Parallel Analytics & Quantifiable Extensions of Datasets

# Introduction

As we’ve explored parallel computing this semester, we’ve gained a greater understanding of its applications in our mundane, day-to-day operations. We've explored various algorithms and applications of parallel computing. That’s why, for our final project, we decided to explore a common and very important application of parallel computing – data processing.

We decided to make a simple program that could take numerical data and perform various operations on it. Operations we performed on data included reduce/sum, counting a particular value, inclusive scan, exclusive scan, sorting, minimum and maximum.

The simple operations (such as the ones above) were mostly implemented using the thrust API in order for us to test the difference in speeds for such a popular API. We decided that if it was already implemented (and it would be implemented better by people who are more knowledgeable about the field than us), then we may as well use the API in this project. We wanted to use the API to both learn how to use it (to further our knowledge in the field) and to see how it compares to serial algorithms through our own testing.

We also implemented some functions that were original, but also somewhat used the thrust API for some parts of the algorithm. The data algorithms we implemented along-side the other functions. The functions we implemented independently include:

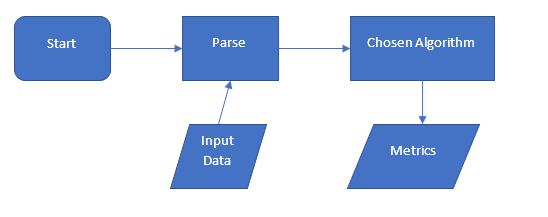
* + TEEN (Threshold-sensitive Energy Efficient sensor Network protocol) [1]
    - Teen is an algorithm that revolves around collecting relevant data from sensor nodes. It collects data based on two thresholds, one being a hard threshold, absolute lower and/or upper bounds where data must be sent. The other threshold is a soft threshold, where the system only wants the data if the value has changed by the amount of the soft threshold from the last value.
  + DSSS (Direct Sequence Spread Spectrum) inspired encryption [1]
    - This algorithm takes data and, using a set of given values, encrypts the data via a bitwise xor operation. Both encryption and decryption were implemented.

As for other programs that are similar to ours, Excel and MySQL do similar operations. The difference between those programs and ours is that it requires no hefty set-up time, can easily be run from a script, and does very simple operations.

For our independently implemented algorithms, we were focused time complexity rather than space complexity, as they took up as much space as possible for the parallelization. Anything we could parallelize easily, we did. We also used thrust API functions within TEEN and DSSS.

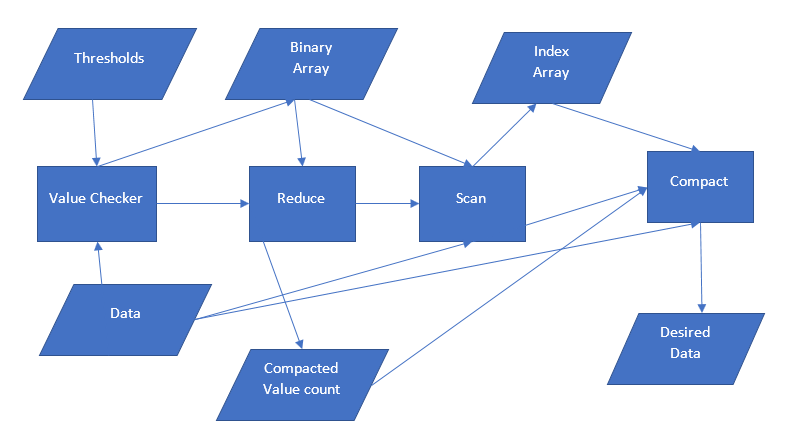
# Design and Optimization Approach

At a high-level, our program ingests data and processes it to produce various statistical results. Thrust is used for a large among of general operations to prove that the operations were faster and more efficient by a good margin. For our in-house algorithms, we used a combination of our own code and some of thrust's operations.



*Figure 1: General operations*

For the TEEN algorithm, the data is brought in along with desired thresholds (a lower hard threshold, an upper hard threshold, and a general soft threshold). Then, a binary array of 1s and 0s is made, with 1s indicating that is has crossed one of the thresholds. From there, a reduce is performed to determine how many values passed the threshold. After that, a scan is performed in order to determine where each value we want to extract is located. After that, a compact is performed to return the values [2].



*Figure 2: TEEN algorithm.*

For the DSSS inspired algorithm, values are taken in. Then, an array for encryption is randomized. After encrypting the values (the message) with the key (the randomized array) via bitwise operations, the encrypted values and key are returned, only to be later used with the decryption algorithm to retrieve the original values.

We had more than a few problems implementing this program. CUDA is always going to be difficult with it's very specific calls and the fact that you can't send things like vectors to kernels. However, thrust was slightly more difficult to integrate than we originally thought it would be. It's device and host vectors are definitely helpful if thrust is all that is going to be used in a program, but we also used kernels. Thrust vectors cannot be sent to kernels and our algorithms were multi-step in some cases [3]. Additionally, we wanted to input and output standard vectors and, while there are provision for changing standard vectors to host vectors and then device vectors, there are no provision for turning them back into standard vectors [3]. The thrust API was very challenging to integrate in these regards. We used the OSC to implement the program and test it, but that also cause problems from slow connection to the system and lags in sending commands. Over all, this project was surprisingly hard to implement given the subject matter.

# Application Performance Analysis and Project Results

For our project, we tested all of the algorithms with a serial and parallel version and used 100,000 values for each to get the times. Our results were:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Function | s-real | s-user | s-sys | p-real | p-user | p-sys |
| Sum | 14.297 | 9.044 | **5.24** | 14.412 | 8.983 | **5.42** |
| Average | 14.342 | 9.071 | **5.253** | 14.48 | 9.106 | **5.347** |
| Minimum | 14.445 | 9.091 | **5.324** | 14.387 | 9.059 | **5.302** |
| Maximum | 14.402 | 9.04 | **5.33** | 14.359 | 9.086 | **5.223** |
| Count | 14.393 | 9.087 | **5.259** | 14.373 | 8.958 | **5.387** |
| TEEN | 15.394 | 9.103 | **5.501** | 15.425 | 9.149 | **5.463** |
| DSSS | 14.434 | 9.212 | **5.19** | 14.457 | 9.225 | **5.212** |
| Scan I | 15.393 | 9.112 | **5.462** | 15.541 | 9.12 | **5.554** |
| Scan E | 15.694 | 9.082 | **5.557** | 15.628 | 9.135 | **5.517** |
| Ingest | 14.261 | 9.018 | **5.227** | 14.261 | 9.018 | **5.227** |

We also did a test on just the functions we created with 10,000 values:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Function | s-real | s-user | s-sys | p-real | p-user | p-sys |
| TEEN | 0.699 | 0.137 | **0.446** | 0.625 | 0.156 | **0.422** |
| DSSS | 0.543 | 0.138 | **0.393** | 0.532 | 0.136 | **0.384** |

Where 's' and 'p' stand for serial and parallel respectively, 'real', 'user' and 'sys' were the times given by bash, and the bold columns were used for comparison. From the times, we were surprised to see that reduce/sum, average, count and exclusive scan were actually quicker via serial. We suspect that this is due to data transfer between the main program and thrust's API, where transforming data is necessary. With less values, the functions actually were faster (data not included, but for 100,000 values those functions are not worth the effort of implementing as parallel.

From the chart, it can also be seen that minimum, maximum, and exclusive scan are quicker. Perhaps such speed-ups have to do with the nature of the algorithms. So, we determined that using those functions from thrust are worth even the obvious data representation hurdles we had to jump around to implement it. With even more values, such an algorithm is even more useful.

For our own algorithms, they ran slower when used with 100,000 values but faster when used with 10,000. We think this is due to partially the data representation (vectors) and partially due to the number of resources we had. 100,000 parallel operations may have been too much, and the 10,000 value metrics shows that the algorithms are indeed faster even with the difficulties and slow-downs due to data representaiton.

As for additional improvements, it would be beneficial to change the data representation from std::vector to array in order to remove some of the information hurdles involved. In other projects involving parallelism, we recommend not using standard vectors. They are highly inefficient in such applications. Additionally, more optimizations and parallelism could be added, along with more interesting functions to test whether they would be better implemented in parallel.

# Division of Work

As a team, we each specialized in various backgrounds. Cicely focused primarily on documentation, parallelization research, and parallelization implementation. Stephen focused predominantly on documentation, framework development/implementation, and parallelization research. We were each responsible to actively shape the final product and hold one another accountable to completing our work in a timely manner. As deadlines approached and conflicts arose, we collectively load balanced to maintain our tracking towards completion and worked to overcome adversity when presented.

## Cicely’s Assessment

Parallel computing is something I find rather difficult to implement. It is not difficult because of the concepts, but it is difficult for me because of all of the API/framework nuances that come with CUDA, thrust, and using C++ as a parallel platform. For this project, I did most of the parallel programming, some serial programming, report-writing, and research. I implemented TEEN and the DSSS inspired algorithm and used the thrust API frequently. I enjoyed working on the project when I was trying to determine how to perform an algorithm, but often became frustrated with errors associated to the complexities of CUDA, thrust, and C++ combined.

## Stephen’s Assessment

As a student, this coursework was challenging and engaging and I was excited to implement it within a project environment. Parallelization I believe will largely shape the future of computing and thus is extraordinarily relevant to my ongoing education and the wellbeing of computer science. In this effort, I believe I was able to actively contribute and shape, grow, envision, research, and implement this project. Though it hit snags along the way, I worked to actively maintain its course and contribute in tandem with Cicely to produce a solid product worthy of this assignment and my partner’s invested time.

# Code Appendix

//MAIN FILE

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

#include <vector>

#include <algorithm>

#include <numeric>

#include <typeinfo>

#include "./parallel\_code.cu"

template <class T>

T sum(std::vector<T> data) {

T result = std::accumulate(data.begin(), data.end(), T());

return result;

}

float average(std::vector<std::string> data) {

std::cout << "Invalid: Cannot average string type" << std::endl;

return 0.0;

}

float average(std::vector<int> data) {

float result = 1.0 \* sum(data) / data.size();

return result;

}

float average(std::vector<float> data) {

float result = 1.0 \* sum(data) / data.size();

return result;

}

template <class T>

T min(std::vector<T> data) {

T result = \*(std::min\_element(data.begin(), data.end()));

return result;

}

template <class T>

T max(std::vector<T> data) {

T result = \*(std::max\_element(data.begin(), data.end()));

return result;

}

std::vector<std::string> scan(std::vector<std::string> data, bool inclusive)

{

std::cout << "Invalid: Cannot scan strings" << std::endl;

return data;

}

std::vector<float> scan(std::vector<float> data, bool inclusive)

{

std::vector<float> results;

if (inclusive)

{

for (int i = 0; i < data.size(); i++)

{

if (i > 0)

{

results.push\_back(data[i] + results[i - 1]);

}

else

{

results.push\_back(data[i]);

}

}

}

else

{

std::vector<float> inclusiveResults = scan(data, true);

for (int i = 0; i < data.size(); i++)

{

results.push\_back(inclusiveResults[i] - data[i]);

}

}

return results;

}

std::vector<int> scan(std::vector<int> data, bool inclusive)

{

std::vector<int> results;

if (inclusive)

{

for (int i = 0; i < data.size(); i++)

{

if (i > 0)

{

results.push\_back(data[i] + results[i - 1]);

}

else

{

results.push\_back(data[i]);

}

}

}

else

{

std::vector<int> inclusiveResults = scan(data, true);

for (int i = 0; i < data.size(); i++)

{

results.push\_back(inclusiveResults[i] - data[i]);

}

}

return results;

}

template <class T>

std::vector<T> split(const std::string& s, char c, std::vector<T> v, int colOfInterest = -1) {

int i = 0;

int j = s.find(c);

std::vector<T> splitData;

while (j >= 0) {

//if (T == int)

//{

std::stringstream is(s.substr(i, j-i));

//}

T tempVal;

is >> tempVal;

splitData.push\_back(tempVal);

i = ++j;

j = s.find(c, j);

if (j < 0) {

//if (T == "int")

//{

std::stringstream is(s.substr(i, s.length()));

//}

T tempVal2;

is >> tempVal2;

splitData.push\_back(tempVal2);

}

}

std::vector<T> concatData = v;

if (colOfInterest == -1)

{

concatData.insert(concatData.end(), splitData.begin(), splitData.end());

}

else

{

concatData.push\_back(splitData[colOfInterest]);

}

return concatData;

}

template <class T>

std::vector<T> loadCSV(std::istream& in, std::vector<T> data, int colOfInterest, bool headersOnly = false) {

std::vector<T> loadedData = data;

std::string tmp;

if (headersOnly)

{

getline(in, tmp, '\n');

loadedData = split<T>(tmp, ',', loadedData, colOfInterest);

}

else

{

getline(in, tmp, '\n');

getline(in, tmp, '\n');

tmp.clear();

while (!in.eof()) {

getline(in, tmp, '\n');

loadedData = split<T>(tmp, ',', loadedData, colOfInterest);

tmp.clear();

}

}

return loadedData;

}

template <class T>

void printVector(std::vector<T> data)

{

for (int i = 0; i < data.size(); i++)

{

std::cout << i << "\t:" << data[i] << std::endl;

}

}

void printhelp ()

{

std::cout << "Syntax: data\_filepath function print column\_index [serial] [count\_value] [hard top threshold] [hard bottom threshold] [soft threshold]" << std::endl;

std::cout << "function: i (ingest only), s (sum), a (average), m (minimum), M (maximum), c (count of value), n (scan - exclusive), N (scan - inclusive), t (TEEN), d (DSSS)" << std::endl;

std::cout << "print: r (read only), p (print function result only), P (print data & result of function) s (serial)" << std::endl;

std::cout << "column\_index: 0-based" << std::endl;

std::cout << "count\_value (if applicable)" << std::endl;

std::cout << "Help Syntax: h" << std::endl;

}

template <class T>

int mainfunc(int argc, char\*\* argv) {

// Should use an actual arg parse library or something

// Only runs if filepath is given - maybe should check if filepath is valid

bool badsyntax = false;

std::ifstream in(argv[1]);

T sumValue = T();

float averageValue = 0.0;

T minValue = T();

T maxValue = T();

int countValue = 0;

std::vector<T> scanValues;

bool serial = false;

if (argc >= 6){

bool serial = (\*argv[5] == 's');

}

if (!in)

return(EXIT\_FAILURE);

std::vector<T> data;

std::vector<T>\* data\_ptr = &data;

data = loadCSV<T>(in, data, atoi(argv[4]));

if (\*argv[2] == 'i')

{

// do nothing else - just ingest data

}

// Sum of data

else if (\*argv[2] == 's')

{

if (serial){

if (typeid(T) != typeid(std::string))

{

sumValue = sum(data);

}

else

{

std::cout << "Invalid: Cannot sum string type" << std::endl;

}

}

else{

sumValue = parallel::reduce(data\_ptr);

}

}

// Average of data

else if (\*argv[2] == 'a')

{

if (serial){

if (typeid(T) != typeid(std::string))

{

averageValue = average(data);

}

}

else{

averageValue = parallel::average(&data);

}

}

// Min of data

else if (\*argv[2] == 'm')

{

if (serial){

minValue = min(data);

}

else{

minValue = parallel::minimum(&data);

}

}

// Max of data

else if (\*argv[2] == 'M')

{ if (serial){

maxValue = max(data);

}

else{

maxValue = parallel::maximum(&data);

}

}

// Count of data

else if (\*argv[2] == 'c' && argv[5] != "")

{

if(serial){

std::stringstream ss(argv[6]);

T countObject;

ss >> countObject;

countValue = std::count(data.begin(), data.end(), countObject);

}

else{

std::stringstream ss(argv[5]);

T countObject;

ss >> countObject;

countValue = parallel::count(&data, countObject);

}

}

else if (\*argv[2] == 'n')

{

if(serial){

scanValues = scan(data, false);

}

else{

parallel::exclusive\_scan(data);

scanValues = data;

}

}

else if (\*argv[2] == 'N')

{

if(serial){

scanValues = scan(data, true);

}

else{

parallel::inclusive\_scan(data);

scanValues = data;

}

}

else if (\*argv[2] == 't')

{

if (serial){

std::stringstream ss1(argv[6]);

T h\_top\_thres;

ss1 >> h\_top\_thres;

std::stringstream ss2(argv[7]);

T h\_bot\_thres;

ss2 >> h\_bot\_thres;

std::stringstream ss3(argv[8]);

T s\_thres;

ss3 >> s\_thres;

std::vector<T>\* temp = TEEN(data, h\_top\_thres, h\_bot\_thres, s\_thres);

data.clear();

for (int i = 0; i<temp->size();i++){

data.push\_back(temp->at(i));

}

}

else{

std::stringstream ss1(argv[5]);

T h\_top\_thres;

ss1 >> h\_top\_thres;

std::stringstream ss2(argv[6]);

T h\_bot\_thres;

ss2 >> h\_bot\_thres;

std::stringstream ss3(argv[7]);

T s\_thres;

ss3 >> s\_thres;

std::vector<T>\* temp = parallel::TEEN(&data, h\_top\_thres, h\_bot\_thres, s\_thres);

data.clear();

for (int i = 0; i<temp->size();i++){

data.push\_back(temp->at(i));

}

}

}

else if (\*argv[2] == 'd'){

if (typeid(T) == typeid(int)){

T pattern\_arr[data.size()];

T vals[data.size()];

if (serial) {

DSSS\_encrypt(&data, pattern\_arr, vals);

DSSS\_decrypt(&data, pattern\_arr, vals);

}

else{

parallel::DSSS\_encrypt(&data, pattern\_arr, vals);

parallel::DSSS\_decrypt(&data, pattern\_arr, vals);

}

}

}

else

{

badsyntax = true;

}

// Print results

if (\*argv[3] == 'P')

{

printVector<T>(data);

}

if (\*argv[3] == 'p' || \*argv[3] == 'P')

{

if (\*argv[2] == 's')

{

std::cout << "Sum: " << sumValue << std::endl;

}

else if (\*argv[2] == 'a')

{

std::cout << "Average: " << averageValue << std::endl;

}

else if (\*argv[2] == 'm')

{

std::cout << "Min: " << minValue << std::endl;

}

else if (\*argv[2] == 'M')

{

std::cout << "Max: " << maxValue << std::endl;

}

else if (\*argv[2] == 'c')

{

std::cout << "Count: " << countValue << std::endl;

}

else if (\*argv[2] == 'n')

{

std::cout << "Scan (exclusive)" << std::endl;

printVector<T>(scanValues);

}

else if (\*argv[2] == 'N')

{

std::cout << "Scan (inclusive)" << std::endl;

printVector<T>(scanValues);

}

else if (\*argv[2] == 't')

{

std::cout << "TEEN" << std::endl;

printVector<T>(data);

}

}

if (badsyntax)

{

printhelp();

}

return 0;

}

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

int result = -1;

if (argc == 2 && \*argv[1] == 'h')

{

printhelp();

return 0;

}

else if (argc >= 4)

{

std::vector<std::string> headerTypes;

std::vector<std::string> headerNames;

std::ifstream in(argv[1]);

if (!in)

return(EXIT\_FAILURE);

headerTypes = loadCSV<std::string>(in, headerTypes, -1, true);

headerNames = loadCSV<std::string>(in, headerNames, -1, true);

if (argc == 4)

{

if (\*argv[3] == 'P')

{

printVector<std::string>(headerTypes);

printVector<std::string>(headerNames);

}

}

else if (argc >= 5)

{

int columnIndex = atoi(argv[4]);

// std::stringstream is(\*argv[4]);

// int columnIndex;

// is >> columnIndex;

if (headerTypes[columnIndex].find("i") != std::string::npos)

{

result = mainfunc<int>(argc, argv);

}

else

{

printhelp();

}

}

}

else

{

printhelp();

return (EXIT\_FAILURE);

}

return 0;

}

//PARALLEL FILE

#include <vector>

#include <thrust/host\_vector.h>

#include <thrust/device\_vector.h>

#include <thrust/reduce.h>

#include <thrust/count.h>

#include <thrust/functional.h>

#include <thrust/sort.h>

#include <iostream>

#include <thrust/extrema.h>

#include <stdlib.h>

#include <time.h>

namespace parallel{

template <class T>

//Retrieves indicies where data from nodes should be forwarded

\_\_global\_\_ void TEEN\_indexes(T \*d\_vals, int \*d\_ind, T\* h\_thres\_top, T\* h\_thres\_bot, T\* s\_thres){

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

//get indexes of values we want

if ((d\_vals[i] > \*h\_thres\_top) || (d\_vals[i] < \*h\_thres\_bot)){

//if the values exceed the given thresholds, collect index

d\_ind[i]= 1;

}

else if (i > 0){

if (abs(d\_vals[i] - d\_vals[i-1]) > \*s\_thres){

//or if the values changed more than the soft threshold from the last value, collect index

d\_ind[i] = 1;

}

}

}

template <class T>

\_\_global\_\_ void compact(T\* d\_vals, int\* d\_ind, T\* d\_vals\_out){

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

if (((i == 0) && (d\_ind[i] == 1)) || ((i>0) && (d\_ind[i] > d\_ind[i-1]))){

d\_vals\_out[d\_ind[i]-1] = d\_vals[i];

}

}

template <class T>

std::vector<T>\* TEEN(std::vector<T>\* in, T h\_thres\_top, T h\_thres\_bot, T s\_thres){

//own algorithms

int size = in->size();

T arr[size];

std::copy(in->begin(),in->end(), arr);

int arr\_ind[size];

for (int i = 0; i < size;i++){

arr\_ind[i] = 0;

}

//device values

T \*d\_arr;

T \*d\_h\_thres\_top, \*d\_h\_thres\_bot, \*d\_s\_thres;

int \*d\_arr\_ind;

//allocate space for the values & copy memory there

cudaMalloc((void\*\*)&d\_arr, size\*sizeof(T));

cudaMalloc((void\*\*)&d\_arr\_ind, size\*sizeof(int));

cudaMalloc((void\*\*)&d\_h\_thres\_top, sizeof(int));

cudaMalloc((void\*\*)&d\_h\_thres\_bot, sizeof(int));

cudaMalloc((void\*\*)&d\_s\_thres, sizeof(int));

cudaMemcpy(d\_arr, arr, size \* sizeof(T),cudaMemcpyHostToDevice);

cudaMemcpy(d\_arr\_ind, arr\_ind, size \* sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(d\_h\_thres\_top, &h\_thres\_top, sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(d\_h\_thres\_bot, &h\_thres\_bot, sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(d\_s\_thres, &s\_thres, sizeof(int),cudaMemcpyHostToDevice);

//get indexes for data we want

TEEN\_indexes<T><<<(size+266)/256, 256>>>(d\_arr, d\_arr\_ind, d\_h\_thres\_top, d\_h\_thres\_bot, d\_s\_thres);

//we can now free some values

cudaFree(d\_h\_thres\_top);

cudaFree(d\_h\_thres\_bot);

cudaFree(d\_s\_thres);

//copy values into thrust device pointer

thrust::device\_ptr<int> d\_ptr = thrust::device\_malloc<int>(size);

thrust::copy(&d\_arr\_ind[0], &d\_arr\_ind[0]+size, d\_ptr);

thrust::device\_vector<int> d\_vec(d\_ptr,d\_ptr + size);

//run a sum for the value count

int sum\_ind = thrust::reduce(d\_vec.begin(),d\_vec.end(), 0, thrust::plus<int>());

//run an inclusive scan

thrust::inclusive\_scan(d\_vec.begin(),d\_vec.end(),d\_vec.begin());

//copy values back into original array

thrust::copy(d\_vec.begin(),d\_vec.end(),d\_arr\_ind);

//create compacted array

T comp\_array[sum\_ind];

T\* d\_comp\_array;

cudaMalloc((void\*\*)&d\_comp\_array, sum\_ind\*sizeof(T));

cudaMemset(&d\_comp\_array, 0, sum\_ind\*sizeof(T));

compact<T><<<(size+266)/256, 256>>>(d\_arr, d\_arr\_ind, d\_comp\_array);

cudaFree(d\_arr);

cudaFree(d\_arr\_ind);

cudaMemcpy(comp\_array, d\_comp\_array, sum\_ind \* sizeof(T),cudaMemcpyDeviceToHost);

cudaFree(d\_comp\_array);

std::vector<T> tmp(comp\_array, comp\_array + sum\_ind);

std::vector<T>\* ret = new std::vector<T>(tmp);

return ret;

}

template <class T>

\_\_global\_\_ void DSSS\_operation(T\* vals, T\* pattern){

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

//use bitwise operator to perform xor

vals[i] = vals[i] ^ pattern[i];

}

template <class T>

void DSSS\_encrypt(std::vector<T>\* in,T\* pattern\_arr, T\* vals){

int size = in->size();

//get random values for encryption

srand(time(NULL));

for(int i = 0; i< size;i++){

pattern\_arr[i] = rand();

}

//copy values, prepare memory

T \*d\_vals, \*d\_pattern;

std::copy(in->begin(),in->end(), vals);

cudaMalloc((void\*\*)&d\_vals, size\*sizeof(T));

cudaMalloc((void\*\*)&d\_pattern, size\*sizeof(T));

cudaMemcpy(d\_vals, vals, size\*sizeof(T),cudaMemcpyHostToDevice);

cudaMemcpy(d\_pattern, pattern\_arr, size\*sizeof(T),cudaMemcpyHostToDevice);

DSSS\_operation<T><<<(size+266)/256, 256>>>(d\_vals, d\_pattern);

cudaFree(d\_pattern);

cudaMemcpy(vals, d\_vals, size\*sizeof(T), cudaMemcpyDeviceToHost);

cudaFree(d\_vals);

}

template <class T>

void DSSS\_decrypt(std::vector<T>\* vals, T\* pattern\_arr, T\* in){

int size = vals->size();

T \*d\_vals, \*d\_pattern;

cudaMalloc((void\*\*)&d\_vals, size\*sizeof(T));

cudaMalloc((void\*\*)&d\_pattern, size\*sizeof(T));

cudaMemcpy(d\_vals, in, size\*sizeof(T),cudaMemcpyHostToDevice);

cudaMemcpy(d\_pattern, pattern\_arr, size\*sizeof(T),cudaMemcpyHostToDevice);

DSSS\_operation<T><<<(size+266)/256, 256>>>(d\_vals, d\_pattern);

cudaFree(d\_pattern);

cudaMemcpy(in, d\_vals, size\*sizeof(T), cudaMemcpyDeviceToHost);

cudaFree(d\_vals);

std::vector<T> temp (in, in + sizeof(in) / sizeof(in[0]));

vals = new std::vector<T>(temp);

}

template <class T>

int reduce(std::vector<T>\* in){

thrust::host\_vector<T> h\_vec = \*in;

thrust::device\_vector<T> d\_vec = h\_vec;

T sm = thrust::reduce(d\_vec.begin(),d\_vec.end(), 0, thrust::plus<T>());

return sm;

}

template <class T>

float average(std::vector<T>\* in){

T sm = reduce(in);

return sm/(static\_cast<float>(in->size()));

}

template <class T>

int count(std::vector<T>\* in, T &val){

thrust::host\_vector<T> h\_vec = \*in;

thrust::device\_vector<T> d\_vec = h\_vec;

return thrust::count(d\_vec.begin(),d\_vec.end(), val);

}

template <class T>

void inclusive\_scan(std::vector<T> &in){

thrust::host\_vector<T> h\_vec = in;

thrust::device\_vector<T> d\_vec = h\_vec;

thrust::inclusive\_scan(d\_vec.begin(),d\_vec.end(),d\_vec.begin());

h\_vec = d\_vec;

for (int i = 0;i < h\_vec.size(); i++){

in[i] = h\_vec[i];

}

}

template <class T>

void exclusive\_scan(std::vector<T> &in){

thrust::host\_vector<T> h\_vec = in;

thrust::device\_vector<T> d\_vec = h\_vec;

thrust::exclusive\_scan(d\_vec.begin(),d\_vec.end(),d\_vec.begin());

h\_vec = d\_vec;

for (int i = 0;i < h\_vec.size(); i++){

in[i] = h\_vec[i];

}

}

template <class T>

int minimum(std::vector<T>\* in){

thrust::host\_vector<T> h\_vec = \*in;

thrust::device\_vector<T> d\_vec = h\_vec;

return \*thrust::min\_element(d\_vec.begin(),d\_vec.end());

}

template <class T>

int maximum(std::vector<T>\* in){

thrust::host\_vector<T> h\_vec = \*in;

thrust::device\_vector<T> d\_vec = h\_vec;

return \*thrust::max\_element(d\_vec.begin(),d\_vec.end());

}

}

template <class T>

std::vector<T>\* TEEN (std::vector<T> &in, T h\_thres\_top, T h\_thres\_bot, T s\_thres){

std::vector<T>\* out = new std::vector<T>();

for (int i = 0;i < in.size();i++){

if ((in[i] > h\_thres\_top) || (in[i] < h\_thres\_bot)){

//if the values exceed the given thresholds, collect index

out->push\_back(in[i]);

}

else if (i > 0){

if (abs(in[i] - in[i-1]) > s\_thres){

//or if the values changed more than the soft threshold from the last value, collect index

out->push\_back(in[i]);

}

}

}

return out;

}

template <class T>

void DSSS\_encrypt(std::vector<T>\* in, T\* pattern\_arr, T\* vals){

int size = in->size();

//get random values for encryption

srand(time(NULL));

for(int i = 0; i< size;i++){

pattern\_arr[i] = rand();

}

for(int i = 0; i < size;i++){

vals[i] = (in->at(i) ^ pattern\_arr[i]);

}

}

template <class T>

void DSSS\_decrypt(std::vector<T>\* in, T\* pattern\_arr, T\* vals){

int size = in->size();

for(int i = 0; i < size;i++){

vals[i] = in->at(i) ^ pattern\_arr[i];

}

}

## Github Link:

https://github.uc.edu/enochisg/parallelproject

# Bibliography

1. Agrawal, Dharam Prakash, Zeng, Quin-An. *Introduction to Wireless and Mobile Systems, 4th ed.* CL Engineering. 2015.
2. Core Algorithm to Compact - Intro to Parallel Programming by Udacity. Online video clip. *YouTube.*
3. Thrust Documentation. URL: <http://thrust.github.io/doc/index.html>