Spring 2019 ME759 Final Project Report
University of Wisconsin-Madison

Default Project 1: Numerical computation of the Jacobian

Dmitrii Turygin

May 7, 2019

Abstract

Using CUDA gave significant speed up (about 20x) over sequential CPU computing for numerical computation of the Jacobian. One of the limitations was the fixed format of the function call, which required an array of arguments to be fixed for the call. This forced me to use a different approach, which was less desirable.

The code is located in CAE git repo https://git.cae.wisc.edu/git/me759-dturygin in folder "Project".

Contents

1. General information

In this important section, please indicate the following, in bulleted form and in this order:

- Mechanical Engineering
- Undergraduate
- Dmitrii Turygin
- No advisor

2. Problem statement

I am working on "Default Project 1: Numerical computation of the Jacobian". Computation of a Jacobian comes up as a part of many problems. For example, in computation fluid dynamics application, where a numerical approximation of a derivative must be computed at every point on a 1000x1000x1000 grid at every time step, so it is beneficial to speed up the simulation as a whole.

3. Solution description

Calculation of a Jacobian is surprisingly tough to parallelize. Due to restriction on the formant of the function call, all the values x_1 through x_m should be in an array in shared memory, ready for a function call. At first, I was planning to parallelize computation of a partial derivatives at a single point. However, this leads to large shared memory overhead. To parallelize computation of partial derivatives at a single point, one needs to store $2 * m$ arrays of m values, so all $2 * m$ function calls can be executed at the same time by multiple CUDA threads:

Thread 1	$x_1 + \varepsilon$	x_2	x_3
Thread 2	$x_1 - \varepsilon$	x_2	x_3
Thread 3	x_1	$x_2 + \varepsilon$	x_3
Thread 4	x_1	$x_2 - \varepsilon$	x_3
Thread 5	x_1	x_2	$x_3 + \varepsilon$
Thread 6	x_1	x_2	$x_3 - \varepsilon$

Shared memory is a scarce resource. On our hardware we only have 48 KB of it. Moreover, one needs to complete $2 * m^2 - m$ memory transactions to make copies and then add/subtract epsilon at appropriate locations. Such approach would also be difficult to scale.

I abandoned previous idea and decided to compute partial derivatives at different points at the same time. One thread completes $2 * m$ function calls to calculate a Jacobian at its point. This allows to compute Jacobian at multiple points at a cost of computing Jacobian at a single point:

	Step 1	Step 2	Step 3	Step 4	…
Thread 1	$x_1 + \varepsilon, x_2, x_3$	$x_1 - \varepsilon, x_2, x_3$	$x_1, x_2 + \varepsilon, x_3$	$x_1, x_2 - \varepsilon, x_3$	
Thread 2	$x_4 + \varepsilon, x_5, x_6$	$x_4 - \varepsilon, x_5, x_6$	$x_4, x_5 + \varepsilon, x_6$	$x_4, x_5 - \varepsilon, x_6$	
Thread 3	$x_7 + \varepsilon, x_8, x_9$	$x_7 - \varepsilon, x_8, x_9$	$x_7, x_8 + \varepsilon, x_9$	$x_7, x_8 - \varepsilon, x_9$	
…					

This approach is much more memory efficient and easy to scale. I envision this algorithm to be used, for example, in computation fluid dynamics application, where a function must be computed at every point on a 1000x1000x1000 grid. Physical functions rarely depend on more than a handful of arguments, so it does not make sense to parallelize calculation of partial derivatives at a point. It is more helpful to calculate Jacobian at more points in the same amount of time.

It is also convenient that threads are truly independent of each other. Only memory fences are required.

I only want to mention that the first approach is definitely better, but would require modification of the function call format, which is not allowed.

Kernel

One block computes Jacobian at *a* points simultaneously. It stores *a*m* arguments and *a*m* partial derivatives in its shared memory. First, every thread out of *a* threads loads its *m* arguments from global memory into shared memory. Then, every thread for every argument x_i adds ε, computes $f(x_1, x_2, \dots x_i + \varepsilon, \dots x_m)$ and stores it in shared memory, subtracts 2ε, computes $f(x_1, x_2, \dots x_i - \varepsilon, \dots x_m)$ and subtracts it from appropriate shared memory location, restores original value of x_i, computes partial derivative approximation and stores it global memory.

$$\frac{df}{dx_i} \approx \frac{f(x_1, x_2, \dots x_i + \varepsilon, \dots x_m) - f(x_1, x_2, \dots x_i - \varepsilon, \dots x_m)}{2\varepsilon}$$

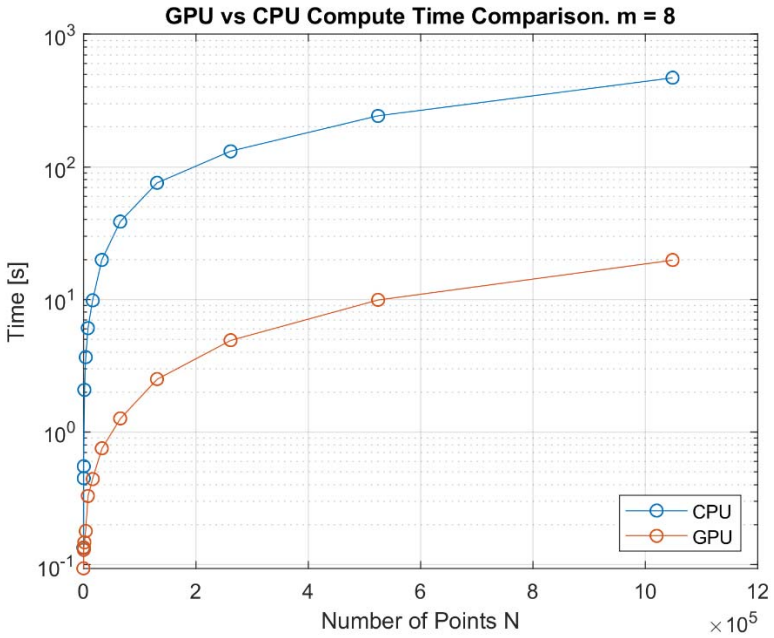
One SM has 48 KB of shared memory, which means that one SM can process a maximum of $a = \frac{48*1024}{8*2*m} = \frac{3072}{m}$ points at the same time. For $m = 8$, $a = 384$ points.

CUDA Streams

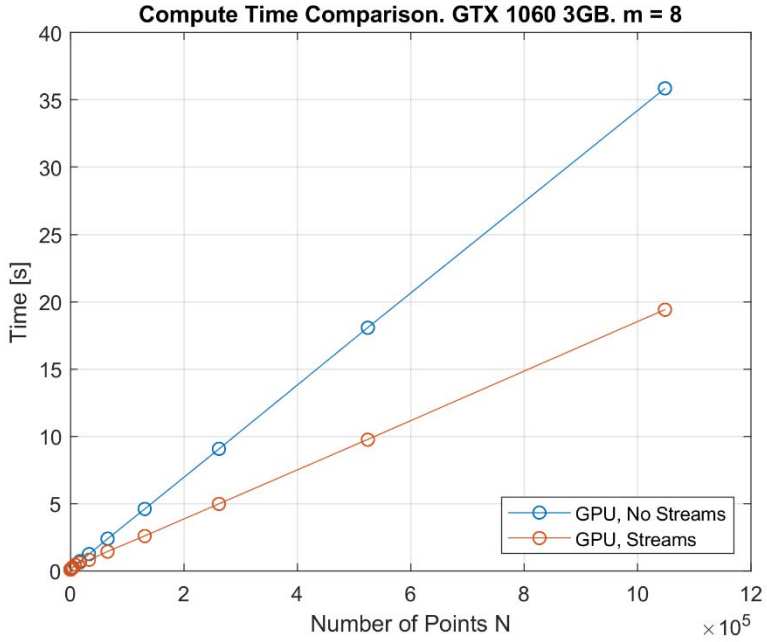
After the core of the program was completed for Milestone 1, I worked on utilizing CUDA streams to take advantage of the fact that the data can be copied to/from the GPU while the kernel executes. Based on empirical evidence, I chunked the data into 384 KB parts. This requires $\frac{384*1024}{8*a*m} = \frac{384*1024}{8*3072} = 16$ SMs. Then I use three streams synchronized using events which copy data in, copy data out or run the kernel. It took effort to make sure that everything works for any *n* and *m* values.

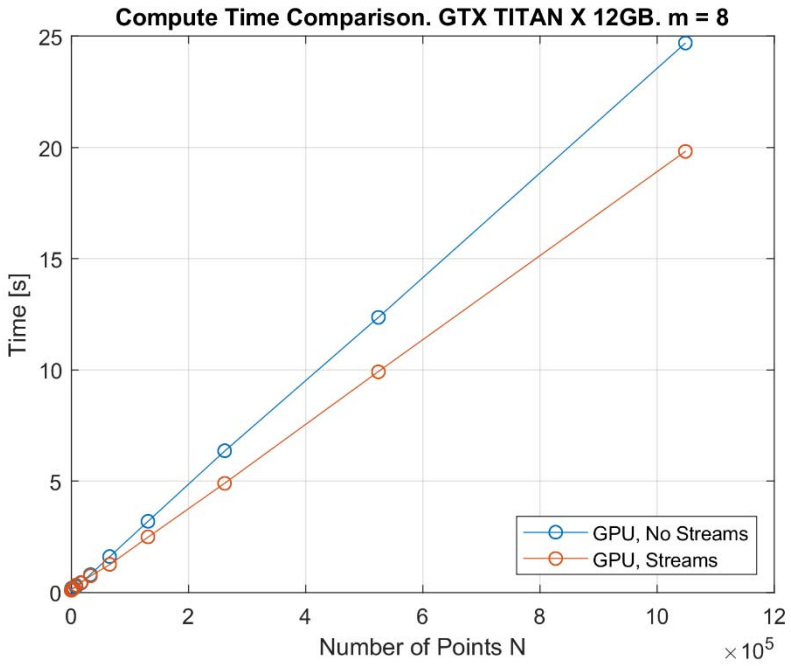
4. Overview of results. Demonstration of your project

I ran a series of studies to see how my algorithm performs. Number of function arguments *m* was kept constant at 8. Number of points where Jacobian must be calculated *n* was varied from 2^7 to 2^{20}. The function *f(..)* simply multiplies together all its arguments. Overall, GPU does allow to compute Jacobian much faster. GPU compute time does slowly increase, as more data needs to be transferred in and out. For $n = 2^{20}$ CPU took 469.02 seconds, while GPU on Euler only took 19.82 seconds.

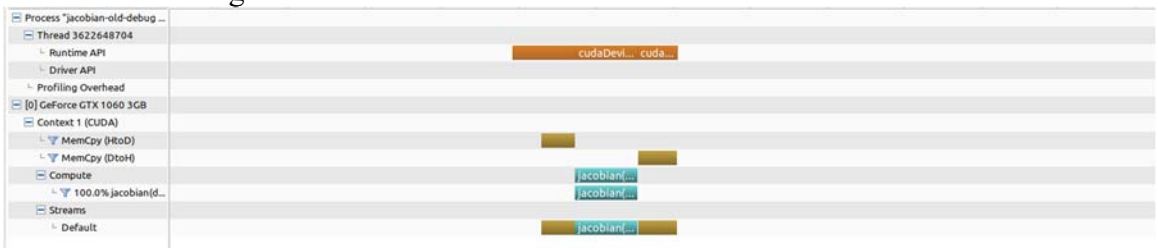


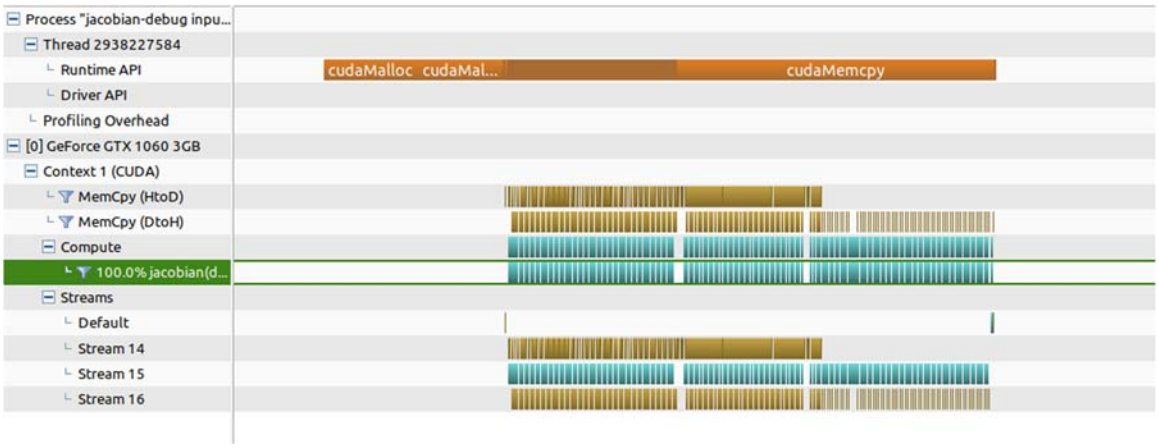
I ran into some consistency issues, so I ran the older version of the code without streams and the newer version with streams both on GTX 1060 3GB at home and GTX TITAN X 12GB on Euler. The boost is significantly larger for GTX 1060.





Here are the results from Nvidia Visual Profiler for $n = 2^{20}$. Version with no streams on the top and with streams on the bottom. Note that the scale is not the same. It is evident that I was able to take overlap data movement and kernel execution, however the chunk size might be too small.





5. Deliverables:

This report is uploaded to Canvas. My code is in CAE git repo https://git.cae.wisc.edu/git/me759-dturygin. It contains code for three executables:

- jacobian.cu – the latest version of code with CUDA streams
- jacobian_old.cu – the previous version without streams
- generator.cu – source for a program that takes n and m as two parameters and generates input files
- jacobian_cpu.cu – sequential implementation

Running **cmake .; make** compiles *jacobian* executable based on jacobian.cu. There is also a results folder with raw results from the runs and some *.sh files used by me to run the code.

6. Conclusions and Future Work

Overall, I am pleased with the results. The computation was sped up significantly (about 20x), and I got a chance to learn about CUDA streams, which did give a substantial performance boost. I believe that chunk size of 384 KB is too small, so there needs to be a study to determine the best size. Also, my code allocates all the required memory to store m*n values, which is wasteful and not necessary when using streams. There only needs to be two chunk-sized memory sections on the device that would be swapped back and forth.