CSCI-4320/6360 - Assignment 2: Parallel CUDA Program for a 32M Bit Carry-Lookahead Adder

Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180-3590

February 3, 2023

DUE DATE: 11:59 p.m., Friday, February 17th, 2023

1 Overview

You are to construct a parallel CUDA program that specifically computes a 33,554,432 bit Carry Lookahead Adder (yes, that's a 32M bit adder!!) using 32 bit blocks which is also the CUDA warp size.

For a list of bitwise operators and the associated C language syntax, please see: `https://en.wikipedia.org/wiki/Operators_in_C_and_C\%2B\%2B`. Make sure you used the older ANSI-C operators and not the newer operator forms/syntax (e.g., XOR is ^ in C).

Recall, that this specific CLA adder can be constructed from the follow:

1. Calculate g_i and p_i for all 33,554,432 bits i.
2. Calculate gg_j and gp_j for all 1,048,576 groups j using g_i and p_i.
3. Calculate sg_k and sp_k for all 32,768 sections k using gg_j and gp_j.
4. Calculate ssg_l and ssp_l for all 1,024 super sections l using sg_k and sp_k.
5. Calculate $sssg_m$ and $sssp_m$ for all 32 super sections m using ssg_l and ssp_l. *Note, it is at this point, we can shift to computing the top-level sectional carries. This is because the number of sections is less than or equal the block size which is 32 bits.*
6. Calculate $sssc_m$ using $sssg_m$ and $sssp_m$ for all m super sections and 0 for $sssc_{-1}$.
7. Calculate ssc_l using ssg_l and ssp_l and correct $sssc_m, m = l \; div \; 32$ as super super sectional carry-in for all sections l.

8. Calculate sc_k using sg_k and sp_k and correct $ssc_l, l = k \; div \; 32$ as super sectional carry-in for all sections k.

9. Calculate gc_j using gg_j, gp_j and correct $sc_k, k = j \; div \; 32$ as sectional carry-in for all groups j.

10. Calculate c_i using g_i, p_i and correct $gc_j, j = i \; div \; 32$ as group carry-in for all bits i.

11. Calculate sum_i using $a_i \oplus b_i \oplus c_{i-1}$ for all i where $\oplus$ is the exclusive-or or XOR operation.

You need to construct a CUDA program that executes on `AiMOS` that reproduces this algorithm. Each step in the above algorithm will be implemented as a separate CUDA kernel function.

A template is provided that generates deterministic random hex input data and outputs the result of a comparison of the CLA result in binary with a Carry Ripple Adder (CRA).

More specifically, your program will do the following:

1. Convert all `calloc` and `malloc` C program memory calls to using `cudaMallocManaged` for the data arrays that the CUDA kernels will need to operate over.

2. Convert all CLA routines specified in the template from C to CUDA kernel calls.

3. Use the CUDA `add.cu` and `reduction.cu` programs as guides. Here, each CUDA thread will operate on a single "bit" at a time. So you are effectively unrolling the for-loops in the C serial code.

4. *Note, you are free to be a little inventive, pull from your own C code in Assignment 1 and you don't have to rely/port the provided C serial code. E.g., you don't have to port the* `grab_slice` *function into CUDA.*

5. *Note, do no modify the timing routines or the comparison test code.*

2 Testing and Correctness

The testing of this program is simple and straightforward. A deterministic random input pattern is generated for two hex inputs which are then converted to binary inputs. Next, the CUDA CLA adder will be invoked as well as a serial Ripple Carry Adder. The results from the two adders will be compared. You must pass this test for your program to be considered correct. If the test fails, you will be informed where the first bit position that differed from the Ripple Carry Adder. The TA will run your programs by hand to make sure you code runs correctly on AiMOS. *This is the only correctness test your adder must pass.*

3 One Page Performance Report

In this report, you will briefly describe how you implemented each of the CLA adder functions as CUDA kernel calls. Next, you will use the provided cycle timer function `clock_now()` to report the total number of clock cycles consumed by the CLA function and Ripple Carry Adder function **IN SERIAL** using the `cla-serial` program provide with the template. Also, report the number of cycles for the CUDA CLA adder function in your `cla-parallel` program for a number of different block sizes as noted below. Note, that on AiMOS, the cycle timer has a clock rate of 512,000,000 cycles. So if you divide your reported number of cycles (cast to `double`) by this clock rate (cast to `double`), it will convert cycles to seconds (in double/64 bit precision).

CUDA block sizes (not CLA block size which is fixed at 32) are: 32, 64, 128, 256, 512 and 1024. Which CUDA block size yields the best performance? Why do you think that is the case?

Note, this template code because of the `clock_now()` function will only compile and run on the AiMOS POWER9 system.

Last, compute the speedup obtained for the CUDA CLA function (using the BEST/fastest block size configuration) relative (1) the serial CLA function and (2) the serial Ripple Carry Adder.

What do you observe?

How much faster is the CUDA CLA function than the serial CLA function?

Why might the Ripple Carry Adder be faster or slower than the CUDA CLA?

This report must be in PDF format!

4 HAND-IN and GRADING INSTRUCTIONS

Please submit your complete CUDA code (all *.c, *.cu, *.h and Makefile files) and your Performance Report in PDF format to the `submitty.cs.rpi.edu` grading system.