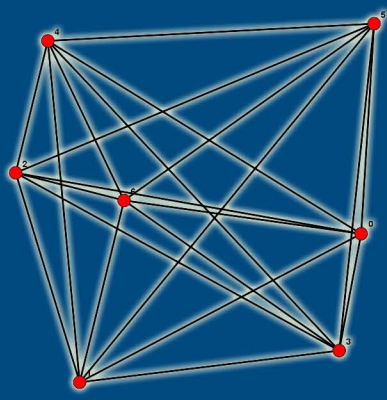
UNIUNEA EUROPEANĂ
GUVERNUL ROMÂNIEI MINISTERUL MUNCII, FAMILIEI ŞI PROTECŢIEI SOCIALE AMPOSDRU
Fondul Social European POSDRU 2007-2013
Instrumente Structurale 2007-2013
MINISTERUL EDUCAŢIEI CERCETĂRII TINERETULUI ŞI SPORTULUI
OIPOSDRU
UNIVERSITATEA POLITEHNICA DIN BUCUREŞTI

Proiect cofinanţat din Fondul Social European prin Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007-2013
Investeşte în oameni!

Răzvan ANDONIE Angel CAȚARON Zoltán GÁSPÁR Honorius GÂLMEANU
Mihai IVANOVICI István LŐRENTZ Lucian SASU

ALGORITMI ȘI STRUCTURI DE DATE

Aplicații în Imagistică și Bioinformatică



Editura Universității *TRANSILVANIA* din Brașov

Adresa: 500091 Braşov,
B-dul Iuliu Maniu 41A
Tel:0268 – 476050
Fax: 0268 476051
E-mail : editura@unitbv.ro

Tipărit la:
Tipografia Universității *Transilvania* din Braşov
B-dul Iuliu Maniu 41A
Tel: 0268 – 476050

Editură acreditată de CNCSIS
Adresa nr.1615 din 29 mai 2002

Referenți științifici: **Prof. univ. dr. ing. Gheorghe TOACȘE**
Prof. univ. dr. ing. Vasile BUZULOIU

Descrierea CIP a Bibliotecii Naționale a României
Algoritmi şi structuri de date : aplicații în imagistică şi bioinformatică /
Răzvan Andonie, Angel Cațaron, Zoltan Gáspár, ... - Braşov : Editura Universității "Transilvania", 2012
Bibliogr.
ISBN 978-606-19-0058-9
I. Andonie, Răzvan
II. Cațaron, Angel
III. Zoltan, Gáspár
004.421(075.8)
510.5(075.8)
004.422.63(075.8)

Răzvan Andonie Angel Caţaron Zoltán Gáspár Honorius Gâlmeanu

Mihai Ivanovici István Lőrentz Lucian Sasu

ALGORITMI ŞI STRUCTURI DE DATE

APLICAŢII ÎN IMAGISTICĂ ŞI BIOINFORMATICĂ

Editura Universităţii *TRANSILVANIA* din Braşov

2012

Cuprins

Prefaţă

Lucrarea de faţă a fost realizată în perioada septembrie 2010 - martie 2012, în cadrul proiectului POS-DRU/86/1.2/S/61756 intitulat *Tehnici de Analiză, Modelare şi Simulare pentru Imagistică, Bioinformatică şi Sisteme Complexe (ITEMS)*. Colectivul ITEMS de la Universitatea Transilvania din Braşov a răspuns de conceperea şi predarea cursului de *Algoritmi şi Structuri de Date* pentru programul de Masterat de Excelenţă organizat sub umbrela acestui proiect la Universitatea Politehnica din Bucureşti. Prezentul volum conţine lucrările de laborator ale acestui curs. Chiar dacă a fost conceput pentru masteranzii ITEMS, suntem convinşi că lucrarea poate fi utilă multor studenţi şi specialişti din domeniul Calculatoare şi Tehnologia Informaţiei.

Încă de la început, ne-am propus să concepem cursul altfel decât este el predat la majoritatea universităţilor. Structurile de date şi algoritmii sunt desigur standard, deoarece este practic imposibilă o schimbare radicală a acestui domeniu. Totuşi, ne-am concentrat mai ales pe tehnici specializate, mai rar prezentate. Şi atunci ce este de fapt nou, în afară de aplicaţiile alese şi implementate de noi?

Nouă este integrarea a trei concepte. Primul dintre acestea este filozofia noastră de predare, care se bazează pe structuri de date generice orientate pe obiect. Am folosit Java, C++ şi Python, toate limbaje care permit o astfel de abordare, pentru a sublinia că nu contează limbajul de programare, ci modul de abordare. C++ este alegerea naturală pentru programarea paralelă în CUDA. Python este frecvent utilizat pentru programare *ad-hoc* sau prototipare rapidă, fiind răspândit mai ales în grupurile de cercetare. În sfârşit, Java se impune prin portabilitate, fiind totodată, alături de C/C++, un limbaj de referinţă. Al doilea concept este că viitorul aparţine programării paralele. Din această cauză, o parte din lucrările de laborator sunt implementate în CUDA, pe plăci grafice cu arhitectură masiv paralelă. Al treilea concept este un accent mare pus pe teoria complexităţii, atât în cazul secvenţial, cât şi în cazul paralel. Analiza complexităţii algoritmilor folosiţi este un pas important, chiar şi atunci când ne referim la metodele definite în containere generice.

Lucrările de laborator prezentate au fost selectate pentru a ilustra aplicaţii în imagistică şi bioinformatică. Fişierele şi programele aferente pot fi accesate prin Internet, la adresa: `http://miv.unitbv.ro/asd`.

Echipa de autori este foarte omogenă. Ne cunoaştem de ani buni şi am realizat multe lucrări de cercetare împreună. Practic, putem vorbi despre o şcoală de algoritmi la Braşov. O parte dintre noi suntem cadre didactice universitare, o alta lucrăm în firme de software. Pentru noi, perioada elaborării acestui material a fost extrem de creativă. Ceea ce a determinat acest lucru a fost în special lucrul în echipă. Este motivul pentru care autorii sunt listaţi alfabetic, fiecare cu contribuţii egale, greu de separat.

Braşov, martie 2012.

Autorii

Capitolul 1

Structuri de date generice

În cadrul acestui capitol ne propunem să vă prezentăm un mod de gândire orientat pe obiecte (Object Oriented Programming - OOP) pentru structurarea datelor. Ne dorim ca cititorul să aibă o cât mai mare flexibilitate în ceea ce privește utilizarea limbajului de programare, motiv pentru care vom prezenta modul de implementare a structurilor de date în două dintre cele mai răspândite limbaje: Java şi C++.

1.1 Breviar teoretic

Printre cele mai utilizate structuri de date sunt listele, în special două tipuri de liste: stiva şi coada. De asemenea, vom prezenta și o structură de stocare asociativă, numită *map*.

O *listă* reprezintă o secvență de zero (lista vidă) sau mai multe elemente de un anumit tip:

$$a_1, a_2, ..., a_n$$

unde $n \geq 0$ şi este numit lungimea listei (n=0 însemnând listă vidă), iar a_i este elementul din listă de de poziţia i ($1 \leq i \leq n$). Cele mai importante operaţii cu o listă sunt:

- *INSERT(x, p, L)*. Adaugă în lista L elementul x la poziţia p, deplasând la dreapta toate elementele care se aflau pe poziţiile $p, ..., n$.
- *LOCATE(x, L)*. Returnează poziţia elementului x în lista L. Dacă x apare de mai multe ori, atunci poziţia primei apariții este returnată.
- *RETRIEVE(p, L)*. Returnează elementul aflat pe poziţia p în lista L.
- *DELETE(p, L)*. Șterge elementul de pe poziţia p din lista L, iar elementele de pe pozițiile $p + 1, ..., n$ sunt mutate cu o poziție la stânga.
- *NEXT(p, L)* şi *PREVIOUS(p, L)*. Returnează poziţia următoare, respectiv anterioară din lista L.

Implementarea listelor se poate face folosind un şir de elemente (array) sau cu pointeri la elementul următor (şi anterior) numite liste simplu (sau dublu) înlănțuite (vezi Figura 1.1). În cadru implementării, folosind un şir, elementele sunt plasate în celule succesive din şir. Folosind această reprezentare, lista poate fi parcursă cu ușurință, iar elemente nou pot fi adăugate rapid la capătul listei. Adăugarea (ștergerea) de elemente în (din) interiorul listei la poziţia p necesită o deplasare a tuturor elementelor de pe pozițiile $p+1, ..., n$.

În cadrul implementării folosind referințe (pointeri), fiecare element are şi o referință către elementul următor din listă. Folosind această implementare, elementele nu mai trebuie să fie plasate într-o zonă continuă de memorie, dar dezavantajul este memoria adițională folosită pentru referinţe. Adăugarea şi ştergerea de elemente din listă sunt operații rapide oricare ar fi poziţia pe care trebuie adăugat (șters) elementul.

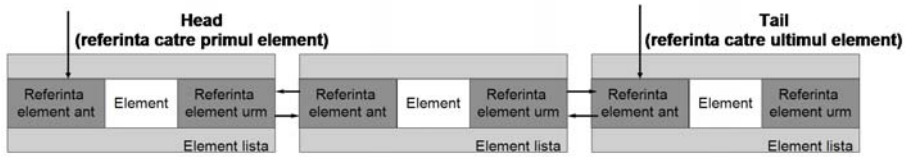


Figura 1.1: Exemplu de listă dublu înlănţuită.

Ambele implementări prezintă avantaje şi dezavantaje, iar alegerea tipului implementării trebuie să fie în funcție de operațiile care se fac cu această listă. Aspectele principale de care trebuie să ținem seamă sunt:

- Implementarea folosind un şir de elemente necesită specificarea lungimii maxime a listei. Dacă nu se poate determina o limită superioară a mărimii listei este de preferat o implementare folosind referinţe;
- Anumite operații necesită mai mult timp în cadrul unei implementări decât în cadrul celeilalte. Ca exemplu, operația de adăugare şi ștergere necesită un număr constant de operații pentru o implementare cu pointeri şi un număr de operaţii proporțional cu poziţia la care se adaugă (șterge) un element în cadrul implementării cu şir. Pe de altă parte, operaţia *RETRIEVE* necesită un număr constant de operații la implementarea cu şir şi un număr de operaţii proporțional cu poziţia elementului în cadrul implementării folosind referinţe;
- Implementarea cu şir nu folosește memoria în mod eficient. Pentru o listă întotdeauna se alocă spațiu pentru numărul maxim de elemente, indiferent de numărul de elemente care sunt folosite la un anumit moment dat. Implementarea folosind referinţe mai are nevoie de memorie pentru salvarea referinței, motiv pentru care mărimea unui element este mai mare.

O *stivă* este un tip special de listă în cadrul căreia toate operaţiile de adăugare şi ştergere au loc doar la capătul listei. Stiva este o structură de date

de tip LIFO (Last In First Out). Operaţiile de regăsire, adăugare şi ştergere au următoarele nume consacrate:

- *TOP(S).* Returnează elementul din capătul stivei S;
- *PUSH(x, S).* Adaugă elementul x la capătul stivei S;
- *POP(S).* Șterge elementul de la capătul stivei S.

Pentru implementarea stivei se pot folosi șiruri sau liste înlănțuite. Pentru implementarea folosind un şir de elemente elementul din capătul stivei va fi ultimul element din şir având indexul cel mai mare.

Coada este un alt tip special de listă în care operaţiile de adăugare şi ştergere au loc la capete diferite ale listei. Coada este o structură de date de tip FIFO (First In First Out). Principalele operaţii în cadrul cozilor au următoarele nume:

- *FRONT(Q).* Returnează primul element din coada Q;
- *ENQUEUE(x, Q).* Adaugă elementul x la capătul cozii Q;
- *DEQUEUE(S).* Șterge primul element din coada Q.

Ca în cazul stivei, ambele tipuri de implementări (cu şiruri sau cu pointeri) pot fi folosite pentru cozi. Pentru implementarea cu pointeri este avantajos să se menționă şi un pointer care arată la capătul cozii pentru o adăugare rapidă, în afară de pointerul care indică începutul cozii. Pentru implementarea folosind şiruri este avantajos să privim şirul ca unul circular (după ultimul element urmează primul) şi să avem doi indecși pentru primul şi ultimul element din coadă.

O *mulțime* (set) este un caz particular al unei succesiuni de elemente, având proprietatea ca nu există elemente duplicate.

Structura de stocare asociativă (map) reprezintă o funcție de asociere (M) de la elemente având un anumit tip (r) la elemente care au (posibil) un alt tip de date (d).

$$M(d) = r$$

Un exemplu de regulă de asociere, când ambele elemente au tipul întreg, este funcţia pătrat.$square(i) = i^2$. Pentru majoritatea asocierilor, însă, nu există modalități convenabile pentru salvarea acestei reguli de asociere. Să luăm un exemplu din contabilitate în care fiecărei persoane angajate i se asociază un salariu. În acest caz asocierea se face între tipul de date string şi tipul integer (sau float). Pentru astfel de situații se poate folosi structura de date asociativă (map).

Implementarea structurii map poate fi realizată folosind şiruri în cazul în care tipul de date r este unul primitiv (de exemplu, număr întreg) sau liste înlănţuite, iar implementările actuale folosesc funcţii de dispersie.

1.2 Java Collections Framework

Java Collections Framework (JCF) este o colecţie de clase şi interfeţe care implementează în limbajul Java cele mai folosite structuri de date. *Standard Template Library* este o bibliotecă C++ de algoritmi generici şi structuri de date.

"Framework" înseamnă în contextul programării o serie de clase, biblioteci de funcţii care joacă rol de "schelet" într-o aplicaţie, permiţând extinderea şi dezvoltarea ei pe baza acestor elemente. În Java, acest context se numeşte "Java Collections Framework" fiind similar ca funcţionalitate cu Standard Template Library (STL) din C++. O scurtă introducere despre JCF se găseşte la adresa http://java.sun.com/developer/onlineTraining/collections/Collection.html.

În Figura 1.2 sunt prezentate clasele care implementează principalele interfeţe din Java Collections Framework. Fiecare obiect poate fi parcurs folosind un iterator.

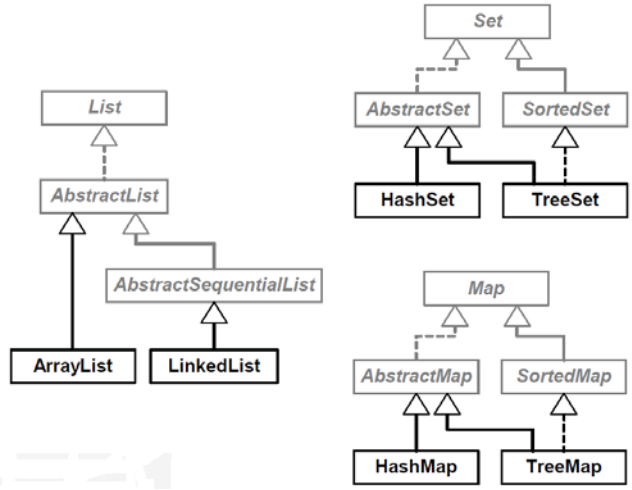


Figura 1.2: Ierarhia principalelor clase din Java Collections Framework.

1.2.1 Liste în Java: clasa *ArrayList*

Codul următor este un exemplu pentru crearea unei clase numită Person, crearea unei liste având obiecte de tip Person şi parcurgerea listei. Copiați următorul cod într-un fișier numit Person.java.

```
import java.util.*;

public class Person {
  private String firstName;
  private String lastName;
```

```
  private int age;
  private String job;

  public Person(String firstName, String lastName, int age, String job) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.job = job;
  }

  public static void main(String[] args) {
    Person alex1 = new Person("Alex", "Daicu", 24, "programator");

    List<Person> people = new ArrayList<Person>();
    people.add(new Person("Dan", "Popovici", 23, "student"));
    people.add(new Person("Ion", "Daicu", 13, "elev"));
    people.add(new Person("Radu", "Corlatescu", 33, "lector"));
    people.add(new Person("Daniela", "Popovici", 24, "student"));
    people.add(new Person("Daicu", "Alex", 24, "programator"));
    people.add(new Person("Dan", "Coradu", 21, "muncitor"));
    people.add(new Person("Raluca", "Balan", 34, "lector"));
    people.add(new Person("Alex", "Daicu", 24, "programator"));
    people.add(new Person("Alex", "Daicu", 25, "programator"));

    // exemplu de parcurgere a unei colectii
    Iterator<Person> i = people.iterator();
    people.iterator();
    while (i.hasNext()){
      Person element = i.next();
      System.out.println(element);
    }
  }
}
```

Exerciţii

1.1 Rulați exemplul de mai sus.

1.2 Implementați o metodă *toString()* membră a clasei *Person* care să întoarcă datele unei persoane sub forma unui şir de caractere. Apelaţi apoi aceeași metodă înlocuind instrucţiunea *System.out.println(element)* cu următoarea instrucțiune: *System.out.println(element.toString())*.

1.3 Scrieţi o funcţie care să caute şi să afişeze câte persoane sunt programator şi câte au vârsta sub 30 de ani.

1.4 Implementaţi o metodă *equals(Person p)* membră a clasei *Person*. Această funcţie returnează *true* dacă două persoane au acelaşi nume şi *false* în caz contrar. În cazul implementării corecte, *System.out.println(people.contains(alex1))* va afişa *true* deoarece metoda *contains* apelează automat metoda *equals*.

1.2.2 Mulţimi în Java: clasele *HashSet* şi *TreeSet*

Clasa *HashSet* implementează interfaţa *Set* şi păstrează o colecţie de elemente neordonate:

```
Set<Integer> set = new HashSet<Integer>();
set.add(-1);
set.add(2);
set.add(1);
//multimea contine 1 2 -1

set.add(2);
//nu se adauga din nou elementul 2
//multimea contine 1 2 -1

set.remove(2);
//multimea contine 1 -1
```

Clasa *HashSet* foloseşte metoda *hashCode()* pentru a determina dacă două elemente sunt identice şi nu garantează ordinea în care sunt redate elementele la o parcurgere a mulţimii.

Clasa *TreeSet* este un exemplu de implementare a unei mulţimi ordonate, iar codul de mai jos prezintă funcţionarea acestei clase pe o mulţime de trei valori întregi.

```
Set<Integer> set = new TreeSet<Integer>();
set.add(-1);
set.add(2);
set.add(1);
//multimea contine -1 1 2

set.add(2);
//nu se adauga din nou elementul 2
//multimea contine  -1 1 2

set.remove(2);
//multimea contine -1 1
```

Exerciţii

1.5 Adăugaţi elementele din listă într-un *HashSet* după care afişaţi conţinutul mulţimii. De câte ori apare persoana Alex Daicu?

1.6 În cazul în care Alex Daicu apare de mai multe ori (indiferent dacă pe prima poziţie apare numele sau prenumele), modificați funcţia *hashCode* astfel încât persoana respectivă să apară o singură dată.

1.7 Implementaţi metoda *compare(Person p1, Person p2)* în clasa Person care să returneze un număr negativ dacă p1<p2, 0 dacă p1=p2 şi un număr pozitiv dacă p1>p2.

1.8 Adăugaţi elementele din listă într-un *TreeSet* după care afişaţi conţinutul mulţimii. În cazul implementării corecte elementele trebuie să fie ordonate crescător.

1.9 Modificaţi funcţia *compare* astfel încât persoanele să fie ordonate descrescător.

1.10 Citiţi fişierul *sample.txt* linie cu linie şi

- Afişaţi numărul total de cuvinte;
- Afişaţi numărul total de cuvinte distincte;
- Realizaţi o histogramă a frecvenţei de apariţie a cuvintelor (de câte ori apare fiecare cuvânt).

1.3 C++ Standard Template Library

Standard Template Library, pe scurt STL, este o bibliotecă C++ de asa numite clase container, algoritmi şi iteratori, care pune la dispoziția programatorului majoritatea algoritmilor fundamentali (sortare, căutare etc.) şi a structurilor de date folosite în ştiinţa calculatoarelor. STL este o bibliotecă generică, însemnând faptul că are componente puternic parametrizate, aproape fiecare componentă STL fiind un șablon (template).

Conceptele folosite în STL pot fi grupate după cum urmează, iar legăturile dintre acestea sunt prezentate în Figura 1.3:

- Clase *containter*
  - Secvenţe (*vector, list*)
  - Containere asociative (*set, map*)
  - Adaptoare de container (*stack, queue, priority_queue*)
  - String (*string, rope*)
- Operaţii/Utilitare
  - Iteratori. Clasă care arată poziţia elementelor într-un container
  - Algoritmi. Rutine ca *find, count, sort* sau *search*

– auto_ptr. Clasă care gestionează pointeri şi evită irosirea spațiului de memorie (memory leak).

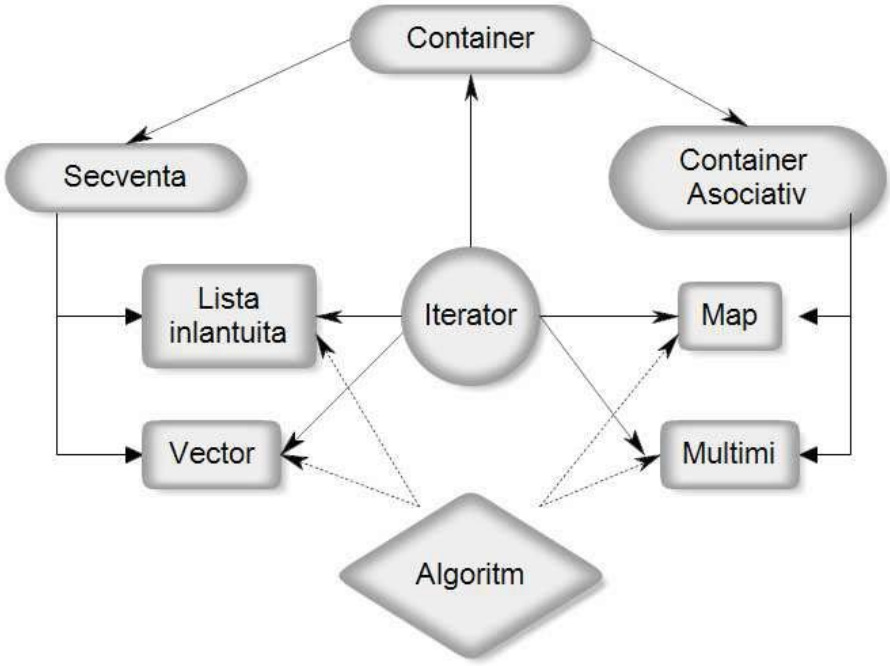


Figura 1.3: Asocierea conceptelor din STL.

Pentru a parcurge o colecţie se foloseşte un iterator. *Iteratorul* este o generalizare a conceptului de index dintr-un şir de elemente. Cu ajutorul iteratorului se poate face o referință la obiectele din colecţie. Pentru a ușura şi eficientiza programarea în STL s-au introdus algoritmi generici, care pot fi aplicați pe o multitudine de structuri de date. Un exemplu de un astfel de algoritm este sortarea.

Exerciții

1.11 Copiaţi următorul program într-un fişier cu numele person.cpp, apoi rulați acest exemplu.

```
#include <iostream>
#include <string>
#include <list>
#include <set>
#include <map>
```

```
#include <fstream>
using namespace std;

class Person
{
private:
  string firstName;
  string lastName;
  int age;
  string job;

public:
  Person() {}
  Person(string firstName, string lastName, int age, string job)
  {
    this->firstName = firstName;
    this->lastName = lastName;
    this->age = age;
    this->job = job;
  }
  friend ostream& operator << (ostream &cout, Person& p)
  {
    cout << p.firstName << ",";
    cout << p.lastName << ",";
    cout << p.age << ",";
    cout << p.job << ";";
    return cout;
  }
};

int main()
{
  Person* alex1 = new Person("Alex", "Daicu", 24, "programator");

  list<Person> people;
  people.push_back(Person("Dan", "Popovici", 23, "student"));
  people.push_back(Person("Ion", "Daicu", 13, "elev"));
  people.push_back(Person("Radu", "Corlatescu", 33, "lector"));
  people.push_back(Person("Daniela", "Popovici", 24, "student"));
  people.push_back(Person("Daicu", "Alex", 24, "programator"));
  people.push_back(Person("Dan", "Coradu", 21, "muncitor"));
  people.push_back(Person("Raluca", "Balan", 34, "lector"));
  people.push_back(Person("Alex", "Daicu", 24, "programator"));
  people.push_back(Person("Alex", "Daicu", 25, "programator"));

  ifstream infile("sample.txt");
```

```
  string line;
  string word;

  getline(infile, line, '\n');
  word = line.substr(0, line.find(' '));
  cout << word << endl;

  return 0;
}
```

1.12 Afişaţi câte persoane sunt programator şi câte au vârsta sub 30 de ani.

1.13 Implementați operatorul "<", pentru clasa *Person*. Această funcţie returnează *true* dacă două persoane au același nume sau numele primei persoane este mai mic (în sensul ordonării lexicografice ca cea dintr-un dicționar sau carte de telefon) decât celei de a doua şi *false* în caz contrar.

1.14 Adaugați elementele din listă într-un *set* după care afișați conținutul mulțimii. De câte ori apare persoana Alex Daicu?

1.15 Citiți fișierul *sample.txt* linie cu linie şi

- Afişaţi numărul total de cuvinte.
- Afişaţi numărul total de cuvinte distincte.
- Realizaţi o histogramă a frecvenţei de apariţie a cuvintelor (de câte ori apare fiecare cuvânt).

Capitolul 2

Grafuri în Java

În cadrul acestui capitol se va studia determinarea izomorfismului a două grafuri, o problemă întâlnită într-o gamă largă de domenii, printre care şi biochimia.

2.1 Breviar teoretic

Un graf este o pereche $G =< V, M >$, unde V este o mulțime de vârfuri, iar $M \subseteq V$ x V este o mulțime de muchii. O muchie de la vârful a la vârful b este notată cu pereche ordonată (a, b), dacă graful este orientat și cu mulțimea $\{a, b\}$ dacă graful este neorientat. În cele ce urmează vom presupune că vârfurile a şi b sunt diferite. Două vârfuri unite printr-o muchie se numesc adiacente. Un drum este o succesiune de muchii de forma:

$$(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n)$$

sau de forma

$$\{a_1, a_2\}, \{a_2, a_3\}, ..., \{a_{n-1}, a_n\}$$

după cum graful este orientat sau neorientat. Lungimea drumului este egală cu numărul muchiilor care îl constituie. Un drum simplu este un drum în care nici un vârf nu se repetă. Un ciclu este un drum care este simplu, cu excepția primului și ultimului vârf, care coincid. Un graf aciclic este un graf fără cicluri. Un subgraf al lui G este un graf $< V', M' >$, unde $V' \subseteq V$, iar M' este formată din muchiile din M care unesc vârfuri din V'. Un graf parțial este un graf $< V, M'' >$, unde $M'' \subseteq M$.

Un graf neorientat este conex, dacă între oricare două vârfuri există un drum. Pentru grafuri orientate, această noțiune este întărită: un graf orientat este tare conex, dacă între oricare două vârfuri i şi j există un drum de la i la j şi un drum de la j la i.

În cazul unui graf neconex se pune problema determinării componentelor sale conexe. O componentă conexă este un subgraf conex maximal, adică un subgraf

conex în care nici un vârf din subgraf nu este unit cu unul din afară printr-o muchie a grafului inițial. Împărțirea unui graf $G =< V, M >$ în componentele sale conexe determină o partiție a lui V şi a uneia lui M.

Un arbore este un graf neorientat aciclic conex. Sau, echivalent, un arbore este un graf neorientat în care există exact un drum între oricare două vârfuri. Un graf parțial care este un arbore se numește arbore parțial.

Vârfurilor unui graf li se pot atașa informații numite uneori valori, iar muchiilor li se pot ataşa informaţii numite uneori lungimi sau costuri.

Există cel puţin trei moduri evidente de reprezentare ale unui graf:

- Printr-o matrice de adiacenţă A, în care $A[i, j] = true$ dacă vârfurile i şi j sunt adiacente, $A[i, j] = false$ în caz contrar. O variantă alternativă este să-i dăm lui $A[i, j]$ valoarea lungimii muchiei dintre vârfurile i şi j, considerând $A[i, j] = +\infty$ atunci când cele două vârfuri nu sunt adiacente. Memoria necesară este în ordinul lui n^2 (unde n este numărul de vârfuri în graf). Cu această reprezentare, putem verifica uşor dacă două vârfuri sunt adiacente. Pe de altă parte, dacă dorim să aflăm toate vârfurile adiacente ale unui vârf dat, trebuie să analizăm o întreagă linie din matrice. Aceasta necesită n operații, independent de numărul de muchi care conectează vârful respectiv.

- Prin liste de adiacenţă, adică prin atașarea la fiecare vârf i a listei de vârfuri adiacente lui (pentru grafuri orientate este necesar ca muchia să plece din i). Într-un graf cu m muchii, suma lungimilor listelor de adiacenţă este $2m$, dacă graful este neorientat, respectiv m, dacă graful este orientat. Dacă numărul muchiilor în graf este mic, această reprezentare este preferabilă din punct de vedere al memoriei necesare. Este posibil să examinăm toți vecinii unui vârf dat, în medie, în mai puţin de n operații. Pe de altă parte, pentru a determina dacă două vârfuri sunt adiacente, trebuie să analizăm lista de adiacenţă a lui i (şi, posibil, lista de adiacenţă a lui j), ceea ce este mai puţin eficient decât consultarea unei valori logice în matricea de adiacenţă.

- Printr-o listă de muchii. Această reprezentare este eficientă atunci când avem de examinat toate muchiile grafului.

2.2 Izomorfismul grafurilor

Două grafuri sunt izomorfe dacă există o corespondenţă unu la unu între nodurile şi arcele care conectează nodurile, de exemplu grafurile a) şi b) din Fig. 2.1. Această proprietate se referă la identitatea dintre structura grafurilor fără a ţine seama de numele atribuite nodurilor.

Graful din Figura 2.1 c) nu are o structură identică cu grafurile a) şi b) şi deci nu este izomorf cu ele.

Un exemplu de algoritm pentru determinarea izomorfismului dintre două grafuri este prezentat în [12], acesta fiind o adaptare a algoritmului descris în [37].

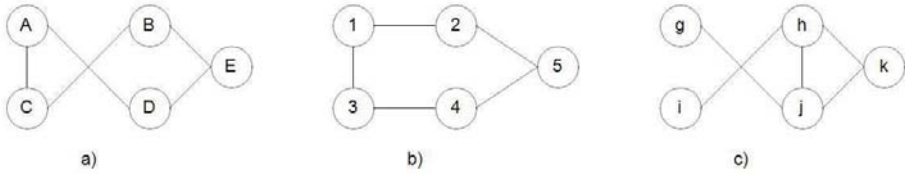


Figura 2.1: Exemple de grafuri: a) şi b) sunt izomorfe iar c) nu este izomorf cu a) şi b)

Ideea de bază a algoritmului este o căutare sistematică a posibilității de "împerechere" a unui nod din primul graf cu unul din al doilea. Procedura *main*, descrisă în continuare în limbaj pseudocod, constă dintr-un test simplu pentru o determinare preliminară a non-izomorfismului dintre grafuri:

```
INPUT: Grafurile G1 si G2
OUTPUT: Sirul f daca exista un izomorfizm intre G1 si G2
         sau NONE daca nu exista un asemenea sir
Procedura MAIN:
   for i = 1 ... n do
      inv1i = un invariant pentru nodul i din G1
      inv2i = un invariant pentru nodul i din G2
   endfor
   if inv1 sortat crescator != inv2 sortat crescator then
      return NONE
   rearanjare noduri in G1 si inv1
   if IZOMORF ( multimeVida, 1, f) then
      retrurn f
   else
      return NONE
```

Procedura *ISOMORPH* este o procedură recursivă care parcurge spaţiul posibilităţiilor într-un mod sistematic (backtracking):

```
INPUT: multimea S, numarul k, si un sir f
OUTPUT: True daca f poate fi extins pe tot graful de intrare
Procedura ISOMORF:
   if k = n + 1 then
      return TRUE
   foreach j in V2/S
      if inv1k != inv2j or ! CAN_MATCH(k, j, f)
         salt la urmatoarea iteratie
      f[k]=j
   if ISOMORF(k+1, S U {j}, f) then
      return true
   endfor
   return FALSE
```

Procedura *CAN_MATCH* returnează TRUE dacă nodurile k şi j pot fi considerate echivalente:

- Au același număr de muchii.
- Dacă există o muchie între noduri considerate echivalente în primul graf ele trebuie să existe şi în al doilea.

Pentru mai multe informaţii despre izomorfismul grafurilor consultați paginile [47] și [51].

2.3 Pachetul de clase JGraphT

Pentru implementarea grafurilor în Java se poate folosi pachetului de clase JGraphT disponibil la `http://www.jgrapht.org/`. Acest pachet definește clasele generice *SimpleGraph* şi *SimpleWeightedGraph* pentru lucrul cu grafuri care nu au, respectiv au, informație pe arce. Aceste clase generice trebuie să fie parametrizate pentru tipul de date al nodurilor, de exemplu *String* şi al arcelor, *DefaultEdge* sau *DefaultWeightedEdge*. Biblioteca JGraphT folosește o reprezentare a grafurilor prin liste de adiacenţă implementate cu ajutorul clasei *LinkedHashSet*.

Construirea grafurilor se face prin adăugarea de noduri (metoda *addVertex*) şi arce (metoda *addEdge*), iar în cazul arcelor cu valoare această valoare se poate seta prin metoda *setEdgeWeight*. Un exemplu de construire a unui graf circular este prezentat mai jos:

```
public static SimpleWeightedGraph<String, DefaultWeightedEdge>
    createCircleStringGraph(int n, int offset) {

  SimpleWeightedGraph<String, DefaultWeightedEdge> g
      = new SimpleWeightedGraph<String, DefaultWeightedEdge>
                (DefaultWeightedEdge.class);

  String first = "v" + (1);
  g.addVertex(first);
  String prev = first;
  for (int i = 2; i <= n; i++)
  {
    String current = "v" + (i);
    g.addVertex(current);
    DefaultWeightedEdge E = g.addEdge(prev, current);
    g.setEdgeWeight(E, (i + offset) % n + 1.0);
    prev = current;
  }
  DefaultWeightedEdge E = g.addEdge(prev, first);
  g.setEdgeWeight(E, (1 + offset) % n + 1.0);
  return g;
```

```
  }
```

Parcurgerea grafului se poate face utilizând clasa *BreadthFirstIterator* ca în exemplul de mai jos în care nodurile grafului *g* sunt de tip *String*:

```
SimpleWeightedGraph<String, DefaultWeightedEdge>
  g = createCircleStringGraph(5, 2);

for (BreadthFirstIterator<String, DefaultWeightedEdge> i
  = new BreadthFirstIterator<String, DefaultWeightedEdge>(g);
            i.hasNext();)
{
  String vertex = (String)i.next();
  System.out.println(vertex);
}
```

Metodele disponibile pentru manipularea arcelor sunt următoarele: *edgesOf, getEdgeSource, getEdgeTarget* şi *getEdgeWeight*.

Exerciţii

2.1 Parcurgeți grafurile generate şi pentru fiecare nod afișați gradul nodului şi arcele din nodul respectiv. Gradul unui nod se definește ca fiind numărul de arce conectate la nodul respectiv;

2.2 Implementați o metodă *izomorfism* care determină dacă două grafuri sunt izomorfe;

2.3 Modificaţi metoda *izomorfism* astfel încât să permită folosirea grafurilor care au informație pe arce.

2.4 Folosirea grafurilor în biochimie

Pentru modelarea moleculelor sau a formulelor chimice, precum şi a legăturilor dintre elemente se pot folosi grafuri. În funcție de ordinea în care se începe descrierea unei molecule grafurile rezultate pot să difere, dar sunt izomorfe pentru aceeași formulă chimică, ca în Figura 2.2.

Pentru reprezentarea şi stocarea formulelor chimice şi a moleculelor există diverse formate de fișier [46] printre care şi formatul Protein Data Bank (PDB). [50]

Am creat clasa *atom* (vezi fişierul atom.java) care implementează o structură de date minimală pentru folosirea clasei *JGraphT* pentru modelarea structurilor moleculare.

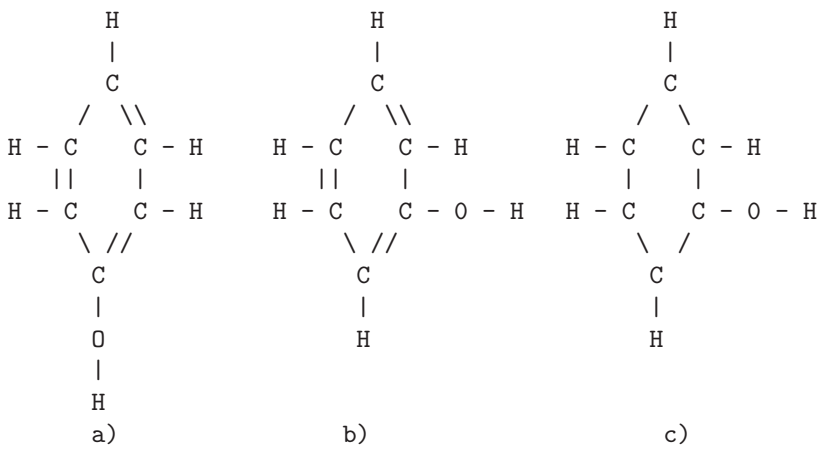


Figura 2.2: Exemplu de formule chimice izomorfe (identice) a) şi b). Reprezentarea folosind un graf a moleculei b) este aratată în c).

Exerciţii

2.4 Implementarea unei metode care determină dacă două descrieri ale unei molecule sunt echivalente.

2.5 Fiind puse la dispoziție o bază de date de formule chimice în directorul "data" şi o serie de molecule neidentificate în directorul "necunoscute", scrieți un program prin care să identificați aceste molecule necunoscute. Mai multe informaţii despre moleculele din directorul data sunt disponibile la adresa `http://www.reciprocalnet.org/recipnet/search.jsp`. Numele fișierului (fără extensia ".pdb") trebuie introdus în căsuța "sample number" după care daţi click pe "search", conform exemplului din Figura 2.3. Un exemplu de rezultat de căutare este prezentat în Figura 2.4.

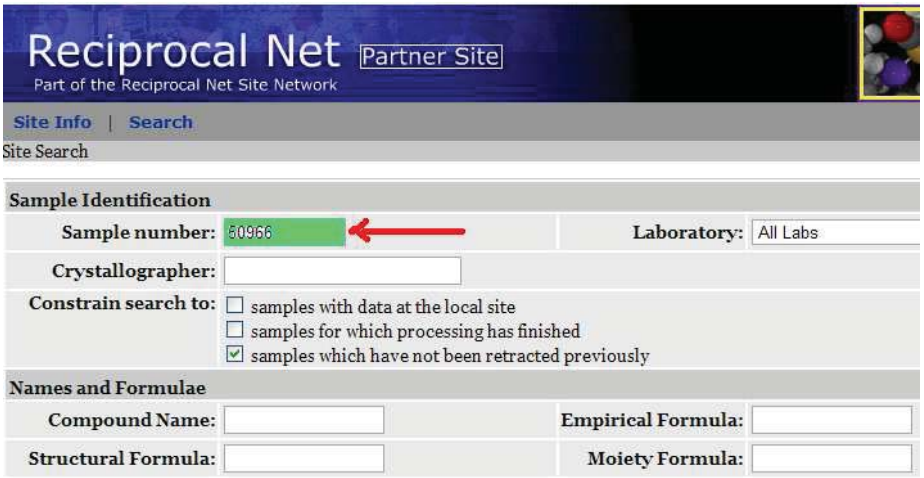


Figura 2.3: Numele fișierului (fără extensia ".pdb") trebuie introdus în căsuța "sample number" după care daţi click pe "search".

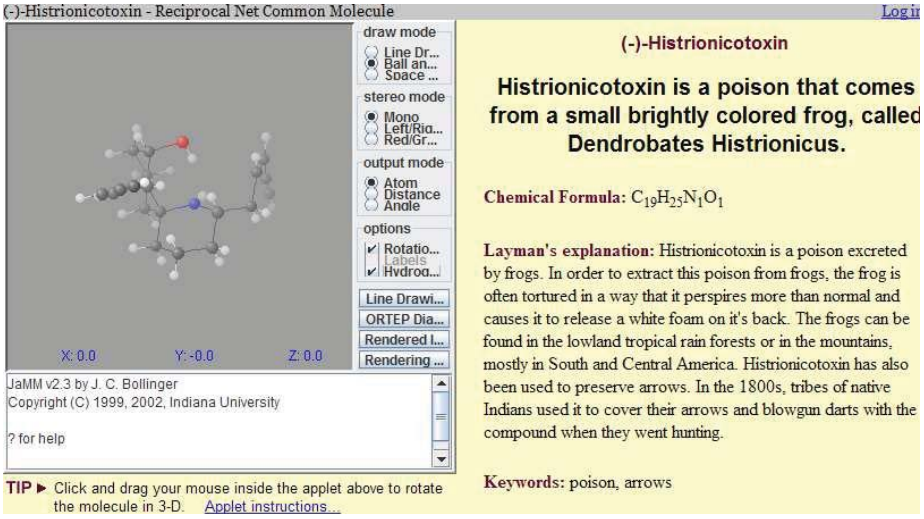


Figura 2.4: Exemplu de detalii despre o moleculă.

Capitolul 3

Arbori în C++

În cadrul acestui capitol se va studia implementarea folosind C++ STL a algoritmului de construcție a supertree-urilor, pornind de la doi arbori care au noduri comune. Breviarul teoretic pentru această aplicație a fost preluat din [2].

3.1 Breviar teoretic

Fie G un graf orientat. G este un *arbore cu rădăcina* r, dacă există în G un vârf r din care oricare alt vârf poate fi accesat printr-un drum unic (vezi Figura 3.1).

Definiția este valabilă şi pentru cazul unui graf neorientat, alegerea unei rădăcini fiind însă în acest caz arbitrară: orice arbore este un arbore cu rădăcină, iar rădăcina poate fi fixată în oricare vârf al său. Aceasta deoarece dintr-un vârf oarecare se poate ajunge în oricare alt vârf printr-un drum unic.

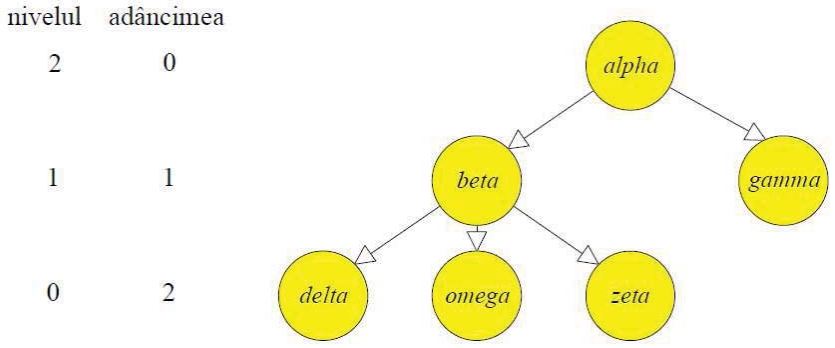


Figura 3.1: Un arbore cu rădăcină.

Când nu va fi pericol de confuzie, vom folosi termenul "arbore" în loc de termenul corect "arbore cu rădăcină". Cel mai intuitiv este să reprezentăm un arbore cu rădăcină, ca pe un arbore propriu-zis. În Figura 3.1, vom spun că *beta* este *tatăl* lui *delta* şi fiul lui *alpha*, că *beta* şi *gamma* sunt *fraţi*, că *delta* este un *descendent* al lui *alpha*, iar *alpha* este un *ascendent* al lui *delta*. Un vârf *terminal* este un vârf fără descendenți. Vârfurile care nu sunt terminale sunt *neterminale*. De multe ori, vom considera că există o ordonare a descendenților aceluiași părinte: *beta* este situat la stânga lui *gamma*, adică *beta* este fratele mai vârstnic al lui *gamma*.

Oricare vârf al unui arbore cu rădăcină este rădăcina unui *subarbore* constând din vârful respectiv şi toți descendenții săi. O mulțime de arbori disjuncți formează o *pădure*.

Într-un arbore cu rădăcină vom adopta următoarele notații: *adâncimea* unui vârf este lungimea drumului dintre rădăcină şi acest vârf; *înălţimea* unui vârf este lungimea celui mai lung drum dintre acest vârf şi un vârf terminal; *înălţimea arborelui* este înălțimea rădăcinii; *nivelul* unui vârf este înălţimea arborelui minus adâncimea acelui vârf.

Reprezentarea unui arbore se poate face prin adrese, ca în cazul listelor înlănţuite. Fiecare vârf va fi memorat în trei locații diferite, reprezentând informația propriu-zisă a vârfului (valoarea vârfului), adresa celui mai vârstnic fiu şi adresa următorului frate. Păstrând analogia cu listele înlănţuite, dacă se cunoaște de la început numărul maxim de vârfuri, atunci implementarea arborilor cu rădăcină se poate face prin tablouri paralele.

Dacă fiecare vârf al unui arbore cu rădăcină are până la n fii, arborele respectiv se numește n-ar. Un arbore binar poate fi reprezentat prin adrese. Într-un arbore binar, numărul maxim de vârfuri de adâncimea k este 2^k. Un arbore binar de înălţime i are cel mult $2^{i+1} - 1$ vârfuri, iar dacă are exact $2^{i+1} - 1$ vârfuri se numeşte *arbore plin*. Vârfurile unui arbore plin se numerotează în ordinea adâncimii. Pentru aceeași adâncime, numerotarea se face în arbore de la stânga la dreapta.

Un arbore binar cu n vârfuri şi de înălţimea i este *complet* dacă se obține din arborele binar plin de înălţime i, prin eliminarea, dacă este cazul, a vârfurilor numerotate cu $n+1, n+2, ..., 2^{i+1} - 1$. Acest tip de arbore se poate reprezenta secvențial folosind un tablou T, punând vârfurile de adâncimea k de la stânga la dreapta, în pozițiile $T[2^k], T[2^k + 1], ..., T[2^{k+1} - 1]$ (cu posibila excepție a nivelului 0, care poate fi incomplet).

3.2 Supertree

Conform teoriei evoluționiste toate organismele au evoluat dintr-un strămoș comun şi de-a lungul timpului a avut loc un proces de diversificare. Astfel din toate organismele existente (sau dispărute) se poate construi un arbore cu rădăcină supranumit şi arborele vieții. Determinarea şi maparea unui proces evoluționist se numeşte în literatura de specialitate crearea unui arbore filogenetic [49].

Astfel de arbori filogenetici se pot construi cu ajutorul unei serii de metode (ca de exemplu studiul ADN), iar procesul de unificare a acestor arbori parţiali se numeşte crearea de supertrees.

În Figura 3.2 putem observa doi arbori filogenetici a) şi b), iar arborele c) este rezultatul combinării celor doi arbori.

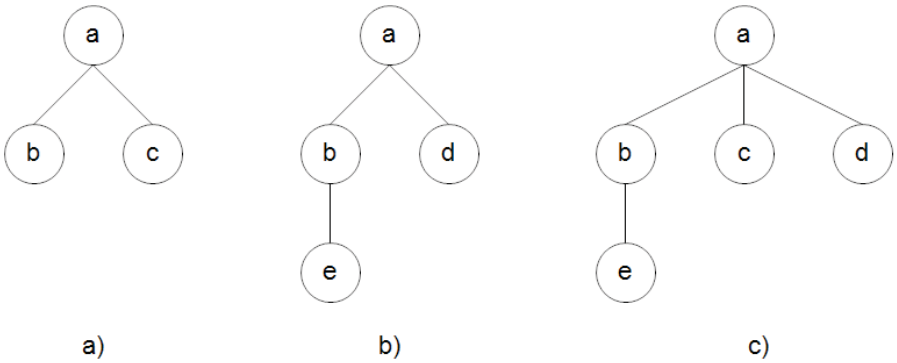


Figura 3.2: Arbori filogenetici a) şi b). Supertree-ul rezultat din combinarea arborilor a) şi b).

Un format de fişier standard simplu pentru stocare pe disc a acestor arbori este formatul Newick [48]. Regulile acestui format sunt:

- Dacă un nod nu are fii (este un nod frunză) se scrie numele lui.
- Dacă un nod are fii, aceștia vor fi puși în paranteze înaintea numelui nodului: (listă fii)nume_nod.
- Dacă un nod are mai mulți fii, aceștia vor fi despărțiți prin virgulă.

Reprezentarea în format Newick a grafurilor din Figura 3.2:

- Fig. 3.2 a): (b,c)a
- Fig. 3.2 b): ((e)b,d)a
- Fig. 3.2 c): ((e)b,c,d)a

O aplicaţie pentru vizualizarea grafurilor reprezentate în formatul Newick a fost dezvoltată de Universitatea Indiana [44] şi poate fi folosită pentru testarea aplicațiilor.

Pentru lucrul cu arborii filogenetici vom folosi clasa ”node” care conține numele nodului şi o listă cu fiii acestui nod (în cazul în care un nod este frunză lista fiilor nodului este nulă). Această clasă este prezentată în continuare.

```
class node
{
private:
    string name;
    list<node> data;

public:
    string getName() {return this->name;}
    void setName(const string &newName) {name = newName;}
    list<node> getData() {return this->data;}
    void setData(const list<node> &newData) {data = newData;}
    node() {}
    virtual ~node() {}

//functie de afisare a unui nod in formatul Newick
    friend ostream& operator << (ostream &cout,const node &n)
    {
        if(n.data.size())
        {
            cout << "(";
            for(list<node>::const_iterator i = n.data.begin() ;
                            i != n.data.end() ; i++)
                if (i == n.data.begin())
                    cout << *i;
                else
                    cout << "," << *i;
            cout << ")";
        }
        cout << n.name;
        return cout;
    }
};
```

Pentru a facilita vizualizarea şi testarea metodelor de construire a supertree-urilor se dă următoarea funcție de conversie de la formatul Newick la reprezentarea de tipul obiectului node:

```
node build_tree(const string &line)
{
string cur_word;
stack<node> st;

node temp;

list<node> cur_data;
for(int i = 0 ; i<line.length() ; i++)
```

```
{
    if(line[i] == '(')
    {
        st.push(temp);
        list<node> t;
        t = temp.getData();
        t.clear();
        temp.setData(t);
        temp.setName("");
    }
    else if(line[i] == ')')
    {
        node temp_arr;
        temp_arr.setName(cur_word);
        temp_arr.setData(cur_data);
        cur_data.clear();
        cur_word = "";
        list<node> t;
        t = temp.getData();
        t.push_back(temp_arr);
        temp.setData(t);
        cur_data = temp.getData();
        temp = st.top();
        st.pop();
    }
    else if(line[i] == ',')
    {
        node temp_arr;
        temp_arr.setName(cur_word);
        temp_arr.setData(cur_data);
        cur_data.clear();
        cur_word = "";
        list<node> t;
        t = temp.getData();
        t.push_back(temp_arr);
        temp.setData(t);
    }
    else cur_word.append(line.substr(i, 1));
}//end of for

        node temp_arr;
        temp_arr.setName(cur_word);
        temp_arr.setData(cur_data);
        temp = temp_arr;
    return temp;
}
```

Ca aplicație în cadrul laboratorului vom trata problema reconstrucției clasificării organismelor (genuri, familii, specii, subspecii etc.) informație disponibilă pe site-ul web Integrated Taxonomic Information System [19].

3.3 Determinarea supertree-ului unor subarbori

Determinarea supertree-ului unor subarbori având aceeași rădăcină

Cea mai simplă formulare a problemei de construcție a supertree-urilor este dacă arborii parțiali au aceeași rădăcină, ca în exemplul din Figura 3.2.

Exerciţii

3.1 Crearea unei metode node *merge_tree*(const node&, const node&) care să returneze supertree-ul format din cei doi arbori dați ca parametru. Testați funcţia pe exemplul din Figura 3.2.

3.2 Reconstrucția familiei *Canidae*. Informația completă referitoare la această familie se poate regăsi la adresa `http://www.itis.gov/servlet/SingleRpt/SingleRpt?search_topic=TSN&search_value=180594`.

În directorul *data/tsn_180595* se găsesc fişierele *partial_0.txt,...,partial_9.txt* care conțin arbori parțiali, iar fișierul *compleate.txt* conține arborele complet (a se folosi pentru validarea rezultatului).

Determinarea supertree-ului unor subarbori care nu au aceeași rădăcină

În acest caz sunt permiși arbori pentru care rădăcina unuia dintre arbori să fie un nod intermediar sau frunză al celuilalt arbore ca în exemplul din Figura 3.3.

Exerciţii

3.3 Crearea unei metode *node build_tree(node,node)* care să returneze supertree-ul format din cei doi arbori dați ca parametru. Testați funcţia pe exemplul din Figura 3.3.

3.4 Reconstrucția clasei ciupercilor. Informația completă referitoare la această familie se poate regăsi la adresa `http://www.itis.gov/servlet/SingleRpt/SingleRpt?search_topic=TSN&search_value=555705`.

În directorul *data/tsn_555705* se găsesc fişierele *partial_0.txt,...,partial_116.txt* care conțin arbori parțiali, iar fișierul *compleate.txt* conține arborele complet (a se folosi pentru validarea rezultatului).

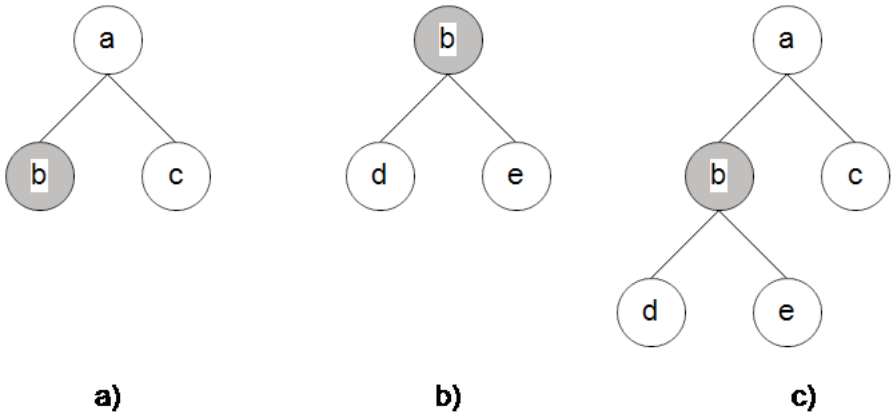


Figura 3.3: Arbori filogenetici având rădăcini diferite a) şi b). Supertree-ul rezultat din combinarea arborilor a) şi b).

Determinarea supertree-ului unor subarbori care nu au aceeași rădăcină sau au noduri lipsă

În acest caz sunt permiși arbori pentru care ierarhia nodurilor unuia dintre arbori să aibă noduri lipsă față de cea din celălalt arbore ca în exemplul din Figura 3.4.

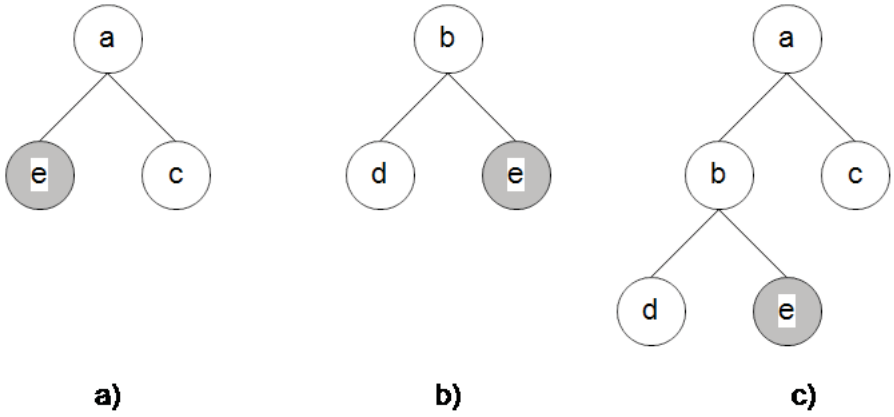


Figura 3.4: Arbori filogenetici având noduri lipsă în ierarhie a) şi b). Supertree-ul rezultat din combinarea arborilor a) şi b).

Exerciţii

3.5 Crearea unei metode *node build_tree(node,node)* care să returneze supertree-ul format din cei doi arbori dați ca parametru. Testați funcţia pe exemplul din Figura 3.4.

3.6 Reconstrucția clasei plantelor. Informația completă referitoare la această familie se poate regăsi la adresa `http://www.itis.gov/servlet/SingleRpt/SingleRpt?search_topic=TSN&search_value=202422`.

În directorul *data/tsn_202422* se găsesc fişierele *partial_0.txt,...,partial_?.txt* care conţin arbori parţiali, iar fişierul *compleate.txt* conține arborele complet (a se folosi pentru validarea rezultatului).

Capitolul 4

Programare paralelă în CUDA

Acest capitol prezintă programarea procesoarelor grafice folosind limbajul CUDA-C. După o introducere în teoria complexității paralele, este prezentată pe scurt, arhitectura și modelul de programare CUDA, urmată de 9 exemple de laborator (în două capitole), menite să acopere (de departe non-exhaustiv) acest domeniu. Se recomandă familiarizarea prealabilă cu programarea paralelă, limbajul C/C++ și arhitectura calculatoarelor. Pentru câteva din exemplele prezentate se recomandă cunoștințe de procesare de imagini, filtrări și transformări integrale. În exemplele de la finalul capitolului, sunt prezentate tehnici de programare orientată pe obiecte și programare generică, precum și programare dinamică.

4.1 Breviar teoretic

Algoritmii secvenţiali sunt denumiţi fezabili[1] dacă ordinul lor de timp este polinomial - $O(n^k)$, unde k este o constantă, iar n mărimea cazului. Astfel, algoritmii căutaţi pentru maşinile secvenţiale sunt cei cu timp polinomial[2]. Practic, pentru toate instanţele unei probleme, algoritmul calculează soluţia în timp polinomial. Mulţimea tuturor problemelor pentru care există algoritmi polinomiali formează clasa P (Figura 4.1).

Problemele rezolvabile polinomial fac parte dintr-o clasă mai largă, cea a problemelor nedeterminist-polinomiale - NP. Problema colorării unui graf, problema satisfiabilităţii circuitului, problema *clique*, problema găsirii celui mai scurt superstring sunt exemple de probleme NP. Clasa NP este determinată de

[1] Concept care înseamnă că implementarea lor pe o maşină von Neumann, secvenţială, va calcula soluţia unei probleme într-un timp rezonabil.

[2] Algoritmul bubble-sort, cu timp pătratic, este polinomial, la fel şi quicksort - $O(n \log n)$. Dar dacă scriem un algoritm care generează toate permutările posibile ale şirului şi le verifică, algoritmul devine exponenţial.

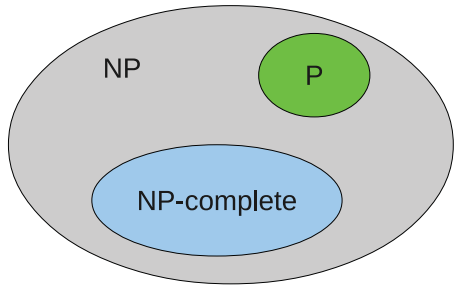


Figura 4.1: Probleme P, NP şi NP-complete.

mulţimea problemelor ce pot fi verificate[3] de un algoritm într-un timp polinomial. Timpul polinomial este asociat cu un timp de rezolvare rapid pe o maşină secvenţială. Intuitiv, clasa P este formată de problemele rezolvabile rapid (în timp rezonabil), în vreme ce clasa NP este formată de problemele ce pot fi verificate rapid[4]. Fireşte că orice problemă ce poate fi rezolvată în timp polinomial poate fi şi verificată în timp polinomial (deci $P \subset NP$)[5].

Un statut special îl au problemele NP-complete. Acestea fac parte tot din clasa NP (Figura 4.1), însă, în plus, pentru o problemă NP-completă se poate demonstra că orice altă problemă din clasa NP se reduce în timp polinomial la ea. Exemple de probleme NP-complete sunt problema satisfacerii (SATI) sau problema colorării unui graf.

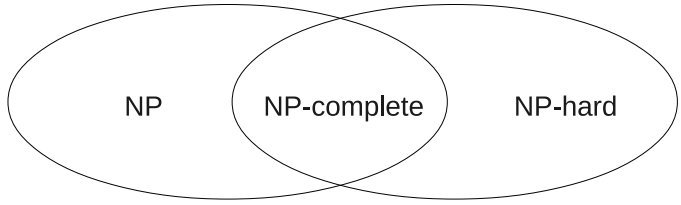


Figura 4.2: Probleme NP şi NP-hard.

Există şi probleme pentru care se poate demonstra că problemele din clasa NP se pot reduce în timp polinomial la ele, însă ele însele nu fac parte din

[3]Verificarea unei probleme presupune existenţa apriori a unei probe, folosită de algoritmul de verificare pentru a decide dacă reprezintă sau nu o soluţie a problemei.

[4]De regulă, algoritmii nedeterminişti realizează mai întâi 'ghicirea' probei, urmată de verificarea acesteia.

[5]Problema sortării este simultan şi în clasa P şi în clasa NP, însă problema celui mai scurt superstring este doar în NP.

clasa NP! Acestea sunt denumite probleme NP-hard (Figura 4.2). Sunt un fel de probleme NP-complete care însă nu fac parte din clasa NP. Cel mai bun exemplu este problema opririi maşinii Turing[6].

Dacă pentru maşinile secvenţiale noţiunea de fezabil este cea asociată timpului polinomial, pentru maşinile paralele corespondentul acesteia este cea de timp poli-logaritmic, $O(log^k n)$, k fiind de asemenea o constantă. În cele ce urmează ne propunem să dăm o interpretare intuitivă a acesteia.

Să luăm un algoritm de sortare, de exemplu *mergesort*. Ordinul de timp al algoritmului *mergesort* este $O(n \log n)$. Folosind o maşină paralelă, ne aşteptăm să obţinem un ordin de timp mai bun. Considerăm că şirul de sortat este stocat în memoria partajată sub forma şirului $T[1 \ldots n]$, de lungime n şi că numărul de procesoare poate fi exprimat ca o funcţie polinomială $p(n)$, depinzînd de mărimea cazului.

Pentru prima abordare[7], ne propunem să împărţim şirul în blocuri de lungime egală $\frac{n}{p(n)}$ tuturor celor $p(n)$ procesoare. Timpul total de sortare este format din:

- împărţirea şirului $T[1 \ldots n]$ în blocuri pentru fiecare procesor se poate face în timp $O(\frac{n}{p(n)})$, timpul necesar fiecărui procesor pentru a-şi copia propriul bloc din memoria partajată în memoria locală;

- sortarea în paralel a fiecărui bloc, de către fiecare procesor, într-un timp $O(\frac{n}{p(n)}\ log(\frac{n}{p(n)}))$;

- interclasarea celor $p(n)$ şiruri rezultate pentru formarea şirului final. Acest lucru se realizează în următorii paşi:

  pas 1: $\frac{p(n)}{2}$ procesoare interclasează $p(n)$ şiruri de lungime $\frac{n}{p(n)}$;

  pas 2: $\frac{p(n)}{4}$ procesoare interclasează $\frac{p(n)}{2}$ şiruri de lungime $2\frac{n}{p(n)}$;

  ...

  pas $log_2 p(n)$: 1 procesor interclasează 2 şiruri de lungime $2^{log_2 p(n)-1}\frac{n}{p(n)} = \frac{n}{2}$.

  Timpul de interclasare pentru toate şirurile, de-a lungul celor $log_2 p(n)$ paşi este dat de lungimea totală a şirurilor, în limitele unei constante multiplicative (notăm $p(n)$ cu p, pentru lizibilitate):

[6] Interpretare intuitivă: dându-se un algoritm, se pune problema să construim un alt algoritm care calculează, în timp finit, dacă pentru orice intrare primul algoritm se opreşte sau nu.

[7] Explicaţia este preluată din [16].

$$\begin{aligned} S &= \frac{n}{p} + 2\frac{n}{p} + 4\frac{n}{p} + \cdots + 2^{log_2 p-1}\frac{n}{p} \\ &= \frac{n}{p}\left(2^0 + 2^1 + \cdots + 2^{log_2 p-1}\right) \\ &= \frac{n}{p}\left(2^{log_2 p} - 1\right) = \frac{(p-1)n}{p} \end{aligned}$$

de unde reiese că timpul de interclasare paralel este în $O(n)$.

Aşadar, pentru interclasarea prin împărţirea de blocuri de mărimi egale, timpul total pentru acest tip de sortare paralelă este $O(n + \frac{n}{p(n)} log \frac{n}{p(n)})$.

Performanţa algoritmului depinde direct de $p(n)$:

- dacă $p(n)$ este o constantă independentă de n, ajungem la un ordin de timp în $O(n \, log \, n)$, adică nu mai rapid decât algoritmul secvenţial!
- dacă însă $p(n) \leq n$, ordinul de timp devine $O(\frac{n \, log \, n}{p(n)})$, adică algoritmul se accelerează optim, timpul secvenţial devenind egal cu produsul dintre timpul paralel şi numărul de procesoare;
- pentru cazul în care $p(n) > n$, ordinul de timp este limitat superior de $O(n)$, oricât de multe procesoare am folosi.

Ceea ce căutăm este ca, odată cu mărirea polinomială a numărului de procesoare, problema să se accelereze, iar timpul paralel să devină logaritmic - $O(log \; n)$ sau poli-logaritmic - $O(log^k n)$.

Ceea ce ne-ar trebui ar fi o soluţie inerent paralelă, în care etapele de 'centralizare' a datelor (etapa de interclasare) să se realizeze în timp cel mult logaritmic[8]. Ar trebui cumva să socotim în mod paralel rangul(poziţia) r_i al fiecărui element în şirul deja sortat. Având rangul, într-un singur pas, n procesoare pot face apoi atribuirea $T[i] \leftarrow T[r_i]$ în timp constant, $O(1)$.

Calculul rangului se poate face dacă dispunem de $p(n) = n^2$ procesoare:

for $k \leftarrow 1$ **to** n^2 **do in parallel**
  $i \leftarrow \lfloor \frac{k}{n} \rfloor + 1$
  $j \leftarrow k - n\lfloor \frac{k}{n} \rfloor + 1$
  if $T[i] > T[j]$ **then** $c_{ij} \leftarrow 1$
    else $c_{ij} \leftarrow 0$

Într-un singur pas, fiecare din cele n^2 procesoare realizează testul $T[i] > T[j]$ şi asignează $c_{ij} = 1$ dacă este adevărat sau 0 în caz contrar, pentru toţi $i, j = 1 \ldots n$ (timp constant).

Calculul sumei a n elemente în timp logaritmic este prezentată în figura 4.3. Se realizează următoarea procedură de tip *reduce-sum*:

[8]Cea mai răspândită procedură de acest fel este paradigma *reduce*, aplicată pentru funcţii logice asociative precum suma, minim, maxim etc.

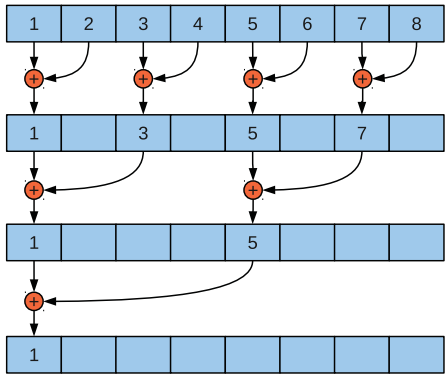


Figura 4.3: Calculul *reduce-sum*.

for $i \leftarrow 1$ **to** $log_2 n$ **do**
  for $j \leftarrow 1$ **to** $\frac{n}{2^i}$ **do in parallel**
    $k \leftarrow (j-1)2^i + 1$
    $C[k] \leftarrow C[k] + C[k + 2^{i-1}]$

Variabila i ia succesiv valorile $1, 2, \ldots log_2 n$. Pentru fiecare pas i, variabila j va avea tot atâtea valori câte procesoare lucrează la pasul i. Mai departe, k va da indexul locaţiei unde se depune rezultatul iar 2^{i-1} dă mărimea "pasului" pentru termenul care se adună.

Acelaşi lucru poate fi scris mai compact şi în pseudocod.

```
for (i = 1 ; i <= n ; i <<= 1) do
    for (j = 1 ; j < n ; j += (i << 1)) do in parallel
        C[j] += C[j + i]
```

Revenind la problema de sortare, pentru fiecare i, n procesoare calculează suma $\sum_{j=1}^{n} c_{ij}$, folosind procedura *reduce-sum* descrisă mai sus. Aceasta va da ca rezultat numărul de elemente faţă de care $T[i]$ este mai mare sau, altfel spus, rangul lui $T[i]$ (dacă $T[i]$ este primul element rangul său este 0). Calculul rangului se face în paralel şi se realizează printr-o procedură de tip *reduce-sum* în timp $O(log\, n)$.

Astfel, prin folosirea a n^2 procesoare algoritmul va sorta n elemente în timp logaritmic. Din cauza lui $n^2 \in n^{O(1)}$ spunem că numărul de procesoare utilizat este polinomial. În general, toate problemele care sunt paralel scalabile cu numărul de procesoare sunt fezabile, adică folosesc un număr de procesoare în ordin polinomial [16].

Aşadar, prin probleme paralel fezabile înţelegem acel tip de probleme pentru care există algoritmi paraleli ce folosesc un număr polinomial de procesoare,

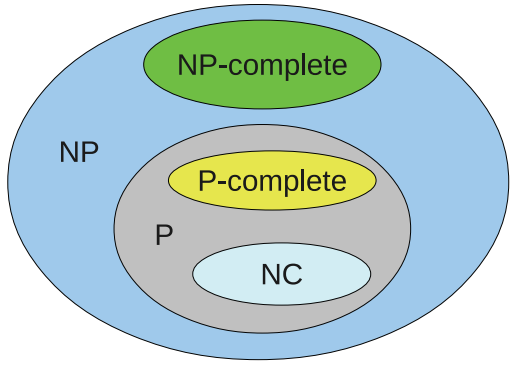


Figura 4.4: Nick's class.

$n^{O(1)}$, iar timpul de execuţie este polinomial, $n^{O(1)}$. Problemele de acest tip determină clasa P a problemelor paralel fezabile. Este clasa P descrisă mai sus pentru maşinile secvenţiale[9].

Legat de clasa P, în contextul algoritmilor paraleli, există o serie de probleme din P denumite P-complete. O problemă P-completă este o problemă pentru care orice problemă din P se poate reduce în timp poli-logaritmic la ea, folosind o maşină paralelă.

Interesante sunt însă problemele rezolvabile pe un număr polinomial de procesoare, $n^{O(1)}$, care admit algoritmi ce rulează în timp poli-logaritmic, $log^{O(1)}n$. Aceste probleme fac parte din clasa NC de probleme (Nick's class[10], Figura 4.4). Algoritmii de acest tip sunt denumiţi algoritmi cu grad ridicat de paralelism.

Pentru toate aceste tipuri de probleme NC există algoritmi secvenţiali polinomiali care le rezolvă. Reciproca nu este însă adevărată, adică există probleme secvenţiale cu timp polinomial care nu acceptă algoritmi paraleli în timp polilogaritmic[11]. Acestea sunt denumite probleme inerent secvenţiale. Astfel, se pare că deocamdată $NC \subset P$.

4.2 CUDA

CUDA (acronim pentru Compute Unified Device Architecture) este o arhitectură paralelă, dezvoltată de firma NVIDIA pentru procesoare grafice. CUDA permite calcul de uz general, folosind procesoarele grafice. CUDA nu oferă un sistem independent (fiind controlat de calculatorul gazdă), dar poate avea rolul

[9] În fond, şi un singur procesor poate fi privit ca funcţie polinomială, $1 \in n^{O(1)}$.

[10] De la numele matematicianului Nick Pippenger care s-a ocupat cu studiul circuitelor de adâncime polilogaritmică.

[11] Nu s-au găsit încă algoritmi de acest tip, dar nici nu s-a demonstrat că acest lucru n-ar fi posibil - problema $NC = P$ rămâne o problemă deschisă.

de accelerator de calcule masive. Astfel, găsim aplicații care folosesc CUDA în domenii ca: procesare de imagini, bio-imagistică (de exemplu, imagistică prin rezonanță magnetică), simulări fizice (hidrodinamică, câmpuri electromagnetice), bio-informatică, etc. Prin CUDA se încearcă aproprierea de domeniul calculului de înaltă performanță (HPC), rezervat în trecut exclusiv centrelor de calcul dotate cu supercalculatoare[12].

Scopul acestui laborator este familiarizarea cu arhitectura și mediul de programare CUDA. Exemplele sunt introductive, fără a intra în detaliile de optimizare.

Arhitectura CUDA este de tip many-core[13], fiind într-o evoluție rapidă, astfel că unele detalii tehnice își pot pierde semnificația de la un an la altul. La nivelul anului 2011, se observă și o tendință de dizolvare a demarcării clare dintre CPU/GPU[14], prin integrarea în procesoarele uzuale a modulelor specifice procesoarelor grafice (de exemplu Intel Sandy Bridge) sau integrarea în procesoarele grafice unități de execuție de uz general (de ex. arhitectura NVidia – Fermi, AMD – Fusion). O prezentare comparativă a arhitecturilor paralele multi- și many-core recente se găsește în [7].

Fenomenul de *revoluție many-core* [4] a apărut din necesitatea creșterii performanței de calcul, constrânsă de frecvența maximă și puterea disipată a procesoarelor. Pentru a utiliza din plin potențialul sistemelor many-core, avem nevoie de o schimbare de paradigmă față de programarea secvențială [5]. De regulă, sistemele actuale many-core sunt eterogene, conținând mai multe tipuri de unități de execuție, ca de exemplu:

- mai multe unități secvențiale “Single Data Single Instruction” – specifice microprocesoarelor scalare sau superscalare;
- unități de tip “Single Instruction Multiple Data” – extensii apărute, de exemplu la Intel MMX, SSE;
- blocuri de procesoare grafice, pentru a prelucra fluxuri mari de date cu instrucțiuni relativ simple.

Astfel, programatorul se confruntă cu multiple resurse de calcul paralele, eterogene, iar găsirea (dezvoltarea) unui limbaj de programare adecvat, de nivel înalt este încă o provocare.

În prezent arhitectura CUDA se poate programa prin:

- **CUDA-C** - extensie a limbajului C++, proprietar NVIDIA, limbajul folosit și în exercițiile de laborator [30];
- **OpenCL (Open Computing Language)** - mediu standardizat pentru calcul paralel pe platforme eterogene CPU-GPU [22];

[12] `http://www.top500.org/`

[13] Numim multi-core sistemele cu mai mult de un procesor, în timp ce termenul de many-core este folosit pentru sistemele cu un număr foarte mare de procesoare [5, 25]

[14] Central Processing Unit / Graphics Processing Unit

- **CUDA-Fortran** - extensie a limbajului Fortran;
- **DirectCompute** - tehnologie DirectX, proprietar Microsoft [27].

4.2.1 Terminologie, abrevieri

În cele ce urmează vom folosi următoarele abrevieri și următorii termeni:

- **CPU** Central Processing Unit - procesorul aflat în calculatorul gazdă.
- **GPU** Graphics Processing Unit - procesorul grafic, unitatea ce urmează a fi programată prin CUDA. Toate exemplele din laborator vor conține părți de cod destinate CPU, iar altele destinate GPU. În Figura 4.5 este prezentată legătura dintre sistemul gazdă (de regulă un calculator personal) și dispozitivele CUDA (una sau mai multe plăci grafice).
- **Thread** - fir de execuție.
- **Kernel** - nucleu, denumire consacrată în terminologia CUDA pentru a desemna procedura asociată unui fir de execuție CUDA.

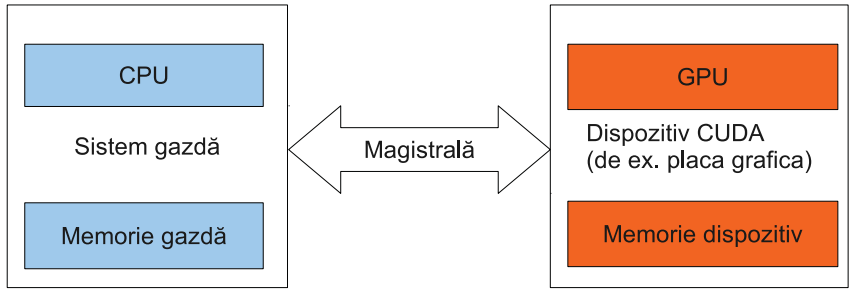


Figura 4.5: Relația sistem gazdă – dispozitive CUDA.

4.2.2 Arhitectura CUDA

Vom prezenta, pe scurt, elementele fundamentale ale arhitecturii CUDA, mai întâi din punct de vedere hardware, apoi al modelului de programare. Arhitectura CUDA este un sistem paralel de procesare ierarhic, oferind mai multe nivele de paralelism. În Figura 4.6 găsim următoarele elemente:

- **Procesorul scalar (Scalar Thread Processor, SP)** este unitatea de execuție de bază, conținând o unitate aritmetico-logică (ALU) pentru operații cu întregi și una pentru operații în virgulă flotantă.
- **Multiprocesorul (streaming multiprocessor, SM)** este unitatea de execuție paralelă de tip “Single-Instruction Multiple-Threads” (SIMT),

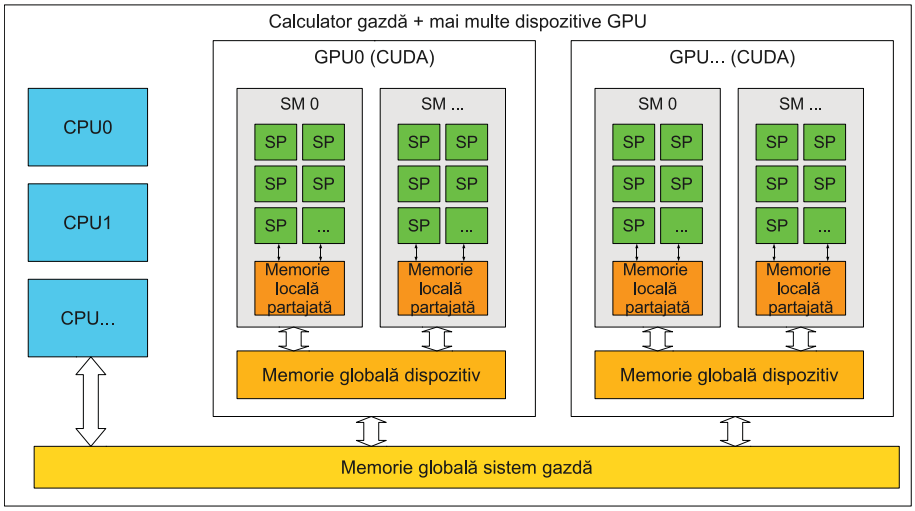


Figura 4.6: Arhitectura unui sistem complet CPU+GPU.

care înglobează mai multe (8-32) procesoare scalare. Multiprocesorul execută sincron, în paralel grupuri de câte 32 de threaduri, grupuri denumite *warps*.

- **Procesorul grafic (GPU)** înglobează unul sau mai multe multiprocesoare, de obicei acestea fiind prezente pe același chip, având acces la memoria globală a dispozitivului.
- **Memoria Globală (Global Memory)**, în terminologia CUDA, desemnează memoria de pe dispozitivul grafic (placa video), la care au acces toate multi-procesoarele. Această memorie este de ordinul 1-2 GB (anul 2011).
- **Memoria locală partajată (Shared Memory)** este un bloc de memorie, de ordinul a 48-64KB, cu acces rapid, servind ca o memorie cache. Fiecare multiprocesor conține o zonă dedicată de memorie partajată. Toate threadurile din același bloc au acces doar la memoria partajată atribuită blocului, nefiind posibilă comunicarea inter-bloc prin această memorie.
- **Sistemul de calcul** complet poate conține unul sau mai multe procesoare secvențiale, procesoare grafice, având acces partajat la memoria gazdă.

4.2.3 Modelul de programare

Din punct de vedere al programării, în CUDA s-a adoptat modelul bazat pe fluxuri (Stream Programming). Astfel, având un set de date, se aplică o serie de operații (nuclee) pe fiecare element, exemplificate în Figura 4.7.

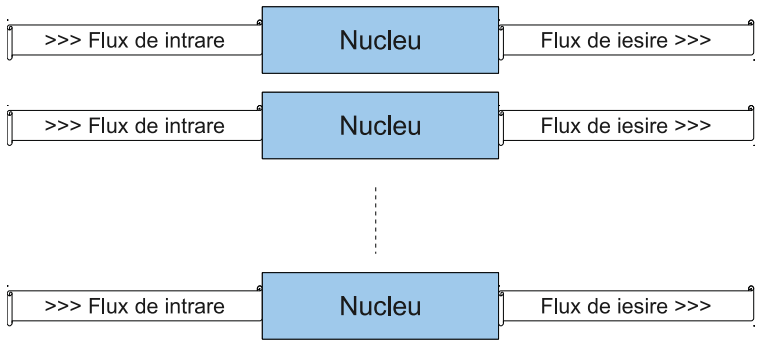


Figura 4.7: Procesarea cu fluxuri de date.

Programarea bazată pe fluxuri este eficientă în domeniile care satisfac următoarele 3 criterii: i) calcul intensiv (raport mare între instrucțiunile aritmetice față de operațiile cu memoria); ii) paralelism de date (aceleași instrucțiuni se pot aplica, în paralel, la un număr mare de date) și iii) localizare de date. O descriere detaliată a acestui model se găsește în [38], iar fundamentele teoretice în [43].

În CUDA, unitatea fundamentală de execuție este *thread*-ul, definit de o funcție *nucleu* (kernel). Fiecare thread are asociat un indice, accesat prin variabilele speciale `threadIdx.x`, `threadIdx.y` și `threadIdx.z`.

Programatorul are libertatea de a-și organiza, din punct de vedere logic, threadurile în blocuri omogene uni-, bi- sau tri-dimensionale. Astfel, pentru operații matematice cu vectori, este natural a alege blocuri unidimensionale, pentru operații matriceale blocuri 2D, etc. În primul caz, variabila `threadIdx.x` va contine indicele linear, iar `threadIdx.y` si `threadIdx.z` = 0.

Blocurile de threaduri se organizează, la rândul lor într-o structură tip grilă (grid). Grila poate fi uni- sau bi-dimensională. Threadurile au acces indicele blocului curent (în interiorul grilei) prin variabilele `blockIdx.x`, `blockIdx.y`, iar dimensiunile blocului curent sunt date de `blockDim.x`, `blockDim.y`. Dimensiunea grilei (numărul de blocuri din grilă) este dată de `gridDim.x`, `gridDim.y`. Această organizare este prezentată în Figura 4.8.

În CUDA, funcția care definește nucleul unui thread se declară ca o funcție C având atributul **__global__**, de exemplu:

```
__global__ void numeFunctie(parametri ... )
{ ... }
```

Codul din interiorul unei funcții-nucleu este executat pe GPU cu următoarele restricții:

- Un nucleu poate apela doar funcții cu atributul **__device__**(dedicate execuției pe GPU).
- Se poate accesa doar memoria globală din dispozitiv (placa video).

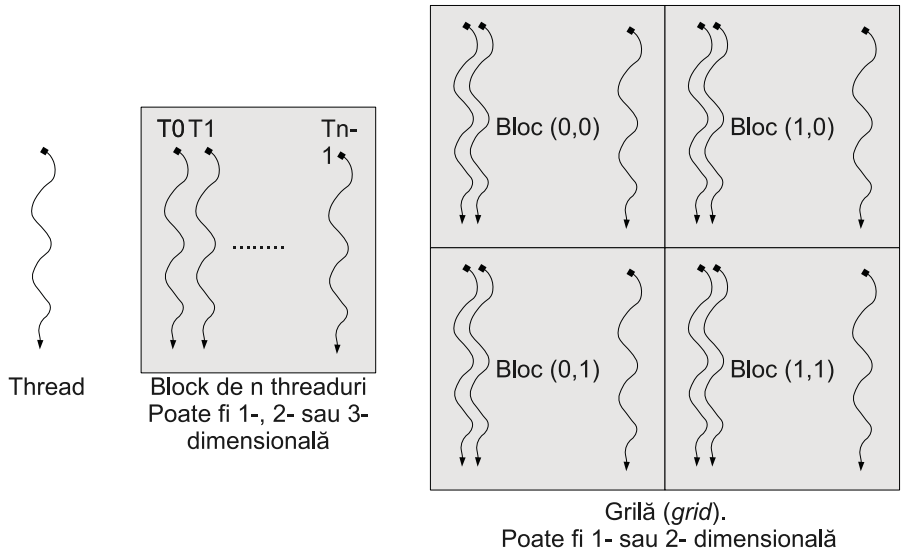


Figura 4.8: Organizarea threadurilor în CUDA.

- Pointerii definiți pe spațiul de adrese al gazdei (CPU) nu sunt interschimbabili cu pointerii dispozitiv (GPU)[15]. Pentru a aloca, transfera date între memoria gazdă (RAM) și dispozitiv se folosesc funcțiile CUDA API `cudaMalloc, cudaMemcpy, cudaMemset`.

- Codul GPU are la dispoziție o serie de funcții matematice pe virgulă mobilă (reprezentate prin `float` sau `double` conform standardului IEEE-754), ca $\sin, \cos, \exp, \ldots$ etc. Majoritatea acestor funcții sunt implementate sub forma unor blocuri de calcul direct în hardware (detalii în [30], [23]).

- Anterior CUDA 2.0 (Fermi), nu se putea folosi recursivitatea, nici pointeri la funcție.

Codul sursă, cu extensia .cu, conține o mixtură de declarații de variabile, funcții, unele destinate GPU, altele doar pentru CPU. Astfel, trebuie să acordăm atenție sporită acestor detalii. Compilatorul CUDA (nvcc) va genera din codul sursă două surse intermediare, una strict pentru gazdă, care va rula pe CPU și una pentru GPU care va rula pe procesoare grafice (acest detaliu este ascuns în compilarea uzuală).

Pentru funcții, avem la dispoziție următoarele atribute:

__device__ : funcție destinată GPU, poate apela doar alte funcții **__device__**;

[15] În versiunile recente de CUDA există operații cu *pinned memory*, zone care poate fi accesate dual, atât de către CPU cât și de GPU, copierea datelor fiind efectuată, în mod transparent, de către sistemul de operare.

__**global**__ : prezentat anterior - nucleul unui thread destinat GPU - este la rândul lui o funcție __**device**__specială;

__**host**__ : funcție destinată CPU (toate funcțiile declarate fără alte atribute, sunt considerate în mod implicit ca destinate CPU). Astfel, un program C clasic este compatibil cu sursa CUDA, dar va fi executat exclusiv pe CPU-ul mașinii gazdă;

__**device**__ __**host**__ : funcție dual ă. Compilatorul va genera o variantă pentru CPU, cât și o variantă GPU. Acest tip de funcție mărește puterea expresivității limbajului, pentru funcții cu surse identice programatorul nefiind nevoit să scrie două variante, una pentru GPU și alta pentru CPU. Acest atribut este utilă și pentru a verifica corectitudinea unei funcții, prin comparația rulării CPU și GPU.

Reamintim, în interiorul unui nucleu (cât și a funcțiilor device) sunt definite următoarele variabile speciale:

- **threadIdx.{x|y|z}** - indicele threadului, în cadrul blocului
- **blockDim.{x|y|z}** - dimensiunea blocului (numărul de threaduri, pe fiecare dimensiune)
- **blockIdx.{x|y|z}** - indicele blocului (în interiorul grilei)
- **gridDim.{x|y|z}** - dimensiunile grilei (numărul de blocuri, pe fiecare dimensiune).

4.2.4 Lansarea în execuție a threadurilor

Threadurile sunt lansate de către programul gazdă, după un *plan de execuție*, care specifică dimensiunile blocurilor și a grilei, cu ajutorul sintaxei următoare (o extensie a limbajului C):

```
numeNucleu <<< numarBlocuri, threaduriPerBloc >>> ( parametri ... );
```

unde numărBlocuri, threaduriPerBloc pot fi valori scalare (în cazul unidimensional) sau de tipul **dim3** (în care se pot specifica dimensiunile x,y,z).

Strategia de organizare în blocuri depinde de problema de rezolvat, trebuie ținut însă cont de faptul că numărul de threaduri dintr-un bloc (dimBloc.x*y*z) este limitat la 512 sau 1024, în funcție de tipul dispozitivului. Uzual se lucrează cu blocuri de dimensiune 256, 512 (multiplu de 32). Reamintim, threadurile dintr-un bloc au acces comun la o memorie partajată (*shared memory*), declarată cu atributul __**shared**__. Fiecare bloc are acces la o zonă distinctă, blocurile neputând inter-comunica, decât prin intermediul memoriei globale. Compilatorul alocă pentru variabilele locale din funcțiile nucleu regiștrii interni. Multiprocesorul dispune de un banc de 8192 - 32768 de regiștri de 32 biți, acesta fiind divizat, în mod egal între threadurile din același bloc. De exemplu, pentru 16K regiștri / 512 threaduri avem max. 32 regiștri / thread.

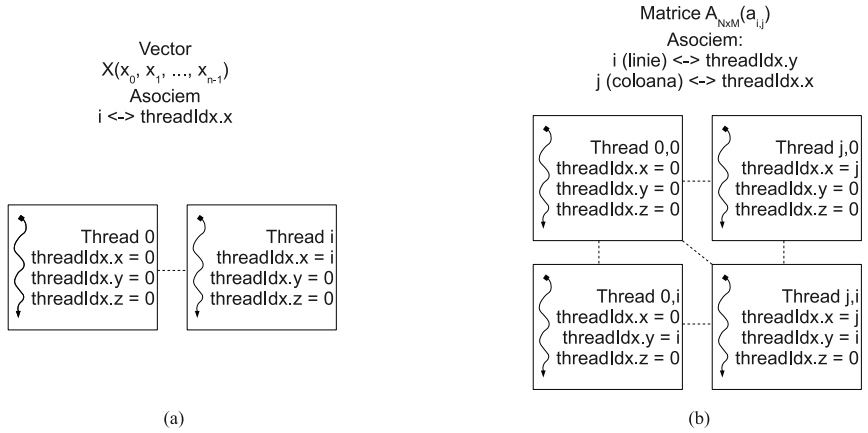


Figura 4.9: O posibilă asociere între threaduri și date- vector (a) și matrice (b)

În Figura 4.9 sunt prezentate două modalități de organizare a thread-urilor: sub formă de vectori (a) și sub formă de matrice (b).

De obicei, se caută încărcarea maximă a resurselor de calcul paralel, în cazul CUDA, aceasta înseamnând lansarea în execuție a multor blocuri. De exemplu în cazul procesării de imagini este uzual a lansa o grilă cu un număr total de 1.000.000 de threaduri.

Etapele unui program CUDA sunt, de regulă, următoarele:

1. alocarea memoriei pe dispozitiv,
2. copierea datelor de intrare din memoria gazdă în memoria dispozitiv,
3. apel nucleu CUDA: lansarea unei grile de threaduri,
4. calcul pe dispozitivul GPU,
5. sincronizarea gazdă-dispozitiv prin așteptarea terminării execuției tuturor threadurilor,
6. preluarea rezultatului din memoria dispozitiv.

Pașii 1-3,5-6 sunt executați de către calculatorul gazdă, iar pasul 4 este executat de procesorul grafic. În acest timp, procesorul gazdă (CPU) este inactiv sau poate fi programat pentru alte activități. Astfel, putem considera GPU ca un *co-procesor* care poate prelua secțiunile de calcul intensiv din program.

4.2.5 Compilarea exemplelor

Pentru elaborarea exemplelor care urmează s-a folosit mediul CUDA SDK versiunea 4.1 [31] care conține compilatorul CUDA, bibliotecile și fișierele header aferente.

Sursele de cod CUDA se găsesc în anexa A şi se pot descărca, împreună cu fișierele proiect pentru Microsoft Visual Studio 2008 si 2010, de la adresa `http://miv.unitbv.ro/asd`. Fiecare exemplu are un fișier sursă corespunzător, `ex1.cu` ... `ex9.cu`.

Sub sistemul de operare Windows s-a folosit mediul de dezvoltare Microsoft Visual Studio 2008 [26], creând un fișier proiect fiecărei lucrări în parte (`ex1.vcproj` ... `ex9.vcproj`). Executabilele rezultate (`ex1.exe` ... `ex9.exe`) sunt depuse în subdirectorul bin/ împreună cu dependințele.

Sub Linux se compilează folosind comanda `nvcc` (NVidia Cuda Compiler), într-un terminal:

```
$ nvcc -o ex1 ex1.cu
```

care va compila sursa 'ex1.cu', producând executabilul 'ex1' sau prin comanda `make`, care va compila toate exemplele.

4.3 Primul program CUDA

Scopul acestui exemplu este de a ne familiariza cu mediul CUDA, compilarea unui program, lucrul cu memoria dispozitiv, modelul de execuție multi-threaded divizat în blocuri.

În loc de tradiționalul "*Hello World*", primul exemplu va consta din stocarea în memorie a șirului de întregi 0, 1, 2, ... N-1, echivalentul paralel al următorului cod C:

```
for (i=0; i<N; i++)
        a[i]=i;
```

Chiar și pentru acest exemplu simplu, este nevoie de următorii pași:

1. alocarea memoriei pe dispozitiv și gazdă (pointerii a_gpu, respectiv a_cpu)
2. apelul CUDA : Lansarea a N threaduri. Fiecare thread are asociat un indice unic, intrinsec, de la 0 la $N-1$. Pseudo-codul este:

   ```
   Thread(i) executa:
       a[i]=i;
   terminare
   ```

3. așteptarea terminării tuturor threadurilor
4. copierea rezultatului din memoria dispozitiv în memoria gazdă[16]
5. afișarea șirului rezultat.

În pașii 2–3, execuția se desfășoară pe procesorul grafic, iar procesorul gazdă așteaptă rezultatele în procedura `cudaThreadSynchronize`(). În continuare, vom prezenta codul unei posibile implementări CUDA, folosind un vector de threaduri.

[16]Copierea are loc, de obicei, pe magistrala pe care se află placa video, iar viteza acesteia limitând performanța sistemului, se caută minimizarea numărului de transferuri.

```
// Codul care va rula pe GPU
__global__ void threadScriere(int * a, int n) {
        // indicele threadului
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < n)
                a[i] = i;
}
// functia main() din C standard, se executa pe CPU
int main(int argc, char ** argv) {
  int n = 1024; // dimensiune
  int *a_cpu;   // pointer la memoria gazda
  int *a_gpu;   // pointer la memoria grafica
  // alocare memorie gazda
  a_cpu = (int*) calloc(n, sizeof(int));
  // alocare memorie grafica
  cudaMalloc((void**) &a_gpu, n * sizeof(int));
  int threadsPerBlock = 256; // dimensiune bloc
  int blocksPerGrid = (n + threadsPerBlock - 1)
             / threadsPerBlock; // numar blocuri

  // apelare GPU - lansare threaduri:
  threadScriere <<<blocksPerGrid,threadsPerBlock>>>
                      (a_gpu, n);
  cudaThreadSynchronize(); // asteptare rezultat
  cudaMemcpy(a_cpu, a_gpu, n * sizeof(int),
              cudaMemcpyDeviceToHost); // copiere mem
  for (int i = 0; i < n; ++i)
      printf("%d ", a_cpu[i]); // afisare
  printf("\n");
  cudaFree(a_gpu); // eliberare memorie
  free(a_cpu);
  return 0;
}
```

În codul de mai sus, întâlnim următoarele elemente specifice CUDA:

__global__ declară o funcție nucleu (kernel), ce urmează a fi executată pe GPU.

cudaMalloc() alocă un bloc în memoria globală dispozitiv (memoria plăcii grafice), corespondentul lui `malloc()` din limbajul C standard.

cudaFree() eliberează memoria alocată de cudaMalloc().

<<<nrBloc, nrThread>>> invocă execuția unui nucleu, pe GPU, organizat în $nrBloc$ blocuri a câte $nrThread$ threaduri paralele.

threadIdx.x Indicele threadului, în interiorul blocului, ia valori cuprinse între 0 și $blockDim - 1$ Aceasta variabilă specială este definită doar în interiorul funcțiilor executate pe GPU.

blockDim.x Dimensiunea blocului (numărul de threaduri).

blockIdx.x Indicele blocului curent.

cudaThreadSynchronize așteaptă terminarea procesării GPU (tuturor threadurilor din GPU), deoarece lansarea threadurilor în execuție, cu operația

<<<>>> este asincronă - în timp ce rulează GPU, procesorul gazdă este liber în a efectua alte calcule. Prin `cudaThreadSynchronize`, execuția CPU se 'blochează' până când toate threadurile GPU s-au terminat. După terminarea acestui apel știm că rezultatele sunt disponibile în memoria globală GPU.

cudaMemcpy() transfer de memorie între dispozitiv și gazdă, direcția fiind dată de ultimul parametru: `cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToDevice`

Cele n=1024 de threaduri din exemplul mai sus se grupează în 4 blocuri de câte 256. Prin formula `i = blockDim.x * blockIdx.x + threadIdx.x`, thread-ul îsi calculează indicele asociat, din indicele relativ la interiorul blocului (`threadIdx.x`), indicele blocului curent (`blockIdx.x`) și dimensiunea blocurilor (`blockDim.x`), calcul ilustrat în Figura 4.10.

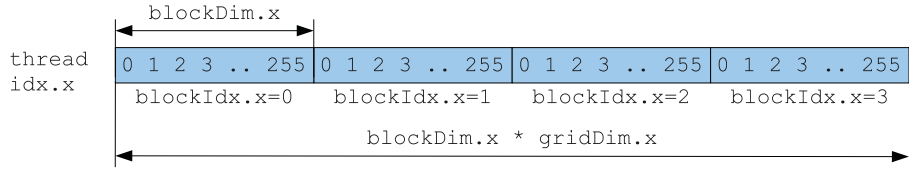


Figura 4.10: Organizarea threadurilor în blocuri, cazul 1-dimensional.

Sursa completă a acestui exemplu se găsește în fișierul `ex1.cu` la Pagina 97.

Exerciții

4.1 Măsurați timpul de execuție a nucleului (de la invocare până la terminarea `cudaThreadSynchronize`), precum și timpul necesar copierii datelor din memoria dispozitiv în memoria gazdă (cudaMemcpy). Pentru a măsura timpul, în microsecunde, folosiți funcția `getTime()` aflată în fișierul header `labTimer.h` la Pagina 132. Calculați ratele de transfer de memorie, în Gigabytes/sec. Ce observați? Comparați datele măsurate cu specificațiile dispozitivului (plăcii video).

4.4 Adunarea a doi vectori

În continuare vom prezenta exemplul adunării a doi vectori **a** și **b**, rezultatul operației fiind stocat în vectorul **c**. Conținutul vectorilor va fi generat în mod aleator, folosind funcția (sistem-gazdă) `rand()`. Exemplul, varianta CUDA a problemei punerii în frigider a unui elefant din 3 mișcări, constă în:

1. copierea vectorilor de intrare din memoria gazdă în memoria dispozitiv;
2. efectuarea calculului;

3. transferul rezultatului din memoria dispozitiv în memoria gazdă, pentru afișare.

Similar cu exemplul anterior, se asignează un thread pentru fiecare poziție a elementelor din vectorii de intrare. Operația elementară, efectuată de fiecare thread este:

```
// Codul care va rula pe procesorul grafic
__global__ void addV(float * c, const float * a,
                           const float * b, int n)
{
int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
    c[i] = a[i] + b[i];
}
```

Nucleul se apelează din programul-gazdă prin fragmentul de cod

```
addV <<< blocksPerGrid, threadsPerBlock >>>
      (c_gpu, a_gpu, b_gpu, n); // apelare GPU

cudaThreadSynchronize(); // asteptare rezultat
```

Sursa completă se găsește în fișierul `ex2.cu` la Pagina 98. În acest exemplu, am introdus două funcții ajutătoare, `allocDual()` și `freeDual()`. Rolul lor este de a aloca un vector în memoria gazdă, cât și un corespondent în memoria dispozitiv, pentru a scurta codul funcției `main()`.

Copierea din memoria gazdă în memoria dispozitivului poate dura mai mult decât calculul propriu-zis. În acest caz, performanța programului va fi limitată de lărgimea de bandă a magistralei memoriei (I/O Bound). Din acest motiv, se evită copierea redundantă dintre CPU-GPU a datelor.

Avantajul GPU apare la un volum mare de date mari (deoarece trebuie încărcate toate unitățile de execuție, de ex. 8×512). Pentru o problemă de dimensiuni mici (de exemplu, adunarea vectorilor de 100 de elemente) nu se justifică folosirea CUDA, problema se poate rezolva mai eficient pe CPU.

Exerciții

4.2 Implementați varianta secvențială a adunării vectorilor pe CPU (sau altă operație matematică) folosind limbajul C și comparați timpii de execuție CPU, respectiv GPU.

4.3 Determinați în mod experimental valoarea minimă a dimensiunii n pentru care implementarea paralelă folosind CUDA este mai eficientă decât implementarea secvențială pe CPU (se poate considera și cazul favorabil, în care nu este nevoie de copierea datelor).

4.5 Însumarea elementelor unui vector

În continuare vă prezentăm exemplul însumării, în paralel, a elementelor unui vector, operație ce poartă numele de "reducție paralelă", prezentată și în secțiunea 4.1.

În exemplele anterioare, nu exista dependență de date între threaduri, fiecare operând doar cu elementul asignat în vector. Pentru a calcula suma elementelor unui vector este nevoie însă de a comunica rezultatele parțiale.

Pentru a calcula suma $x[0] + x[1] + ... + x[N-1]$ în paralel, având $N/2$ procesoare, vom folosi următorul pseudocod:

for $s \leftarrow N/2; s > 0; s \leftarrow s/2$ **do**
  for $i \leftarrow 0, \ldots, s-1$ **do in parallel**
    $x[i] \leftarrow x[i] + x[i+s]$
  end for parallel
end for

Pseudocodul conține $\log(N/2)$ pași secvențiali, iar la fiecare pas numărul de procesoare active se înjumătățește (Figura 4.11).

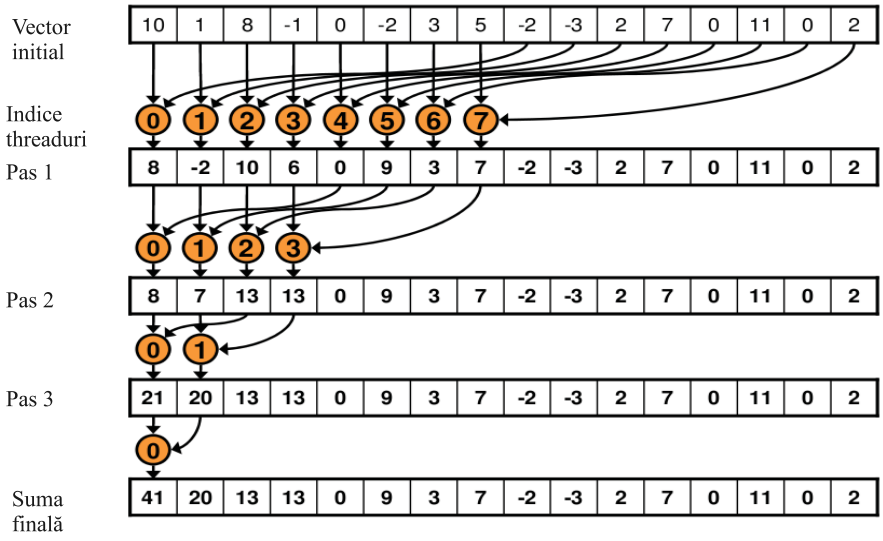


Figura 4.11: Calculul sumei paralele (reproducere după [17]).

Reducția paralelă este aplicabilă oricărui operator asociativ. De exemplu, dacă înlocuim adunarea cu operația $min()$ sau $max()$, putem afla elementul minim sau maxim dintr-un șir.

Timpul de execuție al algoritmului de sumare paralelă, al unui vector de

lungime N, pe P procesoare, este:

$$T_{sp} \in O\left(\frac{N}{P} + \log P\right)$$

în cazul $N = P$ (cel prezentat mai jos), timpul devine $T_{sp} \in O(\log N)$.

În continuare este prezentat nucleul CUDA care rezolvă problema prezentată.

```
__global__ void parallelSum(float * x, int n)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  for (int s=n/2; s>0; s/=2){
    if (i < s)
       x[i] += x[i+s];
    __syncthreads();
  }
}
```

Acest nucleu se apelează din programul-gazdă prin fragmentul de cod

```
parallelSum <<<blocksPerGrid, threadsPerBlock>>>(x_gpu, n);
```

Sursa completă se află în fișierul `ex3.cu` la Pagina 99.

4.5.1 Accesul la memoria globală

În arhitectura CUDA, accesul la memoria globală se face (de către controller, transparent pentru programator), prin transferuri de câte 32, 64 sau 128 octeți. Performanța optimă se obține dacă threadurile accesează memoria globală în paralel după o regulă ordonată, de exemplu: dacă 32 threaduri, cu indicele $0, 1...31$ scriu într-un tablou cu elemente tip `float` sau `int`, (dimensiunea elementelor fiind de octeți) la indicele $k, k+1, ...k+31$, atunci cei 32*4 octeți vor fi trimiși pe magistrala de memorie într-o singură tranzacție. Operația se numește fuziune de acces (*coalescing*). În schimb, dacă accesul este ne-ordonat (de ex: $k + 9, k + 1, k + 3, k + 2, ...$), atunci operația va necesita mai multe transferuri (și implicit mai multe unități de ceas). Pentru regulile de acces, consultați [30], secțiunea 5.3 - Global Memory.

În Figura 4.11 se observă accesul ordonat la elemente, pentru a optimiza transferul de memorie.

4.5.2 Bariera de sincronizare

În programarea paralelă, câteodată este nevoie ca procesele sau firele de execuție concurente (care se pot executa cu viteze diferite) să se aștepte reciproc, la anumite puncte, definite de programator. Astfel, bariera este o instrucțiune (primitivă de sincronizare) care va bloca firele de execuție până când toate ajung la ea. După ce toate procesele ajung la această instrucțiune, bariera se deblochează, reluând execuția tuturor proceselor.

Totodată, bariera impune o ordonare (secvențială) a instrucțiunilor: toate instrucțiunile declarate în codul sursă înaintea barierei se vor executa, în mod

garantat înaintea barierei. Fără această restricție, compilatorul, în faza de optimizare poate reordona instrucțiunile.

În CUDA, avem două tipuri de bariere: Primul tip este funcția-gazdă `cudaThreadSynchronize()`, prezentată în exemplele anterioare, care sincronizează calculatorul-gazdă cu procesorul grafic, totodată garantează terminarea tuturor threadurilor dintr-o grilă.

Al doilea tip, definit prin funcția dispozitiv **__syncthreads**(), are efect local, doar în cadrul unui bloc de threaduri. Threadurile dintr-un bloc care ajung la aceasta instrucțiune se vor opri, așteptând ca toate threadurile din același bloc să ajungă la același punct, reluându-și toate execuția în mod sincron. Threadurile aflate în blocuri diferite nu se pot sincroniza prin aceasta metodă. Este nevoie de această 'barieră' pentru a garanta ordinea execuției instrucțiunilor și a operațiilor de scriere/citire din memorie.

În exemplul de mai sus, suma paralelă se calculează în interiorul blocului, limitând lungime maximă a vectorului la 2*nr. max. threaduri/bloc. Pentru a aduna vectori mai lungi, trebuie folosită o tehnică combinată cu sincronizarea globală. O tratare detaliată, împreuna cu posibile optimizări este prezentată în [17].

Exerciții

4.4 Măsurați timpul de execuție al programului prezentat.

4.5 Modificați nucleul CUDA astfel încât, în loc de suma elementelor, să calculeze elementul minim al unui vector (sau orice altă operație asociativă, la alegere).

4.6 Generalizați implementarea pentru a suma elementele unor vectori de dimensiuni ce depășesc cadrul unui bloc. (Ajutor: la primul pas, un thread poate suma N/p elemente, unde p este numărul de threaduri din bloc).

4.7 Calculul valorii lui π. Numărul π poate fi aproximat prin seria:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots .$$

Implementați nucleul CUDA care calculează primii N termeni ai seriei și îi sumează prin reducție paralelă.

4.6 Înmulțirea matricelor

Scopul acestui laborator este de a arăta importanța organizării datelor în memorie, precum și lucrul cu memoria partajată CUDA (shared memory). Deși exemplul este specific CUDA, tehnica de optimizare paralelă prin descompunerea matricelor în blocuri este generică.

4.6.1 Matrice

Pentru a ușura lucrul cu matrice, precum a copia în/din memoria dispozitiv, a fost creată clasa Matrix, în fisierul `"labMatrix.h"`, listată la Pagina 130. Această clasă este generică, `T` denotând tipul de date al elementelor. O matrice ce conține valori reale de tip `float` de N linii și M coloane se declară prin:

```
Matrix<float> A(M, N);
```

Funcția `subMatrix()` creează o referință la un bloc din matricea principală (fără a copia elementele).

Cu ajutorul operatorului [] (supra-definit) putem accesa elementele unei matrice **A** printr-o sintaxă obișnuită: A[i][j], unde i,j sunt este indicele liniei, respectiv a coloanei.

4.6.2 Cazul înmulțirii secvențiale

Cel mai simplu algoritm secvențial de înmulțire a două matrice, $\mathbf{C}_{N\times P} = \mathbf{A}_{N\times M} * \mathbf{B}_{M\times P}$ este:

```
for (i = 0; i < N; i++) {
  for (j = 0; j < P; j++) {
    C[i][j]=0;
    for (k = 0; k < M; k++)
      C[i][j] += A[i][k] * B[k][j];
   }
 }
```

Observăm că, pentru a calcula un element $\mathbf{C}[i][j]$, avem nevoie de:

- întreaga linie i din matricea **A**
- întreaga coloană j din matricea **B**
- $\mathbf{C}[i][j]$ nu depinde de rezultate parțiale din alte poziții ale matricei **C**.

4.6.3 Cazul paralel

Pentru calcularea fiecărui element $\mathbf{C}[i][j]$ putem atribui un thread, după cum urmează:

```
template<class T>
__global__ void matMulSimpleKernel(Matrix<T> C,
             const Matrix<T> A, const Matrix<T> B) {
  int i = threadIdx.y + blockIdx.y * blockDim.y;
  int j = threadIdx.x + blockIdx.x * blockDim.x;
  float sum = 0;
  for (int k = 0; k < A.width; ++k)
        sum += A[i][k] * B[k][j];
  C[i][j]=sum;
}
```

Pentru a lansa în execuție nucleul `matMulSimpleKernel`, se apelează din codul-gazdă astfel:

```
const int N=1024;
dim3 threads(16,16);
dim3 grid(N/16,N/16);
matMulSimpleKernel <<< grid, threads >>> (C,A,B);
```

Pentru simplitate am considerat matrice pătratice, de aceeași dimensiune, N folosind blocuri de 16×16 threaduri și am ales N multiplu de 16 (pentru a nu fi nevoie de tratarea specială a blocurilor marginale). Timpul de execuție paralel (pentru P procesoare) este:

$$T_{mm} \in O(N^3/P)$$

Acest cod însă este sub-optimal din punct de vedere al accesului de memorie, deoarece:

- accesul la coloanele din $\mathbf{B}$ implică citiri cu pas mare în memorie (elementele de pe aceeași coloană fiind la distanță mare).
- la sistemele de calcul (secvențiale sau paralele) prevăzute cu memorie rapidă (cache), dacă dimensiunile matricelor depășesc dimensiunea memoriei cache, algoritmul de mai sus va utiliza ineficient cache-ul, nerespectând principiul localizării datelor în memorie.

Din fericire, putem reformula algoritmul de înmulțire matriceală, ca produs de sub-matrice. Astfel, dacă descompunem matricele $\mathbf{A}$ și $\mathbf{B}$ în sub-matrice de dimensiune redusă, atunci produsul $\mathbf{C}$ poate fi formulat ca suma de produse de sub-matrice.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1s} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{q1} & \cdots & \mathbf{A}_{qs} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \cdots & \mathbf{B}_{1r} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{s1} & \cdots & \mathbf{B}_{sr} \end{bmatrix}$$

$$\mathbf{C}_{IJ} = \sum_{I=1}^{s} \mathbf{A}_{IK}\mathbf{B}_{KJ}.$$

unde prin I, J, K am notat indicele sub-matricelor.

Această descompunere, din punct de vedere matematic este echivalentă cu prima formulare și are aceeași complexitate algoritmică $O(N^3)$. Avantajul îl reprezintă faptul că putem alege dimensiunea sub-matricelor astfel încât să se încadreze în dimensiunea memoriei rapide.

În CUDA, fiecare bloc de threaduri are la dispoziție o zonă de memorie rapidă, care se declară (în interiorul funcției nucleu) cu atributul **__shared__**. Această memorie nu este accesibilă de procesorul gazdă, nu este adresabilă decât din blocul de care aparține și nu este persistentă între mai multe invocări de threaduri. De obicei, în memoria rapidă se încarcă datele la începutul threadului, copiind din memoria globală, devenind astfel un fel de memorie cache cu pre-fetch explicit.

Nucleul de înmulțire pe blocuri de dimensiune $D \times D$ este următorul:

```
template < class T, unsigned D > __global__ void
matMulBlockKernel(Matrix < T > C, Matrix < T > A, Matrix < T > B)
{
  __shared__ T As[D][D]; // bloc in memoria partajata, rapida
  __shared__ T Bs[D][D];

  int bi = blockIdx.y*D; // indice inceput bloc (bi,bj)
  int bj = blockIdx.x*D;

  int i = threadIdx.y;   // indice relativ la blocul curent
  int j = threadIdx.x;

  // fiecare thread va calcula un element din C, prin
  // acumularea rezultatului partial in variabila c
  T c = 0;               // compilatorul CUDA va aloca un registru pt Cvalue

  // Itereaza sub-matricile lui A si B
  for (int k = 0; k < A.columns(); k += D) {
    // fiecare thread copiaza un element din memoria globala in memoria rapida
    As[i][j] = A[bi+i][k+j];
    Bs[i][j] = B[k+i][bj+j];

    __syncthreads(); // sincronizare, ne asiguram completarea copierii

    // Multplicare blocuri As * Bs
    for (int p = 0; p < D; p++)
      c += As[i][p] * Bs[p][j];
  }

  // Salvam rezultatul in memoria globala
  C[bi+i][bj+j] = c;
}
```

În nucleul de mai sus observăm două bucle. Bucla externă `for (k ...)` iterează pe fiecare submatrice `As` din linia-bloc curentă (bi) si `Bs` din coloana-bloc curentă (bj), iar bucla internă realizează înmulțirea blocului `As` cu `Bs`, folosind doar argumente din memoria partajată rapidă (având latența de aproximativ $T_{Sh} \approx 38$ cicli/instrucțiune). Astfel, în zona critică a algoritmului am evitat accesul la memoria globală (având latența de aproximativ $T_{Gl} \approx 400$ — 800 cicli/instrucțiune). În bucla internă, fiecare intrare din matricile bloc se refolosește de D ori.

În codul sursă complet (`ex4.cu`), aflat la Pagina 101, găsim 3 implementări:

- **Algoritmul 0** - calcul serial, CPU, algoritmul clasic,
- **Algoritmul 1** - CUDA, implementare naivă,
- **Algoritmul 2** - CUDA, înmulțire de matrici prin descompunere în blocuri.

De notat, aceste implementări au rol introductiv în tehnici de optimizare CUDA. Pentru aplicații reale, care au nevoie de înmulțirea matricelor mari, recomandăm folosirea bibliotecilor gen `CUBLAS` (CUDA Basic Linear Algebra Subroutines) ce oferă rutine de algebră lineară optimizate. Pentru tehnici de optimizare CUDA avansată consultați [45].

Exerciții

4.8 Rulați cei 3 algoritmi (pe matrice de dimensiune 1024^2 sau mai mari), măsurând timpii de execuție. Ce observați?

4.9 Determinați experimental pentru ce dimensiuni N implementarea paralelă în CUDA devine mai rapidă decât cea secvențială. Dar față de implementarea paralelă pe CPU multi-core?

4.10 Implementați în CUDA algoritmul Floyd-Warshall de determinare a celor mai scurte drumuri dintr-un graf. Algoritmul este descris în [2] cap. 8.5; vă reamintim algoritmul secvențial pentru un graf format din N noduri:

```
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
     for (j=0; j<N; j++)
          C[i][j] = min ( C[i][j], C[i][k]+C[k][j] );
```

unde matricea C de dimensiune $N \times N$ este inițializată cu: $C(i,i) = 0$, $C(i,j) = \infty$ dacă nu există arc între nodurile i, j sau $C(i,j) = cost(i,j) >= 0$ dacă există arc (legătură directă) între nodurile i, j. După rularea algoritmului, $C(i,j)$ va conține costul celui mai scurt drum dintre nodurile i, j.

4.11 Implementați în CUDA un algoritm eficient de transpunere a unei matrice pătratice mari (de dimensiuni ce depășesc considerabil memoria partajată).

Capitolul 5

Aplicații în CUDA

În acest capitol vom prezenta câteva aplicații CUDA în domeniul procesării de imagini și al bio-informaticii: folosirea texturilor grafice, interoperabilitatea cu OpenGL, folosirea transformatei Fourier, biblioteca de programare generică Thrust și alinierea secvențelor.

5.1 Texturi și imagini color

Domeniul "nativ" al aplicațiilor CUDA este procesarea imaginilor. Acest domeniu depășește cadrul laboratorului; ne vom rezuma doar la câteva exemple cu rol introductiv. Scopul acestei lucrări este de a prezenta câteva exemple mai complexe de programe CUDA, exemplificând în mod special procesarea cu texturi și imagini.

5.1.1 Procesarea texturilor

Textura – un termen împrumutat din grafica 3D – poate fi considerată o matrice uni-, bi- sau tri-dimensională de *texeli* (texture-elements). Texelii pot fi reprezentați prin scalari (byte, float) sau 4-tuple (byte4, float4).

În CUDA, texturile se disting ca o zonă de memorie specială, care poate fi citită cu ajutorul unor funcții de acces speciale `tex1D(x)`, `tex2D(x, y)`, respectiv `tex3D(x, y, z)`. Texturile oferă următoarele facilități:

- pentru dispozitivele mai vechi, citirea din memoria de textură este mai rapidă decât accesul din memoria globală dispozitiv[1];
- se pot citi și elemente de la coordonate ne-întregi, interpolarea (liniară) a valorilor efectuându-se de către hardware (de ex: `float a = tex2D(1.5, 3.25)`);

[1] Arhitectura CUDA 2.x (Fermi) oferă memorie cache și memoriei globale.

- coordonatele care depășesc domeniul texturii $[0 \dots N-1]$ se ajustează automat, fie forțându-le la marginile domeniilor $0, N-1$ fie calculând modulo N, configurabil;

- un dezavantaj major al texturilor este că nu pot fi scrise din nuclee CUDA. Texturile se inițializează prin funcții API speciale de către calculatorul gazdă.

Pentru a contracara acest ultim dezavantaj, în arhitectura CUDA 2.0 (Fermi) au fost introduse *suprafețele* (surfaces), zone de memorie asemănătoare texturilor, cu proprietatea că pot fi scrise dinamic din nuclee.

Pentru a lucra cu imagini color în cadrul exemplelor ce urmează, s-a creat clasa `Image`, a cărui cod sursă se află în fișierul `"labImage.h"`, listat la Pagina 126. Această clasă conține un pointer atât către memoria gazdă, cât și un pointer către memoria dispozitiv. La început, imaginea se încarcă din fișier în memoria gazdă (RAM), prin funcția `load`[1], la adresa dată de pointerul `data`, după care se copiază (prin funcția `deviceToHost`) în memoria video (pentru a fi prelucrată ulterior), la adresa dată de pointerul `data_gpu`. După terminarea procesării, rezultatul aflat în memoria video se copiază înapoi în memoria dispozitiv (prin funcția `hostToDevice`) și se salvează pe disc prin funcția `save`. Clasa `Image` nu face parte din biblioteca CUDA, fiind creată pentru a simplifica lucrul cu imagini în cadrul lucrărilor de laborator. Sursa completă se găsește în fișierele labImage.h și labImage.cpp, în pachetul de surse dezvoltat pentru laboratorul de față.

Procesoarele grafice lucrează în mod nativ cu tipul de dată **float**, iar din motive de performanță este preferat accesul de memorie în unități multiplu de 4. Din acest motiv, vom instanția clasa Image <**float**, 4>, un mod natural pentru procesoarele grafice.

În CUDA există tipul struct **float**4 {**float** x,y,z,w}, echivalent cu un vector **float**[4] din C. Astfel, dacă asociem o structură **float**4 pentru un pixel din imagine, în spațiul de culori RGB, asignăm x = roșu, y = verde, z=albastru, w=α. Fiecare componentă are domeniul normalizat de valori de la 0.0 (intensitate nulă) până la 1.0 (intensitate maximă). Ultimul parametru (canalul α) este folosit la imagini cu semi-transparență, în cursul acestui laborator vom considera $\alpha = 1.0$ (opacitate). Atât în CUDA, cât și în clasa Image, spațiul de culori nu este restrâns la modelul RGB. Pentru laborator s-a ales acest model pentru interoperabilitatea cu OpenGL și cu formatul de fișier PNG.

Astfel, tabloul **float** data[w*h*N] va avea structura dată în Figura 5.1, unde w,h și N sunt lățimea (width), înălțimea (height), respectiv numărul de componente de culoare ale imaginii.

În exemplul următor vom demonstra capabilitatea de interpolare liniară a funcției `tex2D`, mărind o imagine:

```
// referinta la textura
texture<float4, 2, cudaReadModeElementType> texRef;
```

[1]Pentru a încărca sau salva imagini din fișiere cu formatul PNG, s-a folosit biblioteca open-source FreeImage `http://freeimage.sourceforge.net/`.

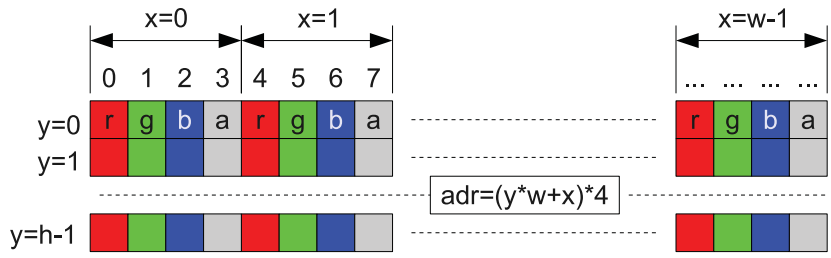


Figura 5.1: Reprezentarea în memorie a unei imagini RGBA.

```
__global__
void rescaleKernel(float4 * output,
                           size_t stride,
                           int width, int height,
                           float zx, float zy)
{
  // indicele thread = coordonate destinatie
  int x=threadIdx.x + blockDim.x * blockIdx.x;
  int y=threadIdx.y + blockDim.y * blockIdx.y;
  // Fiecare thread este asignat exact unui
  // pixel din destinatie (x,y)
  // dar poate accesa orice pixel sursa
  if (x < width && y < height){
    float4 f;
    float sx = zx*x;           // coordonate sursa
    float sy = zx*y;
    f = tex2D(texRef, sx, sy); // acces textura
    // stocare rezultat
    output[x+y*(stride/sizeof(float4))] = f;
  }
}
```

În nucleul `rescaleKernel` atribuit în parte fiecărui pixel din imaginea transformată (destinație), se calculează coordonatele (x, y) pe baza indicelui relativ la bloc (threadIdx.x, threadIdx,y) și coordonatele blocului (blockIdx.x, blockIdx.y), urmate de coordonatele (s_x, s_y) din imaginea sursă (Figura 5.2). Transformarea de coordonate pentru scalare este liniară:

$$\begin{aligned} s_x &= x \cdot z_x \\ s_y &= y \cdot z_y \end{aligned}$$

unde (s_x, s_y) sunt coordonatele pixelului-sursă, iar (z_x, z_y) sunt factorii de scalare (zoom factor).

Testul if (x <width ...) este necesar pentru a nu depăși marginile imaginii destinație, din cauza organizării threadurilor în blocuri cu dimensiune fixă.

S-a ales o dimensiune uzuală a blocului ($16 \times 16 = 256$) , numărul de threaduri / bloc fiind limitat de tipul dispozitivului grafic. O dimensiune prea mică

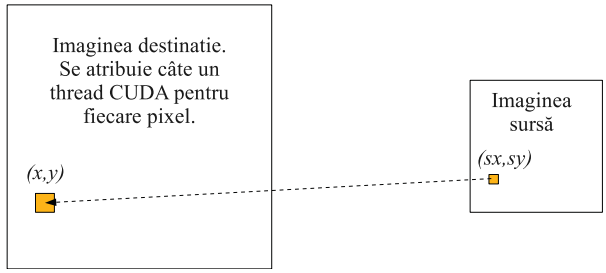


Figura 5.2: Scalarea imaginii de către `rescaleKernel`

poate duce la sub-utilizarea multiprocesoarelor.

Instrucțiunea **float**4 f = `tex2D`(texRef, sx, sy) citește elementul de textură de la coordonatele (s_x, s_y), toate componentele de culoare (r, g, b, a) fiind interpolate în paralel. Codul sursă se află în fișierul `ex5.cu`, listat la Pagina 107. La compilare și rulare este nevoie de biblioteca FreeImage [13] pentru a manipula imagini de tip PNG.

Exerciții

5.1 Modificați exemplul anterior pentru a implementa alte transformări geometrice, de exemplu: inversări, oglindiri, rotiri (parametrul param.x poate da unghiul, variabil prin mouse), sau transformări pe pixel: luminozitate (pixel + param), contrast (pixel * param), negativul (1-pixel) sau o altă funcție arbitrară.

5.2 Interoperabilitatea cu OpenGL

OpenGL – Open Graphics Library – este biblioteca standard, portabilă pentru a programa grafica 3D pe calculatoare. Acest laborator nu va detalia vasta funcționalitate a OpenGL [29], doar câteva puncte de intersecție cu extensia CUDA.

Pentru a inițializa interoperabilitatea OpenGL-CUDA, la începutul programului trebuie apelată funcția `cudaGLSetGLDevice`(deviceId). Primul dispozitiv CUDA se identifică cu deviceId=0.

În general, în OpenGL resursele (texturi, obiecte de tip buffer) sunt identificate printr-un întreg. Pentru a fi accesibile din CUDA, este nevoie de a le înregistra prin apelul `cudaGraphicsGLRegisterBuffer`().

Programul prezentat va încărca o imagine de pe disc, o va transfera în memoria dispozitiv și va apela o procesare folosind CUDA.

Spre deosebire de exercițiul precedent, imaginea prelucrată nu va fi copiată înapoi în memoria gazdă, ci afișată direct pe ecran, din memoria dispozitiv-video. Astfel, prin CUDA - OpenGL avem avantajul de a crea programe performante de prelucrare de imagini și afișare în timp real, pentru rezoluții mari.

De exemplu, transformarea geometrică din Figura 5.3 a fost efectuată de nucleul descris în continuare:

```
__global__
void imageProcessingKernel(float4* output, int stride,
                int width, int height, float4 param)
{
  // indicele thread = coordonate destinatie
  int x=threadIdx.x + blockDim.x * blockIdx.x;
  int y=threadIdx.y + blockDim.y * blockIdx.y;
  if (x < width && y < height){
        float dx=x-width/2.0F; float dy=y-height/2.0F;
        float r=sqrt(dx*dx+dy*dy)/100.0F;
        float sx=x + r*cos(r*param.x);
        float sy=y + r*sin(r*param.y);
        float4 f = tex2D(texRef, sx,sy);
        output[x+y*(stride/sizeof(float4))] = f;
        }
}
```

(a) Imaginea originală (b) Rezultatul transformării

Figura 5.3: "Lenna" nu poate lipsi din exemplele de procesări de imagini. Iată distorsiunea de "unduire" (*ripple effect*) realizat de codul din exemplul prezentat anterior.

Nucleul de mai sus implementează transformarea geometrică inversă

$$\begin{aligned} s_x &= x + r\cos(r\theta) \\ s_y &= y + r\sin(r\phi) \end{aligned}$$

unde (s_x, s_y) sunt coordonatele punctelor din imaginea sursă, (x, y) sunt coordonatele punctelor din imaginea destinație, r este distanța punctului față de centrul imaginii, iar θ și ϕ sunt parametri. De notat transformarea inversă: pornind din coordonatele punctului destinație, determinăm coordonatele punctului sursă.

Nucleul CUDA este apelat dintr-un program interactiv, OpenGL. Prin mișcarea mouse-ului se modifică parametrii `param.x` și `param.y`, variind efectul. Calculul fiind inclus în bucla de afișare OpenGL – GLUT[2], imaginea destinație se va re-calcula pentru fiecare cadru video. Viteza de afișare (*fps*, cadre pe secundă) depinde atât de caracteristicile plăcii video, cât și de setările dispozitivului (dacă s-a activat opțiunea V-Sync, afișarea va fi sincronizată cu frecvența verticală a monitorului).

Sursa completă se află în fisierul `ex6.cu` la Pagina 108. Funcția `compute`() este responsabilă cu apelul nucleului CUDA și preluarea rezultatelor. Destinația se află în memoria dispozitiv, dar de data aceasta nu a fost alocată prin apelul `cudaMalloc`, ci din OpenGL, la inițializarea programului.

Pentru a obține un pointer către memoria dispozitiv, pentru CUDA, se apelează `cudaGraphicsResourceGetMappedPointer`(). După prelucrarea, prin CUDA, pointerul trebuie detașat prin `cudaGraphicsUnmapResources`(), funcție care nu dealocă memoria, doar rupe asocierea cu pointerul (acesta devenind invalid). Secvența de operații este ilustrată în următorul fragment de cod:

```
// ascoiere textura sursa
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc < float4 > ();
cudaBindTexture2D(0, texRef,
                          sourceImage->data_gpu, channelDesc,
                          sourceImage->columns(), sourceImage->rows(),
                          sourceImage->stride_gpu);

// asociere buffer destinatie
cudaGraphicsMapResources(1, &pboCUDA);
void *devPtr = NULL;                      // devPtr este valid doar in CUDA
size_t devSize = 0;
cudaGraphicsResourceGetMappedPointer(&devPtr, &devSize, pboCUDA);

// apelare nucleu
imageProcessingKernel <<< grid, threads >>>
    ((float4 *) devPtr, sourceImage->widthBytes(),
     sourceImage->columns(), sourceImage->rows(), whirlAngle);
cudaThreadSynchronize();

// de-asociere textura sursa
cudaUnbindTexture(texRef);

// de-asociere destinatia CUDA. devPtr se va invalida
cudaGraphicsUnmapResources(1, &pboCUDA);
```

Memoria dispozitiv în care rezidă imaginea destinație este controlată de către OpenGL. Astfel, o putem afișa fără a fi nevoit a o trece înapoi în memoria-gazdă (CPU), ceea ce simplifică rutina de afișare, `display`(), care apelează rutina OpenGL `glDrawPixels`(). Următoarea funcție, `reshape`(), apelată la inițializare și la redimensionarea ferestrei, setează proiecția ortogonală, precum și fereastra grafică. Prezentarea detaliată a noțiunilor OpenGL se află în "Cartea Roșie" [29].

[2]GLUT - *The OpenGL Utility Toolkit* este o bibliotecă auxiliară pentru a ușura folosirea OpenGL, disponibilă la: `http://www.opengl.org/resources/libraries/glut/`

Exerciții

5.2 Operatorul Laplace discret poate fi folosit pentru a extrage conturul imaginilor, fiind definit prin nucleul

$$\mathbf{L} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Dacă P_{xy} reprezintă pixelul de la coordonatele (x, y), atunci operatorul Laplace va calcula, pentru fiecare pixel $D_{x,y}$ în imaginea destinație:

$$D_{x,y} = P_{x-1,y} + P_{x+1,y} + P_{x,y-1} + P_{x,y+1} - 4P_{x,y}$$

Implementați filtrarea Laplace în CUDA pe baza ecuației de mai sus. Pentru detalii consultați [15] și [42].

5.3 Matematicianul John Conway a inventat un automat celular bazat pe reguli foarte simple, pe care l-a numit "Jocul Vieții" (Game of Life)[6][3]. Regulile sunt: jocul se desfășoară într-o matrice booleană (teoretic infinită, practic se alege o dimensiune convenabilă pentru afișare, de exemplu 512×512). Un element al matricei (celulă) poate fi vie (1) sau moartă (0). La fiecare etapă, celulele își modifică starea, în mod sincron, pe baza celor 8 vecini adiacenți:

- O celulă vie cu mai puțin de 2 vecini vii moare.
- O celulă vie cu 2 sau 3 vecini vii supraviețuiește.
- O celulă vie cu mai mult de 3 vecini vii moare.
- O celulă moartă, cu exact 3 vecinii vii învie.

Implementați Jocul Vieții în CUDA. Este nevoie de două matrice, una pentru generația curentă, iar cealaltă pentru generația următoare. Nucleul CUDA va citi dintr-o matrice și va scrie în alta, după care rolul matricelor se inversează. Trebuie generată o populație inițială ne-nulă. Observați efectele populației inițiale asupra evoluției automatului. Ca divertisment, menționăm că, folosind acest automat celular, s-a simulat o mașină Turing universală[4].

5.4 Caleidoscopul este o jucărie formată dintr-un tub care conține trei oglinzi, așezate la 60 de grade. Definiți ecuațiile (simplificate) ale razelor de lumină prin caleidoscop și implementați în CUDA efectul aplicat unei imagini de intrare.

[3] `http://en.wikipedia.org/wiki/Conways_Game_of_Life`
[4] `http://rendell-attic.org/gol/tm.htm`

5.3 Folosirea bibliotecii CUDA FFT 2D

În acest laborator vom demonstra capabilitățile avansate ale CUDA, prin intermediul bibliotecii CUFFT (CUDA Fast Fourier Transform) [32].

Acest laborator necesită cunoștințe prealabile de procesare de semnale și imagini: transformata Fourier, filtrare, convoluție.

Reamintim definiția transformatei Fourier discrete bidimensionale pentru o imagine pătratică, în nivele de gri $f(x, y)$ de $N \times N$ linii și coloane:

$$\mathcal{F}(u, v) = \frac{1}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) e^{\mp i 2\pi \frac{ux+vy}{N}} \quad u, v = 0, \ldots, N-1.$$

unde semnul $\mp$ se ia, prin convenție: -1 pentru transformata directă și +1 pentru transformata inversă. Spunem că transformata Fourier transpune imaginea din domeniul *spațial* (x,y) în domeniul *spectral* (u,v). Una din proprietățile transformatei Fourier este de a transforma convoluția a doi vectori $\mathbf{x}$ și $\mathbf{y}$ în produsul element-cu-element în domeniul spectral (teorema convoluției):

$$\mathbf{x} \otimes \mathbf{y} = \mathcal{F}^{-1} \{\mathbf{X} \cdot \mathbf{Y}\}$$

unde $\mathbf{X} = \mathcal{F}\{\mathbf{x}\}$, $\mathbf{Y} = \mathcal{F}\{\mathbf{y}\}$ reprezintă transformatele Fourier ale vectorilor $\mathbf{x}$, respectiv $\mathbf{y}$.

Această proprietate prezintă interes din punct de vedere algoritmic, deoarece:

- evaluarea directă a convoluției a doi vectori de dimensiune N necesită un timp de execuție în ordinul lui $O(N^2)$, pe un calculator secvențial.
- transformata Fourier rapidă a unui vector de dimensiune N necesită un timp de execuție în ordinul lui $O(N \log N)$ pe un calculator secvențial și $O(N/P \log N)$ rulând în paralel pe P procesoare (pentru $N = 2^k$) [24].
- transformata Fourier bidimensională fiind *separabilă*, se poate calcula prin aplicarea succesivă a transformatei unidimensionale pe coloane și pe linii (ordinea nu contează).

Putem calcula convoluția a doi vectori prin următorul pseudo-cod:

```
X = fft(x);  // O(N/P log N)
Y = fft(y);  // O(N/P log N)
Z = X .* Y;  // O(N/P) inmultire element-cu-element
z = ifft(Z); // O(N/P log N)
```

Timpul total este în ordinul a $3O\left(\frac{N}{P} \log N\right) + O\left(\frac{N}{P}\right) \subset O\left(\frac{N}{P} \log N\right)$, care este semnificativ mai mic decât $O(N^2/P)$, pentru un N suficient de mare.

În exemplul care urmează vom încărca o imagine de pe disc și vom aplica o filtrare Gaussiană cu ajutorul transformatei Fourier implementate în biblioteca CUFFT. Filtrarea Gaussiană reprezintă o convoluție a imaginii (în domeniul spațial) cu nucleul:

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Pentru a implementa filtrarea în domeniul spectral, avem nevoie de transformata Fourier a nucleului Gaussian, care este tot o Gaussiană, dar cu altă variantă [1]:

$$G(u,v) = \mathcal{F}(g(x,y)) \approx e^{-2\pi\sigma^2(u^2+v^2)}$$

Pașii lucrării de laborator sunt următorii:

1. Inițializare: încărcarea imaginii de pe disc, inițializarea bibliotecii CUFFT.
2. Adaptarea imaginii pentru formatul acceptat de rutina FFT.
3. Transformata Fourier a imaginii, pentru a obține spectrul.
4. Prelucrarea în domeniul spectral (filtrarea Gaussiană).
5. Transformata Fourier inversă a spectrului prelucrat.
6. Afișarea imaginii rezultate.

Biblioteca CUFFT trebuie inițializată, prin crearea unui *plan* de execuție, specificând dimensiunile transformatelor:

```
cufftPlan2d(&plan, sourceImage->width, sourceImage->height, CUFFT_C2C);
```

Odată creat planul de execuție, îl refolosim în bucla de calcul-afișare. Rutina FFT este de tipul in-place: adică zona de memorie, care este folosită pentru intrare va fi suprascrisă cu rezultatele (transformata). Această zonă o declarăm astfel:

```
Image<cufftComplex, 1> *spectrum =
                            new Image<cufftComplex, 1>
          (sourceImage->width, sourceImage->height, true);
```

Tipul de date folosit de rutina FFT, denumit `cufftComplex` și declarat în fișierul `cufft.h`, conține valoarea reală și cea imaginară ale unui număr complex, împachetată într-o structură de tip `struct float2 { float x,y;}`. Vom inițializa partea reală a matricei cu valorile de luminozitate, iar partea imaginară cu 0.

Nucleul `convertToCplxKernel`, calculează mai întâi luminozitatea unui pixel din suma ponderată a componentelor de culoare, bazată pe formula $L = 0.2989 \cdot nivel_{rosu} + 0.5870 \cdot nivel_{verde} + 0.1140 \cdot nivel_{albastru}$, după care o stochează ca partea reală a matricei complexe `spectrum`. Acest nucleu este apelat din programul-gazdă, pentru a transfera și adapta imaginea din `sourceImage` în buffer-ul `spectrum`:

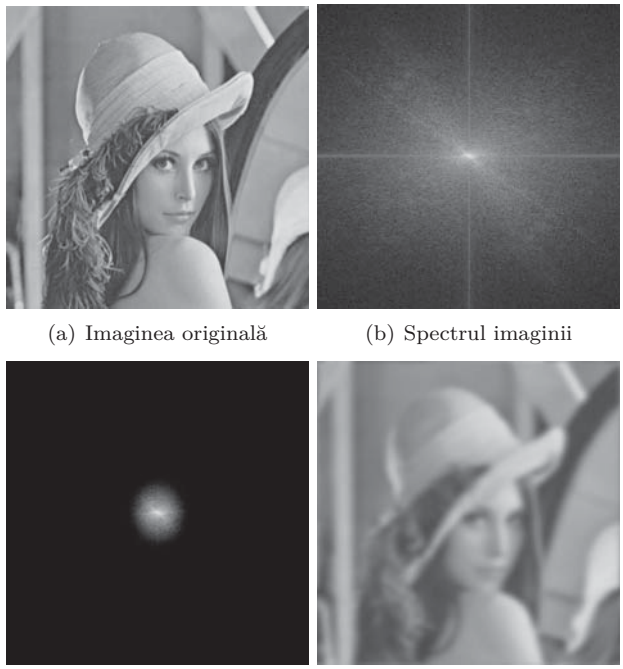
```
convertToCplxKernel <<< grid, threads >>> (
    spectrum->data_gpu, spectrum->strideGPUT(),
    (const float4*)sourceImage->data_gpu,
    sourceImage->stride_gpu/sizeof(float4),
    sourceImage->width, sourceImage->height);
```

Având datele de intrare pregătite, apelăm rutina cuFFT, pentru a calcula transformata Fourier bidimensională:

```
cufftExecC2C(plan, spectrum->data_gpu,
    spectrum->data_gpu, CUFFT_FORWARD);
```

Rutina va suprascrie zona de memorie `spectrum->data_gpu` cu rezultatul transformatei.

În acest punct, putem alege din două opțiuni: i) afișarea spectrului (Figura 5.4(b)) sau ii) prelucrarea spectrului (Figura 5.4(c)), transformarea Fourier inversă și afișarea imaginii rezultate (Figura 5.4(d)).



(a) Imaginea originală (b) Spectrul imaginii

(c) Spectrul filtrat (d) Imaginea filtrată

Figura 5.4: Filtrarea Gaussiană.

Spectrul rezultat este compus din numere complexe. Pentru opțiunea i) - afișare pe ecran - vom calcula logaritmul valorilor absolute în nucleul:

```
__global__
void convertSpectrumForDisplayKernel(float4* dst,
    int dst_stride, const cufftComplex * src,
    int src_stride, int width, int height);
```

În nucleu, apare o re-ordonare a celor 4 cadrane ale transformatei Fourier, în secvența:

```
sx = x < x0 ? x + x0 : x - x0;
sy = y < y0 ? y + y0 : y - y0;
```

astfel încât valoarea spectrală care corespunde frecvenței 0 să ajungă în mijlocul ecranului[5]. Funcția `saturate`(x) limitează argumentul x în intervalul $[0 \dots 1]$, `logf`(x) calculează logaritmul natural al unui număr real, iar `cuCabsf`(c) returnează valoarea absolută a numărului complex c.

Pentru a calcula filtrarea (în domeniul spectral), avem următorul nucleu:

```
__global__
void gaussianKernel(cufftComplex* spect,
    int stride, int width, int height, float4 param);
```

În codul de mai sus, prin variabila `param` se transmit parametrii filtrului Gaussian (variațele pe orizontală și verticală). Invocarea din codul-gazdă este:

```
gaussianKernel <<< grid, threads >>>
      (spectrum->data_gpu, spectrum->strideGPUT(),
      spectrum->width,spectrum->height,params);
```

După procesarea în domeniul spectral, restaurăm imaginea în domeniul spațial prin transformata Fourier inversă:

```
    cufftExecC2C(plan, spectrum->data_gpu,
        spectrum->data_gpu, CUFFT_INVERSE);
```

Un detaliu al implementării bibliotecii `cuFFT` este că rezultatul transformării este scalat cu N^2. Pentru a obține valorile pixelilor în domeniul original $[0 \dots 1]$, fiecare element trebuie împărțit cu N^2. Această normalizare este efectuată în nucleul de adaptare pentru afișare. În final, prezentăm imaginile rezultate din aplicarea acestui algoritm în Figura 5.4.

Codul sursă complet al acestui exemplu se găsește în fișierul `ex7.cu` la Pagina 112.

Exerciții

5.5 Pe baza exemplului anterior, implementați o filtrare de tip 'trece-sus', prin atenuarea frecvențelor spațiale joase (până la un anumit prag).

5.4 Programarea generică folosind Thrust

În exemplele anterioare s-a observat stilul de programare de nivel jos în CUDA: nevoia explicită de a aloca zonele de memorie-dispozitiv, definirea și invocarea explicită a nucleelor ș.a., ceea ce conferă un stil greoi de programare. Biblioteca Thrust[6][18] a fost proiectată întocmai pentru a simplifica lucrul în CUDA și a facilita programarea generică (asemănătoare paradigmelor implementate în STL - Standard Template Library). Astfel, biblioteca Thrust definește clasa `thrust::device_vector` echivalentul clasei `std::vector` din STL; de exemplu în declarația:

```
thrust::device_vector<float> vec(1024);
```

[5] `http://www.dspguide.com/ch24/5.htm`
[6] `http://code.google.com/p/thrust/`

obiectul `vec` încapsulează un vector de 1024 de elemente tip float, aflate în memoria dispozitiv. Programatorul este scutit de alocarea memoriei prin `cudaMalloc`, aceasta fiind automatizat de către constructorul clasei `device_vector`. În mod analog cu vectorii din STL, elementele pot fi accesate prin iterator; se pot copia vectori din/în STL (memoria gazdă) în Thrust (memoria dispozitiv), de exemplu:

```
std::list<int> h_list; // lista STL, in memoria gazda
h_list.push_back(...); // umplere cu date

// vector in memoria dispozitiv
thrust::device_vector<int> d_vec(h_list.size());
// copiere gazda --> dispozitiv
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());
```

În Thrust am implementat, printre altele, algoritmul de reducție paralelă prezentat în secțiunea 4.4. Folosind Thrust acesta se reduce la un singur apel:

```
thrust::device_vector<float> vec(1024);
float s = reduce(vec.begin(), vec.end(), 0.0f, plus<float>());
```

În mod similar, pentru a sorta elementele unui vector (agoritm implementat ca exemplu în fișierul sursă `ex8.cu` Pagina 117), apelăm:

```
thrust::sort(vec.begin(), vec.end());
```

În mod asemănător metodei de sortare din STL, Thrust permite specificarea predicatului de comparație:

```
typedef struct {
  char nume[16];
  int nota;
} Elev;

class compara_nota
{
public:
  __host__ __device__
  bool operator()(const Elev& a,const Elev & b)
  {
     return a.nota < b.nota;
  }
};

thrust::device_vector<Elev> elevi;
elevi.push_back(....)
thrust::sort(elevi.begin(), elevi.end(), compara_nota);
```

Acest exemplu va sorta lista elevilor, în paralel, pe procesorul grafic.

5.4.1 Transformarea generică

Prin biblioteca Thrust se poate defini o transformată ca o operație de tip *functor* (function object)[7] aplicată asupra unui vector. De exemplu:

[7]Functorul este o clasă C++, cu operatorul () supraîncărcat.

```
class patrat
{
public:
  __host__ __device__
  float operator()(float x)
  {
     return x * x;
  }
};
```

declară functorul `patrat`, care are ca efect ridicarea la pătrat a argumentului x. Pentru a calcula, în paralel, pătratul tuturor elementelor unui vector $x, y_i = x_i^2$, apelăm:

```
device_vector<float> y(x.size());
transform(x.begin(), x.end(), y.begin(), patrat());
```

Operația `transform` va aplica functorul `patrat`() fiecărui element din x. Pentru o descriere detaliată a algoritmilor implementați în Thrust consultați [18].

Exerciții

5.6 Implementați calculul valorii lui π din exercițiul de la Secțiunea 4.4, folosind programarea generică și reducția din biblioteca Thrust.

5.7 Implementați sortarea lexicografică a unei liste de 1.000.000 de cuvinte (formate din maxim 16 litere), folosind Thrust. Definiți functorul-predicat pentru a compara două cuvinte. Comparați timpul de execuție al sortării secvențiale, folosind STL, cu cel al sortării paralele folosind Thrust.

5.8 Implementați arborii B (B-Trees) prezentați în curs: operațiile de inserare și căutare. Sfat: dimensionați nodurile din arbore astfel încât să asignați câte un thread pentru fiecare cheie (256-512) dintr-un nod.

5.5 Alinierea secvențelor folosind CUDA

Această lucrare va prezenta implementarea unui algoritm de interes în bioinformatică, alinierea globală a două secvențe.

În biochimie, secvența nucleotidelor ce formează acidul dezoxiribonucleic (ADN), acidul ribonucleic (ARN) sau secvența aminoacizilor din proteine se pot reprezenta ca șiruri, abstractizări utile pentru a fi studiate cu metode informatice. De exemplu, ADN-ul se reprezintă prin șiruri formate peste alfabetul {A,C,G,T}, ARN-ul peste {A,C,G,U}, iar proteinele din organismul uman formează șiruri din 20 de tipuri de aminoacizi. După determinarea prin metode biochimice a unei secvențe, se caută găsirea unor secvențele similare, dintr-o bază de date, conform unui scor care măsoară *similaritatea* lor.

Putem măsura similaritatea a două șiruri, aliniindu-le printr-o succesiune de operații elementare, astfel încât să le aducem la o formă identică. Operațiile elementare și scorurile asociate sunt:

1. inserarea de spații (-), având penalizarea σ
2. potrivire $a_i = b_i = X$, având scorul $\delta(X, X) = \alpha$
3. schimbarea caracterului X în Y (mutație), având scorul $\delta(X, Y) = \beta$, unde $X = a_i$ iar $Y = b_i$.

Alinierea optimă este succesiunea de operații care maximizează scorul total, acest scor Σ fiind măsura de similaritate a celor două secvențe. O abordare duală a problemei este de a asocia o funcție *cost*, $-\delta(X, Y), -\sigma$ operațiilor elementare, căutând să minimizăm *costul* total.

Ponderile (scorurile relative) ale operațiilor determină structura alinierii optime. De exemplu, două alinieri diferite ale șirurilor `ATTAC` și `GAATA` sunt prezentate în Figura 5.5. Prima penalizează spatiile ($\rho = -2$), iar a doua diferențele de caracter ($\beta = -3$).

```
-ATTAC          --ATTAC
 |!||            |  ||
GAATA-          GAA-TA-
```

$\alpha = 1,\ \beta = -1,\ \rho = -2$, $\Sigma = -2$ $\qquad$ $\alpha = 1,\ \beta = -3,\ \rho = -1$, $\Sigma = -1$

Figura 5.5: Două alinieri diferite, în funcţie de alegerea parametrilor.

În primul caz, scorul se calculează prin: $\Sigma = \sigma + \delta(\mathtt{A}, \mathtt{A}) + \delta(\mathtt{T}, \mathtt{A}) + \delta(\mathtt{T}, \mathtt{T}) + \delta(\mathtt{A}, \mathtt{A}) + \sigma$, înlocuind constantele obţinem: $\Sigma = -2 + 1 - 1 + 1 + 1 - 2 = -2$.

În cel mai generic caz (incluzând bioinformatica), funcţia scor $\delta(X, Y)$ nu este uniformă, ea depinzând de tipul simbolurilor. Uzual se specifică printr-o matrice $\Delta_{|A|\times|A|}$, unde $|A|$ este dimensiunea alfabetului. O astfel de matrice folosită în cazul proteinelor este matricea BLOSUM (BLOcks of Amino Acid SUbstitution Matrix) [10]. Costul începerii unui spaţiu față de continuarea lui este de asemenea tratat diferit. Pentru simplitate, în această lucrare de laborator nu vom trata cazurile generice, vom implementa alinierea *globală* a 2 secvențe. Există mai multe tipuri de alinieri: globale, locale și multiple. Pentru mai multe detalii, consultați suportul de curs, cap. 7 "Tehnici de prelucrare a secvențelor de șiruri" și [20], [28].

5.5.1 Algoritmul Needleman-Wunsch de aliniere globală

Fie două șiruri $a_{[n]}, b_{[m]}$ peste un alfabet $\mathbf{V}$ și o funcție de similaritate $\delta(X, Y)$; $X, Y \in \mathbf{V}$, algoritmul Needleman-Wunsch [28] va calcula alinierea globală, care maximizează scorul $s_{n,m}$, definit recursiv ca:

$$s_{0,j} = \sigma i;\ s_{i,0} = \sigma j$$

$$s_{ij} = \max \begin{cases} s_{i-1,j} + \sigma, \\ s_{i,j-1} + \sigma, \\ s_{i-1,j-1} + \delta(a_i, b_j) \end{cases}$$

pentru $i = 1, \ldots, n$, $j = 1, \ldots, m$; σ reprezintă penalizarea asociată inserării unui spațiu. Problema se rezolvă în mod clasic prin programare dinamică: se construiește o matrice de scor $S_{[(n+1)\times(m+1)]}$ element cu element. Menţionăm că folosim convenţia din limbajul C, indicele matricelor începând cu $(0, 0)$.

$S_{0,0} \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $S_{i,0} \leftarrow \sigma i$
end for
for $j = 1, \ldots, m$ **do**
  $S_{0,j} \leftarrow \sigma j$
end for
for $i = 1, \ldots, n$ **do**
  for $j = 1, \ldots, m$ **do**
    $S_{i,j} \leftarrow \max\{S_{i-1,j} + \sigma,\ S_{i,j-1} + \sigma,\ S_{i-1,j-1} + \delta(a_i, b_j)\}$
  end for
end for

Să analizăm rularea algoritmului cu șirurile a=`ATTAC`, b=`GAATA`, folosind parametrii $\sigma = -2$, $\delta(X, X) = 1, \delta(X, Y) = -1$. Matricea $S_{[6\times6]}$ este dată în Tabelul 5.1.

	b	G	A	A	T	A
a	0	$\leftarrow$ **−2**	-4	-6	-8	-10
A	-2	-1	$\nwarrow$ **−1**	-3	-5	-7
T	-4	-3	-2	$\nwarrow$ **−2**	-2	-4
T	-6	-5	-4	-3	$\nwarrow$ **−1**	-3
A	-8	-7	-4	-3	-3	$\nwarrow$ **0**
C	-10	-9	-6	-5	-4	$\uparrow$ **−2**

Tabelul 5.1: Matricea S, completată prin programare dinamică.

Timpul de execuție al algoritmului este $O(n \cdot m)$ sau $O(n^2)$ pentru matrice pătratice. Săgețile indică direcția de parcurgere pentru a reconstrui aliniamentul, dat de următorul algoritm: pornind din colțul dreapta-jos al matricei, încercăm să ne reconstruim drumul optim spre colțul stânga-sus. La fiecare pas, urmărim decizia luată de algoritmul 5.5.1.

$i \leftarrow n$; $j \leftarrow m$
$k \leftarrow max(m, n) - 1$
while $i > 0$ **or** $j > 0$ **do**
  $score \leftarrow S_{i,j}$
  if $i > 0$ **and** $j = 0$ **and** $score = S_{i-1,j-1} + \delta(a[i-1], b[j-1])$ **then**
    $direction_k \leftarrow$ '$\nwarrow$'
    $i \leftarrow i - 1$; $j \leftarrow j - 1$
  else if $i > 0$ **and** $score = S_{i-1,j} + \sigma$ **then**
    $direction_k \leftarrow$ '$\uparrow$'
    $i \leftarrow i - 1$

```
    else
        direction_k ← ' ←'
        j ← j − 1
    end if
    k ← k − 1
  end while
```

După rularea algoritmului, vectorul *direction* va conține succesiunea de operații. Astfel, ↑ indică inserarea unui spațiu în șirul b, ↖ indică potrivire de caracter sau schimbare de caracter, iar ← inserare de spațiu în șirul a., Complexitatea algoritmului de reconstituire este $O(max(n, m))$. Menționăm că pot exista mai multe alinieri optime (drumul invers nefiind unic), algoritmul de sus listează doar una din aceste soluții.

Algoritmul paralel

Pentru a implementa în paralel algoritmul care calculează scorul optim trebuie să observăm interdependența datelor: Astfel, $S_{i,j}$ depinde doar de valorile celulelor $S_{i-1,j}, S_{i,j-1}, S_{i+1,j}$, dar nu depinde de celulele pe aceeași antidiagonală: $S_{i+1,j-1}, S_{i-1,j+1}$ (Figura 5.6).

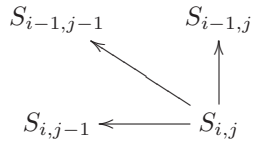


Figura 5.6: Dependența datelor în algoritmul 5.5.1.

Inițializăm prima linie și prima coloană a matricei, după care, parcurgem matricea antidiagonală cu antidiagonală, pornind din colțul stânga-sus, procesând celulele antidiagonalei în paralel (Figura 5.7). Ordinea în care se actualizează elementele matricei este similară propagării unui front de undă. Algoritmul paralel este următorul:

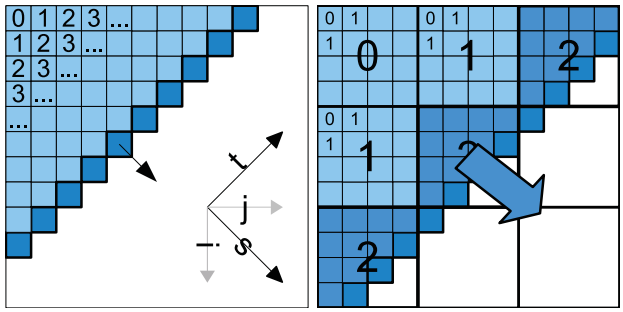
```
for s = 0 to n + m − 2 do                          ▷ Indice antidiagonala
    if s < min(n, m) then
        T = 1 + s                                  ▷ Nr. elemente pe antidiagonala s
    else
        T = min(n, m, n + m − s)
    end if
    for t = 0 to T − 1 do in parallel              ▷ Din indicele antidiagonalei s
        if n ≥ m then                              ▷ și indicele threadului t,
            i = 1 + s − t                          ▷ obţinem coordonatele i, j
            j = 1 + t                              ▷ printr-o schimbare de variabilă
        else
            i = 1 + t
            j = 1 + s − t
```

end if

$S_{i,j} = \max\{S_{i-1,j} + \sigma,\ S_{i,j-1} + \sigma,\ S_{i-1,j-1} + \delta(a_i, b_j)\}$

end for parallel

end for



(a) Simplu (b) Pe blocuri

Figura 5.7: Parcurgerea pe diagonală a matricei. Elementele antidiagonalei se procesează în paralel. Cifrele din celule indică ordinea de procesare.

Din programul gazdă, nucleele se apelează pentru fiecare antidiagonală formată din blocuri. Programul va avansa la următoarea antidiagonală-bloc doar după terminarea tuturor threadurilor din blocurile curente.

Timpul de execuție al algoritmului paralel, având la dispoziție p procesoare este $O(\lceil \min(m,n)/p \rceil \cdot (m+n))$. Observăm că, sub această formă, algoritmul nu este foarte eficient, la antidiagonalele cu număr de elemente mai mici decât p, nu putem asigna toate procesoarele, iar din cauza dependențelor de date, nu putem paraleliza bucla externă.

Implementarea în CUDA, prima variantă

Implementarea directă a algoritmului în CUDA constă din definirea nucleului corespunzătoare buclei for paralelă din algoritm:

```
__global__
void alignKernel(Matrix<int> S, int s, const int *  a, const int * b){
  int i,j;
  const int n=S.rows();
  const int m=S.columns();
  int t = blockIdx.x*blockDim.x + threadIdx.x;
  i = 1 + (n >= m ? s-t : t   );
  j = 1 + (n >= m ?   t : s-t );
  if (i<1 || j<1 || i>=n || j>=m) return;
  int x = max(S[i-1][j-1] + simil(a[i-1],b[j-1]),
          max(S[i-1][j] + gap,
              S[i][j-1] + gap));
  S[i][j] = x;
}
```

Nucleul se apelează iterativ, pentru $s = 0, 1, \ldots, m+n-2$ din programul gazdă. Funcţia de similaritate este notată cu `simil(X,Y)`$=\delta(X, Y)$, iar constanta `gap` $= \sigma$ este penalizarea asociată unui spațiu.

Un neajuns al acestei abordări este accesul la memorie: deoarece se accesează elementele în diagonală ale matricei, acestea fiind la distanță mare în reprezentarea uzuală, nu este folosit optim cache-ul. O soluție ar fi folosirea memoriei rapide din multiprocesoare. Pentru aceasta, folosim o strategie asemănătoare celei din Secțiunea 4.6, împărțirea în blocuri a matricei, încărcând fiecare bloc în memoria locală partajată.

Implementarea CUDA, varianta optimizată pe blocuri

Primele instrucțiuni din nucleu copiază datele din memoria globală corespunzătoare blocului curent în memoria partajată, declarată prin

```
__shared__ int B[BLOCK_SIZE+1][BLOCK_SIZE+1]|.
```

Prima linie și coloană ale matricei locale se suprapun cu ultima linie și coloană ale blocurilor învecinate. Nucleul `alignKernelBlock` este compus din etapele:

- Copiere bloc din memoria globală în memoria partajată (matricea B);
- Algoritmul de programare dinamică, pe blocul B;
- Copiere rezultat în memoria globală.

Sursele complete ale programului se găsesc în fișierul `ex9.cu`, listat la Pagina 118.

După compilare, executabilul ex9 se invocă prin

`ex9␣[opțiuni]␣secventa1␣secventa2`

Programul va afișa aliniamentul și scorul optim. Opțiunile din linia de comandă sunt:

-a *alg* Alege implementarea:

0 = CPU, secvențial

1 = CUDA, prima variantă

2 = CUDA, a doua variantă

-v *nivel* În funcție de nivelul specificat, programul va afișa mai multe detalii pe parcursul rulării:

1 = scorul alinierii

2 = timpul de execuție

3 = șirurile aliniate

4 = matricea S

-r n Generează secvențe aleatoare de lungime n. Util pentru a experimenta cu cazuri mari (în limita memoriei disponibile). Reamintim că programul necesită memorie $O(n^2)$.

Exemplu de rulare:

```
ex9 -v 4 MASTER ITEMS
               I     T     E     M     S
         0    -2    -4    -6    -8   -10
   M    -2    -1    -3    -5    -5    -7
   A    -4    -3    -2    -4    -6    -6
   S    -6    -5    -4    -3    -5    -5
   T    -8    -7    -4    -5    -4    -6
   E   -10    -9    -6    -3    -5    -5
   R   -12   -11    -8    -5    -4    -6

MASTE-R
--ITEMS
Scor=-6
```

Exerciții

5.9 Compilați și rulați cele 3 implementări (secvențială pe CPU, CUDA 1. și CUDA pe blocuri), măsurând timpii de execuție.

5.10 Determinați experimental pentru ce dimensiuni de șiruri implementarea CUDA devine mai rapidă decât cea pe CPU.

5.11 Structura algoritmilor de calcul al aliniamentului global și al distanței de editare Levenshtein sunt similare. Transformați nucleul CUDA de calcul al aliniamentului global în calculul distanței Levenshtein.

5.12 Dacă ne interesează doar scorul final de similaritate, fără a avea nevoie de a reconstitui secvența operațiilor care conduc la alinierea optimă, putem renunța la stocarea completă a matricei S, memorând doar antidiagonala curentă și cea anterioară, reducând consumul de memorie de la $(1 + n) \cdot (1 + m)$ la $2(1 + \min(n, m))$. Implementați în CUDA această variantă.

5.13 Implementați o altă optimizare, "algoritmul celor patru ruși" [3], care se bazează tot pe descompunerea matricei în blocuri mici. Observând că blocurile depind doar de datele de pe marginile stânga și sus, rezultatele posibile se pot precalcula și stoca în memoria constantă CUDA.

Capitolul 6

Algoritmi de aproximare

În cadrul acestui capitol se vor studia algoritmi de aproximare pentru rezolvarea problemei celui mai scurt superstring, problemă NP-completă. Algoritmul rezultat este de complexitate polinomială, însă obţine doar o aproximare a soluţiei optime.

6.1 Breviar teoretic

Pentru rezolvarea problemelor NP-complete se cunosc variantele de lucru:

- pentru dimensiuni mici ale instanţelor se foloseşte un algoritm de rezolvare exactă;
- daca se urmăreşte doar rezolvarea unor subprobleme particulare, acestea se izolează şi se încearcă rezolvarea cu complexitate polinomială;
- se foloseşte un algoritm care dă soluţii *aproape optime*, dar în timp polinomial.

Pentru ultima variantă rezultă aşa-numiţii algoritmi de aproximare, compromis realizat între exactitatea soluţiei şi resursele computaţionale existente. Algoritmii de aproximare sunt dezvoltaţi pentru probleme de optimizare, de exemplu determinarea celui mai scurt drum, maximizarea profitului, găsirea celei mai lungi/scurte secvenţe de caractere cu o anumită proprietate etc.

Calitatea unui algoritm de aproximare se determină comparând costul soluţiei optime C^* şi cel determinat de algoritmul de aproximare C. Spunem că algoritmul de aproximare are *raportul de aproximare* $\rho(n)$ dacă pentru orice dimensiune n a problemei avem:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

şi, în acest caz, algoritmul care determină o soluţie de cost C este un $\rho(n)$-algoritm de aproximare.

Se observă că $\rho(n) \geq 1$. Dorim ca $\rho(n)$ să fie cât mai apropiat de 1, *i.e.* aproximarea să fie cât mai bună. Uneori ρ este independent de n, în alte situaţii poate să depindă în plus şi de timpul de rulare alocat.

O categorie specială de algoritmi de aproximare este schema de aproximare. O *schemă de aproximare* primeşte o instanţă a problemei de optimizare şi pentru o orice valoare $\varepsilon > 0$ se obţine un $(1 + \varepsilon)$-algoritm de aproximare. Dacă schema rulează în timp polinomial, atunci avem *schemă de aproximare în timp polinomial*. Dacă timpul de rulare este polinomial atât în n, cât şi în $(1/\varepsilon)$, atunci avem o *schemă de aproximare în timp complet polinomial*.

Menţionăm în final că pentru o problemă oarecare se pot da mai mulţi algoritmi de aproximare, care fie duc la aproximări cu rapoarte tot mai apropiate de 1, fie au complexitate mai mică.

6.2 Introducere în Python

În acest capitol se va face o scurtă prezentare a elementelor de bază ale limbajului Python, cu ajutorul cărora să se poată rezolva exerciţiile din prezenta lucrare. Pentru o descriere detaliată a limbajului se recomandă consultarea documentaţiei de pe site-ul python.org [34], [35], [36].

Limbajul Python este un limbaj de programare scriptic, motiv pentru care programul nu este compilat, ci este rulat folosind un interpretor. Fiecare instrucţiune trebuie scrisă pe o linie separată, motiv pentru care nu mai este nevoie de folosirea caracterului ";" la sfârşitul instrucţiunilor. Mai multe instrucţiuni formează un bloc de intrucţiuni dacă au aceeaşi indentare (nu există metode de grupare precum acolade deschise/închise).

```
if(a>2):
    print "a mai mare ca doi \n"
    print "o alta instructiune pe ramura then"
else:
    print "a nu este mai mare ca doi\n"
    print "o alta instructiune pe ramura else"
print "acest print nu este in ramura else"
```

O linie de comentariu începe cu caracterul #. Numele şi tipul variabilelor nu trebuie declarate înainte de folosirea lor, interpretorul determină tipul variabilei la prima folosire a ei.

```
#numere
i = 1
j = j+3

#stringuri
s = "acesta este un string"
s1 = "acesta este un alt string"
```

```
#tablou de numere
a = [1,2,3,4]
#tablou de stringuri
a1=["unu","doi","trei"]
```

Afişarea se face folosind funcția *print*:

```
#afisarea unui string
print "Hello world"

#afisarea unui numar
i=3
print i

#afisarea unui string si a unui numar
print "valoarea lui i=", i
```

Sintaxa pentru instrucţiunea *if* este:

```
if (<conditie>) :
  <instructiune sau instructiuni ramura then>
else:
  <instructiune sau instructiuni ramura else>
```

Instrucţiunile din ramura *then*, respectiv din ramura *else*, trebuie indentate faţă de instrucţiunea *if*.

Sintaxa pentru bucla *for* diferă de cea din C sau Java. Pentru bucla *for* o variabilă contor va lua pe rând valorile dintr-o mulţime; valorile din mulţime pot să fie de orice tip nu doar numerice:

```
for <variabila contor> in <variabila multime>:
  <corpul buclei>

#Exemple
a=["unu","doi","trei"]
for i in a:
  print i
#afiseaza elementele din vectorul a

b=[1,4,7]
for i in b:
  print i
#afiseaza elementele din vectorul b

for i in range(0,10):
  print i
#afiseaza numerele de la 0 la 9
```

Declararea unei funcţii se face folosind sintaxa de mai jos. Corpul funcţiei trebuie să fie indentat faţă de linia de declarare a ei. În cazul în care funcţia nu returnează o valoare, instrucţiunea *return* poate fi eliminată.

```
def <nume functie>(<lista parametrii>):
  <corpul functiei>
  return <valoare>
```

Pentru rezolvarea exerciţiilor din cadrul laboratorului sunt utile următoarele operaţii şi funcţii exemplificate:

```
s1 = "abc"
s2 = "def"
#concatenarea stringurilor
s3 = s1 + s2
#s3 va contine "abcdef"

#lungimea unui string
print len(s1)
#va afisa 3
print len(s3)
#va afisa 6

#substring dintr-un string
s4 = s3[1:3]
#s4 va contine "bcd"
#1 este indicele primului caracter din string
#de la care se formeaza substringul
#3 este indicele caracterului din string
#pana la care se formeaza substringul

a=["unu","doi","trei"]
#copierea valorilor unui tablou
b=a[:]

#citirea datelor dintr-un fisier
f = open('data/ex_1.txt', 'r')
strings1=f.readlines()
f.close()

#scrierea datelor in fisier
mesaj="Acest string va fi scris in fisier"
f= open("data/output.txt","w")
f.write(mesaj)
f.close()

#Accesarea timpului
```

```
#la inceputul fisierului se include functia time
from time import time

#dupa care se poate apela
t=time()
print t
#t este un numar in virgula mobila
```

Exerciţii

6.1 Proiectaţi şi implementaţi o metodă care realizează o suprapunere maximă a două stringuri: sufixul primului string trebuie să coincidă cu prefixul celui de al doilea. Exemplu:

ACCC şi CCCCC = ACCCCC
GTA şi GTCCC = GTAGTCCC

6.2 Realizaţi un program care, folosind metoda backtracking, să realizeze toate combinările posibile ale stringurilor de intrare şi să returneze cel mai scurt superstring. Testaţi programul pe exemplul din Figura 6.1. Un exemplu de program Python pentru generarea permutărilor elementelor dintr-o listă este prezentat în cele ce urmează, în care *perm_partiala* este variabila care conţine stringul aferent permutării parţiale, *elemente_de_permutat* reprezintă lista elementelor care mai trebuie permutate şi *copie_elemente* este o copie locală a elementelor de permutat.

```
def perm (perm_partiala,elemente_de_permutat):
    for i in elemente_de_permutat:
        copie_elemente=elemente_de_permutat[:]
        copie_elemente.remove(i)
        if(len(copie_elemente)):
            perm(perm_partiala+str(i)+",",copie_elemente)
        else:
            print perm_partiala+str(i)

lista = ["a","b","c"]
perm("",lista)
```

6.3 Problema celui mai scurt superstring

În cadrul eforturilor de asociere a secvenţelor ADN a diferitelor organisme o primă problemă a fost imposibilitatea asocierii secvenţelor de lungime mai mare de câteva sute sau mii de nucleotide. Acesta este motivul pentru care s-a

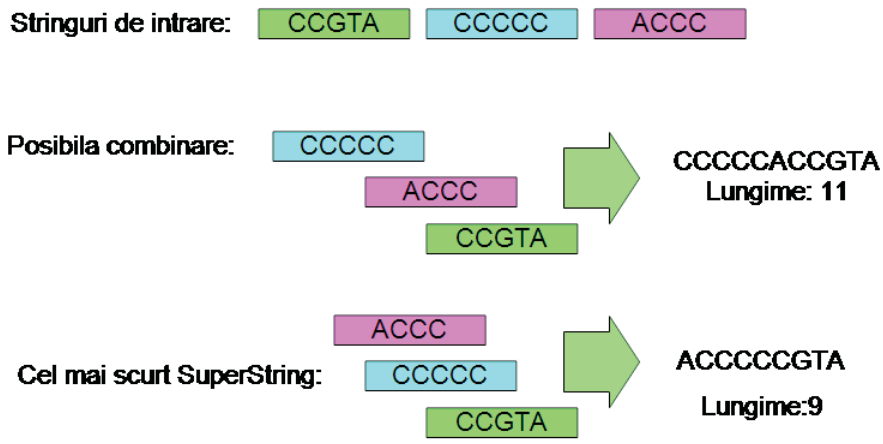


Figura 6.1: Combinarea stringurilor în SuperString.

adoptat strategia ruperii ADN-ului în secvenţe de dimensiuni rezonabile, care pot fi asociate individual, urmând ca într-o fază ulterioară aceste secvenţe să fie recombinate obţinând genomul organismului respectiv. Evident, pentru a putea reconstrui genomul, bucăţile de secvenţe ADN trebuie să aibă porţiuni comune (două câte două).

Dată fiind o mulţime de stringuri S, un superstring este un string care are proprietatea de a conţine fiecare string din S. Având o serie de stringuri (în cazul ADN-ului un string va reprezenta o succesiune de nucleotide), se presupune că genomul corect este cel mai scurt superstring alcătuit din stringurile de intrare. Stringurile se ordonează astfel încât să existe o suprapunere maximă între ele. În Figura 6.1 se prezintă trei secvenţe precum şi două moduri posibile de a combina cele 3 secvenţe într-un superstring. În total există 6 permutări posibile a acestor 3 stringuri, dar doar una conduce la cel mai scurt superstring.

Numărul de permutări a n elemente este egal cu $n!$, deci ordinul de timp pentru determinarea celui mai scurt superstring este in $\Omega(n!)$. Din aproximarea lui Stirling avem:

$$n! \geq \left(\frac{n}{e}\right)^n \sqrt{2n\pi} \tag{6.1}$$

rezultă că algoritmul care găseşte cel mai scurt superstring prin considerarea tuturor permutărilor are o complexitate mai mare decât exponenţială. Conform [14], problema găsirii celui mai scurt superstring este de fapt NP-completă.

Dat fiind timpul prohibitiv necesar rezolvării instanţelor de dimensiune mare, preferăm folosirea unui algoritm de aproximare, chiar cu riscul de a nu avea garanţia soluţiei optime. Mai clar, vom utiliza un algoritm de complexitate polinomială, care să aibă o soluţie apropiată de soluţia optimă.

În cele ce urmează este dat un algoritm greedy aproximativ. Algoritmul menţine o mulţime T de stringuri; iniţial T este mulţimea de substringuri dată.

La fiecare pas se calculează suprapunerea maximă a două stringuri (lungimea maximă a sufixului din primul string care este egală cu prefixul din al doilea string). Aceste stringuri sunt scoase din mulţimea T şi se reintroduce stringul obţinut din concatenarea lor. După $n-1$ paşi (unde n este numărul iniţial de stringuri) mulţimea T va conţine un singur string care este rezultatul algoritmului.

Exerciţii

6.3 Găsiţi un contraexemplu pentru care algoritmul greedy nu dă rezultatul optim.

6.4 Care este complexitatea algoritmului greedy?

6.5 Implementaţi algoritmul greedy dat mai sus şi rulaţi-l pe exemplul din Figura 6.1.

6.6 Directorul *data* conţine o serie de fişiere de intrare, iar fiecare fişier are pe fiecare linie un string. Creaţi un program care să ruleze ambii algoritmi (backtracking şi greedy aproximativ) construind superstringul pe fiecare fişier de intrare.

- Măsuraţi timpul de rulare a implementării algoritmului backtracking şi a celui greedy pentru fiecare fişier de intrare. Calculaţi factorul de speed-up exprimat ca fiind

$$SpeedUp = \frac{TimpBacktracking}{TimpGreedyAproximativ} \tag{6.2}$$

  Calculaţi media acestor valori pe toate fişierele de intrare.

- Calculaţi factorul de aproximare ca fiind

$$FactorDeAproximare = \frac{LungimeSuperstringGreedy}{LungimeSuperstringBacktracking} \tag{6.3}$$

  Calculaţi media acestor valori pe toate fişierele de intrare.

Capitolul 7

Algoritmi euristici în grafuri

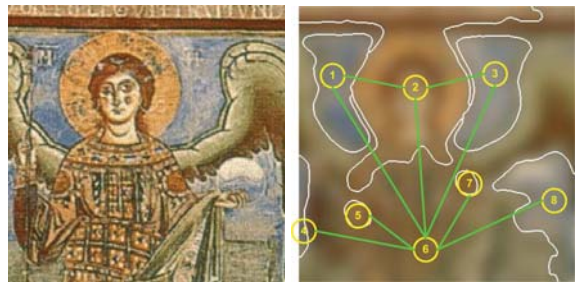
În acest capitol, cititorul va avea ocazia să experimenteze lucrul cu grafuri, în special cu algoritmi de graph matching, folosind biblioteca `igraph`[1] disponibilă pentru limbajele Python şi R. Lucrarea de faţă a fost elaborată folosind următoarele versiuni: `igraph` 0.5.4 şi Python 2.7.1.

7.1 Breviar teoretic

În biologia computaţională sunt folosiţi în mod frecvent *arborii filogenetici*. Aceştia sunt arbori cu sau fără rădăcină ai căror noduri terminale sunt etichetate cu denumiri ale unor entităţi din biologie. Reţelele filogenetice sunt grafuri orientate aciclice ale căror noduri terminale sunt etichetate, de asemenea, cu denumiri ale unor elemente biologice. Nodurile interne pot fi de tip *arbore* dacă au doar un părinte sau de tip *hibrid* dacă au doi sau mai mulţi părinţi. O reţea filogenetică se numeşte *fully resolved* dacă fiecare nod intern de tip arbore are doi copii şi fiecare nod intern are doi părinţi şi un copil.

În imagistică se folosesc grafuri de adiacenţă de regiuni. Nodurile descriu diversele regiuni ale unei imagini segmentate, iar arcele descriu relaţiile de adiacenţă dintre regiuni [39] [40] (vezi Figura 7.1). Grafurile pot fi integrate într-o reprezentare multidimensională a imaginii, care să permită căutarea unui obiect la diverse nivele de rezoluţie. În cazul recunoaşterii de forme într-o scenă simplă (care conţine doar forma ce se doreşte a fi recunoscută), graful de adiacenţă a regiunilor nu prezintă interes. Structurile de graf sunt în acest caz bazate pe contur sau schelet, cele mai cunoscute fiind grafurile "şoc" (în engleză "shock graphs") care reprezintă grafuri orientate etichetate [41].

[1] http://igraph.sourceforge.net/



(a) (b)

Figura 7.1: Imaginea originală şi graful de adiacenţă de regiuni obţinut după o segmentare a imaginii originale.

Grafurile pot fi utilizate pentru descrierea unei forme în vederea recunoaşterii sau urmăririi[2] acesteia. Graful este construit pornind de la schelet, fiecare grup de pixeli adiacenţi cu aceeaşi etichetă fiind reprezentat printr-un nod, arcele legând grupuri adiacente. Algoritmii de scheletonizare se bazează de regulă pe operaţii morfologice sau pe transformarea de distanţă[3] [11], un exemplu de astfel de algoritm fiind prezentat în [52]. Prelucrarea scheletului în vederea obţinerii grafului reprezentativ include detecţia de vârfuri şi arce, calculul de atribute ş.a.m.d. În Figura 7.2 este ilustrată descrierea, folosind grafuri, a scheletului unui individ uman, în contextul unei aplicaţii de recunoaţere a acţiunilor acestuia folosind o cameră ToF (Time of Flight) [33], care pe lângă imaginea de luminanţe furnizează şi o imagine de distanţe. Aceasta din urmă este segmentată prin prăguirea histogramei, iar apoi scheletul obiectului de interes este obţinut și reprezentat sub forma unui graf, având punctele cheie în extremităţile libere ale arcelor.

(a) (b) (c) (d)

Figura 7.2: Reprezentarea obiectului de interes cu ajutorul unui graf, pornind la imaginile de luminanţă şi de distanţă şi folosind segmentare şi scheletonizare.

În biochimie, izomorfismul de grafuri este utilizat pentru identificarea unei

[2]În engleză “tracking”.
[3]În engleză “distance transform”.

substanţe chimice într-o bază de date sau pentru generarea de grafuri moleculare, în vederea sintezei unei substanţe chimice cu ajutorul calculatorului.

7.1.1 Grafuri în Python cu `igraph`

Înainte de toate, trebuie verificat dacă biblioteca `igraph` [9] este instalată şi că totul este în regulă pentru buna desfăşurare a lucrării de laborator. Pentru aceasta, tastaţi următoarele două linii în interpretorul Python:

```
>>> import igraph.test
>>> igraph.test.test()
```

Pentru a afla versiunea bibliotecii `igraph` instalate:

```
>>> import igraph
>>> print igraph.__version__
0.5.4
```

Pentru a folosi metodele bibliotecii `igraph` este suficient să importaţi modulul Python cu acelaşi nume sau toate clasele şi metodele din modulul `igraph`:

```
>>> from igraph import *
```

Crearea grafurilor

Să creăm un prim graf - pentru început unul care conţine un singur nod şi să aflăm câteva informaţii despre el:

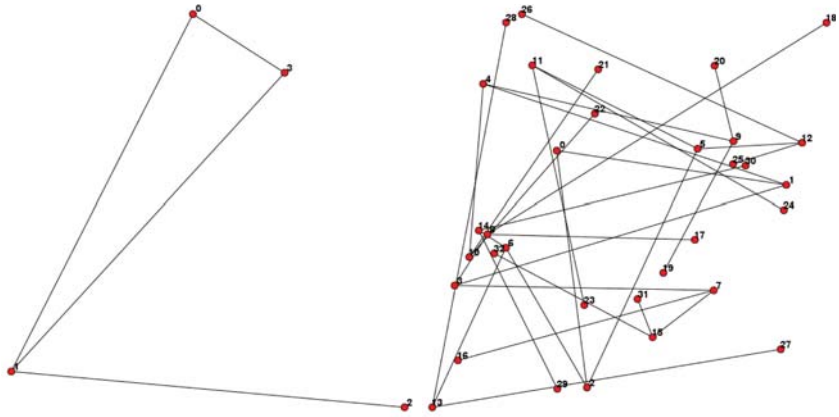
```
>>> g = Graph(1)
>>> g
<igraph.Graph object at 0x02B50F10>
>>> print(g)
Undirected graph (|V| = 1, |E| = 0)
```

Pentru adăugarea de noduri se foloseşte funcţia `add_vertices()`, având ca parametru numărul de noduri ce se doreşte a fi adăugate la graful deja existent. Folosind funţia `add_edges()` se pot adăuga muchii, specificând ca parametru o listă de dublete reprezentând nodurile ce determină fiecare muchie, ca în exemplul de mai jos.

```
>>> g.add_vertices(3)
<igraph.Graph object at 0x02B50F10>
>>> g.add_edges([(0,1), (1,2), (0,3), (1,3)])
<igraph.Graph object at 0x02B50F10>
```

Folosind funcţia `plot()` pentru vizualizarea grafului obţinut, se obţine un graf asemănător cu cel prezentat în Figura 7.3(a).

```
>>> plot(g)
```



(a) Un graf simplu (b) Un graf de tip `Tree`

Figura 7.3: Exemple de grafuri.

Generarea grafurilor

Biblioteca `igraph` include un număr mare de funcţii generatoare de grafuri, clasificabile în două categorii: deterministe şi aleatoare. Un generator deterministic va produce acelaşi graf atunci când este apelat cu aceeaşi parametri, pe când un generator aleator va produce de fiecare dată un alt graf. Funcţiile de generare deterministe includ metode pentru crearea de arbori, inele, grafuri celebre, iar generatoarele aleatoare sunt capabile să genereze reţele aleatoare de tip Erdös-Rényi, reţele Barabási-Albert, grafuri aleatoare geometric ş.a.m.d.

De exemplu, funcţia `Graph.Tree()` generează un arbore. Funcţia are ca parametri de intrare numărul total de noduri şi numărul de copii ai fiecărui nod, excepţie făcând nodurile de tip "frunză" care bineînţeles că nu au copii. Numărul total de noduri N va include şi rădăcina. De câte ori este apelată această funcţie, cu aceeaşi parametri, se obţine acelaşi arbore. Iată un exemplu de utilizare:

```
>>> g = Graph.Tree( 33, 2 )
>>> summary(g)
33 nodes, 32 edges, undirected
Number of components: 1
Diameter: 9
Density: 0.0606
Average path length: 5.1288
>>> plot(g)
```

Rezultatul este un arbore de forma celui din Figura 7.3(b).

Pentru a afla lista de muchii se foloseşte funcţia `get_edgelist()`. Dacă vrem să aflăm primele 5 elemente din lista de muchii ale grafului generat:

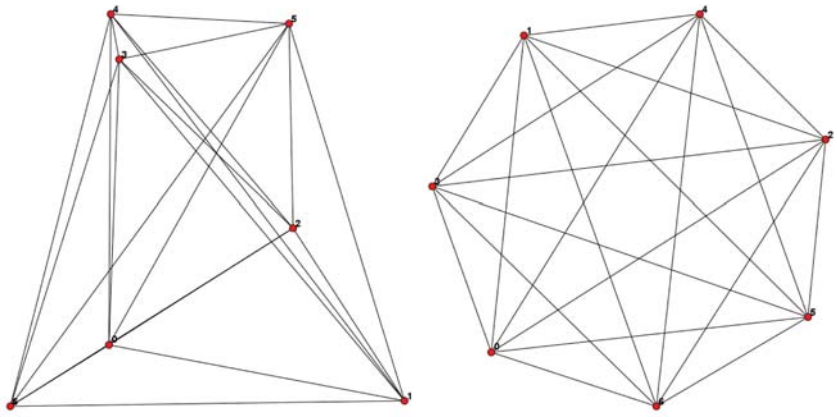
```
>>> g.get_edgelist()[0:5]
[(0, 1), (0, 2), (1, 3), (1, 4), (2, 5)]
```

Afişarea grafurilor

Un alt exemplu de generare de grafuri, de această dată complete:

```
>>> g = Graph.Full(7)
>>> plot(g)
>>> layout = g.layout( "kk" )
>>> plot(g, layout = layout )
```

Rezultatul afişării grafului atunci când nu se foloseşte nici o metodă de reprezentare[4] şi atunci când se foloseşte metoda Kamada-Kawai de reprezentare este cel din Figura 7.4.



(a) No layout (b) Kamada-Kawai

Figura 7.4: Exemplu de utilizare a metodelor de reprezentare.

În Tabelul 7.1 sunt prezentate metodele de reprezentare disponibile pentru afişarea grafurilor:

[4] În engleză "layout".

Nume reprezentare	Abreviere	Descriere
layout_circle	circle, circular	Reprezentare deterministă ce plasează nodurile grafului pe un cerc
layout_drl	drl	Reprezentare utilizată pentru grafuri mari (alg. distribuit recursiv)
layout_fruchterman_reingold	fr	Algoritmul Fruchterman-Reingold
layout_fruchterman_reingold_3d	fr3d, fr_3d	Algoritmul Fruchterman-Reingold 3D
layout_grid_fruchterman_reingold	grid_fr	Algoritmul Fruchterman-Reingold pentru grafuri mari
layout_kamada_kawai	kk	Algoritmul Kamada-Kawai
layout_kamada_kawai_3d	kk3d, kk_3d	Algoritmul Kamada-Kawai 3D
layout_lgl	large, lgl, large_graph	Algoritmul Large Graph Layout pentru grafuri mari
layout_random	random	Plasează nodurile grafului în mod aleator
layout_random_3d	random_3d	Plasează nodurile grafului în mod aleator în 3D
layout_reingold_tilford	rt, tree	Reprezentarea Reingold-Tilford pentru arbori
layout_reingold_tilford_circular	rt_circular_tree	Reprezentarea Reingold-Tilford tree layout cu transformare de coordonate polare
layout_sphere	sphere, spherical, circular, circular_3d	Reprezentarea deterministă ce plasează nodurile grafului uniform pe o sferă

Tabelul 7.1: Metode de reprezentare pentru afişarea grafurilor.

Pentru modificarea altor parametri, cum ar fi dimensiunea nodurilor sau grosimea arcelor, este prezentat un exemplu de cod Python în continuare, precum şi rezultatul grafic obținut (Figura 7.5).

```
>>> visual_style = {}
>>> visual_style["vertex_size"] = 20
>>> visual_style["edge_width"] = 3
>>> plot(g, **visual_style)
```

Atribute modificabile ale nodurilor sunt prezentate în Tabelul 7.2.

Atribut	Semnificaţie
vertex_color	culoarea unui nod
vertex_label	eticheta unui nod
vertex_label_angle	plasarea etichetei unui nod pe un cerc în jurul nodului; exprimat în radiani; 0 grade - dreapta nodului
vertex_label_color	culoarea etichetei unui nod
vertex_label_dist	distanţa de la etichetă la nod, relativă la dimensiunea nodului
vertex_label_size	dimensiunea fontului pentru eticheta nodului
vertex_shape	forma nodurilor: rectangle, circle, hidden, triangle-up, triangle-down
vertex_size	dimensiunea nodului exprimată în număr de pixeli

Tabelul 7.2: Atributele nodurilor unui graf.

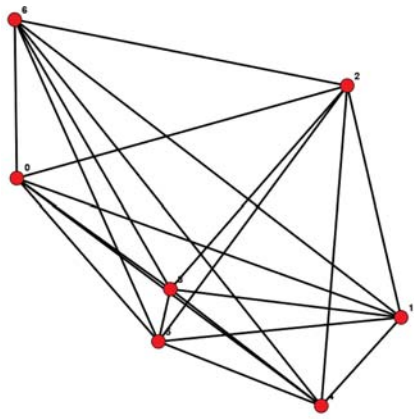


Figura 7.5: Exemplu de modificare a atributelor unui graf.

Atribute modificabile ale arcelor sunt prezentate în Tabelul 7.3.

Atribut	Semnificaţie
edge_color	culoarea arcului
edge_arrow_size	lungimea săgeţii pentru arce orientate, relativă la 15 pixeli
edge_arrow_width	lăţimea săgeţii pentru arce orientate, relativă la 10 pixeli
edge_width	lăţimea arcului exprimată în pixeli

Tabelul 7.3: Atributele arcelor unui graf.

Pentru specificare culorilor pentru un nod sau un arc se folosesc liste, tupleuri sau un string de valori separate cu virgulă, reprezentând în fiecare caz tripletul RGB. De exemplu: (255, 128, 0), [255, 128, 0] sau ”255, 128, 0”.

7.2 Înălţimea unui nod într-o reţea filogenetică

Două noduri dintr-o reţea filogenetică pot fi conectate prin unul sau mai multe drumuri, iar o muchie este parcursă cel mult o dată de un drum. *Înălţimea* unui nod i dintr-o reţea filogenetică este lungimea celui mai lung drum de la nod până la nodurile terminale.

O implementare Python a acestui algoritm, practic o parcurgere în lăţime[5], este realizată prin funcţia network_height:

```
from igraph import *
def network_height(net) :
  net.vs["height"] = [0]*len(net.vs)
```

[5]În engleză “breadth-first search”.

```
net.vs["visited"] = [False]*len(net.vs)
Q = net.vs.select(_outdegree = 0).indices
while (len(Q)>0) :
  j = Q[:1][0]
  Q = Q[1:]
  net.vs[j]["visited"] = True
  I = net.neighbors(j,"in")
  for i in I :
    net.vs[i]["height"] = max(net.vs[i]["height"],net.vs[j]
                                                  ["height"]+1)
    if all(net.vs.select(net.neighbors(i,"out"))["visited"]) :
  Q.append(i)
print net.vs["height"]
```

Fie următoarea reţea filogenetică din Figura 7.6:

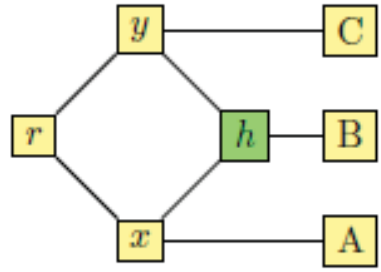


Figura 7.6: Reţea filogenetică.

Funcţia network_height poate fi testată astfel:

```
g = Graph([(0,1), (0,2), (1,4), (1,3), (2,3), (2,6), (3,5)])
g.vs["name"] = ["r","x","y","h","A","B","C"]
network_height(g)
```

şi se obţine rezultatul:

```
[3, 2, 2, 1, 0, 0, 0]
```

7.3 Graph matching

Fiind date două grafuri $G_1 = (N_1, A_1)$ şi $G_2 = (N_2, A_2)$, o relaţie $M \subset N_1 \times N_2$ este un *izomorfism* dacă şi numai dacă este o funcţie bijectivă care păstrează structura celor două grafuri, adică M asociază fiecare arc al lui G_1 pe un arc de-al lui G_2 şi vice-versa. M se spune că este un izomorfism de tip

graf-subgraf dacă şi numai dacă M este un izomorfism între G_2 şi un subgraf din G_1.

Pentru algoritmul prezentat în continuare [8], cele două grafuri sunt considerate grafuri orientate[6], extensia algoritmului pentru grafuri ne-orientate fiind intuitivă. Intrarea algoritmului este reprezentată de o stare intermediară s, asociată unei relaţii parţiale $M(s)$. Pentru starea iniţială s_0 se verifică $M(s_0) = \emptyset$. Rezultatul algoritmului este relaţia $M(s)$ între cele două grafuri. $P(s)$ reprezintă mulţimea perechilor candidate pentru includerea în mulţimea $M(s)$, iar $F(s, n, m)$ este o funcţie booleană, numită *funcţie de fezabilitate*, utilizată pentru a reteza[7] arborele de căutare. Dacă valoarea sa este $TRUE$ atunci este garantat faptul că adăugând (n, m) la s se obţine un izomorfism parţial: starea finală va fi deci fie un izomorfism între G_1 şi G_2 sau un izomorfism de tip graf-subgraf între un subgraf al lui G_1 şi graful G_2.

```
procedure Match(s)
   input: o stare intermediara s
   output: relatia intre cele doua grafuri

   if M(s) acopera toate nodurile lui G2 then
      output M(s)
   else
      Calculeaza multimea P(s) de perechi posibil de inclus in M(s)
      for each (n,m) in P(s)
         if F(s,n,m) then
            Calculeaza starea s2 obtinuta prin adaugarea perechii (n,m) la M(s)
            call Match(s2)
         end if
      end for
   end if
end procedure
```

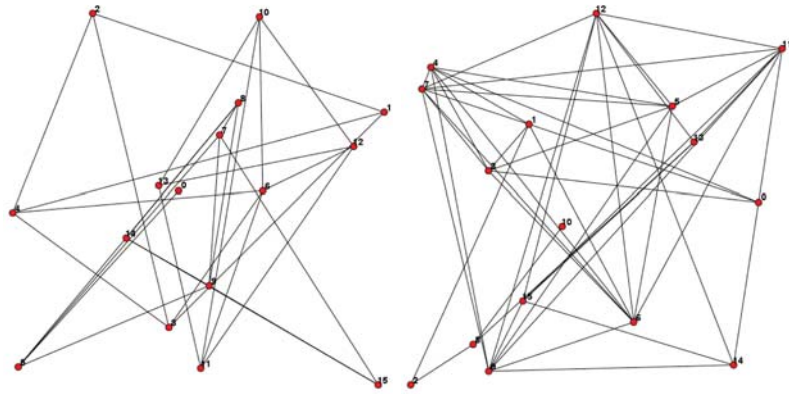
Biblioteca `igraph` pune la dispoziţie metoda `isomorphic()`, disponibilă pentru orice obiect de tip `Graph`, pentru a determina dacă două grafuri sunt izomorfe sau nu. Alte metode `isomorphic_vf2` care implementează algoritmul VF2 [8] şi `isomorphic_bliss` care implementează algoritmul BLISS [21] de testare a izomorfismului sunt de asemenea disponibile. În cele ce urmează vom exemplifica utilizarea acestor metode.

Să generăm două grafuri aleatoare, ambele având 16 noduri. Rezultatul este prezentat în Figura 7.7. Invocând metoda `isomorphic` aflăm dacă cele două grafuri sunt izomorfe sau nu, răspunsul fiind, evident, negativ.

```
>>> g1 = Graph.GRG( 16, 0.4 )
>>> plot( g1 )
>>> g2 = Graph.GRG( 16, 0.4 )
>>> plot( g2 )
>>> g1.isomorphic(g2)
False
```

[6] Arcul (i, j) este diferit de arcul (j, i).

[7] În engleză "to prune".



(a) g1 (b) g2

Figura 7.7: g1 şi g2.

Exerciţii

7.1 Creați două grafuri corespunzătoare formulelor chimice următoare: Na_2SO_4 şi Ag_2SO_4 şi afişați-le. Verificați izomorfismul celor două grafuri, ţinând cont de izomorfismul substanţelor chimice, ambele având structura cristalină hexagonală. Alte exemple de substanţe ce prezintă izomorfism structural: $ZnSO_4$ şi $NiSO_4$ sau $CaCO_3$ şi $NaNO_3$.

```
        O                              O
        ||                             ||
Na - O - S - O - Na            Ag - O - S - O - Ag
        ||                             ||
        O                              O
```

7.2 Problema recunoașterii de caractere dintr-o bază de date, folosind tehnica de graph matching. Încărcați din fişierul 4.txt un graf reprezentând un caracter (4) şi afisați-l. De verificat izomorfismul cu un graf reprezentativ pentru caracterul generic "4", pe care îl creați în prealabil. Repetați operaţiile pentru alte caractere reprezentate sub formă de grafuri, verificându-se izomorfismul dintre ele.

Obţinerea grafului folosind Python şi `igraph`:

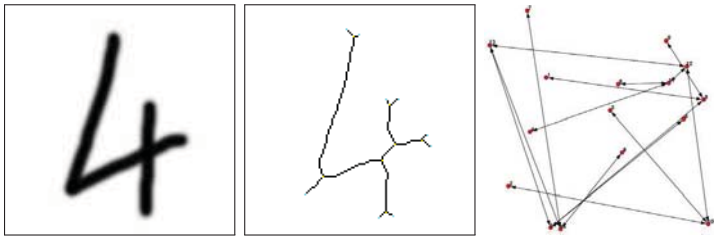
```
from scipy.sparse import *
from scipy import *
from igraph import *
import pdb

f = open('D:/tmp/4Graph.txt', 'w')
g = Graph.Read_Adjacency('4AdMatrix.txt', None, '#', 'distance')
```

```
summary(g)
plot(g)

write(g, 'D:/tmp/4Graph.txt');
f.close()
```

În Figura 7.8 este prezentat un caracter (4), scheletul obținut folosind funcția de scheletonizare disponibilă în MATLAB și graful asociat, obținut prin identificarea în schelet a punctelor ce reprezintă noduri terminale și de intersecție.



(a) original (b) schelet (c) graf

Figura 7.8: Imagine originală reprezentând caracterul “4”, scheletul obţinut şi graful asociat.

Lista de adiacenţă pentru graful corespunzător caracterului “4” este prezentată în Tabelul 7.4.

8 2 60
8 9 50
8 1 1
9 11 3
9 4 23
10 3 1
10 5 1
10 11 22
11 12 52
12 6 1
12 7 2

Tabelul 7.4: Lista de adiacenţă a grafului asociat caracterului “4”.

Matricea de adiacenţă corespunzătoare, având distanţele între noduri reprezentate ca număr de pixeli este următoarea (Tabelul 7.5):

0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	60	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	23	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	2
1	60	0	0	0	0	0	0	50	0	0	0
0	0	0	23	0	0	0	50	0	0	3	0
0	0	1	0	1	0	0	0	0	0	22	0
0	0	0	0	0	0	0	0	3	22	0	52
0	0	0	0	0	1	2	0	0	0	52	0

Tabelul 7.5: Matricea de adiacență a grafului asociat caracterului “4”.

Anexa A

Surse aferente capitolului 4

Pachetul de surse (listate în continuare), împreună cu fisierele proiect pentru Microsoft Visual Studio 2008 si 2010 se pot descărca de la adresa `http://miv.unitbv.ro/asd`.

Exemplul 1

```
/** \file ex1.cu

    ITEMS 2011 - Laborator CUDA - exemplul 1

 Depunerea in memorie al sirului de intregi 0, 1, 2, ... N-1

 1.alocarea memoriei pe dispozitiv si gazda
 2.apelarea kernelului CUDA
 3.asteptarea rezultatului
 4.copierea rezultatului din memoria dispozitiv in memoria gazda.
 5.afisarea sirului rezultat
 */

#include <stdio.h>

/** Codul care va rula pe procesorul grafic */
__global__ void primulKernel(int *a, int n)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;        // indicele threadului
  if (i < n)
    a[i] = i;
}

int main(int argc, char **argv)
{
  int n = 1024;                 // dimensiune
  int *a_cpu;                   // pointer la memoria gazda
  int *a_gpu;                   // pointer la memoria grafica

  a_cpu = (int *)calloc(n, sizeof(int));        // alocare memorie gazda
```

```
  cudaMalloc((void **)&a_gpu, n * sizeof(int)); // alocare memorie grafica

  int threadsPerBlock = 256;    // dimensiune bloc
  int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;// nr. blocuri

  primulKernel <<< blocksPerGrid, threadsPerBlock >>> (a_gpu, n); // apelare GPU
  cudaThreadSynchronize();      // asteptare rezultat

  // copiere rezultat din memoria video in memoria sistemului gazda
  cudaMemcpy(a_cpu, a_gpu, n * sizeof(int), cudaMemcpyDeviceToHost);

  for (int i = 0; i < n; ++i)
    printf("%d ", a_cpu[i]);    // afisare
  printf("\n");

  cudaFree(a_gpu);              // eliberare memorie
  free(a_cpu);
  return 0;
}
```

Exemplul 2

```
/** \file ex2.cu

    ITEMS 2011 - Laborator CUDA - exemplul 2

    Adunarea vectorilor

1.alocarea memoriei pe dispozitiv si gazda
2.apelarea kernelului CUDA
3.asteptarea rezultatului
4.copierea rezultatului din memoria dispozitiv in memoria gazda.
5.afisarea sirului rezultat
*/

#include <stdio.h>
#include <stdlib.h>

/** Codul care va rula pe procesorul grafic */
__global__ void addV(float *c, const float *a, const float *b, int n)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;        // indicele threadului
  if (i < n)
    c[i] = a[i] + b[i];
}

// aloca un bloc de memorie gazda si un bloc de memorie dispozitiv
void allocDual(float **a_cpu, float **a_gpu, int n)
{
  *a_cpu = (float *)calloc(n, sizeof(float));   // alocare memorie gazda
  if (*a_cpu == NULL) {
    perror("Memorie gazda insuficienta");
    exit(1);
  }

  // alocare memorie pe placa grafica
  if (cudaMalloc((void **)a_gpu, n * sizeof(float)) != cudaSuccess) {
```

```
    perror("Memorie dispozitiv insuficienta");
    exit(1);
  }
}

// eliberare memorie gazda si memorie dispozitiv
void freeDual(float *a_cpu, float *a_gpu)
{
  free(a_cpu);
  cudaFree(a_gpu);
}

int main(int argc, char **argv)
{
  int n = 1024;                  // dimensiune
  float *a_cpu, *a_gpu, *b_cpu, *b_gpu, *c_cpu, *c_gpu;

  // alocari 3x2 vectori
  allocDual(&a_cpu, &a_gpu, n);
  allocDual(&b_cpu, &b_gpu, n);
  allocDual(&c_cpu, &c_gpu, n);

  for (int i = 0; i < n; i++) {
    a_cpu[i] = rand() / (float)RAND_MAX;
    b_cpu[i] = rand() / (float)RAND_MAX;
  }

  int threadsPerBlock = 256;    // dimensiune bloc
  int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock; // nr blocuri
  cudaMemcpy(a_gpu, a_cpu, n * sizeof(float), cudaMemcpyHostToDevice);//gpu<-cpu
  cudaMemcpy(b_gpu, b_cpu, n * sizeof(float), cudaMemcpyHostToDevice);//gpu<-cpu
  addV <<< blocksPerGrid, threadsPerBlock >>> (c_gpu, a_gpu, b_gpu, n);//apelGPU
  cudaThreadSynchronize();      // asteptare rezultat

  // copiere vectorului rezultat C din memoria gpu in memoria gazda
  cudaMemcpy(c_cpu, c_gpu, n * sizeof(float), cudaMemcpyDeviceToHost);

  for (int i = 0; i < n; ++i)
    printf("%10f + %10f = %10f\n", a_cpu[i], b_cpu[i], c_cpu[i]);  // afisare
  // eliberare memorie
  freeDual(a_cpu, a_gpu);
  freeDual(b_cpu, b_gpu);
  freeDual(c_cpu, c_gpu);
  return 0;
}
```

Exemplul 3

```
/** \file ex3.cu

 ITEMS 2011 - Laborator CUDA - exemplul 3
 Suma (reductia) paralela.

*/

#include <stdio.h>
#include <stdlib.h>
```

```
__global__ void parallelSum(float *x, int n)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;        // indicele threadului
  for (int s = n / 2; s > 0; s /= 2) {
    if (i < s)
      x[i] += x[i + s];
    __syncthreads();
  }
}

// aloca un bloc de memorie gazda si un bloc de memorie dispozitiv
void allocDual(float **a_cpu, float **a_gpu, int n)
{
  *a_cpu = (float *)calloc(n, sizeof(float));   // alocare memorie gazda
  if (*a_cpu == NULL) {
    perror("Memorie gazda insuficienta");
    exit(1);
  }

  // alocare memorie pe placa grafica
  if (cudaMalloc((void **)a_gpu, n * sizeof(float)) != cudaSuccess) {
    perror("Memorie dispozitiv insuficienta");
    exit(1);
  }
}

// eliberare memorie gazda si memorie dispozitiv
void freeDual(float *a_cpu, float *a_gpu)
{
  free(a_cpu);
  cudaFree(a_gpu);
}

int main(int argc, char **argv)
{
  // dimensiune vector. In acest exemplu, dimensiunea maxima este 1024
  // deoarece folosim un singur bloc de procesare (max 512 threaduri)
  int n = 1024;
  float *x_cpu, *x_gpu;
  float suma;

  allocDual(&x_cpu, &x_gpu, n);
  for (int i = 0; i < n; i++)   // preparare sir de intrare
    x_cpu[i] = (float)(i + 1);

  int threadsPerBlock = n / 2;  // dimensiune bloc
  int blocksPerGrid = 1;
  cudaMemcpy(x_gpu, x_cpu, n * sizeof(float), cudaMemcpyHostToDevice);
  parallelSum <<< blocksPerGrid, threadsPerBlock >>> (x_gpu, n);
  cudaThreadSynchronize();      // asteptare rezultat

  // copiere suma rezultata (un scalar de tip float)
  cudaMemcpy(&suma, x_gpu, 1 * sizeof(float), cudaMemcpyDeviceToHost);

  // afisare rezultat
  printf("suma = %-16.1f control = %-16.1f\n", suma,
```

```
        ((double)n * (n + 1)) / 2.0);

  freeDual(x_cpu, x_gpu);
  return 0;
}
```

Exemplul 4

```
/** \file ex4.cu

   ITEMS 2011 - Laborator CUDA - exemplul 4
   Produsul matriceal.

 Acest cod calculeaza produsul C = A x B, in trei moduri:

 Algoritmul 0 = calcul secvential pe CPU, algoritmul clasic
 Algoritmul 1 = CUDA, mapare simpla
 Algoritmul 2 = CUDA, pe blocuri
 */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "labTimer.h"
#include "labMatrix.h"

template < class T > __host__ void
matMulCPU(Matrix < T > &C, const Matrix < T > &A, const Matrix < T > &B)
{
  int i, j, k;
  for (i = 0; i < C.rows(); i++) {
    for (j = 0; j < C.columns(); j++) {
      float c = 0;
      for (k = 0; k < A.columns(); k++)
        c += A[i][k] * B[k][j];
      C[i][j] = c;
    }
  }
}

// pasam obiectele A,B,C prin valoare!
// CUDA va realiza o copie a obiectelor din RAM in memoria GPU

template < class T > __global__ void
matMulSimpleKernel(Matrix < T > C, const Matrix < T > A, const Matrix < T > B)
{
  int i = threadIdx.y + blockIdx.y * blockDim.y;
  int j = threadIdx.x + blockIdx.x * blockDim.x;
  T sum = 0;
  for (int k = 0; k < A.columns(); ++k)
    sum += A[i][k] * B[k][j];
  C[i][j] = sum;
}

// inmultire C = AxB, prin blocuri de DxD
// D = dimensiune sub-matrici (D x D)
template < class T, unsigned D > __global__ void
```

```
matMulBlockKernel(Matrix < T > C, Matrix < T > A, Matrix < T > B)
{
  __shared__ T As[D][D]; // bloc in memoria partajata, rapida
  __shared__ T Bs[D][D];

  int bi = blockIdx.y*D; // indice inceput bloc (bi,bj)
  int bj = blockIdx.x*D;

  int i = threadIdx.y;   // indice relativ la blocul curent
  int j = threadIdx.x;

  // fiecare thread va calcula un element din C, prin
  // acumularea rezultatului partial in Cvalue
  T Cvalue = 0;          // compilatorul CUDA va aloca un registru pt Cvalue

  // Itereaza sub-matricile lui A si B
  for (int k = 0; k < A.columns(); k+=D) {
    // fiecare thread copiaza un element din memoria globala in memoria rapida
    As[i][j] = A[bi+i][k+j];
    Bs[i][j] = B[k+i][bj+j];

    __syncthreads(); // sincronizare, ne asiguram completarea copierii

    // Multplicare blocuri As * Bs
    for (int e = 0; e < D; ++e)
      Cvalue += As[i][e] * Bs[e][j];
  }

  // Salvam rezultatul in memoria globala
  C[bi+i][bj+j] = Cvalue;
}

// Inmultire de matrici (cod control, CPU)
// alg:
//     0 = calcul serial, CPU (fara CUDA)
//     1 = algoritmul CUDA simplu
//     2 = algoritmul CUDA pe blocuri

template < class T > void
matMul(Matrix < T > &C, const Matrix < T > &A, const Matrix < T > &B, int alg)
{
  const int D = 16;             // dimensiune blocuri (sub-matrici)
  long long dur, t0;
  if (A.columns() % D) {
    fprintf(stderr,
            "EROARE - Dimensiunile matricei trebuie sa fie multiple de %d\n",
            D);
    exit(1);
  }
  if (alg == 0) {
    t0 = getTime();
    matMulCPU < T > (C, A, B);
    dur = getTime() - t0;
  } else {
    Matrix < T > d_A(A.rows(), A.columns(), true);
    Matrix < T > d_B(B.rows(), B.columns(), true);
    Matrix < T > d_C(C.rows(), C.columns(), true);
```

```
    d_A.copyHostToDevice(A);
    d_B.copyHostToDevice(B);
    // Invoke kernel
    dim3 dimBlock(D, D);
    dim3 dimGrid((unsigned)(B.columns() / dimBlock.x),
                 (unsigned)(A.rows() / dimBlock.y));

    printf("threads[%d %d], grid[%d %d] alg=%d\n", dimBlock.x, dimBlock.y,
           dimGrid.x, dimGrid.y, alg);
    t0 = getTime();
    switch (alg) {
    case 0:
      break;
    case 1:
      matMulSimpleKernel < T > <<<dimGrid, dimBlock >>> (d_C, d_A, d_B);
      break;
    case 2:
      matMulBlockKernel < T, D > <<<dimGrid, dimBlock >>> (d_C, d_A, d_B);
      break;
    default:
      fprintf(stderr, "Nr. Algoritm invalid");
      exit(1);
    }
    CUVERIFY(cudaThreadSynchronize());
    dur = getTime() - t0;
    C.copyDeviceToHost(d_C);
    d_A.freeMemory();
    d_B.freeMemory();
    d_C.freeMemory();
  }
  printf("Alg = %d, Timp = %f ms\n", alg, (double)dur / 1000.0);
}

template < class T > void randomM(Matrix < T > &A)
{
  assert(!A.inDeviceMemory());
  for (int i = 0; i < A.rows(); ++i)
    for (int j = 0; j < A.columns(); ++j)
      A[i][j] = (float)rand() / RAND_MAX;
}

template < class T > void printMatrix(const char *name, const Matrix < T > &A)
{
  assert(!A.inDeviceMemory());
  printf("%s [%d x %d]:\n", name, (int)A.rows(), (int)A.columns());
  for (int i = 0; i < A.rows(); ++i) {
    for (int j = 0; j < A.columns(); ++j)
      printf("%8.3f ", A[i][j]);
    printf("\n");
  }
  printf("\n");

}

// calculeaza eroarea medie ||A - B||
template < class T > double
```

```
matDiff(const Matrix < T > &A, const Matrix < T > &B)
{
  double d, s;
  s = 0;
  for (int i = 0; i < A.rows(); ++i) {
    for (int j = 0; j < A.columns(); ++j) {
      d = A[i][j] - B[i][j];
      s += d * d;
    }
  }
  s /= (double)A.rows();
  s /= (double)A.columns();
  return sqrt(s);
}

/**
 * Testeaza inmultirea matricilor.
 *
 * alg  = 0,1,2 tipul algoritmului selectat:
        0 = CPU,
        1 = GPU naiv
        2 = GPU pe blocuri
 *  pr  = true = afisare matrice
 * diff = true se calculeaza si prin alg. secvential si se compara rezultatele
 *                                                                    CPU,GPU
 */

template < class T > void test(int N, int alg, bool pr, bool diff)
{
  Matrix < T > A(N, N, false);
  Matrix < T > B(N, N, false);
  Matrix < T > C(N, N, false);
  Matrix < T > *Ccpu = NULL;

  randomM < T > (A);
  randomM < T > (B);

  if (diff) {
    Ccpu = new Matrix < T > (N, N, false);
    matMulCPU < T > (*Ccpu, A, B);      // pentru control
  }

  matMul < T > (C, A, B, alg);

  if (pr) {
    printMatrix < T > ("A", A);
    printMatrix < T > ("B", B);
    printMatrix < T > ("C", C);
  }
  if (diff) {
    double e = matDiff < T > (C, *Ccpu);
    printf("Eroare = %e %s\n", e, (e > 1E-5) ? "!!!!!" : "");
  }
  A.freeMemory();
  B.freeMemory();
  C.freeMemory();
  if (Ccpu != NULL)
```

```
    delete Ccpu;
}

int main(int argc, char **argv)
{
  int N = 0;
  int alg = 0;
  bool pr = false;
  bool diff = false;
  if (argc < 2) {
    printf("Sintaxa: %s [optiuni]\n", argv[0]);
    printf("Optiuni:\n"
           "    -n N    = dimensiunea matricei (multiplu de 16)\n"
           "    -a alg  = algoritmul folosit:\n"
           "              0 = CPU, secvential\n"
           "              1 = CUDA, prima varianta\n"
           "              2 = CUDA, a doua varianta\n"
           "    -p      = afisare matrice\n"
           "    -t      = test, comparare rezultat GPU cu CPU\n");
    return 1;
  }
  for (int i = 1; i < argc; i++) {
    if (!strcmp(argv[i], "-n"))
      N = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-a"))
      alg = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-p"))
      pr = true;
    else if (!strcmp(argv[i], "-t"))
      diff = true;
    else {
      fprintf(stderr, "Optiune invalida: %s\n", argv[i]);
      return 1;
    }
  }
  if (N < 16 || (N % 16)) {
    fprintf(stderr, "Dimensiunile matricei trebuie sa fie multiple de 16");
    return 1;
  }
  test < float >(N, alg, pr, diff);
  return 0;
}
```

Anexa B

Surse aferente capitolului 5

Exemplul 5

```
/** \file ex5.cu
   ITEMS - Laborator CUDA - exemplul 5
   Lucrul cu texturi si imagini

 */

#include "labImage.h"

// referinta la textura
texture < float4, 2, cudaReadModeElementType > texRef;

/* Re-scalare imagine, prin citire din textura texRef.
 output = adresa destinatie
 stride = numarul de bytes/linie in destinatie
 width,height = dimensiuni destinatie, in pixeli
 */

__global__ void
zoomKernel(float4 * output, size_t stride, int width,
           int height, float zx, float zy)
{
  // indicele thread = coordonate destinatie
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  if (x < width && y < height) {
    float4 f;
    float sx = zx * x;          // coordonate sursa
    float sy = zx * y;
    f = tex2D(texRef, sx, sy);  // acces textura
    output[x + y * (stride / sizeof(float4))] = f;      // stocare rezultat
  }
}

int main(int argc, const char **argv)
{
  Image < float, 4 > img;
```

```
  int BLOCK_WIDTH = 16;          // dimensiuni bloc kernel CUDA
  int BLOCK_HEIGHT = 16;

  float zx = 8;                  // factori scalare (marire)
  float zy = 8;
  // citire fisier intrare
  if (img.load("Lenna.png") < 0)
    return 1;
  img.print("Lenna.png");        // afisare parametri
  // imagine destinatie
  Image < float, 4 > dest((size_t) (img.columns() * zx),
                          (size_t) (img.rows() * zy));

  // setare mod: cudaFilterModeLinear = interpolare
  // cudaFilterModePoint = trunchiere
  texRef.filterMode = cudaFilterModeLinear;

  // texturi adresabile prin coordonate ne-normalizate
  // (0...width, 0...height)
  texRef.normalized = false;
  // copiere imagine din mem. gazda in mem. GPU
  img.hostToDevice();

  // asociere memorie imagine cu referinta texRef
  cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc < float4 > ();
  CUVERIFY(cudaBindTexture2D
           (0, texRef, img.data_gpu, channelDesc, img.columns(), img.rows(),
            img.stride_gpu));

  // invocare nucleu
  dim3 grid((unsigned)round((float)dest.columns() / BLOCK_WIDTH),
            (unsigned)round((float)dest.rows() / BLOCK_HEIGHT));
  dim3 threads(BLOCK_WIDTH, BLOCK_HEIGHT);
  printf("Kernel threads(%d):[%d,%d] grid[%d,%d] ptr_gpu=%p\n", threads.x
         * threads.y, threads.x, threads.y, grid.x, grid.y, dest.data_gpu);
  zoomKernel <<< grid, threads >>> ((float4 *) dest.data_gpu, dest.stride_gpu,
                                    dest.columns(), dest.rows(), 1.0f / zx,
                                    1.0f / zy);
  CUVERIFY(cudaThreadSynchronize());
  // de-asociere textura
  CUVERIFY(cudaUnbindTexture(texRef));

  // copiere rezultat in memoria gazda
  dest.deviceToHost();

  // salvare imagine
  dest.print("out-ex5.png");
  dest.save("out-ex5.png");

  return 0;
}
```

Exemplul 6

```
/** \file ex6.cu

   ITEMS - Laborator CUDA - exemplul 6
```

```
   Interoperabilitatea OpenGL

  - Citire imagine din fisier
  - Copiere in memoria dispozitiv
  - Prelucrare prin CUDA
  - Afisare prin OpenGL, direct din memoria dispozitiv

 */

#include "labImage.h"
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <cuda_gl_interop.h>

// dimensiuni fereastra grafica */
static int windowWidth = 512, windowHeight = 512;
// PixelBufferObject(pbo)
// un PBO este o zona de memorie dispozitiv (video), alocata de OpenGL
// In acest program, pbo va referi bufferul destinatie
static GLuint pbos[1] = { 0 };

// resursa pbo[0] = mapare CUDA
struct cudaGraphicsResource *pboCUDA = NULL;

// imagine sursa
static Image < float, 4 > *sourceImage = NULL;
// referinta CUDA la textura sursa
texture < float4, 2, cudaReadModeElementType > texRef;

static float4 whirlAngle = { 0.5, 0.5, 0, 0 };

/** kernel CUDA: procesare de imagini
 * imaginea sursa se ataseaza texturii texRef, in prealabil
 * output[] = buffer destinatie
 * width,height = dimensiunile imaginii de intrare/iesire
 * stride = numarul de bytes/linie
 * param = parametrii generici (dependente de kernel)
 * */
__global__ void
imageProcessingKernel(float4 * output, int stride,
                      int width, int height, float4 param)
{
  // indicele thread = coordonate destinatie
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  if (x < width && y < height) {
    float dx = x - width / 2.0F;
    float dy = y - height / 2.0F;
    float r = sqrt(dx * dx + dy * dy) / 100.0F;
    float sx = x + r * cos(r * param.x);
    float sy = y + r * sin(r * param.y);
    float4 f = tex2D(texRef, sx, sy);
    output[x + y * (stride / sizeof(float4))] = f;
  }
}
```

```
#define GLCHECKERROR() { int err = glGetError(); \
 if (err) fprintf(stderr, "%s(%d): GL Error: %s\n", __FILE__,__LINE__, (const
     GLchar*)gluErrorString(err)); \
}

// trateaza evenimente mouse (cat timp butonul este apasat)
// x,y = coordonatele cursorului
void motion(int x, int y)
{
  whirlAngle.x = x / 10.0f;
  whirlAngle.y = y / 10.0f;
}

// intoarce (x/n) cu rotunjire 'in sus', pentru argumente pozitive.
inline int ceilDiv(int x, int n)
{
  return (x + n - 1) / n;
}

// invocare procesare imagine
static void compute()
{
  int BLOCK_WIDTH = 16;          // dimensiuni bloc kernel CUDA
  int BLOCK_HEIGHT = 16;
  // ascoiere textura sursa
  cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc < float4 > ();
  CUVERIFY(cudaBindTexture2D(0, texRef,
                             sourceImage->data_gpu, channelDesc,
                             sourceImage->columns(), sourceImage->rows(),
                             sourceImage->stride_gpu));

  // mapare buffer destinatie
  CUVERIFY(cudaGraphicsMapResources(1, &pboCUDA));
  void *devPtr = NULL;
  size_t devSize = 0;
  CUVERIFY(cudaGraphicsResourceGetMappedPointer(&devPtr, &devSize, pboCUDA));
  // devPtr este valid doar in CUDA, pana la cudaGraphicsUnmapResources
  dim3 grid(ceilDiv(sourceImage->columns(), BLOCK_WIDTH),
            ceilDiv(sourceImage->rows(), BLOCK_HEIGHT));
  dim3 threads(BLOCK_WIDTH, BLOCK_HEIGHT);
  imageProcessingKernel <<< grid, threads >>>
      ((float4 *) devPtr, sourceImage->widthBytes(),
       sourceImage->columns(), sourceImage->rows(), whirlAngle);
  CUVERIFY(cudaThreadSynchronize());
  // de-asociere textura sursa
  CUVERIFY(cudaUnbindTexture(texRef));
  // de-asociere destinatia CUDA. devPtr se va invalida
  CUVERIFY(cudaGraphicsUnmapResources(1, &pboCUDA));
}

// apelat, de catre OpenGL, in mod repetat, pentru afisare
static void display()
{
  compute();
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbos[0]);
  // glDrawPixels poate avea implementari sub-performante,
  // varianta este de a folosi dreptunghi+textura
```

```
  glDrawPixels(sourceImage->columns(), sourceImage->rows(), GL_RGBA,
               GL_FLOAT, 0);
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
  glutSwapBuffers();
  //dest->print("dest");
}

// apelat, de catre OpenGL la redimensionarea ferestrei;
// setam proiectie ortografica
static void reshape(int w, int h)
{
  windowWidth = w;
  windowHeight = h;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, windowWidth, 0, windowHeight);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glutPostRedisplay();
}

static void idle()
{
  glutPostRedisplay();
}

// initializare OpenGL
static void initGL(int *argc, char **argv)
{
  glutInit(argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutInitWindowSize(windowWidth, windowHeight);
  glutCreateWindow("Exemplul 5 - CUDA - OPENGL");
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMotionFunc(motion);
  glewInit();
  if (!glewIsSupported("GL_VERSION_2_0 ")) {
    fprintf(stderr, "ERROR: Support for necessary OpenGL extensions missing.");
    fflush(stderr);
    exit(1);
  }
  CUVERIFY(cudaGLSetGLDevice(0));
  // creare PixelBufferObject (OpenGL)
  glGenBuffers(1, pbos);

  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbos[0]);
  glBufferData(GL_PIXEL_UNPACK_BUFFER,
               sourceImage->sizeBytes(), NULL, GL_DYNAMIC_COPY);
  GLCHECKERROR();
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
  // inregistrare PBO pentru CUDA
  CUVERIFY(cudaGraphicsGLRegisterBuffer(&pboCUDA, pbos[0],
                                        cudaGraphicsMapFlagsWriteDiscard));

  // stergere ecran OpenGL
  glClearColor(0.0, 0.0, 0.0, 1.0);
```

```
  glDisable(GL_DEPTH_TEST);
  glutIdleFunc(idle);
}

int main(int argc, char **argv)
{
  sourceImage = new Image < float, 4 > ();
  // citire fisier intrare
  if (sourceImage->load("Lenna.png") < 0)
    return 1;
  sourceImage->print("Lenna.png");      // afisare parametri

  initGL(&argc, argv);          // initalizare OpenGL
  // copiere imagine din mem. gazda in mem. GPU
  sourceImage->hostToDevice();

  // setare mod: cudaFilterModeLinear = interpolare
  // cudaFilterModePoint = trunchiere
  texRef.filterMode = cudaFilterModeLinear;

  // texturi adresabile prin coordonate ne-normalizate
  // (0...width, 0...height)
  texRef.normalized = false;

  printf("Click + Drag pentru a modifica parametrii\n");
  // bucla principala OpenGL, va afisa repetat
  glutMainLoop();

  delete sourceImage;
  return 0;
}
```

Exemplul 7

```
/** \file ex7.cu
 ITEMS - Laborator CUDA - exemplul 7
 Procesare de imagini in domeniul spectral, folosirea biblioteci CUFFT

 - Citire imagine din fisier
 - Copiere in memoria dispozitiv
 - Aplicare transformata Fourier
 - Prelucrare spectru
 - Aplicare transformata inversa Fourier
 - Afisare prin OpenGL, direct din memoria dispozitiv

 !!!! A se rula din directorul 'src' (Lenna.jpg) !!!

 */

#include "labImage.h"
#include <cufft.h>
#include "GL/glew.h"
#include "GL/freeglut.h"

#include "cuda_gl_interop.h"

// dimensiuni fereastra grafica */
```

```
extern int windowWidth, windowHeight;
// PixelBufferObject(pbo)
// un PBO este o zona de memorie dispozitiv (video), alocata de OpenGL
// In acest program, pbo va referi bufferul destinatie
extern GLuint pbos[1];
// resursa pbo[0] = mapare CUDA
extern struct cudaGraphicsResource *pboCUDA;
// imagine sursa, in format RGBA
extern Image < float, 4 > *sourceImage;
// Matrice de numere complexe (float2)
// pentru domeniul spectral
Image < cufftComplex, 1 > *spectrum = NULL;
extern cufftHandle plan;
int fftDims = 2;

// parametrii filtrare, controlabili prin mouse
// params.x = [-1 .. +1]
// params.y = [-1 .. +1]
extern float4 params;
/**
 controleaza tipul afisarii
 true: se afiseaza imaginea reconstruita,
 dupa transf. F. inversa.
 false:  se afiseaza spectrul imaginii (magnitudinile).
 Nu se mai calculeaza transf. F. inversa.
 */
extern bool enableIFFT;

/** selecteaza tipul de filtrare
 0 = fara filtrare (pass-through)
 1 = filtrare Gaussiana
 */
extern int filterType;

/** Conversie din pixel RGBA in nivele de gri,
 *  stocate ca numere complexe.
 *  (Adaptare pentru intrarea FFT)
 *  partea reala = nivel gri
 *  partea imaginara = 0
 *  stride = latimea liniei (in unitati tip)
 */
__global__ void
convertToCplxKernel(cufftComplex * dst, int dst_stride,
                    const float4 * src, int src_stride, int width, int height)
{
  // indicele thread = coordonate destinatie
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  if (x < width && y < height) {
    float4 c = src[x + y * src_stride];
    float gray = saturate(0.2989F * c.x + 0.5870F * c.y + 0.1140F * c.z);
    dst[x + y * dst_stride] = make_float2(gray, 0);
  }
}

/**
 *  Adaptarea din iesirea IFFT (numere complexe) in formatul de afisare RGBA.
```

```
 */
__global__ void
convertFromCplxKernel(float4 * dst, int dst_stride,
                      const cufftComplex * src, int src_stride, int width,
                      int height)
{
  // indicele thread = coordonate destinatie
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  if (x < width && y < height) {
    cufftComplex c = src[x + y * src_stride];
    // normalizare, deoarece FFT a scalat numerele
    float gray = saturate(c.x / (width * height));
    dst[x + y * dst_stride] = make_float4(gray, gray, gray, 1.0F);
  }
}

/**
 * Adaptare spectru 2D pentru afisare:
 * - se reordoneaza cvadrantele (interschimba 1 - 3 si 2 - 4
 * - se calculeaza logartimul magnitudinii valorilor spectrale
 */
__global__ void
convertSpectrumForDisplayKernel(float4 * dst,
                                int dst_stride, const cufftComplex * src,
                                int src_stride, int width, int height)
{
  // indicele thread = coordonate destinatie
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  int x0 = width / 2;
  int y0 = height / 2;
  if (x < width && y < height) {
    int sx = x < x0 ? x + x0 : x - x0;
    int sy = y < y0 ? y + y0 : y - y0;
    cufftComplex c = src[sx + sy * src_stride];
    float gray = saturate(logf(1.0f + cuCabsf(c)) / 10.0F);
    dst[x + y * dst_stride] = make_float4(gray, gray, gray, 1.0F);
  }
}

/** kernel CUDA: procesare de imagini in domeniul spectral
 * Filtrare gaussiana
 *
 * imaginea sursa se ataseaza texturii texRef, in prealabil
 * spect[] = spectru imagine (intrare/iesire)
 * width,height = dimensiunile imaginii de intrare/iesire
 * stride = numarul de bytes/linie
 * param.x = parametrii generici (dependente de kernel)
 * */
__global__ void
gaussianKernel(cufftComplex * spect,
               int stride, int width, int height, float4 param)
{
  // indicele thread = coordonate destinatie
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
```

```
  int x0 = width / 2;
  int y0 = height / 2;
  float2 sigma = make_float2(fabs(param.x) / 2.0F, fabs(param.y) / 2.0F);
  if (x < width && y < height) {
    float2 d = make_float2((float)(x - x0 - param.y) / x0,
                           (float)(y - y0 - param.y) / y0);
    float g =
        exp(-
            (d.x * d.x / (sigma.x * sigma.x) +
             d.y * d.y / (sigma.y * sigma.y)));

    /*
     Nota - este corect sa lipseasca normalizarea cu 1/(2 pi sigma^2)
     deoarece, lucram in domeniul spectraL, Fourier( exp(-x^2/sigma^2)
     iar transf. Fourier a unei Gaussiene da o alta Gaussiana (cu alta varianta)
     iar factorul de scalare devine unitar.
     ref: http://mathworld.wolfram.com/FourierTransformGaussian.html
     */
    int2 s = make_int2(x < x0 ? x + x0 : x - x0, y < y0 ? y + y0 : y - y0);
    cufftComplex c = spect[s.x + s.y * stride];
    c.x *= g;
    c.y *= g;
    spect[s.x + s.y * stride] = c;
  }
}

#define GLCHECKERROR() { int err = glGetError(); \
 if (err) fprintf(stderr, "%s(%d): GL Error: %s\n", __FILE__,__LINE__, \
 (const GLchar*)gluErrorString(err)); \
}
// intoarce (x/n) cu rotunjire 'in sus', pentru argumente pozitive.
inline int ceilDiv(int x, int n)
{
  return (x + n - 1) / n;
}

// invocare procesare imagine
void compute()
{
  int BLOCK_WIDTH = 16;          // dimensiuni bloc kernel CUDA
  int BLOCK_HEIGHT = 16;

  if (!sourceImage->data_gpu || !spectrum->data_gpu)
    return;

  dim3 grid(ceilDiv(sourceImage->columns(), BLOCK_WIDTH),
            ceilDiv(sourceImage->rows(), BLOCK_HEIGHT));
  dim3 threads(BLOCK_WIDTH, BLOCK_HEIGHT);

  // 1. conversie din RGB in nivele de gri
  convertToCplxKernel <<< grid, threads >>> (spectrum->data_gpu,
                                             spectrum->strideGPUT(),
                                             (const float4 *)
                                             sourceImage->data_gpu,
                                             sourceImage->stride_gpu /
                                             sizeof(float4),
                                             sourceImage->columns(),
```

```
                                              sourceImage->rows());
  CUVERIFY(cudaThreadSynchronize());

  // 2. Transformata Fourier

  if (cufftExecC2C(plan, spectrum->data_gpu,
                   spectrum->data_gpu, CUFFT_FORWARD) != CUFFT_SUCCESS) {
    fprintf(stderr, "cufftExecC2C returned error\n");
    return;
  }
  // 3. procesare spectru
  switch (filterType) {
  case 0:                       // nu facem nimic
    break;
  case 1:
    gaussianKernel <<< grid, threads >>> (spectrum->data_gpu,
                                          spectrum->strideGPUT(),
                                          spectrum->columns(),
                                          spectrum->rows(), params);
    break;
  case 2:
    // alte tipuri de filtrari, ca exercitii
    // ... de implementat ...
    break;
  }

  // 4. Transformata Fourier Inversa
  if (enableIFFT)
    cufftExecC2C(plan, spectrum->data_gpu, spectrum->data_gpu, CUFFT_INVERSE);

  // 5. Conversie din complex in RGB (destinatie: bufferul OpenGL)
  // mapare buffer destinatie
  CUVERIFY(cudaGraphicsMapResources(1, &pboCUDA));
  void *devPtr = NULL;
  size_t devSize = 0;
  CUVERIFY(cudaGraphicsResourceGetMappedPointer(&devPtr, &devSize, pboCUDA));
  // devPtr este valid doar in CUDA, pana la cudaGraphicsUnmapResources
  if (enableIFFT) {
    convertFromCplxKernel <<< grid, threads >>>
        ((float4 *) devPtr, sourceImage->stride_gpu / sizeof(float4),
         spectrum->data_gpu, spectrum->strideGPUT(),
         sourceImage->columns(), sourceImage->rows());
  } else {
    convertSpectrumForDisplayKernel <<< grid, threads >>>
        ((float4 *) devPtr, sourceImage->stride_gpu / sizeof(float4),
         spectrum->data_gpu, spectrum->strideGPUT(),
         sourceImage->columns(), sourceImage->rows());
  }

  CUVERIFY(cudaThreadSynchronize());
// de-asociere destinatia CUDA. devPtr se va invalida
  CUVERIFY(cudaGraphicsUnmapResources(1, &pboCUDA));
}

void createSpectrum()
{
  spectrum =
```

```
      new Image < cufftComplex, 1 > (sourceImage->columns(),
                                     sourceImage->rows(), true);
  spectrum->print("Spectrum");
}
```

Exemplul 8

```
/** \file ex8.cu
 ITEMS - Laborator CUDA - exemplul 8

 Lucrul cu biblioteca Thrust

 Exemplu: Sortarea unui vector de intregi, consta din:

 1. Generarea unui vector de numere aleatoare, pe gazda
 2. Transferarea in memoria dispozitiv
 3. Sortarea (prin biblioteca Thrust)
 4. Transferarea rezultatului inapoi in memoria gazda.

 referinta: http://code.google.com/p/thrust/
 */

#include "thrust/host_vector.h"
#include "thrust/device_vector.h"
#include "thrust/generate.h"
#include "thrust/sort.h"
#include "thrust/copy.h"
#include "labTimer.h"

int main(void)
{
  unsigned int N = 1 << 26;
  // generam N numere aleatoare, pe CPU
  thrust::host_vector < int >h_vec(N);
  thrust::generate(h_vec.begin(), h_vec.end(), rand);

  // copiem in memoria dispozitiv
  thrust::device_vector < int >d_vec = h_vec;

  long long t0 = getTime();
  // sortam vectorul pe dispozitiv
  thrust::sort(d_vec.begin(), d_vec.end());
  printf("Sortarea prin GPU      a %10d numere a durat %8d microsecunde\n", N,
         (int)(getTime() - t0));

  t0 = getTime();

  // transferam rezultatul sortat din mem. dispozitiv inapoi in mem. gazda
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
  printf("Transferul DISP->GAZDA a %10d numere a durat %8d microsecunde\n", N,
         (int)(getTime() - t0));

  return 0;
}
```

Exemplul 9

```
/** \file ex9.cu
ITEMS - Laborator CUDA - exemplul 9

Alinierea a doua secvente
algoritmul Needleman-Wunsch
Referinte:

[1] http://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm

[2] Bioinformatics High Performance Parallel Computer Architectures
                                                    CRC Press, 2010 pg. 8

*/

#include "thrust/host_vector.h"
#include "thrust/device_vector.h"

#include "labMatrix.h"
#include "labTimer.h"

static int gap = -2;            // penalizare 'pauza'
static int alpha = 1;           // scor similaritate
static int beta = -1;           // penalizare diferenta


// dim. blocuri. sirurile sunt completate la multiplu de BLOCK_SIZE
static const size_t BLOCK_SIZE = 32;
static double durTotal = 0;

// cat de mult sa afisam:
// 1 = scorul alinierii 2 = viteza 3 = sirurile aliniate
// 4 = matricea 5 = pasii interni
static int verbose = 2;


/** Functia de similaritate intre doua caractere
(se lucreaza cu tipul de date int (nu char) din motive de acces la memorie CUDA)
Se pot defini mai multe tipuri de masuri de similaritate, dependente de problema.
De ex, consultati BLOSUM  (http://en.wikipedia.org/wiki/BLOSUM)
pentru alinierea secventelor de proteine.

Pentru simplitate, in cazul de fata s-a ales o functie simpla.
*/
#define simil(a, b) ((a==b) ? alpha : beta)

/// afiseaza matricea A, pana la dimensiunile n+1, m+1
template < class T > void
dump(const thrust::host_vector < int >&a, const thrust::host_vector < int >&b,
     Matrix < T > A, int n, int m)
{
  int i, j;
  char sep = ' ';               // separator coloane
  printf("     ");
  for (j = 0; j < m + 1; j++)
    printf("%5c%c", (j > 0 && b[j - 1]) ? b[j - 1] : ' ', sep);
```

```
  printf("\n");
  for (i = 0; i < n + 1; i++) {
    printf("%4c%c", (i > 0 && a[i - 1]) ? a[i - 1] : ' ', sep);
    for (j = 0; j < m + 1; j++)
      printf("%5d%c", A[i][j], sep);
    printf("\n");
  }
  printf("\n");
}

/// reconstruieste si afiseaza aliniamentul sirurilor a,b, pornind de la D[n,m]
void
trace_back(Matrix < int >&D, const thrust::host_vector < int >&a,
           const thrust::host_vector < int >&b, int n, int m)
{
  int i, j;
  char *alignmentA = (char *)calloc(1 + n + m, 1);
  char *alignmentB = (char *)calloc(1 + n + m, 1);
  char *pa = alignmentA + n + m;
  char *pb = alignmentB + n + m;
  i = (int)n;
  j = (int)m;
  if (verbose > 4)
    printf("drumul (inapoi): ");
  while (i > 0 && j > 0) {
    int score = D[i][j];
    int scoreDiag = D[i - 1][j - 1];
    int scoreUp = D[i][j - 1];
    int scoreLeft = D[i - 1][j];
    if (score == scoreDiag + simil(a[i - 1], b[j - 1])) {
      *(--pa) = a[i - 1];
      *(--pb) = b[j - 1];
      i--;
      j--;
      if (verbose > 4)
        printf("\\");
    } else if (score == scoreLeft + gap) {
      *(--pa) = a[i - 1];
      *(--pb) = '-';
      i--;
      if (verbose > 4)
        printf("^");
    } else {
      *(--pa) = '-';
      *(--pb) = b[j - 1];
      j--;
      if (verbose > 4)
        printf("<");
    }
  }
  while (i > 0) {
    *(--pa) = a[i - 1];
    *(--pb) = '-';
    i--;
  }
  while (j > 0) {
    *(--pa) = '-';
```

```
    *(--pb) = b[j - 1];
    j--;
  }
  if (verbose > 4)
    printf("\n");
  puts(pa);
  puts(pb);
  free(alignmentA);
  free(alignmentB);
}

/** afisarea timpii de executie
* dur1 = timp in nuclee (milisecunde)
* dur2 = timp total (include transfer memorie, milisecunde)
*/
void printTiming(double dur1, double dur2)
{
  durTotal += dur1;
  if (verbose >= 1) {
    printf(
"Timp de executie nuclee = %.2f ms, timp total (nuclee+transfer)= %.2f ms \n",
        dur1, dur2);
  }
}

// calculeaza aliniamentul sirurilor a,b pe CPU
int
align_cpu(thrust::host_vector < int >&a, thrust::host_vector < int >&b, int n,
          int m)
{
  int i, j;
  Matrix < int >D(n + 1, m + 1, false);
  long long t0 = getTime();
  D[0][0] = 0;
  for (i = 1; i <= n; i++)
    D[i][0] = i * gap;
  for (j = 1; j <= m; j++)
    D[0][j] = j * gap;
  long long t1 = getTime();
  for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++) {
      int x = max(D[i - 1][j - 1] + simil(a[i - 1], b[j - 1]),  //
                  max(D[i - 1][j] + gap,        //
                      D[i][j - 1] + gap));
      // daca modificam cu: max(0, ... ) obtinem algoritmul Smith-Waterman
      D[i][j] = x;
    }
  double dur1 = (double)(getTime() - t1);
  double dur2 = (double)(getTime() - t0);

  if (verbose >= 4) {
    dump < int >(a, b, D, n, m);
  }
  printTiming(dur1 / 1E3, dur2 / 1E3);
  if (verbose >= 3)
    trace_back(D, a, b, n, m);
  return D[n][m];
```

```
}

// Initializeaza prima linie si prima coloana a matricei D
__global__ void initKernel(Matrix < int >D, int gap)
{
  int i, j;
  i = j = (1 + blockIdx.x * blockDim.x + threadIdx.x);
  if (i >= 1 && i < D.rows())
    D[i][0] = i * gap;
  if (j >= 1 && i < D.columns())
    D[0][j] = j * gap;
}

__global__ void
alignKernel(Matrix < int >S, int s, const int *a, const int *b, int alpha,
            int beta, int gap)
{
  int i, j;
  const int n = S.rows();
  const int m = S.columns();
  int t = blockIdx.x * blockDim.x + threadIdx.x;
  i = 1 + (n >= m ? s - t : t);
  j = 1 + (n >= m ? t : s - t);
  if (i < 1 || j < 1 || i >= n || j >= m)
    return;
  int x = max(S[i - 1][j - 1] + simil(a[i - 1], b[j - 1]),      //
              max(S[i - 1][j] + gap,    //
                  S[i][j - 1] + gap));
  S[i][j] = x;
}

__global__ void
alignKernelBlock(Matrix < int >D, int bs, const int *a, const int *b,
                 int alpha, int beta, int gap)
{
  __shared__ int B[BLOCK_SIZE + 1][BLOCK_SIZE + 1];
  int bt = blockIdx.x;
  int bn = (D.rows() - 1) / BLOCK_SIZE;
  int bi, bj;
  if (bs < bn) {
    bi = bs - bt;
    bj = bt;
  } else {
    bj = 1 + bs - bn + bt;
    bi = bn - bt - 1;
  }

  bi *= BLOCK_SIZE;
  bj *= BLOCK_SIZE;
  int t = threadIdx.x;
  // copiem marginea stanga-sus to memoria partajata
  __syncthreads();
  if (bi >= 0 && bj >= 0 && bi < D.rows() && bj < D.columns()) {
    B[0][t + 1] = D[bi][bj + t + 1];
    B[t + 1][0] = D[bi + t + 1][bj];
    if (t == 0)
      B[0][0] = D[bi][bj];
```

```
  }
  __syncthreads();
  // calcul bloc
  int s, i, j;
#pragma unroll
  for (s = 0; s < BLOCK_SIZE; s++) {
    i = s - t;
    j = t;
    i++;
    j++;
    if (i >= 1)
      B[i][j] = max(B[i - 1][j - 1] + simil(a[bi + i - 1], b[bj + j - 1]),
                    max(B[i - 1][j] + gap,
                        B[i][j - 1] + gap));
    __syncthreads();
  }
#pragma unroll
  for (s = BLOCK_SIZE; s < 2 * BLOCK_SIZE - 1; s++) {
    j = 1 + s - BLOCK_SIZE + t;
    i = BLOCK_SIZE - t - 1;
    i++;
    j++;
    if (j >= 1 && j <= BLOCK_SIZE)
      B[i][j] = max(B[i - 1][j - 1] + simil(a[bi + i - 1], b[bj + j - 1]),
                    max(B[i - 1][j] + gap,
                        B[i][j - 1] + gap));
    __syncthreads();
  }

  // copiere in mem. globala
  if (bi >= 0 && bj >= 0 && bi < D.rows() && bj < D.columns())
    for (int i = 1; i <= BLOCK_SIZE; i++) {
      D[bi + i][bj + t + 1] = B[i][t + 1];
    }
}

// diviziune cu rotunjire
static int rdiv(int x, int blockSize)
{
  return (x + blockSize - 1) / blockSize;
}

int
align_gpu(const thrust::host_vector < int >&a,
          const thrust::host_vector < int >&b, size_t n, size_t m)
{
  long long t0 = getTime();
  // copiaza din host_vector in device_vector

  thrust::device_vector < int >A = a;
  thrust::device_vector < int >B = b;
  // alocam matricea in GPU
  Matrix < int >D(a.size() + 1, b.size() + 1, true);
  initKernel <<< BLOCK_SIZE, rdiv(max((int)m, (int)n), BLOCK_SIZE) >>> (D, gap);
  CUVERIFY(cudaThreadSynchronize());
  long long t1 = getTime();
```

```
  for (int s = 0; s < m + n; s++) {
    alignKernel <<< BLOCK_SIZE, rdiv(min((int)m, (int)n), BLOCK_SIZE) >>> (D, s,
        thrust::raw_pointer_cast(&A[0]),thrust::raw_pointer_cast(&B[0]),
                alpha, beta, gap);
    CUVERIFY(cudaThreadSynchronize());
  }
  double dur1 = (double)(getTime() - t1);

  Matrix < int >hD(D.rows(), D.columns(), false);
  hD.copyDeviceToHost(D);
  double dur2 = (double)(getTime() - t0);
  if (verbose >= 4) {
    dump < int >(a, b, hD, n, m);
  }
  printTiming(dur1 / 1E3, dur2 / 1E3);
  if (verbose >= 3)
    trace_back(hD, a, b, n, m);
  D.freeMemory();
  return hD[n][m];
}

int
align_gpuBlock(const thrust::host_vector < int >&a,
               const thrust::host_vector < int >&b, size_t n, size_t m)
{

  long long t0 = getTime();
  // copiaza din host_vector in device_vector

  thrust::device_vector < int >A = a;
  thrust::device_vector < int >B = b;

  // alocam matricea in GPU
  Matrix < int >D(a.size() + 1, b.size() + 1, true);
  Matrix < int >hD(D.rows(), D.columns(), false);
  initKernel <<< BLOCK_SIZE, rdiv(max((int)m, (int)n), BLOCK_SIZE) >>> (D, gap);
  CUVERIFY(cudaThreadSynchronize());
  long long t1 = getTime();

  size_t nstages = rdiv(a.size() + b.size(), BLOCK_SIZE) - 1U;
  if (verbose >= 5)
    printf("m=%d n=%d nstages=%d\n", m, n, nstages);
  int bn = rdiv(n, BLOCK_SIZE);
  int bm = rdiv(m, BLOCK_SIZE);
  for (size_t bs = 0; bs < nstages; bs++) {
    size_t grid = bs < bn ? bs + 1 : nstages - bs;
    if (grid > bm)
      grid = bm;
    if (verbose >= 5)
      printf("alignKernelBlock<<<%d,%d>>>(bs=%d bn=%d)\n", grid, BLOCK_SIZE,
             bs, bn);
    alignKernelBlock <<< grid, BLOCK_SIZE >>> (D, bs,
                thrust::raw_pointer_cast(&A[0]),thrust::raw_pointer_cast(&B[0]),
                alpha, beta, gap);
  }
  CUVERIFY(cudaThreadSynchronize());
  double dur1 = (double)(getTime() - t1);
```

```
  hD.copyDeviceToHost(D);
  double dur2 = (double)(getTime() - t0);
  if (verbose >= 4) {
    dump < int >(a, b, hD, n, m);
  }

  printTiming(dur1 / 1E3, dur2 / 1E3);
  if (verbose >= 3)
    trace_back(hD, a, b, n, m);
  D.freeMemory();
  return hD[n][m];
}

/** conversie din sir de caractere (char[]) in vector<int>
 * alocam vectorul cu dim. multipla de BLOCK_SIZE */

void intVectorFromChar(thrust::host_vector < int >&dst, const char *src)
{
  int i, n, nb;
  n = (int)strlen(src);
  nb = BLOCK_SIZE * rdiv(n, BLOCK_SIZE);
  dst.resize(nb);
  for (i = 0; i < n; i++)
    dst[i] = src[i];
  for (; i < nb; i++)
    dst[i] = ' ';
}

static inline char randchar()
{
  char *A = "ACDEFGHIKLMNPQRSTVWY";
  return A[rand() % 20];

}

int main(int argc, char **argv)
{
  char *sa = NULL;
  char *sb = NULL;
  int alg = 0;
  size_t n = 0, m = 0;
  int score = 0;
  int nrep = 1;
  bool vspec = false;
  for (int i = 1; i < argc; i++) {
    if (!strcmp(argv[i], "-r"))
      n = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-v")) {
      verbose = atoi(argv[++i]);
      vspec = true;
    } else if (!strcmp(argv[i], "-a"))
      alg = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-i"))
      nrep = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-gap"))
      gap = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-alpha"))
```

```
    alpha = atoi(argv[++i]);
  else if (!strcmp(argv[i], "-beta"))
    beta = atoi(argv[++i]);
  else if (argv[i][0] == '-') {
    fprintf(stderr, "Unknown arg %s\n", argv[i]);
    return 1;
  } else {
    if (sa == NULL)
      sa = argv[i];
    else
      sb = argv[i];
  }
}

if (n > 0) {
  srand(1);
  m = n;
  sa = (char *)calloc(1 + n, 1);
  sb = (char *)calloc(1 + m, 1);
  for (int i = 0; i < n; i++) {
    sa[i] = randchar();
    sb[i] = randchar();
  }
} else {
  if (sa)
    n = strlen(sa);
  if (sb)
    m = strlen(sb);
}
if (sa == NULL || sb == NULL) {
  fprintf(stderr,
          "Programul calculeaza alinierea globala a 2 secvente\n"
          "cu ajutorul algoritmului Needleman-Wunsch\n"
          "sintaxa: %s [optiuni] SecventaA SecventaB\n", argv[0]);
  fprintf(stderr,
          "optiuni:\n"
          " -a a       Alege algoritmul:\n"
          "            0 = CPU\n"
          "            1 = CUDA, naiv\n"
          "            2 = CUDA, pe blocuri\n"
          " -gap x     penalizare spatii (-2)\n"
          " -alpha x   scor caractere identice (+1)\n"
          " -beta x    penalizare diferenta (-1)\n"
          " -v nivel   Detalii afisate\n"
          "            1 = scorul alinierii\n"
          "            2 = timp executie\n"
          "            3 = sirurile aliniate\n"
          "            4 = matricea\n"
          " -i iter    Numar iteratii\n"
          "            Algoritmul este rulat de iter ori\n"
          "            pentru a masura timpii de executie\n"
          "\n" " -r n       Genereaza secvente aleatoare de lungime n\n");
  return 1;
}

if (!vspec && n < 80)
  verbose = 3;
```

```
  thrust::host_vector < int >a, b;
  if (verbose >= 5) {
    puts(sa);
    puts(sb);
  }
  intVectorFromChar(a, sa);
  intVectorFromChar(b, sb);
  for (int k = 0; k < nrep; k++)
    switch (alg) {
    case 0:
      score = align_cpu(a, b, n, m);
      break;
    case 1:
      cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
      score = align_gpu(a, b, n, m);
      break;
    case 2:
      cudaDeviceSetCacheConfig(cudaFuncCachePreferShared);
      score = align_gpuBlock(a, b, n, m);
      break;
    default:
      fprintf(stderr, "Algorithm invalid %d\n", alg);
    }

  if (verbose >= 2)
    printf("Scor=%d alpha=%d beta=%d gap=%d\n", score, alpha, beta, gap);

  printf("%.3f", durTotal / nrep);
  fflush(stdout);
  return 0;
}
```

Clasa Image

```
/**
 ITEMS 2011 - Laborator CUDA

 clasa Image
 - reprezinta o imagine bidimensionala in memoria gazda si/sau GPU.
 - metode de incarcare si salvare in fisiere

 */

#pragma once
#include <stdio.h>
#include <stdlib.h>

#include <cuda_runtime_api.h>

/** Verifica rezultatul apelului CUDA, si afiseaza mesaj in caz de eroare */
#ifndef CUVERIFY
#define CUVERIFY(x) {cudaError_t err=(x); if ( err!=cudaSuccess) { \
        fprintf(stderr,"%s(%d): Cuda error: %s\n",__FILE__,__LINE__,\
        cudaGetErrorString(err)); exit(1); } }
#endif

/**
```

```
 * Reprezinta o imagine, in memoria gazda, si optional in
 * memoria dispozitiv (video, GPU)
 * Aceasta clasa reprezinta un pixel ca un tip generic de data T
 *
 * T poate fi:
 *   float, byte
 * N = numarul de componente de culoare
 *     1 = nivel gri
 *     4 = R,G,B,A
 *
 */

template < class T, unsigned N > class Image {
 private:
  size_t width;
  size_t height;
  size_t stride;            // latimea, in octeti a liniei = width*N*sizeof(T)
  bool hasAlpha;
  T *data;
 public:
  // buffer in memoria dispozitiv
   T * data_gpu;
  size_t stride_gpu;        // latimea, in octeti a liniei = width*N*sizeof(T)
 public:
  // constructor
   Image() {
    width = height = stride = 0;
    data = NULL;
    hasAlpha = false;
    data_gpu = NULL;
    stride_gpu = 0;
  } Image(size_t w, size_t h, bool allocGPU = true) {
    width = w;
    height = h;
    stride = w * N * sizeof(T);
    data = (T *) calloc(height, stride);
    data_gpu = NULL;
    stride_gpu = 0;
    hasAlpha = false;
    if (allocGPU)
      allocOnDevice();
  }

 private:
  // inhibam constructorul de copiere
  Image < T, N > (const Image < T, N > &i);

  // inhibarea operatorului =
  Image < T, N > &operator=(const Image < T, N > &i);
 public:
  size_t sizeBytes()const {
    return stride * height;
  }
  size_t widthBytes() const {
    return width * N * sizeof(T);
  }
  size_t strideT() const {
```

```
    return stride / sizeof(T);
  }
  size_t strideGPUT() const {
    return stride_gpu / sizeof(T);
  }
  __device__ __host__ size_t rows() const {
    return height;
  }
  __device__ __host__ size_t columns() const {
    return width;
  }
// Redimensionare (si realocare)
void setSize(int w, int h) {
      if (data && w == width && h == height)
    return;
  width = w;
  height = h;
  stride = w * N * sizeof(T);
  if (width > 0 && height > 0)
    data = (T *) realloc(data, height * stride);
  else {
    if (data)
      free(data);
    data = NULL;
  }
}

// Aloca in memoria GPU
void allocOnDevice()
{
  if (width > 0 && data_gpu == NULL) {
    stride_gpu = 0;
    CUVERIFY(cudaMallocPitch
             ((void **)&data_gpu, &stride_gpu, width * N * sizeof(T), height));
    CUVERIFY(cudaMemset2D
             (data_gpu, stride_gpu, 0, width * N * sizeof(T), height));
  }
}

void freeDeviceMem()
{
  if (data_gpu != NULL) {
    cudaFree(data_gpu);
    data_gpu = NULL;
  }
}

void hostToDevice()           // copiaza din memoria gazda in memoria dispozitiv
{
  if (data == NULL)
    return;
  if (data_gpu == NULL)
    allocOnDevice();

  CUVERIFY(cudaMemcpy2D(data_gpu, stride_gpu, data, stride,
                        width * N * sizeof(T), height, cudaMemcpyHostToDevice));
}
```

```
void deviceToDevice(Image < T, N > &dest)
{
  if (data == NULL)
    return;
  if (dest.data_gpu == NULL)
    dest.allocOnDevice();
  size_t w = width;
  if (dest.width < w)
    w = dest.width;
  size_t h = height;
  if (dest.height < h)
    h = dest.height;
  CUVERIFY(cudaMemcpy2D
           (dest.data_gpu, dest.stride_gpu, data_gpu, stride_gpu,
            w * N * sizeof(T), h, cudaMemcpyDeviceToDevice));
}

void deviceToHost()          // copiaza din memoria dispozitiv in memoria gazda
{
  if (data == NULL || data_gpu == NULL)
    return;
  CUVERIFY(cudaMemcpy2D(data, stride,
                        data_gpu, stride_gpu,
                        width * N * sizeof(T), height, cudaMemcpyDeviceToHost));
}

~Image()
{
  freeDeviceMem();
  if (data)
    free(data);

}

void print(const char *msg)
{
  printf("%s [w=%d h=%d alpha=%d s=%d data_gpu=%p s_gpu=%d]\n",
         msg, (int)width, (int)height, hasAlpha,
         (int)stride, data_gpu, (int)stride_gpu);
}

  /** conversie la format standard Windows Bitmap 32-bit */
template < class Convert >
    void convertToBGR32(unsigned char *dst, unsigned dstStride);

  /** conversie din format standard Windows Bitmap 32-bit */
template < class Convert >
    void convertFromBGR32(const unsigned char *src, size_t srcStride);

 public:
  /* Intrare/iesire */
int load(char const *file_name);
int save(char const *file_name);
};

extern int testLabImage();
```

Clasa Matrix

```
/**
  ITEMS 2011 - Laborator CUDA

    * Clasa Matrix (CPU, GPU)

 */

#pragma once
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/** Verifica rezultatul apelului CUDA, si afiseaza mesaj in caz de eroare */
#ifndef CUVERIFY
#define CUVERIFY(x) {cudaError_t err=(x); if ( err!=cudaSuccess) { \
  fprintf(stderr,"%s(%d): Cuda error: %s\n",__FILE__,__LINE__,\
    cudaGetErrorString(err)); exit(1); } }
#endif
/**
 * Clasa Matrix este duala, accesata atat de CPU cat si de GPU
 *
 */
template < class T > class Matrix {
 private:
  unsigned width;               // numar coloane
  unsigned height;              // numar linii
  unsigned stride;              // distanta dintre elementele pe aceasi coloana
  T *elements;
  bool device;                  // true, daca se afla in memoria dispozitiv
  bool externalAlloc;           // daca mem. fost alocata din afara clasei

 private:
  // inhibarea operatorului =
  __host__ Matrix < T > &operator=(const Matrix < T > &rhs);

 public:
  __device__ __host__ Matrix < T > () {
    width = height = stride = 0;
    elements = NULL;
    device = false;
    externalAlloc = false;
  }
  __host__ Matrix < T > (size_t nrows, size_t ncols, bool dev) {
    this->width = ncols;
    this->height = nrows;
    this->stride = ncols;
    this->device = dev;
    this->externalAlloc = false;
    if (!dev) {
      elements = (T *) calloc(1, sizeBytes());
      if (elements == NULL) {
        fprintf(stderr, "not enough host memory for %ldMB\n",
                (long)(sizeBytes() >> 20));
        fflush(stderr);
        throw "out of memory";
```

```
      }
    } else {
      CUVERIFY(cudaMalloc(&elements, sizeBytes()));
      CUVERIFY(cudaMemset(elements, 0, sizeBytes()));
    }
  }

  __host__
  Matrix < T > (T * externalData, size_t nrows, size_t ncols, bool dev) {
    this->width = ncols;
    this->height = nrows;
    this->stride = ncols;
    this->device = dev;
    this->externalAlloc = true;
    this->elements = externalData;
  }

// constructorul de copiere este folosit la pasarea matricei spre kernelul CUDA
  __host__ Matrix < T > (const Matrix < T > &s) {
    this->width = s.width;
    this->height = s.height;
    this->stride = s.stride;
    this->elements = s.elements;
    this->device = s.device;
    // copia nu mai e 'proprietar' la pointerul elements!
    this->externalAlloc = true;
  }

/** pointer catre inceputul liniei i */
  __device__ __host__ T *operator [] (size_t i) {
    return elements + (i * stride);
  }

  __device__ __host__ const T *operator [] (size_t i) const {
    return elements + (i * stride);
  }
// intoarce submatricea (i,j), de dimensiuni DxD
// submatrice va contine pointer catre matricea-parinte
  __device__ __host__ Matrix < T > subMatrix(unsigned i, unsigned j, int D)
  {
    Matrix < T > S;
    S.width = D;
    S.height = D;
    S.stride = this->stride;
    S.elements = &elements[stride * D * i + D * j];
    S.device = this->device;
    S.externalAlloc = true;
    return S;
  }

  // o alta varianta permite dimensiuni nepatratice
  __device__ __host__
   Matrix < T > subMatrix(unsigned i, unsigned j, unsigned subRows,
                          unsigned subCols) {
    Matrix < T > S;
    S.width = subCols;
    S.height = subRows;
```

```
    S.stride = this->stride;
    S.elements = &elements[stride * i + j];
    S.device = this->device;
    S.externalAlloc = true;
    return S;
  }
  __device__ __host__ size_t sizeBytes() const {
    return stride * height * sizeof(T);
  }
  __device__ __host__ size_t rows() const {
    return height;
  }
  __device__ __host__ size_t columns() const {
    return width;
  }
  __device__ __host__ bool inDeviceMemory() const {
    return device;
  }
  __host__ void freeMemory() {
    if (elements == NULL)
      return;
    if (!externalAlloc) {
      if (device)
        cudaFree(elements);
      else
        ::free(elements);
    }
    elements = NULL;
  }

  __host__ __device__ ~ Matrix() {
#ifndef __CUDACC__
    freeMemory();
#endif
  }
  __host__ void copyHostToDevice(const Matrix < T > &src) {
    assert(this->device && elements != NULL && src.elements != NULL
           && !src.device);
    CUVERIFY(cudaMemcpy
             (this->elements, src.elements, sizeBytes(),
              cudaMemcpyHostToDevice));
  }

  __host__ void copyDeviceToHost(const Matrix < T > &src) {
    assert(!this->device && elements != NULL && src.elements != NULL
           && src.device);
    CUVERIFY(cudaMemcpy
             (this->elements, src.elements, sizeBytes(),
              cudaMemcpyDeviceToHost));
  }
}; // class Matrix
```

Funcția Timer

```
/** \file labTimer.h
```

```
 Acest fisier contine functia
 getTime() care returneaza timpul curent in microsecunde
 (de la pornirea programului).

*/

#pragma once

#if defined(WIN32) || defined(_WINDOWS) || defined(_WIN32)

#include <windows.h>
// in microsecunde (10^-6 sec)
unsigned long long getTime()
{
  LARGE_INTEGER t;
  static LARGE_INTEGER timer_t0, timer_f;
  static bool timer_inited = false;
  if (!timer_inited) {
    if (!QueryPerformanceCounter(&timer_t0)) {
      fprintf(stderr, "QueryPerformanceCounter FAILED, cannot measure time\n");
      timer_t0.QuadPart = 0;
    }
    if (!QueryPerformanceFrequency(&timer_f)) {
      fprintf(stderr,
              "QueryPerformanceFrequency FAILED, cannot measure time\n");
      timer_f.QuadPart = 1000;
    }
    timer_inited = true;
  }
  QueryPerformanceCounter(&t);
  fflush(stdout);
  return (t.QuadPart - timer_t0.QuadPart) * 1000000LL / timer_f.QuadPart;
}

#else
// linux
#include <time.h>
#include <sys/time.h>
unsigned long long getTime()
{

  static bool timer_inited = false;
  static struct timeval timer_t0;

  struct timeval t;

  if (!timer_inited) {
    gettimeofday(&timer_t0, NULL);
    timer_inited = true;
  }
  gettimeofday(&t, NULL);
  return 1000000ULL * (t.tv_sec - timer_t0.tv_sec) + t.tv_usec;
}

#endif
```

Bibliografie

[1] Acton, S. T. şi A. C. Bovik: *Basic linear filtering with application image enhancement.* The Handbook of Image and Video Processing, Second Edition, paginile 99–108, 2005.

[2] Andonie, Răzvan şi Ilie Gârbacea: *Algoritmi fundamentali - o perspectivă C++.* Editura Libris, Cluj-Napoca, 1995, ISBN 973-96494-5-9. `http://www.cwu.edu/~andonie/Cartea%20de%20algoritmi/cartea%20de%20algoritmi.pdf`.

[3] Arlazarov, V. L. *et al.*: *On economical construction of the transitive closure of a directed graph.* Soviet Mathematics—Doklady, 11(5):1209–1210, 1970.

[4] Asanović, Krste, David Patterson, *et al.*: *The Manycore Revolution: Will the HPC Community Lead or Follow?* SciDAC Review, 2009.

[5] Asanovic, Krste, David Patterson, *et al.*: *The Landscape of Parallel Computing Research: A View from Berkeley.* Raport tehnic, EECS Department, University of California, Berkeley, 2006.

[6] Berlekamp, E., J. Conway, şi R. Guy: *Winning Ways for your Mathematical Plays*, volumul 2. Academic, 1982.

[7] Blake, G., R. G. Dreslinski, şi T. Mudge: *A survey of multicore processors.* Signal Processing Magazine, IEEE, 26(6):26–37, octombrie 2009.

[8] Cordella, L. P. *et al.*: *An improved algorithm for matching large graphs.* Î *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, paginile 149–159, 2001.

[9] Csardi, Gabor şi Tamas Nepusz: *The igraph software package for complex network research.* InterJournal, Complex Systems:1695, 2006. `http://igraph.sf.net`.

[10] Eddy, Sean R.: *Where did the BLOSUM62 alignment score matrix come from?* Nature Biotechnology, 22(8):1035–1036, 2004.

[11] Fabbri, Ricardo *et al.*: *2D Euclidean distance transform algorithms: A comparative survey.* ACM Comput. Surv., 40:2:1–2:44, February 2008.

[12] Fortin, S.: *The graph isomorphism problem*. Department of Computing Science, University of Alberta, 1996.

[13] *FreeImage - Open Source Library for graphics image formats* , 2012. `http://freeimage.sourceforge.net/`.

[14] Garey, Michael R. şi David S. Johnson: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[15] Gonzalez, Rafael C. şi Richard E. Woods: *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Ediţia 2nd, 2001, ISBN 0201180758.

[16] Greenlaw, Raymond, H. James Hoover, şi Walter L. Ruzzo: *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[17] Harris, Mark: *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology, 2008.

[18] Hoberock, Jared şi Nathan Bell: *An Introduction to Thrust*, 2008. `http://thrust.googlecode.com/files/AnIntroductionToThrust.pdf`.

[19] Itis: *Integrated taxonomy information system*, 2012. `http://www.itis.gov/`.

[20] Jones, Neil C. şi Pavel A. Pevzner: *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)*. The MIT Press, 2004, ISBN 0262101068.

[21] Junttila, Tommi şi Petteri Kaski: *Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs*.

[22] Khronos OpenCL Working Group: *The OpenCL Specification, version 1.1*, 2010. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[23] Kirk, David B. şi Wen mei W. Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Ediţia 1st, 2010, ISBN 0123814723, 9780123814722.

[24] Kumar, Vipin *et al.*: *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994, ISBN 0-8053-3170-0.

[25] Maliţa, Mihaela şi Gheorghe Ştefan: *On the Many-Processor Paradigm*. Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing, 2008.

[26] Microsoft: *Mediul de dezvoltare Visual C++ 2008 Express Edition*, 2011. `http://msdn.microsoft.com/en-us/express/future/bb421473`.

[27] Microsoft Corporation: *Compute Shader Overview (DirectX documentation)*, 2011. `http://msdn.microsoft.com/en-us/library/ff476331.aspx`.

[28] Needleman, S. B. şi C. D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of molecular biology, 48(3):443–453, 1970.

[29] Neider, Jackie şi Tom Davis: *OpenGL Programming Guide: The Official Guide to Learning OpenGL.* Addison-Wesley Longman Publishing Co., Inc., 2005. `http://glprogramming.com/red/`.

[30] NVIDIA Corporation: *CUDA C Programming Guide*, 2011. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

[31] NVIDIA Corporation: *Pachetul de dezvoltare CUDA 4.1*, 2012. `http://developer.nvidia.com/cuda-toolkit-41`.

[32] NVIDIA Corporation: *The CUFFT Library*, 2012. `http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf`.

[33] Oprisescu, S. *et al.*: *Action Recognition for Simple and Complex Actions using Time of Flight Cameras.* Î *International Conference of Image Processing, Computer Vision & Pattern Recognition*, 2009.

[34] *The Python Tutorial*, 2012. `http://docs.python.org/tutorial/`.

[35] *The Python Language Reference*, 2012. `http://docs.python.org/reference/`.

[36] *The Python Standard Library*, 2012. `http://docs.python.org/library/`.

[37] Reingold, E.M., J. Nievergelt, şi N. Deo: *Combinatorial algorithms: theory and practice.* Prentice Hall College Div, 1977, ISBN 013152447X.

[38] Rixner, Scott: *Stream processor architecture.* Kluwer Academic Publishers, Norwell, MA, USA, 2002, ISBN 0-7923-7545-9.

[39] Rosenfeld, Azriel: *Adjacency in digital pictures.* Information and Control, 26(1):24 – 33, 1974.

[40] Rosenfeld, Azriel: *Digital topology.* The American Mathematical Monthly, 86(8):621 – 630, 1979.

[41] Siddiqi, Kaleem *et al.*: *Shock Graphs and Shape Matching.* Î *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, paginile 222–, Washington, DC, USA, 1998. IEEE Computer Society.

[42] Smith, Steven W.: *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997, ISBN 0-9660176-3-3.

[43] Stephens, Robert: *A survey of stream processing*. Acta Informatica, 34:491–541, 1997, ISSN 0001-5903.

[44] University, Indiana: *Tree print Application*, 2012. `http://iubio.bio.indiana.edu/treeapp/treeprint-form.html`.

[45] Volkov, Vasily şi James Demmel: *LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs*. Raport tehnic UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html`.

[46] Wikipedia: *Chemical file format — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/wiki/Chemical_file_format`.

[47] Wikipedia: *Graph isomorphism — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/wiki/Graph_isomorphism`.

[48] Wikipedia: *Newick Format — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/wiki/Newick_format`.

[49] Wikipedia: *Phylogenetic tree — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/wiki/Phylogenetic_tree`.

[50] Wikipedia: *Protein Data Bank file format — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/wiki/Protein_Data_Bank_(file_format)`.

[51] Wikipedia: *Subgraph isomorphism — Wikipedia, The Free Encyclopedia*, 2012. `http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem`.

[52] Zhang, Y. Y. şi P.S. P. Wang: *A Parallel Thinning Algorithm with Two-Subiteration that Generates One-Pixel-Wide Skeletons*. Pattern Recognition, International Conference on, 4:457, 1996.

Titlul programului: Programul Operațional Sectorial Dezvoltarea Resurselor Umane 2007 - 2013
Titlul proiectului: Tehnici de Analiză, Modelare și Simulare pentru Imagistică, Bioinformatică și Sisteme Complexe (ITEMS)
Editorul materialului: Universitatea POLITEHNICA din București și Universitatea TRANSILVANIA din Brașov
Data publicării: aprilie 2012

Răzvan Andonie este profesor universitar, conducător de doctorat la Universitatea Transilvania din Brașov. Este absolvent al Facultății de Matematică-Informatică, Universitatea Babeș-Bolyai din Cluj-Napoca. A obținut în 1984 doctoratul în matematică-informatică la Universitatea București, sub conducerea Acad. Solomon Marcus. Este activ în următoarele domenii de cercetare: inteligență computațională, sisteme de învățare, calculul paralel și distribuit, chimie computațională.

Angel Cațaron este cadru didactic titular la Departamentul de Automatică, Electronică și Calculatoare al Universității Transilvania din Brașov. A obținut titlul de doctor în anul 2004 la Universitatea Politehnica din București. Domeniile sale de interes sunt data mining și inteligență computațională.

Zoltán Gáspár este cadru didactic asociat la Universitatea Transivania din Brașov. În 2006 a absolvit specializarea Electronică Aplicată, iar în 2011 a primit titlul de doctor în electronică și telecomunicații în cadrul Universității Transilvania din Brașov. Domeniile sale de interes sunt arhitectura microprocesoarelor și arhitecturi de sisteme paralele și distribuite.

Honorius Gâlmeanu este inginer software la Siemens România și cadru didactic asociat la Universitatea Transilvania din Brașov. A obținut titlul de doctor în 2008 la aceeași universitate, subiectul tezei fiind "support vector machines". În cadrul specializării Calculatoare predă cursurile de algoritmi și arhitecturi paralele și algoritmi și calculabilitate. Domeniile sale de interes sunt data mining și bioinformatică, în special alinierea secvențelor.

Mihai Ivanovici este cadru didactic titular la Universitatea Transilvania din Brașov, unde predă cursurile de bazele prelucrării semnalelor și prelucrarea și analiza imaginilor. În 2006 a obținut titlul de doctor în electronică și telecomunicații la Universitatea Politehnica din București, pe baza rezultatelor cercetărilor desfășurate la CERN, Geneva, sub conducerea prof. Vasile Buzuloiu. Domeniile sale de interes includ analiza texturilor, segmentarea imaginilor și prelucrarea și analiza imaginilor biomedicale.

István Lőrentz a obținut diploma de inginer în anul 2001 și actualmente este doctorand al Facultății de Inginerie Electrică și Știința Calculatoarelor, Universitatea Transilvania din Brașov, având ca temă de cercetare algoritmii de calcul paralel pe arhitecturi multi-core, utilizând CUDA și OpenCL, sub conducerea prof. dr. Răzvan Andonie. Domeniile sale de interes sunt procesarea imaginilor, bioinformatica, calculul evolutiv.

Lucian Sasu este doctor in știinta calculatoarelor din 2006 și lector universitar în cadrul Universității Transilvania din Brașov, Facultatea de Matematică și Informatică. Principalele sale domenii de interes sunt algoritmii și structurile de date, inteligența artificială, "machine learning" și data mining.

ISBN 978-606-19-0058-9