cuCALC - A CUDA calculus library

Dheemanth Manur[1] and Aditya Hegde[2]

***Abstract*— This paper presents a new shared CUDA library for numerical methods for calculus functions, i.e a collection of highly accurate standard calculus functions. This library interface also provides a way to pass the functions dynamically.**
Keywords - //later

I. INTRODUCTION

Over the last decade parallel computing have been on the rise, as the hardware architecture is improving and making ways to better software implementation in different areas. There are multiple libraries dedicated to mathematical methods in CUDA, but none specific to calculus. Calculus is one of the area which demands for high compute power. Since It has the potential to be parallelized, when ran in a massively parallel environment shows significant improvement in runtime and accuracy. We provide a new dynamic CUDA library with calculus functions which run on GPU, available to all CUDA C or C++ applications. The cuCALC library ensures that your application benefits from high performance calculus routines optimized for the NVIDIA GPU architectures.

The remainder of paper is organized into 6 sections. Section 2 discusses the background about CUDA architecture. Section 3 reviews the implementation of library and techniques used. Section 4 discusses the calculus methods implemented and parallel technique used in each method. Section 5 is related works and Section 6 is conclusion.

II. BACKGROUND

A. NVIDIA CUDA Programming Model

NVIDIA's CUDA (Compute Unified Device Architecture) is a scalable parallel programming model and software platform for the GPU and other parallel processors that allow the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++. CUDA uses a SPMT (single-program, multiple Threads) style parallelism, programs are written for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU.

B. Need for cuCALC

Rise of Machine Learning/Artificial Intelligence and Scientific Computing has led towards more GPU intensive computation which helps to accelerate the execution by breaking down massively complex problems into multiple smaller simultaneous calculations. This makes them ideal for handling the massively distributed computational processes. As calculus is core part of solving problems and is used on a day to day basis, there is need for a library dedicated to calculus methods which users can use directly to make use of functions and get result without much hassle.

C. Related Work

- CUDA libraries related to mathematics at the time of this paper cuBLAS, cuFFT, cuRAND, Thrust, MAGMA, cuSPARSE, ArrayFire, NVRTC. These libraries does not include any numerical calculus methods.
- The paper *A numerical differentiation library exploiting parallel architectures* discusses regarding how numerical differentiation methods can be parallelized.
- *GSL - GNU Scientific Library* is the current industry standard library for scientific computing and is the backend for *scipy* - A popular python library. It provides all the widely used mathematical functions covering numerical analysis including interpolation, calculus and linear algebra. But the library does not make use of any parallelism and GPU for accelerating its functions.

III. IMPLEMENTATION DETAILS

Library is provided as a CUDA dynamic library. We're using C++11 and CMake for building the project. The library allows users to pass simple mathematical functions written as C++ functions marked with `__device__` attribute, which will be executed in parallel within the libraries kernel.

A. Separable Compilation

Our library is shipped as dynamic library, which allows device functions to be passed across the library boundary. There is no straightforward way to do this in CUDA, as compilation of CUDA works by embedding device code into host objects. Due to that in default mode device functions of different files are not visible to each other. Since CUDA 5.0, separate compilation of device code is supported. In separate compilation, we embed relocatable device code into the host object, and run `nvlink`, the device linker, to link all the device code together. The output of nvlink is then linked together with all the host objects by the host linker to form the final executable. CMake has first class support for separate compilation, with the option `CUDA_SEPARABLE_COMPILATION` set, It will handle the entire process.

*This work was not supported by any organization

[1]D. Manur is a Graduate Computer Science student at Bourns College of Engineering, University of California, Riverside dmanu006 at ucr.edu

[2]A. Hegde is a Graduate Computer Science student at Bourns College of Engineering, University of California, Riverside ahegd005 at ucr.edu

B. Steps to build and run the test executables

Clone the git repo and execute the following commands,

1) `mkdir build && cd build`
2) `cmake3 ..`
3) `make`
4) `./tests/test_cucalc_integration` (And other tests are present in the same directory)

Note : cmake3 minimum version required is 3.17, which is available on the bender server, ensure `cmake3` is the command used.

IV. Calculus Methods implemented in cuCALC

A. Numerical Differentiation

Numerical Differentiation methods have been implemented for first-order techniques, encompassing backward difference, forward difference and central difference for given steps between two points.

Parallelism is achieved by launching a kernel to calculate function values between two points for each step in parallel and another kernel is launched to calculate differentiation based on invoked backward difference, forward difference or central difference and final resulting array of the difference are returned to application program. Parallelism is needed to calculate a derivative curve for all the points within the limits. These are divided into threads and differentiate at that particular point.

$$\text{Central difference} = \frac{df(x)}{dx} = \frac{f(x+h)-f(x-h)}{2h}$$

$$\text{Forward difference} = \frac{df(x)}{dx} = \frac{f(x+h)-f(x)}{h}$$

$$\text{Backward difference} = \frac{df(x)}{dx} = \frac{f(x)-f(x-h)}{h}$$

B. Numerical Integration

Numerical Integration is method of calculating area under the curve within upper and lower bound. We have implemented 4 variation of the newton-cotes formulas, which are

1) Trapezoidal Rule
2) Simpson's 1/3rd Rule
3) Simpson's 3/8th Rule
4) Boole's Rule

These algorithm's work by dividing the area under the curve into smaller and smaller objects and summing up the area. Due to this nature it these algorithms naturally tends to parallelism. Another benefit of using parallelism in these algorithms is algorithm's accuracy also depends on the number of smaller objects that we can break down the area. Therefore massive parallelism makes the algorithm faster and as well as more accurate. These individual areas are each processed by on thread, calling the original function and applying the algorithm to estimate the areas based on that.

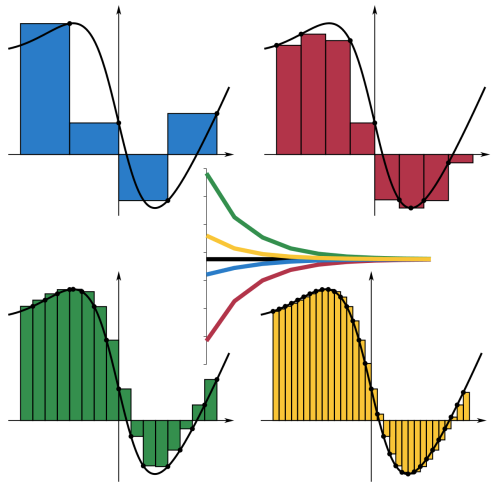

Fig. 1. Image depiting how accuracy increases on smaller sub areas. Image credit : I, KSmrq, CC BY-SA 3.0

C. Reduction Sum

A Reduction Sum method is implemented in common utility which is used by multiple inner methods to get sum of values. Implementation is an optimized reduction sum which takes an array of values and sum is calculated by performing repeated call to reduction method until a single block is left and a single value is returned.

V. Performance

For performance we will compare the sequential execution times integral calculation with parallel version. For 2^{28} steps

```
$ ./tests/test_cucalc_integration
GPU output : 1024.000000
GPU time : 0.891498 s
CPU output : 1024.000000
CPU time : 6.530141 s
cucalc_integration_trapez: test passed!!
```

We see that GPU provides a massive speed up compared to the CPU version.

VI. Future Work

We aim to completely provide all the mathematical functions used in scientific computing, covering partial differential equations and ordinary differential equations, competing with *GSL*. We also plan to enable architecture specific optimizations to accelerate further.

VII. Conclusions

In conclusion, this comprehensive library stands as a robust and versatile solution, offering a multitude of highly optimized numerical methods for solving calculus functions. With a focus on accuracy, efficiency, and harnessing the parallel computing power of CUDA-enabled GPUs, this library empowers researchers, scientists, and engineers across domains.

ACKNOWLEDGMENT

We extend our sincere gratitude to Professor Danial Wong and teaching assistants for their invaluable guidance, support during the development of this project. Their expertise and encouragement have been instrumental in shaping this project.

REFERENCES

[1] Programming Massively Parallel Processors: A Hands-on Approach Book by David Kirk and Wen-mei Hwu
[2] https://en.wikipedia.org/wiki/Numerical_differentiation
[3] https://www.gnu.org/software/gsl/doc/html/#gnu-scientific-library
[4] https://www.sciencedirect.com/science/article/abs/pii/S0010465509000484
[5] https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas
[6] https://commons.wikimedia.org/w/index.php?curid=2347919