

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**LYGIAGRETUSIS PROGRAMAVIMAS**

Individualus darbas

**Atliko:**

IFF 5/8 grupės stud.

Rokas Palionis

**KAUNAS, 2017**

**TURINYS**

[1. UŽDUOTIS 2](#_Toc470076436)

[1.1. UŽDUOTIES ANALIZĖ IR SPRENDIMO METODAS 2](#_Toc470076437)

[2. PROGRAMOS APRAŠYMAS 2](#_Toc470076438)

[2.1. PROGRAMOS PAGRINDINIS KODAS 4](#_Toc470076439)

[2.2. PROGRAMOS TESTAVIMAS 8](#_Toc470076440)

[2.3. PROGRAMOS INSTALIAVIMO BEI VYKDYMO INSTRUCKIJA 9](#_Toc470076441)

[2.4. VYKDYMO LAIKO KITIMO TYRIMAS 9](#_Toc470076442)

[3. IŠVADOS 10](#_Toc470076443)

[4. LITERATŪRA 10](#_Toc470076444)

1. UŽDUOTIS

Aš pasirinkau užduotį: ištirti pasirinkto algoritmo (Merge sort) greitaveikos pokytį, kai jis vykdomas ant vieno procesoriaus (vienos gijos) ir kai tam panaudojama daugiau gijų (CUDA branduolių). Šią užduotį pasirinkau realizuoti naudojant CUDA programavimo modelį (naudojantį C / C++ kalbas).

Panaudosiu Microsoft Visual Studio 2015 aplinką bei CUDA 8.0 runtime versiją. Visi laiko tyrimai bus atlikti ant GTX 1050 *(Pascal, 2GB GDDR5, 640 CUDA branduolių)* vaizdo plokštės.

* 1. UŽDUOTIES ANALIZĖ IR SPRENDIMO METODAS

Visų pirma reikia ištirti kaip pritaikyti Merge sort rikiavimo algoritmą lygiagretumui. Tai padaryti būtų galima keliais žingsniais.

* Pirmiausia reikėtų pradėti su dviem sąrašais (įvedimo ir laikinas masyvai).
* Tada apsibrėžti plotį (width), kuris su kiekviena iteracija dvigubėtų.
* Kol plotis būtų mažiau nei 2N, kiekvienai pločio dydžio duomenų daliai surikiuoti ją į laikiną sąrašą, tada sukeisti rodyklės tarp tų sąrašų (sutaupoma laiko kopijuojant tarp masyvų arba kuriant mažus naujus masyvus).
* Galiausiai visi duomenys būtų galutiniame sąraše ir sukeitus dar kartą įvedimo masyvą su laikiniu, laikinas turės surikiuotas reikšmes.

1. PROGRAMOS APRAŠYMAS

Programoje yra kelios funkcijos, pagrindinė funkcija vykdo visą procesą, pagalbinės reikalingos duomenų generavimui, bei pačio Merge sort rikiavimo algoritmo vykdymui.

Pagrindinė funkcija kuri pradeda vykdyti mergesort ant nustatyto kiekio gijų bei blokų:

/\*\*

\* Pradeda vykdyti MergeSort algoritmą: paruošia atmintį, perkopijuoja duomenis į GPU

\* paleidžia gpu\_mergesort rikiavimą kiekvienai duomenų porcijai

\*/

void mergesort(long\* data, long size, dim3 threadsPerBlock, dim3 blocksPerGrid) {

long\* D\_data;

long\* D\_swp;

dim3\* D\_threads;

dim3\* D\_blocks;

cudaMalloc((void\*\*)&D\_data, size \* sizeof(long));

cudaMalloc((void\*\*)&D\_swp, size \* sizeof(long));

cudaMemcpy(D\_data, data, size \* sizeof(long), cudaMemcpyHostToDevice);

cudaMalloc((void\*\*)&D\_threads, sizeof(dim3));

cudaMalloc((void\*\*)&D\_blocks, sizeof(dim3));

// Nukopijuojama gijų / blokų kiekio informacija į GPU

cudaMemcpy(D\_threads, &threadsPerBlock, sizeof(dim3), cudaMemcpyHostToDevice);

cudaMemcpy(D\_blocks, &blocksPerGrid, sizeof(dim3), cudaMemcpyHostToDevice);

// Masyvų sukeitimui naudojami papildomi masyvai (galiausiai į juos įrašomi surikiuoti duomenys)

long\* A = D\_data;

long\* B = D\_swp;

long nThreads = threadsPerBlock.x \* threadsPerBlock.y \* threadsPerBlock.z \*

blocksPerGrid.x \* blocksPerGrid.y \* blocksPerGrid.z;

//

// Slice up the list and give pieces of it to each thread, letting the pieces grow

// bigger and bigger until the whole list is sorted

//

// Padalija sąrašą kiekvienai gijai po tam tikrą kiekį duomenų, leidžiant

// dalims vis didėti, kol visas sąrašas yra surikiuojamas

for (int width = 2; width < (size << 1); width <<= 1) {

long slices = size / ((nThreads)\* width) + 1;

gpu\_mergesort << <blocksPerGrid, threadsPerBlock >> >(A, B, size, width, slices, D\_threads, D\_blocks);

// Switch the input / output arrays instead of copying them around

// Sukeičiai įvedimo / išvedimo masyvus vietoj to, kad juos nukopijuoti vienas tarp kito

A = A == D\_data ? D\_swp : D\_data;

B = B == D\_data ? D\_swp : D\_data;

}

// Iškopijuojamas sąrašas iš GPU į CPU atmintį

cudaMemcpy(data, A, size \* sizeof(long), cudaMemcpyDeviceToHost);

cudaFree(A);

cudaFree(B);

}

Kernel‘io (CUDA) gpu\_mergesort funkcija (pradeda vykdyti su duomenų dalimi Merge sort rikiavimą):

/\*\*

\* Pateiktai duomenų daliai vykdo MergeSort rikiavimą

\*/

\_\_global\_\_ void gpu\_mergesort(long\* source, long\* dest, long size, long width, long slices, dim3\* threads, dim3\* blocks) {

unsigned int idx = getIdx(threads, blocks);

long start = width\*idx\*slices,

middle,

end;

for (long slice = 0; slice < slices; slice++) {

if (start >= size)

break;

middle = min(start + (width >> 1), size);

end = min(start + width, size);

gpu\_bottomUpMerge(source, dest, start, middle, end);

start += width;

}

}

Papildoma funkcija kuri apskaičiuoja kiekvienos gijos id:

/\*\*

\* Suskaičiuoja dabartinės gijos id

\*/

\_\_device\_\_ unsigned int getIdx(dim3\* threads, dim3\* blocks) {

int x;

return threadIdx.x +

threadIdx.y \* (x = threads->x) +

threadIdx.z \* (x \*= threads->y) +

blockIdx.x \* (x \*= threads->z) +

blockIdx.y \* (x \*= blocks->z) +

blockIdx.z \* (x \*= blocks->y);

}

Paskutinė funkcija, kuri sujungia (apjungia) elementus juos surikiuodama:

/\*\*

\* Sujungia duomenis, kiekvienai duomenų porcijai (daliai)

\*/

\_\_device\_\_ void gpu\_bottomUpMerge(long\* source, long\* dest, long start, long middle, long end) {

long i = start;

long j = middle;

for (long k = start; k < end; k++) {

if (i < middle && (j >= end || source[i] < source[j])) {

dest[k] = source[i];

i++;

}

else {

dest[k] = source[j];

j++;

}

}

}

* 1. PROGRAMOS PAGRINDINIS KODAS

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <fstream>

#include <iostream>

#include <string>

#include <time.h>

/\*\*

\* Rokas Palionis IFF-5/8 individualus darbas

\*

\* Šiame individualiame darbe panaudojant CUDA bus ištirtas lygiagretus MergeSort algortimas.

\*

\* Duomenys yra paimami atsitiktinai sugeneruoti arba įrašyti į failą atsitiktiniai duomenys.

\* Testavimas atliekamas su keliais duomenų rinkiniais ir skirtingais jų dydžiais.

\*

\* Surikiuoti rezultatai yra išspausdinami į failą.

\*

\* Ištyrus algoritmą, matyti, kad daugiausia laiko skiriama darbui su atmintimi, pats rikiavimas

\* įvykdomas gan greitai. Todėl, šis algoritmas naudos prieš nuoseklūjį algoritmo variantą turės

\* tik tada, kai bus daug duomenų kuriuos reikės surikiuoti.

\*/

// Apibrėžia min(a, b) funkciją, a - pirmas elementas, b - antras elementas

#define min(a, b) (a < b ? a : b)

/\*\*

\* Sujungia duomenis, kiekvienai duomenų porcijai (daliai)

\*/

\_\_device\_\_ void gpu\_bottomUpMerge(long\* source, long\* dest, long start, long middle, long end) {

long i = start;

long j = middle;

for (long k = start; k < end; k++) {

if (i < middle && (j >= end || source[i] < source[j])) {

dest[k] = source[i];

i++;

}

else {

dest[k] = source[j];

j++;

}

}

}

/\*\*

\* Suskaičiuoja dabartinės gijos id

\*/

\_\_device\_\_ unsigned int getIdx(dim3\* threads, dim3\* blocks) {

int x;

return threadIdx.x +

threadIdx.y \* (x = threads->x) +

threadIdx.z \* (x \*= threads->y) +

blockIdx.x \* (x \*= threads->z) +

blockIdx.y \* (x \*= blocks->z) +

blockIdx.z \* (x \*= blocks->y);

}

/\*\*

\* Pateiktai duomenų daliai vykdo MergeSort rikiavimą

\*/

\_\_global\_\_ void gpu\_mergesort(long\* source, long\* dest, long size, long width, long slices, dim3\* threads, dim3\* blocks) {

unsigned int idx = getIdx(threads, blocks);

long start = width\*idx\*slices,

middle,

end;

for (long slice = 0; slice < slices; slice++) {

if (start >= size)

break;

middle = min(start + (width >> 1), size);

end = min(start + width, size);

gpu\_bottomUpMerge(source, dest, start, middle, end);

start += width;

}

}

/\*\*

\* Pradeda vykdyti MergeSort algoritmą: paruošia atmintį, perkopijuoja duomenis į GPU

\* paleidžia gpu\_mergesort rikiavimą kiekvienai duomenų porcijai

\*/

void mergesort(long\* data, long size, dim3 threadsPerBlock, dim3 blocksPerGrid) {

// Sukuriami du masyvai

// Vykdymo metu jie yra keičiami vienas su kitu

long\* D\_data;

long\* D\_swp;

// Gijų / blokų kiekio informacija

dim3\* D\_threads;

dim3\* D\_blocks;

// Išskiriama atmintis masyvams ant GPU

cudaMalloc((void\*\*)&D\_data, size \* sizeof(long));

cudaMalloc((void\*\*)&D\_swp, size \* sizeof(long));

// Kopijuoja duomenis iš duomenų sąrašo į GPU atmintį

cudaMemcpy(D\_data, data, size \* sizeof(long), cudaMemcpyHostToDevice);

// Išskiriama atmintis gijų / blokų kiekio informacijai ant GPU

cudaMalloc((void\*\*)&D\_threads, sizeof(dim3));

cudaMalloc((void\*\*)&D\_blocks, sizeof(dim3));

// Nukopijuojama gijų / blokų kiekio informacija į GPU

cudaMemcpy(D\_threads, &threadsPerBlock, sizeof(dim3), cudaMemcpyHostToDevice);

cudaMemcpy(D\_blocks, &blocksPerGrid, sizeof(dim3), cudaMemcpyHostToDevice);

// Masyvų sukeitimui naudojami papildomi masyvai (galiausiai į juos įrašomi surikiuoti duomenys)

long\* A = D\_data;

long\* B = D\_swp;

// Paskaičiuoja bendrą gijų kiekį x\*y\*z\*X\*Y\*Z

long nThreads = threadsPerBlock.x \* threadsPerBlock.y \* threadsPerBlock.z \*

blocksPerGrid.x \* blocksPerGrid.y \* blocksPerGrid.z;

//

// Slice up the list and give pieces of it to each thread, letting the pieces grow

// bigger and bigger until the whole list is sorted

//

// Padalija sąrašą kiekvienai gijai po tam tikrą kiekį duomenų, leidžiant

// dalims vis didėti, kol visas sąrašas yra surikiuojamas

for (int width = 2; width < (size << 1); width <<= 1) {

long slices = size / ((nThreads)\* width) + 1;

// Iškviečia GPU funkciją ir vykdo MergeSort duomenų daliai (slices)

gpu\_mergesort << <blocksPerGrid, threadsPerBlock >> >(A, B, size, width, slices, D\_threads, D\_blocks);

// Switch the input / output arrays instead of copying them around

// Sukeičiai įvedimo / išvedimo masyvus vietoj to, kad juos nukopijuoti vienas tarp kito

A = A == D\_data ? D\_swp : D\_data;

B = B == D\_data ? D\_swp : D\_data;

}

// Iškopijuojamas sąrašas iš GPU į CPU atmintį

cudaMemcpy(data, A, size \* sizeof(long), cudaMemcpyDeviceToHost);

// Išlaisvinama paskirta GPU atmintis

cudaFree(A);

cudaFree(B);

}

/\*

\* Papildoma funkcija sugeneruojanti duomenų masyvą

\*/

void generateData(long array[], int length) {

for (long i = 0; i < length; i++)

array[i] = rand();

}

/\*

\* Papildoma funkcija duomenų įvedimui į masyvą

\*/

void readData(long data[], int size, std::ifstream & readStream) {

for (int i = 0; readStream.good() && i < size; i++) {

readStream >> data[i];

}

return;

}

/\*

\* Papildoma funkcija duomenų išvedimui iš masyvo į failą

\*/

void writeData(long data[], int size, std::ofstream & writeStream) {

for (int i = 0; writeStream.good() && i < size; i++) {

writeStream << data[i] << std::endl;

}

return;

}

int main() {

//Nustato lokalizaciją, kad tinkamai būtų nuskaitomi lietuviški rašmenys

setlocale(LC\_ALL, "");

// Gijų bei blokų skaičius

dim3 threadsPerBlock;

dim3 blocksPerGrid;

threadsPerBlock.x = 32;

threadsPerBlock.y = 1;

threadsPerBlock.z = 1;

blocksPerGrid.x = 8;

blocksPerGrid.y = 1;

blocksPerGrid.z = 1;

// Duomenų kiekis

int size = 125000;

// Kintamieji saugantys pradžios bei pabaigos laikus

clock\_t startTime, endTime;

// Duomenų masyvas

long \*data = new long[size];

std::cout << "Įveskite duomenų kiekį (n > 0), gijų kiekį (k >= 1), blokų kiekį (l >= 1): ";

std::cin >> size >> threadsPerBlock.x >> blocksPerGrid.x;

// Jei elementų dydis yra 0 ar mažiau

if (size < 1) {

std::cout << "\n\tElementų kiekis turi būti didesnis už 0!\n\nPrograma nesėkmingai baigė darbą!\n";

return -1;

}

//Skaitymo failas

std::string readFileName = "data/" + std::to\_string(size) + ".data.txt";

//Spausdinimo failas

std::string writeFileName = "results/" + std::to\_string(size) + ".result.txt";

// Įvedimo srautas

std::ifstream readStream(readFileName);

// Tikrinama ar failas egzistuoja

if (readStream.good()) {

// Nuskaitomi duomenys iš failo

readData(data, size, readStream);

readStream.close();

}

else {

// Sugeneruojami duomenys

generateData(data, size);

// Išspausdinamas sugeneruotas masyvas į failą

std::ofstream writeStreamData(readFileName);

writeData(data, size, writeStreamData);

writeStreamData.close();

}

// Vykdomas MergeSort algoritmas

startTime = clock();

mergesort(data, size, threadsPerBlock, blocksPerGrid);

endTime = clock();

std::cout << "\n\tVykdymo laikas (merge sort): " << endTime - startTime << " ms\n";

// išvedimo srautas

std::ofstream writeStream(writeFileName);

// Išspausdinamas surikiuotas masyvas į failą

writeData(data, size, writeStream);

// Uždaromas išvedimo srautas srautai

writeStream.close();

std::cout << "\nPrograma sėkmingai baigė darbą!\n";

return 0;

}

* 1. PROGRAMOS TESTAVIMAS

Programa buvo testuojama su įvairiais duomenų kiekiais bei skirtingais gijų bei blokų kiekiais. Tai reikalinga tam, kad būtų įsitikinta, jog algoritmas veikia tinkama ir efektyviai. Taip pat, jog pavyko algoritmą padaryti lygiagrečiu (rikiavimą Merge sort algoritmu).

* 1. PROGRAMOS INSTALIAVIMO BEI VYKDYMO INSTRUCKIJA

Programa buvo vykdoma tiesiog ją paleidžiant, suvedant duomenų kiekį, gijų skaičių bei blokų skaičių. Tada jeigu egzistuoja jau sugeneruotas *n* dydžio failas, tai iš jo nuskaito duomenis ir atlieka Merge sort. Kitu atveju sugeneruoja naują *n* dydžio failą ir atlieka rikiavimą.

* 1. VYKDYMO LAIKO KITIMO TYRIMAS

Algoritmas buvo tirtas su skirtingais kiekis gijų ir blokų bei įvairiais duomenų dydžiais. Gauti rezultatai yra pavaizduoti lentelėse bei grafikuose. Iš jų matyti, kad esant didesniam gijų / blokų skaičiui, programa yra vykdoma sparčiau nei tik ant vienos gijos (vykdant nuosekliai).

*1 lentelė 31250 duomenų kiekis*

|  |  |
| --- | --- |
| Procesų skaičius (gijos \* blokai) | Laikas, *ms* |
| 1 \* 1 | 517 |
| 2 \* 1 | 321 |
| 2 \* 2 | 196 |
| 4 \* 2 | 140 |
| 4 \* 4 | 107 |
| 8 \* 4 | 93 |
| 8 \* 8 | 86 |

*1 pav. Grafikas pagal 31250 duomenų rinkinį*

*2 lentelė 125000 duomenų kiekis*

|  |  |
| --- | --- |
| Procesų skaičius (gijos \* blokai) | Laikas, *ms* |
| 1 \* 1 | 2152 |
| 2 \* 1 | 1406 |
| 2 \* 2 | 816 |
| 4 \* 2 | 574 |
| 4 \* 4 | 432 |
| 8 \* 4 | 367 |
| 8 \* 8 | 339 |

*2 pav. Grafikas pagal 125000 duomenų rinkinį*

*3 lentelė 500000 duomenų kiekis*

|  |  |
| --- | --- |
| Procesų skaičius (gijos \* blokai) | Laikas, *ms* |
| 1 \* 1 | 2112 |
| 2 \* 1 | 2252 |
| 2 \* 2 | 2122 |
| 4 \* 2 | 2100 |
| 4 \* 4 | 1822 |
| 8 \* 4 | 1511 |
| 8 \* 8 | 1388 |

*3 pav. Grafikas pagal 500000 duomenų rinkinį*

1. IŠVADOS

Pasirinktą algoritmą pavyko panaudoti lygiagrečiam programavimui. Algoritmas veikia sparčiau kai duomenys yra kuo daugiau išskaidomi ant didesnio gijų skaičiaus, tačiau nemažą dalį laiko užima duomenų kopijavimas tarp CPU ir GPU. Todėl norint gauti naudos iš lygiagretaus Merge sort, reikėtų dirbti su dideliais kiekiais duomenų ir dideliu gijų skaičiumi.

1. LITERATŪRA

*Merge sort (sorting algorithm)*:

[***https://en.wikipedia.org/wiki/Merge\_sort***](https://en.wikipedia.org/wiki/Merge_sort)

*CUDA toolkit (documentation)*:

[***https://docs.nvidia.com/cuda/cuda-c-programming-guide***](https://docs.nvidia.com/cuda/cuda-c-programming-guide)