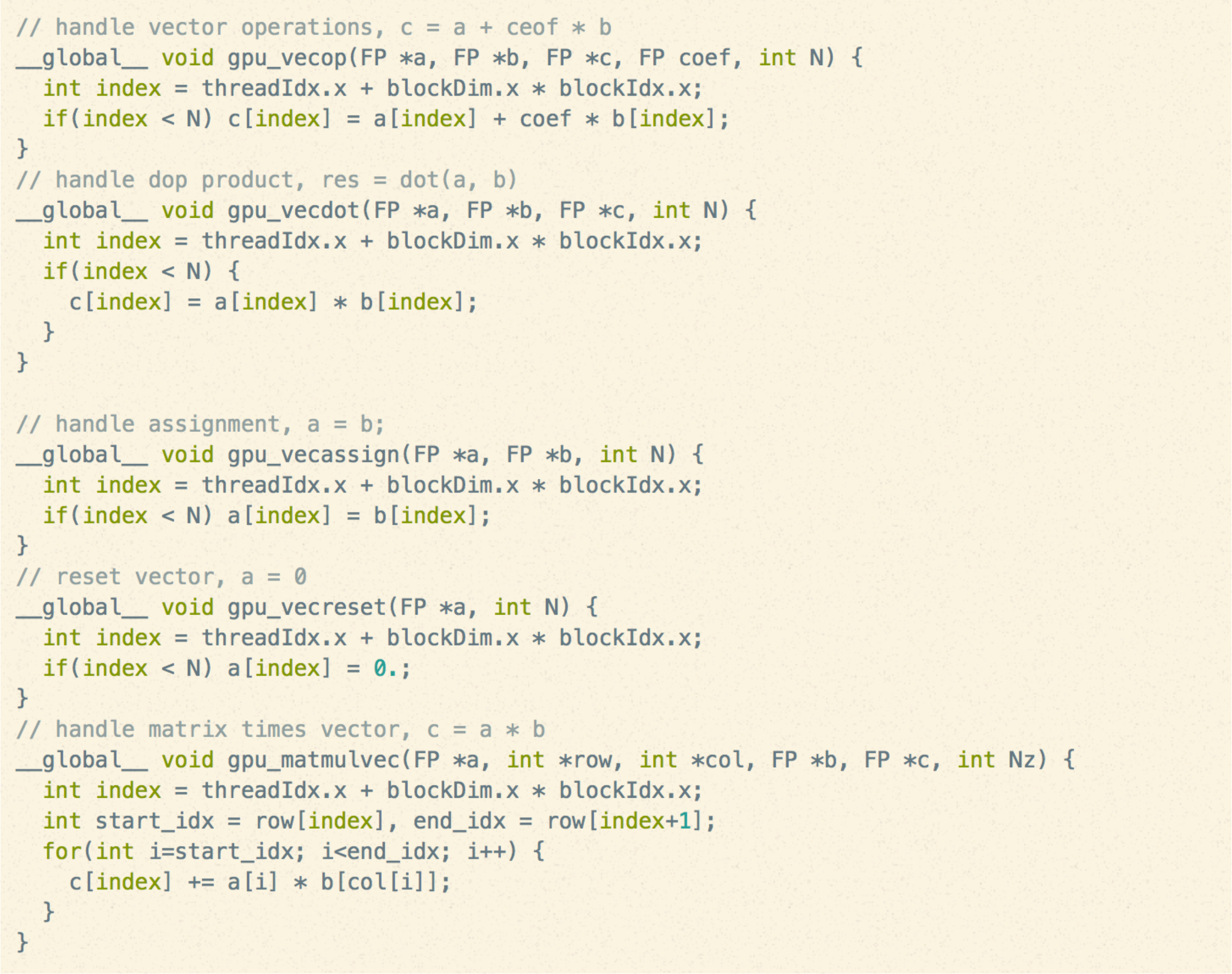
Serial Program Utility Functions

I need to provide utility functions like matrix multiplication and vector operations like add and minus. Below is the code to support these functions.



```
//handle vector dot product
FP vecdot(FP *vec1, FP *vec2, int N) {
    FP res = 0;
    for (int i = 0; i < N; i++) {
        res += vec1[i] * vec2[i];
    }
    return res;
}
//handle vector operation, res = vec1 + coef * vec2
void vecop(FP *vec1, FP *vec2, FP *res, FP coef, int N) {
    for (int i = 0; i < N; i++) {
        res[i] = vec1[i] + coef * vec2[i];
    }
}
//handle matrix times vector, res = mat * vec
void matmulvec(int Nz, FP *mat, int *row, int *col, FP *vec, FP *res, int N) {
    for (int i = 0; i < Nz; i++) {
        res[row[i]] += mat[i] * vec[col[i]];
    }
}
```

CUDA Utility Functions



```
// handle vector operations, c = a + ceof * b
__global__ void gpu_vecop(FP *a, FP *b, FP *c, FP coef, int N) {
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  if(index < N) c[index] = a[index] + coef * b[index];
}
// handle dop product, res = dot(a, b)
__global__ void gpu_vecdot(FP *a, FP *b, FP *c, int N) {
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  if(index < N) {
    c[index] = a[index] * b[index];
  }
}

// handle assignment, a = b;
__global__ void gpu_vecassign(FP *a, FP *b, int N) {
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  if(index < N) a[index] = b[index];
}
// reset vector, a = 0
__global__ void gpu_vecreset(FP *a, int N) {
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  if(index < N) a[index] = 0.;
}
// handle matrix times vector, c = a * b
__global__ void gpu_matmulvec(FP *a, int *row, int *col, FP *b, FP *c, int Nz) {
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  int start_idx = row[index], end_idx = row[index+1];
  for(int i=start_idx; i<end_idx; i++) {
    c[index] += a[i] * b[col[i]];
  }
}
```

Evaluation

In this section I will evaluate the performance of both serial and CUDA programs in terms of CPU time and elapsed time. Also I will evaluate the two storage schemes that I use in terms of storage efficiency.

Performance

Table 1 below shows the number of CG iterations, the 2-norms of the initial and final residuals, and the total elapsed time and CPU time spent in the CG function (in seconds) for each case for serial program. Table 2 shows the corresponding values for CUDA program.

	#CG iterations	2-norms initial residual	2-norms final residual	Elapsed time	CPU time
128	1500	7.211105e-04	3.088311e-05	0.507	0.499
512	1500	4.559808e-05	4.837400e-06	10.779	10.763
1024	1500	1.142177e-05	1.889278e-06	46.997	46.973
2048	1500	2.858231e-06	7.299413e-07	188.581	188.507

Table 1. Performance for serial program

	#CG iterations	2-norms initial residual	2-norms final residual	Elapsed time	GPU time
128	1500	7.211105e-04	3.088311e-05	0.462	0.348
512	1500	4.559808e-05	4.837401e-06	5.450	5.331
1024	1500	1.142177e-05	1.889277e-06	18.688	18.534
2048	1500	2.858231e-06	7.299419e-07	70.368	70.105

Table 2. Performance for CUDA program

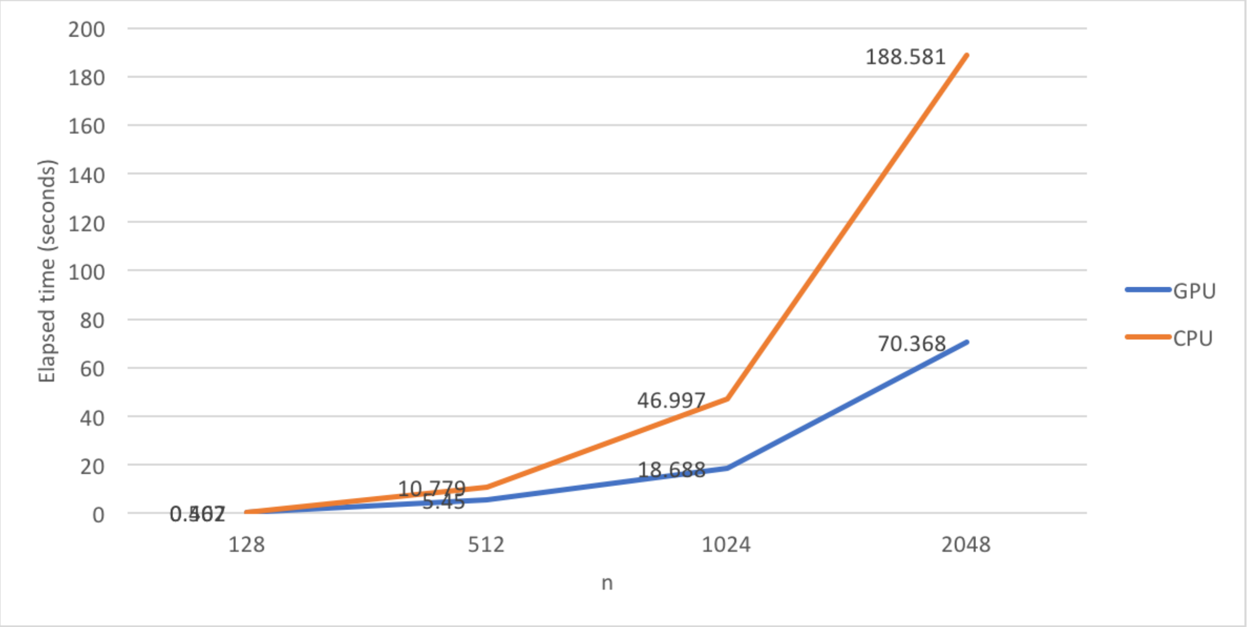


Figure 2. Elapsed time VS n

Figure 2 illustrates the relationship between elapses time and n. As the dimension grows, the elapsed time increases rapidly. But it is clear that GPU only uses less than 50% time than CPU does.

Storage Efficiency

Here we can define storage efficiency as the ratio of total bytes stored (including extra indexing data) divided by number of non-zeroes in A. The table shows the storage efficiency for two schemes in each case. It's not a surprise that CSR are more efficient than CF. Even though, CF is much better than storing the whole sparse array.

Scheme\n	128	512	1024	2048
CF	12.000000	12.000000	12.000000	12.000000
CSR	8.449138	8.445606	8.445024	8.444734

Table 3. Storage efficiency

Bibliography

Iterative Methods for Sparse Linear Systems (2nd Edition ed.). (2004). PWS.

Nvidia. (2016). *Cuda C Best Practices Guide.*

Nvidia. (2016). *CUDA Programming Guide.*