MAP55616-03 - Cuda Exponential Integral calculation
Posted on: Tuesday, 27 May 2025 09:59:11 o'clock IST
Please note the extension of the deadline to Tuesday the 3rd of June of 2025, at 17:00

MAP55616-03 - Cuda Exponential Integral calculation
The goal of this assignment is to develop a fast cuda implementation from the provided exponential integral source code. This code computes the integrals of the exponential functions, from E_0 to E_n. You can find more information about the algorithm in:
http://mathworld.wolfram.com/En-Function.html
http://mathworld.wolfram.com/ExponentialIntegral.html
Or in any of the Numerical Recipes books (for example, in page 266 in the Numerical Recipes Third Edition C++).
You must submit proof of progress in the form of git repositories for both the source code and the report, so make sure that you label the commits at the most significant points of the development (cpu code working, transfers to the device, basic gpu implementation, each advanced gpu implementation, transfers back to the host, and so on.). If you do not include proof of progress, the marks for the parts involved in the source code and report will be halved!
Please note that since, at this time of the year, you will be busy starting with your project, this assignment can be done very quickly (by turning the exponentialIntegralFloat and exponentialIntegralDouble functions into cuda kernels) or it can take quite time time (if you try to add all the optional choices). It is meant to give you a reason to test cuda advanced features, so if you are short on time, you can stick to a basic cuda implementation. On the other hand, if you want to test advanced cuda techniques - specially so if you want to familiarize yourself with them so you can use them in your summer project - your time investment will be rewarded with extra marks.
Task 1 - cuda implementation
Starting from the provided source code (in exponentialIntegralCPU.tar), modify the main.cpp and add .h and .cu files that contain your cuda code to calculate both the floating point and double precision versions of the algorithm that has already been implemented in the CPU. Alter the code so the GPU code is executed unless the program is run with the "-g" flag (see the parseArguments function in the provided code, which skips the cpu part when passing a "-c" flag as an argument).
The cuda implementation must time *all* the cuda parts of the code (this means including memory transfers and allocations). Add as well separate time measures for both the single and double precision versions (so we can see the difference in performance between either precision in cuda). Calculate the speedup for the total cuda timing (that is, including the memory allocations, transfers and execution) relative to the CPU.
Add a numerical comparison between the results obtained from the gpu and the cpu. If any values diverge by more than 1.E-5 (which shouldn't happen), report them.
There are no restrictions on which cuda techniques can be used for this assignment, except for not using libraries other than cuda itself.
Most of the marks will be given for good performance of the implemented code (this means including the memory transfers when measuring the time). Additional marks will be rewarded for any extra work that can be conducted as an optional choice, such as:
* Testing the usage of the other memories (such as constant, shared, texture, surface, etc) to save register memory, so it doesn't get demoted to local memory, even if it does not provide performance benefits.
* Using streams to let the compute and memory transfers overlap, even if it does not provide performance benefits.
* Testing any other advanced cuda technique (such as dynamic parallelism, cooperative groups, etc), even if they do not provide performance benefits.
Run the final versions of the program with the following sizes:
-n 5000 -m 5000
-n 8192 -m 8192
-n 16384 -m 16384

-n 20000 -m 20000
Find the best grid sizes for each problem size (do not forget to test cases in which the n and m values are different, as they have to work as well!) and report the times and speedups relative to the CPU.
Task 2 - LLM implementation
Input the relevant code (this would be the loops calling exponentialIntegralFloat and exponentialIntegralDouble and the contents of the functions) into an LLM such as ChatGPT (or cuBot), Cursor, Copilot, etc. Compare the code that they provide with your results. Make sure to report which particular LLM you tested.
Check and report if the LLM code generated correct results.
Report the features that the LLM recommended to implement, and if they indeed increased the performance.
If you test more than one LLM, report which one produced the best results (as in better performance with correct results).

==========

Submit a tar ball with your source code files (including a working Makefile for cuda01), speedup graphs and a writeup of what you did and any observations you have made on the behaviour and performance of your code, as well as problems that you came across while writing the assignment. You do not need to include your source code in the report as it will be checked independently. Focus the report on why you chose a particular method to optimize the code and how it worked (or didn't).
Note: Again: Since, at this time of the year, you will be busy starting with your project, this assignment can be done very quickly (by turning the exponentialIntegralFloat and exponentialIntegralDouble functions into cuda kernels) or it can take quite time time (if you try to add all the optional choices). It is meant to give you a reason to test cuda advanced features, so if you are short on time, you can stick to a basic cuda implementation. On the other hand, if you want to test advanced cude features - specially so if you want to familiarize yourself with them so you can use them in your summer project - your time investment will be rewarded with extra marks.
Note: Do not forget to include the proof of progress!
Note: Marks will be deducted for tarbombing. http://en.wikipedia.org/wiki/Tar_%28computing%29#Tarbomb
Note: Remember than in this case, instead as it was in previous assignments, the same piece of software has to compute both the single precision and double precision versions.
Note: Extra marks will be given for separating the C/C++ and cuda code. You can find examples on how to do that on the "makefileCpp" and "makefileExternC" sample code.
Note: Remember that the code must work for non-square systems too, even when, for this assignment, we are using square ones for benchmarking. You can run use compute-sanitizer to test that, as in: /usr/local/cuda-12.8/bin/compute-sanitizer ./my_exec
Note: When you are benchmarking the performance of your code, you can check the current load on cuda01 with the nvidia-smi command.
Note: When benchmarking the gpu code, remember to use the -c flag so the cpu part doesn't have to be run each time (the -n 20000 -m 20000 takes about 120 seconds)
Note: When benchmarking the second assignment, you probably noticed that cuda01 usually takes a while to run the first cuda command if the GPU is idle. I think that I have fixed the problem, so it should not be a problem when measuring the initial transfers.
Deadline: Monday the 26th of May of 2024, at 17:00

==========

Course Link/Files/MAP55616-03 - Cuda Exponential Integral calculation