VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULACE CELULÁRNÍCH AUTOMATŮ NA GPGPU

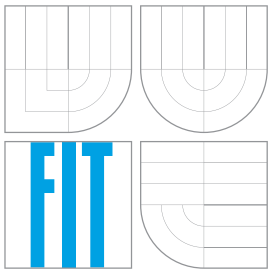
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PŘEMYSL VLČEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULACE CELULÁRNÍCH AUTOMATŮ NA GPGPU

SIMULATION OF CELLULAR AUTOMATA ON GPGPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PŘEMYSL VLČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. PAVOL KORČEK
SUPERVISOR

BRNO 2014

Abstrakt

Cílem práce je navrhnout a otestovat akceleraci speciálního případu celulárního automatu nazývaném Nagel-Schreckenbergova modelu mikrosimulace dopravy bez grafického výstupu na různých platformách a následně naměřené výsledky porovnat.

Abstract

The goal of this thesis is to develop and test an acceleration of special case of celular automata called Nagel-Schreckenberg model of traffic microsimulation without a graphic output on different platforms and then compare the measured results.

Klíčová slova

mikrosimulace dopravy, Nagel-Schreckenberg, CUDA, OpenCL, SSE, akcelerace na GP-GPU, GPU

Keywords

traffic microsimulation, Nagel-Schreckenberg, CUDA, OpenCL, SSE, GPGPU acceleration, GPU

Citace

Přemysl Vlček: Simulace celulárních automatů na GPGPU, bakalářská práce, Brno, FIT VUT v Brně, 2014

Simulace celulárních automatů na GPGPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Pavla Korčeka

.....
Přemysl Vlček
21. května 2014

Poděkování

Chtěl bych poděkovat vedoucímu bakalářské práce panu Ing. Pavlovi Korčekovi za poskytnutí odborných materiálů.

Obsah

Kapitola 1

Úvod

Simulace dopravy je v současnosti, díky čím dál se zvětšujícímu počtu jak osobních tak nákladních vozidel na silnicích, jednou z klíčových veličin pro plánování ekonomiky státu. Existují státní instituce zabývající se dopravou, její kontrolou a budováním nové infrastruktury. Jelikož je výstavba nové a udržování již vybudované dopravní sitě velmi nákladnou investicí, měla by proběhnout po předchozím důkladném a efektivním naplánování, aby se ušetřily peníze, které by v budoucnu byly třeba na úpravy a chyby způsobené špatným a nedostatečným návrhem, k čemuž nám můžou pomoci různé simulační modely dopravy.

Existuje nepřeberné množství prostředků pro simulování dopravy na počítačích. Některé z nich provádí simulaci z makroskopického hlediska, tedy zajímají je spíše statistická data než chování jednotlivých entit podílejících se na vzniku těchto dat. Protikladem k nim jsou simulátory, které zkoumají chování dopravy z mikroskopického hlediska na základě chování jednotlivých entit. Jedním z těchto mikrosimulátorů je i v této práci rozebíraný Nagel-Schreckenbergův model, který využívá matematického modelu zvaného Celulární Automat.

Díky svým vlastnostem jsou grafické karty vhodné pro simulaci procesů, které lze počítat pro obrovské množství veličin paralelně. Právě takovým procesem je simulace celulárních automatů, které jsou díky svým vlastnostem více než vhodnými kandidáty. Navíc v době, kdy je možné provádět paralelní výpočty pomocí speciálních instruckí jak na procesoru, tak s velkou intenzitou na grafické kartě, se jeví klasický sekvenční přístup pomocí klasických programovacích metod, jako plýtvání výkonem.

Cílem práce je tedy akcelerace vybraného celulárního automatu (Nagel-Schreckenbergova modelu) na GPGPU pomocí vhodných dostupných a rozšířených prostředků, následné měření a porovnání výkonnosti jednotlivých implementací tohoto automatu na různých platformách, kterými jsou procesory architektury x86 s a bez použití speciálních instrukcí pro paralelní výpočty a grafické karty běžně používané v osobních počítačích s podporou GPGPU výpočtů.

Kapitola 2

Návrh

2.1 Objektově orientovaný návrh aplikace

Program byl napsán v objektově orientovaném jazyce C++ a byl vyvíjený v prostředí Microsoft Visual Studio 2012, byl úspešně přeložen a otestován na 64-bitovém OS Linux 12.04.4 LTS, Precise Pangolin. Knihovny pořebné pro kompilaci jsou následující: OpenCL verze 1.2 a vyšší pro simulaci v OpenCL, NVidia Cuda verze 5.5 pro simulaci na CUDA zařízení, která byla v době psaní programu aktualní. Vyšší verze CUDA nebyla testována. OpenCL verze 1.1 a vyšší. Z hardwarových prostředků je to pro Cuda simulaci grafická karta s compute capability 2.0 a vyšší. Pro simulaci s pomocí SSE instrukcí je třeba CPU s podporou SSE4.1 a vyšší.

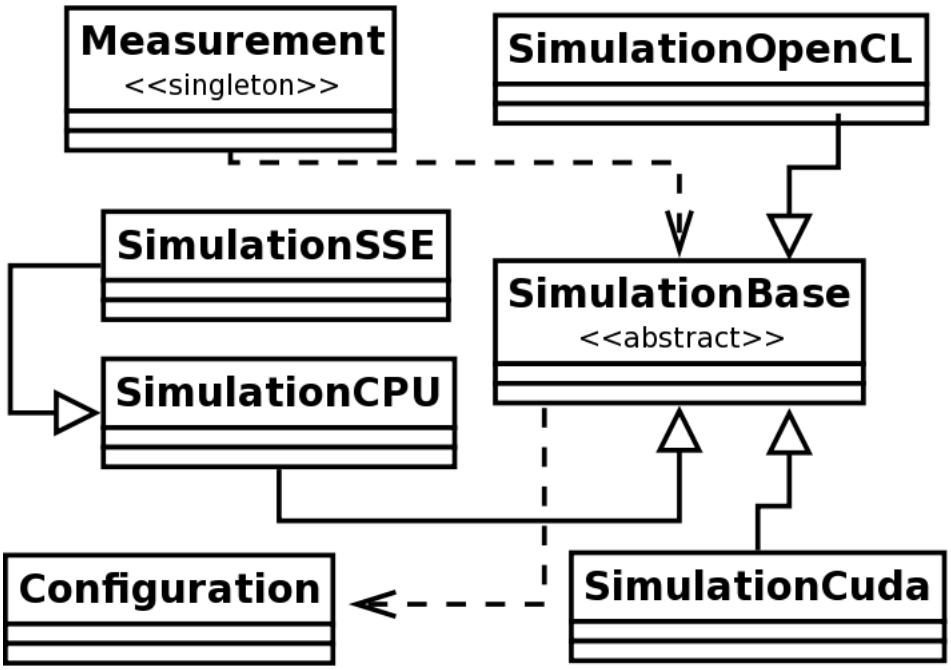
Třídy programu a vzájemné vztahy mezi nimi znázorňuje zjednodušený diagram tříd na obrázku 7.1. Zjednodušený je proto, že jsou z něj vypuštěny některé nepodstatné části stejně jako atributy a metody. Ty jsou popsány v následující kapitole při detailním popisu všech tříd.

2.2 Popis jednotlivých tříd

2.2.1 Třída Configuration

Třída Configuration slouží k přečtení konfigurace simulátoru ze souboru zadaného pomocí parametru -c a k následnému uložení potřebných načtených proměnných. Atributy této třídy jsou následující:

- *simuSteps*: počet simulačních kroků, které se budou programem provádět
- *roadLength*: délka cesty, tedy počet míst, která mohou být obsazena nebo neobsazena vozidly
- *density*: hustota vozidel na cestě, která přímo určuje, kolik vozidel bude při inicializaci simulace vytvořeno
- *vMax*: maximální rychlost vozidla
- *threadsPerBlock*: maximální počet CUDA vláken na blok v případě CUDA simulace, nebo velikost *work group* v OpenCL simulaci
- *slowingProbability*: pravděpodobnost, že auto zpomalí o jednu jednotku rychlosti



Obrázek 2.1: Zjednodušený diagram tříd

- *pinnedMemory*: použití připnuté paměti
- *randomCars*: určuje, zda se na začátku vygenerují auta náhodně podle zvolené hustoty dopravy, nebo deterministicky tak, že auto bude na každé třetí pozici na cestě, což je parametr užitečný pro ladění programu, aby bylo možné porovnávat jednotlivé implementace z hlediska správnosti provedení algoritmu
- *simuDevice*: určuje, jaká simulace se bude provádět, jestli to bude simulace na CPU, na CPU za použití SSE instrukcí, na GPU s využitím CUDA nebo na GPGPU s využitím OpenCL
- *dataMode*: datový mód

Tato třída obsahuje tyto metody:

- *readConfig*: Metoda pro načtení konfigurace ze souboru zadaného parametrem `-c soubor`, při chybě je vyvolána výjímka typu NSCHError
- *substrPara*: Slouží k ořezání jednotlivých parametrů po načtení jednotlivého řádku ze souboru s konfigurací simulátoru
- *printDebug*: Tiskne na obrazovku ladící výstupy neboli načtené parametry a jejich hodnoty

2.2.2 Třída SimulationBase

Základní třída, ze které dědí všechny třídy, které zprostředkovávají vlastní simulaci. Jedná se o abstraktní třídu, nelze ji tedy instanciovat. Obsahuje tyto atributy:

- *velocities*: Pole rychlostí jednotlivých vozidel
- *positions*: Pole pozic, na kterých se nachází vozidla
- *carsCount*: Počet vygenerovaných vozidel
- *config*: Konfigurace načtená třídou Configuration ze souboru
- *generatedPositions*: Proměnná pro dočasné uložení vyegenerovaných pozic vozidel

Třída SimulationBase má následující metody:

- *simulate*: Virtuální metoda, která musí být implementována v každé odvozené třídě, slouží k zavolání samotné simulace
- *printRoad*: Slouží k ladícímu výpisu cesty na standardní výstup
- *generateCarsPositionsRandom*: Vytváří vozidla náhodně. Pozice těchto vozidel poté se ukládá do pole *positions*
- *generateCarsPositions*: Vytváří vozidlo na každé třetí pozici na cestě a ukládá tyto pozice do *positions*.
- *generateCarsVelocitiesRandom*: Náhodně generuje rychlosti vozidel a výsledky ukládá do pole *velocities*
- *generateCarsVelocities*: Generuje rychlosti vozidel a výsledky ukládá do pole *velocities*

2.2.3 Třída SimulationCPU

Třída, která implementuje simulaci běžící na procesoru počítače bez dalších speciálních optimalizací. Nemá vlastní atributy krom těch, které dědí z třídy SimulationBase. Metody třídy jsou:

- *simulate*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace
- *congRandom*: Implementuje kongruetní generátor
- *calculateVelocities*: Funkce, ve které se provádí výpočty rychlostí při aktuálním kroku simulace
- *moveCars*: Implementuje pohyb aut o rychlost předem spočítané metodou calculate-Velocities

2.2.4 Třída SimulationSSE

- *indexes*: Uložení indexů od 0 až počet aut
- *nrOfLoops*: Počet iterací, které se budou provádět v závislosti na zadaném počtu automobilů
- *simulate*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace

- *moveCars*: Implementuje samotný pohyb aut
- *calculateVelocities*: Funkce, ve které se provádí výpočty rychlostí při aktuálním kroku simulace
- *moveCars*: Implementuje pohyb aut o rychlosti předem spočítané metodou calculateVelocities
- *setupCongRandom*: Inicalizuje náhodný generátor
- *congRandom*: vlastní generování náhodných čísel

2.2.5 Třída SimulationCuda

- *blocksCount*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace
- *d_velocities*: Pole rychlostí v paměti CUDA zařízení
- *d_positions*: Pole pozic v paměti CUDA zařízení
- *d_congStates*: Uložení stavů generátoru náhodných čísel v CUDA zařízení
- *simulate*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace

2.2.6 Globální funkce pro běh CUDA simulace

Tyto funkce jsou definované v souboru SimulationCuda.cuh a implementované v souboru SimulationCuda.cu

2.3 Kernely

Kernely:

- *kernelSetupCudaCongRandom*: Kernel pro inicializaci generátoru pseudonáhodných čísel
- *kernelCalculateVelocities*: Kernel pro pohyb vozidel
- *kernelMoveCars*: Kernel zajišťující pohyb aut.

2.4 Funkce volající kernely

- *calculateVelocities*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace.
- *moveCars*: Funkce k volání kernelu pro pohyb automobilu.
- *setupCongRand*: Funkce k volání kernelu pro inicializaci generátoru pseudonáhodných čísel
- *setDeviceConstants*: Funkce, která nastavuje konstanty používané v CUDA zařízení

2.5 Globální konstanty pro CUDA simulaci

- *d_roadLength*: Konstanta pro délku cesty
- *d_vMax*: Konstanta pro uložení maximální rychlosti
- *d_probability*: Kosntanta pro pravděpodobnost zpomalení auta o jednotku rychlosti
- *d_carsCount*: Konstanta pro uložení počtu aut
- *d_carsCountMinusOne*: Konstanta pro uložení počtu aut zmenšená o jedna
- *d_seed*: Pole počátečních hodnot generátoru pseudonáhodných čísel

2.5.1 Třída SimulationOpenCL

- *d_velocities*: Buffer rychlostí vozidel pro práci v OpenCL zařízení
- *d_positions*: Buffer pozic vozidel pro práci v OpenCL zařízení
- *d_constants*: Buffer konstant
- *d_seed*: Počáteční hodnota generátoru pseudonáhodných čísel
- *d_states*: Pro uložení stavů generátoru pseudonáhodných čísel
- *deviceUsed*: Index používaného zařízení
- *devices*: Vektor všech zařízení dané platformy
- *queue*: Fronta OpenCL příkazů
- *context*: Vlastní kontext
- *program*: OpenCL program
- *kernelCalculateVelocities*: Kernel pro výpočet rychlosti v daném kroku simulace
- *kernelMoveCars*: Kernel pro pohyb na základě vypočtených rychlostí jednotlivých vozidel
- *kernelSetupCongRand*: Kernel pro inicializaci generátoru pseudonáhodných čísel
- *simulate*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace
- *loadProgram*: Metoda, která načítá kód OpenCL programu
- *initKernel*: Inicializuje všechny proměnné potřebné pro běh programu

2.5.2 Třída NSCHError

Třída, která obstarává komentáře k výjímkám na základě chybových kódů uložených ve výčtovém typu errorCode. Všechna chybová hlášení v programu ve finále vyvolávají tuto výjímku, která je zachycována ve funkci `main`.

- *comment*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace
- *eCode*: Kód chyby určený k tisku příslušné chybové zprávy
- *what*: Virtuální metoda. Implementuje metodu zděděnou z třídy SimulationBase. Zajišťuje běh samotné simulace
- *oclErrorString*: Tiskne na standardní chybový výstup informaci o chybě při běhu OpenCL simulace

2.5.3 Třída Measurement

Třída Measurement je implementací návrhového vzoru *singleton*[1] neboli jedináčka.

- *startTime*: proměnná pro uložení počátečního stavu časovače
- *endTime*: konečný stav časovače
- *getInstance*: Složí k přístupu k jedináčkovi
- *startPAPICounters*: Spouští nízkoúrovňové čítače knihovny PAPI[2]
- *readAndResetPAPICounters*: Slouží k přečtení a resetování nízkoúrovňových čítačů knihovny PAPI
- *startPAPITimer*: Spouští časovač knihovny PAPI
- *stopPAPITimer*: Vypíná časovač knihovny PAPI

[1] http://msdn.microsoft.com/en-us/library/ms998426.aspx
[2] icl.cs.utk.edu/papi/

Kapitola 3

Implementace

3.1 Popis implementovaného algoritmu

Vlastní jádro implementace algoritmu je stejné pro všechny platformy. Skládá se ze tří částí. V první části je pro každé vozidlo vygenerováno jedno náhodné číslo v rozsahu 0 až 1 včetně, v druhé části se vypočítá rychlost pohybu pro každé vozidlo v závislosti na vozidlu, které se nachází na pozici před ním. V třetí části se všechna vozidla pohnou o v jednotek kupředu, kde v je aktualní rychlost vozidla.

- Na počátku je pole náhodných čísel inicializováno náhodnými čísly tak, že každý automobil dostane jedno náhodné číslo. Na začátku každého dalšího kroku jsou pro všechna vozidla vygenerována čísla nová.
- Výpočet rychlosti automobilu je o něco složitější. Nejprve se vypočítá index auta, které se nachází před aktuálně zpracovávaným. Pokud se jedná o poslední automobil v poli, které představuje cestu, se kterou pracujeme, pokračujeme znovu od začátku cesty, tedy indexem dalšího auta bude první index v poli 0. Poté se vypočítá vzdálenost k tomuto dalšímu vozidlu jako rozdíl aktuální pozice a pozice tohoto předcházejícího vozidla. Pokud došlo k přetečení cesty, je nutné ještě k této vzdálenosti přičíst délku cesty.

  Dále se provádí tři první kroky Nagel-Schreckenbergerova simulačního modelu. Prvně se provede zrychlování a to tak, že pokud je rychlost aktuálního auta menší, než je jeho možná maximální rychlost a vzdálenost k autu před ním je větší než jeho aktuální rychlost zvýšená o jedna, zvýší se jeho rychlost o jednu jednotku. Druhým krokem je zpomalení. Pokud je vzdálenost k dalšímu autu snížená o jedna, větší, než rychlost aktuálního vozidla a vzdálenost k autu před ním je menší nebo rovna maximální rychlosti, sníží se jeho rychlost na tuto vzdálenost sníženou o jedna. Třetím krokem je randomizace, který platí jen pro vozidla, která mají rychlost větší než nulovou, tedy nestojí. Zde se porovná náhodně generované číslo, které bylo vygenerováno na začátku simulačního kroku, s pravděpodobností zpomalení. Pokud je toto číslo menší než pravděpodobnost zpomalení, auto zpomalí svojí rychlost o jedna.
- Pohyb aut se provede pro každý automobil, kdy se k jeho pozici přičte jeho rychlost vypočítaná v předchozí části. Pokud je vypočítaná pozice vozidla větší nebo rovna delce cesty, odečte se od ní délka cesty, tudíž dojde k tomu, že automobil je posunut na začátek pole představujícího cestu.

Algoritmus 1: FASTSLAM

Input: (velocities, positions, randomNumbers, vMax, slowingProbability, roadLength)
Output: velocities, positions

```
1  initRandom(seed);
2  for i to simulations do
3      for carIndex to carsCount do
4          randomNumbers_carIndex = congRandom(randomNumbers_carIndex)
5      end for
6      for carIndex to carsCount do
7          if carIndex < carsCount − 1 then
8              nextCarIndex = carIndex + 1
9          else
10             nextCarIndex = 0
11         end if
12         distanceToNextCar = velocities_nextCarIndex − velocities_carIndex
13         if positions_nextCarIndex < positions_carIndex then
14             distanceToNextCar = distanceToNextCar + roadLength
15         end if
16         if
           velocities_carIndex < vMax and distanceToNextCar > velocities_carIndex + 1
           then
17             velocities_carIndex = velocities_carIndex + 1
18         end if
19         if velocities_carIndex > distanceToNextCar − 1 and distanceToNextCar <=
           vMax then
20             velocities_carIndex = distanceToNextCar − 1
21         end if
22         if velocities_carIndex > 0 and random_carIndex <= slowingProbability then
23             velocities_carIndex = distanceToNextCar − 1
24         end if
25     end for
26     for carIndex to carsCount do
27         positions_carIndex = positions_carIndex + velocities_carIndex
28         if positions_carIndex >= roadLength then
29             positions_carIndex = positions_carIndex − 1
30         end if
31     end for
32 end for
```

3.2 Složitost algoritmu

Složitost implementovaného algoritmu je O(3n). Náročnost výpočtu závisí pouze na počtu vozidel na cestě. S každým dalším bodem se lineárně zvýší náročnost algoritmu. Důvodem je použití tří for cyklů, jednoho pro výpočet rychlosti vozidla pro aktuální krok simulace, jenž musí být vypočítán pro všechna vozidla před zahájením jejich pohybu a druhého pro samotný pohyb vozidel. Třetí for cyklus se používá pro generování pseudonáhodných čísel.

3.3 Automatické testování

Automatické testování probíhalo pomocí automatického skriptu napsaného v jazyce Python verze 3[1]. Tak bylo možné určit validitu výsledků jednotlivých implementací na různých zařízeních při zadání stejných vstupních parametrů. Za těchto podmínek bylo nutné vypnout randomizaci, aby bylo možné výsledky úspěšně validovat.

3.4 Generátor pseudonáhodných čísel

V oblasti simulací na počítači je velmi důležitá problematika generování pseudonáhodných čísel, pseudonáhodná se nazývaj kvůli skutečnosti, že nejde o čistě náhodná čísla, nepůsobí v nich žádné náhodné veličiny, jelikož je počítač diskrétní a deterministikcý systém. Je tedy možné vygenerovat stejným algoritmem stejné číslo.

Pro potřeby simulace je použit jednoduchý generátor pseudoáhodných čísel nazvaný lineárně kongruentní generátor. Tento generátor jsem použil jednak proto, že jeho implementace je jednoduchá, ale zároveň je rychlý, paměťově nenáročný a dobře implementovatelný pro GPU.

Kongruetní generátor pseudonáhodných čísel obecně generuje čísla takto[12]:

$$x_{i+1} = (ax_i + b) mod(m)$$

kde a, b, m jsou vhodně zvolené konstanty, mod je operace zbytek po celočíselném dělení.

Tento generátorgeneruje celá čísla s rovnoměrným rozložením $0 <= x_i < m$. Chcemeli převést výsledné číslo na pro naše potřeby požadovaný rozsah $< 0, 1)$, je nutné dělit výsledek modulem m.

Je nezbytností před generováním samotného pseudonáhodného čísla inicializovat x_0 na určitou zvolenou hodnotu. Každé další použití generátoru generuje další číslo. Z důvodu omezení počtu čísel, které patří do intervalu $0 <= x_i < m$, začne se po určitém intervalu generovaných čísel jejich posloupnost opakovat. Říkáme, že generátor má takzvanou periodu. Perioda generátoru může způsobit problémy u rozsáhlých simulačních experimentů, je proto třeba vhodně zvolit parametry generátoru, aby byla tato perioda co největší. Pro volbu konstant generátoru je pro vykazování potřebných statistických vlastností nutné dodržovat následující pravidla[2]:

1. m by mělo mít hodnotu 2^n, kde n je počet bitů čísel typu unsigned integer, aby nebylo nutné provádění operace modulo, generování se, díky tomu, že je operace modulo poměrně výpočetně náročná, urychlí.

2. b a m jsou nesoudělná čísla

[1] www.python.org/download/releases/3.0

3. a - 1 je dělitelné všemi prvočíselnými faktory m

4. a - 1 je násobkem 4, jestliže m je násobek 4

Pro běh na 64-bitovém systému, který se nachází na serveru pcjaroš byly vhodně zvolené konstanty následující:

1. a = 2862933555777941757

2. b = 3037000493

3. implicitní modulo dané datovým typem unsigned long

Takto implementovaný generátor má periodu $p = 2^{64}$.

Jako alternativu k tomuto generátoru jsem zkoumal i použití známého generátoru pseudonáhodných čísel Mersenne Twister a jeho implementaci MT19937. Ve výsledku jsem ho ale nepoužil, existuje sice již existující knihovna, která tento algoritmus implementuje na CPU, CPU s SSE instrukcemi v implementaci SFMT[2] , ale použití MTGP implementace pro GPU, kterou je možné použít jak pro CUDA tak pro OpenCL je těžkopádné. [3] Ale jelikož jsem potřeboval porovnávat stejné implementace na různých zařízeních, zvolil jsem raději jednoduchý generátor, jehož perioda postačuje, jehož implementaci jsem optimalizoval pro běh na všech použitých platformách.

3.5 Měření výkonnosti algoritmu

Pro měření efektivity jednotlivých simulací na různých zařízeních bylo použito knihovny PAPI[4], která umožňuje snadnou a hlavně multiplatformní práci s hardwarovými časovači, která jsou na daném počítači k dispozici. Implicitně používá časovač s největším rozlišením, tedy s nejmenší velikostí časových jednotek. Na školním serveru pcjaros-gpu.fit.vutbr.cz se jedná o jednotky mikrosekund.

Knihovna PAPI umožňuje vysokoúrovňové a nízkoúrovňové profilování aplikací.

3.6 Hardwarové specifikace systému

3.6.1 Hardwarové specifikace použitého CPU

Program běžící na CPU byl testován na školním serveru pcjaros-gpu.fit.vutbr.cz, přehled důležitých parametrů procesoru, který je zde osazen, se nachází v tabulce [6] [4]8.1.

Každé jádro má svojí vlastní L1 a svojí vlastní L2 keš. L1 keš je rozdělena na dvě poloviny - 32KB pro instrukce a 32KB pro data. L3 keš je sdílená mezi všemi čtyřmi jádry procesoru. Pro dobu přístupu k jednotlivým keším platí, že nejrychlejší přístup je k L1 keši, poté k L2 keši a nejpomalejší je přístup k L3 keši.

[2] http://www.math.sci.hiroshima-u.ac.jp/ %20m-mat/MT/SFMT/index.html

[3] http://www.math.sci.hiroshima-u.ac.jp/ %20m-mat/MT/MTGP/index.html#MTGP.

[4] icl.cs.utk.edu/papi/

CPU	Intel© Core^{TM} i7 CPU 920
Frekvence	2,66GHz
Frekvence s TurboBoost	2,93GHz
Počet jader	4
Počet vláken	8
Podpora SSE verze	4,2
Výrobní proces	45nm
Maximální propustnost paměti	25,6GB/s
Velikost L1 cache	64KB
Velikost L2 cache	256KB
Velikost L3 cache	8MB

Tabulka 3.1: Specifikace CPU

3.6.2 Hardwarové specifikace použitého GPU

Z hlediska optimalizace algoritmu pro výpočet na GPU je důležité znát parametry použité grafické karty pro kterou se tato optimalizace provádí. Grafické karty NVIDIA jsou rozdělené podle tzv. compute capability. Všechny grafické karty stejné compute capability mají stejný počet procesorů na stream multirpocesor, tedy stejný počet maximálních vláken na blok, maximální velikostí sdílené paměti, počty registrů na vlákno atp. Čím se grafické karty se stejnou compute capability odlišují, je takt počet a množství globální paměti, případně šířkou paměťové sběrnice. Compute capability udává schopnost využívat možnosti platformy CUDA.

Program běžící na zařízení CUDA byl rovněž testován na serveru pcjaros-gpu.fiit.vutbr.cz. Nachází se zde grafická karta NVIDIA GeForce 580 GTX společnosti ASUS s velikostí paměti 1536MB architektury Fermi s compute capability 2.0. Důležité parametry karty jsou uvedeny v tabulce[5][1]8.2.

GPU	NVIDIA GeForce 580 GTX
Frekvence jádra	772MHz
Frekvence procesoru	1544MHz
Počet stream multiprocesorů	16
Počet CUDA jader na stream multiprocesor	32
Velikost globální paměti	1536MB
Propustnost paměti	192,4GB/s
Velikost L1 cache/velikost sdílené paměti na blok vláken	48KB/16KB nebo 16KB/48KB
Velikost L2 cache	768KB
Velikost souboru registrů	32768

Tabulka 3.2: Specifikace GPU

Kapitola 4

Porovnání naměřených výsledků

V této kapitole budou rozebrány a porovnány naměřené výsledky implementací s různými vstupními parametry. Počet simulačních kroků byl ve všech případech roven počtu 100. Stejný fakt platí pro maximální rychlost vozidel. Zpočátku byl ve všech simulacích vygenerován počet aut náhodně podle zadané hustoty.

4.1 Vstupní konfigurace s malou délkou cesty

V této sekci se budeme zabývat tím, jaký má vliv malá délka cesty a s tím související počet vozidel na dobu běhu simulačního algoritmu.

Délka cesty	100
Hustota vozidel	0.5
Vláken na blok	512
Pravděpodobnost zpomalení	0

Naměřené hodnoty poté byly následující:

Velikost skupiny vláken	32	64	128	256	512
Čas CUDA [μs]	604	642	939	606	603
Čas OpenCL [μs]	3669	3679	3725	2561	2553

Čas CPU 88[μs], Čas CPU s SSE 78[μs]

Zde se projevily slabiny grafických karet, nemá smysl na nich provádět výpočty, které pracují s malým množstvím dat, právě z toho důvodu, že každé jedno jádro v CUDA zařízení je mnohem pomalejší než jádro v běžném CPU.

4.2 Vliv velikosti skupin vláken na dobu výpočtu

Naměřené hodnoty poté byly následující:

Doba běhu algoritmu na procesoru byla pro porovnání 756071[μs] a na procesoru využitím SSE instrukcí 859217[μs].

Délka cesty	1000000
Hustota vozidel	0.5
Pravděpodobnost zpomalení	0.5

Počet vláken na blok	64	128	256	512	768	1024
Čas CUDA [μs]	15977	11324	10491	10560	10997	13368
Čas OpenCL [μs]	17420	12056	11112	11100	11583	13993

Z tabulky je možné vyčíst, že nejefektivněji pracují karty NVIDIA s bloky velikosti kolem 512 vláken. Při tomto počtu bloků se podařilo simulaci oproti CPU zrychlit v případě CUDA implementace téměř 72-krát a v případě OpenCL 68-krát.

4.3 Vliv pravděpodobnosti zpomalení vozidla na dobu výpočtu

Délka cesty	1000000
Hustota vozidel	0.5
Počet vláken na blok	512

Pravděpodobnost, že některé z aut zpomalí nemá na simulaci žádný významný vliv.

4.4 Vliv hustoty dopravy na dobu výpočtu

Hustota dopravy rovněž nezpůsobuje výraznější výkyvy v dobách trvání výpočtů.

Pravděpodobnost zpomalení	0	0.1	0.3	0.5	0.7	0.9
Čas CUDA [μ s]	10579	11146	10567	10564	10554	10537
Čas OpenCL [μ s]	10587	11060	11092	11093	11146	11090
Čas CPU [μ s]	1233537	1257190	1233537	1252775	1253978	1239454
Čas CPU s SSE [μ s]	876233	683929	1117616	1123920	1111909	1126084

Tabulka 4.1: Tabulka s naměřenými hodnotami na základě pravděpodobnosti zpomalení

Délka cesty	1000000
Počet vláken na blok	0.5
Pravděpodobnost zpomalení	0.5

Tabulka 4.2: Neměnné parametry v simulaci, kde jsme sledovali vliv pravděpodobnosti zpomalení auta

Hustota vozidel	0.1	0.3	0.5	0.7	0.9
Čas CPU s SSE [μ s]	1122541	1127129	1114939	1124524	
Čas CUDA [μ s]	10535	10568	10532	10592	10592
Čas OpenCL [μ s]	11080	11110	11079	11095	11103
Čas CPU [μ s]	758740	764523	764523	1261514	1249810

Tabulka 4.3: Tabulka s naměřenými hodnotami na základě hustoty dopravy

Kapitola 5

Závěr

Cílem práce bylo seznámení se s celulárními automaty a prostředky pro jejich akceleraci na GPGPU. Tohoto cíle se podařilo úspěšně dosáhnout. V některých případech se podařilo algoritmus urychlit až 72-krát v případě implementace na programovací platformě CUDA a 68-krát v případě implementace pomocí OpenCL. Lze tedy považovat více než za vhodné danou akceleraci implementovat tam, kde se pracuje s velkým počtem vozidel, protože jak se projevilo v sekci s malou délkou cesty č. 9.1, GPGPU se v žádném případě nehodí k řešení malých problémů, kde nemůžou profitovat ze své škálovatelné vysoce paralelní SIMT architektury.

Práce mi dala představu a zkušenosti s implementací algoritmů na GPU, což jsem od ní i očekával. Dozvěděl jsem se, že optimalizace pro GPU je oproti klasickému programování velice náročná činnost pro programátora.

V budoucnu by bylo vhodné vylepšit algoritmus pro simulaci na CPU s použitím SSE instrukcí, nejlépe jeho úplným předěláním, abychom se zbavili instrukcí větvení, které se těžko implementují pomocí SIMD instrukcí. Dosavadní algoritmus nevyužívá naplno potenciál vektorizace a paralelního zpracování na CPU. Zajímavým rozšířením práce by mohlo být napojení na grafické API OpenGL nebo úprava implementace pro běh na více GPU současně případně další optimalizace pomocí sdílené paměti, kterou jsem však také testoval, nicméně tento problém nedokázala efektivně urychlit.

Literatura

[1] GeForce GTX 580 Specifications.
URL <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications>

[2] *The Art Of Computer Programming, Volume 2: Seminumerical Algorithms, 3/E.* Pearson Education, 1998, ISBN 9788177583359.
URL <http://books.google.cz/books?id=OtLNKNVh1XoC>

[3] *Intel 64 and IA-32 architectures software developer's manual.* Denver: Intel Corporation, 2006.
URL <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf>

[4] *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem).* Intel Corporation, 2008.
URL <http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf>

[5] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi.* NVIDIA Corporation, 2009.
URL <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>

[6] *Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series.* Intel Corporation, první vydání, 2010.

[7] *CUDA C Best Practices Guide.* NVIDIA Corporation, 2014.
URL <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>

[8] *NVIDIA CUDA C Programming Guide.* 2014.
URL <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

[9] *OpenCL programming guide.* Upper Saddle River: Addison-Wesley, c2012.

[10] Gaster, B.; Howes, L.: *The OpenCL C++ Wrapper API.* Khronos OpenCL Working Group.
URL <http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.2.pdf>

[11] Nagel, K.; Schreckenberg, M.: A cellular automaton model for freeway traffic. *Journal de Physique I*, ročník vol. 2, č. issue 12, 1992: s. 2221–2229.
URL <http://www.edpsciences.org/10.1051/jp1:1992277>

[12] Peringer, P.: *modelování a simulace IMS*. 2006.

[13] Sarkar, P.: A brief history of cellular automata. *ACM computing Surveys*, ročník 32, č. 1, 3 2000.

[14] Wolfram, S.: *Cellular Automata and Complexity: Collected Papers*. 1-2150-A; Louisiana Barrier Island, Addison-Wesley Publishing Company, 1994, ISBN 9780201626643.

Příloha A

Obsah CD

1. Zdrojové soubory simulace ve složce src
2. Konfigurační soubory ve složce tests
3. Makefile
4. Readme.txt soubor s popisem spuštění aplikace
5. Skript start.h pro automatické načtení více konfiguračních souborů
6. Tento text v souboru thesis.pdf