%%cu

#include<iostream>

#include <string>

#include <vector>

#include <list>

#include <iostream>

#include <chrono>

#include <cstdlib>

const char ETX = '$';

std::pair<std::string,int\*> bwt\_with\_suffix\_array(const std::string sequence);

std::string bwt(const std::string sequence);

const int blockSize = 256;

/\*

generates a list of ints where the value of each item in the list is it's index,

unless it is greater than the sequence length, in which case it is set to -1.

\*/

\_\_global\_\_ void generate\_table(int\* table, int table\_size, int n) {

int index = blockIdx.x \* blockDim.x + threadIdx.x;

int stride = blockDim.x \* gridDim.x;

for(int i = index; i < table\_size; i+=stride) {

if( i < n) {

table[i] = i;

} else {

table[i] = -1;

}

}

}

/\*

compare two rotations of the input sequence lexicographically

a, b are index pointers to the index of the start of each rotation

\*/

\_\_device\_\_ bool compare\_rotations(const int& a, const int& b, char\* genome, int n) {

if (a < 0) {

return false;

}

if (b < 0) {

return true;

}

for(size\_t i = 0; i < n; i++) {

if (genome[(a + i) % n] != genome[(b + i) % n]) {

return genome[(a + i) % n] < genome[(b + i) % n];

}

}

return false;

}

\_\_global\_\_ void bitonic\_sort\_step(int\* table, int table\_size, int j, int k, char\* genome, int n) {

unsigned int i = threadIdx.x + blockDim.x \* blockIdx.x;

unsigned int ixj = i ^ j;

if(i < table\_size) {

if(ixj > i) {

if ((i & k) == 0) {

if (compare\_rotations(table[ixj], table[i], genome, n)) {

int temp = table[i];

table[i] = table[ixj];

table[ixj] = temp;

}

}

if ((i & k) != 0) {

if (compare\_rotations(table[i], table[ixj], genome, n)) {

int temp = table[i];

table[i] = table[ixj];

table[ixj] = temp;

}

}

}

}

}

\_\_global\_\_ void reconstruct\_sequence(int\* table, char\* sequence, char\* transformed\_sequence, size\_t n) {

int index = blockIdx.x \* blockDim.x + threadIdx.x;

int stride = blockDim.x \* gridDim.x;

for(int i = index; i < n; i+=stride) {

transformed\_sequence[i] = sequence[(n + table[i] - 1) % n];

}

}

/\*

returns a std::pair object

the first item is the burrows wheeler transform of the input sequence in a std::string,

the second item is the suffix array of the input sequence, represented as indicies of the given suffix, as an int\*

assumes input sequence already has ETX appended to it.

\*/

std::pair<std::string,int\*> bwt\_with\_suffix\_array(const std::string sequence) {

const size\_t n = sequence.size();

size\_t table\_size = sequence.size();

// round the table size up to a power of 2 for bitonic sort

table\_size--;

table\_size |= table\_size >> 1;

table\_size |= table\_size >> 2;

table\_size |= table\_size >> 4;

table\_size |= table\_size >> 8;

table\_size |= table\_size >> 16;

table\_size++;

int\* table\_cu;

cudaMalloc(&table\_cu, table\_size \* sizeof(int));

int\* table = (int\*) malloc(table\_size \* sizeof(int));

int numBlocks = (table\_size + blockSize - 1) / blockSize;

std::cout << "generating table" << std::endl;

generate\_table<<<numBlocks,blockSize>>>(table\_cu, table\_size, n);

// wait for cuda kernel to finish

cudaDeviceSynchronize();

std::cout << "sending data to device" << std::endl;

char\* sequence\_cu;

cudaMalloc(&sequence\_cu, n \* sizeof(char));

cudaMemcpy(sequence\_cu, sequence.c\_str(), n \* sizeof(char), cudaMemcpyHostToDevice);

std::cout << "running sort" << std::endl;

size\_t j,k;

for (k = 2; k <= table\_size; k <<= 1) {

for (j = k >> 1; j > 0; j = j >> 1) {

bitonic\_sort\_step<<<numBlocks,blockSize>>>(table\_cu, table\_size, j, k, sequence\_cu, n);

}

}

cudaDeviceSynchronize();

std::cout << "done sorting" << std::endl;

char\* transformed\_sequence\_cu;

cudaMalloc(&transformed\_sequence\_cu, n \* sizeof(char));

numBlocks = (n + blockSize - 1) / blockSize;

std::cout << "reconstructing sequence" << std::endl;

reconstruct\_sequence<<<numBlocks,blockSize>>>(table\_cu, sequence\_cu, transformed\_sequence\_cu, n);

char\* transformed\_sequence\_c = (char\*) malloc(n \* sizeof(char));

cudaDeviceSynchronize();

// old time for cpu method:

// killed after 40 min

std::cout << "copying data from device" << std::endl;

cudaMemcpy(transformed\_sequence\_c, transformed\_sequence\_cu, n \* sizeof(char), cudaMemcpyDeviceToHost);

std::string transformed\_sequence(transformed\_sequence\_c, n);

cudaMemcpy(table, table\_cu, table\_size \* sizeof(int), cudaMemcpyDeviceToHost);

cudaFree(table\_cu);

cudaFree(sequence\_cu);

return std::make\_pair(transformed\_sequence,table);

}

std::string bwt(const std::string sequence) {

auto data = bwt\_with\_suffix\_array(sequence);

free(data.second);

return data.first;

}

#define NOW() std::chrono::high\_resolution\_clock::now()

std::string bwt\_cpu(const std::string sequence) {

const size\_t n = sequence.size();

const char\* c\_sequence = sequence.c\_str();

std::vector<int> table(n);

for (size\_t i = 0; i < n; i++){

table[i] = i;

}

std::list<int> sorted\_table(table.begin(), table.end());

sorted\_table.sort([c\_sequence,n](const int& a, const int& b) -> bool {

for(size\_t i = 0; i < n; i++) {

if(c\_sequence[(a + i) % n] != c\_sequence[(b + i) % n]) {

return c\_sequence[(a + i) % n] < c\_sequence[(b + i) % n];

}

}

return false;

});

std::string transformed\_sequence;

for(auto r = sorted\_table.begin(); r != sorted\_table.end(); ++r){

transformed\_sequence += c\_sequence[(n + \*r - 1) % n];

}

return transformed\_sequence;

}

int main(int argc, char const \*argv[])

{

std::string alphabet("ATCG");

const int N = (argc > 1) ? atoi(argv[1]) : 1E6;

std::cout << "running sample of " << N << std::endl;

char\* sequence = (char\*) malloc((N+1) \* sizeof(char));

for (size\_t i = 0; i < N; i++) {

sequence[i] = alphabet[rand() % alphabet.size()];

}

sequence[N] = ETX;

// TODO: make time dynamic so we don't end up with 0

std::cout << "running cpu version..." << std::endl;

auto start = NOW();

// auto cpu\_seq = bwt\_cpu(sequence);

auto cpu\_seq = "";

auto cpu\_time = std::chrono::duration\_cast<std::chrono::milliseconds>(NOW() - start);

std::cout << "running gpu version..." << std::endl;

start = NOW();

auto gpu\_seq = bwt(sequence);

auto gpu\_time = std::chrono::duration\_cast<std::chrono::milliseconds>(NOW() - start);

std::cout << "cpu version: " << cpu\_time.count() << "ms" << std::endl;

std::cout << "gpu version: " << gpu\_time.count() << "ms" << std::endl;

// TODO: make this optional

// if(cpu\_seq.compare(gpu\_seq) == 0) {

// std::cout << "outputs match!" << std::endl;

// } else {

// std::cout << "uh oh, outputs mismatch, something went wrong!" << std::endl;

// }

return 0;

}