Spring 2019 ME759 Final Project Report
University of Wisconsin-Madison

Numerical Computation of the Jacobian

Yue Sun
Yang Liu

May 7, 2019

Abstract

In this project, we targeted to increase the performance of Jacobian operations by utilizing several different methods, including GPU and OpenMp. We also constructed an CPU naïve implementation of this algorithm so that the result can be verified against. The result shows that the OpenMp is the best among all since it does not require much data copy and reduces memory access overhead. CUDA has shown to have the worst performance. We believe the reason lies in the heavy memory allocation and data transfer overhead inside the kernel in which we called malloc() and memcpy() for each thread to allocate temporary space in kernel memory and copy data from either global memory or shared memory. It is possible that the result can be improved if the evaluated function is computing intensive, for which it can hide the latency of memory access. However, this needs further evaluation.

The Git repo address is: https://git.cae.wisc.edu/git/me759-ysun276

Contents

1. General information

- Home Department:
  - Computer Science Department at UW-Madison
- Current status:
  - Undergraduate Student
- Group Members:
  - Yue Sun (Team Leader)
  - Yang Liu
- Advisor Information:
  - No Advisor

2. Problem statement

- General Statement:
  - The problem we were trying to solve is ***Default Project 1: Numerical Computation of the Jacobian***, in which we calculated the gradient or the sensitivity of a scalar function *f* with respect to its m variables x_1 through x_m and n sets of such variables are given (such as (5, 3, 2, …, 8), m numbers in total).

- Motivation/Rationale:
  - As stated in the default project, Jacobian information comes into play in a variety of algorithm from optimization to the solution of nonlinear systems, which is an interesting and meaningful topic for us to explore under the circumstance that we do not have valuable and concrete thoughts on the project idea. We realized that Jacobian computation does not always rely on the analytical computation, but it must resort to approximating the Jacobian via numerical differentiation in some cases, which motivates us to take a further step.

3. Solution description

- Parallel Computing Interface Used:
  - CUDA and OpenMP
- Programming Language Used:
  - C Programming Language
- Host Side Implementation:
  - Data Structure of Input Data:
    - Considering the efficiency and performance issue of CUDA, we decided to use a dynamically allocated (malloc) 1-dimentional array to store our input data (which is a N * M matrix).
    - The reason why we do not use a 2-dimentional array is that first the allocation of a 2D array of N rows requires N + 1 cudaMalloc() call which is pretty slow and inefficient. Second, our solution also needed dynamically allocate memory in the kernel function. It would ruin the

performance of CUDA computing if we allocate 2D array in every thread.

- File I/O method Used:
  - We used fscanf() to write in the input data from a file and fprintf() to output the result to a file.
- OpenMP Implementation:
  - Used "#pragma parallel for collapse ()" to parallelize the work in the loop.

- Kernel Side Implementation:
  - General Strategy of Parallel Computing:
    - Job of each thread:
      - As we have N sets of M variables and we need to calculate every Jacobian of each variable of each set (in total, we need to calculate N * M Jacobian), we decide to calculate one Jacobian each thread.
    - Calculate each Jacobian:
      - Because one Jacobian is calculated by its derivative of the set of variables it belongs to (calculated by its Taylor series expansion), we need the data of the array (M variables) to calculate. Also, as the function *f* is not fixed, we cannot accurately manipulate the change (+/- h) of a particular variable in the process of Taylor computation in the f_eval.cuh (CUDA header function that store the function information and do the Taylor computation). Therefore, we can only make change (+/- h) to the particular variable that we want to calculate Jacobian in the set of data and pass the whole array to the f_eval.cuh function to do computation.
    - A necessary memory copy of each thread:
      - As we need to change a variable in the set of data and the remaining data should not be change in this variable's Jacobian calculation, we were facing the dilemma that different threads calculating different Jacobians of variables in the same set would influence each other. For example, it would be problematic if thread A changed the value of a variable in the set of data to calculate the Jacobian, but thread B used the changed data before thread A changed it back. Therefore, instead of making the threads sequential, we decided to make a temparay copy of the current set of for each thread and used the copied data to do computation. We freed the space which assigned to store the copied data after the computation for every thread.
  - Shared Memory Implementation Strategy and Block Size Assignment:
    - For functions whose number of variables (M) are more than 1024:
      - We did not use the shared memory in our implementation because it would be problematic if one block cannot calculate all variables in a set of data.

- Therefore, we used (N * M + 1023) / 1024 blocks to ensure all the data would be calculated. We used 1024 thread per block to maximize the efficiency of GPU computing. For each thread, temporary data copy was copied from the whole data passed by the kernel function using the thread index to identify which set we needed.

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
memcpy(temp_array, d_data + (index / M * M), M * sizeof(double));
```

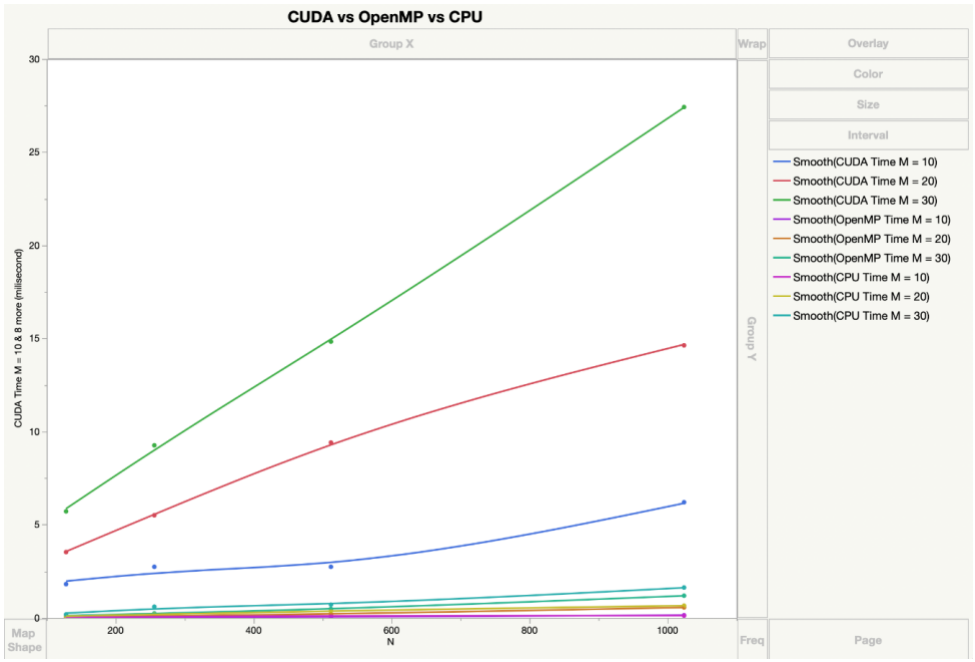
- For function whose number of variables (M) are less or equal than 1024:
  - We used the shared memory in this scenario because we can ensure that a block can handle at least one set of data. Therefore, we can store the sets of data that we are going to use in the shared memory. Then we copied the temporary data directly from the shared memory using the thread index to identify which set we needed.

```
int which_m = threadIdx.x / M;
int position = threadIdx.x - which_m * M;
memcpy(temp_array, shared_data + which_m*M, M*sizeof(double));
```

  - For the assignment of block size and the number of blocks we are going to use, we need to ensure that we do not split the Jacobian calculation of a set of data into two or more blocks. Therefore, we used (1024 / M) * M threads in one block to ensure that one block can calculate whole sets of data. We used (M * N + block size) / block size to make sure all data are calculated.

4. Overview of results. Demonstration of your project

For this project we have implemented the CUDA version, OpenMP version, and a pure CPU version which serves as the golden benchmark. Since we are uncertain with types of algorithm that will be used to test the result, we have implemented a sample evaluation function with M (number of variables in the function) value of 10, 20 and 30, with N (number of input points) value ranges from 128 to 1024. The result is shown below in the form of graph.



It can be seen that the OpenMP implementation significantly increase the performance compare to the CPU. However, CUDA version is not that simple, since some of the value is not quite positive as we would expect. We believe the result lies in the high overhead in the memory copy operations both from the host side to GPU side, and also the global memory in GPU to shared memory of each block. Specifically, we called malloc() and memcpy() in each thread to allocate temporary space in kernel memory and copy data from either global memory or shared memory. We believe the heavy overhead in this process badly influenced the performance of our CUDA solution.

One other issue we would like to point out is that the CUDA implementation has a limitation that M can not go over 39 when N = 1024. We believe this is a memory issue, such that at this point there is a shortage in memory. The possible reason behind is that we used malloc() to allocate some memory space in kernel for each thread. As the number of threads which are running concurrently increased, the kernel memory would be filled up before the free() instruction executed. However, we could not exactly pinpoint where is the problem as we think the general logic should be correct.

Other then problems mentioned above, the result have shown positive for all implementations. The result, compared to CPU golden mark, are all the same with same input data. Thus, the implementation is accurate base on our tests result.

5. Deliverables:

- This report will be in Canvas.
- File Description:
  - jacobian.cu:
    - source file of the GPU CUDA parallel computing of Jacobian

- jacobian_cpu_op.c:
  - source file of the CPU computing of Jacobian using OpenMP
- jacobian_cpu.c:
  - source file of the CPU computing of Jacobian for verification purpose
- matrix.c:
  - source file of generating the input file "inputArray.inp" (modify the file to change the value of M and N)
- generate.sh:
  - used the batch file to run the executable file and generate the input file "inputArray.inp"
- inputArray.inp:
  - input file that stores the data information. The first line is the value of N; The second line is the value of M; The following N lines are values. Each line contains M values.
- f_eval.cuh:
  - CUDA header function that store the information of the function *f* and does partial calculation of the Jacobian.
- f_eval.h:
  - C header function that store the information of the function *f* and does partial calculation of the Jacobian.
- CMakeLists.txt:
  - We used CMake to compile the
    - CUDA solution file (jacobian.cu). The compiled executable will be named as "jacobian"
    - CPU solution file without OpenMP (jacobian_cpu.c). The compiled executable will be named as "jacobian_cpu"
    - matrix generating file (matrix.c). The compiled executable will be named as "generate"
- build.sh:
  - used to compile the OpenMP solution file (jacobian_cpu_op.c). The compiled executable will be named as "jacobian_op".

- To run the code:
  - Modify the "matrix.c" file to set the M and N values.
  - Compile "jacobian.cu", "jacobian_cpu.c" and "matrix.c" with CMake.
  - Compile "jacobian_cpu_op.c" with build.sh.
  - Run "generate" to generate a "inputArray.inp" file. No argument is needed.
  - Run solution to generate the result:
    - Run jacobian (CUDA)
      - Argument: input file name ("inputArray.inp"), output file name, value of h.
    - Run jacobian_op (OpenMP)
      - Argument: input file name ("inputArray.inp"), output file name, value of h.
    - Run jacobian_cpu (CPU)
      - Argument: input file name ("inputArray.inp"), output file name, value of h.

6. Conclusions and Future Work

We have implemented the logic in both OpenMP and CUDA and have shown that in the heavy memory access workloads, CUDA may not be the best candidate. It is possible since our evaluation functions are simply addition and multiplication, it can not hide the access latency. We propose to evaluate much heavier computing functions in future work and try to propose a different algorithm to compute Jacobian without copying memory in kernel.