Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Využití GPU pro paralelní simulační výpočty

Plzeň, 2019 Bc. Daniel Rajf

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 2.5.2019 Daniel Rajf,

Abstract

Utilization of GPU for Parallel Simulation Computations

The purpose of this thesis is to implement road traffic simulation that uses a GPU for computations and compare simulation run time to a reference implementation that uses a CPU for computations.

The created application allows generating a road network for simulation using model with cellular automaton or car following model. Also, it can compute individual steps of the simulation using a GPU or a CPU. Simulation state can be monitored using simple visualization.

The application is written in C# programming language and uses .NET Framework. The application is using OpenCL technology and NOpenCL library to run computations on a GPU.

Abstrakt

Tato práce se zabývá implementací simulace silniční dopravy, která pro výpočty využívá jednotku GPU, a porovnáním doby běhu výpočtů simulace s referenční implementací, která pro výpočty využívá jednotku CPU.

Vytvořená aplikace umožňuje vygenerovat silniční síť pro simulaci s využitím modelu s celulárním automatem nebo car following modelu a následně počítat jednotlivé kroky simulace s pomocí jednotky GPU nebo CPU. Stav simulace je možné sledovat pomocí jednoduché vizualizace.

Aplikace je napsaná v jazyce C# a využívá technologii .NET Framework. Pro spouštění výpočtu na jednotce GPU se využívá technologie OpenCL a knihovna NOpenCL.

Obsah

1 Úvod

Simulace se obecně využívají v mnoha odvětvích lidské činnosti. Jeden z možných druhů simulací je simulace silniční dopravy, která může být využita například k analýze konkrétní silniční sítě nebo případně může i pomoci v návrhu nové silniční sítě.

Protože silniční síť může nabývat velkých rozměrů (klidně i desetitisíce křižovatek), aby zahrnula celou požadovanou oblast, simulace takové rozsáhlé sítě může být časově náročná. Cílem této práce je tedy implementovat simulaci silniční dopravy, která bude počítána pomocí jednotky Graphics Processing Unit (GPU), a vyzkoušet, zda je pro tento typ výpočtů jednotka GPU vhodná, dále pak ideálně stejný algoritmus implementovat i pro výpočet pomocí jednotky Central Processing Unit (CPU), výsledky těchto dvou implementací navzájem porovnat a určit, zda je na řešení tohoto problému vhodnější využít jednotku GPU nebo CPU.

Práce tedy bude porovnávat doby běhu implementace pro GPU a pro CPU u dvou různých simulačních modelů, konkrétně půjde o model s celulárním automatem a model následování vozidel (car following model), které se navzájem liší hlavně v reprezentaci jízdních pruhů silniční sítě. Oba tyto modely se běžně používají. Je tedy vhodné zjistit, zda jsou oba vhodné pro výpočet na GPU.

2 Popis jednotek GPU

Jednotka Graphics Processing Unit (GPU) je procesor, který je specializovaný na grafické výpočty a na jiné výpočty, které je možné provádět paralelně. Moderní jednotky GPU obsahují velké množství tzv. unifikovaných shaderů. Jedná se nejčastěji o procesory, které umožňují provádět téměř jakékoliv výpočty paralelně. Každý shader tedy zpracovává jedno vlákno. Využívá se zde nejčastěji model SIMT (Single Instruction, Multiple Threads), kdy jedna instrukce je prováděna v několika vláknech současně [1].

2.1 Historie jednotek GPU

Dřívější grafické karty unifikované shadery neobsahovaly, ale obsahovaly dva druhy specializovaných shaderů, které byly zaměřené na konkrétní výpočty [2]. Jednalo se o vertex shader, který měl za úkol provádět transformace pozice jednotlivých vrcholů z 3D prostoru do 2D roviny, která se pak následně vykreslí na obrazovku. Druhý z nich se nazývá pixel shader (někdy také fragment shader). Ten se staral o výpočet barvy a případně i dalších atributů, které jsou spojeny s konkrétními pixely na obrazovce, aby bylo možné sestavit finální obraz z jednotlivých polygonů. Ve fázi pixel shaderu lze také aplikovat například nasvícení scény, stíny nebo průhlednost. Oba typy shaderů se mezi sebou liší hlavně v jejich instrukční sadě. Vertex shadery byly optimalizovány hlavně na maticové výpočty, zatímco pixel shadery navíc umožňovaly například číst hodnoty z textury. Toto omezení umožňovalo snížit výrobní náklady, ale způsobovalo hlavně nerovnoměrné rozložení zátěže. Pokud například aplikace vykreslovala polygony o vysokém počtu vrcholů, ale na nízkém výstupním rozlišení, počet vertex shaderů nemusel být dostatečný, zatímco počet pixel shaderů mohl být více než dostatečný.

Unifikované shadery toto omezení odstranily, protože nejsou specializované na určité operace a je možné je použít univerzálně. Začaly se používat v roce 2006 [3]. Instrukční sada unifikovaných shaderů je i nadále rozšiřována o další instrukce. Díky tomuto přechodu na unifikované shadery je dále možné provádět na grafické kartě nejrůznější výpočty, které již nemusí souviset s vykreslováním grafické scény.

2.2 Porovnání vlastností GPU a CPU

Výpočty pomocí GPU mají ve srovnání s Central Processing Unit (CPU) několik výhod i nevýhod. Některé druhy výpočtů je vhodnější provádět raději na CPU, jiné na GPU. Jednotky GPU mají zpravidla více jader, než jednotky CPU. To umožňuje paralelně zpracovávat větší objem dat. Naopak sekvenční výpočet je na jednotce GPU velice neefektivní. Jednotky GPU mají dále zpravidla vyšší výkon v aritmetických výpočtech, zejména v plovoucí desetinné čárce, a také v případech, kde není potřeba častý přístup do paměti a většina výpočtů se vejde do registrů. Je to z toho důvodu, že jednotky GPU zpravidla obsahují velké množství registrů. Nevýhoda GPU je pomalý přístup do paměti DRAM, proto (stejně jako CPU) obsahuje i různé druhy mezipamětí pro každé jádro.

Pro zajištění paralelního přístupu k paměti je paměť rozdělena na tzv. banky. Aby nebylo nutné uzamykat celou paměť, je uzamknuta pouze ta banka, ke které vlákno právě přistupuje. Jednotlivé banky jsou prokládány, proto je nejefektivnější sekvenční přístup, kde každé vlákno přistupuje pouze ke své části paměti. Naopak náhodný přístup do paměti výkon znatelně snižuje [4].

Protože všechna vlákna vykonávají stejnou instrukci, časté využívání podmínek a větvení snižuje celkový výkon. Pokud některá vlákna větví neprochází, musí i tak zpravidla čekat na ostatní vlákna, než danou větev zpracují a dorazí tedy zase ke stejné instrukci jako ostatní. Proto také jednotky GPU neobsahují predikci větvení a spekulativní provádění [5].

3 Existující prostředky pro paralelní výpočty s využitím GPU

Pro spuštění nějakého výpočtu na GPU je třeba využít pro to určených technologií. Některé z nich jsou univerzální, dostupné pro širší skupinu jednotek GPU a některé jsou naopak dostupné pouze na jednotkách GPU konkrétních výrobců. Mezi nejrozšířenější řešení patří Nvidia CUDA, OpenCL, C++ AMP a SYCL.

3.1 Nvidia CUDA

Technologie CUDA je vytvořená společností Nvidia pro vykonávání paralelních výpočtů na grafických kartách společnosti Nvidia [6]. Na jednotkách GPU jiných společností nelze tuto technologii provozovat. Výpočty lze pak spouštět na operačních systémech Microsoft Windows, Linux a macOS [7].

```
#include <math.h>
#include "cuda_runtime.h"
#include "kernel.h"

__global__ void kernel_sum(
    const float* a, const float* b, float* result, int length)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n_el) result[i] = a[i] + b[i];
}

void compute_sum(
    const float* a, const float* b, float* result, int length)
{
    int threadsPerBlock = 512;
    int blocksPerGrid = ceil(double(length)/double(threadsPerBlock));

    kernel_sum<<<blocksPerGrid,threadsPerBlock>>>(
        a, b, result, length);
}
```

Obrázek 1: Ukázka programu v C++ pro Nvidia CUDA

CUDA je jedna z nejstarších technologií, které jsou určeny pro tyto účely. První verze byla vydána v roce 2007. Programy pro Nvidia CUDA je možné psát v jazycích C, C++ a Fortran a následně je přeložit pomocí překladače z CUDA SDK, které je nutné samostatně nainstalovat. Ukázka zdrojového kódu jednoduchého programu v C++, který provede součet prvků dvou polí, je na Obrázku 1.

Tato technologie má stálé aktivní podporu. Poslední verze SDK (verze 10.1) byla vydána 28. února 2019 [7].

3.2 OpenCL

Technologie OpenCL je nejrozšířenější v oblasti paralelních výpočtů. Je podporována všemi hlavními společnostmi [8], které vyvíjí jednotky GPU, tzn. AMD [9], Intel [10], Nvidia [11] a další. Navíc je podporována i celou řadou operačních systémů včetně Microsoft Windows, Linux a macOS. Proto jsou aplikace využívající OpenCL velmi dobře přenositelné. První verze byla vydána v roce 2009.

Obsahuje dvě části. První z nich je API, které aplikace používají pro spouštění programů na cílovém zařízení – jednotce GPU nebo CPU. Jedná se o nízkoúrovňové API, které lze využít například i v jazyce C. Existují však i různé knihovny, jež umožňují tuto technologii využívat i z mnoha jiných programovacích jazyků.

Druhou částí je programovací jazyk OpenCL C určený právě pro programování paralelních výpočtů pro zařízení OpenCL. Tento jazyk je založen na jazyce C (konkrétně specifikace C99). Obsahuje však nová klíčová slova pro potřeby OpenCL. Dále obsahuje funkce, které je možné použít pro samotné paralelní výpočty. Navíc zavádí několik omezení, jež se ve specifikaci C99 nevyskytují. Například nelze provádět rekurzivní volání funkce, vytvářet pole proměnné délky, deklarovat ukazatele na funkce nebo zapisovat do polí číselných typů menších než 32 bitů [12]. Navíc platí to, že data nemohou obsahovat ukazatele na jiná data, protože všechna data mají v paměti GPU jinou adresu než původní data v paměti RAM. Tento nedostatek byl odstraněn ve verzi 2.0, kdy byla přidána funkcionalita Shared Virtual Memory (SVM), která umožňuje sdílení adresního prostoru mezi hostitelem a OpenCL zařízením [13].

Některé nedostatky řeší různá rozšíření technologie OpenCL, která jsou však často podporována pouze některými výrobci, případně některými jednotkami GPU. Dále se uvádí, že technologie je sice poměrně snadno přenositelná, ale daný program může mít různé výkonnostní charakteristiky na různých zařízeních OpenCL [12]. Ukázka zdrojového kódu jednoduchého programu, který provede součet prvků dvou polí, je na Obrázku 2.

Ve verzi 2.2 byla technologie OpenCL rozšířena o jazyk OpenCL C++ kernel language, ve kterém je možné psát paralelní výpočty s využitím některých možností jazyka C++14 [8].

```
void kernel compute_add(
    global int* a, global int* b, global int* result, int length)
{
    int i = get_global_id(0);
    if (i < length) result[i] = a[i] + b[i];
}
```

Obrázek 2: Ukázka programu v OpenCL C

Tato technologie je stále udržována. Poslední verze (verze 2.2) byla vydána v roce 2017 [14].

3.3 C++ AMP

Další je technologie C++ AMP [15]. Tato technologie umožňuje psát výpočty využívající GPU přímo v jazyce C++ spolu se standardním kódem, který je prováděn na CPU. To velmi zkracuje kód, který je nutné napsat. Dále je možné výpočty zapisovat ve formě lambda výrazů. Ukázka zdrojového kódu jednoduchého programu, který provede součet prvků dvou polí, je na Obrázku 3.

C++ AMP je vytvořen společností Microsoft. Využívá pro výpočty technologii DirectX 11, proto potřebuje pro běh operační systém Microsoft Windows 7 nebo novější. Pro překlad využívající technologii C++ AMP je také nutné mít speciální překladač jazyka C++ s podporou této technologie. Tím je pouze překladač Visual C++ od společnosti Microsoft, proto lze výsledný program spustit pouze na operačních systémech Microsoft Windows. Technologie dále umožňuje provést výpočet na CPU, například pokud v počítači není dostupná vhodná jednotka GPU.

Technologie však není příliš rozšířená, a to hlavně kvůli uzavřenosti a podpoře pouze operačního systému Microsoft Windows. Navíc se zdá být i poměrně neudržovaná, protože na oficiálních webových stránkách ještě není zmíněna podpora operačního systému Microsoft Windows 10 [15].

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

void compute_sum(
    int* a, int* b, int* result, int length)
{
    array_view<const int, 1> a_(length, a);
    array_view<const int, 1> b_(length, b);
    array_view<int, 1> result_(length, result);
    result_.discard_data();

    parallel_for_each(
        result_.extent,
        [=](index<1> idx) restrict(amp) {
            result_[idx] = a_[idx] + b_[idx];
        }
    );

    result_.synchronize();
}
```

Obrázek 3: Ukázka programu v C++ s použitím C++ AMP

3.4 SYCL

SYCL je technologie, která umožňuje psát paralelní výpočty pomocí vysokoúrovňových konstrukcí jazyka C++ [16] podobně jako technologie C++ AMP. Na rozdíl od technologie C++ AMP ale nevyžaduje žádná rozšíření jazyka C++ ani speciální překladač. Využívá pouze konstrukce jazyka C++11. Pro samotnou interakci s GPU (případně CPU) pak využívá nízkoúrovňovou technologii OpenCL. Ukázka zdrojového kódu jednoduchého programu, který provede součet prvků dvou polí, je na Obrázku 4.

Mezi výhody patří typová bezpečnost a zjednodušení zápisu výpočtů. Využívá šablony a lambda výrazy jazyka C++. Je však stále umožněn přístup k nízkoúrovňovým funkcím technologie OpenCL.

Poslední verze (verze 1.2.1) byla vydána 13. února 2019 [16]. Předchozí verze (verze 1.2) byla vydána před 4 lety.

```
#include <sycl.hpp>
using namespace cl::sycl

void compute_sum(
    std::vector& a, std::vector& b, std::vector& result, int length)
{
    buffer a_b(h_a); buffer b_b(h_b); buffer result_b(h_d);
    queue my_queue;

    command_group(my_queue, [&]() {
        auto a_ = a_b.get_access<access::read>();
        auto b_ = b_b.get_access<access::read>();
        auto result_ = result_b.get_access<access::write>();

        parallel_for(length, kernel_functor([=](id<> item) {
            int i = item.get_global(0);
            result_[i] = a_[i] + b_[i];
        }));
    });
}
```

Obrázek 4: Ukázka programu v C++ s použitím SYCL

4 Úvod do simulací

Simulace se zpravidla snaží napodobit nějaký reálný, případně abstraktní systém pomocí počítačových výpočtů. Měl by obsahovat všechny klíčové vlastnosti systému, aby mohl napodobit chování tohoto systému co nejvěrohodněji. Využívá se nejčastěji pro zkoumání vlastností tohoto systému [17].

Často je možné vytvořit simulaci pouze části původního systému, nebo je případně nutné některé jeho vlastnosti zjednodušit, protože jinak by mohla být simulace příliš náročná na výpočet.

4.1 Dělení simulací podle typu

Analytické simulace jsou zpravidla vytvořeny pro získávání nějakých statistických dat systému, který modeluje. V některých případech je nutné simulace spustit vícekrát, aby mohl být získán dostatečný vzorek dat pro vyvození závěru, proto je vhodné, aby simulace byla co nejrychlejší [18]. Je žádoucí, aby čas výpočtu simulace byl kratší než čas, který v simulaci uplynul během této doby. V některých případech simulace obsahuje i vizualizaci, aby bylo možné sledovat její průběh.

Druhý typ simulace je simulace virtuálního prostředí, která se liší tím, že simuluje nějaké prostředí v reálném čase a umožňuje uživateli simulaci ovládat a ovlivňovat její běh [18]. Hlavním cílem je poskytnout uživateli dojem, že je součástí simulovaného systému. Tento druh simulace je často využíván například v některých počítačových hrách, ale také např. pro výcvik.

4.2 Dělení simulací podle simulačního času

Simulace mohou být dále rozděleny na diskrétní a spojité podle reprezentace času v simulaci. Ve spojitých simulacích se stav systému průběžně mění. Chování je často popsáno diferenciálními rovnicemi [18]. V diskrétních simulacích se stav mění pouze v určených bodech simulačního času. Diskrétní simulace se mohou dále dělit na časově krokované (time-stepped) a řízené událostmi (event-driven).

V časově krokovaných simulacích je čas rozdělen na stejně dlouhé časové kroky. Simulace postupně přechází z jednoho kroku na druhý a při tomto přechodu se mění stav simulace. Všechny akce, které se stanou během jednoho časového kroku, jsou obvykle považovány za současné, proto při velmi dlouhých časových krocích může simulace ztrácet přesnost. Z těchto důvodů je nutné zvolit rozumnou délku časového kroku. V každém kroku je přepočítaný celý stav simulace.

Event-driven simulace jsou řízené událostmi. Stav simulace se nemění v určitých časových krocích, ale pouze v případě, že obdrží nějakou událost [18]. Každá událost zpravidla nese časovou značku, která určuje, v jakém čase událost nastala, a akci, která popisuje inkrementální změnu simulačního stavu. Čas v simulaci tedy přechází mezi časovými značkami jednotlivých událostí namísto časových kroků. Tyto události mohou například být odlet a přílet letadla v případě simulování letiště [18].

5 Simulace silniční dopravy

Tato práce se bude dále zaměřovat na simulování silniční dopravy. Jedná se o takovou simulaci, která modeluje pohyb vozidel po silniční síti.

5.1 Úroveň detailů simulace silniční dopravy

Simulace silniční dopravy může být rozdělena do několika kategorií podle úrovně detailů na makroskopické, mezoskopické a mikroskopické simulace.

5.1.1 Makroskopické simulace

Makroskopická simulace je nejjednodušší druh simulace, kdy se nesimulují jednotlivá vozidla, ale pouze agregované toky dopravy. Díky těmto zjednodušením je tato simulace nejméně výpočetně náročná [19].

5.1.2 Mezoskopické simulace

Mezoskopická simulace je zpravidla počítána po skupinách vozidel, které jedou stejnou cestou [20]. Mohou zahrnovat i některé charakteristiky jednotlivých vozidel. Úroveň detailů těchto simulací je tedy mezi makroskopickými a mikroskopickými simulacemi [21].

5.1.3 Mikroskopické simulace

V mikroskopických simulacích jsou zvlášť simulována jednotlivá vozidla. Simulace dále obsahuje jednotlivé jízdní pruhy, po kterých se vozidla pohybují, proto tento druh simulace simuluje reálný svět nejvěrněji. Lze ji použít například pro výpočet délky front na křižovatkách nebo průměrné doby čekání na křižovatkách. Takto detailní simulace je zároveň i výpočetně nejnáročnější. U těchto simulací se nejčastěji používá model s celulárním automatem [22] nebo model následování vozidel (car following model) [23].

Simulace se ve většině případů posouvá v čase ve fixních krocích. Jedná se tedy o time-stepped simulaci. Každé vozidlo má v daném kroku konkrétní polohu, rychlost a zrychlení. Většinou jsou tyto simulace počítány v jednosekundových krocích.

V každém kroku je nutné přepočítat polohu vozidla, případně i další data o vozidle. Když jsou dokončeny všechny výpočty, je možné zahájit výpočet dalšího kroku. Tento proces se opakuje stále dokola [24].

Namísto časových kroků může být simulace také řízena událostmi. Mezi tyto události se pak počítá například příjezd vozidla ke křižovatce. Tento druh simulace se však v mikroskopické simulaci dopravy nevyužívá příliš často a nebude dále v této práci uvažován.

5.2 Modely silniční dopravy v mikroskopické simulaci

Tato práce se bude dále zaměřovat na mikroskopické simulace. V této kapitole bude popsán model s celulárním automatem [22] a model následování vozidel (car following model) [23]. Tyto modely mají společné prvky. Liší se však hlavně v reprezentaci jízdních pruhů.

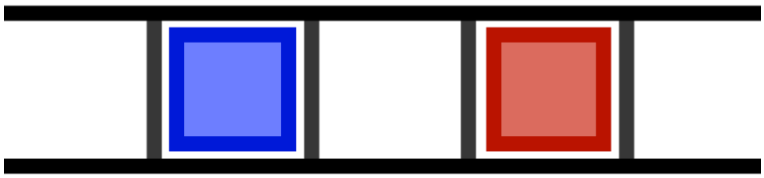
5.2.1 Model s celulárním automatem

Model s celulárním automatem byl navržen Kaiem Nagelem a Michaelem Schreckenbergem v roce 1992 [25]. V tomto modelu jsou jízdní pruhy, po kterých vozidla jezdí, rozděleny na buňky o pevné velikosti (viz Obrázek 5). Každá tato buňka může být buď volná, nebo zabrána právě jedním vozidlem, které na ní zrovna stojí. Každé vozidlo zabírá jednu buňku. Každé vozidlo se podle své rychlosti v každém časovém kroku přesune o daný počet buněk [25].

Délka časového kroku u původního modelu je 1 sekunda a velikost buňky je 7,5 metru. Všechna vozidla jsou stejně dlouhá a zabírají právě jednu buňku. Rychlost v jednotlivých vozidel je reprezentována celým číslem v rozsahu od 0 do v_{max}, kde v_{max} je maximální rychlost vozidel. Všechna vozidla mají definované následující chování. Pokud je rychlost vozidla v nižší než maximální rychlost v_{max} a nejbližší vozidlo před ním je vzdálené alespoň $v + 1$ buněk, rychlost vozidla je zvýšena o 1. Pokud nejbližší vozidlo před ním je vzdálené pouze i buněk, kde $i \leq v$, dojde ke zpomalení vozidla (tzn. ke snížení rychlosti na $v = i - 1$). Další pravidlo bere v úvahu kolísání rychlosti z důvodu měnící se dopravní situace a chování řidiče. Pokud je rychlost vozidla vyšší než 0, dojde s pravděpodobností p ke snížení rychlosti o 1. Bez

tohoto pravidla by byla simulace deterministická. Následně je vozidlo posunuto o příslušný počet buněk, který je dán rychlost v tohoto vozidla [25].

Existuje mnoho modifikací tohoto modelu. Mezi tyto změny patří například změna velikosti buňky, která je snížena ze 7,5 metru na 2,5 metru. Model může být také upraven tak, aby umožňoval vozidlům zabírat i více než jednu buňku, pokud se jedná o delší vozidlo. Vozidla pak mohou mít různou délku [26].



Obrázek 5: Ukázka reprezentace jízdního pruhu v modelu s celulárním automatem

5.2.2 Car following model

Car following model je starší než model s celulárním automatem. Tento model byl představen už v 50. letech 20. století [27]. V tomto modelu jízdní pruhy nejsou rozděleny na buňky. Vozidlo může být umístěno na kteroukoliv pozici v daném jízdním pruhu (viz Obrázek 6) [28]. Pozice vozidla v pruhu, rychlost i zrychlení je proto reprezentována reálným číslem.

Model je založen na reakci vozidla na chování vozidla, které jede před ním. V případě, že vozidlo před ním zpomaluje nebo jede nižší rychlostí, dané vozidlo také zpomalí, pokud se začne příliš přibližovat. V případě, že je cesta volná, vozidlo může postupně zrychlovat až na maximální rychlost.

Obrázek 6: Ukázka reprezentace jízdního pruhu v car following modelu

Existuje mnoho různých modelů, které jsou na tomto modelu založeny. Liší se hlavně chováním jednotlivých vozidel. V některých modelech je například simulováno plynulé udržování vzdálenosti mezi vozidly. V modelu GHR, který byl vytvořen v roce 1958 společností General Motors, je například zrychlení vyjádřeno následující rovnicí [29]:

$$a_n(t) = cv_n^m(t)\frac{\Delta v(t-T)}{\Delta x^l(t-T)} \tag{5.1}$$

$a_n(t)$ je zrychlení vozidla n v určitém čase t. v_n je rychlost vozidla v čase t. Δx je relativní vzdálenost mezi vozidly n a $(n-1)$. Δv je relativní rychlost mezi vozidly n a $(n-1)$. T je reakční doba řidiče. l, m a c jsou konstanty, které musí být kalibrovány přímo pro určitou silniční síť.

Lineární model, který byl založen na modelu GHR, obsahuje upravenou rovnici zrychlení [30]:

$$a_n(t) = C_1\Delta v(t-T) + C_2[\Delta x(t-T) - D_n(t)] \tag{5.2}$$

$a_n(t)$ je opět zrychlení vozidla n v určitém čase t. $D_n(t)$ je požadovaná vzdálenost mezi vozidly v čase t. Ta je dána touto rovnicí:

$$D_n(t) = \alpha + \beta v(t-T) + \gamma a_n(t-T) \tag{5.3}$$

v je rychlost vozidla n. Δx je relativní vzdálenost mezi vozidly n a $(n-1)$. Δv je relativní rychlost mezi vozidly n a $(n-1)$. T je reakční doba řidiče. α, β, γ, C_1 a C_2 jsou konstanty, které musí být opět kalibrovány přímo pro určitou silniční síť.

6 Návrh aplikace

6.1 Volba technologií

Jako programovací jazyk jsem zvolil jazyk C# [31] a technologii .NET Framework [32]. Jako technologii pro paralelní výpočty jsem zvolil OpenCL [8]. Celá tato práce je vyvíjena pomocí vývojového prostředí Visual Studio 2017 od společnosti Microsoft. Důvodem pro výběr těchto prostředků byla dohoda se zadavatelem práce, který doporučil použít prostředky co nejbližší jazyku Java.

6.1.1 Jazyk C#

Jazyk C# je vysokoúrovňový objektově orientovaný programovací jazyk vyvinutý firmou Microsoft v roce 2000 a stále se aktivně rozvíjí. Tento jazyk byl přímo ovlivněn jazyky C++ a Java [33], a proto obsahuje podobné prvky a syntaxi.

Oproti jazyku C++ neumožňuje vícenásobnou dědičnost. Navíc je oproti C++ typově bezpečnější, protože implicitní přetypování lze provést, pouze pokud je považováno za bezpečné [34]. Dále neumožňuje používat globální proměnné, všechny proměnné a metody musí být definovány uvnitř tříd. Pro zpřehlednění zdrojových kódů nepotřebuje ani nepodporuje dopřednou deklaraci, na pořadí metod ve zdrojovém kódu nezáleží.

Mezi výhody patří možnost pracovat s pamětí na nízké úrovni [35]. Jazyk C# a technologie .NET Framework dále umožňuje jednoduše volat funkce z libovolných nativních a systémových knihoven.

Jazyk C# je zpravidla překládán do jazyka Common Intermediate Language (CIL) [36]. Jedná se o jazyk nižší úrovně zásobníkového typu, který je zpracováván virtuálním strojem. Ten ho pomocí just-in-time (JIT) kompilace překládá do kódu podporovaného přímo procesorem. Tento převod probíhá přímo za běhu programu.

6.1.2 Technologie .NET Framework

Rozhraní .NET Framework je prostředí potřebné pro běh aplikace napsané v jazyce C#, které dále obsahuje sadu knihoven, kterou mohou aplikace využívat. Tyto knihovny obsahují sadu základních funkcí pro interakci s uživatelem, matematické

funkce, kolekce, funkce pro vytváření procesů a vláken, funkce pro práci se soubory a proudy a nástroje pro tvorbu grafického uživatelského rozhraní [37].

Jedná se o volitelnou součást operačního systému Microsoft Windows a zpravidla je automaticky nainstalováno při instalaci operačního systému [38]. Poslední verze rozhraní .NET Framework je 4.7.2 [39], která je součástí operačního systému Microsoft Windows 10, dále je podporována všemi staršími verzemi systému až k verzi Windows 7. Pro vývoj aplikací na operační systém Windows XP je nutné použít verzi rozhraní .NET Framework 4.0 [40].

Mezi hlavní nevýhody patří zejména složitější spouštění těchto aplikací na jiných operačních systémech než Microsoft Windows. Na dalších operačních systémech, například Linux a Mac OS X, lze pro spouštění a překlad aplikací využít open-source řešení Mono [41], které však není firmou Microsoft přímo podporováno a neobsahuje všechny funkce dostupné v rozhraní .NET Framework. Stále se však aktivně vyvíjí a snaží se dále zvyšovat kompatibilitu s rozhraním .NET Framework [42].

Aplikace, které využívají jen základní funkce .NET Framework, lze pomocí řešení Mono bez úprav spustit. Aplikace využívající grafické uživatelské rozhraní je vhodnější přepracovat, aby využívalo grafickou knihovnu pro daný operační systém, protože každý operační systém podporuje odlišné grafické prvky.

6.1.3 Technologie OpenCL pro paralelní výpočty

Jedná se o nejrozšířenější technologii a zároveň snadno přenositelnou. Lze ji poměrně snadno využít i v aplikaci psané v jazyce C#. Pravděpodobně nejvhodnější je využít pro to knihovnu NOpenCL [43].

Tato knihovna pouze volá funkce OpenCL API, takže zde nedochází ke zbytečné ztrátě výkonu. Jednotlivé metody této knihovny jsou pojmenovány podobně jako původní funkce OpenCL API, což usnadňuje práci s touto knihovnou. Protože knihovna pouze volá OpenCL API, i OpenCL programy jsou nadále psány ve speciálním jazyce OpenCL C a jsou tak zcela přenositelné. Proto výsledná aplikace také může být spuštěna na všech podporovaných operačních systémech.

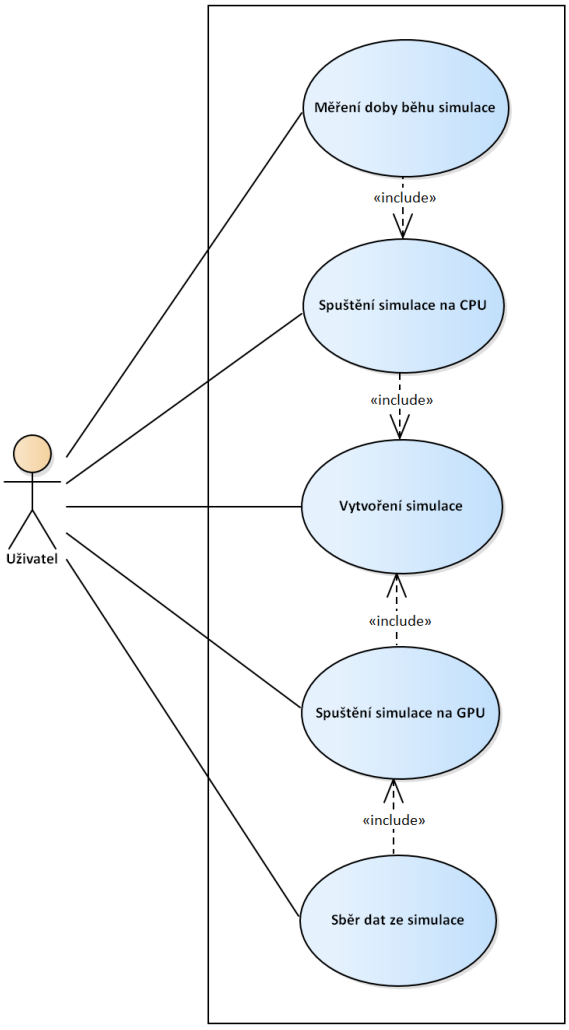
Na cílovém počítači je pouze nutné mít nainstalované příslušné ovladače OpenCL. Ty jsou zpravidla nainstalovány spolu s ovladači od výrobce grafického

adaptéru, který je v počítači nainstalován a na kterém budou paralelní výpočty spouštěny. Tato podpora je garantována všemi hlavními výrobci grafických karet a čipů, konkrétně společnostmi AMD [9], Intel [10] i Nvidia [11].

Další možností by bylo využít například knihovnu Campy [44] nebo GpuLinq [45]. Tyto knihovny umožňují psát paralelní výpočty přímo v jazyce C# pomocí vysokoúrovňových konstrukcí. To by však snižovalo přenositelnost a nedovolovalo by to využít veškeré funkce technologie OpenCL, proto byla vybrána knihovna NOpenCL.

6.2 Případy užití

Diagram případů užití je zachycen na Obrázku 7. Jednotlivé případy užití jsou popsány v následujících podkapitolách.



Obrázek 7: Diagram případů užití

6.2.1 Vytvoření simulace

Uživatel aplikace vybere simulační model (model s celulárním automatem nebo car following model) a zadá všechny parametry simulace (vzdálenost mezi křižovatkami, počet křižovatek, počáteční počet vozidel, maximální počet vozidel a pravděpodobnost generování vozidel). Aplikace s využitím těchto parametrů vytvoří příslušnou simulaci, která bude obsahovat odpovídající silniční síť.

6.2.2 Spuštění simulace na CPU

Uživatel na vytvořené simulaci spustí výpočet kroku simulace na jednotce CPU. Aplikace využije referenční implementaci a výpočet provede. Nakonec uživateli prezentuje nový stav simulace.

6.2.3 Spuštění simulace na GPU

Uživatel nejprve vybere jednotku GPU, na které bude proveden výpočet. Následně na vytvořené simulaci spustí výpočet kroku simulace, pro který se využije vybrané zařízení. Aplikace využije implementaci, která používá technologii OpenCL, a výpočet provede. Nakonec uživateli prezentuje nový stav simulace.

6.2.4 Měření doby běhu simulace

Uživatel spustí výkonnostní testy za účelem změření doby běhu jednotlivých implementací. Uživatel si zvolí, které testy budou provedeny. Bude tedy možné zvolit, který simulační model bude otestován (model s celulárním automatem nebo car following model), zda bude použita referenční implementace nebo implementace využívající OpenCL a velikost silniční sítě, která bude pro testování použita. Aplikace vybrané testy automaticky provede s dostatečným množstvím zopakování. Výsledky testů uloží do souboru, aby je bylo možné dále analyzovat a porovnat dobu běhu výpočtu simulace, kdy je pro výpočet použita jednotka CPU, s dobou běhu, kdy je použita jednotka GPU. Následně z těchto dat uživatel může určit, zda je vhodnější pro výpočet použít CPU nebo GPU.

6.2.5 Sběr dat ze simulace

Uživatel bude moci v libovolném kroku simulace sbírat data. Aplikace pro tento účel poskytne vizualizaci silniční sítě, některé důležité informace vypíše i textově. Mezi

tyto informace patří například celkový počet vozidel, která se zrovna nachází na silniční síti, a počet stojících vozidel na určité křižovatce. Aplikace však primárně neslouží ke sběru těchto informací. Hlavním cílem práce je porovnání doby běhu, kdy je pro simulaci použita jednotka CPU, s dobou běhu, kdy je použita jednotka GPU, a následné porovnání.

7 Popis implementace

V následujících podkapitolách je podrobně popsána samotná implementace aplikace.

7.1 Struktura aplikace

Aplikace se skládá z jednotlivých menších součástí. Pro rozdělení aplikace na větší celky jsou využity jmenné prostory, které jsou součástí jazyka C#. Ty obsahují jednotlivé třídy.

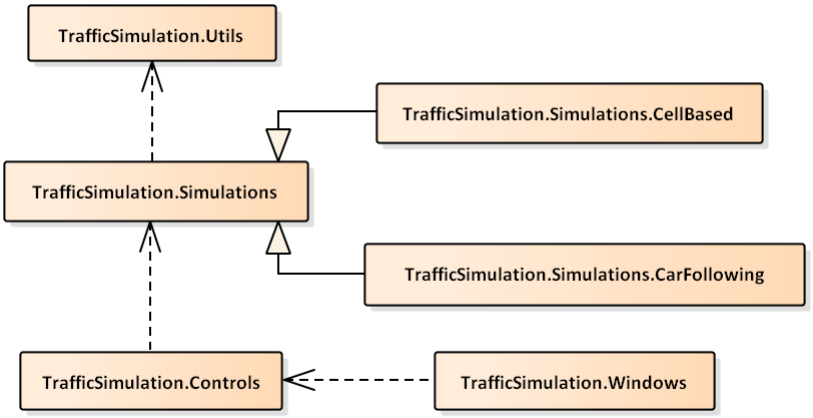
Diagram závislostí jmenných prostorů je zachycen na Obrázku 8. Kompletní diagram tříd je umístěn v příloze B. Všechny jmenné prostory jsou součástí hlavního jmenného prostoru `TrafficSimulation`. Třídy a rozhraní, které se týkají samotné simulace, jsou uloženy ve jmenném prostoru `TrafficSimulation.Simulations`. Jednotlivé simulační modely se nacházejí v samostatných jmenných prostorech. Model s celulárním automatem se tedy nachází ve jmenném prostoru v `TrafficSimulation.Simulations.CellBased` a car following model v prostoru `TrafficSimulation.Simulations.CarFollowing`.

Jmenný prostor `TrafficSimulation.Utils` obsahuje různé pomocné třídy. Zejména jde o různé třídy, které například usnadňují přístup k funkcím technologie OpenCL.

Jmenný prostor `TrafficSimulation.Controls` obsahuje všechny vytvořené prvky uživatelského rozhraní. Mezi ně patří prvek, který vykresluje samotnou vizualizaci jednotlivých modelů.

Jmenný prostor `TrafficSimulation.Windows` obsahuje všechna okna a dialogy. Aplikace využívá pro uživatelské rozhraní sadu tříd Windows Forms, která je přímo součástí rozhraní .NET Framework.

Zdrojové soubory obsahující programy pro technologii OpenCL byly uloženy samostatně do složky `Kernels`. Při sestavování aplikace jsou tyto soubory automaticky vloženy jako vlastní data přímo do EXE souboru aplikace a následně jsou při běhu aplikace z těchto dat načteny.



Obrázek 8: Diagram jmenných prostorů

7.2 Provádění výpočtů s využitím technologie OpenCL

Pro usnadnění práce s technologii OpenCL bylo implementováno několik tříd ve jmenném prostoru `TrafficSimulation.Utils`. Využívá knihovnu NOpenCL (viz kapitola 6.1.3).

7.2.1 Třída OpenCLDevice

Třída `OpenCLDevice` představuje zařízení OpenCL. Před zahájením výpočtů na daném zařízení je nutné nejprve zařízení vybrat, vytvořit pro něj kontext a frontu příkazů, do které se budou postupně přidávat jednotlivé požadavky pro vypočítání. O tuto funkcionalitu se stará právě tato třída.

Obsahuje proto atributy, které se vážou ke konkrétním zařízením. Konkrétně jde o kontext (třída `Context`), frontu příkazů (třída `CommandQueue`) a slovník, který obsahuje zkompilované kernely pro dané zařízení. Tak je možné vytvořit instance těchto tříd pouze jednou a využívat je po celou dobu běhu aplikace. Kompilování kernelů je navíc poměrně časově náročné, záleží však na rychlosti překladače, který je součástí ovladačů pro OpenCL.

7.2.2 Třída OpenCLBuffer

Třída `OpenCLBuffer` představuje blok paměti, se kterým může zařízení OpenCL manipulovat při výpočtech. Také je to jediný způsob, jakým lze přesunout připravená data z paměti RAM do paměti zařízení OpenCL. OpenCL buffer musí být nejprve vytvořen s využitím kontextu zařízení. Jako parametr je nutné dodat ukazatel na zvolenou paměť, velikost paměti v bajtech a informace o tom, zda má být daný buffer pouze pro čtení.

Dále třída umožňuje zkopírovat změněný blok paměti ze zařízení OpenCL zpět do původního umístění pomocí metody `Synchronize()`. Interně využívá funkci `clEnqueueMapBuffer()` [46], která namapuje daný buffer zpět do paměťového prostoru hostitele. Protože byl buffer vytvořen s příznakem `CL_MEM_USE_HOST_PTR`, je buffer namapován přímo na původní místo. Následně je mapování zrušeno pomocí funkce `clEnqueueUnmapMemObject()` [47].

Pro případ, kdy je nutné obsah bufferu změnit ze strany hostitele bez nutnosti vytvářet nový, existuje metoda `MapAsWritable()`. Aby bylo možné provést změnu paměti, než dojde opět ke zrušení mapování, využívá se pomocná struktura `BufferWriteAccessToken`, která implementuje rozhraní `IDisposable` [48]. Třídy nebo struktury, které toto rozhraní implementují, pak mohou být použity v konstrukci `using` jazyka C# [49]. Při použití této konstrukce je zajištěno, že na konci daného bloku je automaticky zavolána metoda `Dispose()`, a tím je zrušeno mapování daného bufferu.

7.2.3 Třída OpenCLKernel

Třída `OpenCLKernel` představuje kernel zkompilovaný pro určité zařízení. Samotná kompilace probíhá automaticky s využitím původních zdrojových kódů programů pro OpenCL zavoláním příslušné funkce OpenCL. Třída se stará o přiřazování vstupních parametrů i o následné spuštění výpočtu. Parametry mohou být přiřazeny postupně v daném pořadí, nebo ručně podle indexu.

Pomocí metody `BindBuffer()` je možné přiřadit vytvořený buffer jako parametr, aby s ním bylo možné v OpenCL programu pracovat. Případně je možné nejprve nechat buffer vytvořit a následně přiřadit jako parametr. Třída je pak sama stará o uvolnění prostřední po skončení výpočtu.

Metoda `BindValue()` slouží pro přiřazení číselné hodnoty jako parametr. Je možné přiřadit celočíselnou hodnotu typu `int` nebo číslo s plovoucí desetinnou čárkou typu `float`. Pro přiřazení hodnoty jako konkrétní parametr podle indexu je možné využít metodu `BindValueByIndex()`.

Po přiřazení všech parametrů je možné zahájit výpočet pomocí metody `Run()`. Metoda provede přidání příkazu pro výpočet do fronty příkazů zařízení. Parametrem je předána velikost dat, na kterých bude výpočet proveden. Metodu je možné volat opakovaně.

Po provedení všech výpočtů je nutné zavolat metodu `Finish()`, která nejprve počká na provedení všech příkazů ve frontě a následně uvolní všechny automaticky vytvořené buffery.

7.2.4 Třída OpenCLKernelSet

Třída `OpenCLKernelSet` představuje množinu kernelů, které jsou obsaženy v jednom OpenCL programu. Jeden program může obsahovat několik různých kernelů, lze to chápat jako různé vstupní body jednoho programu. Každý kernel je označen jménem (název dané funkce) a může mít různý počet parametrů.

7.2.5 Třída OpenCLDispatcher

Třída `OpenCLDispatcher` slouží k inicializaci zařízení OpenCL a vytváření instancí zmíněných tříd.

Vlastnost `Devices` slouží k získání seznamu dostupných zařízení OpenCL (instance třídy `OpenCLDevice`), se kterými lze dále pracovat.

Dále obsahuje metodu `DisposeDevices()` sloužící k uvolnění všech inicializovaných zařízení. Metoda `CreateBuffer()` umožňuje vytvářet instance třídy `OpenCLBuffer` podle zadaných parametrů.

Pro zkompilování OpenCL programu pro dané zařízení se využívá metoda `Compile()`. Tato metoda nejprve inicializuje zařízení zavoláním metody `InitializeDevice()`. Tím se vytvoří příslušný kontext a fronta příkazů. Následně je načten zdrojový kód OpenCL programu, který je zkompilován pro dané zařízení. Metoda vrací instanci třídy `OpenCLKernelSet`, takže je pak možné vybrat příslušný

kernel v OpenCL programu a výpočet zahájit. Tato instance je navíc uložena do slovníku k danému zařízení a při opakovaném volání se stejnými parametry je instance pouze z tohoto slovníku získána. Tím je zajištěno, že kompilace programu na daném zařízení je provedena pouze jednou.

7.3 Simulace silniční dopravy

V práci byly implementovány dva dopravní modely – celulární automat a car following model. Oba modely mají některé společné vlastnosti, v obou se vyskytují stejné elementy silniční sítě a oba modely využívají krokový běh času s délkou kroku 1 sekunda. U těchto modelů bylo dále nutné umožnit generování silniční sítě a následně spustit simulaci na jednotce CPU nebo GPU.

7.3.1 Společné vlastnosti obou modelů

Základ pro oba modely představuje abstraktní třída `SimulationBase` ve jmenném prostoru `TrafficSimulation.Simulations`.

Tato obsahuje několik abstraktních metod, které musí jednotlivé konkrétní třídy simulace implementovat. Abstraktní metoda `GenerateNew()` slouží pro vygenerování nové simulace podle zadaných parametrů, konkrétně jde o vzdálenost mezi křižovatkami, počet křižovatek, počáteční počet vozidel, maximální počet vozidel a parametr λ exponenciálního rozdělení, které je poté využíváno pro generování náhodných čísel při generování nových vozidel.

Dále je zde abstraktní metoda, která se využívá pro vykonání jednoho časového kroku pomocí referenční implementace. V tomto případě je pro výpočet použita jednotka CPU. Také se zde nachází další abstraktní metoda, která se využívá pro vykonání jednoho časového kroku pomocí technologie OpenCL. Při volání metody je nutné specifikovat OpenCL zařízení, na kterém bude výpočet proveden. Abstraktní metoda `DoBatchOpenCL()` umožňuje vykonat libovolný počet časových kroků pomocí technologie OpenCL. Tuto metodu je vhodné využít, pokud je předem známý počet požadovaných časových kroků a je větší než 1. Implementace této abstraktní metody pak může použít vhodnější postup provedení výpočtu, například může omezit počet kopírování paměti.

Nakonec je zde abstraktní metoda sloužící pro ověření, zda je aktuální stav simulace platný. Simulace může například ověřit, zda mezi sebou vozidla nekolidují. Abstraktní metoda `InnerLoadFromStream()` slouží pro načtení dat stavu simulace, která jsou svázaná přímo s určitým typem simulace, z datového proudu a abstraktní třída `InnerSaveToStream()` naopak slouží k uložení dat stavu simulace do datového proudu.

7.3.2 Perzistence stavu simulace

Součástí třídy `SimulationBase` je i metoda `LoadFromStream()`, která umožňuje vytvořit novou instanci simulace podle dat uloženého stavu simulace z datového proudu. Metoda využívá pro načítání třídu `BinaryReader` [50], která je obsažená v prostředí .NET Framework. Data jsou uložena ve vlastním binárním datovém formátu. Všechna čísla jsou ukládána jako little-endian. Nejprve je načteno 16-bitové číslo představující verzi datového formátu. To se musí aktuálně shodovat s hodnotou 1. Následně je načtena 16-bitová hodnota určující typ simulace. Může nabývat těchto hodnot:

- 0 – model s celulárním automatem,
- 1 – car following model.

Poté je načteno číslo posledního časového kroku simulace a je zavolána metoda `InnerLoadFromStream()` konkrétní třídy, která načte zbytek dat uloženého stavu.

Statická metoda `SaveToStream()` umožňuje uložit stav simulace do datového proudu, aby jej bylo možné opět načíst. Metoda pracuje obdobně. Využívá pro zápis třídu `BinaryWriter` [51] a ve stejném pořadí data zapíše do příslušného datového proudu. Dále volá metodu `InnerSaveToStream()` pro načtení zbytku dat.

7.4 Model s celulárním automatem

Třída `CellBasedSim` obsahuje implementaci simulace silniční dopravy využívající model s celulárním automatem, který je založen na modelu Nagel-Schreckenberg [25]. Implementovaný model se však liší tím, že používá jinou velikost buňky a umožňuje

použít různou velikost u jednotlivých vozidel. Třída je uložena ve jmenném prostoru `TrafficSimulation.Simulations.CellBased`.

Implementace počítá s tím, že velikost jedné buňky je 2,5 metru a maximální rychlost v_{max} je 54 km/h (resp. 6 buněk za sekundu), což je o 4 km/h více než maximální povolená rychlost v obci. Minimální délka vozidla je rovna 1 buňce (2,5 metru) a maximální délka je rovna 6 buňkám (15 metrů).

7.4.1 Struktura dopravní sítě

Tato třída využívá pro uložení stavu dopravní sítě několik struktur. V následujících podkapitolách budou postupně vysvětleny.

Cell

Struktura `Cell` představuje buňku. Buňky spolu tvoří jednotlivé jízdní pruhy v silniční síti. Struktura obsahuje proměnné `T1`, `T2`, `T3`, `T4` a `T5`, které obsahují index buňky, jež následuje za současnou buňkou. Ze současné buňky tedy může vést až 5 rozdílných cest. Tento počet byl zvolen z toho důvodu, že by měl být dostatečný pro všechny případy použití. Proměnné pro cesty, které nejsou využity, obsahují hodnotu -1. Tímto způsobem jsou spojeny jednotlivé buňky. Ke každé cestě patří jedna z proměnných `P1`, `P2`, `P3`, `P4` nebo `P5`, které určují pravděpodobnost, že se vozidlo touto cestou vydá. Součet těchto proměnných se musí rovnat 1.

V případě, že je buňka součástí křižovatky, proměnná `JunctionIndex` je rovna indexu dané křižovatky. Pokud není, je rovna hodnotě -1. Pokud buňka obsahuje terminátor, má proměnná `JunctionIndex` speciální hodnotu -2. Proměnná `NearestJunctionIndex` je index křižovatky, která je nejblíže v daném jízdním pruhu. Tato hodnota se využívá při zjišťování počtu vozidel, které čekají na dané křižovatce. Během výpočtu simulace nejsou v této struktuře upravována žádná data.

CellToCar

Struktura `CellToCar` obsahuje dodatečné informace pro každou buňku. Na rozdíl od struktury `Cell` je ale umožněna změna dat struktury během výpočtu simulace. Struktura obsahuje pouze jednu proměnnou `CarIndex`. Tato proměnná značí index vozidla, které se právě nachází na dané buňce. V případě, že je buňka prázdná, je hodnota rovna -1.

Junction

Struktura `Junction` představuje jednotlivé křižovatky. Proměnná `CellIndex` je rovna indexu buňky, která danou křižovatku obsahuje. Proměnná `WaitingCount` pak obsahuje počet vozidel, která na dané křižovatce právě čekají. To jsou vozidla, která mají nulovou rychlost.

Generator

Struktura `Generator` obsahuje informace o jednotlivých generátorech vozidel. Ty jsou při generování silniční sítě umístěny po obvodu sítě. Mohou být však umístěny kdekoliv. Proměnná `CellIndex` je rovna indexu buňky, na které se mají vozidla vytvářet. Proměnná `ProbabilityLambda` je rovna parametru λ exponenciálního rozdělení, které je použito při generování náhodných čísel. Proměnná `TimeLeft` obsahuje počet zbývajících časových kroků, než dojde k vygenerování nového vozidla. Po uplynutí této doby je generátor aktivován, je vytvořeno nové vozidlo a proměnná je nastavena na novou hodnotu, která je vygenerována exponenciálním rozdělením s parametrem λ.

Car

Struktura `Car` představuje jednotlivá vozidla. Proměnná `Position` je rovna indexu buňky, na které se vozidlo právě nachází. V případě, že vozidlo zabírá více než jednu buňku, jedná se o poslední buňku, na které stojí zadní část vozidla. Pokud se vozidlo na silniční síti nenachází, například bylo odstraněno terminátorem, je hodnota proměnné `Position` rovna -1.

Proměnná `Speed` je rovna aktuální rychlosti vozidla. Ta je vyjádřena v počtu buněk za časový krok (resp. za sekundu). Proměnná `Size` obsahuje délku vozidla, resp. počet buněk, které dané vozidlo zabírá.

CellUi

Struktura `CellUi` obsahuje další proměnné pro jednotlivé buňky, které jsou nutné pouze pro potřeby uživatelského rozhraní a nejsou potřebné pro výpočet simulace. Tyto informace jsou uložené zvlášť, ale vždy odpovídají příslušné buňce (viz struktura `Cell`) v poli buněk. Tyto proměnné jsou souřadnice `X` a `Y` pro vykreslení buňky na správném místě v uživatelském rozhraní.

CarUi

Struktura `CarUi` obsahuje další proměnné pro jednotlivá vozidla, které jsou nutné pouze pro potřeby uživatelského rozhraní a nejsou potřebné pro výpočet simulace. Tyto informace jsou opět uložené zvlášť, ale vždy odpovídají příslušnému vozidlu (viz struktura `Car`) v poli vozidel. Obsahuje proměnnou `Color`, která značí barvu vozidla, která je poté využita ve vizualizaci.

Uložení struktur

Všechny tyto struktury jsou použity v polích a všechny vazby mezi položkami v těchto polích jsou realizovány pomocí indexů, aby je bylo možné použít i pomocí GPU. Nelze zde použít ukazatele, protože jednotky CPU a GPU mezi sebou nesdílí adresní prostor a na straně GPU by měly ukazatele rozdílnou hodnotu než na straně CPU.

7.4.2 Generování silniční sítě

Třída `CellBasedSim` dále obsahuje generátor sítě křižovatek, který umožňuje vygenerovat síť o daném počtu křižovatek, počtu buněk, které mezi těmito křižovatkami jsou, počtu vozidel, které budou na této síti náhodně rozmístěny, maximálním počtu vozidel a pravděpodobnosti generování vozidel. Generátor u každé křižovatky vygeneruje čtyři výjezdy (na okrajových křižovatkách je tento počet nižší) a ke každé cestě je náhodně přiřazena pravděpodobnost, že se danou cestou vozidla vydají (proměnné `P1`, `P2`, `P3`, `P4` a `P5` ve struktuře `Cell`).

Generování cesty mezi křižovatkami je implementováno v metodě `CreateLane()`. Dále vygeneruje příslušné generátory a terminátory na okraji sítě. To je implementováno zvlášť v metodě `CreateGeneratorsAndTerminators()`.

Poté naplní síť zadaným počtem vozidel. Jejich pozici a délku vygeneruje náhodně tak, aby vozidla navzájem nekolidovala. Délka vozidla je vygenerována v rozsahu 1 až 6. Vygenerování nové sítě je provedeno zavoláním metody `GenerateNew()`.

7.4.3 Referenční implementace

Dále obsahuje metodu `DoStepReference()`, která umožňuje vykonat jeden časový krok simulace pomocí referenční implementace s využitím jednotky CPU. Tato

implementace je napsána přímo v jazyce C#. Metoda využívá pro paralelní výpočty statickou metodu `Parallel.ForEach()` [52], která je součástí knihoven rozhraní .NET Framework. Tato metoda automaticky zvolí optimální počet vláken, která budou použita pro dané paralelní výpočty. Dále je využita metoda `Partitioner.Create()` [53], která umožňuje rozdělit zvolený interval, nad kterým budou výpočty prováděny, na několik menších intervalů a tím dále zefektivnit práci v jednotlivých vláknech, aby v každém vlákně byla ideálně zpracována více než jedna položka.

Při výpočtu simulačního kroku je nejprve vynulován počet vozidel, která čekají na křižovatkách (proměnná `WaitingCount` ve struktuře `Junction`). Dále jsou paralelně zpracována všechna vozidla, která se nachází v silniční síti, zavoláním metody `DoStepCar()`, která má jeden parametr představující index vozidla, které se má zpracovat. Tato metoda nejprve najde všechny buňky, na kterých je dané vozidlo umístěno, zavoláním metody `FindCarCells()`. Metoda `FindCarCells()` začne na buňce s indexem, který je rovný pozici vozidla (proměnná `Position` ve struktuře `Car`), a postupně dál prochází buňky jízdního pruhu, dokud počet nalezených buněk neodpovídá délce vozidla. Všechny indexy nalezených buněk jsou postupně uloženy do pole C_1.

Po nalezení okupovaných buněk je zavolána metoda `FillFrontCells()`. Tato metoda najde buňky, které se nachází v daném jízdním pruhu dál před vozidlem. Metoda hledá, dokud nenajde počet buněk, který odpovídá rychlosti v daného vozidla. To se dále využívá pro kontrolu, že je cesta před vozidlem volná. Využívá k tomu metodu `FindNextCell()`. Všechny indexy nalezených buněk jsou postupně uloženy do druhého pole C_2.

S využitím získaných informací je dále vypočítána nová rychlost daného vozidla. Nejprve je zkontrolováno, že vozidlo má před sebou dostatečné množství místa (prázdných buněk) vzhledem k jeho rychlosti v. Využívá se k tomu pole C_2. Pokud ano, vozidlo může dále zrychlovat až na maximální povolenou rychlost 6 buněk za časový krok (resp. sekundu). V každém kroku simulace může vozidlo zrychlit o 1 buňku za časový krok. V případě, že před vozidlem není dostatečný počet prázdných buněk, vozidlo zpomalí na rychlost odpovídající počtu prázdných buněk. Pokud je vozidlo v pohybu, s pravděpodobností p, která je rovna hodnotě 0,2, dojde ke snížení rychlosti vozidla o 1 buňku za časový krok.

Po vypočtení nové rychlosti vozidla je vypočítána nová pozice vozidla. Vozidlo je postupně přesouváno po prázdných buňkách z pole C_2. Pokud narazí na buňku, která obsahuje terminátor (hodnota proměnné `JunctionIndex` buňky je rovna -2), vozidlo je ze silniční sítě odstraněno. Proměnná `Position` tohoto vozidla je pak nastavena na hodnotu -1 a proměnná Speed je nastavena na 0. U buněk s indexy, které byly uloženy do pole C_1, se nastaví proměnná `CarIndex` na hodnotu -1. Tím jsou dané buňky, na kterých vozidlo stálo, opět označeny jako prázdné.

V případě, že následující buňka není terminátor, pomocí atomické operace `CompareExchange` dojde k zabrání následující buňky daným vozidlem. Využívá se k tomu statická metoda `Interlocked.CompareExchange()` [54]. Je volána se třemi parametry – referencí na proměnnou `CarIndex` následující buňky, indexem vozidla a hodnotou -1 představující prázdnou buňku. Tím je zajištěno, že do proměnné `CarIndex` je zapsán index vozidla, pouze pokud je daná buňka prázdná. V případě, že není, vozidlo je ihned zastaveno nastavením proměnné `Speed` na hodnotu 0. To se může stát například, když na křižovatce stihne jiné vozidlo vjet do stejného jízdního pruhu. Dále jsou uvolněny buňky, na kterých se vozidlo nacházelo před přesunem, nastavením proměnné `CarIndex` na -1. Proměnná `Position` vozidla je nastavena také na novou hodnotu.

V případě, že vozidlo stojí (proměnná `Speed` je rovna 0), je nalezena nejbližší křižovatka pomocí proměnné `NearestJunctionIndex` buňky, na které vozidlo stojí. U této křižovatky je inkrementován počet stojících vozidel o 1 pomocí atomické operace. Využívá se k tomu statická metoda `Interlocked.Increment()` [55].

Následně jsou obdobně paralelně zpracovány všechny generátory zavoláním metody `SpawnCars()`, která má jeden parametr představující index generátoru, který se má zpracovat. Nejprve je ověřeno, že proměnná `TimeLeft` generátoru je rovna 0. Pokud není, je proměnná dekrementována a výpočet je ukončen. Pokud je, generátor může být opět znovu aktivován. Nejprve je vypočítána nová hodnota proměnné `TimeLeft` pomocí exponenciálního rozdělení. Následně je vygenerováno náhodné číslo představující délku nového vozidla a je zkontrolováno, že v oblasti, ve které má být nové vozidlo vygenerováno, je dostatek místa. Využívá se k tomu opět metoda `FillFrontCells()`. Index buňky, na které daný generátor generuje vozidla, je uveden jako parametr při volání této metody. Následně je zkontrolováno, že získané buňky jsou

neobsazené. V případě, že jsou všechny podmínky splněny, je nutné projít pole vozidel a nalézt neobsazené místo. Využívá se zde metoda `Interlocked.CompareExchange()`. Je volána s referencí na proměnnou `Position` vozidla, indexem buňky generátoru a hodnotou -1 představující prázdnou buňku. Výsledek volání metody je porovnán s hodnotou -1. Tím je zaručeno, že index buňky generátoru je zapsán do proměnné `Position`, pouze když byla předtím rovna -1. Pokud není, je otestováno další vozidlo v poli. Následně je nastavena rychlost nového vozidla na 0, délka vozidla na vygenerovanou hodnotu a vozidlo je přidáno do silniční sítě (viz struktura `CellToCar`).

7.4.4 Spouštění výpočtů pomocí technologie OpenCL

Třída dále implementuje metody `DoStepOpenCL()` a `DoBatchOpenCL()`. Obě metody potřebují jako parametry instanci třídy `OpenCLDispatcher` a instanci třídy `OpenCLDevice`, která představuje zařízení, na kterém bude výpočet vykonán.

V případě metody `DoStepOpenCL()` je nejprve zkompilován připravený OpenCL program. Následně jsou získány ukazatele na jednotlivá pole struktur, které obsahují stav simulace. Poté jsou vytvořeny příslušné buffery a je spuštěn kernel `DoStepCar`, který přímo odpovídá metodě `DoStepCar()` referenční implementace. Následně je spuštěn i kernel `SpawnCars`, který opět odpovídá metodě `SpawnCars()` referenční implementace. Po dokončení těchto výpočtů je změněná paměť automaticky zkopírována do původního umístění (viz kapitola 7.2.3) a tím je simulace posunuta o jeden časový krok.

V případě metody `DoBatchOpenCL()`, která umožňuje provést libovolné množství časových kroků najednou, je opět zkompilován OpenCL program a jsou získány ukazatele na jednotlivá pole struktur. Následně jsou ručně vytvořeny jednotlivé buffery, které se využívají během celého výpočtu. Nemusí se tak zbytečně kopírovat paměť do zařízení OpenCL a zpět při každém kroku a tím je celý výpočet urychlen. Následně jsou přiřazeny vytvořené buffery jako parametry jednotlivých kernelů. To je nutné provést pouze jednou. Při opakovaném spuštění kernelu pak budou využity ty samé parametry. Poté jsou pomocí konstrukce `for` zavolány oba potřebné kernely (`DoStepCar` a `SpawnCars`). Tím je proveden potřebný počet časových kroků. Nakonec je zavolána

u použitých kernelů metoda `Finish()`. Všechny buffery jsou uvolněny automaticky, protože je použita konstrukce `using` [49].

7.4.5 Implementace s využitím technologie OpenCL

Zdrojový kód OpenCL je uložen v souboru `CellBasedSim.cl` ve složce `Kernels`. Kód je do určité míry shodný s kódem referenční implementace, protože kód referenční implementace byl psán tak, aby se následně co nejvíce shodoval s implementací pro OpenCL a neobsahovat nějaké speciální konstrukce, které nejsou v jazyce OpenCL C dostupné.

Místo pojmenovaných konstant využívá konstrukci `#define`. Dále obsahuje opět definici všech použitých struktur. Ta je zapsána podobně, jako se zapisuje v jazyce C, ale navíc obsahuje parametr `__attribute__((aligned(4)))` [56], který vynutí zarovnání na 4 bajty. V jazyce C# je toto vynuceno pomocí atributu `StructLayout` [57]. Tím je zaručeno, že na straně CPU i GPU budou struktury v paměti zarovnány stejně.

Po definici struktur následují jednotlivé funkce, které se využívají při výpočtu. Ty se liší hlavně tím, že jako parametry mají navíc ukazatele na jednotlivá pole struktur, která při výpočtu využívají. Navíc musí být opatřeny klíčovým slovem `global` [58], protože ukazují na buffery, které jsou uloženy v globální paměti. Funkce `FindCarCells()`, `FindNextCell()` a `FillFrontCells()` jsou prakticky shodné s odpovídajícími metodami referenční implementace. Místo referencí, které jsou součástí jazyka C#, se nyní používají ukazatele.

Větší rozdíl je patrný až ve funkci `DoStepCar()`, která je označena klíčovým slovem `kernel` [59]. Tím může být následně použita jako kernel pro zahájení výpočtů. Také obsahuje všechny parametry, které byly přiřazeny před zavoláním samotného kernelu (viz kapitola 7.4.4). Pro získání indexu, ke kterému se bude výpočet vztahovat, je využita funkce `get_global_id()` [60] s parametrem 0, protože byl specifikován pouze jeden rozsah, se kterým se bude pracovat. Dále je nutné zkontrolovat, že daný index opravdu nepřesahuje hranice vstupního pole (v tomto případě se jedná o pole vozidel).

Další rozdíl se nachází ve využitých atomických operacích. Operace `CompareExchange` je v OpenCL definována jako funkce `atomic_cmpxchg()` [61]

s rozdílným pořadím parametrů. První parametr je opět ukazatel na paměť, která se má změnit, dále je očekávaná původní hodnota a nakonec je nová hodnota, která se má zapsat. Operace pro atomickou inkrementaci je v OpenCL realizována funkcí `atomic_inc()` [62], která má pouze jeden parametr – ukazatel na místo v paměti, které se má inkrementovat.

Zmíněné úpravy jsou dále aplikovány i ve funkci `SpawnCars()`. Opět bylo přidáno klíčové slovo `kernel` a byly změněny názvy funkcí atomických operací.

7.4.6 Ukládání a načítání stavu simulace

Načítání stavu simulace z datového proudu je dále implementováno v metodě `InnerLoadFromStream()`. Nejprve je načten celkový počet buněk (32-bitová hodnota). Pomocí této hodnoty jsou vytvořena pole obsahující struktury `Cell`, `CellToCar` a `CellUi` a následně jsou načteny i všechny proměnné struktur ve stejném pořadí, jako jsou v daných strukturách definovány. Poté jsou takto načteny i všechny křižovatky (struktura `Junction`), generátory (struktura `Generator`) a vozidla (struktura `Car` a `CarUi`).

Ukládání stavu do datového proudu je dále implementováno v metodě `InnerSaveToStream()`. Nejprve je vždy zapsána velikost pole a následně jsou opět zapsány všechny proměnné daných struktur ve stejném pořadí.

7.5 Car following model

Třída `CarFollowingSim` obsahuje implementaci simulace silniční dopravy s využitím car following modelu. Třída je uložena ve jmenném prostoru `TrafficSimulation.Simulations.CarFollowing`.

Pro přesnější následné porovnání modelu s celulárním automatem s car following modelem bylo nejvhodnější implementovat podobné chování vozidel, jako bylo popsáno u modelu s celulárním automatem (viz kapitola 5.2.1). Délka vozidla u tohoto modelu může být také reprezentována reálným číslem, ale z důvodu porovnání bude reprezentována celým číslem. Rozdíl bude tedy čistě jen v reprezentaci jízdních pruhů a v použití reálných čísel u tohoto modelu.

U tohoto modelu není jízdní pruh rozdělen na buňky, ale jde o souvislý pás. Abychom zachovali délku jednotlivých pruhů stejnou jako u modelu s celulárním automatem, každý pruh bude mít délku určenou v jednotkách. Tato jednotka bude mít stejnou velikost jako velikost jedné buňky, jednotka se bude tedy rovnat 2,5 metru. Maximální rychlost v_{max} bude opět 54 km/h (resp. 6 jednotek za sekundu). Minimální délka vozidla je rovna 1 jednotce (2,5 metru) a maximální délka je rovna 6 jednotkám (15 metrů).

7.5.1 Struktura dopravní sítě

Tato třída využívá opět několik struktur pro uložení stavu simulace. Pro jednodušší orientaci v kódu používají shodná pojmenování jako u modelu s celulárním automatem. Jedná se tedy o struktury, které budou popsány v následujících podkapitolách.

Cell

Struktura `Cell` představuje celý jízdní pruh. Struktura obsahuje opět proměnné `T1`, `T2`, `T3`, `T4` a `T5`, které obsahují index pruhu, který následuje za současným pruhem. Ze současného pruhu tedy může vést opět až 5 rozdílných cest. Proměnné pro cesty, které nejsou využity, obsahují hodnotu -1. Ke každé cestě patří jedna z proměnných `P1`, `P2`, `P3`, `P4` nebo `P5`, které určují pravděpodobnost, že se vozidlo touto cestou vydá. Součet těchto proměnných se musí rovnat 1.

V případě, že se jedná o úsek pruhu, který je součástí křižovatky, proměnná `JunctionIndex` je rovna indexu dané křižovatky. Pokud není, je rovna hodnotě -1. Pokud tento úsek pruhu obsahuje terminátor, má proměnná `JunctionIndex` speciální hodnotu -2. Proměnná `NearestJunctionIndex` je index křižovatky, která je nejblíže v daném jízdním pruhu. Tato hodnota se využívá při zjišťování počtu vozidel, které čekají na dané křižovatce.

Dále obsahuje reálnou proměnnou `Length`, která určuje délku jízdního pruhu v dříve definovaných jednotkách. Navíc může mít délka i hodnotu 0. Tyto jízdní pruhy jsou pomocné a využívají se například v úsecích, které jsou označeny jako terminátory. Vozidlo, které se na tento úsek dostane, je pak ihned odstraněno ze silniční sítě, případně je přemístěno na novou pozici.

Dále je zde navíc proměnná `Lock`, která slouží k uzamykání jízdního pruhu. V případě, že nějaké vlákno nastaví její hodnotu na hodnotu 1, žádné jiné vlákno nemůže v daném jízdním pruhu provádět žádné změny.

Junction

Struktura `Junction` představuje opět jednotlivé křižovatky. Proměnná `CellIndex` je rovna indexu buňky, která danou křižovatku obsahuje. Proměnná `WaitingCount` pak obsahuje počet vozidel, která na dané křižovatce právě čekají. To jsou vozidla, která mají nulovou rychlost.

Generator

Struktura `Generator` obsahuje informace o jednotlivých generátorech vozidel. Ty jsou při generování silniční sítě umístěny po obvodu sítě. Mohou být však umístěny kdekoliv. Proměnná `CellIndex` je rovna indexu jízdního pruhu, na kterém se mají vozidla vytvářet. Proměnná `ProbabilityLambda` je rovna parametru λ exponenciálního rozdělení, které je použito při generování náhodných čísel. Proměnná `TimeLeft` obsahuje počet zbývajících časových kroků, než dojde k vygenerování nového vozidla. Po uplynutí této doby je generátor aktivován, je vytvořeno nové vozidlo a proměnná je nastavena na novou hodnotu, která je vygenerována exponenciálním rozdělením s parametrem λ.

Car

Struktura `Car` představuje jednotlivá vozidla. Proměnná `Position` je rovna indexu jízdního pruhu, na kterém se vozidlo právě nachází. Pokud se vozidlo na silniční síti nenachází, například bylo odstraněno terminátorem, je hodnota proměnné `Position` rovna -1. Reálná proměnná `PositionInCell` představuje konkrétní pozici v jízdním pruhu. Může nabývat hodnoty od 0 až do délky daného jízdního pruhu (proměnná `Length` struktury `Cell`). Označuje polohu přední části vozidla. Během jízdy v daném pruhu se proměnná postupně zvyšuje.

Proměnná `Speed` je rovna aktuální rychlosti vozidla. Ta je vyjádřena v počtu jednotek za časový krok (resp. za sekundu). Proměnná `Size` obsahuje délku vozidla, resp. počet jednotek, které dané vozidlo zabírá. Proměnná `AlreadyProcessed` označuje, zda bylo vozidlo v daném časovém kroku již plně zpracováno. Nabývá pak hodnoty 1, jinak nabývá hodnoty 0.

CellUi

Struktura `CellUi` obsahuje další proměnné pro jednotlivé jízdní pruhy, které jsou nutné pouze pro potřeby uživatelského rozhraní a nejsou potřebné pro výpočet simulace. Tyto informace jsou uložené zvlášť, ale vždy odpovídají příslušnému pruhu (viz struktura `Cell`) v poli pruhů. Tyto proměnné jsou souřadnice `X` a `Y` pro vykreslení pruhu na správném místě v uživatelském rozhraní.

CarUi

Struktura `CarUi` obsahuje další proměnné pro jednotlivá vozidla, které jsou nutné pouze pro potřeby uživatelského rozhraní a nejsou potřebné pro výpočet simulace. Tyto informace jsou opět uložené zvlášť, ale vždy odpovídají příslušnému vozidlu (viz struktura `Car`) v poli vozidel. Obsahuje proměnnou `Color`, která značí barvu vozidla, která je poté využita ve vizualizaci.

Uložení struktur

Všechny tyto struktury jsou použity v polích a všechny vazby mezi položkami v těchto polích jsou realizovány opět pomocí indexů, aby je bylo možné použít i pomocí GPU. Pole `CellsToCar` je nyní přímo definováno jako pole indexů vozidel o rozměrech (počet jízdních pruhů $*$ maximální počet vozidel v jízdním pruhu). Maximální počet vozidel v jednom jízdním pruhu je uložen zvlášť v proměnné `CarsPerCell`.

7.5.2 Generování silniční sítě

Třída `CarFollowingSim` také obsahuje generátor sítě křižovatek, který umožňuje vygenerovat síť o daném počtu křižovatek, délky jízdních pruhů, které mezi těmito křižovatkami jsou, počtu vozidel, které budou na této síti náhodně rozmístěny, maximálním počtu vozidel a pravděpodobnosti generování vozidel. Generátor u každé křižovatky vygeneruje čtyři výjezdy (na okrajových křižovatkách je tento počet nižší) a ke každé cestě je náhodně přiřazena pravděpodobnost, že se danou cestou vozidla vydají (proměnné `P1`, `P2`, `P3`, `P4` a `P5` ve struktuře `Cell`).

Generování cesty mezi křižovatkami je implementováno také v metodě `CreateLane()`. Na rozdíl od simulace s celulárním automatem vytváří pouze jeden silniční pruh, který je ještě poté napojen na křižovatku pomocí pomocného úseku o nulové délce (proměnná `Length` struktury `Cell`). Dále opět s použitím metody `CreateGeneratorsAndTerminators()` vygeneruje příslušné generátory a terminátory na

okraji sítě. U jízdního pruhu, který je ukončen terminátorem, je opět použit pomocný úsek o nulové délce.

Poté naplní síť zadaným počtem vozidel. Jejich pozici a délku vygeneruje náhodně tak, aby vozidla navzájem nekolidovala. Také je přidá do odpovídající části pole `CellsToCar`, která přísluší danému jízdnímu pruhu. Délka vozidla je vygenerována v rozsahu 1 až 6. Po vygenerování všech vozidel jsou vozidla v poli `CellsToCar` seřazena tak, aby vozidlo, které stojí nejdále v daném jízdním pruhu, bylo uloženo jako první. Proměnná `CarsPerCell`, která určuje maximální počet vozidel, který se do jízdního pruhu vejde, je při generování automaticky volena tak, aby pruh dokázal pojmout takový počet vozidel, který se do pruhu fyzicky vejde. Je tedy rovna délce jízdního pruhu, která je specifikována jako parametr. Vygenerování nové sítě je provedeno zavoláním metody `GenerateNew()`.

7.5.3 Referenční implementace

Dále obsahuje metodu `DoStepReference()`, která umožňuje vykonat jeden časový krok simulace pomocí referenční implementace s využitím jednotky CPU. Tato implementace je napsána opět přímo v jazyce C#. Metoda také využívá pro paralelní výpočty statickou metodu `Parallel.ForEach()` [52] a `Partitioner.Create()` [53]. Při výpočtu simulačního kroku je nejprve vynulován počet vozidel, která čekají na křižovatkách (proměnná `WaitingCount` ve struktuře `Junction`).

V této části se implementace liší od implementace pro model s celulárním automatem. Zpracování vozidel je u tohoto modelu rozděleno na dvě fáze. V první fázi je zavolána metoda `DoStepCarPre()`, která vždy zpracuje celý jeden jízdní pruh. Metoda má parametr představující index jízdního pruhu, který se má zpracovat. Nejprve projde všechna vozidla, která se nachází v daném jízdním pruhu, nastaví jejich proměnnou `AlreadyProcessed` na hodnotu 0 a přičte uraženou vzdálenost v daném jízdním pruhu (proměnná `PositionInCell`) podle aktuální rychlosti vozidla.

V případě, že vozidlo stojí, je nalezena nejbližší křižovatka pomocí proměnné `NearestJunctionIndex` buňky, na které vozidlo stojí, a u této křižovatky je inkrementován počet stojících vozidel o 1 pomocí atomické operace. Využívá se k tomu opět statická metoda `Interlocked.Increment()` [55].

Dále zavolá metodu `CarFollowing()`, která má jako parametr index vozidla, pro které je nutné přepočítat aktuální rychlost, a index vozidla, které jede před ním. V případě, že vozidlo jede jako první v daném jízdním pruhu, je hodnota nastavena na -1. V této metodě je nejprve zkontrolováno, jestli vozidlo jede jako první v daném jízdním pruhu. Pokud ano, cesta je volná a vozidlo může akcelerovat. Pokud ne, je nejprve spočítána mezera mezi vozidly. Pokud je mezera menší než rychlost vozidla, vozidlo musí zpomalit – rychlost vozidla je nastavena na velikost mezery. V opačném případě může vozidlo akcelerovat až na maximální povolenou rychlost. V každém časovém kroku může vozidlo zrychlit až o 1 jednotku. Pokud je vozidlo v pohybu, s pravděpodobností p, která je rovna hodnotě $0{,}2$, dojde ke snížení rychlosti vozidla o 1 jednotku.

Ve druhé fázi se zavolá metoda `DoStepCarPost()`, která se stará o přesouvání vozidel mezi jízdními pruhy. Metoda opět vždy zpracuje celý jeden jízdní pruh a je volána opakovaně v jednom časovém kroku simulace, dokud nejsou dokončeny všechny přesuny. Opět se prochází všechna vozidla, která zrovna v daném jízdním pruhu jedou. Následně je s využitím atomické operace `Exchange` nastavena hodnota proměnné `AlreadyProcessed` na hodnotu 1. Využívá se k tomu metoda `Interlocked.Exchange()` [63]. Pokud původní hodnota nebyla rovna 0, vozidlo bylo již jednou zpracováno a není potřeba ve výpočtu pokračovat. Pokud ještě zpracováno nebylo, je zkontrolováno, že vozidlo dojelo za hranici jízdního pruhu, ve kterém se zrovna nachází (porovnáním proměnné `PositionInCell` vozidla a proměnné `Length` jízdního pruhu). V tomto případě je nutné vozidlo přemístit do jízdního pruhu, který následuje. Pro nalezení tohoto pruhu se opět používá metoda `FindNextCell()`.

Následně je zkontrolováno, zda nenásleduje terminátor. V tomto případě je vozidlo odstraněno ze silniční sítě. Index vozidla je také odstraněn z pole `CellsToCar` a indexy dalších vozidel jsou posunuty tak, aby zaplnily vzniklou mezeru v poli. Během této operace je jízdní pruh uzamčen použitím metody `TryLockCell()` a následně opět odemčen zavoláním metody `UnlockCell()`. V případě, že byl jízdní pruh již uzamčen, je vozidlo označeno jako nezpracováno (opět pomocí atomické operace `Exchange`) a bude zpracováno znovu při dalším volání metody `DoStepCarPost()`. Tyto akce prování metoda `RemoveFromCellsToCar()`.

Pokud následuje standardní jízdní pruh, je tento jízdní pruh nejprve uzamčen využitím metody `TryLockCell()`. V případě, že byl jízdní pruh již uzamčen, je vozidlo opět označeno jako nezpracováno. V opačném případě je nejprve nalezeno první volné místo v části pole `CellsToCar` pro daný jízdní pruh. Na toto místo v poli je zapsán index vozidla, které je právě přesouváno. Následně je spočítána vzdálenost d, kterou vozidlo přesahuje minulý jízdní pruh. Pokud je následující jízdní pruh prázdný, vozidlo je posunuto do vzdálenosti d v tomto novém jízdním pruhu. Pokud prázdný není, je nejprve zkontrolováno, že je zde dostatečná mezera za poslední vozidlem, konkrétně, že mezera je větší nebo rovna vzdálenosti d. Pokud mezera dostatečná není, ale je větší než 0, je vozidlo také přesunuto, ale jeho rychlost je snížena tak, aby nedošlo ke kolizi. Pokud je mezera rovna 0, k přesunutí vůbec nedojde. Při přesouvání je opět využita metoda `RemoveFromCellsToCar()`. V případě, že je jeden z jízdních pruhů uzamčen, je opět přesun odložen do dalšího volání metody `DoStepCarPost()`.

Metody `TryLockCell()` a `UnlockCell()` využívají atomické operace `Exchange`. Využívá se k tomu opět statická metoda `Interlocked.Exchange()` [63]. Metoda `TryLockCell()` mění hodnotu proměnné `Lock` jízdního pruhu na 1 a kontroluje, zda byla původní hodnota rovna 0. Metoda `UnlockCell()` pouze mění hodnotu proměnné `Lock` zpět na původní hodnotu 0.

Následně jsou opět obdobně paralelně zpracovány všechny generátory zavoláním metody `SpawnCars()`. Nejprve je ověřeno, že proměnná `TimeLeft` generátoru je rovna 0. Pokud není, je proměnná dekrementována a výpočet je ukončen. Pokud je, generátor může být znovu aktivován. Nejprve je vypočítána nová hodnota proměnné `TimeLeft` pomocí exponenciálního rozdělení. Následně je vygenerováno náhodné číslo představující délku nového vozidla. Dále je nutné projít pole vozidel a nalézt v něm neobsazené místo. Využívá se zde opět metoda `Interlocked.CompareExchange()`.

Poté je nutné nalézt volné místo v části pole `CellsToCar` pro jízdní pruh, do kterého se vozidla generují. Na toto místo v poli je zapsán index vozidla, které je právě generováno. I zde je zkontrolováno, že je za posledním vozidlem dostatečně velká mezera pro přidání nového vozidla. Pokud jsou všechny podmínky splněny, je vozidlo přidáno do silniční sítě, jeho délka je nastavena na vygenerovanou hodnotu a jeho rychlost je nastavena na hodnotu 0.

7.5.4 Spouštění výpočtů pomocí technologie OpenCL

Třída dále implementuje metody `DoStepOpenCL()` a `DoBatchOpenCL()`. Obě metody také potřebují jako parametry instanci třídy `OpenCLDispatcher` a instanci třídy `OpenCLDevice`, která představuje zařízení, na kterém bude výpočet vykonán.

V případě metody `DoStepOpenCL()` je nejprve zkompilován připravený OpenCL program. Následně jsou získány ukazatele na jednotlivá pole struktur, která obsahují stav simulace. Poté jsou vytvořeny příslušné buffery a je spuštěn kernel `DoStepCarPre`, který přímo odpovídá metodě `DoStepCarPre()` referenční implementace. Následně je opakovaně spouštěn kernel `DoStepCarPost`, který odpovídá metodě `DoStepCarPost()`, dokud dochází k přesunu vozidel mezi jízdními pruhy. Pro ověření, že stále dochází k přesunu vozidel, je vytvořen speciální buffer `isChanged`, který obsahuje pouze jednu celočíselnou proměnnou. Nakonec je spuštěn i kernel `SpawnCars`, který opět odpovídá metodě `SpawnCars()` referenční implementace. Po dokončení těchto výpočtů je změněná paměť automaticky zkopírována do původního umístění a tím je simulace posunuta o jeden časový krok.

V případě metody `DoBatchOpenCL()`, která umožňuje provést libovolné množství časových kroků najednou, je opět zkompilován OpenCL program a jsou získány ukazatele na jednotlivá pole struktur. Následně jsou ručně vytvořeny jednotlivé buffery, které se využívají během celého výpočtu. Tyto buffery jsou přiřazeny jako parametry jednotlivých kernelů. Při opakovaném spuštění kernelu pak budou využity ty samé parametry. Poté jsou pomocí konstrukce `for` zavolány všechny potřebné kernely (`DoStepCarPre`, `DoStepCarPost` a `SpawnCars`). Navíc je použita metoda `MapAsWritable()` u bufferu `isChangedBuffer`, která umožňuje kontrolovat hodnotu proměnné `isChanged` i měnit její hodnotu, a tak opakovaně spouštět kernel `DoStepCarPost`, dokud nebudou přesunuta všechna vozidla. Tímto způsobem je proveden potřebný počet časových kroků. Nakonec je zavolána u použitých kernelů metoda `Finish()`. Všechny buffery jsou uvolněny automaticky, protože je použita konstrukce `using` [49].

7.5.5 Implementace s využitím technologie OpenCL

Zdrojový kód OpenCL je uložen v souboru `CarFollowingSim.cl` ve složce `Kernels`. Kód je do určité míry opět shodný s kódem referenční implementace,

protože kód referenční implementace byl psán tak, aby se následně co nejvíce shodoval s implementací pro OpenCL a neobsahovat nějaké speciální konstrukce, které nejsou v jazyce OpenCL C dostupné.

Místo pojmenovaných konstant opět využívá konstrukci `#define`. Dále obsahuje definici všech použitých struktur.

Po definici struktur následují jednotlivé funkce, které se využívají při výpočtu. Funkce `FindNextCell()`, `RemoveFromCellsToCar()` a `CarFollowing()` jsou prakticky shodné s odpovídajícími metodami referenční implementace. Místo referencí, které jsou součástí jazyka C#, jsou použity ukazatele.

Ve funkcích `DoStepCarPre()`, `DoStepCarPost()` a `SpawnCars()` je pro získání indexu, ke kterému se bude výpočet vztahovat, opět využita funkce `get_global_id()` [60] s parametrem 0. Navíc jsou označeny klíčovým slovem `kernel` [59], aby bylo možné spustit výpočet. Dále obsahují kontrolu, že daný index opravdu nepřesahuje hranice vstupního pole.

Opět byly využity příslušné atomické operace, které jsou dostupné v OpenCL. Operace `CompareExchange` je v OpenCL definována jako funkce `atomic_cmpxchg()` [61], ale s rozdílným pořadím parametrů. Operace pro atomickou inkrementaci je v OpenCL realizována funkcí `atomic_inc()` [62] a atomická operace `Exchange` je realizována funkcí `atomic_xchg()` [64].

7.5.6 Ukládání a načítání stavu simulace

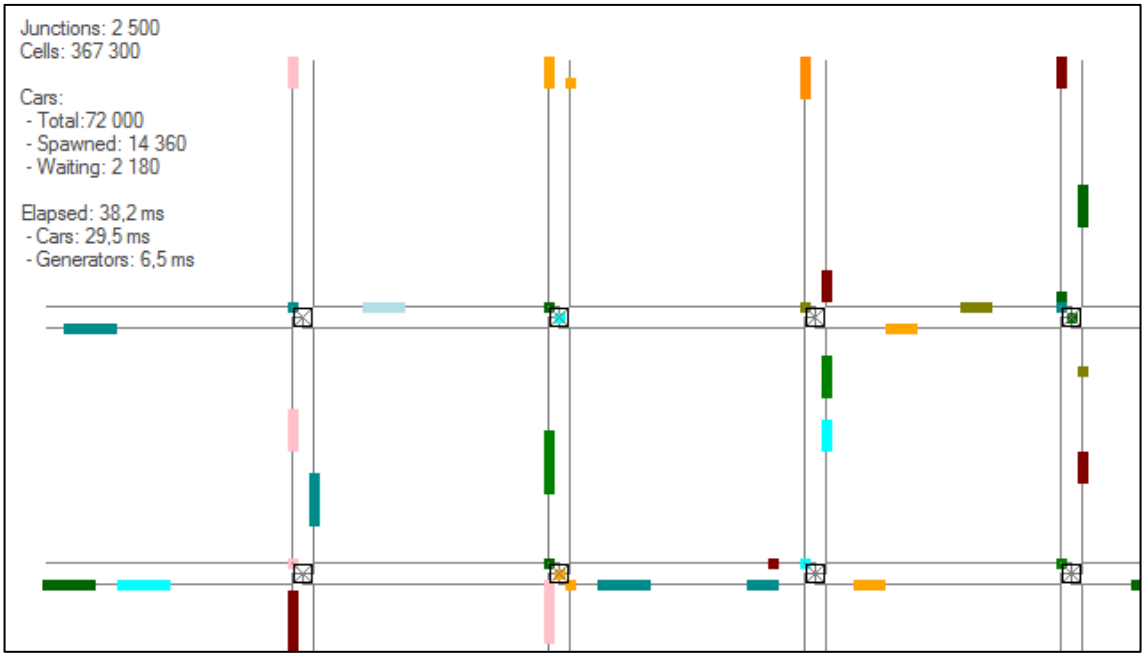
Načítání stavu simulace z datového proudu je dále implementováno v metodě `InnerLoadFromStream()`. Nejprve je načten celkový počet buněk (32-bitová hodnota). Pomocí této hodnoty jsou vytvořena pole obsahující struktury `Cell` a `CellUi` a následně jsou načteny i všechny proměnné struktur ve stejném pořadí, jako jsou v daných strukturách definovány. Následně je načtena hodnota proměnné `CarsPerCell`. S využitím této hodnoty může být vytvořeno pole `CellsToCar` a načteny indexy jednotlivých vozidel. Poté jsou načteny i všechny křižovatky (struktura `Junction`), generátory (struktura `Generator`) a vozidla (struktura `Car` a `CarUi`).

Ukládání stavu do datového proudu je dále implementováno v metodě `InnerSaveToStream()`. Nejprve je vždy zapsána velikost pole a následně jsou opět zapsány všechny proměnné daných struktur ve stejném pořadí.

7.6 Vizualizace simulace silniční dopravy

Vizualizace je implementována ve třídě `TrafficView`. Tato třída dědí od třídy `Control` [65]. Třída `Control` představuje základ všech prvků uživatelského rozhraní, které využívají knihovnu Windows Forms, jež je součástí rozhraní .NET Framework. Prvek lze pak snadno přidat do libovolného okna.

Abychom mohli vizualizaci vykreslit, je nutné překrýt metodu `OnPaint()`. Ta je automaticky volána knihovnou pokaždé, když je nutné prvek znovu překreslit. Voláním je předána instance třídy `Graphics` [66], která umožňuje kreslit na plátno. Tím je v tomto případě oblast určená pro tento prvek uživatelského rozhraní. Metodou `Invalidate()` [67] je možné překreslení vynutit.



Obrázek 9: Ukázka vizualizace simulace

Ve vizualizaci jsou postupně vykresleny všechny buňky (případně jízdní pruhy) a vozidla, která se na nich nachází. Buňky, které jsou aktuálně za hranicí plátna, vykresleny nejsou. Tato optimalizace byla zavedena pro zvýšení odezvy vizualizace

během posouvání pohledu. Vozidlům je vždy předělena jedna z deseti barev, aby je bylo možné od sebe odlišit. Při přiblížení vyšším než 130% je navíc kolem každé buňky vykreslen černý rámeček pro zvýraznění jednotlivých buněk. Na závěr je v rohu vizualizace vykreslen text, který informuje o stavu simulace. Ukázka vizualizace je zachycena na Obrázku 9.

Překrytím metod `OnMouseDown()`, `OnMouseUp()`, `OnMouseDoubleClick()`, `OnMouseMove()` a `OnMouseWheel()` je umožněno ovládání prvku pomocí myši. Po každé akci je vždy vynuceno překreslení prvku.

7.7 Implementace výkonnostních testů

Výkonnostní testy jsou implementovány ve zvláštním projektu `Benchmarks`. Pro spouštění testů je použita knihovna NBench [68], která je určena přesně k tomuto účelu. Knihovna sama provádí spouštění připravených testů a následně vygeneruje hlášení ve formátu Markdown [69].

Jednotlivé testy jsou vždy uloženy ve vlastní metodě. Tato metoda musí obsahovat atribut `PerfBenchmark`, podle kterého knihovna NBench pozná, že jde o test. Dále musí obsahovat atribut `TimingMeasurement`, aby knihovna poznala, že má být změřena doba běhu u daného testu. V jedné třídě může být definováno několik testů. Pomocí tohoto atributu lze k testu přiřadit název, který je potom použit v hlášení. Dále je možné specifikovat akci, která se provede před zahájením testu. Lze to udělat tak, že se k dané metodě přidá atribut `PerfSetup`. Pro specifikování akce, která se vykoná po skončení testu, je nutné přidat atribut `PerfCleanup`.

Takto byly postupně vytvořeny všechny testy pro oba typy modelů a všechny použité rozměry silniční sítě. Každá třída s testy obsahuje 2 testy – jeden testuje referenční implementaci a druhý testuje implementaci pro OpenCL. V obou případech se vždy provede zadaný počet kroků simulace. V metodě `Setup()`, která obsahuje atribut `PerfSetup`, je vždy vygenerována simulace s příslušnými parametry. V metodě `Cleanup()`, která obsahuje atribut `PerfCleanup`, jsou uvolněny zdroje a simulace je odstraněna z paměti.

Ve statické třídě `BenchmarkUtils` jsou všechny pomocné metody, které jsou využity v jednotlivých testech, protože testy by jinak z velké části obsahovaly shodný kód.

8 Testování

V této kapitole budou popsány jednotlivé druhy testů, které byly u popisované aplikace provedeny. Aplikace byla otestována jednotkovými testy a testy uživatelského rozhraní pro ověření správnosti aplikace. Následně byly provedeny i výkonnostní testy pro ověření rychlosti popsaných implementací.

8.1 Jednotkové testy

Během vývoje byly průběžně vytvářeny jednotkové testy pro jednotlivé třídy. Byla k tomu využita knihovna MSTest, která je integrovaná přímo do vývojového prostředí Microsoft Visual Studio. Toto vývojové prostředí též obsahuje nástroje pro analýzu pokrytí kódu.

Jednotkovými testy jsou z větší části pokryty všechny třídy kromě prvků uživatelského rozhraní. Celkem bylo vytvořeno 38 jednotkových testů.

8.1.1 Testování modelu s celulárním automatem

Třída `CellBasedSim` je pokryta 14 jednotkovými testy. Je zde otestována správnost generování simulace, je ověřeno, že simulace obsahuje příslušný počet buněk, křižovatek, generátorů a vozidel.

Dále je testována samotná simulace. Pro zaručení, že správně pracuje načítání i ukládání stavů, je otestována tato funkcionalita. Je načten připravený stav, následně je uložen a je zkontrolováno, že oba stavy obsahují stejná data. Konkrétně že oba soubory stavů jsou bitově shodné. Dále byly zařazeny i testy, které kontrolují jednotlivě některé dílčí funkce.

Jeden test kontroluje, že se jedno vozidlo správně pohybuje po síti se dvěma křižovatkami. Druhý test kontroluje, že se na síti s jednou křižovatkou správně generují vozidla na koncích sítě ve směru ke křižovatce, takže je vygenerována jedna křižovatka a následně je vygenerováno jedno vozidlo. Poslední test spočívá v kontrole terminátorů. Je vygenerována opět jedna křižovatka s jedním vozidlem a je otestováno, že když vozidlo dojede na hranici silniční sítě, je správně vyjmuto ze silniční sítě.

Všechny tyto testy byly napsány jak pro referenční implementaci, tak i pro implementaci pro OpenCL. Nakonec jsou zvlášť otestovány i další metody této třídy.

8.1.2 Testování car following modelu

Třída `CarFollowingSim` je pokryta 14 jednotkovými testy a byla otestována stejným způsobem, jako byl otestován druhý typ simulace. Opět je testování zajištěno porovnáním dvou stavů. I v tomto případě se testuje, jak referenční implementace, tak i implementace pro OpenCL, aby byla zaručena shodnost implementací. Zároveň byly otestovány obdobně i některé dílčí funkce simulace, konkrétně pohyb vozidla, generování vozidel a terminování vozidel.

Protože obě sady testují dva různé simulační modely, které však používají stejné rozhraní, obě sady testů by vypadaly velice podobně. Některé shodné části se tedy nachází v pomocné statické třídě `TestUtils` a jsou volány příslušnými jednotkovými testy.

8.1.3 Testování spuštění OpenCL

Dále byly zvlášť otestovány i třídy, které se starají o spouštění OpenCL programů. Ty byly pokryty 10 jednotkovými testy. Testují, zda jsou OpenCL programy správně zkompilovány a zda je možné je následně spustit. Také kontrolují, zda výsledné hodnoty z jednoduchého testovacího OpenCL programu odpovídají předpokládaným hodnotám.

8.2 Testy uživatelského rozhraní

Uživatelské rozhraní bylo otestováno ručně tak, že bylo provedeno několik testovacích scénářů. Těmito testovacími scénáři je zcela otestováno uživatelské rozhraní aplikace.

8.2.1 První testovací scénář

První scénář spočíval v tom, že byla vygenerována nová simulace modelu s celulárním automatem. Bylo otestováno, že daná simulace obsahuje příslušný počet

křižovatek a vozidel. Dále bylo provedeno několik kroků simulace. Bylo ověřeno, že simulace dané kroky opravdu provedla.

8.2.2 Druhý testovací scénář

Druhý scénář byl totožný, akorát byl vybrán car following model. Opět byl otestován počet křižovatek a vozidel, bylo provedeno několik kroků a ověřeno jejich provedení.

Oba testovací scénáře byly provedeny nejprve s vybráním referenční implementace. Poté byly provedeny znovu s vybráním zařízení OpenCL pro vykonávání algoritmu, takže byly pokryty všechny kombinace.

8.2.3 Třetí testovací scénář

Poslední testovací scénář spočíval v ověření funkčnosti ukládání a načítání existující simulace. Byla vygenerována nová simulace. Bylo provedeno několik kroků simulace a následně byla uložena do souboru. Aplikace byla poté zavřena a znovu otevřena. Následně byl načten uložený soubor a bylo ověřeno, že simulace je ve stejném stavu jako před jejím uložením.

8.3 Výkonnostní testy

Aplikace byla otestována na několika počítačích. Byly testovány různé parametry simulace. Zároveň bylo testování postupně provedeno s využitím obou dostupných modelů – modelu s celulárním automatem a car following modelu.

V měření je porovnávána doba běhu referenční implementace, která využívá jednotku CPU, s dobou běhu verze pro OpenCL spuštěné na příslušném zařízení, tedy na grafické kartě. Vždy bylo provedeno 10 kroků (tzn. 10 sekund času v simulaci) před začátkem měření. To bylo zvoleno proto, aby se všechna vozidla stihla rozjet a simulace tedy simulovala nejvěrněji reálné prostředí. Poté byla změřena doba běhu provedení 600 kroků simulace. Během tohoto počtu kroků vozidla dokážou urazit dostatečně dlouhou vzdálenost pro otestování všech funkcí simulace. Pokud je cesta volná, vozidlo jedoucí maximální rychlostí stihne během této doby projet až 150 křižovatkami.

Maximální počet vozidel byl vždy volen jako šestinásobek počátečního počtu. Ve všech případech to bylo o něco více vozidel, než kolik se na danou síť křižovatek může maximálně vejít. Také byly hodnoty zvoleny tak, aby vozidla pokrývala procentuálně stejnou plochu silniční sítě.

Každý test byl zopakován desetkrát, aby byl minimalizován vliv vnějšího prostředí.

8.3.1 Popis testovaných konfigurací

Testování bylo provedeno na třech různých počítačích s rozdílnými parametry. U všech konfigurací byla vybrána jako zařízení OpenCL zmíněná grafická karta.

První počítač (konfigurace č. 1), na kterém bylo provedeno testování, obsahoval procesor AMD FX-6300 (6 jader; 6 vláken; 4,1 GHz), grafickou kartu AMD Radeon HD 7870 (s 1280 steam procesory s frekvencí 1050 MHz; s pamětí GDDR5 o velikosti 2GB s šířkou sběrnice 256 bitů a přenosovou rychlostí 153,6 GB/s), 8GB RAM (DDR3 s frekvencí 1866 MHz; dual-channel) a SSD.

Druhý počítač (konfigurace č. 2) obsahoval procesor Intel Core i7-8750H (6 jader; 12 vláken; 2,2 GHz), grafickou kartu Nvidia GeForce GTX 1060 (s 1280 CUDA jádry s frekvencí 1506 MHz; s pamětí GDDR5 o velikosti 6GB s šířkou sběrnice 192 bitů a přenosovou rychlostí 192 GB/s), 16GB RAM (DDR4 s frekvencí 2666 MHz; dual-channel) a SSD.

Třetí počítač (konfigurace č. 3) obsahoval procesor Intel Core i5-3230M (2 jádra; 4 vlákna; 2,6 GHz), grafickou kartu Nvidia GeForce GT 740M (s 384 CUDA jádry s frekvencí 980 MHz; s pamětí DDR3 o velikosti 2GB s šířkou sběrnice 64 bitů a přenosovou rychlostí 14,4 GB/s), 8GB RAM (DDR3 s frekvencí 1600 MHz; dual-channel) a SSD.

8.3.2 Popis testovaných parametrů simulace

Testování bylo provedeno na čtyřech různých sadách parametrů nastavení simulace.

Každá sada byla použita pro model s celulárním automatem i pro car following model. Každá také byla použita pro simulaci pomocí referenční implementace i pro

simulace pomocí implementace pro OpenCL. Celkově tedy jde o 16 testů pro každou testovanou konfiguraci (viz kapitola 8.3.1). Dohromady bylo tedy provedeno 48 výkonnostních testů. Nyní budou popsány jednotlivé sady parametrů, které byly při testování použity.

Testování bylo nejprve provedeno na simulaci malého rozsahu. Byly zadány následující parametry:

- Počet křižovatek: 10 x 10 (100 celkem)
- Vzdálenost mezi křižovatkami: 24
- Počáteční počet vozidel: 500
- Maximální počet vozidel: 3 000
- Parametr λ pravděpodobnosti pro generování nových vozidel: 0,003

Následně bylo provedeno testování na simulaci středního rozsahu. Byly zadány následující parametry:

- Počet křižovatek: 50 x 50 (2 500 celkem)
- Vzdálenost mezi křižovatkami: 24
- Počáteční počet vozidel: 12 000
- Maximální počet vozidel: 72 000
- Parametr λ pravděpodobnosti pro generování nových vozidel: 0,005

Poté bylo provedeno na simulaci velkého rozsahu. Byly zadány následující parametry:

- Počet křižovatek: 100 x 100 (10 000 celkem)
- Vzdálenost mezi křižovatkami: 24
- Počáteční počet vozidel: 50 000
- Maximální počet vozidel: 300 000
- Parametr λ pravděpodobnosti pro generování nových vozidel: 0,01

Nakonec na simulaci velmi velkého rozsahu. Byly zadány následující parametry pro vytvoření simulace:

- Počet křižovatek: 220 x 220 (48 400 celkem)
- Vzdálenost mezi křižovatkami: 24

- Počáteční počet vozidel: 250 000
- Maximální počet vozidel: 1 500 000
- Parametr λ pravděpodobnosti pro generování nových vozidel: 0,01

8.3.3 Výsledky testů pro model s celulárním automatem

V této kapitole budou popsány výsledky výkonnostních testů pro model s celulárním automatem. Tyto testy byly popsány v předchozích kapitolách.

Simulace malého rozsahu

Nejprve budou srovnány výsledky simulace malého rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 1. Souhrnné informace jsou pak v Tabulce 2.

Tabulka 1: Jednotlivá měření simulace malého rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	36	41	37	35	37	39	36	35	37	35
OpenCL	49	49	48	48	48	48	49	47	46	46

Tabulka 2: Souhrnné informace o měření simulace malého rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	36,8	35,0	41,0	1,9
OpenCL	47,8	46,0	49,0	1,1

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 3. Souhrnné informace z měření se pak nalézají v Tabulce 4.

Tabulka 3: Jednotlivá měření simulace malého rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	17	17	17	17	16	16	17	17	17	17
OpenCL	25	23	24	24	23	23	23	24	23	23

Tabulka 4: Souhrnné informace o měření simulace malého rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	16,8	16,0	17,0	0,4
OpenCL	23,5	23,0	25,0	0,7

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 5. Souhrnné informace jsou v Tabulce 6.

Tabulka 5: Jednotlivá měření simulace malého rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	21	21	20	20	41	23	22	22	23	23
OpenCL	34	34	33	34	33	34	33	34	34	34

Tabulka 6: Souhrnné informace o měření simulace malého rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	23,6	20,0	41,0	6,2
OpenCL	33,7	33,0	34,0	0,5

Z výsledku lze vidět, že pro simulaci malého rozsahu je referenční implementace zhruba 1,3krát rychlejší než implementace pro OpenCL.

Simulace středního rozsahu

Dále budou srovnány výsledky simulace středního rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 7. Souhrnné informace se nalézají v Tabulce 8.

Tabulka 7: Jednotlivá měření simulace středního rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	810	795	842	812	823	830	800	807	825	835
OpenCL	400	400	403	405	404	399	399	404	400	410

Tabulka 8: Souhrnné informace o měření simulace středního rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	817,9	795,0	842,0	15,5
OpenCL	402,4	399,0	410,0	3,5

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 9. Souhrnné informace jsou v Tabulce 10.

Tabulka 9: Jednotlivá měření simulace středního rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	328	324	327	332	340	326	329	325	326	328
OpenCL	61	60	62	61	63	61	62	61	60	63

Tabulka 10: Souhrnné informace o měření simulace středního rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	328,5	324,0	340,0	4,6
OpenCL	64,4	60,0	63,0	1,1

Výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 11. Souhrnné informace jsou v Tabulce 12.

Tabulka 11: Jednotlivá měření simulace středního rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	542	546	535	528	551	520	530	536	522	531
OpenCL	278	279	280	277	280	280	279	278	279	280

Tabulka 12: Souhrnné informace o měření simulace středního rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	534,1	520,0	551,0	10,0
OpenCL	279,0	277,0	280,0	1,0

Z těchto výsledku lze vidět, že pro simulaci středního rozsahu je naopak implementace pro OpenCL na některých konfiguracích až 5krát rychlejší oproti referenční implementaci.

Simulace velkého rozsahu

Následují výsledky simulace velkého rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 13. Souhrnné informace jsou pak v Tabulce 14.

Tabulka 13: Jednotlivá měření simulace velkého rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	4322	4292	3898	4412	4181	4462	4090	4219	4276	4623
OpenCL	1852	1867	1877	1885	1886	1853	1875	1871	1895	1871

Tabulka 14: Souhrnné informace o měření simulace velkého rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	4277,5	3898,0	4623,0	201,5
OpenCL	1873,2	1852,0	1895,0	13,8

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 15. Souhrnné informace se pak nalézají v Tabulce 16.

Tabulka 15: Jednotlivá měření simulace velkého rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	2023	1996	1999	2062	2073	1996	2008	1995	1992	1996
OpenCL	207	204	203	202	203	204	203	203	203	206

Tabulka 16: Souhrnné informace o měření simulace velkého rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	2014,2	1992,0	2073,0	29,7
OpenCL	203,8	202,0	207,0	1,6

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 17. Souhrnné informace jsou pak v Tabulce 18.

Tabulka 17: Jednotlivá měření simulace velkého rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	3150	3160	3148	3133	3129	3129	3158	3165	3103	3149
OpenCL	1317	1306	1323	1327	1316	1320	1327	1320	1334	1315

Tabulka 18: Souhrnné informace o měření simulace velkého rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	3142,4	3103,0	3165,0	18,9
OpenCL	1320,5	1306,0	1334,0	7,8

I zde lze vidět, že pro simulaci velkého rozsahu je implementace pro OpenCL na konfiguraci č. 2 dokonce až 10krát rychlejší oproti referenční implementaci.

Simulace velmi velkého rozsahu

Poslední sada měření patří simulaci velmi velkého rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 19. Souhrnné informace z měření jsou pak v Tabulce 20.

Tabulka 19: Jednotlivá měření simulace velmi velkého rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	19814	21881	22694	22106	21879	21537	19713	20793	20180	20315
OpenCL	8799	8764	8813	8877	8846	8812	8868	8782	8858	8774

Tabulka 20: Souhrnné informace o měření simulace velmi velkého rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	21091,2	19713,0	22694,0	1059,1
OpenCL	8819,3	8764,0	8877,0	40,7

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 21. Souhrnné informace jsou pak v Tabulce 22.

Tabulka 21: Jednotlivá měření simulace velmi velkého rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	11018	11118	11169	11131	11245	11223	11128	11136	11201	11264
OpenCL	901	903	898	903	896	903	904	905	903	902

Tabulka 22: Souhrnné informace o měření simulace velmi velkého rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	11163,3	11018,0	11264,0	73,1
OpenCL	901,8	896,0	905,0	2,8

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 23. Souhrnné informace se nalézají v Tabulce 24.

Tabulka 23: Jednotlivá měření simulace velmi velkého rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	16284	16341	16317	16316	16293	16354	16287	16341	16294	16301
OpenCL	6533	6517	6539	6530	6536	6572	6550	6552	6541	6538

Tabulka 24: Souhrnné informace o měření simulace velmi velkého rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	16312,8	16284,0	16354,0	25,2
OpenCL	6540,8	6517,0	6572,0	14,8

U simulace velmi velkého rozsahu je implementace pro OpenCL také rychlejší než referenční implementace. Na konfiguraci č. 1 je 2,4krát rychlejší oproti referenční implementaci, na konfiguraci č. 2 je až 12krát rychlejší a na konfiguraci č. 3 je 2,5krát rychlejší.

8.3.4 Výsledky testů pro car following model

V této kapitole budou popsány výsledky výkonnostních testů pro car following model.

Simulace malého rozsahu

Nejprve budou opět srovnány výsledky simulace malého rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 25. Souhrnné informace jsou pak v Tabulce 26.

Tabulka 25: Jednotlivá měření simulace malého rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	38	38	38	39	38	38	39	39	39	38
OpenCL	214	212	213	215	215	213	212	214	212	213

Tabulka 26: Souhrnné informace o měření simulace malého rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	38,4	38,0	39,0	0,5
OpenCL	213,3	212,0	215,0	1,1

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 27. Souhrnné informace se nalézají v Tabulce 28.

Tabulka 27: Jednotlivá měření simulace malého rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	21	21	22	21	21	22	22	21	21	22
OpenCL	241	245	246	246	243	244	245	245	249	245

Tabulka 28: Souhrnné informace o měření simulace malého rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	21,4	21,0	22,0	0,5
OpenCL	244,9	241,0	249,0	2,1

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 29. Souhrnné informace se nalézají v Tabulce 30.

Tabulka 29: Jednotlivá měření simulace malého rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	17	16	17	16	16	17	17	17	18	16
OpenCL	23	23	24	23	24	23	24	24	24	24

Tabulka 30: Souhrnné informace o měření simulace malého rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	16,7	16,0	18,0	0,7
OpenCL	23,6	23,0	24,0	0,5

Z výsledku lze vidět, že pro simulaci malého rozsahu je referenční implementace zhruba 1,4krát rychlejší než implementace pro OpenCL. To se podobá výsledkům u testování modelu s celulárním automatem. V některých případech je rozdíl však daleko větší.

Simulace středního rozsahu

Dále budou srovnány výsledky simulace středního rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 31. Souhrnné informace se nalézají v Tabulce 32.

Tabulka 31: Jednotlivá měření simulace středního rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	749	748	746	745	745	747	743	746	744	743
OpenCL	1831	1834	1828	1836	1833	1826	1824	1837	1839	1834

Tabulka 32: Souhrnné informace o měření simulace středního rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	745,6	743,0	749,0	2,0
OpenCL	1832,2	1824,0	1839,0	4,9

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 33. Souhrnné informace z měření jsou v Tabulce 34.

Tabulka 33: Jednotlivá měření simulace středního rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	310	319	314	313	322	315	316	316	314	314
OpenCL	348	350	345	347	353	352	351	352	351	350

Tabulka 34: Souhrnné informace o měření simulace středního rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	315,3	310,0	322,0	3,3
OpenCL	349,9	345,0	353,0	2,5

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 35. Souhrnné informace jsou v Tabulce 36.

Tabulka 35: Jednotlivá měření simulace středního rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	649	642	637	643	637	642	639	638	636	635
OpenCL	767	770	766	766	765	764	770	769	768	770

Tabulka 36: Souhrnné informace o měření simulace středního rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	639,8	635,0	649,0	4,2
OpenCL	767,5	764,0	770,0	2,2

I zde je referenční implementace rychlejší než implementace pro OpenCL. Rozdíl však není tak velký. U konfigurace č. 2 je referenční implementace pouze 1,1krát rychlejší.

Simulace velkého rozsahu

Následují výsledky simulace velkého rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 37. Souhrnné informace jsou pak v Tabulce 38.

Tabulka 37: Jednotlivá měření simulace velkého rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	4527	4506	4509	4507	4507	4503	4514	4512	4505	4507
OpenCL	8065	8045	8030	8108	8071	8014	8045	8083	8072	8054

Tabulka 38: Souhrnné informace o měření simulace velkého rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	4509,7	4503,0	4527,0	6,9
OpenCL	8058,7	8014,0	8108,0	27,1

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 39. Souhrnné informace jsou pak v Tabulce 40.

Tabulka 39: Jednotlivá měření simulace velkého rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	2605	2354	2325	2369	2349	2358	2352	2339	2354	2341
OpenCL	577	569	570	569	565	568	578	571	563	561

Tabulka 40: Souhrnné informace o měření simulace velkého rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	2374,7	2325,0	2605,0	81,8
OpenCL	569,1	561,0	578,0	5,5

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 41. Souhrnné informace se nalézají v Tabulce 42.

Tabulka 41: Jednotlivá měření simulace velkého rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	3502	3498	3502	3504	3507	3501	3501	3504	3502	3495
OpenCL	2245	2246	2243	2243	2238	2253	2248	2239	2243	2243

Tabulka 42: Souhrnné informace o měření simulace velkého rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	3501,6	3495,0	3507,0	3,3
OpenCL	2244,1	2238,0	2253,0	4,3

S větším množstvím dat naopak začíná být rychlejší implementace pro OpenCL. Pro simulaci velkého rozsahu je implementace pro OpenCL na konfiguraci č. 2 až 4krát rychlejší oproti referenční implementaci.

Simulace velmi velikého rozsahu

Poslední sada měření patří simulaci velmi velkého rozsahu. Výsledky jednotlivých měření při použití konfigurace č. 1 pro referenční implementaci a implementaci pro OpenCL jsou zaznamenány v Tabulce 43. Souhrnné informace z měření jsou v Tabulce 44.

Tabulka 43: Jednotlivá měření simulace velmi velkého rozsahu pro konfiguraci č. 1

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	25839	25936	25953	25932	25841	25867	25869	25815	25949	25963
OpenCL	35994	36356	36164	36088	36126	36168	36202	36210	36162	36197

Tabulka 44: Souhrnné informace o měření simulace velmi velkého rozsahu pro konfiguraci č. 1

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	25896,4	25815,0	25963,0	55,6
OpenCL	36166,7	35994,0	36356,0	92,8

Výsledky jednotlivých měření při použití konfigurace č. 2 jsou zaznamenány v Tabulce 45. Souhrnné informace jsou pak v Tabulce 46.

Tabulka 45: Jednotlivá měření simulace velmi velkého rozsahu pro konfiguraci č. 2

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	13977	13483	13390	13271	13276	13152	13152	13141	13267	13140
OpenCL	1248	1243	1246	1250	1241	1246	1251	1261	1253	1253

Tabulka 46: Souhrnné informace o měření simulace velmi velkého rozsahu pro konfiguraci č. 2

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	13324,9	13140,0	13977,0	256,2
OpenCL	1249,2	1241,0	1261,0	5,8

A nakonec výsledky jednotlivých měření při použití konfigurace č. 3 jsou zaznamenány v Tabulce 47. Souhrnné informace jsou v Tabulce 48.

Tabulka 47: Jednotlivá měření simulace velmi velkého rozsahu pro konfiguraci č. 3

	Časy jednotlivých měření [ms]									
Měření č.	1	2	3	4	5	6	7	8	9	10
Reference	17767	17772	17777	17807	17821	17778	17773	17777	17780	17772
OpenCL	9430	9447	9433	9469	9466	9476	9459	9478	9452	9468

Tabulka 48: Souhrnné informace o měření simulace velmi velkého rozsahu pro konfiguraci č. 3

	Průměr [ms]	Minimum [ms]	Maximum [ms]	Odchylka [ms]
Reference	17782,4	17767,0	17821,0	17,4
OpenCL	9457,8	9430,0	9478,0	16,9

U simulace velmi velkého rozsahu je implementace pro OpenCL také rychlejší než referenční implementace. Na konfiguraci č. 2 je opět 11krát rychlejší. Pouze na konfiguraci č. 1 je 1,4krát pomalejší.

8.3.5 Zhodnocení výsledků

Průměrné doby běhu na jednotlivých konfiguracích pro model s celulárním automatem a car following model byly srovnány v Tabulce 49 a Tabulce 50.

Tabulka 49: Průměrné doby běhu na jednotlivých konfiguracích pro malou a střední síť

		Průměrná doba běhu [ms]			
	Velikost	Malá		Střední	
Konfigurace	Implementace	Cel. aut.	Car follow.	Cel. aut.	Car follow.
1	Reference	36,8	38,4	817,9	745,6
	OpenCL	47,8	213,3	402,4	1832,2
2	Reference	16,8	21,4	328,5	315,3
	OpenCL	23,5	244,9	64,4	349,9
3	Reference	23,6	16,7	534,1	639,8
	OpenCL	33,7	23,6	279,0	767,5

Tabulka 50: Průměrné doby běhu na jednotlivých konfiguracích pro velkou a velmi velkou síť

		Průměrná doba běhu [ms]			
	Velikost	Velká		Velmi velká	
Konfigurace	Implementace	Cel. aut.	Car follow.	Cel. aut.	Car follow.
1	Reference	4277,5	4509,7	21091,2	25896,4
	OpenCL	1873,2	8058,7	8819,3	36166,7
2	Reference	2014,2	2374,7	11163,3	13324,9
	OpenCL	203,8	569,1	901,8	1249,2
3	Reference	3142,4	3501,6	16312,8	17782,4
	OpenCL	1320,5	2244,1	6540,8	9457,8

Z naměřených hodnot je patrné, že ve většině případů byla rychlejší implementace pro OpenCL, v některých případech dokonce až zhruba desetkrát. Pouze u malé silniční sítě je rychlejší výpočet přes CPU. Tento rozdíl je pravděpodobně dán tím, že před zahájením výpočtů je nutné jednotku GPU připravit na zahájení výpočtu, zkopírovat potřebná data z RAM do paměti GPU a po skončení výpočtu zkopírovat nově vypočtená data zpět do RAM. Z naměřených hodnot lze dále usoudit, že čím větší silniční síť je, tím je implementace pro OpenCL efektivnější.

Dále můžeme vidět, že model s celulárním automatem je o několik procent rychlejší než car following model. Tento rozdíl je pravděpodobně dán tím, že přesouvání vozidla u car following modelu je výpočetně náročnější než u modelu

s celulárním automatem hlavně z toho důvodu, že je nutné provádět výpočet opakovaně, než dojde k přesunu všech vozidel, a kontrolovat při tom, zda ještě stále čekají některá vozidla na přesun.

U testované konfigurace č.1 s grafickou kartou AMD Radeon HD 7870 je navíc pozorován nižší výkon u simulace s car following modelem než při použití testovaných grafických karet od společnosti Nvidia. To je pravděpodobně způsobeno odlišností architektur jednotlivých jednotek GPU, případně i rozdílnými implementacemi ovladačů OpenCL, které poskytují výrobci daných jednotek GPU. Pravděpodobně by však bylo možné dále implementaci optimalizovat i pro tuto grafickou kartu.

9 Závěr

V rámci této diplomové práce byla vytvořena aplikace obsahující dva mikroskopické silniční modely – model s celulárním automatem a model založený na následování vozidel (car following model). Simulaci s využitím obou modelů je možné provádět jak na GPU tak na CPU, což umožňuje porovnání času potřebného k doběhnutí simulace. Z tohoto důvodu je kladen důraz na podobnost implementace pro GPU a implementace pro CPU. Pro pozorování průběhu simulace byla implementována i jednoduchá vizualizace.

Dále bylo provedeno srovnání výkonu obou simulačních modelů. Rychlosti jednotlivých implementací pro oba modely byly otestovány na třech různých počítačových konfiguracích. Při měření doby běhu bylo zjištěno, že ve většině případů je rychlejší implementace pro OpenCL s využitím jednotky GPU. V některých případech byla jednotka GPU až desetkrát rychlejší než při použití CPU. Navíc bylo patrné, že efektivita implementace pro OpenCL roste s velikostí silniční sítě.

Seznam zkratek

API	Application Programming Interface
CIL	Common Intermediate Language
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
GDDR5	Graphic Double Data Rate 5
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
JIT	Just-in-Time
RAM	Random Access Memory
SDK	Software Development Kit
SIMT	Single Instruction, Multiple Threads
SSD	Solid-state Drive
SVM	Shared Virtual Memory

Literatura

[1] Kreinin, Yossi. SIMD < SIMT < SMT: parallelism in NVIDIA GPUs. *Proper Fixation.* [Online] 10. listopadu 2011. [Citace: 10. ledna 2019.] https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html.

[2] Case, Loyd. GeForce 8800 GTX: 3D Architecture Overview. *ExtremeTech.* [Online] 8. listopadu 2008. [Citace: 20. února 2019.] https://www.extremetech.com/computing/73206-geforce-8800-gtx-3d-architecture-overview/2.

[3] The Evolution of Computer Graphics. *Nvidia.* [Online] 2008. [Citace: 1. března 2019.] https://www.nvidia.com/content/nvision2008/tech_presentations/Technology_Keynotes/NVISION08-Tech_Keynote-GPU.pdf.

[4] Kashkovsky, Alexander V., Shershnev, Anton A. a Vashchenkov, Pavel V. Aspects of GPU perfomance in algorithms. [Online] 2017. [Citace: 4. února 2019.] https://aip.scitation.org/doi/pdf/10.1063/1.5007505.

[5] Ryoo, Shane. Why don't GPUs have branch predictors? *Quora.* [Online] 25. června 2017. [Citace: 6. ledna 2019.] https://www.quora.com/Why-dont-GPUs-have-branch-predictors.

[6] CUDA Zone. *Nvidia Accelerated Computing.* [Online] [Citace: 19. ledna 2019.] https://developer.nvidia.com/cuda-zone.

[7] CUDA Toolkit 10.1 Download. *Nvidia Accelerated Computing.* [Online] [Citace: 19. ledna 2019.] https://developer.nvidia.com/cuda-downloads.

[8] OpenCL Overview. [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/opencl/.

[9] AMD Drivers and Support. *AMD.* [Online] [Citace: 20. ledna 2019.] https://www.amd.com/en/support.

[10] OpenCL™ Runtimes for Intel® Processors. *Intel.* [Online] 30. října 2018. [Citace: 20. ledna 2019.] https://software.intel.com/en-us/articles/opencl-drivers.

[11] OpenCL. *Nvidia Accelerated Computing.* [Online] [Citace: 20. ledna 2019.] https://developer.nvidia.com/opencl.

[12] OpenCL on the GPU. *Nvidia.* [Online] 30. září 2009. [Citace: 20. února 2019.] https://www.nvidia.com/content/GTC/documents/1409_GTC09.pdf.

[13] OpenCL™ 2.0 Shared Virtual Memory Overview. *Intel Software Developer Zone.* [Online] 10. září 2014. [Citace: 20. ledna 2019.] https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-

overview.

[14] OpenCL: History & Future. *Forschungszentrum Jülich.* [Online] [Citace: 10. ledna 2019.] https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/opencl/opencl-10-history-future.pdf?__blob=publicationFile.

[15] C++ AMP Overview. *Microsoft Docs.* [Online] 18. listopadu 2018. [Citace: 20. února 2019.] https://docs.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-overview?view=vs-2017.

[16] C++ Single-source Heterogeneous Programming for OpenCL. [Online] [Citace: 2. března 2019.] https://www.khronos.org/sycl/.

[17] Winsberg, Eric. Computer Simulations in Science. *The Stanford Encyclopedia of Philosophy (Spring 2019 Edition).* [Online] 6. května 2013. [Citace: 22. ledna 2019.] https://plato.stanford.edu/entries/simulations-science/.

[18] Fujimoto, Richard M. *Parallel and Distributed Simulation Systems.* New York : John Wiley & Sons, Inc., 2000. ISBN: 0-471-18383-0.

[19] Potužák, Tomáš. Current Trends in Distributed Traffic Simulation. *Proceedings of 41th Spring International Conference MOSIS '07 – Modelling and Simulation of Systems.* 2007.

[20] Nökel, Klaus a Schmidt, Matthias. Parallel DYNEMO: Meso-Scopic Traffic Flow Simulation on Large Networks. *Networks and Spatial Economics.* 2002.

[21] Savrasovs, Mihails. Traffic Flow Simulation Using Mesoscopic Approach. *Transport ant Telecomunication Institute.* [Online] [Citace: 4. února 2019.] http://ttlog.civ.uth.gr/wp-content/uploads/2017/11/UTH_22112017.pdf.

[22] Schreckenberg, M., Neubert, L. a Wahle, J. Simulation of traffic in large road networks. *Future Generation Computer Systems.* 2001.

[23] Wagner, Peter a Lubashevsky, Ihor. Empirical basis for car-following theory development. *eprint arXiv:cond-mat/0311192.* 2003.

[24] Potužák, Tomáš. Distributed Traffic Simulation. *State of the Art and Future Research.* Technical Report No. DCSE/TR-2008-02, 2008.

[25] Nagel, Kai a Schreckenberg, Michael. A Cellular Automaton Model for Freeway. *Journal de Physique I.* 1992.

[26] Hartman, David. Head Leading Algorithm for Urban Traffic Model. *In Proceedings of the 16th International European Simulation Symposium and Exhibition ESS.* 2004.

[27] Chandler, Robert E., Herman, Robert a Montroll, Elliott W. Traffic Dynamics: Studies in Car Following. *Operations Research 6, no. 2.* 1958.

[28] Potužák, Tomáš. Methods for Reduction of Interprocess Communication in Distributed Simulation of Road Traffic. 2009.

[29] Panwai, Sakda a Dia, Hussein. Comparative Evaluation of Microscopic Car-Following. *IEEE Transaction on Intelligent Transportation Systems.* 2005.

[30] Helly, W. Simulation of bottlenecks in single lane traffic flow. *Proceedings of the Symposium on Traffic Flow Theory.* 1959.

[31] Introduction to the C# Language and the .NET Framework. *Microsoft Docs.* [Online] 20. července 2015. [Citace: 7. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.

[32] Overview of the .NET Framework. *Microsoft Docs.* [Online] 30. března 2017. [Citace: 2. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview.

[33] C# Language Specification 5.0. [Online] 7. června 2013. [Citace: 10. prosince 2018.] https://www.microsoft.com/en-us/download/details.aspx?id=7029.

[34] Casting and Type Conversions (C# Programming Guide). *Microsoft Docs.* [Online] 20. července 2015. [Citace: 10. února 2019.] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions.

[35] Unsafe Code and Pointers (C# Programming Guide). *Microsoft Docs.* [Online] 20. července 2015. [Citace: 7. prosince 2018.] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/unsafe-code-pointers/index.

[36] Chapter 1: Introducing C#. *Microsoft Docs.* [Online] 7. června 2010. [Citace: 12. prosince 2018.] https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/hh145616(v=vs.88).

[37] .NET Framework Class Library. *Microsoft Docs.* [Online] 29. července 2015. [Citace: 12. ledna 2019.] https://docs.microsoft.com/en-us/previous-versions/gg145045(v=vs.110).

[38] Stebner, Aaron. What version of the .NET Framework is included in what version of the OS? [Online] 12. června 2017. [Citace: 4. ledna 2019.] https://blogs.msdn.microsoft.com/astebner/2007/03/14/mailbag-what-version-of-the-net-framework-is-included-in-what-version-of-the-os/.

[39] .NET Framework Versions and Dependencies. *Microsoft Docs.* [Online] 25. července 2018. [Citace: 16. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/versions-and-dependencies.

[40] .NET Framework 4.5 and Windows XP. *Microsoft.* [Online] [Citace: 2. ledna 2019.] https://devblogs.microsoft.com/dotnet/p/dotnet45xp/.

[41] Mono - Cross platform, open source .NET framework. *Mono Project.* [Online] [Citace: 2. ledna 2019.] http://www.mono-project.com/.

[42] Mono Releases. *Mono Project.* [Online] 21. prosince 2018. [Citace: 13. ledna 2019.] http://www.mono-project.com/docs/about-mono/releases/.

[43] tunnelvisionlabs/NOpenCL: .NET wrapper for OpenCL with abstraction. *GitHub.* [Online] [Citace: 20. ledna 2019.] https://github.com/tunnelvisionlabs/NOpenCL.

[44] Domino, Ken. Campy. *Campy.* [Online] [Citace: 18. února 2019.] http://campynet.com/.

[45] Nessos. GpuLinq. *GitHub.* [Online] [Citace: 20. února 2019.] https://github.com/nessos/GpuLinq.

[46] clEnqueueMapBuffer. *OpenCL Reference Pages.* [Online] [Citace: 10. prosince 2018.] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueMapBuffer.html.

[47] clEnqueueUnmapMemObject. *OpenCL Reference Pages.* [Online] [Citace: 10. prosince 2018.] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueUnmapMemObject.html.

[48] Implementing a Dispose method. *Microsoft Docs.* [Online] 7. dubna 2017. [Citace: 14. prosince 2018.] https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose.

[49] Using objects that implement IDisposable. *Microsoft Docs.* [Online] 4. dubna 2017. [Citace: 2014. prosince 2018.] https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/using-objects.

[50] BinaryReader Class. *Microsoft Docs.* [Online] [Citace: 20. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=netframework-4.7.2.

[51] BinaryWriter Class. *Microsoft Docs.* [Online] [Citace: 20. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.7.2.

[52] Parallel.ForEach Method. *Microsoft Docs.* [Online] [Citace: 10. listopadu 2018.] https://docs.microsoft.com/cs-cz/dotnet/api/system.threading.tasks.parallel.foreach?view=netframework-4.5.2.

[53] Partitioner.Create Method. *Microsoft Docs.* [Online] [Citace: 10. listopadu 2018.] https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.partitioner.create?view=netframewo

rk-4.5.2.

[54] Interlocked.CompareExchange Method. *Microsoft Docs.* [Online] [Citace: 20. ledna 2019.] https://docs.microsoft.com/cs-cz/dotnet/api/system.threading.interlocked.compareexchange?view=netframework-4.7.2.

[55] Interlocked.Increment Method. *Microsoft Docs.* [Online] [Citace: 20. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked.increment?view=netframework-4.7.2.

[56] Attributes of Variables. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/attributes-variables.html.

[57] StructLayoutAttribute Class. *Microsoft Docs.* [Online] [Citace: 20. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.structlayoutattribute?view=netframework-4.7.2.

[58] __global. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/global.html.

[59] Function Qualifiers. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/functionQualifiers.html.

[60] get_global_id. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/get_global_id.html.

[61] atomic_cmpxchg. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/atomic_cmpxchg.html.

[62] atomic_inc. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/atomic_inc.html.

[63] Interlocked.Exchange Method. *Microsoft Docs.* [Online] [Citace: 20. ledna 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked.exchange?view=netframework-4.7.2.

[64] atomic_xchg. *OpenCL Reference Pages.* [Online] [Citace: 20. ledna 2019.] https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/atomic_xchg.html.

[65] Control Class. *Microsoft Docs.* [Online] [Citace: 5. března 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.control?view=netframework-4.7.2.

[66] Graphics Class. *Microsoft Docs.* [Online] [Citace: 10. března 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.drawing.graphics?view=netframework-4.7.2.

[67] Control.Invalidate Method. *Microsoft Docs.* [Online] [Citace: 10. března 2019.] https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.control.invalidate?view=netframework-4.7.2.

[68] NBench. *GitHub.* [Online] [Citace: 24. února 2019.] https://github.com/petabridge/NBench.

[69] Cone, Matt. Getting Started. *Markdown Guide.* [Online] [Citace: 20. března 2019.] https://www.markdownguide.org/getting-started.

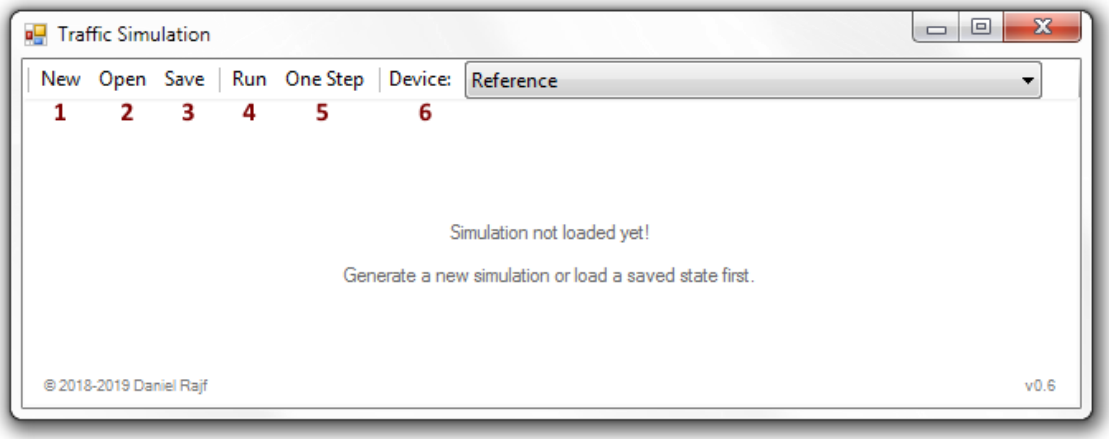
Přílohy

A Uživatelská příručka

Aplikaci je možné spustit na operačním systému Microsoft Windows 7 nebo novějším. Na počítači musí být nainstalováno rozhraní .NET Framework verze 4.5.2[1] nebo novější. Aplikaci lze spustit na 32-bitové i 64-bitové verzi operačního systému.

A.1 Hlavní aplikace

Otevřením spustitelného souboru `TrafficSimulation.exe` se aplikace spustí a zobrazí se hlavní okno aplikace, viz Obrázek 10.



Obrázek 10: Hlavní okno aplikace po spuštění

Pomocí hlavního panelu, který se nachází na horní straně okna, je možné aplikaci ovládat.

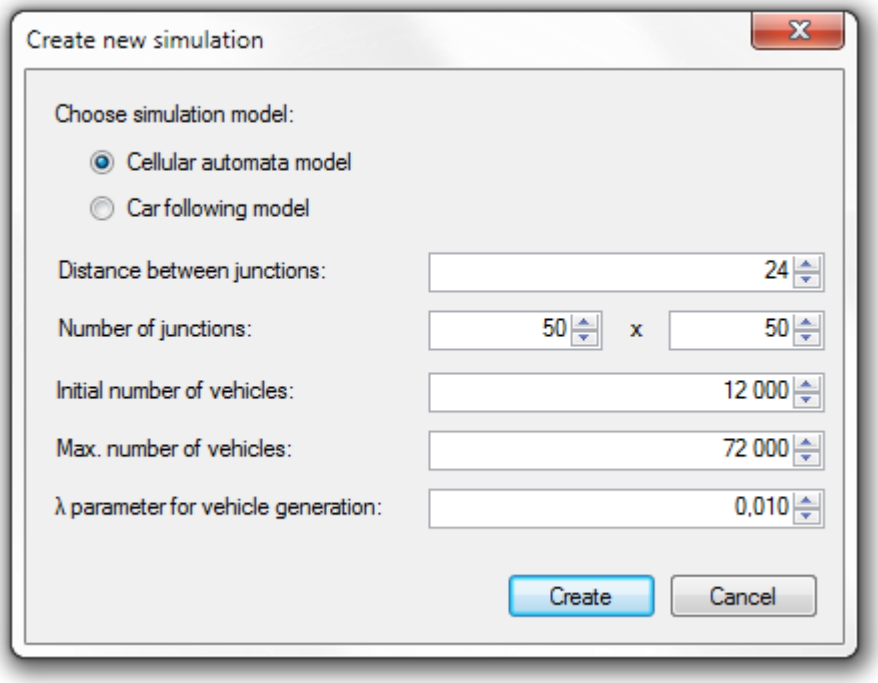
Tlačítko „New“ (označeno číslem 1) slouží pro generování nové simulace. Tlačítko „Open“ (označeno číslem 2) slouží k načtení uloženého stavu simulace ze souboru. Tlačítko „Save“ (označeno číslem 3) slouží k uložení aktuálního stavu simulace do souboru.

[1] Rozhraní .NET Framework lze stáhnout na adrese: https://www.microsoft.com/cs-cz/download/details.aspx?id=42642

Tlačítko „Run“ (označeno číslem 4) slouží pro spuštění automatického běhu simulace v sekundových krocích. Simulace je pak každou sekundu automaticky přepočítána. Stejným tlačítkem lze automatický běh vypnout.

Tlačítko „One Step“ (označeno číslem 5) slouží pro spuštění simulace pouze jednoho sekundového kroku. Toto tlačítko je možné stisknout opakovaně pro odsimulování libovolného počtu kroků.

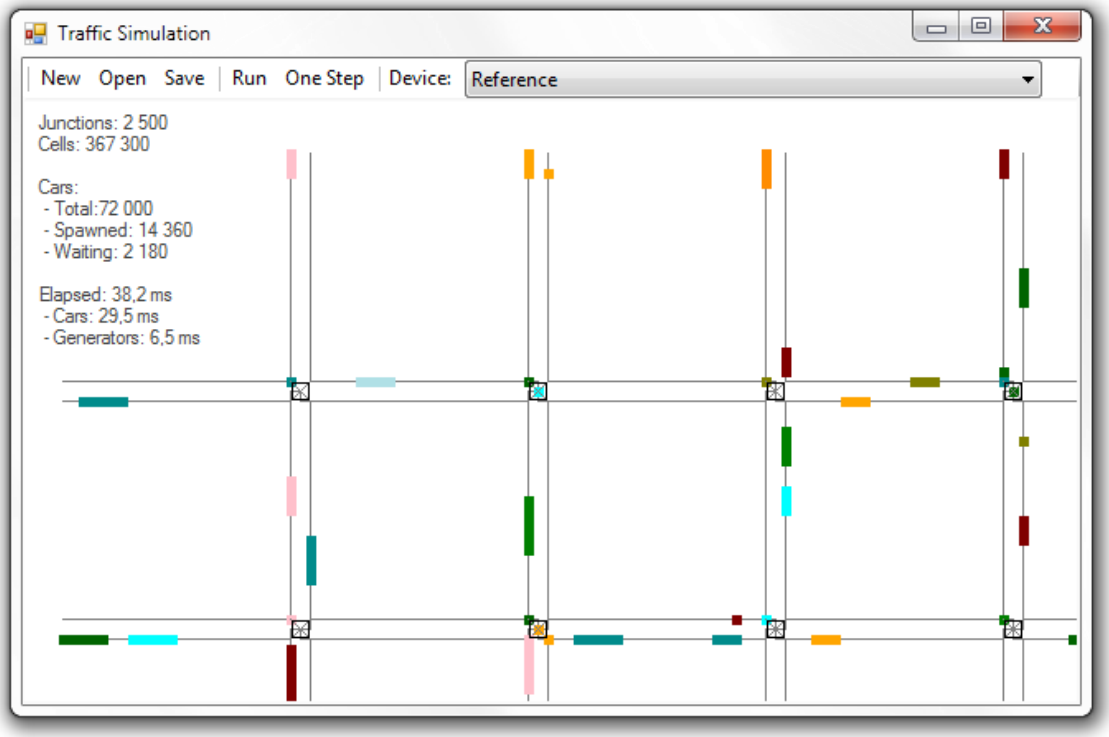
Výběr „Device“ (označený číslem 6) slouží pro výběr zařízení OpenCL, na kterém bude simulace počítána. U každého zařízení je zobrazeno, zda se jedná o GPU, nebo o CPU, výrobce daného zařízení a podporovaná verze OpenCL. Případně je možné také vybrat možnost „Reference“. Vybráním této volby bude pro výpočet použita referenční implementace simulace pomocí jednotky CPU.



Obrázek 11: Dialogové okno pro vygenerování nové simulace

Po stisknutí tlačítka pro generování nové simulace se zobrazí dialogové okno pro zadání parametrů simulace, viz Obrázek 11. V tomto okně lze zvolit simulační model, pro který bude simulace vygenerována. Dále pak vzdálenost mezi jednotlivými křižovatkami, počet křižovatek v silniční síti, počáteční počet vozidel, maximální počet vozidel a parametr λ, který bude použit v exponenciálním rozdělení pro generování nových vozidel.

Po potvrzení zadaných parametrů se vygeneruje nová simulace a řízení se vrátí zpět do hlavního okna aplikace, viz Obrázek 12. Nyní je možné pomocí tlačítek „Run“ a „One Step“ ovládat chod simulace. Dále je možné pomocí tlačítka „Save“ aktuální stav simulace uložit do souboru. V levém horním rohu se zobrazují různé informace o aktuálním stavu simulace. Mezi ně patří i celkový počet vozidel, které se mohou v jednu chvíli v simulaci objevit, označený jako „Total“, počet vozidel, které se aktuálně v simulaci nachází, označený jako „Spawned“, a počet vozidel, která aktuálně stojí a tedy pravděpodobně čekají na křižovatce, označený jako „Waiting“. Nakonec obsahuje i dobu výpočtu posledního kroku simulace.



Obrázek 12: Hlavní okno aplikace po vygenerování simulace

Vizualizaci je možné posouvat podržením levého tlačítka myši a tažením na příslušnou stranu. Dále je možné měnit úroveň přiblížení vizualizace pomocí kolečka myši. Dvojitým kliknutím na libovolnou křižovatku se daná křižovatka vybere a v levé části okna se následně zobrazí počet vozidel, která na dané křižovatce čekají.

A.2 Spouštění výkonnostních testů

Pomocí konzolové aplikace `Benchmarks.exe` lze spustit výkonnostní testy implementovaných simulací. Toto testování je popsáno v kapitole 8.3. Při spuštění bez parametrů jsou postupně spuštěny všechny dostupné výkonnostní testy.

Při použití parametrů lze spustit jenom zvolené výkonnostní testy. To lze provést tak, že je aplikace spuštěna s následujícími parametry:

```
Benchmarks.exe XYZ
```

Případně je možné spustit aplikaci pro testování v tomto tvaru:

```
Benchmarks.exe XY
```

nebo případně i v tomto tvaru:

```
Benchmarks.exe X
```

U písmen, která nejsou v zadaném parametru zastoupena, se provedou všechny dostupné možnosti. Písmeno `X` představuje simulační model, který bude testován. Lze použít následující hodnoty:

- `a` – pro model s celulárním automatem,
- `b` – pro car following model.

Písmeno `Y` představuje implementaci, která bude použita. Lze použít následující hodnoty:

- `r` – pro použití referenční implementace,
- `o` – pro použití implementace pro OpenCL.

Písmeno `Z` představuje velikost testované simulace, které jsou popsány v kapitole 8.3.2. Lze použít následující hodnoty:

- `s` – pro simulaci malého rozsahu,
- `m` – pro simulaci středního rozsahu,
- `l` – pro simulaci velkého rozsahu,
- `v` – pro simulaci velmi velkého rozsahu.

Například pro spuštění testování simulace s car following modelem s použitím implementace pro OpenCL a otestování všech dostupných velikostí simulace je nutné program spustit jako:

```
Benchmarks.exe bo
```

Výsledky jsou vždy vypisovány do standardního výstupu (resp. do konzole). Dále je automaticky vytvořeno hlášení z měření, které je automaticky uloženo do souboru ve formátu Markdown [69] do složky `Results`.

B Diagram tříd