\documentclass[11pt,a4paper]{article}

%Packages

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\input{contents/packages}

\title{%

Algoritmusok Optimalizálása GPU-n

}

\author{Jost Márk Benedek}

\date{\today}

\begin{document}

%Titlepage

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\input{contents/titlepage}

% Table of Contents

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\tableofcontents

\newpage

\section{Összefoglaló}

Önálló laboratóriumi tanulmányaim célja az volt, hogy különböző, nagy jelentőséggel bíró algoritmusok futását gyorsítsam videókártyán párhuzamosítás segítségével. Vizsgálódásom során először polinomidejű algoritmusokat, majd bizonyítottan NP-teljes problémákat valósítottam meg, közben kipróbáltam több elviekben különböző ötletet is arra, hogy a gyorsítás ne menjen a végeredmény helyességének kárára.

% Felhasznált Irodalom

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\input{contents/Felhasznalt\_Irodalom}

\section{Motiváció}

Az 1980-as években megjelentek az első személyi számítógépek (PC-k), melyek központi feldolgozóegységei (CPU) mintegy 1MHz-es belső órajellel működtek. Akkor az volt a számítástechnikai fejlesztők fő célja, hogy az órajelfrekvencia emelésével növeljék a számítási gyorsaságot. Számos kiváló mérnöki megoldás született, talán az egyik legjelentősebb a fáziszárt hurok, ami nélkül képtelenség lett volna 50-60 MHz fölé menni. Nagyjából 30 évvel később elérték, hogy a legtöbb asztali processzor órajele 1GHz és 4GHz között legyen, ami az eredeti PC-k frekvenciájának több ezerszerese. Napjainkban változás látható a fejlesztési trendekben, ugyanis az órajelnövelést a processzorok disszipációja felülről korlátozza. Egyelőre nem tűnik könnyen lehetségesnek 5GHz fölé menni úgy, hogy közben az eszköz helyes működése garantálható legyen.

A számítógépgyártók éppen ezért új, alternatív megoldásokat kerestek a számítási teljesítmény növelésére. Legjobb ötletnek a feladatok párhuzamosítása bizonyult. Napjainkban a kutatásoknak két nagy témája van. Egyik a kvantumszámítószépek témája, amit dolgozatomban nem részletezek. A másik a párhuzamosítás. Már a CPU-k fejlesztésénél is megfigyelhető, hogy inkább a minél több processzormag telepítése az iparági trend.

\begin{figure}[ht!]

\centering

\includegraphics[width=0.5\textwidth]{Images/CPU-cores-trend.png}

\caption{Látható, hogy kb. 2010-re befejeződött a CPU-k órajelfrekvencia-növekedése, helyette egyre nőni kezdtek a magok számai. \cite{CPUcores} }

\end{figure}

A párhuzamosításra azonban sokkal alkalmasabb a grafikus segédprocesszor, a GPU. Amíg a CPU feladata, hogy műveletek egy adott szekvenciáját, és annak minden utasítását a lehető leggyorsabban hajtsa végre, addig a GPU célja minél több szál (akár több ezer) párhuzamos futtatása. A videókártyák előnye akkor válik láthatóvá, ha ugyanazt az utasítást több, nagy adattömbön kell végrehajtani. Ez az úgynevezettt SIMD megközelítés (Single Instruction Multiple Data). \cite{kvantum\_optim}

Az []. ábra szemlélteti, hogy a GPU-n arányában több tranzisztor van adatfeldolgozásra rendelvelve, cserébe a gyorsítótárazás és a folyamatvezérlés kisebb hangsúlyt kapott.

\begin{figure}[ht!]

\centering

\includegraphics[width=0.5\textwidth]{Images/gpu-devotes-more-transistors-to-data-processing.png}

\caption{Látható, hogy a gyorsítótárak és a vezérlés rovására nőtt az adatfeldolgozásra szánt tranzisztorok számára. Ez alkalmas lebegőpontos műveletek nagyfokú párhuzamosítására. \cite{CUDAdoc} }

\end{figure}

A videókártya sokkal nagyobb utasítás-áteresztőképességet, valamint memória-sávszélességet biztosít, mint a CPU hasonló ár és energiafogyasztás mellett. Egyéb számítási eszközök, például FPGA-k is lehetnek nagyon energiatakarékosak, viszont azok sokkal kevésbé rugalmasan programozhatóak a GPU-knál, és a fejlesztési idő is sokkal hosszabb lesz. \cite{CUDAdoc}

\section{Technológiai háttér}

Ebben a fejezetben szeretném ismertetni az általam felhasznált szoftver és hardver együttest. Először a grafikus segédprocesszoron történő általános célú programozást tárgyalom, majd bemutatom az Nvidia\textsuperscript{TM} által erre kifejlesztett párhuzamos számítási platformot, a CUDA keretrendszert. Részletezem a megértéshez szükséges fontosabb fogalmakat, valamint bemutatom a CUDA platformon fejlesztéshez elengedhetetlen lépéseket.

\subsection{GPGPU}

A GPGPU (general-purpose computing on graphics processing units) egy olyan

szoftverfejlesztési gyakorlat, melynek során a grafikus feldolgozóegységet (GPU) általános

célú számítási műveletek elvégzésére használjuk. \cite{kvantum\_optim}

\subsection{CUDA}

A választásom a CUDA platformra esett. A CUDA (Compute Unified Device Architecture) egy, az NVIDIA által feljesztett párhuzamos számítási platform, amely szoftveres támogatást nyújt az ezzel kompatibilis grafikus feldolgozóegységek általános célú programozására. A gyártó bővítményt is adott ki, mely a Visual Studio nevű fejlesztői környezetbe importálható. Telepítés után ha új projekt létrehozását választjuk (File/New/Project), akkor "CUDA [verziószám] Runtime" néven kiválasztható a projekt típusának a CUDA. A programozás C vagy C++ nyelven történhet, melyhez minimális nyelvi kiegészítéseket tettek, hogy többek között a szálkezelés egyszerűbbé váljon. A CUDA használatához elérhető egy rendkívül kiterjedt dokumentáció a gyártó weboldalán. \cite{CUDAdoc}

\subsection{Programozási modell}

A továbbiakban összefoglalom a legfontosabb fogalmakat úgy, hogy ismertetem, hogyan lettek megvalósítva C++ nyelven.

\subsubsection{Kernel, és a többi függvénytípus}

A programozó speciális függvényeket definiálhat, melyeket kernelnek nevezünk. A kernel létesít kapcsolatot a CPU (host) és GPU (device) között úgy, hogy előbbi meghívja a függvényt, majd átadja utóbbinak a vezérlést, tehát a kernel a videókártyán fut. Minden egyes kernel példányt egy számára megadott szál hajt végre.

A kernel a "\textit{\\_\\_global\\_\\_}" kulcsszóval definiálható. Ezt a függvény fejléc elé kell írni, ekkor tudja a szoftverkörnyezet, hogy mostantól GPU-kódként értelmezze.

(Megjegyzendő, hogy egyéb kulcsszavak is léteznek. Egyik a "\textit{\\_\\_host\\_\\_}", ami jelzi, hogy CPU által hívott, majd ugyanúgy általa végrehajtandó kód következik. Ha nem adunk meg egy függvény elé kulcsszót, akkor azt tiszta CPU kódként értelmezi, úgy, mintha nem is lenne a szoftverkörnyezet mögött a CUDA platform. Másik a "\textit{\\_\\_device\\_\\_}", amely tisztán GPU függvényt jelez. A két kulcsszó vegyíthető: amennyiben azt írjuk, hogy "\textit{\\_\\_global\\_\\_ \\_\\_host\\_\\_}", a fordító ezt minden egyes híváskor a végrehajtó saját kódjának tekinti, vagyis nem hajt végre vezérlésátadást. Utóbbi hasznosítható például függvénykönyvtárak GPU-ra kiterjesztésére.

\paragraph{}

Az, hogy a kernelt egy adott híváskor hány CUDA szálon szeretnénk futtatni, az új nyelvi elemként megjelenő <<< · · · >>> végrehajtási konfiguráció szintaxissal specifikálható. Sajnos a Visual Studio még szintaxishibaként kezeli [verziószám], ezért a programozó érdemes, hogy odafigyeljen, milyen IntelliSense hibaüzeneket vesz figyelembe. Minden, a kernelt végrehajtó szál egy egyedi thread azonosítót kap, mely a beépített threadIdx változón keresztül érhető el a kernelből.

\paragraph{Példa:} A hivatalos dokumentáció az alábbi példát adja kernel definícióra. A kódrészlet az N méretű A és B vektorok összeadását végzi és az eredményt a C vektorban tárolja:

\begin{lstlisting}[style=CStyle]

// Kernel definition

\_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C)

{

int i = threadIdx.x;

C[i] = A[i] + B[i];

}

int main()

{

...

// Kernel invocation with N threads

VecAdd<<<1, N>>>(A, B, C);

...

}

\end{lstlisting}

\section{Mátrixműveletek vizsgálata}

Dolgozatom célja minél jobb módszereket találni olyan algoritmusokra, amik gráfokon futnak. A gráfok egyik legalapvetőbb megadása a szomszédossági mátrixával történő reprezentáció. Ennek előnye, hogy az egyes csúcsokhoz tartozó élek könnyen, direkt lekérdezhetőek, ezért a rajtuk végzett műveletek gyorsabbak lehetnek. Hátránya azonban, hogy minél kevesebb él van a gráfban, annál helypazarlóbbá válik ez a megadás. Később tekintünk egyéb megadási módokat is, de előbb érdemes megvizsgálnunk, adott tárolási mód mellett a futásidő hogyan változik különböző programozási paradigmák esetén.

\subsubsection{Implementációs választási lehetőségek}

\paragraph{Input:}

Szükségünk van inputmátrix megadására. Kezdetleges C programokban ezt úgy tettük meg, hogy a felhasználótól kértük az adatok begépelését konzolba. Ez nem szép megoldás, nagy méret esetén pedig egyszerűen kivitelezhetetlen. Programom egy fájlból olvassa be az adatokat, mely fájl nevét pedig parancssori argumentumban várja. Ennek a C típusú megvalósítása például ilyen módon történhet:

\begin{lstlisting}[style=CStyle]

// Variables for reading file

FILE\* pFile; // File pointer

int fileNameIdx;

bool foundMatrix = false; // For error handling

int size; // Number of matrix rows (=matrix coloumns)

int i; // Iterator

for (i = 1; i < argc; ++i) // Processing command line arguments

{

// Command Line Syntax: ... --m [file\_name]

if ((strcmp(argv[i], "--m") == 0))

{

pFile = fopen(argv[++i], "r"); // Opening file to read

if (pFile == NULL) // Unable to open file

{

return -1;

}

fileNameIdx = i;

foundMatrix = true;

}

}

if (!foundMatrix)

{

fprintf(stderr, "Required Command Line Syntax: --m [data\_file]\n");

return -1;

}

// Read data

// ...

if (fclose(pFile) != 0) // Unable to close file

{

return -1;

}

// Process data

// ...

\end{lstlisting}

\paragraph{Adattárolás módja:}

A mátrixelemeket valahogyan el kell tárolnunk ahhoz, hogy műveleteket végezhessünk rajtuk.

Nem mindegy, hogy milyen adattípust kap egy elemet tároló változó. Integer jelen esetben nem alkalmas választás, mert az input lehet nem egész számokból álló mátrix is. Válasszunk lebegőpontos ábrázolást, ebből a két népszerű szabványos példa a float és a double. Ahhoz, hogy a későbbiekben össze tudjuk hasonlítani a két opciót, vezessük be az alábbi define-t:

\begin{lstlisting}[style=CStyle]

// Define for used data types: now chosen double

#define DATATYPE double

\end{lstlisting}

Most éppen double-t választottunk használt adattípusnak. Ha változtatni szeretnénk, elég itt, egy helyen átírni a kódot pl. float-ra.

Másik fontos megfontolás az adatstuktúra megválasztása. Statikus tömböt nem használhatunk, mert fordítási időben nem ismert az input mérete. Láncolt listákat device kódban nem támogat jelenleg a CUDA keretrendszer (létezik a Thrust nevű kezdeményezés \cite{Thrust}, amely host kódban enged device-on listamemória használatára, de most ennek használatára nem térek ki). Én dinamikus tömbben tárolom a mátrixot. Mátrixelemeket lehet egy- illetve kétdimenziós tömbben is tárolni. Egydimenziós tömb foglalására példakód:

\begin{lstlisting}[style=CStyle]

// size read from file

DATATYPE\* Matrix = (DATATYPE\*)calloc(size \* size, sizeof(DATATYPE));

\end{lstlisting}

\paragraph{Blokkszám:}

Szinkronizáció szempontjából nagyon nem mindegy, hogy hány konkurens blokkban fut programunk. (Lásd grid szinkonizáció, szükséges ide másolni később!). Jelenleg minden mátrixelemre jut egy külön szál, ezért most nincs ráhatásunk a blokkok számára (blokkméretnek érdemes a lehető legnagyobbat választani), más alkalmazások esetén erre figyelnünk kell.

\subsection{Determinás számítása}

Tekintsünk egy alapvető lineáris algebrai fogalmat, a determinánst. A determinánsszámítás egy művelet, ami egy NxN-es (négyzetes) mátrixhoz egy skalárszámot rendel. Egy mátrix determinánsának kiszámítására számos módszer létezik, én most Gauss-eliminációt alkalmazok.

\newline Matematikából ismertek a determinás alábbi tulajdonságai (teljesség igénye nélkül):

\begin{enumerate}

\item ha egy mátrix egyik sorához (vagy oszlopához) egy másik sor (vagy oszlop) skalárral való szorzatát adjuk, az a determináns értékét nem változtatja

\item két sor (vagy oszlop) felcserélése esetén a determináns a (-1)-szeresére változik

\item felsőháromszög-mátrix determinánsa kiszámolható a főátlóbeli elemek összeszorzásával

\end{enumerate}

Előbbi tulajdonságok alapján ha Gauss-elimináció segítségével felsőháromszög-mátrixszá alakítjuk az eredeti mátrixot, annak determinánsa a főátlóbeli elemek szorzata. Megtörténhet, hogy az elimináció közben az aktuális vezérelem 0. Ekkor ha van az adott oszlopban alatta nemnulla elem, akkor sorcserét végzünk. Ha nincs alatta nemnulla, akkor a determináns nulla.

\subsubsection{Device kód }

Az alábbi GPU kódhoz illeszthető kiszolgáló CPU kód. A teljes kód megtekinthető a mellékelt github repo-ban (majd később lesz hozzáadva).

Segédfüggvényem 0 vezérelem esetén megvizsgálja, hogy ki lehet-e cserélni az adott sort egy alatta levővel.

\begin{lstlisting}[style=CStyle]

// Megkeresi a vezérelem alatti első nem nulla elemet az oszlopban

// Visszatérési értéke, hogy talált-e

\_\_device\_\_ bool firstNotZero(DATATYPE\* Matrix, int size, int k, int\* idx)

{

int i;

for (i = k + 1; i < size; ++i) {

if (Matrix[i \* size + k]) {

\*idx = i;

return true;

}

}

return false;

}

\end{lstlisting}

Az 1x1-es Gridek esetén ez a kernel hívandó. Látható, hogy több egyszerűsítés is meg van benne téve, ami kihasználja a Grid méretét.

\begin{lstlisting}[style=CStyle]

// 1x1-es Grid

// Grid: 1x1

\_\_global\_\_ void detKernel\_1Block(DATATYPE\* Matrix, int size, DATATYPE\* det)

{

cooperative\_groups::thread\_block block = cooperative\_groups::this\_thread\_block();

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

int j = blockIdx.y \* blockDim.y + threadIdx.y; // sorváltozó

if (i >= size || j >= size)

return;

int k;

int idx; // Sorcserénél használt változó

DATATYPE temp;

\_\_shared\_\_ int sign;

\_\_shared\_\_ bool fullZeroColoumn;

// Kezdeti érték adás a 0. thread által

if (i == 0 && j == 0)

{

sign = 1;

fullZeroColoumn = false;

}

// Gauss elimináció

for (k = 0; k < size - 1; ++k)

{

//a már nem kellő szálak kiléphetnek

if (i < k || j < k)

return;

block.sync();

// Mi van akkor, amikor a vezérelem 0?

if (Matrix[k \* size + k] == 0) {

// Keresünk másik sort, ahol nem 0 vezérelem van különben det=0

// Mindig csak az egyik szálon teszteljük a vezérelem 0 voltát mert megosztott változót állítunk

if (i == k && j == k)

{

fullZeroColoumn = !firstNotZero(Matrix, size, k, &idx);

if (fullZeroColoumn) {

// Csupa 0 oszlopot találtunk

\*det = 0;

}

sign = -sign;

}

// Bevárjuk a [k][k] threadet, hogy megfelelőre állítsa a változót

block.sync();

// A többi threadet is értesítjük arról ha kész vagyunk; értéket már nem kell állítaniuk

if (fullZeroColoumn)

return;

// Kicseréljük a k. és idx. sort: ilyenkor a determináns a -1 -szeresére változik

// 1 dimenziós párhuzam, mert vektorművelet

block.sync();

if (i == k) {

temp1 = Matrix[k \* size + j];

temp2 = Matrix[idx \* size + j];

Matrix[k \* size + j] = temp2;

Matrix[idx \* size + j] = temp1;

}

// Sorcsere után újabb szinkronizáció

block.sync();

}

// Nem nulla a vezérelem, kezdődhet a Gauss elimináció,

// a k-adik oszlopot felesleges kinullázni, többet nem kellenek

if (i > k && j > k)

Matrix[i \* size + j] -= Matrix[i \* size + k] / Matrix[k \* size + k] \* Matrix[k \* size + j];

}

// Deternináns a főátlóbeli elemek szorzata alapján

if (i == size - 1 && j == size - 1)

{

\*det = sign;

for (k = 0; k < size; ++k)

\*det \*= Matrix[k \* size + k];

}

};

\end{lstlisting}

A több blokkból álló gridek esetében a multiblokk kernel hívandó. \newline

Itt a szinkronizációért a \textit{grid.sync()} API hívás felelős. A függvény használ globális változókat annak érdekében, hogy elérje mindegyik thread blokk. Ez nem feltétlen szerencsés megközelítés, másik lehetséges megoldás az, hogy függvényparaméterként adjuk át őket.

\begin{lstlisting}[style=CStyle]

// Számoljuk a sorcserék számának paritását

\_\_device\_\_ int global\_sign;

// Ha bármikor csupa 0 oszlopot találunk, tudjuk, hogy 0 a determináns

\_\_device\_\_ bool global\_fullZeroColoumn;

// Grid: >1x1

\_\_global\_\_ void detKernel\_multiBlock(DATATYPE\* Matrix, int size, DATATYPE\* det) {

// Szinkronizációs változó a teljes griden belül

cooperative\_groups::grid\_group grid = cooperative\_groups::this\_grid();

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

int j = blockIdx.y \* blockDim.y + threadIdx.y; // sorváltozó

int k;

int idx; // Sorcserénél használt változó

DATATYPE temp1, temp2;

if (!grid.is\_valid())

return;

grid.sync();

// Kezdeti érték adás a 0. thread által

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

global\_sign = 1;

global\_fullZeroColoumn = false;

}

for (k = 0; k < size - 1; ++k)

{

grid.sync();

// Mi van akkor amikor a vezérelem 0?

if (Matrix[k \* size + k] == 0)

{

// Keresünk másik sort, ahol nem 0 vezérelem van különben det=0

// Mindig csak az egyik szálon teszteljük a vezérelem 0 voltát, mert megosztott változót állítunk

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

global\_fullZeroColoumn = !firstNotZero(Matrix, size, k, &idx);

if (global\_fullZeroColoumn) {

// Csupa 0 oszlopot találtunk

\*det = 0;

}

global\_sign = -global\_sign;

}

// Bevárjuk a [k][k] threadet, hogy megfelelőre állítsa a változót

grid.sync();

// A többi threadet is értesítjük arról, ha kész vagyunk; értéket már nem kell állítaniuk

if (global\_fullZeroColoumn)

return;

grid.sync();

if (i == k) {

temp1 = Matrix[k \* size + j];

temp2 = Matrix[idx \* size + j];

Matrix[k \* size + j] = temp2;

Matrix[idx \* size + j] = temp1;

}

// Sorcsere után újabb szinkronizáció

grid.sync();

}

// Nem nulla a vezérelem, kezdődhet a Gauss elimináció

// a k-adik oszlopot felesleges kinullázni, többet nem kellenek

if (i > k && j > k && i < size && j < size) // diagnosztika végett nem j>=k lehetséges

Matrix[i \* size + j] -= Matrix[i \* size + k] / Matrix[k \* size + k] \* Matrix[k \* size + j];

}

if (i == size - 1 && j == size - 1)

{

\*det = global\_sign;

for (k = 0; k < size; ++k)

\*det \*= Matrix[k \* size + k];

}

};

\end{lstlisting}

\subsubsection{Tesztelések}

A profilozást az NVIDIA Nsight\textsuperscript{TM} Compute programjának segítségével végeztem.

\newline

Verziószám: 2023.1.0.0 (build 32451174) (public-release)

A determinánsszámító programomat többféle módon teszteltem: több inputra megnéztem a futásidőt, a pontos eredménytől való százalékos eltérést (referenciának a Wolfram Alpha \cite{WolframAlpha}, illetve a Matrix Reshish \cite{matrixCalculator} nevű internetes számológépekkel ellenőriztem): egydimenziós tárolás float adattípusokon, egydimenziós tárolás double adattípusokon, kétdimenziós tárolás float adattípusokon, és végül kétdimenziós tárolás double adattípusokon. Az eredményeket az alábbi táblázat tartalmazza.

Az eredmény összehasonlítását automatizálhatjuk azzal, hogy a kiegészítő program saját maga hasonlítja össze a GPU által számolt eredményt az elméleti helyes eredménnyel. Én parancssori argumentumba adom meg opcionálisan a helyes eredményt, és ekkor \%-ban megadva kiírja a program a relatív hibát. Ennek egy fajta implementálása:

\begin{lstlisting}[style=CStyle]

bool givenSolution = false;

double solution;

int i; // Iterator

for (i = 1; i < argc; ++i) // Processing command line arguments

{

// Command Line Syntax: ... --s [Solution]

if ((strcmp(argv[i], "--s") == 0) || (strcmp(argv[i], "--Solution") == 0))

{

if (sscanf(argv[++i],"%lf", &solution) != 1) {

fprintf(stderr, "Unable to read given solution!\n");

}

else {

printf("Given solution : %f\n", solution);

givenSolution = true;

}

}

}

// Process data

// ...

if (givenSolution) {

double errorPercentage = abs((double)calculatedSolution - solution) / solution \* 100;

printf("Difference from correct solution: %.12f %%\n", errorPercentage);

}

\end{lstlisting}

\subsection{Egy lépéssel tovább: Mátrix inverz számítás}

Az előző fejezetben taglalt determinánsnak sok alkalmazása van a lineáris algebrában, az egyik az, hogy felhasználható egy mátrix inverzének meghatározása során. Matematikából ismert képlet:

$A^{-1} = (adj A)/(det A)$ , ahol A\textsuperscript{-1} az A mátrix inverze, adj A az adjungáltmátrix, azaz az eredeti mátrix előjeles aldeterminánsaiból alkotott mátrix transzponáltja, det A pedig a mátrix determinánsa.

Én jelen esetben egy másik algoritmust alkalmaztam: az LU dekompozíció alapú mátrixinvertálást. Elvileg gyorsabb algoritmus, cserébe speciális esetekben nem működik (amikor az inputnak nem létezik LU dekompozíciója, csak PLU dekompozíciója, P a permutációmátrix, nem foglalkozunk most vele). Ilyen esetben a program figyelmeztet, hogy nem tud működni. Továbbfejlesztési lehetőség, hogy amikor nem létezik LU felbontás, olyankor egy másik (potenciálisan lassabb) algoritmus fut le.

\paragraph{LU dekompozíció}

Egy matematikai művelet, mely során egy négyzetes mátrixot felbontunk egy felsőháromszögmátrix (U), valamint egy alsóháromszögmátrix (L) szorzatára.

Az LU dekompozíció a következőképpen alkalmas invertálás számítására: [kifejtés később]

Belátható, hogy az így végzett invertálás O(n\textsuperscript{3})-ös algoritmust eredményez. Algoritmusom során megpróbálok hol n, hol n\textsuperscript{2} számú műveletet végezni, ezáltal gyorsítani a futást. Neminvertálható mátrixoknak is létezhet LU felbontása, ezért az invertáló kernel előtt lefuttatom az előzőekben bemutatott determináns számító kernelt azért, hogy elkerüljünk különböző hibázási lehetőségeket.

\subsubsection{Device kód}

Az alábbi GPU kódhoz illeszthető kiszolgáló CPU kód. A teljes kód megtekinthető a mellékelt github repo-ban [majd később lesz hozzáadva].

Egy blokkos esetben alábbi kernel futtatandó:

\begin{lstlisting}[style=CStyle]

\_\_global\_\_ void inversionKernel\_1Block(DATATYPE\* Matrix, DATATYPE\* InvMatrix,

DATATYPE\* L, DATATYPE\* U, DATATYPE\* Z, int size)

{

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

int j = blockIdx.y \* blockDim.y + threadIdx.y; // sorváltozó

if (i >= size || j >= size)

return;

thread\_block block = this\_thread\_block();

// 1. lepes: LU dekompozíció

// Előformázás

U[i \* size + j] = Matrix[i \* size + j];

if (i < j)

L[i \* size + j] = 0;

else if (i == j)

L[i \* size + i] = 1;

block.sync();

// Gauss elimináció az U mátrixban

for (int m = 0; m < size; m++)

{

if (U[m \* size + m] == 0)

{

if (i == 0 && j == 0)

{

printf("Inversion not calculatable with LU Decomposition!\n");

}

return;

}

DATATYPE x = U[i \* size + m] / U[m \* size + m]; // Ez soronként ugyanaz

if (j == m && i > m)

{

// Ha a nullázandó oszlopban vagyunk

U[i \* size + j] = 0;

L[i \* size + j] = x;

}

else if (i > m && j > m) {

U[i \* size + j] -= x \* U[m \* size + j];

}

block.sync();

}

// Felbontottuk a mátrixot egy felsőháromszög-, és egy alsóháromszögmátrixra

int k;

int l;

Z[i \* size + j] = 0;

block.sync();

// 1 dimenziós párhuzamosítás (szinkronizálások mindenkire vonatkoznak)

// size db teljesen független müveletvégzés Matrix[][i] oszlopvektorokon

// 2. lepes: = (oszlopvektor, azért van i j forditva) meghatározésa

if (j == 0)

{

Z[i \* size + i] = 1;

}

block.sync();

if (j == 0)

{

for (k = i + 1; k < size; k++)

{

for (l = 0; l < k; l++)

{

Z[k \* size + i] -= Z[l \* size + i] \* L[k \* size + l];

}

}

}

block.sync();

// Elkészült Z

// 3. lépés: InvMatrix[][i] oszlopvektor meghatározása, módszer: backpropagation

InvMatrix[j \* size + i] = Z[j \* size + i];

block.sync();

if (j == 0)

{

for (k = size - 1; k >= 0; k--)

{

for (l = k + 1; l < size; l++)

{

InvMatrix[k \* size + i] -= U[k \* size + l] \* InvMatrix[l \* size + i];

}

InvMatrix[k \* size + i] /= U[k \* size + k];

}

}

}

\end{lstlisting}

A több blokkos kivitel hasonlóan megírható, teljes kód megtalálható a repo-ban. [később adom hozzá!]

\subsubsection{Tesztelések}

A teszteléshez inputnak különböző méretű mátrixokat használtam. Megvizsgáltam, hogy az adattípus, valamint a blokkszám hogyan befolyásolja a futásidőt, valamint az eredményt. Blokkszámot úgy lehet befolyásolni, ha módosítjuk a CUDA rendszerre jellemző blokkméretet. (Megjegyzés: ezt már azért is megéri megtenni, mert különböző architektúrákon kisebb vagy nagyobb lehet egy blokkon belül maximálisan futtatható szálak száma, ezáltal más-más kártyákon jelentősen eltérő teljesítményt tapasztalhatunk).

Összehasonlítási alapon matlabban is elvégeztem a beépített inv(Matrix) utasítással a műveletet, és összehasonlítottam az eredményeket. A matlab eredményét tekintem referenciának.

Az eredmény összehasonlítását automatizálhatjuk azzal, hogy a kiegészítő program saját maga hasonlítja össze a GPU által számolt eredményt az elméleti helyes eredménnyel. Én parancssori argumentumba adom meg opcionálisan a helyes eredményt, és ekkor \%-ban megadva kiírja a program az átlagos, illetve maximális (a mátrixelemekre vett) relatív hibát. Ennek egy fajta C nyelvű

implementálása (ha matlab segítségével kiírjuk fájlba az általa számolt eredményt):

\begin{lstlisting}[style=CStyle]

FILE\* pSolutionFile = NULL;

bool givenSolution = false;

double\* Solution;

int i; // Iterator

for (i = 1; i < argc; ++i) // Processing command line arguments

{

// Command Line Syntax: ... --s [SolutionFileName]

if ((strcmp(argv[i], "--s") == 0) || (strcmp(argv[i], "--Solution") == 0))

{

pSolutionFile = fopen(argv[++i], "r");

if (pSolutionFile == NULL)

{

fprintf(stderr, "Unable to open file \"%s\"", argv[i]);

return -1;

}

else

{

givenSolution = true;

}

}

}

// ...

// Reading solution values

if (givenSolution)

{

Solution = (double\*)malloc(size \* size \* sizeof(double));

for (int ii = 0; ii < size; ++ii)

{

double temp;

for (int jj = 0; jj < size; ++jj)

{

if (fscanf\_s(pSolutionFile, "%lf ", &temp) != 1)

{

fprintf(stderr, "Error reading SolutionMatrix(%d %d)\n", ii, jj);

fclose(pFile);

fclose(pSolutionFile);

return -1;

}

Solution[ii \* size + jj] = temp;

}

//fscanf\_s(pFile "\n");

}

fclose(pSolutionFile);

}

// Calculations ...

// Testing the punctuality of the results

if (givenSolution)

{

double sumError = 0.0;

double maxError = 0.0;

for (i = 0; i < size \* size; i++)

{

double erroruPercentage = abs(((double)InvMatrix[i] - Solution[i]) / Solution[i]) \* 100000000;

sumError != erroruPercentage;

if (erroruPercentage > maxError)

maxError = erroruPercentage;

}

sumError /= size \* size;

printf("Average difference from correct solution: %.12f u%%\n", sumError);

printf("Max difference from correct solution: %.12f u%%\n", maxError);

}

\end{lstlisting}

Látható, hogy mikroszázalékban adja meg a hibát (10\textsuperscript{-8}) azért mert a double esetében gyakran ilyen pontossággal kell összehasonlítani ahhoz, hogy eltérést tapasztaljunk.

\subsection{Eredmények értékelése}

Az eredmények alapján sokat gyorsíthat az eljáráson, hogyha egyszeres pontosságú lebegőpontos számokkal dolgozunk (ilyenkor az eredmény pontosságára oda kell figyelni). Főleg kisebb memóriájú videókártyák esetén lehet lényeges a spórolás: például ha szükségünk van 10 millió változóra, és a double 8bájtos, a float pedig 4 bájtos, akkor float használatával 40millió bájtot, azaz mintegy 38MB memóriát takaríthatunk meg.

Másik észrevétel, hogy kevéssel ugyan, de gyorsabb futású a program, ha egydimenziós tömböt használunk. Ez nagyobb odafigyelést igényel, viszont eredményes lesz, ha megtesszük.

\section{Genetikus algoritmusok vizsgálata}

Ebben a fejezetben útkeresési problémákkal foglalkom, amelyek NP-teljes problémák, ezért ún. genetikus algoritmusokkal igyekszem közelíteni a végeredményt.

\paragraph{Jármű útvonaltervezési probléma \cite{alg\_optim}}

A jármű útvonaltervezési probléma (Vehicle Routing Problem, továbbiakban: VRP) egy fajta optimalizációs probléma, mely során járműveknek (itt: a teljes) útiköltség minimalizálásával kell végigjárniuk megadott célpontokat. A problémát különböző megkötésekkel lehet feltenni az alkalmazás igénye alapján. Ezek lehetnek például:

\begin{itemize}

\item járművek maximális száma

\item az egyes járművek szállítási kapacitása

\item az egyes járművek által megtehető maximális úthossz

\end{itemize}

Az egyszerűség kedvéért a járművek ugyanabból a pontból indulnak (0. gráfcsúcs, warehouse), a kiindulási pontot leszámítva minden állomást legalább egyszer érinteniük kell és az útjuk végén vissza kell térniük a közös kiindulópontba. A járművek minden állomáson pontosan egyszer rakhatnak le árut, és részleges kiszállítás nincs megengedve.

Matematikai megfogalmazás : a problémát gráfokkal modellezhetjük.

\newline

Legyen

Legyen G = (V,E) gráf, n az állomások száma (a ), m az állomások között futó elérhető utak száma, 0 a kiindulási állomás, k az elérhető járművek maximális száma.

\[ V = {v\_0,v\_1, ... v\_{n-1} }\] állomások halmaza,

\[ E = {e\_1,e\_2, ... e\_m}\] elérhető utak halmaza,

\[ D : E(G) -> \mathbb{Z}\] költségfüggvény,

\[0 \in V\]

Az élek azonosítása érdekében éljünk a következő jelöléssel: e\textsubscript{ij} a v\textsubscript{i}-ből v\textsubscript{j}-be mutató él, és d\textsubscript{ij} az e\textsubscript{ij} költsége (itt távolság, Distance : nagy betű, mert mátrix). Adott minden csúcshoz a c\textsubscript{i} áruigény, amit ki kell elégíteni (ez a valóságban lehet darab, kg, stb. ). Legyen k\textsubscript{i} az i-edik jármű szállítási kapacitása.

Állítsuk elő útvonalak (Route) olyan R\textsubscript{i} listáit, ahol

\[ R\_i = {0, v\_{i\_1}, v\_{i\_2}, ... 0}) \] az i-edik jármű azon útvonalát adja meg, amelyet alkotó élek

\[(e\_{0,i\_1}, e\_{i\_1,i\_2}, … e\_{i\_m,0})\] . Az útvonal költsége az azt alkotó élek összköltsége.

\paragraph{Utazóügynök probléma \cite{alg\_optim}}

Az utazóügynök probléma (Travelling Salesman Problem, TSP) a VRP egy speciális esete, amikor egyetlen jármű van és nincsenek egyéb korlátozó feltételek. A nevét onnan kapta, hogy a XX. században utazó porszívóügynökök autóval járták az Egyesült Államok útjait házalás céljából. Az olajválság során megdrágult az üzemanyag, és nagyon nem lett mindegy, hogy mekkora utat tesznek meg egy üzleti út során.

\newline

A problémának azóta több alkalmazása is lett, ebből a villamosmérnöki gyakorlathoz egyik legközelebb az SMD beültetőgép bejárása áll. A gép feladata, hogy egy adott nyákterv alapján lepakolja az alkatrészeket a NYÁK lemezre. Az iparban fontos a sebesség, ugyanis ha felére csökkentjük a beültetési időt, akkor akár duplaanyi terméket gyárthatunk azonos idő alatt.

\newpage

\printbibliography

\end{document}