VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

SADA TESTOVACÍCH ÚLOH PRO GRAFICKÉ KARTY

BENCHMARK SUITE FOR GRAPHICS CARDS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **JOSEF OŠKERA**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

154573

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Oškera Josef, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Superpočítání
Název:	**Sada testovacích úloh pro grafické karty**
Kategorie:	Paralelní a distribuované výpočty
Akademický rok:	2023/24

Zadání:

1. Seznamte se s architekturou grafických karet Nvidia a AMD a jejich programovacími jazyky.
2. Prostudujte dostupné sady testovacích úloh, vyhodnoťte techniky a metriky, na které se zaměřují. Vyhodnoťte názornost a srozumitelnost implementace.
3. Osvojte si metodiku vývoje akcelerovaných aplikací ve výzkumné skupině SC@FIT.
4. Navrhněte vlastní sadu testovacích úloh, která prověří jednotlivé komponenty grafické karty a bude brát v potaz nízkoúrovňové i vysokoúrovňové programovací jazyky.
5. Otestujte tuto sadu na multi-GPU systému s minimálně 4 grafickými kartami.
6. Vyhodnoťte dosažené výsledky a zdokumentujte navrženou sadu.
7. Diskutujte přínos implementované sady pro testování grafických karet a výukové účely.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Cílem této práce je vytvořit sadu úloh pro testování grafických karet, tak aby si je mohl vyzkoušet někdo další. K tomu mu dopomůže podpůrná aplikace pro spouštění, ladění a měření těchto úloh.

Abstract

The goal of this work is to create a set of tasks for testing graphics cards so that someone else can try them out. This will be aided by a supporting application to run, debug and measure these tasks.

Klíčová slova

Position Based Dynamics, C++, CUDA, OpenMP, ImGui

Keywords

Position Based Dynamics, C++, CUDA, OpenMP, ImGui

Citace

OŠKERA, Josef. *Sada testovacích úloh pro grafické karty.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Sada testovacích úloh pro grafické karty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana docenta Jířího Jaroše. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Josef Oškera
17. května 2024

Poděkování

Tímto bych chtěl poděkovat mému vedoucímu, panu docentu Jiřímu Jarošovi, za jeho podporu a velmi milý přístup, ochotu vždy pooci.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V dnešní době kdy je stále větší hlad po výpočením výkonu nabývá problematika programování na grafických kartách stále většího významu a komplexnosti. Z tohoh důvodu je dobré mít alespoň nějaké základní znalosti ohledně programování grafických karet nezávisle na tom zda je sami programujete, nebo využíváte nějakou knihovnu pro urychlení programu, nebo jen používáte nějaký software co urychluje své výpočty na grafické kartě.

Tato práce je určená pro kohokoliv kdo má zájem začít s programováním na grafických kartách a má za cíl poskytnout možnost lépe porozumět aspektům programování na grafických kartách. Toho dosáhne pomocí úloh pro otestování různých programátorských technik a jejich dopadů na výkon. Tyto úlohy jsou podpořeny vývojovým prostředím, které má za cíl usnadnit vývoj, vizualizovat problematiku a měřit výkon.

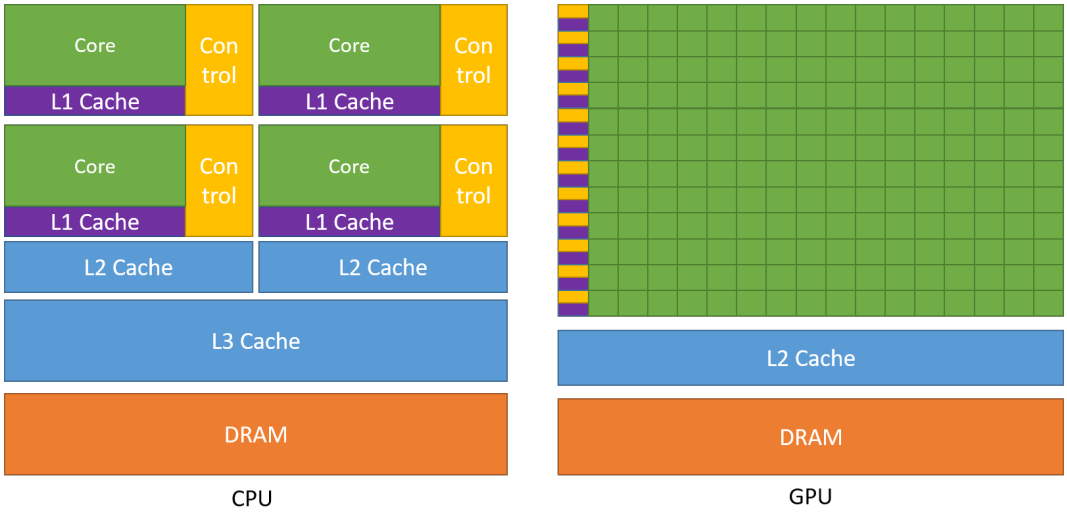
Hlavnímy cíly tedy jsou:

1. Výběr základní úlohy s rozumným kompromisem mezi programátorskou jednoduchostí a akcelerovatelností na grafické kartě.
2. Sada úkolů s cílem vytvořit implementace, které modifikují základní úlohu, tak aby ukázaly přínos určité programátorské techniky.
3. Program pro uživatelsky jednoduché spouštění, ověřování funkčnosti a ladění uživatelem vypracovaných implementací.
4. Referenční řešení implementací.
5. Program pro testování výkonnosti různých implementací.

Kapitola 2

Architektury grafických karet

V dnešní době se grafické karty již velice podobají běžným procesorům. Na rozdíl od dřívějších dob, kdy grafické karty fungovaly pouze jako zobrazovadla s fixním vykreslovacím řetězcem, nebo později s programovatelným zobrazovacím řetězcem jsou dnes tvořeny univerzálními procesory ve kterých je tato funkcionalita realizována programově a to kvůli vyvážení zátěže, protože někdy je při procesu rasterizace více náročnější vertex shader, jindy třeba pixel shader. Dalším následkem je možnost realizace pokročilých programů jako geometry shader. Jelikož při vykreslování grafiky je potřeba pro hromadu pixelů obrazovky vykonat hromadu stejné práce, tak architektura grafických karet obecně vypadá jako tříúrovňová složenina: grafický procesor složený z několika samostatných procesorů, které jsou složené z mnoha nesamostatných aritmeticko logických jednotek. Na obrázku 2.1 můžeme vidět velmi jednoduché znázorněni 4 jádrového procesoru a 192 jádrového grafického procesoru.



Obrázek 2.1: cpu vs. gpu[11]

Názvosloví

Pro jednodušší pochopení architektury je potřeba ujasnit si některé názvy.

Názvosloví architektur

Názvy pro části grafických procesorů nejsou sjednocené, každá firma, organizace nebo software používá své vlastní názvy ikdyž značí něco stejného nebo velmi podobného co má konkurence. Tabulka 2.1 zobrazuje názvy s podobným významem.

Nvidia / CUDA	AMD / HIP	Intel	další
CUDA Core	Shader Core / Streaming Processor	-	Shading Unit
Streaming Multiprocessor	Compute Unit / Workgroup Processor	Execution Unit	-
Thread	Work-item / Thread	Thread	Work-item
Warp	Wavefront	Subslice	-
(Thread) Block	Workgroup / Block	Subgroup	Workgroup / Teams
Grid	Grid	-	NDRange / NumOfWork-groups / NumOfTeams

Tabulka 2.1: Podobné výrazy [11, 13, 4]

V následujícím textu se budou používat tyto názvy:

- “jádro” pro CUDA Core
- “SM-procesor” pro streaming multiprocesor
- “vlákno” pro Thread
- “warp” pro warp
- “blok vláken” pro thread block
- “grid” pro grid
- “grafický procesor” pro GPU
- “procesor“ pro standardní CPU
- “CPU-jádro” pro jádro procesoru

Názvosloví programů

Shader je jednoduše program který běží na SM-procesoru. Existuje několik ustálených typů jako Vertex Shader, Tessellation Control Shader, Geometry Shader, Fragment Shader, Compute Shader [18], podle jejich očekávaných vstupů a výstupů.

Kernel je také program běžící čistě na grafické kartě a je to název ekvivaletní Compute Shaderu.

2.1 Architektura Obecně

Obecně jak je vidět na obrázku 2.1 je tedy grafický procesor složený z velkého počtu jednoduchých jader, které se slučují do skupin a utváří sm-procesory. Grafický procesor může a většinou mívá více sm-procesorů. Zatímco dnešní procesory mají jednotky až desítky komplexních superskalárních CPU-jader, schopných multi-threadingu, pracujících v režimu SISD (single instrucion single data) a případně SIMD (single instruction multiple data) pokud obsahují příslušné vylepšení (SSE, AVX...), tak grafické procesory jsou skalární nebo superskalární a obsahují stovky až tisíce jader, která jsou spřažená do skupin, typicky po 32, a rozdělená do sm-procesorů, které pracují v režimu SIMT (single instruction multiple thread). Výraz jádro nemusí být úplně vhodné protože pod tímto výrazem se většinou skrývá ALU typu MAD / FMA (součet + násobení) pracující s 32 bitovým celočíselným datovým typem a zároveň datovým typem s plovoucí čárkou (též 32 bitů), nebo pouze s jedním z nich a v jednom sm-procesoru může být více typů jader. Případně se pod jádrem skrývá vektorová jednotka. Navíc v sm-procesorech bývají další výpočetní jednotky například pro složitější matematické funkce jako odmocnina, sinus atd.

Jádra v rámci jednoho sm-procesoru, až na výjimky, zpracovávají všechny stejnou instrukci vydanou v rám sm-procesoru, a to proto, že sdílejí řídící logiku vydávající instrukce. Dále v sm-procesoru jádra sídlí L1 cache a rychlou sdílenou paměť. Toto je asi největší rozdíl od klasického procesoru, kde každé cpu-jádro má vlastní řídící logiku a proto může každé vykonávat jinou instrukci, nebo mít jiné taktování. Sm-procesory jsou na sobě nezávislé a nemohou spolu komunikovat jinak než přes globální paměť. Je to tak pro zachování jednoduchosti při škálování grafického procesoru. Dále grafický procesor mívá L2 cache a pokud není integrován v procesoru, tak disponuje vlastní oddělenou pamětí typu RAM nezávislou na paměti hostitelského počítače.

Vlákno, Warp, Blok

Warp je skupina vláken, typicky 32 nebo 64, jednoduše podle toho kolik je v sm-procesou spřažených jader. Pro jednoduchost si zadefinujme že warp má 32 vláken. Spražení jader tedy pro programátora znamená, že neprogramuje chod jednoho jádra, ale vždy tolik jader, jaká je velikost warpu. Podobně jako procesorový program se chová jako 1 vlákno, tak jeden warp se chová velmi podobně, jen se tím rozdílem, že je to 32 vláken vykonávající stejný program ve stejném pořadí, ale nad různými nebo stejnými daty. Na jednom sm-procesoru může běžet a pravděpodobně i poběží více warpů, které se vykonávají souběžně a pokud nejsou explicitně synchronizovány, tak i nezávisle. Warpy se seskupují do bloku vláken. Počet warpů v bloku omezen hardwarovým stropem, nebo vytížením prostředků sm-procesoru. Blok vláken je pak spuštěn na sm-procesoru jako celek. V případě potřeby může mít blok vláken i jen jedno vlákno a tím pádem bude mít jen jeden warp s jedním vláknem, ale je to mrhání prostředky.

2.2 Nvidia

Nvidia vytvořila několik programovatelných architektur: G80, Fermi (GF100), Maxwell (GM204), Kepler (GK110), Pascal (GP100), Volta (GV100), Turing (TU102), Ampere (GA100), Ada Lovelace, Hopper.

Nvidia Ampere [11, 14, 17, 15]

Jelikož v aktuálně nevýkonnějším českém superpočítači Karolina jsou grafické karty NVIDIA A100 řady Ampere, tak si rozebereme některé části této architektury.[1]

Struktura grafického procesoru této architektury je členěná následovně:

- Grafický procesor
  - GPC (Graphics Processing Cluster)
    - TPC (Texture Processing Cluster)
      - SM (Streaming Multiprocessor)

V příloze A na obrázku A.2 lze vidět, jak velký může grafický procesor být. Je zde zobrazen grafický čip GA102, který obsahuje 82 SM-procesorů.

Grafický procesor obsahuje paměťový řadič, L2 cache společnou pro všechny GPC a několik GPC. Dále může obsahovat NVLink řadič.

Graphics Processing Cluster (GPC) je nejvyšší dekompozice grafického procesoru. GPC obsahuje 6x TPC, Raster Engine, 2x8 ROP, PolyMorph Engine.

Texture Processing Cluster (TPC) obsahuje 2x SM-Procesor, PolyMorph Engine.

NVLink je speciální řadič pro propojení více grafických karet. Pro 4 linkový NVlink je maximální propustnost 112.5 GB/s mezi 2 kartami. Slouží pro přímou komunikaci mezi kartami, aby nebylo nutné přenášet data pomalou cestou přes procesor.

L2 cache je vyrovnávací paměť, která je společná pro všechny SM-Procesory. Slouží stejně jako L2 cache v procesoru pro kompenzaci rychlosti pomalé globální paměti. Je také používána pro globální atomické operace. [13]

SM-Porcesor obsahuje RT Jádro, texturovací jednotky, 2 FP64 jednotky, L1 cache a dále se dělí na 4 bloky. Znázornění sm-procesoru je vidět na obrázku 2.2. SM-procesor je jediná jednotka ve které lze provádět synchronizace vláken za běhu kernelu.

Blok SM-procesoru je nejmenší jednotka co nezávisle na ostatních vykonává instrukce programu. Nachází se v něm L0 instrukční cache, plánovač warpů (warp scheduler), vydavač warpů (dispatch), pole registrů, load/store jednotky, SFU, tensorové jádro, 16 FP32 CUDA jader a 16 duálních CUDA jader.

Za takt je schopen zpracovat 32 FP32 operací, nebo 16 INT32 plus 16 FP32 operací. Takže umí vykonávat v jednu chvíli instrukci pro float a pro int, ale jen s poloviční rychlostí (ovšem celkový počet operací je stále 32). Pokud by program pracoval pouze s celými čísly, tak rychlost se snižuje na polovinu (16 operací za takt). [14]

CUDA Jádro je jednotka pro zpracování dat. Umí pracovat pouze s FP32 (32-bitový float), nebo je duální a umí FP32 i INT32 (integer). Pokud umí oba datové typy, tak v jednu chvíli je schopno zpracovávat pouze jeden z nich. Za tak zvládne jednu FMA operaci.

[1] https://docs.it4i.cz/karolina/hardware-overview/

RT Jádro je specializované jádro určené pro akceleraci operací používaných v zobrazovací metodě založené na sledování paprsku (Raytracing). Těmito operacemi jsou například průnik paprsku (polopřímky) s trojúhelníkem, nebo obalovým tělesem.

L1 cache slouží jako datová cache pro všechny 4 smp-bloky a zároveň je použita jako sdílená paměť a jako texturní paměť. Její velikost je 128KB a v několika přednastavených poměrech se rozděluje na datovou cache a sdílenou paměť. Tato cache slouží jako nejrychlejší zdroj dat, hned po registrech.

Registrové pole je pole registrů, které se rozdělují mezi všechny aktivní vlákna v smp-bloku. Je tudíž jednou z metrik, kolik vláken lze spustit na sm-procesoru.

Super Function Unit (SFU) je jednotka pro akceleraci složitějších matematických operací jako odmocnina, dělení, sinus atd.

Tenzorové Jádro je jádro specializované na tenzorové/maticové operace, které se velmi využívají například ve strojovém učení. Hlavním přínosem je akcelerace maticového součinu. Umí pracovat i s řídkými maticemi.

Datový typ	FLOP/TAKT/SMP
FP32	128
FP64	64
matice FP16	2048/4096
matice FP32	1024/2048
matice FP64	128

Tabulka 2.2: Ampere FLOPS

Obrázek 2.2: Ampere sm-procesor [14]

2.3 AMD

Architektury: Teracale 1,2,3, GCN 1,2,3,4,5, RDNA / CDNA 1,2,3.

CDNA 2 [2, 3]

Zajímavým kandidátem je grafická karta AMD Radeon Instinct MI250X architektury CDNA 2, která se aktuálně nachází v Finském superpočítači LUMI, který je nejvýkonnějším superpočítačovým systémem v Evropě. [2]

Struktura grafického procesoru této architektury je členěná následovně:

- Grafický procesor
  - CE (Compute Engine)
    - CU (Compute Unit)

Grafický procesor Nachází se zde paměťový řadič, 4 Compute Engine, L2 cache společnou pro všechny CE, až 7 dedikovaných Infinity Fabric linek a 1 kombinovaná IF linka s PCI-E 4.0,

Compute Engine (CE) slučuje několik Compute Unit.

Infinity Fabric Pomocí této technologie je možné propojit více grafických procesorů na jedné kartě, nebo celých grafických karet mezi sebou. Případně je možné spojení s procesorem místo klasického PCI-E. Rychlost 1 linky mezi CPU a GPU je 64 GB/s. Rychlost 1 linky mezi grafickými procesory je 100 GB/s. Tudíž je možné dosáhnout rychlosti mezi 2 grafickými procesory až 700 GB/s (protože jednu IF linku musíme použit k propojení s procesorem).

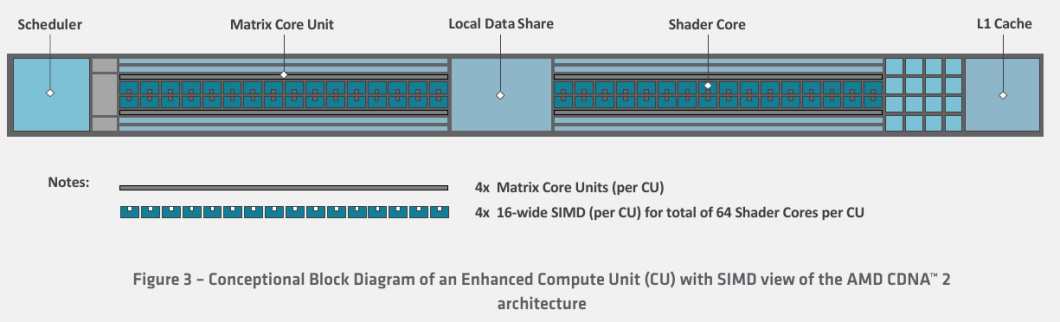
Compute Unit (CU) Skládá se ze 64 Shader jader, které jsou rozděleny do 4 skupin a každá skupina se dá použít jako vektorová jednotka o šířce 16x FP64, nebo jako maticová jednotka. Výkon jednoho CU lze vidět v tabulce 2.3. Schema je na obrázku 2.3.

Shader jádro Umí zpracovat jednu FMA operaci pro 64-bitový double za takt. Jádro umí zpracovat pouze jeden FP32, ale pokud je zabalen do tzv. Packed FP32, což je vektor dvou FP32, tak zvládne zpracovat 2 FP32 za takt.

Datový typ	FLOP/TAKT/CU
FP32	128
Packed FP32	256
FP64	128
matice FP16	1024
matice FP32	256
matice FP64	256

Tabulka 2.3: CDNA2 FLOPS

[2] https://eurohpc-ju.europa.eu/3-eurohpc-supercomputers-make-global-top-10-worlds-most-powerful-supercomputers-2023-11-13_en



Obrázek 2.3: CDNA2 Compute Unit[2]

Kapitola 3

Technologie

Seznam technologií pro obecné programování grafických karet.

1. Přímé
    - CUDA Runtime API (CUDA)
    - ROCm (HIP)
    - OpenCL
    - OpenGL (GLSL)
    - DirectX (HLSL)
    - Vulkan (SPIR-V)
        - SYCL, Circle, MLIR, Kompute, GLSL, OpenCL
    - CuPy
2. Knihovní
    - OpenMP
    - OpenACC
    - Numba

Programování

Pokud chceme programovat grafické karty je zásadní si uvědomit, že kus kódu co napíšeme, tak se neprovádí jednou, ale spoustu-krát zároveň, protože jádra co jsou spřažená do warpu, tak mohou vykonávat pouze stejnou instrukci, ale mohou se lišit daty, jaké do nich vstupují. To je jediný způsob jak ovládat celý warp. Všechna jádra ve warpu provádí stejný kód ale nad různými daty. Což je ideální pro zpracování většího množství dat.

Programování je tedy podobné jako kdyby jste se snažili kompletně rozbalit smyčku o 32 s tím že si nesmíte předávat informace mezi zbývajícími iteracemi. Je to taková výrobní linka, kde se děje hromada stejných věcí najednou.

Moderní technologie se to ale snaží ulehčit možnostmi, které nabízí. Jako využití speciálních komponent. Například sdílená paměť. Proto je nutné se o nich dozvědět co nejvíce.

3.1 CUDA

Tato sekce převážne vychází z [11, 12].

CUDA (Compute Unified Device Architecture) je paralelní výpočetní platforma a model programování vytvořený společností NVIDIA. Umožňuje vývojářům využívat výpočetní výkon grafických procesorů (GPU) pro obecné výpočetní účely. CUDA poskytuje sadu rozšíření jazyka C/C++, které umožňují vývojářům psát funkce, známé jako "kernely", které běží na GPU. Tyto kernely mohou být spuštěny na tisících vláken současně, což umožňuje výrazně zrychlit výpočty, které lze efektivně paralelizovat. CUDA také poskytuje API pro správu paměti na GPU, kopírování dat mezi hostitelem (CPU) a zařízením (GPU), a další funkce potřebné pro vývoj vysokovýkonných výpočetních aplikací.

Kernel Funkce, která běží na grafické kartě. Oproti běžnému volání funkcí v C/C++, CUDA zavádí další parametry spuštění, které se naléhají mezi třemi ostrými závorkami «<»>, kterými se nastaví počty spouštěných vláken v bloku, počet bloků vláken, nebo velikost sdílené paměti pro jeden blok vláken. Volání kernelu dokáže přímo předávat parametry do grafické karty. Volání vypadá následovně:

```
1 __global__ void subStepKernel(args, ...);
2 dim3 threadsPerBlock;
3 dim3 numBlocks;
4 ...
5 subStepKernel <<<numBlocks, threadsPerBlock>>>(args, ...);
```

JIT kompilace Kompilátor nvcc umí předkompilovat *.cu soubor do .ptx formátu, který je nezávislý na konkrétní GPU a pak později při behu programu zkompilovat pro konkrétní stroj. Další možností je použití knihovny NVRTC a kompilovat za běhu přímo ze zdrojového kódu bez předchozího zpracování.

Blokování L2 CUDA dovoluje rezervovat určitou část L2 cache pro určitá data. Toto není možné použít pokud je karta v režimu Multi-Instance GPU.

Sdílená paměť Sdílená paměť je jedním z typů paměti dostupných na GPU v architektuře CUDA. Je rychlá, ale omezená paměť, kterou mohou sdílet všechna vlákna v rámci jednoho bloku. Sdílená paměť je velmi užitečná pro zlepšení výkonu programů na grafické kartě. Může být použita k ukládání dat, která jsou často přistupována nebo jsou používána více vlákny. Tímto způsobem může sdílená paměť snížit počet přístupů k pomalejší globální paměti na GPU, což může výrazně zlepšit výkon programu. Je důležité poznamenat, že přístup k sdílené paměti musí být pečlivě koordinován mezi vlákny, aby se předešlo problémům s race conditions. K tomu se často používá funkce ___syncthreads(), která synchronizuje všechna vlákna v bloku a zajišťuje, že všechna předchozí čtení a zápisy do sdílené paměti byly dokončeny.

Použití sdílené paměti má velmi velký pozitivní dopad na výkon aplikace.

Kooperativní skupiny Kooperativní skupiny poskytují abstrakci pro skupiny vláken, které mohou spolupracovat při provádění výpočtů. Tato vlastnost rozšiřuje schopnost synchronizace vláken, která byla původně omezena pouze na vlákna ve stejném bloku.

V CUDA jsou vlákna organizována do bloků a mřížek. Vlákna ve stejném bloku mohou spolupracovat pomocí sdílené paměti a synchronizačních primitiv, jako je ___syncthreads().

Kooperativní skupiny snižují dopad tím, že poskytuje mechanismus pro vytváření skupin vláken, které mohou být v různých blocích. Tyto skupiny vláken mohou být synchronizovány pomocí funkce sync().

Konstantní paměť Je typ paměti, který sice sídlí v hlavní paměti, ale je speciálně kešován v sm-procesoru a je ideální pro typ přístupu, kdy všechny vlákna z warpu čtou stejnou hodnotu.

Texturní paměť, cuda arrays Texturní paměť v CUDA je speciální typ paměti na GPU, který je optimalizován pro situace, kdy vlákna přistupují k datům v blízkých lokalitách a kdy je přístup k datům nepravidelný. Texturní paměť je uložena v hlavní paměti GPU a je cachována v texturní cache. Texturní paměť také poskytuje hardwarovou interpolaci pro čtení hodnot mezi diskrétními body v paměti, což je užitečné pro operace jako je například texturové mapování a zpracování obrazu. Dovoluje alternativní indexování pomocí intervalu a předdefinované okrajové podmínky (co vrátí pokud se čte mimo pole).

Unifikovaná paměť CUDA umí uživatele odstínit od práce s více ukazateli na stejná data do různých pamětí. Uživatel již nemusí ve svém programu udržovat ukazatele jak do paměti hostitelského počítače, tak to paměti grafické karty, ale stačí mu jeden ukazatel a CUDA zařídí zbytek a to i s kopírováním dat mezi zařízeními.

Asynchronní souběžné funkce Některé funkce volané z hosta jsou neblokující. Spuštění kernelu, asynchronní datové přenosy mezi hostem a gpu, datové přenosy v rámci jednoho gpu, přenos dat mezi grafickými kartami.

Implicitně synchronní funkce Alokace zarovnané paměti na hostu, alokace paměti na gpu, memset na gpu, kopírování mezi 2 místy v paměti gpu. Pokud uživatel provede některou z těchto operací, tak se automaticky synchronizuje běh programu mezi hostitelem a grafickou kartou.

Streamy Kernely a datové přenosy lze sdružovat do tzv. streamů, kdy streamy jsou na sobě nezávislé a běží souběžně a vše co je v jednom streamu běží postupně. Stream funguje jako fronta. Všude kde není explicitně udán stream, tak je přiřazeno do výchozího streamu. Kompilačním příznakem je možno vynutit, že každé vlákno bude mít svůj vlastní nezávislý výchozí stream. Také je možné přepnout výchozí stream na tzv. NULL stream, který jako jediný je synchronní s hostem.

Abychom se vyhnuli nechtěné synchronizaci, tak je nejlepší se úplně vynout výchozímu streamu a vše explicitně rozdělovat mezi streamy.

Streamy mohou být vytvářeny s prioritami.

CUDA Graphs CUDA Graphs je funkce v CUDA, která umožňuje vývojářům definovat a spustit soubor propojených operací jako graf. Tyto operace mohou zahrnovat spuštění kernelů, kopírování dat mezi hostitelem a zařízením, a další. Výhodou použití CUDA Graphs je, že umožňuje efektivnější plánování a provádění operací. Když je graf jednou definován,

může být opakovaně spuštěn bez nutnosti opětovného nastavování operací. Navíc CUDA runtime může automaticky optimalizovat provádění grafu pro lepší výkon.

Multi GPU CUDA dovoluje kontrolu více grafických karet zároveň. Vždy se pomocí funkce cudaSetDevice() vybere aktivní zařízení a veškeré operace jsou mířeny na toto zařízení.

Synchronizace warpu Funkce ___syncwarp() synchronizuje všechna vlákna ve warpu. Tato funkce je užitečná v situacích, kdy některá vlákna ve warpu potřebují počkat na výsledky operací provedených jinými vlákny ve warpu.

Synchronizace vláken v bloku Funkce ___syncthreads() synchronizuje všechna vlákna v bloku. To znamená, že žádné vlákno v bloku nepokračuje dále v kódu, dokud všechna ostatní vlákna v bloku nedosáhnou tohoto bodu. Tato funkce je užitečná pro situace, kdy některá vlákna v bloku potřebují počkat na výsledky operací provedených jinými vlákny v bloku například načítání dat do sdílené paměti.

Synchronizace bloků vláken Synchronizovat bloky vláken, lze pouze ukončením kernelu, protože bloky vláken mohou být v extrémín případě vykonány sekvenčně na jednom sm-procesoru.

Atomické operace CUDA poskytuje řadu atomických funkcí, které mohou být použity na různé typy dat a operací. Některé z nich zahrnují:

- atomicAdd: Atomicky přičte hodnotu.
- atomicSub: Atomicky odečte hodnotu.
- atomicExch: Atomicky vymění hodnotu za novou hodnotou.
- atomicMin a atomicMax: Atomicky aktualizují hodnotu na minimum nebo maximum z aktuální a nové hodnoty.
- atomicInc a atomicDec: Atomicky inkrementují nebo dekrementují hodnotu.
- atomicCAS: Atomická operace porovnej a vyměň (compare and swap).

Tyto funkce mohou být použity na globální a sdílenou paměť a podporují různé datové typy, včetně celých čísel a plovoucích čísel.

Klastry vláken Novejší grafické procesory mohou pracovat s další dekompozicí vláken. Podle počtu vláken je Klastr nadřazený Bloku vláken a dovoluje jejich vzájemnou synchronizaci. Má ale omezení. Podobně jako Blok vláken beží v rámci jednoho SM-Procesoru, tak Klastr vláken musí bežet na jednom stejném Graphics Processing Cluster (GPC). Všem vláknům bežících v rámci jednoho Klastru je umožněno přistupovat do sdílených pamětí nevlastních Bloků. Tomu se pak říká Distributed Shared Memory.

3.2 HIP

HIP vyvýjí AMD a velice se podobá technologii CUDA z pohledu použití. Po vzoru CUDA implementuje vlastnosti s podobným názvem a chováním. Disponuje nástroji pro konverzi mezi CUDA a HIP kódem. Díky tomu lze mít jeden kód, který lze spouštět na na kártách více výrobců. Aktuálně neumí všechny vlastnosti CUDA.

Podporováno	Nepodporováno
ovládání zařízení	textury
správa paměti	dynamický paralelismus
streamy	kooperace s OpenGL nebo Direct3D
kernely	CUDA IPC
ohlašování chyb	CUDA arrays

Tabulka 3.1: Podpora

Spuštění kernelu může vypadat následovně:

```
1 __global__ void subStepKernel(args, ...);
2 dim3 threadsPerBlock;
3 dim3 numBlocks;
4 ...
5 hipLaunchKernelGGL(subStepKernel, numBlocks, threadsPerBlock, 0, 0, args,
    ...);
```

3.3 OpenGL

OpenGL není primárně určeno pro obecné výpočy, ale i přes to nabízí určité možnosti pro obecné výpočty v podobě compute shaderu. Nedosahuje sice tak rozsáhlých možností jako CUDA, HIP, nebo třeba OpenCL, ale základní vymoženosti poskytuje. Shader se programuje v jazyce GLSL, který vychází z C s učitými omezeními a spouští se obvyklým způsobem jako u ostatních a to počtem bloků ve 3 dimenzích. Dále umožňuje JIT kompilaci shaderu, práci se sdílenou pamětí, hlavní pamětí, konstantní paměť, texturní paměť a nastavením počtu vláken v bloku také ve 3 dimenzích, synchronizaci warpu, pamětové bariéry a další.

Má ale i dost omezení a to třeba, že počet vláken v bloku se dá měnit jen rekompilací shaderu, atomické operace umí pouze pro datový typ integer, neumí zámky, rekurzi, ukazatele, dynamickou alokaci uvnitř shaderu, nezle použít print z shaderu a tím pádem jsou debugovací možnosti silně omezeny, a také programátor musí explicitně kontrolovat jestli nepřekročil maximální velikost sdílené paměti..

Nespornou výhodou je velmi rozšířená podpora, kdy compute shadery zvládne grafická karta, která má podporu pro OpenGL verze 4.3 a vyšší, nebo OpenGL ES 3.1. a vyšší.

Pro spuštění kernelu je třeba uvést počet vláken. OpenGL nedoákže stejně jako CUDA předávat argumenty voláním a proto je nutné všechny informace do grafické karty dostat přes paměťové buffery, nebo uniformní proměnné. Spuštění vypadá například takto:

```
1 std::string subStepKernel;
2 int numThreadsX, numThreadsY, numThreadsZ;
3 ...
```

```
4 glUseProgram(subStepKernel);
5 glDispatchCompute(numThreadsX, numThreadsY, numThreadsZ);
```

3.4 OpenMP [1]

OpenMP je knihovna určená pro jednuduché paralelní programování. Primárně je cílená na procesory, ale dovoluje akceleraci na grafických kartách. Pro potřebu lěpšího rozvržení práce poskytuje OpenMP klauzule jako TARGET, TEAMS. Aby bylo možné zkompilovat kód na grafickou kartu, tak je stále potřeba speciální kompilátor (CUDA, HIP), který ovšem může být atomaticky vyžit při kopilaci pomocí gcc.

Pro nejjednodušší akceleraci kódu jsou potřeba tyto klauzule: target, teams, distribute, parallel, for, map.

Target deklaruje že kód a proměnné mají být dostupné na cílovém zařízení.

Teams určuje že práce bude rozdělena mezi týmy, což je ekvivalent jako zodělení práce do bloků vláken.

Distribute zajistí že práce v týmu bude rozdělena mezi vlákna

Map slouží pro přenos dat mezi hostem a zařízením

Velmi primitivní příklad může vypadat takto:

```
1 int * data;
2 ...
3 #pragma omp target teams distribute parallel for map(tofrom:data[0:1024])
4 for(int i = 0; i < 1024*1024; ++i){
5       for(int j = 0; j < 10240; ++j){
6               data[i]++;
7       }
8 }
```

Kapitola 4

Existující možnosti

CUDA code samples

Nvidia vydává i sbírku úloh pro jazyk CUDA, které převážné podporují pochopení možností CUDA popsaných v "CUDA C++ Programming Guide".

Příklady jsou dobře popsány, komentovány a rozděleny do skupin podle toho co se snaží ukázat. Jsou k dobré k výuce programování, ale nejsou vhodné pro výkonostní měření grafických karet.

Introduction: Tyo příklady se zaměřují na základní koncepty programování na grafické karty jako sdílená paměť, práce se streamy, použití více karet, a mnoho dalšího. Většina příkladů je poměrně krátká, jednoduchá a lehce pochopitelná. Utilities: Krátké programy pro zjištění základních informací o grafické kartě. Concepts and Techniques: Tato skupina se zaměřuje na realizaci pokročilých algoritmů jako například cosinová transformace, histogram, konvoluční filtr, redukce, scan, řadící algoritmy. CUDA Features: Jsou praktické příklady ohledně některých vymožeností CUDA jako použití tenzorových jader, CUDA Graph, práce se streamy. CUDA Libraries: Demostrace použití CUDA knihoven jako CUBLAS atd. Domain specific: Komplexnejší programy demonstrující výpočetně náročnejší úlohu a propojení s zobrazovacími technologiemi jako OpenGL, Vulkan, Directx. Performance: Zde jsou pouze 4 příklady demonstrující výkonnostní dopad vybraných možností jazyka CUDA.

Tato sbírka se tedy hodí pro pochopení jak použít některé vlastnosti CUDA, ale nepůjdou jednoduše složit do nějakého většiho celku.

HIP examples

I AMD má menší sbírku úloh, která je ale o dost menší než od Nvidie. Ale stále se hodí pro základní porozumnění využití grafické karty.

Mixbench

Tento program se zaměřuje na meření výkonu pomocí operací násobení sčítání a porovnání. Zaměřuje se tedy jen na surový výkon, ale za to pro několik datových typů a na více platformách jako CUDA, HIP, SYCL, OpneCL a cpu. Tento benchmark umí s dobrou přesností změřit výkon blížící se teoretickému výkonu. [5] Jeho implementace je velmi krátká účelná, díky tomu je lehce čitelná.

Sám o sobě se tedy moc nehodí pro cíle této práce, jelikož nevyužívá zajímavé možnosti grafických karet jako sdílená paměť, ale mohl by být dobrým doplňkem aby uživatel mohl porovnat své implementace jestli mají očekáváné zrychlení na různem hardwaru.

ViennaCL

Je dalším nástrojem určeným k testování výkonu procesorů a grafických karet. Zameřuje se na testování pomocí oprerací lineární algebry jako jsou sčítání a násobení skalárních hodnot, vektorů a matic a dalších. Běží na technologiích jako OpenMP, CUDA, OpneCL. Implementace testů jsou celkem krátké a stručné, ale zanořené do dalších zdrojových souborů.[1]

Hashcat

Je nástroj na získávání hesel šifrovaných dat, ale jelikož k tomu vyžaduje velký výpočetní výkon, tak se jeví jako dobrou volbou i pro testování výkonu samotného. Pro grafické karty je implementován v jazyce OpenCL. Jeho zdrojové kódy jsou ale velmi špatně čitelné, což pravděpodobně silně souvisí s optimalizacemi a také se samotným fungováním šifrovacích algoritmů.[2]

[1] https://viennacl.sourceforge.net/viennacl-about.html
[2] https://github.com/hashcat/hashcat/tree/master

Kapitola 5

Návrh

Vzhledem k architektuře grafického procesoru není vhodné zvolit jakýkoliv typ úlohy, protože by mohla být příliž složitá pro výukové účely, nebo může být tak primitivní, že by programátroské techniky měly jen malý či žádný vliv. Proto je potřeba najít takovou úlohu, která pracuje s větším množstvím dat, jelikož je u ní vyšší pravděpodobnost snadnější paralelizace, než sekvenční úloha, ale zároveň to není jen něco naprosto jednoduchého jako sčítání vektoru. Je vhodné aby řešený problém nad daty měl závislosti jak prostorové (tím je myšleno například že součet potřebuje 2 operandy), tak sekvenční (například součet následovaný dělením). Vhodným kandidátem se jeví nějaký matematický problém, jelikož matematika nám může poskytnout nepřeberné množství možností jak pracovat s až nekonečným počtem dat.

Dalším cílem návrhu je zvolit úlohu takovou, která půjde snadno reprezentovat. Jako ideální se tedy jeví možnost vizuálně zobrazovat běh úlohy. Toho bude docíleno pomocí GUI aplikace, která bude v sobě zahrnovat jak reprezentaci úlohy, tak i její jednoduché spouštění a zobrazování výkonnostních výsledků.

5.1 Přeběžný návrh výpočtu

Základní úlohu bude tvořit systém částic, které se budou pohybovat v omezeném prostoru (omezeném z důvodu snadné zobrazovatelnosti) ve kterém působí síla imitující gravitaci. Částice budou na sobě závislé nebo budou spolu nějak interagovat. Jak bude určeno podle vybrané metody simulace tohoto systému. Tím bychom měli získat dostatečně výpočetně náročný problém s neomezenou škálovatelností.

5.2 Simulační metoda

Tato sekce vychází z [10, 9, 8] a přebírá z ní příklady pro vysvětlení funkčnosti.

Pro simulaci systému částic jsem zvolil Extended Position Based Dynamics (XPBD), která rozšiřuje Position Based Dynamics. PBD je iterativní metoda simulace dynamických systémů, která se zaměřuje přímo na manipulaci s pozicemi objektů, místo tradičních metod založených na silách nebo impulzech. Tato metoda je významná svou stabilitou, rychlostí a především schopností přesněji kontrolovat simulované scény.

Výsledkem výpočtu PBD je pro objekt n poloha x_n v čase t. Jako jednoduchý příklad výpočtu jedné iterace (X)PBD metody poslouží korálek na kroužku. Jak je možné vidět na obrázku 5.1, tak máme červený korálek na kroužku. Korálek má vlastnost že se pohybuje ve

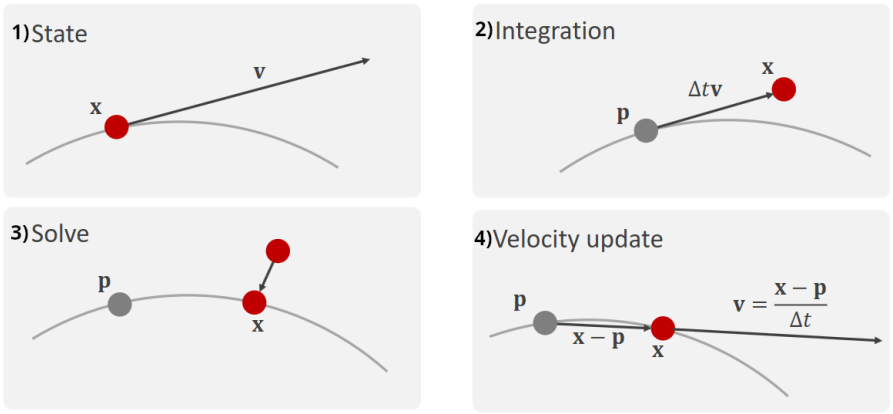
směru jeho pohybu. Ale má i další vlastnost a to že je navlečen na kroužku, který omezuje oblasti ve kterých se může v prostoru vyskytovat. A to je reprezentováno tzv. *omezující podmínkou* (constraint). Výpočet probíhá následovně: Nejprve získáme polohu x a rychlost v korálku. Jelikož rychlost v je derivace polohy x, tak pro výpočet nové polohy x z aktuální polohy p je použit integrační krok explicitní eulerovy metody:

$$y_{n+1} = y_n + h * f(t_n, y_n) \tag{5.1}$$

, která po substituci za naše proměnné vypadá takto:

$$x = p + \triangle t * v$$

. Třetím krokem je přímá manipulace s polohou objektu za účelem uspokojení omezující podmínky, která definuje chování objektu. To je realizováno tak, že se korálek přesune na nejbližší místo, kde uspokojí podmínku, a to na kroužek. Posledním krokem je aktualizace rychlosti. Jelikož uspokojení podmínky může změnit směr pohybu objektu, tak je třeba upravit vektor rychlosti aby odpovídala nové derivaci polohy.



Obrázek 5.1: PBD Euler [9]

PBD dále zavádí obecný zápis omezení, tudíž je možné jednoduše využít mnohem komplexnější omezení než vzdálenost. Omezení je vyjádřeno funkcí $C(x_1, x_2, .., x_n)$, jejímž výsledkem je míra odchylky od splnění. Uvažme příklad kde máme objekty A_n s polohou $\overline{x}_n$, které mají být od sebe vzálené o délu l_0. Omezující funkce bude vypadat takto:

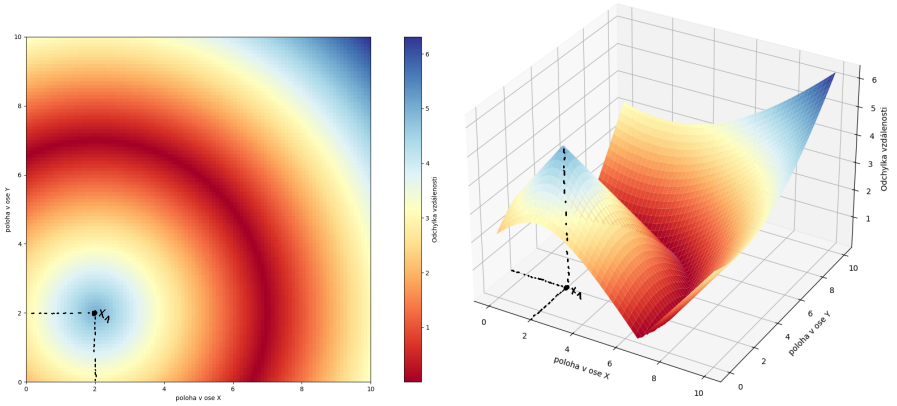
$$C_{dist}(\overline{x}_1, \overline{x}_2) = |\overline{x}_2 - \overline{x}_1| - l_0 \tag{5.2}$$

Pokud zvolíme $\overline{x}_1 = [2, 2]$, $l_0 = 5$, tak hodnotu funkce $C_{dist}(\overline{x}_1, \overline{x}_2)$ pro $\forall \overline{x}_2 | x_{21}, x_{22} \in < 0, 10 >$ znázorňuje obrázek 5.2, kde tmavě červená barva indikuje možné polohy $\overline{x}_2$ objektu A_2, které se blíží splnění podmínky. Je na něm i zaznačena poloha x_1 objektu A_1.

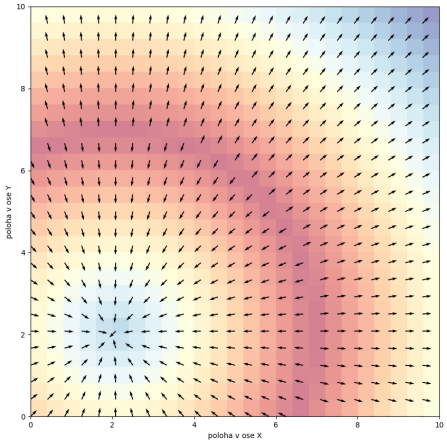
PBD dále závádí gradient omezení $\nabla C(x_1, x_2, .., x_n)$, což je vektor, který udává jakým směrem se bude nejrychleji zvyšovat odchylka neboli hodnota funkce omezení C. Pro náš příklad vzdálenosti 5.2 vypadá takto:

$$\nabla C_2(\overline{x}_1, \overline{x}_2) = \frac{\overline{x}_2 - \overline{x}_1}{|\overline{x}_2 - \overline{x}_1|} \tag{5.3}$$

jak je možno vidět na obrázku 5.3. Aby se objekt A_2dostal do správné polohy, tak ho musíme posunout proti směru jeho ∇C_2.



Obrázek 5.2: Funkce omezení



Obrázek 5.3: Gradient omezení

Výpočet korekce polohy $\triangle x$ je pak dán vztahem:

$$\triangle x_i = \lambda w_i \nabla C_i(x_1, .., x_n) \tag{5.4}$$

$$\lambda = -\frac{C(x_1, .., x_n)}{\frac{\alpha}{\triangle t^2} + \sum_j w_j \left|\nabla C_j(x_1, .., x_n)\right|^2} \tag{5.5}$$

, kde w_i je inverzní hmotnost objektu A_i, α je koeficient pro imitaci tuhosti, přičemž $\alpha = 0$ způsobí nekonečnou tuhost a $\triangle t$ je délka kroku.

Algoritmus PBD

Algoritmus PBD 1 vychází z klasické Eulerovy metody, kdy nejprve se spočítá nová poloha pro všechny objekty, následně vyřeší všechny omezení a nakonec se aktualizují rychlosti. Jelikož uspokojení omezení se okamžitě aplikuje, tak řešení omezení připomíná Gauss-Seidelovu iterační metodu. Pro zvýšení přesnosti je buď provedeno více iterací řešení omezení, nebo využití tzv. *substep[9, 8]* což je pouhé rozdělení kroku původní simulace na kratší kroky.

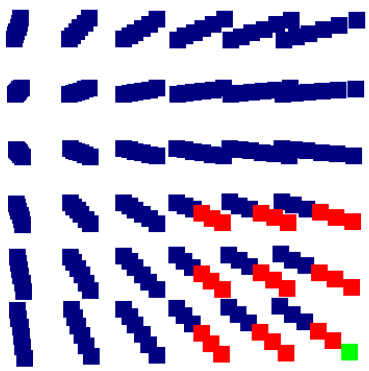
Algorithm 1 PBD Algoritmus

```
1  while simulation
2          for all particles i
3                  Pi = Xi
4                  Xi = Xi + dT * Vi
5
6          for all constrain C
7                  solve (C, dT)
8
9          for all particles i
10                 Vi = (Xi − Pi) / dT
11
12 solve(C, dT):
13 for all particles i of C
14         compute dXi
15         Xi = Xi + dXi
```

5.3 Finální návrh výpočtu

Základní úlohu tvoří systém hmotných částic, na které působí gravitace $\overline{G}$. Částice $\check{C}$ jsou tvořeny rychlostí $\overline{V}$, pozicí $\overline{X}$, polohou v systému částic $\overline{P}$. Částice jsou organizovány do 3-dimenzionální krychle a nikdy neměni polohu v systému (mohou měnit pouze polohu v prostoru). Poloha částic je omezená prostorem tvořící kvádr. Tyto částice tvoří měkkou krychli o hraně H *o velikosti* $2..n$ a toho je docíleno pomocí vzdálenostních omezení pro zachování maximální jednoduchosti úlohy. Každá částice má sousedy v každé dimenzi do volitelné vzdálenosti S a spolu se sousedy tvoří podkrychli. Rozsah sousednosti S je $1..H-1$. To znamená že každá částice má tolik omezení jako má sousedů, což je maximálně $(2S+1)^3-1$. Na obrázku 5.4 lze vidět vizualizaci kostky o hraně $H=6$ se sousedností $S=2$, kde modře jsou zaznačeny ostatní částice a červeně částice sousedící se zelenou částicí. Další důležité veličiny jsou: čas t a délka kroku simulace $\triangle t$.

Další částí úlohy je spočítání energie systému, tzn. potenciální + kinetická energie částic. Částice mají všechny shodnou hmotnost $m=1$.

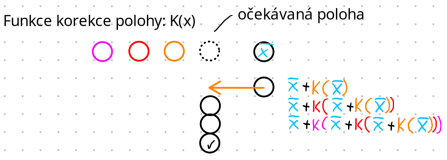


Obrázek 5.4: Sousednost

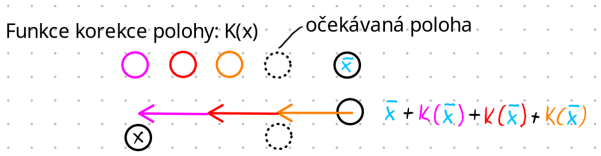
Algoritmus

Aby bylo možné výpočet velmi jednoduše paralelizovat, tak je nutné de-serializovat uspokojení omezení. To je provedeno tak, že místo toho aby výpočet probíhal stylem výpočet korekce 1 -> aplikace korekce 1 -> výpočet korekce 2 -> aplikace korekce 2, tak se všechny korekce polohy počítají zároveň a pak se aplikuje jejich suma. Od originálního algoritmu připomínající Gauss-Seidelovu metodu, nyní výpočet připomíná Jacobiho iterační metodu. Dále místo toho, aby se omezení počítalo pro 2 částice a těm se zároveň upravila poloha, tak pro jednoduchost se spočtení a korekce polohy počítá zvlášť pro každou částici, tím se eliminuje potřeba počítat dvojice a je možné aby se o každou částici staralo jedno vlákno/proces. Tyto změny ale způsobují destabilizaci výpočtu.

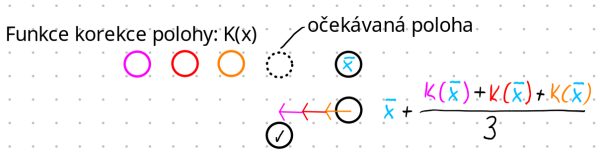
Na obrázku 5.5 můžeme vidět jak se chová korekce polohy pro systém 4 částic, které se snaží být 1 políčko od sebe. Nejpravější částice se snaží splnit 3 omezení. Pokud to dělá sériove, tak je vše v pořádku, ale pokud to dělá paralelně, tak místo opravy polohy, se chyba polohy mnohonásobně zvětší. Proto je všechny korekce polohy podělit počtem omezení, možná se tím trochu zpomalí konvergence k ustálenému řešení, ale získáme zpět stabilitu.



(a) Sériově



(b) Paralelně Chybně



(c) Paralelně Správně

Obrázek 5.5: Problém paralelní nestability

Modifikovaný algoritmus 1 tedy vapadá následovně:

Algorithm 2 Algoritmus základní úlohy

```
1  while simulation
2          for all particles p
3                  apply_forces(p)
4
5          for all particles p
6                  euler_step(p)
7
8          k−times do
9                  for all particles p
10                         compute_correction(p)
11
12                 for all particles p
13                         apply_correction(p)
14
15         for all particles p
16                 euler_finish(p)
17
18         for all particles p
19                 check_borders(p)
20
21         compute_energy()
```

Nejprve se pro všechny částice provede integrační krok eulerovy metody, poté se k-krát pro všechny částice spočítají opravy polohy, které se následně provedou. Provede se aktualizace rychlosti a nakonec se pro všechny částice ověří zda se nenáchází mimo prostor a případně vrátí do prostoru a spočítá jejich kinetická a potenciální energie. Tento algoritmus je možné snadno paralelizovat, protože všechny úkony v rámci jednoho kroku, jsou na sobě nezávislé. Závislé jsou pouze kroky mezi sebou.

Vztahy

Algoritmus základní úlohy je také možné v určitém případě vyjjádřit čistě matematicky. Jedna iterace simulace pro částici $\check{C}_i$ a množinu indexů jejích sousedů M_i, v případě kdy $k = 1$, vypadá následovně:

1. Poloha $\overline{X}$ (euler_step + apply_correction):

$$\overline{X}_{i_t} = \overline{X}_{i_{t-1}} + \frac{\triangle \overline{X}_{i_{t-1}}}{|M_i|} + \triangle t \overline{V}_{i_{t-1}} \tag{5.6}$$

2. Vzdálenost D pro kteroukoliv dvojici ($\check{C}_i$,$\check{C}_j$), kde d je volitelný skalární parameter vzdálenosti sousedů, je dána vztahem:

$$D_{ij} = \left|(\overline{P}_j - \overline{P}_i) * d\right| \tag{5.7}$$

3. Funkce omezení C:

$$C(\overline{X}_1, \overline{X}_2) = \left|\overline{X}_2 - \overline{X}_1\right| - D_{12} \tag{5.8}$$

4. Funkce gradientu omezení ∇C:

$$\nabla C(\overline{X}_1, \overline{X}_2) = \frac{\overline{X}_2 - \overline{X}_1}{\left|\overline{X}_2 - \overline{X}_1\right|} \tag{5.9}$$

5. Lambda paramter λ. Jelikož omezení má vždy jen 2 účastníky a částice mají identickou hmotnost a druhá mocnina délky výsledku funkce gradinetu omezení bude vždy 1 (protože omezení vzdálenosti je výpočet stejný jako normalizace vektoru), tak vztah 5.5 lze upravit na:

$$\lambda(\overline{X}_1, \overline{X}_2) = -\frac{C(\overline{X}_1, \overline{X}_2)}{\frac{\alpha}{\triangle t^2} + 2} \tag{5.10}$$

6. Korekce polohy $\triangle\overline{X}$ (compute_correction). Protože jedno omezení tvoří vždy jen 2 částice a všechny se počítají paralelně, tak vztah 5.4 lze upravit na:

$$\triangle\overline{X}_i = \sum_j \lambda(\overline{X}_i, \overline{X}_j)\nabla C(\overline{X}_i, \overline{X}_j) | j \in M_i \tag{5.11}$$

7. Aktualizace rychlosti (euler_finish + apply_correction + apply_forces), kde $\overline{F}$ je externí síla působící na částici a $\overline{G}$ je gravitace:

$$\overline{V}_{i_t} = \frac{\overline{X}_{i_t} + \triangle\overline{X}_{i_t} - \overline{X}_{i_{t-1}}}{\triangle t} + \triangle t\overline{F} + \triangle t\overline{G} \tag{5.12}$$

Výpočet energie E pro jednu iteraci simulace (compute_energy) je dán vztahem:

$$E = \sum_i mgh_i + \sum_i m\left|\overline{V}_i\right|^2 \tag{5.13}$$

,kde $m = 1$, $g = \left|\overline{G}\right|$, h_i je vzálenost částice $\check{C}_i$ od konce prostoru ke kterému směřuje vektor gravitace $\overline{G}$.

5.4 Návrh obslužného programu

Aby nebylo nutné psát vše od nuly, což by pravděpodobně zbytečně zatěžovalo uživatele, je vhodné k úlohám mít nějaký další program co by tohle zajistil.

Pro urychlení vývoje se hodí grafické zobrazení dat. Na rozdíl od hromady čísel, je tak jednodušší zobrazit vetší počet dat s menším počtem parametrů, nás zajímá primárně poloha částic v čase, takže se to jeví jako vhodná volba. Pokud uživatel bude vědět jak má průběh simluace vypadat, tak velmi rychle odhalí že je něco špatně a pokud alespoň částečně rozumí základní úloze, tak by mohl snáze odhadnout příčinu problému.

Uživatelské rozhraní

Rozhraní bude grafické grafické, rozložené na 2 obrazovky, jedna pro vývoj, druhá pro testování výkonnosti.

Obrazovka vývoje bude obsahovat vykreslování simulace, ovládací prvky pro kontrolu běhu a zobrazování simulace, nastavení parametrů simulace, okno pro zobrazení surových hodnot.

Obrazovka testování výkonu bude obsahovat prvek pro sestavení vlastního “skriptu” s možností nastavení velikosti simulace, počet sousedů a výběru jednotlivých uživatelských implementací. Dále také bude zobrazovat vykreslení simulace a bude také vykreslovat grafy pro dobu iterace, zrychlení vybrané implementace vůči ostatním a graf pro odhalení možné závislosti mezi implementací a počtem částic a sousedů.

Datové uložiště

Pro vlastní experimenty uživatele by se mohla hodit možnost ukládat vlastní simulace pro konkrétní nastavení, aby bylo možné krok po kroku porovnávat dvě různé implementace. Je bude tedy potřeba ukládat jak nastavení simulace, tak jednotlivé kroky a statistiky výkonu. Pro ukládání nastavení a průběhu simulace bude zvolen formát HDF5, kvůli jeho širokým možnostem použití.

Začlenění uživatelských implementací

Dalšímu urychlení práce pomůže automatické začlenění uživatelského kód do projektu, bez modifikace obslužného programu. Uprávy neznámého kódu můžou způsobit nepředvítelné chyby a to by mohlo uživatele velmi zpomalit. Kopírování celého programu také není ideální.

Uživatelské implementace se budou chovat jako samostatné moduly, bez nutnosti jakkoliv upravovat další kóď aplikace. Přidání nebo Odstranění implementace nijak neovlivní jiný kód nebo implementaci.

5.5 Návrh úloh

5.5.1 Naivní implementace

Cílem je co nejjednodušeji modifikovat výchozí implementaci smiluace. Tím získáme první pohled na to o kolik je grafická karta výkonější než klasický prosesor a jestli zrychlení odpovídá teoretickému rozdílu výkonu cpu a gpu.

5.5.2 Datové rozestupy

Pro grafickou kartu je přirozenější, když vlákna sahají do hlavní paměti po 32-bit rozestupech.[12] Pokud máme data upořádané v poli struktur a vlákna začnou přistupovat ke stejným členům struktury, tak do paměti sahají s rozestupem, který je tak velký jako velikost dané struktury. Proto je vhodné pole struktur rozbít na pole členů struktury.

Cílem této úlohy je tedy vyzkoušet dopad snižení rozestupů dat.

5.5.3 Dekompozice

Možnost rozložení vláken v Bloku je možné ve 3 dimenzích. Cílem této úlohy je tedy vyzkoušet kdy a za jakých podmínek je vhodné využít 1D, 2D, nebo 3D blok vláken.

5.5.4 Sdílená paměť

V naivní implementaci přistupuje vlákno do hlavní paměti několikrát během jedné iterace simluace. Hlavní paměť má jen omezenou propustnost a proto může v určitých případech zvýšit výkon omezení její zátěže. V případech kdy každé vlákno přistoupí jen jednou do hlavní paměti nám to nijak nepomůže. To je velmi blízké případu kdy v simulaci sousednost $S = 1$. Pokud ovšem vlákna používají data, která využivají i ostatní vlákna v bloku, tak místo opakovaného čtení stejné oblasti paměti je možné ji načíst do sdílené paměti a pak opakovaně číst z ní. Největší zisk výkonu by měl být tedy v případě, kdy $S = H - 1$, protože každé vlákno musí postupně z hlavní paměti přečíst polohy všech částic v systému.

Cílem této úlohy je tedy ukázat jaký dopad a v jakém případě má využití sdílené paměti.

5.5.5 Odmocnina

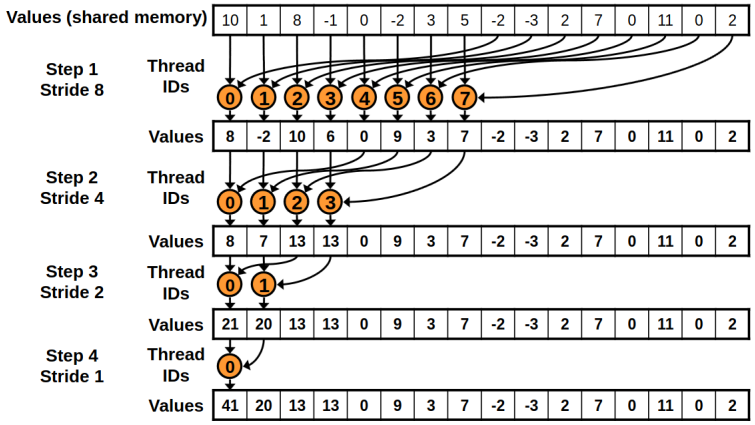
Výchozí implementace obsahuje ve výpočtu několik operací odmocnina a děleno na výpočet jednoho omezení. Některé oprace jsou velmi pomalé jako například odmocnina, kde je výkon i 8-krát menší než u operace FMA.[11] Dále je možné využít vestavené funkce, které jsou akcelerovány ve SFU jednotkách. Jsou sice méně přesné, ale zato přináší i 10-krát větší výkon.[6]

Cílem této úlohy je ukázat výkonnostní nárůst v případě když by se podařilo odmocnině vyhnout i za cenu snížení přesnosti výpočtu. Například pomocí Rychlé inverzní odmocniny ze hry Quake III, nebo použití odmocniny a dělení se sníženou přesností pomocí přepínačů "-prec-div=false" a "-prec-sqrt=false" případně přímeho využití vestavěných CUDA funkcí jako "___fdividef()".

5.5.6 Redukce

Výpočet energie v základní úloze vyžaduje sumu vypočtených hodnot ze všech částic. Využití aotmických operací není ideální protože se pokaždé musí v této operaci serializovat všechna vlákna.

Cílem této úlohy je tedy využít sdílené paměti jako cíle pro atomické operace a algoritmus *Sequential Addressing*[16] pro paralelní redukce. Jeho princip je možné vidět na obrázku 5.6 a funguje, tak že se v cyklu postupně půlí počet pracujících vlákenm které dělají parciální redukce z dat neaktivních vláken.



Obrázek 5.6: *Sequential Addressing[16]*

5.5.7 Více GPU

Další možností jak zvýšit výpočetní výkon je prosté použití více hardwaru.

Cílem uje tedy rozdělit výpočet mezi více grafických karet, nejprve s přenosem dat přes hostitele a pro porovnání i peer-to-peer komunikaci.

5.5.8 OpenMP

Cílem této úlhoy je zjistit jak moc výhodné je využít vyšší abstrakci pro programování GPU než je CUDA.

5.6 Shrnutí Návrhu

Důležitými možnostmi co nám tento návrh dává je možnost měnit velikost problému a to jak z pohledu pamětové náročnosti, tak i aritmetické složitosti bez změny paměťové složitosti. Dále je to pravděpodobně jenoduchá paralelizace daná tím že spočítání částic není nijak závislé na průběhu počítání jiné částice, ale pouze jsou zde synchronizacní body.

Kapitola 6

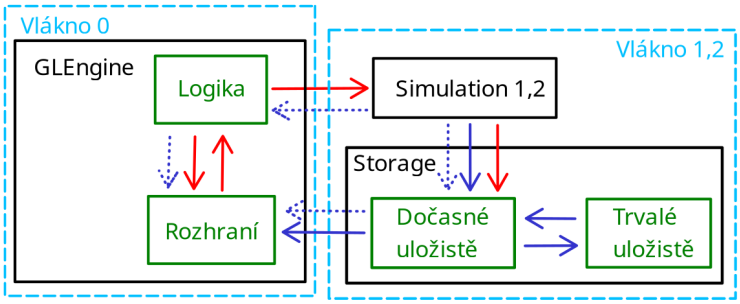
Implementace

Za jazyk pro implementaci úloh byla zvolena CUDA, protože nabízí nejpokročilejší možnosti pro využití možností grafických karet. Pro implementaci obslužného programu bylo zvoleno C++, kvůli snadné integraci s CUDA a OpenMP. Pro grafický výstup bylo zvoleno OpneGL z důvodu relativní jednoduchosti použití a dobrého výkonu. Pro grafické uživatelské rozhraní byla zvolena knihovna ImGui[1] z důvodu jednoduché integrace do projektu, malého počtu závislostí a také velké flexibility použití. Pro ukládání dat byla zvolena knihovna HDF5.

6.1 Implementace obslužného programu

Jádro programu tvoří třída GLEngine, které zajišťuje vykreslování, obsluhu uživatelských vstupů, řízení simulací a generování grafů. Druhou důležitou je třída Simulation, což je bázová třída pro třídy uživatelských implementací, které z ní dědí a částečně ji implementují. Poslední zásadní třída je Storage, která zajišťuje uchování dat pro vykreslení a zajišťuje případné trvalé uložení na disk.

Na obrázku 6.1 je možné vidět zjednodušenou dekompozici programu. Červeně jsou vyznačeny příkazy / volání funkcí, modře tok velkých dat a tečkovaně tok malých dat jako třeba stav simulace. Černou barvou jsou vyznačeny třídy a zelenou dekompozice pro lepší pochopení. Jelikož obslužná smyčka OpenGL je synchronní s obnovovací frekvencí monitoru na kterém program běží, tak bylo nutné program rozdělit do více vláken, aby nebyla simulace brzděna tímto faktem. Dalším důvodem pro využití více vláken je možnost běhu 2 simulací zároveň, aby bylo možné přímé srovnání správnosti.



Obrázek 6.1: Schéma programu

[1] https://github.com/ocornut/imgui

6.1.1 GLEngine

Je největší a nejsložitější třídou, proto je její implementace rozdělená do několika souborů, které vždy souvisí s tím co příslušná část dělá což je většinou podle uživatelského rozhraní.

Hlavní

GLEngine.cpp Zde se nachází hlavní smyčka programu, inicializace OpenGL, ImGui, shader programu a jeho bufferu. Z hlavní smyčky je také voláno vykreslování GUI a přímo se zde vykreslují částice simulace.

GLEngineDevelScreen.cpp V tomto souboru se nachází kompletní logika vývojové obrazovky. Tzn. kompletní ovládání simulací, jak jejich běhu, tak jejich parametrů, a jednoduché zobrazovadlo surových hodnot ze simulace.

GLEngineBenchScreen.cpp Zde se nalézá logika testovací obrazovky tzn. vytváření a spouštění skriptu simulací a generování grafů z výsledků měření.

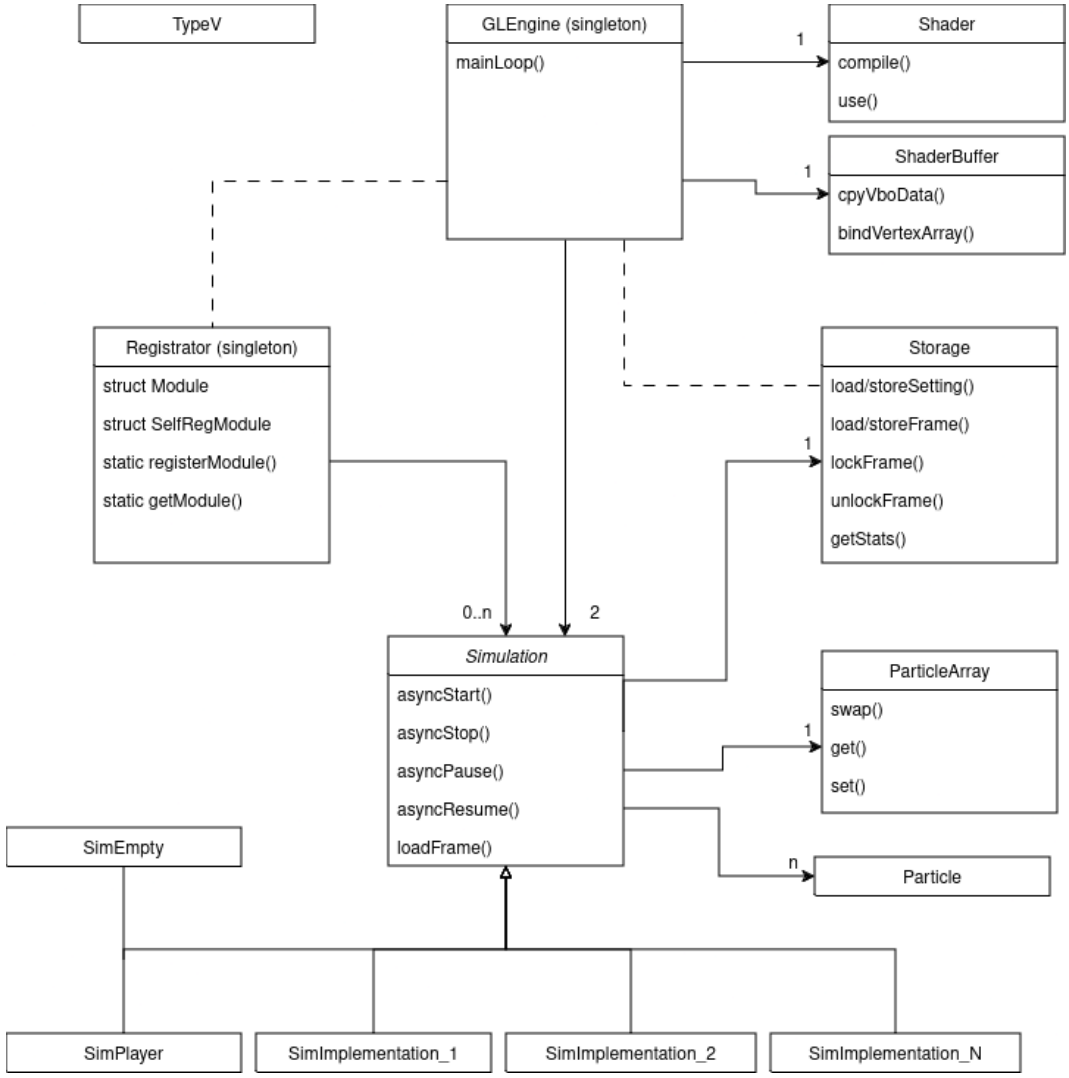
Podpůrné

- GLEngineMainScreen.cpp - hlavní obrazovka
- GLEngineModals.cpp - pomocné podokna
- GLEngineObjects.cpp - upravené objekty jako tlačítka atd.
- GLEngineWindows.cpp - okna společná pro obě obrazovky
- GLEngineInputHandlers.cpp - obsluha vstupů z klávesnice a myši.
- GLEngineSimControll.cpp - pomocné funkce pro ovládání třídy Simulation

6.1.2 Registrator

Tato třída je jedináček a zajišťuje integraci uživatelských implementací do programu, aniž by se musely jakkoliv programově přidat do obslužné aplikace. To umožňuje pomocí samoregistrace implementace přímo z jejího souboru zdrojového kódu. Pro tento trik je využito konstruktoru. Do implementace se přidá instance třídy Registrator::SelfRegModule, která ve svém konstruktoru přidá do zásobníku informace o implementaci, jako ukazatel na funkci vracející ukazatel na novou dynamickou instanci třídy uživatelské implementace.

Aby bylo možné objekt SelfRegModule vytvořit ve více implementacích a vícekrát v jednom zdrojovém souboru, tak pro pohodlí jsou přidána makra 3 co zajišťují toto chování.



Obrázek 6.2: Diagram tříd

Algorithm 3 Makra registrátoru

```
1 #define CONCAT(A) __registrator_ ## A
2 #define EXPAND(A) CONCAT(A)
3 #define REGISTER_MODULE(name, sim, priority, desc) static
    Registrator::SelfRegModule EXPAND(__COUNTER__) (name, sim, priority,
    desc)
4 #define REGISTER_SIMULATIUON(classname, name, priority, desc)
    REGISTER_MODULE(name, [](){return static_cast<Simulation*>(new
    classname());}, priority, desc)
5 #define REGISTER_USER_SIMULATIUON(classname, name, desc)
    REGISTER_MODULE(name, [](){return static_cast<Simulation*>(new
    classname());}, 4, desc)
```

Makro REGISTER_MODULE vytvoří statickou instanci třídy SelfRegModule, takže je možné ji tímto makrem vytvořit ve více souborech a její název je vytvořen pomocí vestavěného makra ___COUNTER___, které pro každé použití v rámci jednoho souboru představuje inkrementující se číslo. Jelikož ale název proměnné nemůže být číslo, tak je potřeba k němu přiřadit nějaký další řetězec. To zajišťuje makro CONCAT a aby se makro ___COUNTER___ správně zaměnilo za číslo, tak je nutné ho správně expandovat před zřetězením pomocí operátoru ##, jinak by mohl vzniknout název proměnné ___registrator_____COUNTER___ místo očekávaného ___registrator__0.

Výsledný název implementace (modulu) v obslužném programu tvoří argument name tzn., že samotné jméno třídy není nijak dále využito.

6.1.3 Simulation

Je bázovou třídou pro třídy uživatelských implementací, které z ní dědí a částečně ji implementují, a zajišťuje rozhraní pro ovládání z GLEngine. Dále zajišťuje generování počátečních dat a nastavení simulace podle parametrů. Též implementuje funkce, které mají za cíl oddělit uživatele od nutnosti interakce s obslužným programem, jako například funkce pro ukládání dat, hlášení stavu a měření doby výpočtu.

Toho je docíleno pomocí třídy Registrator. Aby uživatel nemusel pro každou dědičnou třídu s implementací simulace vytvářet nové jméno a přejmenovávat metody konstruktory atd., tak doporučuji uzavřít celou implementaci v anonymním jmenném prostoru (namespace). Aby uživatel nemusel ani měnit název implementace, tak je jméno implementace předávané do registrátoru vytvořeno automaticky při kompilaci ze jména souboru, ve kterém se nachází daná implementace, pomocí vestavěného makra ___FILE_NAME___. Nyní tedy pro vytvoření nové implementace stačí aby každá implementace byla v souboru s unikátním názvem a tudíž pro vytvoření nové implementace stačí zkopírovat soubor s nějakou existující implementací. Zde je vidět příklad použití v kódu výše popsaného způsobu 4.

Algorithm 4 Automatická implementace

```
1 namespace {
2         class SimClass : public Simulation
3         {
4                 ...
5         }
6         REGISTER_USER_SIMULATIUON(SimClass, __FILE_NAME__, "");
7 }
```

6.1.4 Particle, ParticleArray

Particle slouží k uchování stavu částice a typické použití je pole objektů této třídy. ParticleArray obsahuje několik polí členů třídy Particle.

6.1.5 TypeV

Implementace vektorové matematiky, pro 3 prvkový vektor. implementuje operace jako vektorový součet, součin, vynásobení vektoru hodnotou, skalární součin, délku vektoru, normalizaci atd.

Pro zjednodušení jsou definovány typy FloatV, IntV a DloubleV.

Tato třída také pokud je kompilována pomocí překladače, který definuje makro ___CUDACC___ (což je typicky CUDA překladač), tak třída přidává všem metodám typ ___HOST___ a ___DEVICE___, čím jsou označeny pro kompilaci jak na hostitelský systém, tak na grafickou kartu.

6.1.6 Storage

Slouží pro předávání dat ze simulace dál. Vždy uchovává pozice barvy částic z posledního kroku, protože z této třídy si tyto data za běhu asynchronně kopíruje GLEngine aby je mohl vykreslit. Pokud je nastaveno, tak tato třída ukládá každý krok simulace na disk ve formátu HDF5. Dále také shromažďuje dobu výpočtu kroku běhu simulace, pro všechny kroky.

6.1.7 Shader

Třída zastřešující práci s OpenGL shadery, jako je jejich kompilace a zvolení.

6.1.8 ShaderBuffer

Další pomocná třída pro GLEngine, která obaluje OpenGL buffery.

6.2 Implementace úloh

Jako výchozí ukázková je implementace pro 1 jádro obyčejného procesoru. Implementace jsou tvořeny vždy jen jedním zdrojovým souborem.

Základní kostru tvoří 2 virtuální metody metody SIMULATION::USERRUN() a SIMULATION::USERSTEP() a lze ho vidět v algoritmu 5. Všechna potřebná vstupní data jsou k nalezení v bázové třídě SIMULATION a jsou to členové MPARTICLES (alternativně MALTPARTICLES), MTIME, MINITSETT a MLIVESETT. Funkčnost doplňují metody, které musí uživatel použít:

- userSignalStart() - signalizuje ovládacímu programu úspěšný start implementace
- userSignalStepBegin() - nastaví časomíru pro měření doby výpočtu
- userSignalStepEnd() - zastaví časomíru, spočítá dobu výpočtu, spustí časomíru, předá data do třídy STORAGE, v případě že si to žádá ovládací program, tak pozastaví běh simulace, a pokud neběží benchmark, tak znovu spustí a tím resetuje časomíru
- userStopRequested() - vrací hodnotu bool, zda má uživatelská implementace ukončit svůj běh
- userSignalFinish() - signalizuje třídě STORAGE, že má uzavřít výstupní soubor a signalizuje ovládací aplikace, že implementace úspěšně končí svůj běh.

Algorithm 5 Kostra implementace

```
1  void SimClass::userRun()
2  {
3          userSignalStart();
4          while (mTime < mInitSett.mEndTime)
5          {
6                  userSignalStepBegin();
7                  userStep();
8                  userSignalStepEnd();
9                  if(userStopRequested())
10                         break;
11         }
12         userSignalFinish();
13 }
14
15 bool SimClass::userStep()
16 {
17         mTime += mLiveSett.mDT;
18     return mTime < mInitSett.mEndTime;
19 }
```

V metodě userRun() se očekává, že uživatel připraví svoji simulace, jako například alokuje paměť na GPU atd. V metodě userStep() se očekává jeden krok simulace.

Před voláním userSignalStepEnd() musí uživatel mít pozice částic aktualizované v poli mParticles->mPosArray, ze kterého tato metoda přebírá data a předává je dál. Volitelně může uživatel využít mParticles->mColorsArray, ve kterém může pro příslušné částice nastavit barvu v ovládacím programu. Toto pole je tvořeno vektory 3 float hodnot s očekávanými hodnotami <0,1>. Složky vektoru mX, mY, mZ odpovídají RGB barvě.

Opáčko

Systém tvoří částice uspořádané do 3D mřížky ve tvaru krychle. Pozice částic v systému se nikdy nemění. Mění se pouze jejich rychlost a poloha v prostoru.

- $\check{C}_i$ je částice s globálním indexem i
- $\overline{P}_i$je 3D vektor polohy v systému částic i a $\overline{P}_i$lze vždy převádět mezi sebou
- $\overline{V}_i$ je 3D vektor rychlosti částice (myšleno částice i)
- $\overline{X}_i$je 3D vektor polohy v prostoru částice
- H je počet částic v hraně krychle, tzn. jak velký je jeden rozměr krychle.
- S je rozsah sousednosti a neb kolikátá částice ode mě je ještě můj soused.
- M_i je množina sousedů zvolené částice.
- $\overline{F}$ a $\overline{G}$ jsou 3D vektory zrychlení. $\overline{G} = (0, -9.81, 0)$

- $\triangle t$ je délka jednoho kroku v čase.

6.2.1 Základní procesorová implementace

Je naprogramována ve 4 verzích: pro 1 jádro, pro 2 jádra, pro všechny jádra a pro všechny jádra s využiTF2 tím alternativního pole částic. Slouží jako výchozí bod pro první implementaci na grafickou kartu. Implementace je napsána s ohledem na pozdější paralelizaci. Nalézá se v modulech SimCPU, SimMultiCPU, SimMultiAlt. Díky využití šablon je verze pro 2 a všechny jádra ve stejném modulu.

Implementace následuje algoritmus 2 a v pozměněné podobě implementuje vztahy 5.3. Celý ho implementuje v metodě userStep(). Vztahy jsou lehce zpřeházeny, aby lépe seděly imperativnímu programování.

6.2.1.1 Jako První krok

se provádí euler_step() dohromady s apply_forces(), modifikací vztahů 5.6 a 5.12 jsou implementovány vztahy $\overline{V}'_{i_t} \leftarrow \overline{V}_{i_t} + \triangle t\overline{F} + \triangle t\overline{G}$ a $\overline{X}'_i \leftarrow \overline{X}_{i_t} + \triangle t\overline{V}'_{i_t}$. Výsledek je možné vidět v algoritmu 6.

V poli mParticles->mPosArray[i] se nalézá $\overline{X}_{i_t}$a v mParticles->mPosArray[i] se nalézá $\overline{V}_{i_t}$. V proměnné mLiveSett.mDT je $\triangle t$ a $\overline{F}$ a $\overline{G}$ jsou v mLiveSett.mExternalAcceleratio a mLiveSett.mGravityAccell. Následně se $\overline{X}'_i$ uloží do pomocného pole mParticles->mPosNewArray[i].

Nakonec po zpracování všech částic se přehodí pole mParticles->mPosArray[i] s mParticles->mPosNewArray[i] což odpovídá $\overline{X}_{i_{t+1}} \leftarrow \overline{X}'_{i_t}$. A máme připravené nové polohy částic.

Algorithm 6 euler_step()

```
1 for (int i = 0 ; i < mParticles->mSize; ++i) {
2       mParticles->mVelocityArray[i] += mLiveSett.mDT *
            mLiveSett.mExternalAcceleration + mLiveSett.mDT *
            mLiveSett.mGravityAccell;
3       mParticles->mPosNewArray[i] = mParticles->mPosArray[i] +
            mLiveSett.mDT * mParticles->mVelocityArray[i];
4 }
5 mParticles->swapPos();
```

6.2.1.2 Druhým krokem

je výpočet korekce poloh. To se musí provést pro všechny iterace mLiveSett.mSubsteps, pro všechny částice. Dílčí korekce, které se nakonec sečtou se počítají pro vybranou částici a všechny její sousedy. Sestává se z kroků compute_correction() a apply_correction() algoritmu 2. Základem je iterování nad sousedy $\check{C}_j$ částice $\check{C}_i$. To je provedeno tak že se ve 3 vnořených cyklech vybírá sousední částice podle polohy $\overline{P}_i$ v systému, ve 3 dimenzích x,y,z. Pro lepší představu lze průchod nad sousedy možno vidět v algoritmu 7 s tím že mLS.mP je zkrácenina pro mLiveSett.mParticleNeighbours.

Algorithm 7 iterace nad sousedy

```
1  for (int pIdx = 0 ; pIdx < mParticles->mSize; ++pIdx) {
2          auto [x, y, z] = fromGId(pIdx, mInitSett.mCubeSide);
3          for (int dx = -mLS.mP; dx <= mLS.mP; ++dx) {
4                  for (int dy = -mLS.mP; dy <= mLS.mP; ++dy) {
5                          for (int dz = -mLS.mP; dz <= mLS.mP; ++dz) {
6                                  int nIdx = toGId(x + dx, y + dy, z + dz,
                                       mInitSett.mCubeSide);
7                                  ...
8                          }
9                  }
10         }
11 }
```

Iterujeme tedy nad dvojicemi částice a její soused a tím počítáme sumu dílčích korekcí, což odpovídá vztahu 5.11. Jedna dílčí korekce je tedy dána vztahem $\lambda(\overline{X}_i, \overline{X}_j)\nabla C(\overline{X}_i, \overline{X}_j)$. Výpočet λ je přímo začleněn do kódu a přesně odpovídá vztahu 5.10.

Funkce omezení C ze vztahu 5.8 je implementována v pomocné funkci computeConstrain() s tím, že aby přesně odpovídala, tak se od jejího výsledku musí odečíst očekávaná vzdálenost částice a souseda.

Funkce gradientu omezení ∇C ze vztahu 5.9 se nachází ve funkci computeConstrainGradient().

Očekávaná vzdálenost souseda se počítá přesně podle vztahu 5.7.

Nakonec po spočtení korekce polohy provede $\overline{X}_i' \leftarrow \overline{X}_i + \frac{\triangle \overline{X}_i}{|M_i|}$, tzn. nové polohy se uloží do mParticles->mPosNewArray. $|M_i|$je počet sousedů částice. Nejjednodušším způsobem jak to spočítat je mít proměnnou a tu inkrementovat při procházení sousedů.

Po korekci všech částic se opět prohodí mParticles->mPosNewArray s mParticles->mPosArray.

6.2.1.3 Třetím krokem

je aktualizace rychlosti euler_finish(). Úpravou vztahu 5.12 (protože některé kroky jsou prováděny v kroku 6.2.1.1 a 6.2.1.2) vznikne $\overline{V}_{i_t} = \frac{\overline{X}_{i_t} - \overline{X}_{i_{t-1}}}{\triangle t}$. Dále je v v tomto kroku jsou uloženy polohy částic $\overline{X}_{i_t}$ z mParticles->mPosArray do dalšího pomocného pole mParticles->mPosStartArray, aby se mohly v dalším cyklu použít jako $\overline{X}_{i_{t-1}}$.

6.2.1.4 Čtvrtý krok

implementuje ošetření hranic prostoru check_borders(), a to jednoduchým způsoben, že pokud je částice mimo prostor, tak se vrátí zpět a obrátí se její rychlost a sníží modifikátorem simulující tření.

6.2.1.5 Pátý krok

compute_energy() je prostá suma podle vztahu 5.13.

Pomocné funkce Pro kompletní funkčnost bylo potřeba ještě vytvořit 2 pomocné funkce, fromGId(), která bere globální polohu částice v systému a vrací trojici int hodnot reprezentující 3D vektor polohy v systému.

Všechny tyto kroky jsou implementovány v metodě userStep() a metoda userRun() zůstává ve výchozím stavu jako lze vidět v algoritmu 5.

6.2.2 Naivní implementace a dekompozice

Modul GNaive

Naivní implementace GNaive si jako základ bere modul SimCPU. Využívá ruční správu paměti jinak nepřináší žádné vylepšení.

Nejprve je nutné rozdělit výpočet na jednotlivé kernely podle toho jak jsou na sobě datově závislé tzn. kdy bude potřeba synchronizace nad všemi bloky vláken. K tomu velice dobře poslouží rozdělení podle kroků ze základní implementace. Ve výsledku tedy každý krok má vlastní kernel a pokud v tom kroku se volá funkce mParticles->swapPos() tak to je nutné provést mimo kernel. Pokud se tato funkce volá v cyklu, tak je nutné i celý cyklus naprogramovat mimo kernel. Proces přeprogramování do kernelů je velmi přímý, protože třída TypeV je již připravena na použití v GPU a tak je možné všechny výpočty nad typy int, float, FloatV, IntV atd. používat z nezměněné podobně.

Jednotlivé kroky z SimCPU jsou rozděleny do kernelů cudaEulerStep(), cudaSubStep(), cudaStepFinish() a calculateEnergy().

Kernel cudaEulerStep() přímo implementuje *První krok* 6.2.1.1.

Kernel cudaSubStep() implementuje jednu iteraci cyklu for (int substep = 0; substep < mLiveSett.mSubsteps; ++substep)... z *Druhého kroku* 6.2.1.2. Je to tak protože se v tomto cyklu nachází volání mParticles->swapPos().

Kernel cudaStepFinish() implementuje *Třetí krok* 6.2.1.3 a *Čtvrtý krok* 6.2.1.4.

Kernel calculateEnergy() implementuje krok 6.2.1.5 pomocí funkce atomicAdd().

Všechny kernely jsou téměř shodné s kódem ze základní implementace, jen místo použití ukazatelů mParticles->... jsou využity ukazatele z argumentů kernelu.

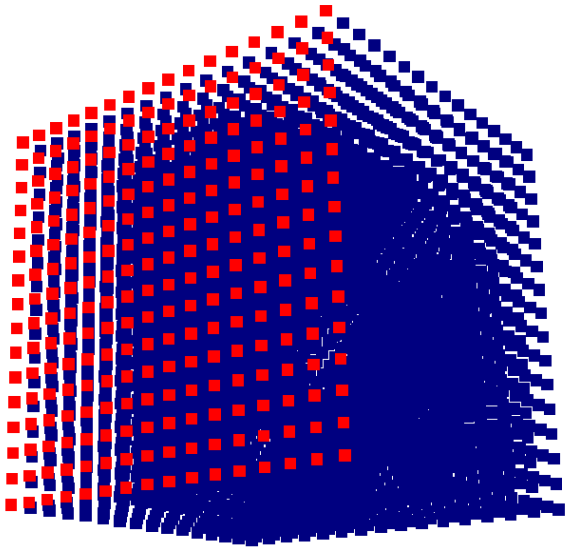
V metodě userStep() se tyto kernely volají. Dekompozice vláken je 2D tzn. že každému bloku vláken je přidělena jedna rovina z kostky částic. To přináší výhody i nevýhody. Výhodou je že tato dekompozice je velmi jednoduchá a neplýtvá vlákny protože se velmi dobře přizpůsobuje velikosti roviny. Jelikož ale kernely nepočítají s tím, že by jedno vlákno obsluhovalo více než jednu částici, tak je velikost dat omezená na počet vláken v bloku a tudíž není možné zpracovat například kostku o hraně $H > 32$ (záleží na limitu karty). Dekompozici kostky $H = 16$ lze vidět na obrázku 6.3, kde červené body označují jeden blok vláken obsluhující své částice. Modrou barvou jsou značeny všechny ostatní vlákna a jejich částice.

V metodě userRun() se provádí alokace a kopírování dat na grafickou kartu pomocí funkcí cudaMalloc() a cudaMemcpy(). Uvnitř smyčky se po volání userStep() kopírují data z karty zpět do mParticles->mPosArray, aby se mohly předat dále do ovládacího programu.

Algoritmus metody userStep() je lze vidět v algoritmu 8.

Modul SimCUDA_01

Implementace SimCUDA_01 je také téměř shodná s Naivní implementací, ale využívá unifikovanou paměť, díky které je potřeba držet pouze jeden ukazatel a program na po-



Obrázek 6.3: 2D dekompozice vláken

Algorithm 8 GNaive:::userStep()

```
1  userStep():
2          cudaEulerStep<<<>>>()
3          sync and swap
4
5          for all substeps:
6                  cudaSubStep<<<>>>()
7                  sync and swap
8
9          cudaStepFinish<<<>>>()
10
11         calculateEnergy<<<>>>()
```

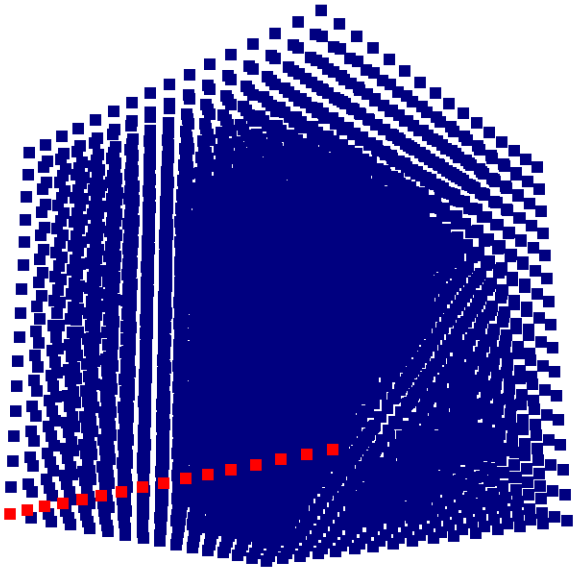
zadí sám zajistí použití ukazatele do paměti správného zařízení a také synchronizaci mezi zařízeními.

Modul GNaiveAlt

Implementace GNaiveAlt je téměř stejná jako *Naivní implementace,* ale místo polí členů částice mParticles, využívá pole částic mAltParticles. Díky tomu je potřeba spravovat méně ukazatelů. Nevýhodou je místo prohazování ukazatelů nutnost ručně kopírovat data z mAltParticles->mPosNew do mAltParticles->mPosNew pomocí dalšího kernelu.

Modul GNaive1D

Tato implementace je totožná s GNaive, jen místo 2D dekompozice používá 1D dekompozici. Největší nevýhodou je že až do velikosti kostky $H < 32$ nedokáže v bloku dat mít dost částic na užití všech vláken ve warpu.



Obrázek 6.4: 1D dekompozice vláken

Modul GNaiveTC

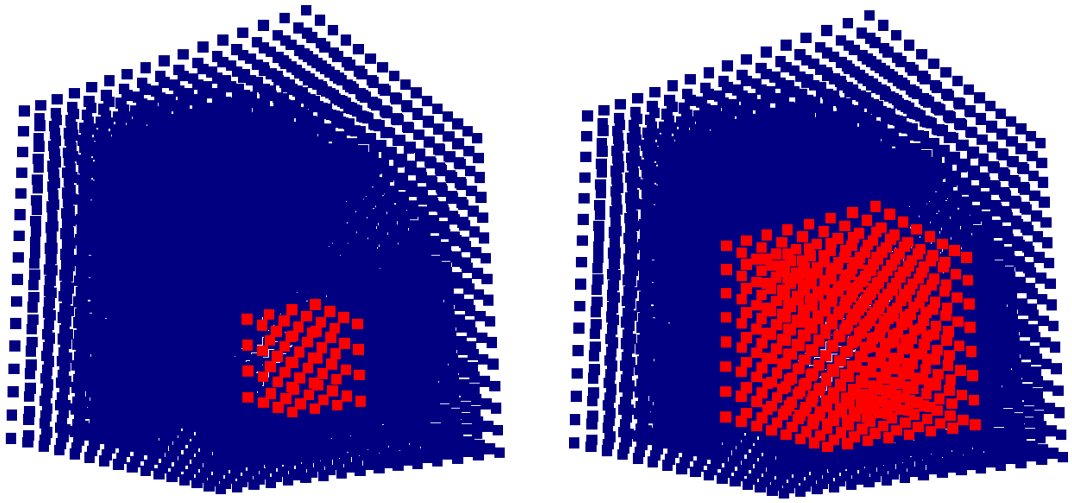
Implementace GNaiveTC vychází z *Naivní implementace* a modifikuje pouze dekompozici v metodě userStep(). Místo 2D dekompozice je zde použita 3D dekompozice ve tvaru subkostek, které dohromady složí kostku částic. To je možné vidět na obrázku 6.5 pro kostku $H = 16$, kde opět červená barva značí jeden blok vláken.

Výhodou této dekompozice je jednodušší zpracování více dat než je vláken v bloku, protože na rozdíl od 2D dekompozice se zde systém částic dekomponuje ve všech 3 směrech a to přes to že každé vlákno obsluhuje pouze jednu částici. Další výhodou by mohla být lepší lokalita dat při přístupu s sousedním částicím. Nevýhodou je, že pokud kostka nebude dělitelná velikostí bloků vláken, tak některé okrajová vlákna nebudou mít co na práci a bude to mrhání výkonem.

6.2.3 Modul GSharedTC

Modul GSharedTC vychází z implementace GNaiveTC, takže využívá 3D dekompozici a přidává použití sdílené paměti. Vzhledem k návrhu kdy sousednost S je uživatelem volitelná bylo vyřešeno tak, že z pohledu bloku vláken byly částice rozděleny do subkostek o velikosti bloku vláken. Poté jsou tyto subkostky postupně načítány do sdílené paměti s tím, že velikost sdílené paměti je určena tak, že se do ní vejde přesně tolik částic jako je vláken v bloku vláken, a vyhodnocovány, tzn., že pokud blok obsahuje alespoň jednoho souseda částice, kterou obsluhuje tento blok vláken, tak je načten do sdílené paměti a zpracován pro výpočet dílčích korekcí. Způsob lze vidět v algoritmu 9.

To ovšem může způsobit zbytečné načítání některých částic. To je patrné v obrázku 6.6, kde $H = 16$ a $S = 2$. Zelenou barvou jsou vyznačeny částice zpracovávané blokem vláken, červenou barvou jsou vyznačeny částice, které jsou sousedy nějaké částice z bloku, ale samy nejsou zpracovávány blokem vláken a šedou barvou jsou vyznačeny částice, které nejsou sousedy, tudíž nejsou použity k výpočtu, ale jsou stále načítány do sdílené paměti a vlákna kterým byla přiřazena stojí.



(a) 4x4x4 (b) 8x8x8

Obrázek 6.5: 3D dekompozice vláken

Další možností by bylo podle sousednosti zvětšovat sdíleno paměť, aby se do ní vešly jak obsluhované částice, tak jejich sousedi. To sebou jiné nevýhody jako složitější načítání do sdílené paměti a jistotu, že při určité velikosti sousednosti bude sdílená paměť nedostatečná.

6.2.4 Odmocnina

Moduly GNaiveTC-Fast, GSharedTC-Fast, GSharedTC-Opt, GSharedTC-Fast-Opt, jsou všechny téměř shodnou kopií s jejich předchůdcem.

Přípona -Fast značí, že místo odmocniny z matematické knihovny, byla použita rychlá inverzní odmocnina ze hry Quake III, která je již implementovávána ve třídě TypeV, kde přidává metody TypeV::lenFast() a TypeV::invLenFast(), které jsou modifikací TypeV::len(). Tyto implementace tedy modifikují pomocné funkce computeConstrain() a computeConstrainGradient() a také v kernelu cudaSubStep() modifikují výpočet očekávané vzdálenosti souseda.

Přípona -Opt značí použití speciální přepínačů při překlad, kterými jsou -prec-div=false a -prec-sqrt=false. Toho je dosaženo modifikací Makefile, přidáním speciálních cílů pro tyto soubory s přepínači. Je to nutné takto udělat, protože se nezadařilo použít tyto optimalizační přepínače pomocí maker přímo ve zdrojovém kódu.[2]

6.2.5 Redukce

Moduly GSharedTC-F-Reduction, GSharedTC-F-Reduction_2 vycházejí z GSharedTC-Fast a modifikují kernel calculateEnergy().

GSharedTC-F-Reduction provádí redukci pomocí atomických operací nad sdílenou pamětí o velikosti 1 float.

GSharedTC-F-Reduction_2 implementuje algoritmus *Sequential Addressing*[16] tak, že každé vlákno nahraje do sdílené paměti spočítanou energii částice, následuje synchroni-

[2] https://gcc.gnu.org/onlinedocs/gcc/Function-Specific-Option-Pragmas.html#Function-Specific-Option-Pragmas

Algorithm 9 Sdílená paměť

```
1  cudaSubStep():
2         loadBlock() #load my subcube
3         localIter() #process my subcube
4
5         syncthreads()
6
7         globalIter() #iterate over all subcubes
8
9         apply_correction()
10
11 globalIter():
12        for all others subcubes:
13                if contain neighour:
14                        loadBlock() #load it
15                        localIter() #process it
16                        syncthreads()
17
18 localIter():
19        for all particles: #in loaded subcube
20                if it is neighbour:
21                        computeStepLocal() #compute it
```

zace vláken a pak cyklus o počtu iterací $log_2(T)$, kde T je počet vláken v bloku vláken, ve kterém se v každém cyklu půlí počet aktivních vláken a aktivní vlákna přičtou do svého políčka ve sdílené paměti hodnoty z políčka neaktivního vlákna, které je na pozici aktivní vlákno + počet aktivních vláken.

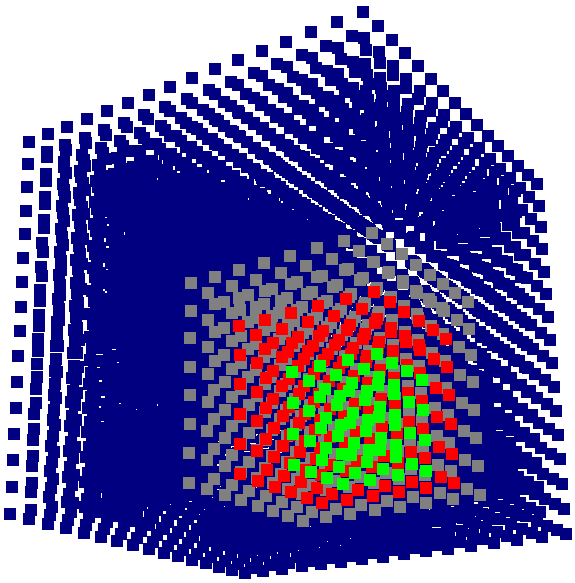
6.2.6 Více GPU

Modul GNaive-MultiG implementuje použití 4 grafických karet zároveň. Jako dekompozice byla zvoleno rozdělení dat podle osy Z což je znázorněno na obrázku 6.7, kde každá barva značí částice které obsluhuje jedna grafická karta. Tato dekompozice byla zvolena z důvodu jednoduchého sdílení dat mezi kartami, protože částice pro každou kartu jsou v paměti spojitě za sebou. Dekompozice částic v rámci jedné karty vychází z implementace GNaive1D. Vzhledem k poměrně malé paměťové náročnosti simulace, byl zvolen přístup, kdy každá karta má kompletní kopii celé simulace, ve které si počítá vlastní část a tu po dokončení výpočtu rozesílá mezi ostatní karty.

Všechny kernely dostaly nový argument zOffset, který se přičítá ke globální pozici vláken v ose Z. Tento argument je volen podle čísla karty. Jinak kernely nebyly nijak oproti GNaive1D.

Spouštěni kernelů, kopírování paměti atd. je dosaženo pomocí smyček a opakovaného volání cudaSetDevice() , které nastaví určitou kartu jako aktivní. Alternativní možností by bylo použit více procesů nebo procesorových vláken.

Více karet znamená více nutnost držet více sad ukazatelů do paměti (každá karta má svoji) a také režii navíc. Dále je nutné zjistit které karty mohou spolu komunikovat napřímo přes NVLink pomocí funkce cudaDeviceCanAccessPeer() a také tuto možnost



Obrázek 6.6: Sdílená paměť

aktivovat pomocí cudaDeviceEnablePeerAccess(). U dvojic karet které toto spojení podporují je poté možné pro kopírování paměti místo cudaMemcpy() použít cudaMemcpyPeer().

Metoda userRun() se dočkala pouze menší změny kdy alokace paměti jsou dělány pro každou grafickou kartu. Pro každou kartu se hledají všechny NVLink spojení a pro každou kartu je drženo na jakém offsetu obsluhuje částice a jejich počet, protože dekompozice mezi karty nemusí být vždy rovnoměrná.

Pro tyto informace byla zavedena nová struktura Gpu, ve které jsou všechny tyto informace a existuje instance pro každou použitou kartu.

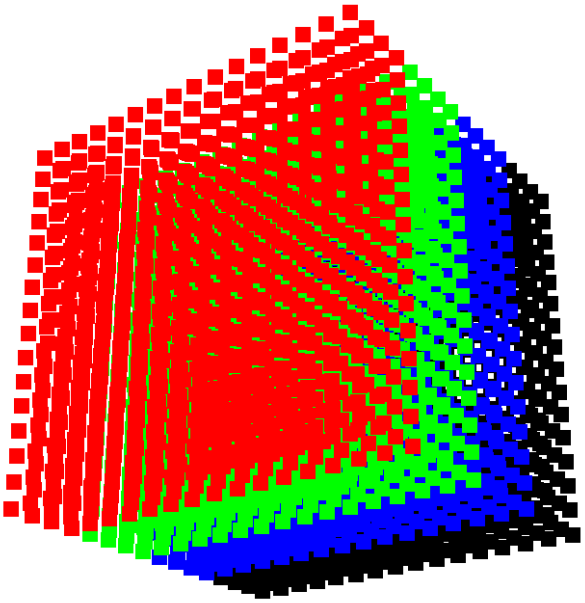
Pro distribuci dat mezi kartami byly vytvořeny 2 nové pomocné funkce: copyAllToAll() a copyFinish(). Jelikož kopírování paměti pomocí cudaMemcpy() synchronní, tak aby bylo možné přenášet data mezi všemi kartami zároveň, tak bylo nutné využít asynchronní kopírování paměti pomocí cudaMemcpyAsync() a cudaMemcpyPeerAsync(). Po jejich použití je ale nutné explicitně synchronizovat jim přiřazený stream a to zajišťuje funkce copyFinish(). Funguje to tedy tak, že copyAllToAll() spustí přenosy mezi všemi kartami a copyFinish() počká na jejich dokončení.

Metoda userStep() je modifikovaná o něco více zásadní změny oproti algoritmu 8 je možné vidět na algoritmu 10:

V rámci implementace byla vyzkoušeno zda by nešlo použít pro všechny 4 karty jednu unifikovanou paměť a ne 4 kopie. Nešlo protože rychlost výpočtu byla třeba 100x menší a navíc simulace dávala na první pohled chybné výsledky.

6.2.7 OpenMP

Modul SimGpuMP je postaven na SimMultiCPU a přidává potřebné informace pro kompilátor, aby mohl přenést výpočet na akcelerátor jímž je v našem případě grafická karta. Všechny výpočetní smyčky, které odpovídají krokům v základní implementaci a kernelům



Obrázek 6.7: 4xGPU dekompozice

v ostatních byly dekorovány direktivou #pragma omp target teams distribute parallel for, která působí následovně:

omp target: určuje, že následující blok kódu se má vykonat na "target" zařízení, což je obvykle GPU. Pokud není k dispozici žádné target zařízení, kód se vykoná na hostitelském zařízení (tj. CPU).

teams: tato direktiva vytváří skupinu "týmů" vláken, které mohou být vykonány paralelně. Každý tým je nezávislý a má svůj vlastní vláknový tým. Počet týmů, které jsou vytvořeny, závisí na implementaci a může být ovlivněn klauzulí num_teams. Toto odpovídá blokům vláken a počtům vláken v bloku.

distribute: direktiva rozděluje iterace následujícího cyklu for mezi týmy vláken. Každý tým dostane podmnožinu iterací k vykonání.

parallel for: určuje, že iterace cyklu for v rámci každého týmu se mají vykonávat paralelně. Každé vlákno v týmu dostane podmnožinu iterací k vykonání.

V místech kde je potřeba, tak je toto "hlavní" makro doplněno o direktivy map(to: alpha2), map(to: mLiveSett) a map(to: mInitSett), které nasdílí tyto datové položky na zařízení.

Pro redukci při výpočtu energie systému částic je použita direktiva map(from: tmpE) reduction(+:tmpE), která vykoná sumu do nasdílené proměnné tmpE.

Hlavní data, tzn. data částic jsou nasdílena na zařízení v mimo smyčku v metodě userRun() pomocí direktiv jako například tato: #pragma omp target enter data map(to: mPosArray[0:size]). Toto makro způsobí že až do jeho protějšku #pragma omp target exit data map(delete: mPosArray[0:size]), bude na zařízení pole mPosArray k dispozici.

Po vykonání každého kroku simulace jsou data na hostiteli aktualizována pomocí direktivy #pragma omp target update from(mPosArray[0:size]), aby se mohla simulace předat dál do ovládací aplikace.

Algorithm 10 GNaive-MultiG:::userStep()

```
1  userStep():
2          for all gpus:
3                  cudaEulerStep<<<>>>()
4          for all gpus:
5                  sync and swap
6          copyAllToAll();
7          copyFinish();
8
9          for all substeps:
10                 for all gpus:
11                         cudaSubStep<<<>>>()
12                 for all gpus:
13                         sync and swap
14                 copyAllToAll();
15                 copyFinish();
16
17         for all gpus:
18                 cudaStepFinish<<<>>>()
19         for all gpus:
20                 sync and swap
21         copyAllToAll();
22         copyFinish();
23
24         for all gpus:
25                 calculateEnergy<<<>>>()
```

Kapitola 7

Testování

Testování probíhalo na 3 strojích v tabulce 7.1.

Stroj	Procesor	Mixbench[5] GFLOPS[1][2]	1xGPU/$\sum$CPU
Stařec	AMD A8-7650K @ 4.4Ghz	136.92	
	NVIDIA GeForce GT 1030	1307.87	9.55x
SC-GPU	2x Intel Xeon E5-2620 v3 @ 2.40GHz	779.67	
	4x NVIDIA GeForce GTX 1080	4 * 9019.18	11.57x
SC-GPU2	2x Intel Xeon Silver 4314 @ 2.40GHz	2785.25	
	4x NVIDIA RTX A5000	4 * 29090.83	10.44x

(a) Osazení a výkon

Stroj	CPU	Jader celkem	Vláken celkem	GPU
Stařec	1	4	4	1
SC-GPU	2	12	12	4
SC-GPU2	2	32	32	4

(b) Konfigurace

Tabulka 7.1: Testovací stroje

Typy grafů

Obslužná aplikace nabízí 3 druhy výkonnostních grafů.

Prvním je *obyčejný graf* výkonu, který na ose X znázorňuje počet spočtených omezení. To znamená počet částic krát počet jejich sousedů. Jednoduše se toto číslo dá odhadnout pokud sousednost S je maximální. Tedy pro kostku $H = 32, S = 31$ je počet omezení $O = H^3(S^3 - H)$. Pro sousednost $S = 1$ je přibližná hodnota $O \sim H^3 * 26$. Na ose Y se nalézá doba potřebná pro provedení jednoho kroku simulace v milisekundách.

Druhým typem je *graf zrychlení*. Tento graf podle zvolené výchozí implementace přepočítává zrychlení všech ostatních. Na ose X se nalézají stejné hodnoty jako v *obyčejném grafu* a na ose Y je zrychlení. Zrychlení pro zvolenou výchozí implementaci je tudíž vždy 1-krát.

[1] Byl vybrán vždy nejlepší výsledek

[2] Mixbench testuje všechny procesory, ale pouze 1 grafickou kartu

Třetím grafem je 2D *tepelná mapa* zobrazující závislost mezi velikostí kostky H a sousedností S vždy pouze pro jednu implementaci. Na ose X je je sousednost S a na ose Y je velikost hrany kostky H. Hodnoty které tato mapa zobrazuje je počet spočtených omezení za 1 milisekundu $O * ms^{-1}$. Tudíž tedy zobrazuje rychlost podle kombinace H a S. Neexistující hodnoty jsou vyplněny 0.

Na obrázku C.1 je možné vidět všechny 3 typy grafů z toho obyčejný a zrychlení jsou v log měřítku. Je na nich patrné že se velice klikatí. To je způsobeno tím, že grafy jsou vytvářeny v pořadí testů. Když k tomu přidáme fakt, že hodnota O je tvořena dvěma hodnotami H a S, jejichž kombinace jsou vytvářeny pomocí 2 cyklů, tak hodnota O se mění nahoru i dolů. Toho se lze případně zbavit pokud pro testování použijeme omezení sousednosti. Například na obrázku C.2 lze vidět podobný obyčejný graf a graf zrychlení jako na obrázku C.1, ale jen pro sousednosti *1* a maximální.

7.1 Testování úloh

Pokud nebude řečeno jinak, tak implementace byly testovány pro $\forall H \forall S | H \in < 2, 32 > \wedge S \in < 1, H - 1 >$. Kvůli lepší přehlednosti byly pro *obyčejný graf* a *graf zrychlení* vybrány pouze testy, kde $S = H - 1$, nebo $S = 1$.

Pokud nebude řečeno jinak, tak grafy platí pro stroj SC-GPU2, pro ostatní stroje je možné nalézt grafy v příslušných přílohách. Dále pokud není řečeno jinak, tak grafy jsou v lineárním měřítku.

7.1.1 Základní procesorová implementace

Na obrázku 7.1 je možné vidět výsledky měření v $\log_{10}$ měřítku pro implementace SimCPU, SimMultiCPU s tím že SimMultiCPU-2 je omezeno na 2 procesorová jádra. Z výsledků je patrné že paralelní provedení pomocí OpenMP se velmi rychle dostane do vedení. Při použití všech 32 jader se zhruba maximální výkon dostaví už při $H = 16$ ($O \sim 1.67731e + 07$).

Zrychlení 2 jader oproti 1 je cca 2-násobné. Zrychlení 32 jader proti 1 je cca 20 až 24 krát větší. Při $S = 1$, tudíž nižší aritmetické intenzitě na stejném kusu paměti, je maximální zrychlení jen asi 10-ti násobné a dostaví se až pří větším H.

Z tepelné mapy viditelné na obrázku 7.2 je patrné že více jádrový výpočet je poměrně univerzální a dobře snáší různé hodnoty H a S.

Tento test dává předběžné informace o tom jak vhodná je základní úloha pro paralelizaci.

7.1.2 GNaive

Na obrázku 7.1 je možné vidět porovnání GNaive proti procesoru, kdy dosahuje maximální zrychlení cca 250x oproti SimCPU a až 12-ti násobné zrychlení proti SimMultiCPU. Když to porovnáme s tabulkou strojů 7.1, tak je vidět že toto zrychlení je blízko zrychlení očekávanému. Při $S = 1$, je zrychlení GNaive proti SimMultiCPU pouze asi 4-násobné.

Na obrázku 7.3 můžeme vidět že pro tuto implementaci je ideální sousednost 4 až 8 a také je možné vidět, že na rozdíl od SimMultiCPU je mnohem méně univerzální a efektivní je jen v určitých oblastech.

7.1.3 GNaive1D, GNaiveAlt, GNaiveTC

Na obrázku 7.4 je možno vidět porovnání těchto 3 implementací s GNaive. Jako nejlepší se jeví GNaiveTC se zrychlením až 1.8 a GNaive1D se zrychlením až 1.4 pro vysoké S. Na obrázku 7.5 můžeme vidět zefektivnění výpočtu pro velké velikosti kostky.

Implementace GNaiveAlt ukazuje velmi špatnou výkonnost pro malou i velkou sousednost.

7.1.4 GSharedTC-4, GSharedTC-8

Na obrázku 7.7 jde vidět že tyto 2 implementace jsou velice citlivé na to zda velikost kostky je dělitelná dimenzemi bloku vláken. Tuto tezi podporuje i tepelná mapa GSharedTC-4 na obrázku 7.6, kde lze zřetelně poznat kdy velikost H je dělitelná 4. Zrychlení při použití sdílené paměti oproti GNaiveTC je až 1.74-krát.

V případě $S = 1$ je ovšem výkon s použitím sdílené paměti velmi špatný.

7.1.5 Odmocnina

Jak lze vidět na obrázku 7.8, tak využití optimalizačních přepínačů pro odmocninu a dělení přináší až 50 % výkonu, zatím co využití aproximace odmocniny může vést k poklesu výkonu až o 25 %.

7.1.6 Redukce

Jak lze vidět na obrázku 7.8, tak využití pokročilé redukce v implementaci GSharedTC-F-Reduction_2 nemá téměř žádný vliv na rychlost.

Implementace GSharedTC-F-Reduction se ukázala jako nefunkční.

7.1.7 Více GPU

Tento test byl proveden pro nastavení $\forall HS | H \in < 2, 64 > \wedge S = H - 1$. Na obrázcích 7.9 a 7.10 lze vidět porovnání GNaive1D a GNaive-MultiG, která z ní vychází. Je zajímavé že zrychlení použitím 4 karet je způsobeno propadem výkonu při použití jen jedné karty hned za hranicí $H = 32$. Konečné maximální zrychlení při použití 4 karet je 3.8-krát.

7.1.8 OpenMP

Testování implementace SimGpuMP bylo provázeno řadou problémů. Hlavním problémem je že GCC kompilátor v distribuci nemusí mít podporu pro *nvptx*. To se projevuje tím, že buď kompilátor odmítne kompilaci s chybou, že nepodporuje offload na tento typ zařízení, nebo je tento fakt tiše ignorován a kompilátor úspěšně zkompiluje a slinkuje program, ten ovšem je paralelizován pouze na klasický procesor. Další problém, který se objevil, že při úspěšné kompilaci a spuštění, vlákno ve kterém běží simulace akcelerovaná na grafické kartě, tak z neznámého důvodu ignoruje pokusy o řízení vlákna přes volatilní proměnné. Také nefungovaly pokusy o řízení pomocí atomických proměnných z std::atomic. V Případě pokus o kompilaci pomocí *NVHPC* kompilátoru dojde po spuštění simulace k pádu na chybu "symbol lookup error: ./prog: undefined symbol: GOMP_teams4, version GOMP_5.1".

Na obrázku 7.11 je možné vidět úspěšné měřeni implementace SimGpuMP, která dosahuje velmi špatných výsledku. Ač se může být výkon podivně blízký tomu procesorovému,

tak bylo nezávisle pomocí programu *nvtop*[3] potvrzeno, že tato implementace opravdu využívala grafickou kartu.

7.1.9 Ostatní

Grafy ze strojů *SC-GPU* a *Stařec* je možné nalézt v přílohách D a E jejichž výsledky jsou dost podobné těm ze stroje *SC-GPU2.*

7.2 Vyhodnocení

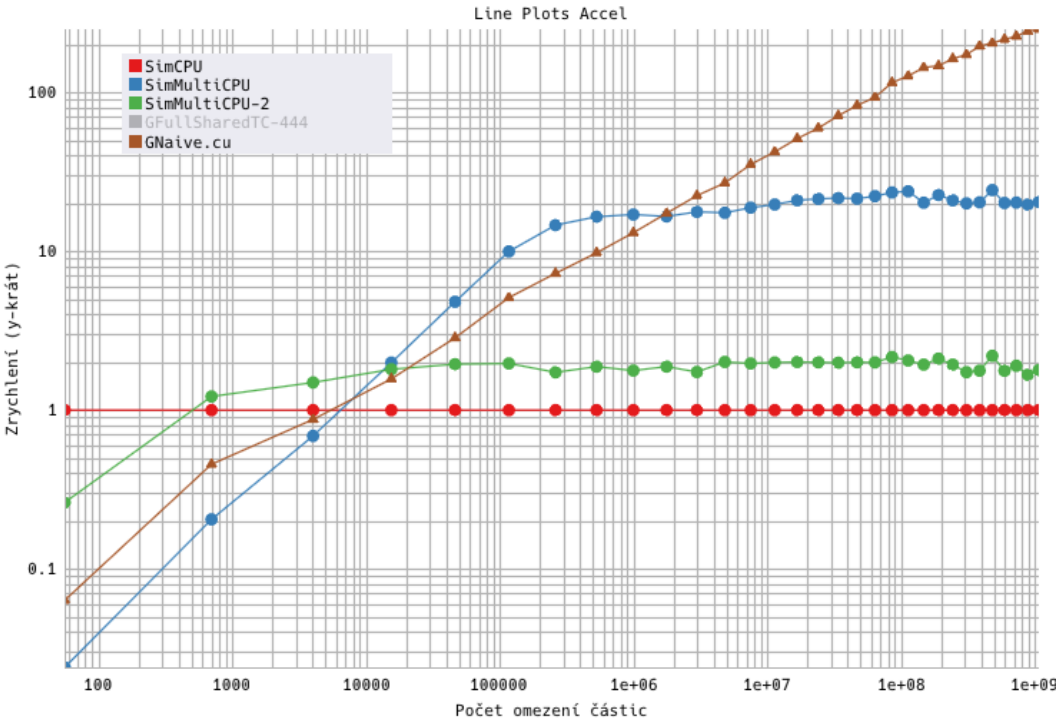
Pomocí akcelerace kódu na jedné grafické kartě se podařilo dosáhnout od 10-ti do 45-ti násobného zrychlení oproti použití všech jader procesoru (ale bez nějakých speciálních optimalizací pro klasický procesor).

Co se samotné optimalizace týče, tak se podařilo dosáhnout zrychlení až 4-násobného proti naivní implementaci. Při využití 4 karet se podařilo v dosáhnout až 3.8-násobného zrychlení.

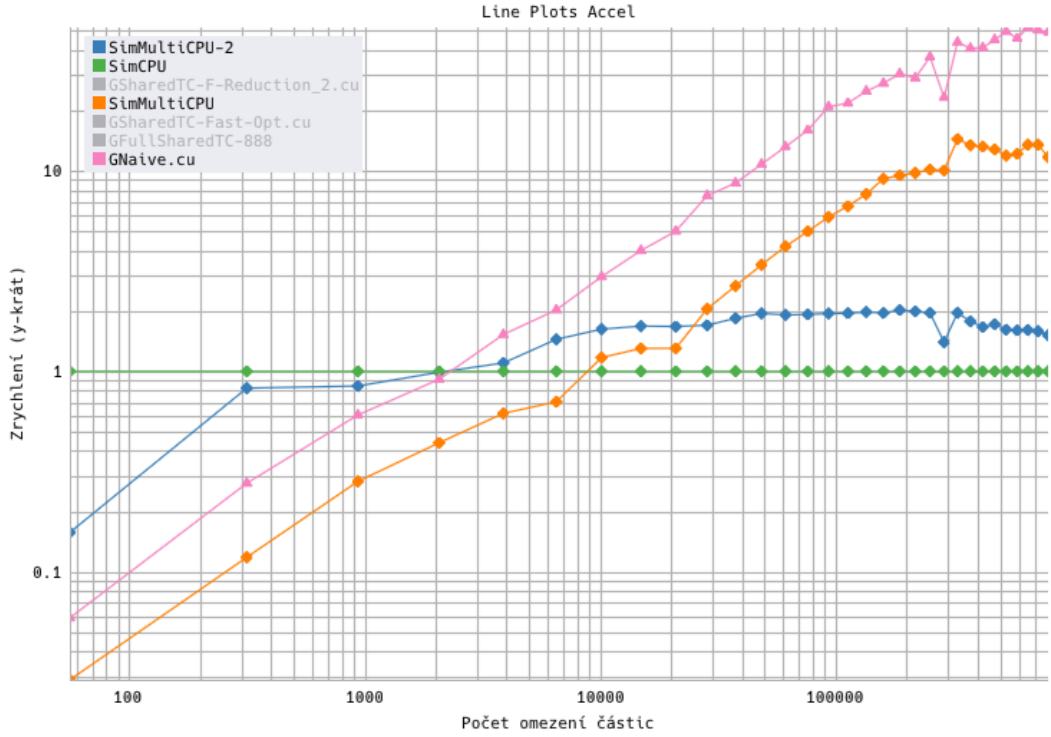
Využití vyššího programovacího jazyka pro programování grafických karet se moc neosvědčilo ani výkonem ani podporou a bezproblémovostí.

Některé úlohy již tak úspěšné nebyly, jako například redukce. Její malý vliv je nejspíše způsoben nedostatečnou náročností výpočtu energie.

[3] *https://github.com/Syllo/nvtop*

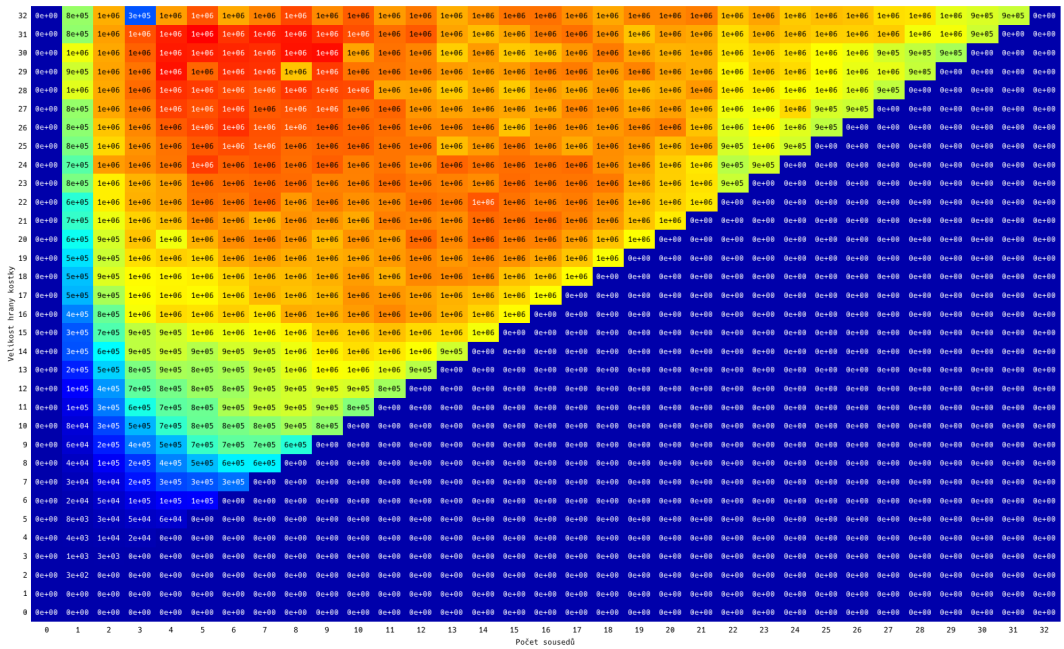


(a) S=H-1

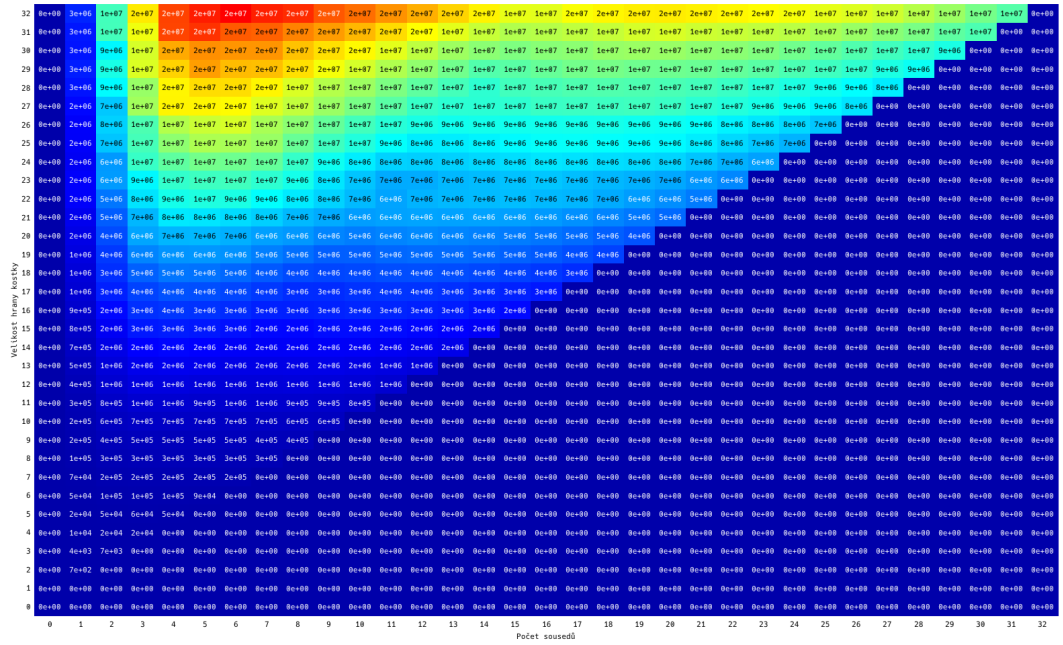


(b) S=1

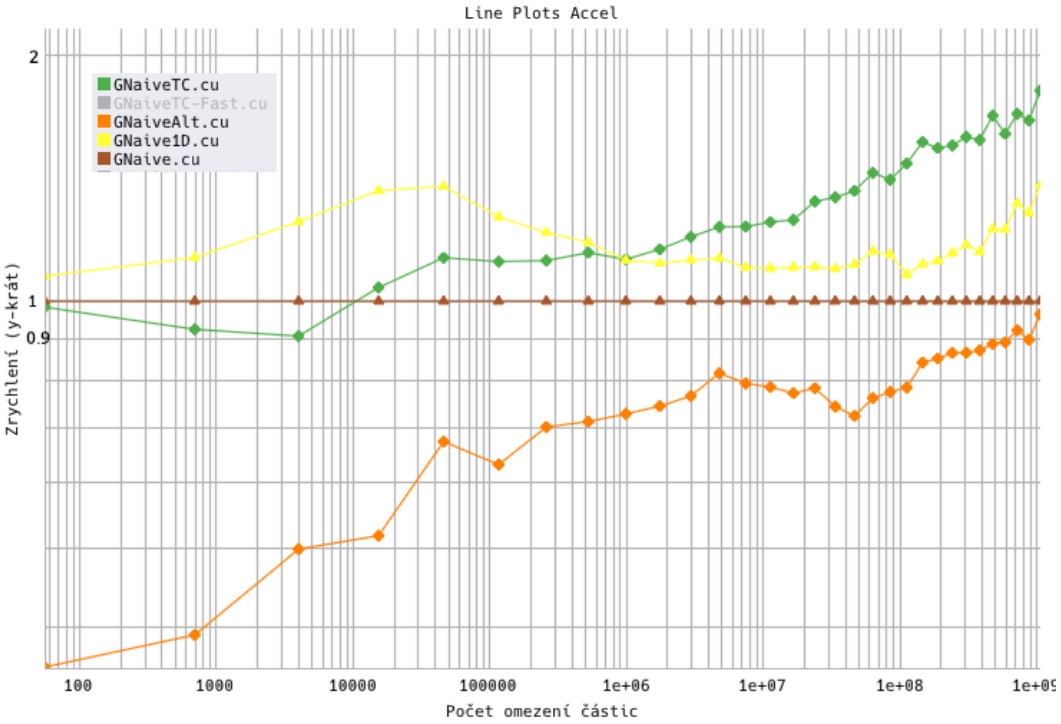
Obrázek 7.1: Zrychlení (SC-GPU2) (log)



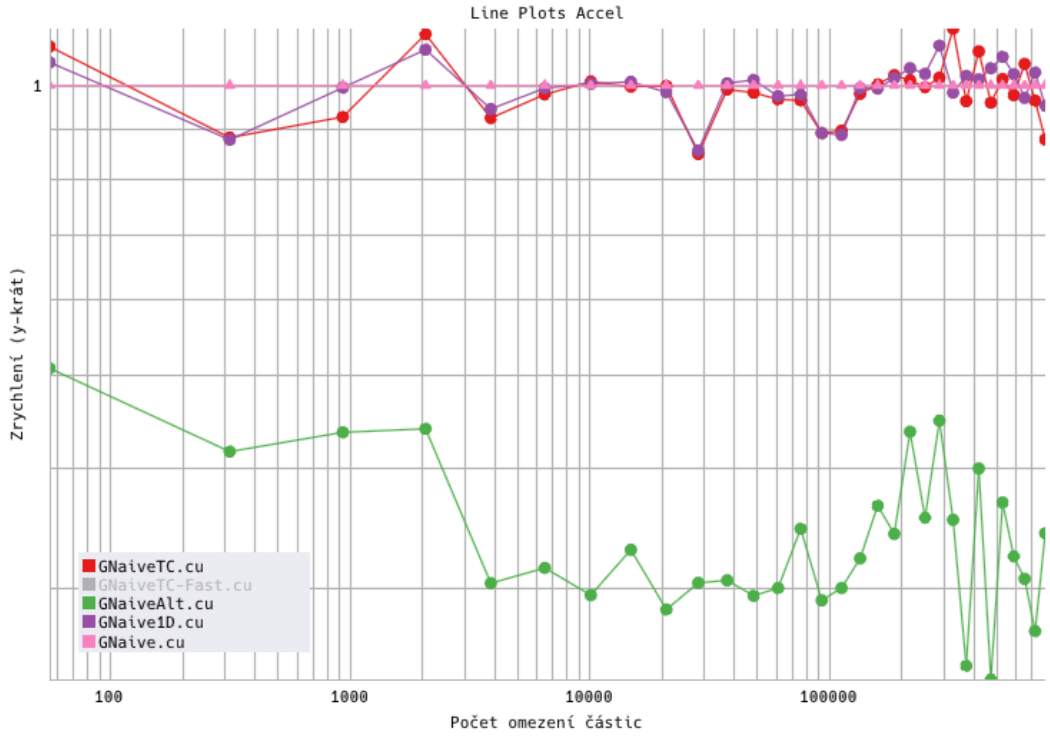
Obrázek 7.2: Mapa SimMultiCPU (SC-GPU2)



Obrázek 7.3: Mapa GNaive (SC-GPU2)

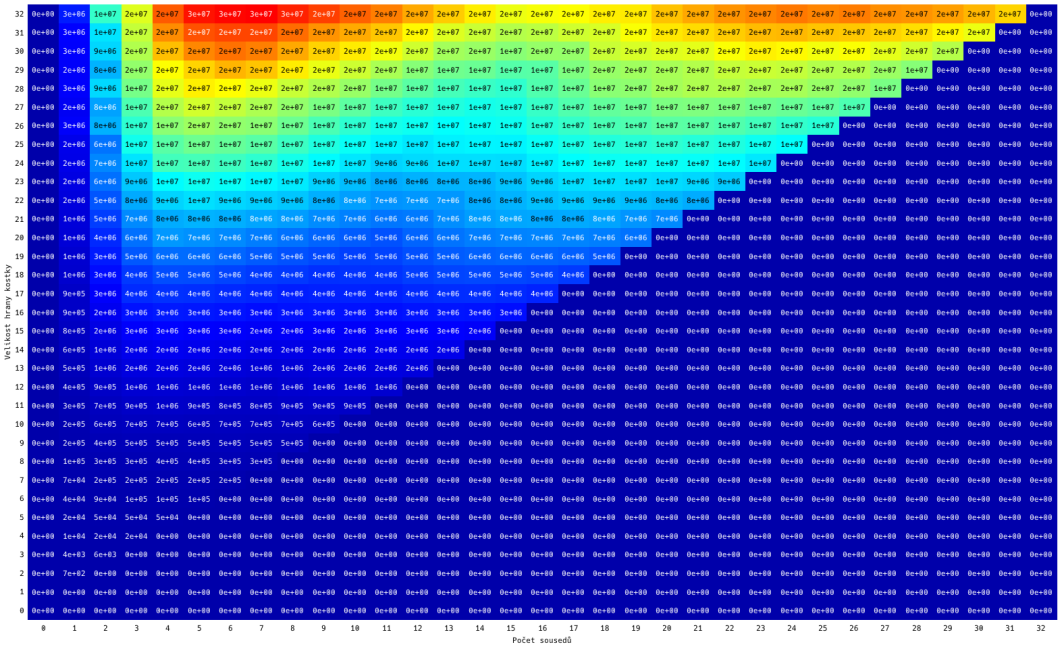


(a) S=H-1

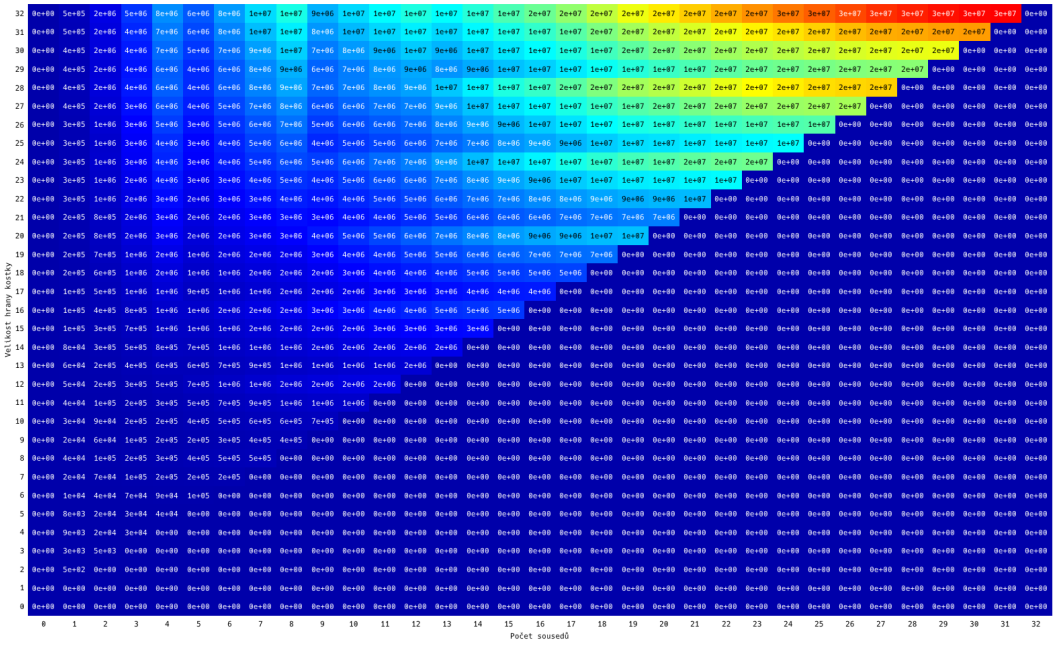


(b) S=1

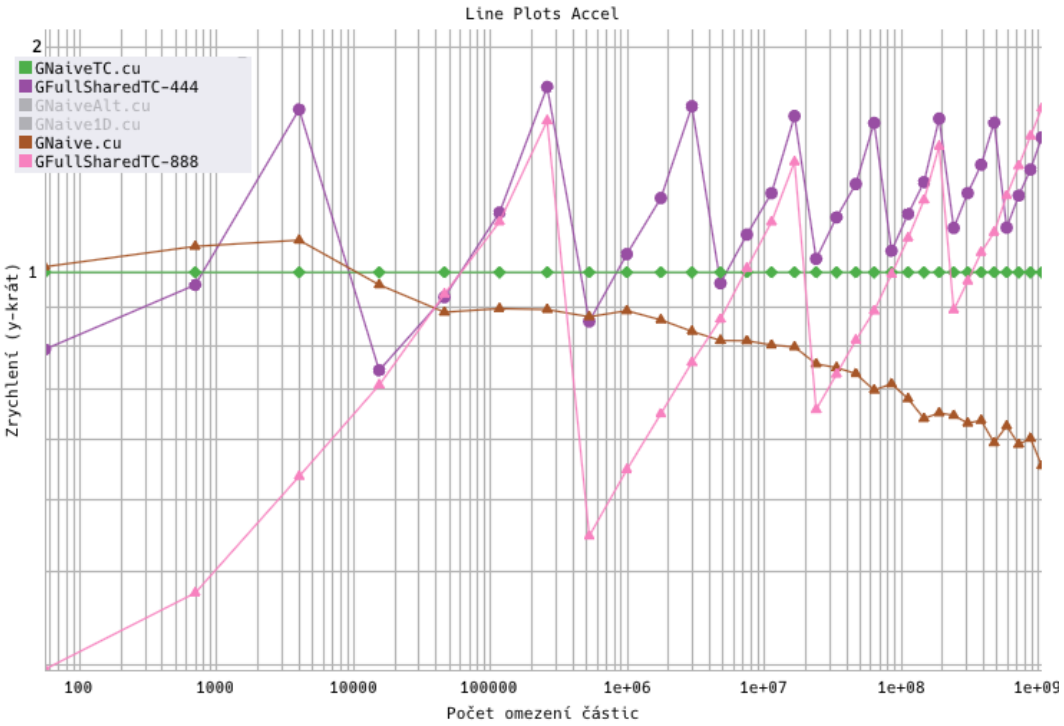
Obrázek 7.4: Zrychlení 2 (SC-GPU2) (log)



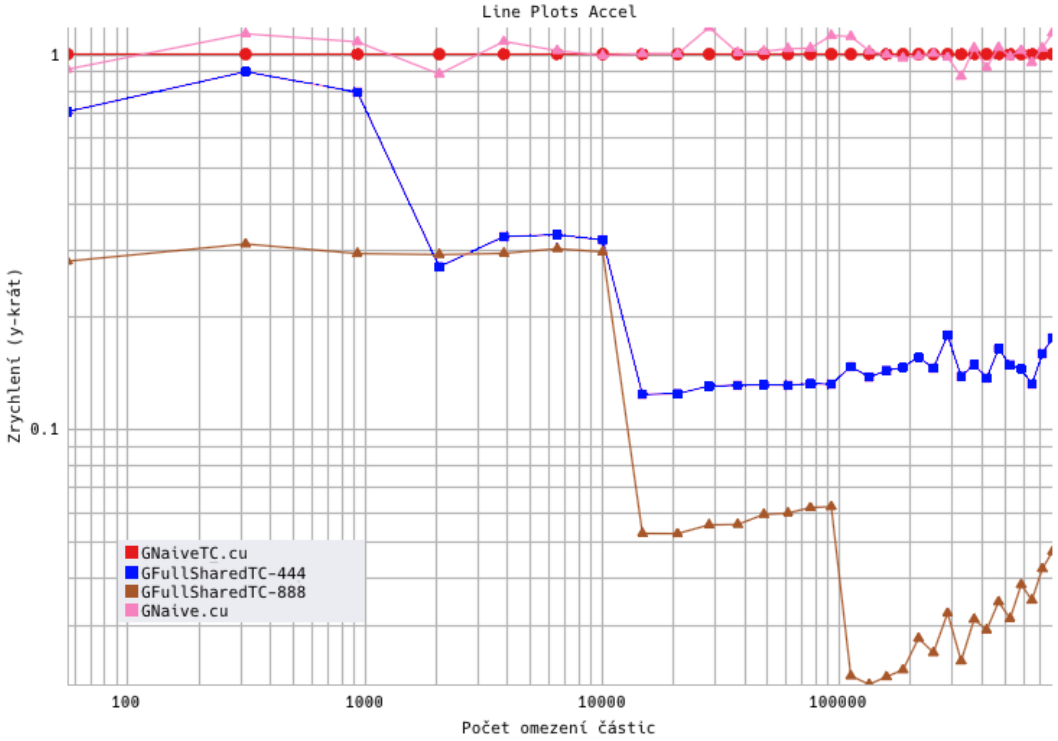
Obrázek 7.5: Mapa GNaiveTC (SC-GPU2)



Obrázek 7.6: Mapa GShared-4 (SC-GPU2)

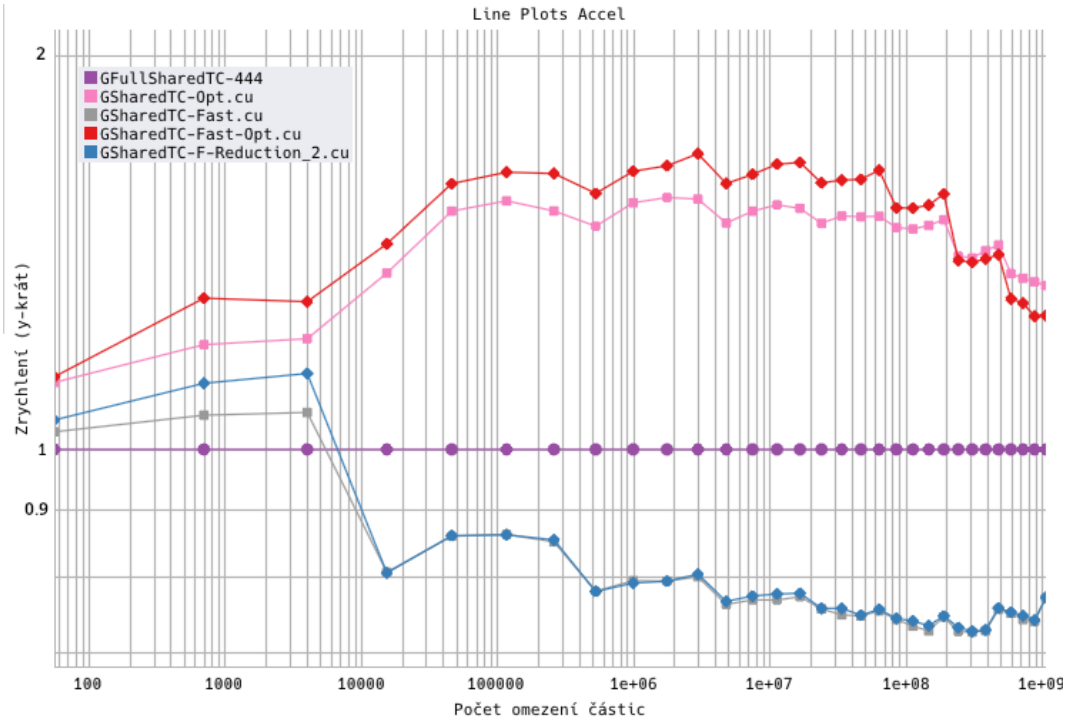


(a) S=H-1

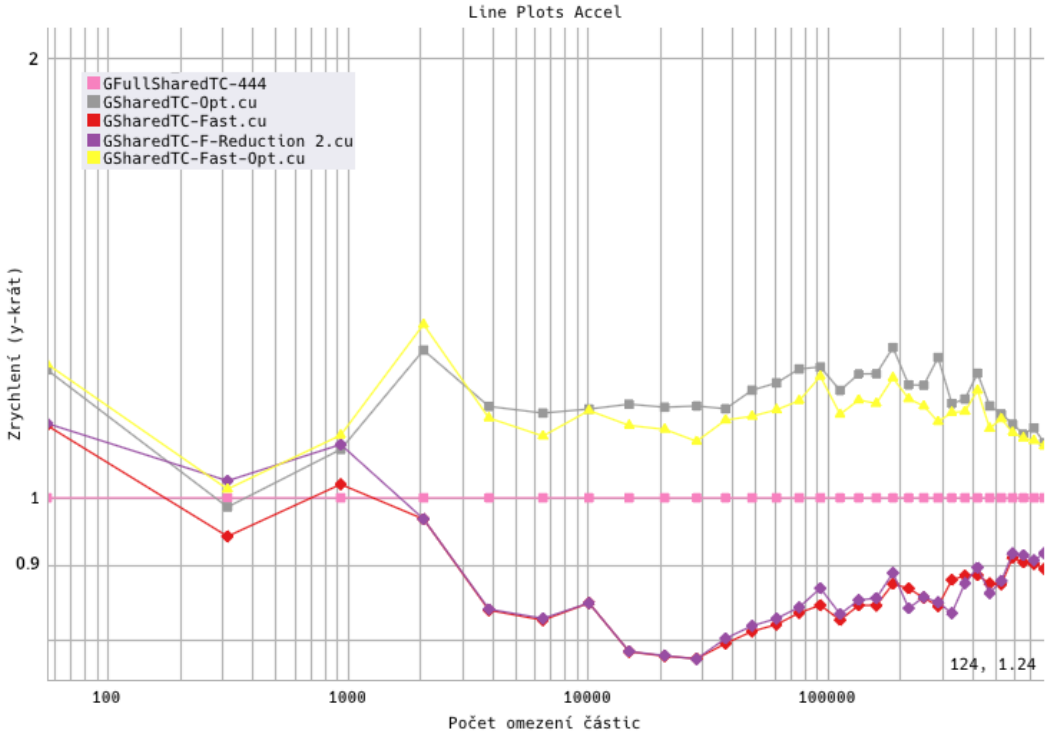


(b) S=1

Obrázek 7.7: Zrychlení 3 (SC-GPU2) (log)

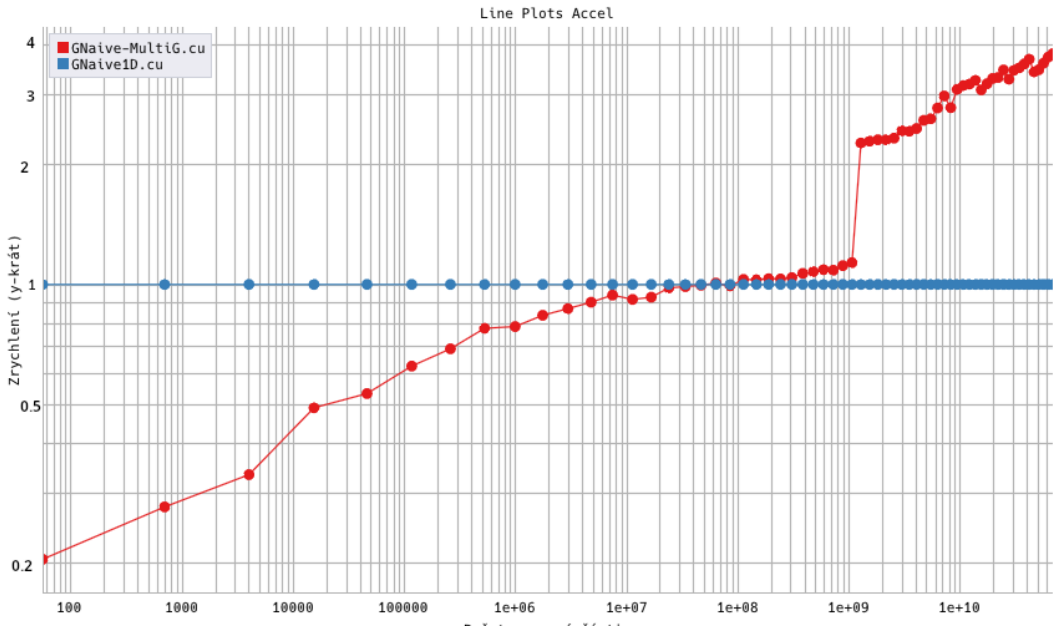


(a) S=H-1

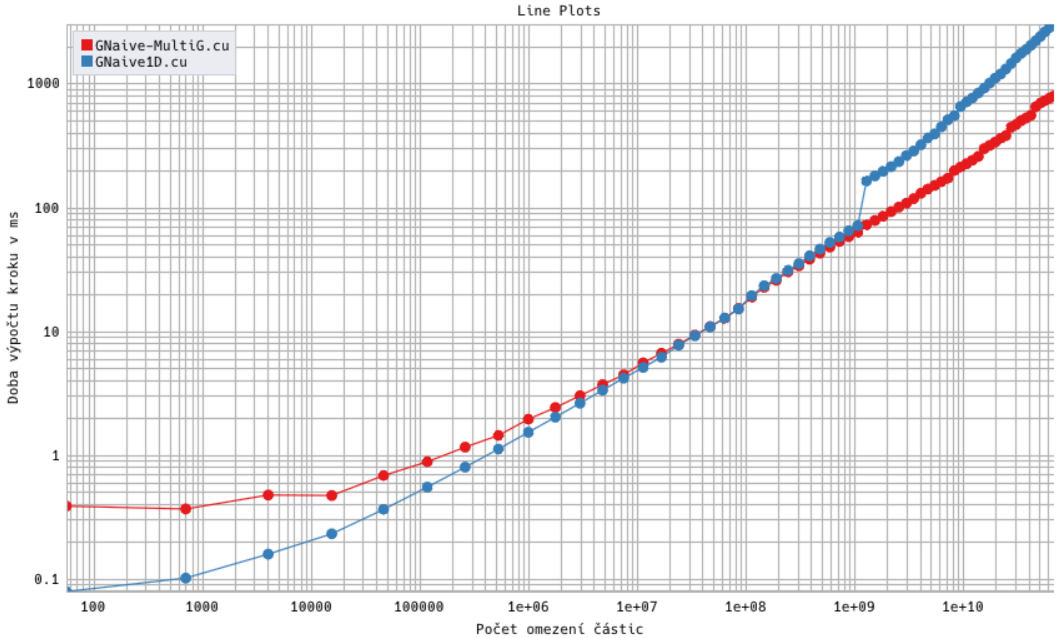


(b) S=1

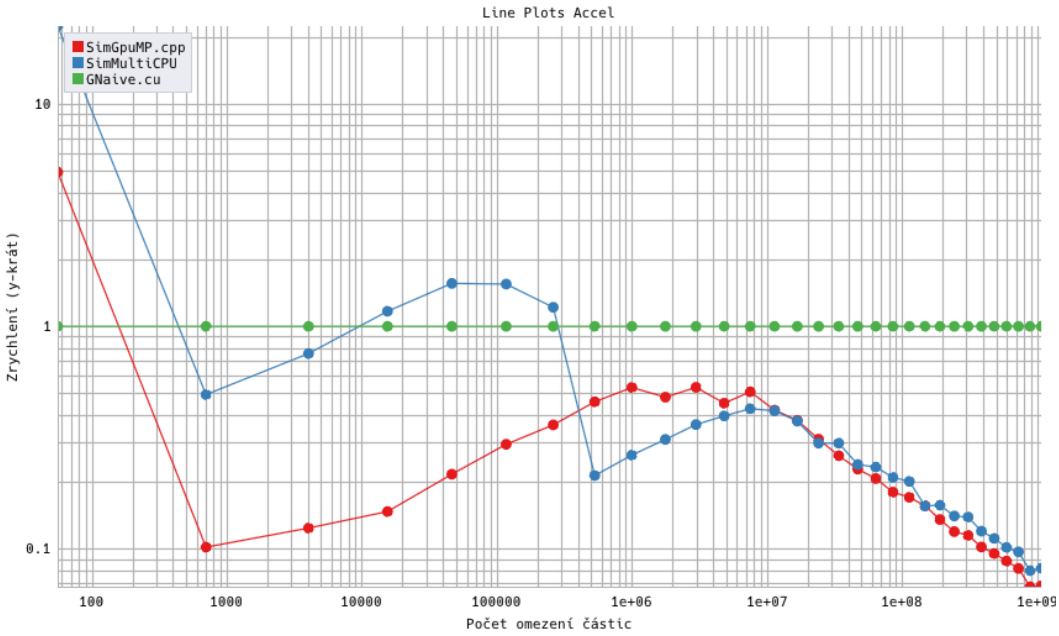
Obrázek 7.8: Zrychlení 4 (SC-GPU2) (log)



Obrázek 7.9: Zrychlení 5 (SC-GPU2) (log)



Obrázek 7.10: Rychlost 5 (SC-GPU2) (log)



Obrázek 7.11: Zrychlení 6 (SC-GPU2) (log)

Kapitola 8

Závěr

Cílem této práce bylo vytvořit sadu testovacích úloh s výukovým potenciálem. Hlavním úkolem tedy bylo najít vhodnou počáteční úlohu, která má potenciál pro akceleraci na grafické kartě. Z této úlohy měla být vytvořena sada úloh zohledňující různé aspekty programování na grafické karty.

Toho bylo dosaženo vytvořením úlohy založené na Position Based Dynamics při zachování co největší jednoduchosti výpočtu, aby byla co nejvýší šance, že i někdo další ho může celý pochopit, a následném vytvoření sady úloh, kterou tuto základní úlohu nějak modifikují.

Vedlejším úkolem bylo všemožně ulehčit práci uživateli co by si chtěl tyto úlohy sám naprogramovat.

Toho bylo dosaženo grafickou ovládací aplikací, co je schopná úlohy spouštět, pomáhat při jejich ladění a jednoduše tyto úlohy hromadně testovat a následně vytvářet grafy pro lepší pochopení výsledků.

Na závěr bych chtěl shrnout několik možných rozšíření. Očividnou možností je otestovat jestli všechny typy úloh budou mít stejný nebo podobný vliv při použití na 4 grafických kartách. Dalším možným rozšířením by mohlo být využití ještě více grafických karet například na klastrech s pomocím technologie OpenMPI. Dalším možným rozšířením by bylo použít jinou základní úlohu postavenou na Position Base Dynamics jako například Position Based Fluids[7], kde je vhodné využít nějaký řadící algoritmus. Případně rozšířit portfolio úloh o úplně novou, která by prověřila další aspekty, jako třeba nějaký druh video filtru, nebo jejich sekvence co by se daly řadit do pipeline. Stačí aby nová úloha se dala reprezentovat jako množina barevných bodů a díky to by nemělo být moc složité ji integrovat.

Bibliography

[1] Board, O. A. R. *OpenMP Application Programming Interface* [online]. [cit. 2024-04-15]. Dostupné z: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. (document), 3.4

[2] Devices, A. M. *AMD CDNA 2 ARCHITECTURE* [online]. [cit. 2024-04-15]. Dostupné z: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf. (document), 2.3, 2.3, B.1, B.2

[3] Devices, A. M. *AMD CDNA ARCHITECTURE* [online]. [cit. 2024-04-15]. Dostupné z: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna-white-paper.pdf. 2.3

[4] Devices, A. M. *RDNA3 Instruction Set Architecture* [online]. [cit. 2024-04-15]. Dostupné z: https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf. 2.1

[5] Konstantinidis, E. a Cotronis, Y. A Practical Performance Model for Compute and Memory Bound GPU Kernels. In:. Březen 2015. DOI: 10.1109/PDP.2015.51. 4, 7.1a

[6] Li, A., Song, S. L., Wijtvliet, M., Kumar, A. a Corporaal, H. SFU-Driven Transparent Approximation Acceleration on GPUs. *Proceedings of the 2016 International Conference on Supercomputing*. 2016. Dostupné z: https://api.semanticscholar.org/CorpusID:12938205. 5.5.5

[7] Macklin, M. a Müller, M. Position based fluids. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. jul 2013, sv. 32, č. 4. DOI: 10.1145/2461912.2461984. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/2461912.2461984. 8

[8] Müller, M. *09 Getting ready to simulate the world with XPBD* [online]. [cit. 2024-04-15]. Dostupné z: https://www.youtube.com/watch?v=jrociOAYqxA. 5.2, 5.2

[9] Müller, M. *XPBD, Extended Position Based Dynamics* [online]. [cit. 2024-04-15]. Dostupné z: https://matthias-research.github.io/pages/tenMinutePhysics/09-xpbd.pdf. (document), 5.2, 5.1, 5.2

[10] Müller, M., Heidelberger, B., Hennix, M. a Ratcliff, J. Position Based Dynamics. In: Mendoza, C. a Navazo, I., ed. *Vriphys: 3rd Workshop in Virtual Realitiy, Interactions, and Physical Simulation*. The Eurographics Association, 2006. DOI: 10.2312/PE/vriphys/vriphys06/071-080. ISBN 3-905673-61-4. 5.2

[11] NVIDIA. *CUDA C++ Programming Guide* [online]. [cit. 2024-05-05]. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (document), 2.1, 2.1, 2.2, 3.1, 5.5.5

[12] NVIDIA. *CUDA C++ Programming Guide* [online]. [cit. 2024-05-05]. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. 3.1, 5.5.2

[13] NVIDIA. *GPU Performance Background User's Guide* [online]. [cit. 2024-05-05]. Dostupné z: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html. 2.1, 2.2

[14] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE* [online]. [cit. 2024-04-15]. Dostupné z: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf. (document), 2.2, 2.2, 2.2, A.1, A.2

[15] NVIDIA. *NVIDIA Tesla P100* [online]. [cit. 2024-04-15]. Dostupné z: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf. 2.2

[16] NVIDIA. *Optimizing Parallel Reduction in CUDA* [online]. [cit. 2024-04-15]. Dostupné z: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. (document), 5.5.6, 5.6, 6.2.5

[17] Ronny Krashinsky, S. J. N. S. a Ramaswamy, S. *NVIDIA Ampere Architecture In-Depth* [online]. [cit. 2024-05-05]. Dostupné z: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/. 2.2

[18] Wiki, O. *Shader* [online]. [cit. 2024-05-05]. Dostupné z: https://www.khronos.org/opengl/wiki/Shader. 2

Příloha A

Nvidia Ampere

Figure A.1: Ampere GPC [14]

Figure A.2: GA102 [14]

Příloha B

AMD CDNA 2

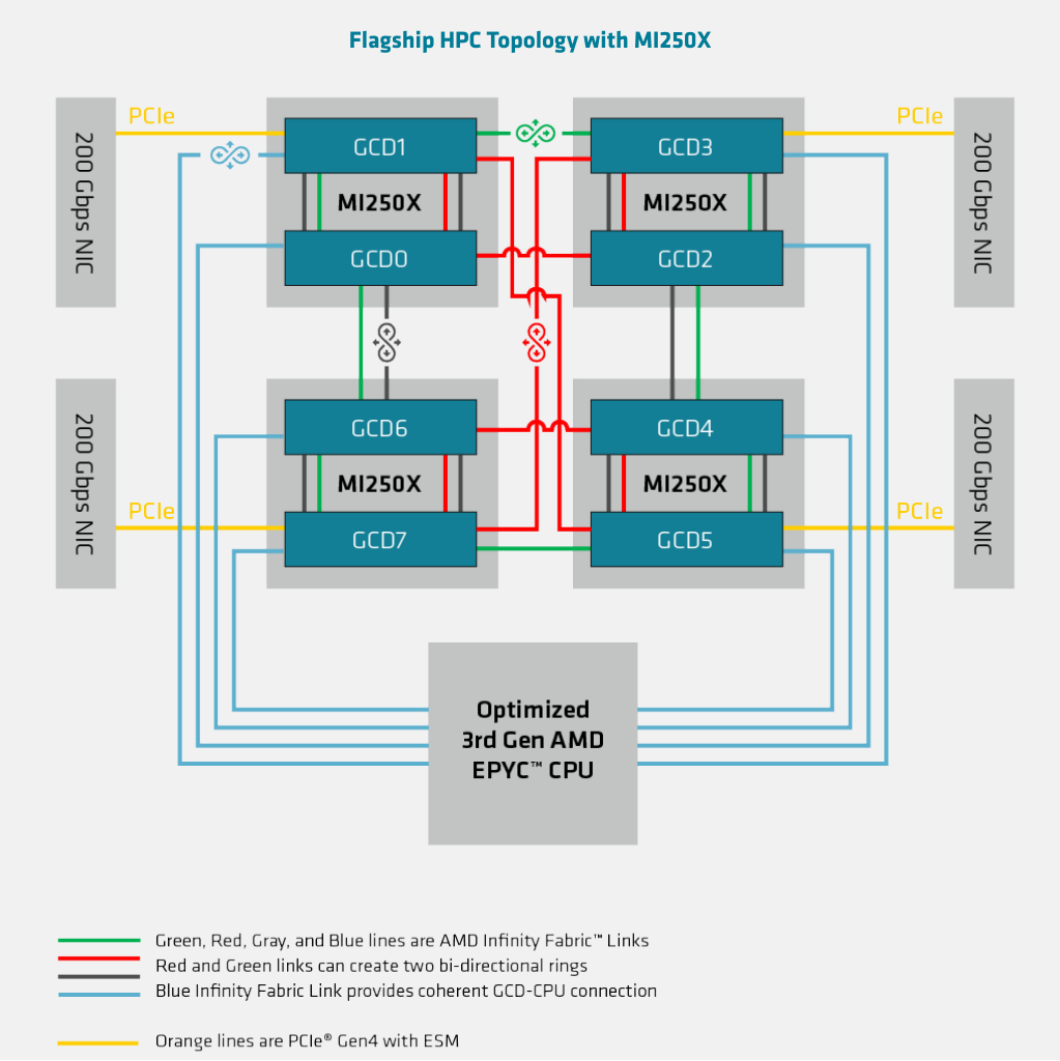


Figure B.1: Infinity Fabric [2]

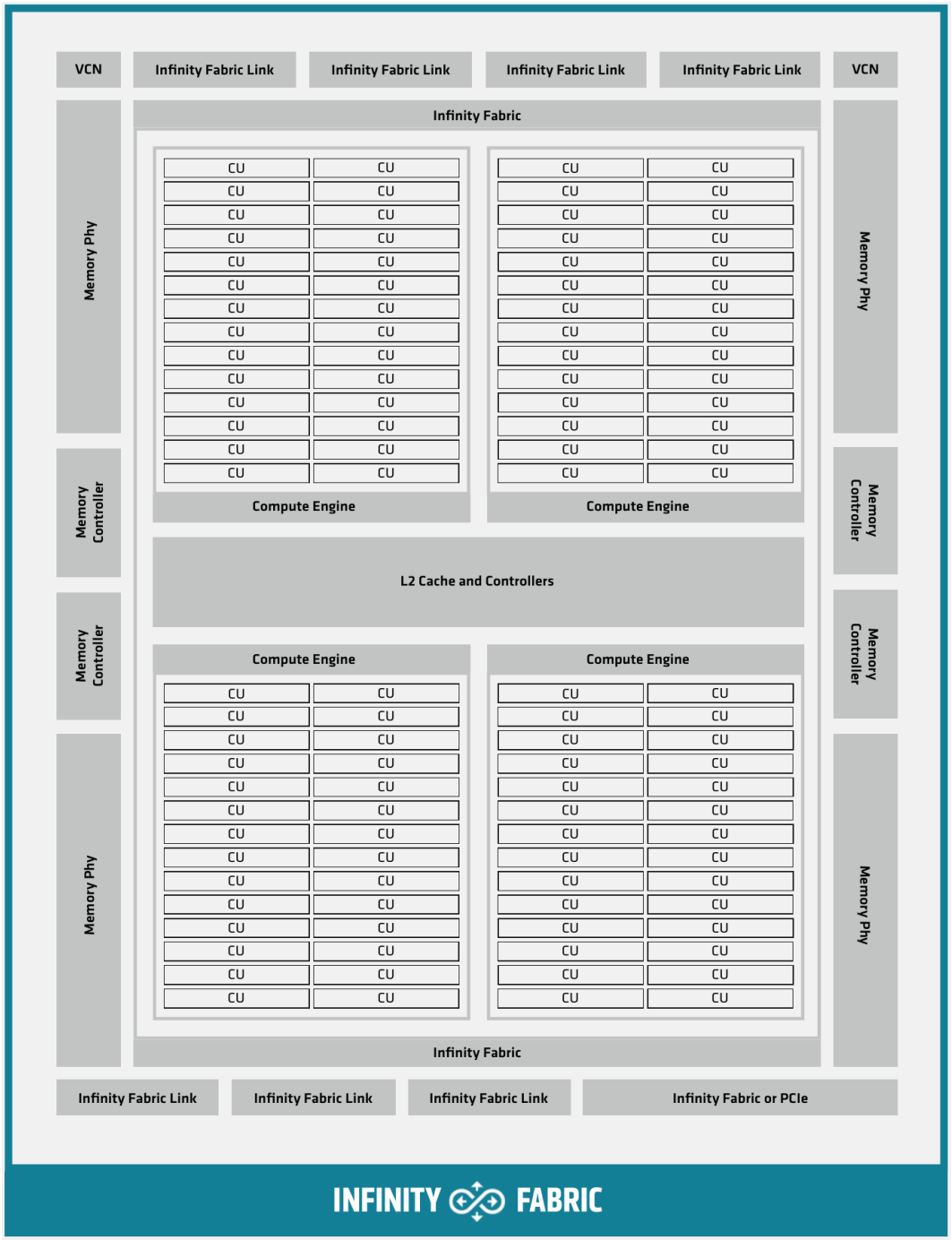
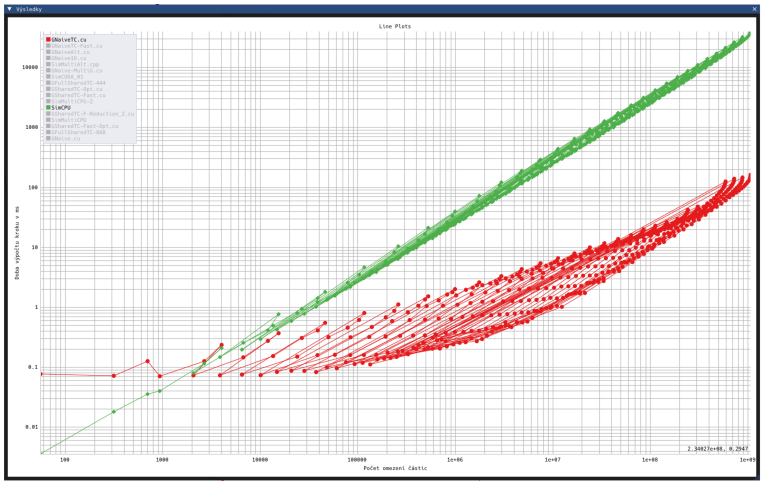


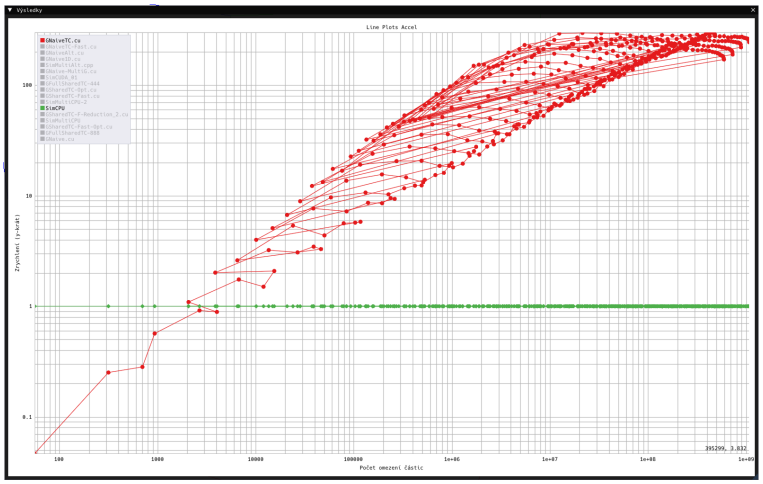
Figure B.2: MI200 [2]

Příloha C

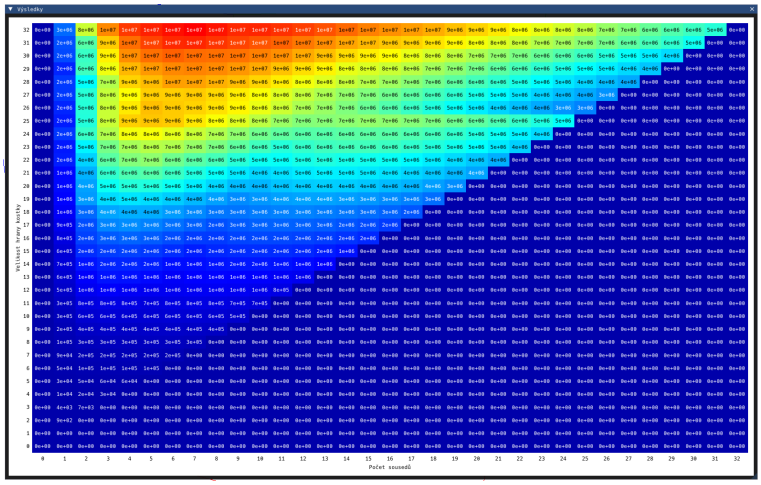
Ovladací aplikace ukázka grafů



(a) Obyčejný graf

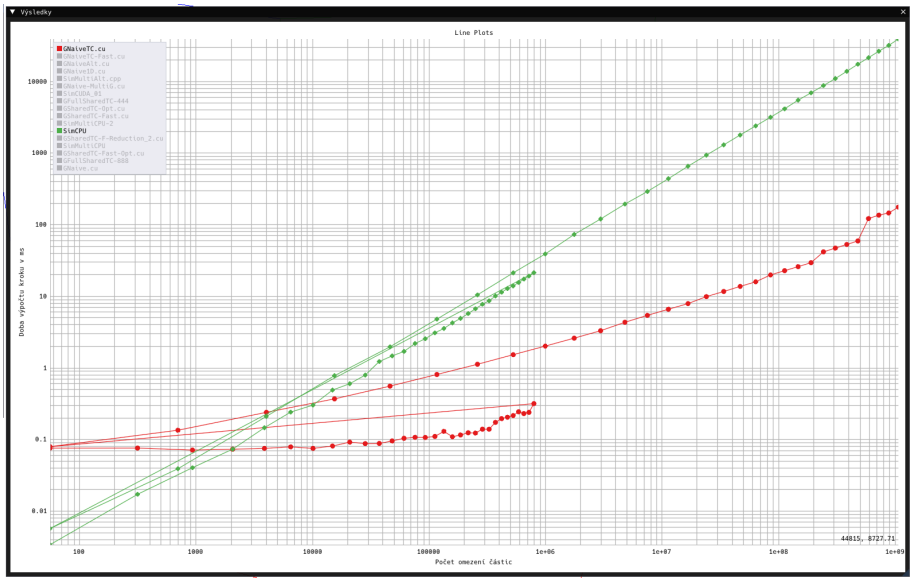


(b) Graf zrychlení

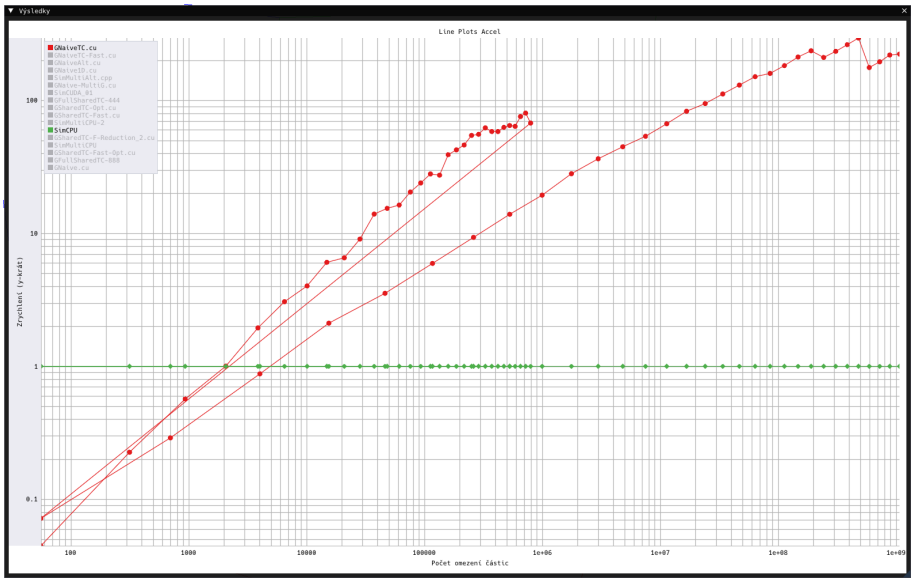


(c) Tepelná mapa

Figure C.1: Typy grafů



(a) Obyčejný graf

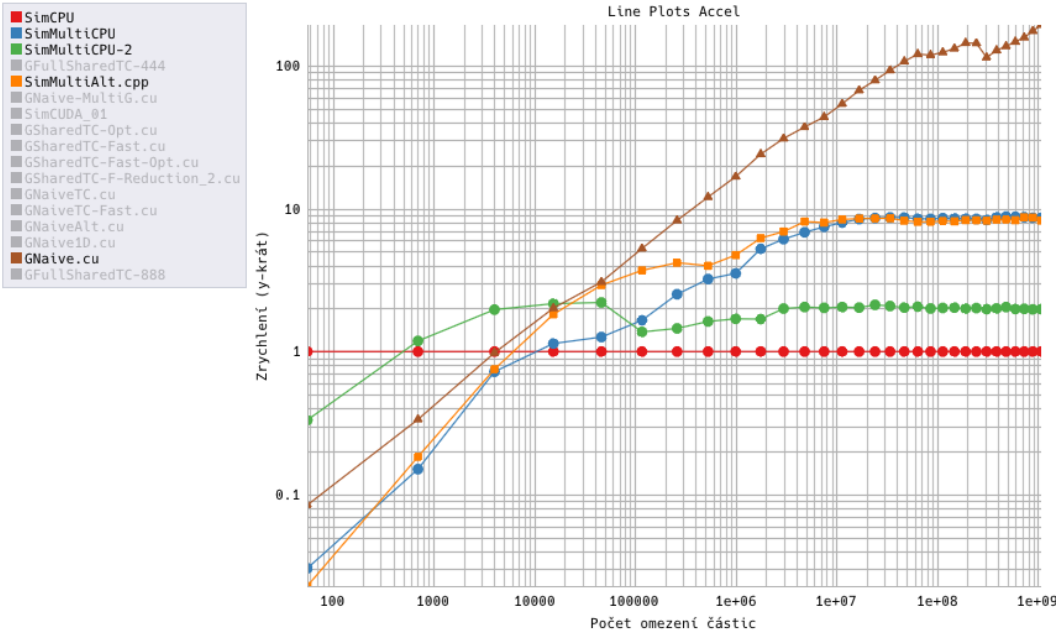


(b) Graf zrychlení

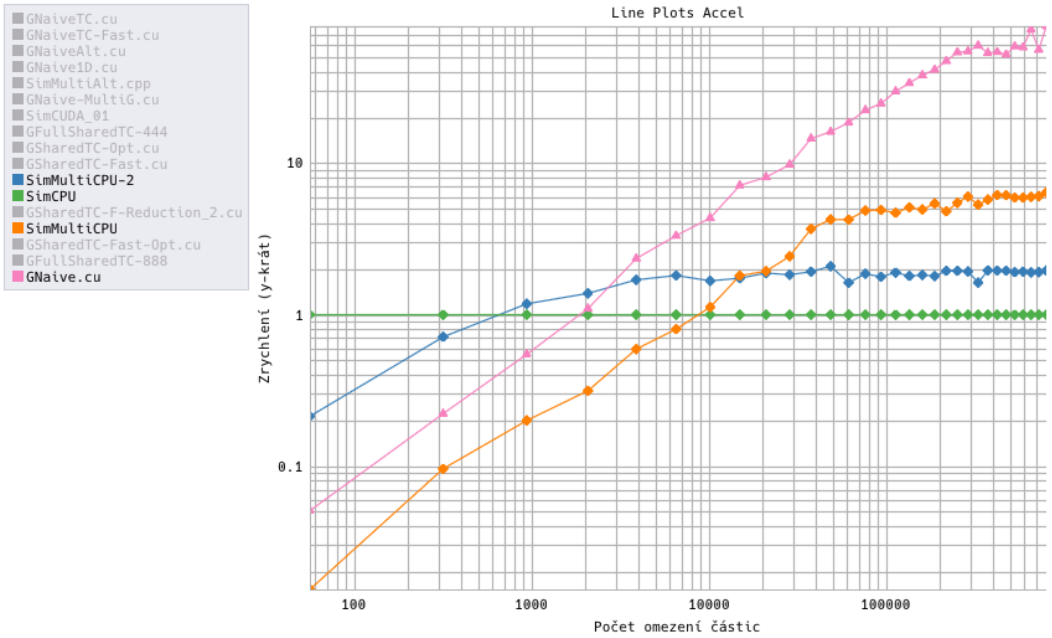
Figure C.2: Sousednost 1 a všichni

Příloha D

SC-GPU



(a) S=H-1



(b) S=1

Figure D.1: Zrychlení (SC-GPU) (log)

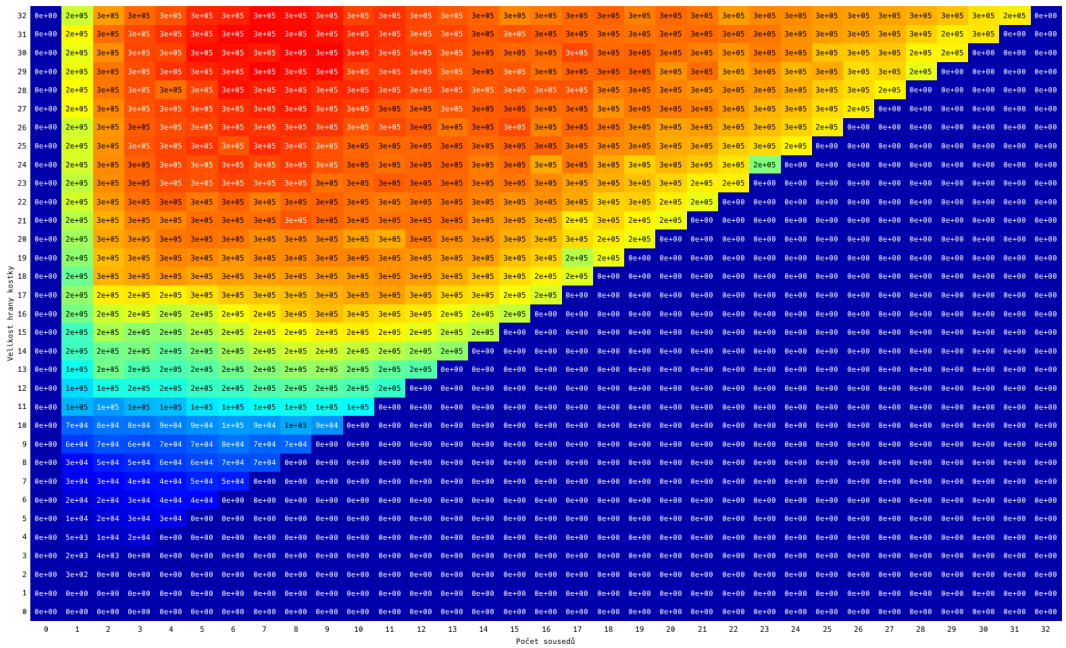


Figure D.2: Mapa SimMultiCPU (SC-GPU)

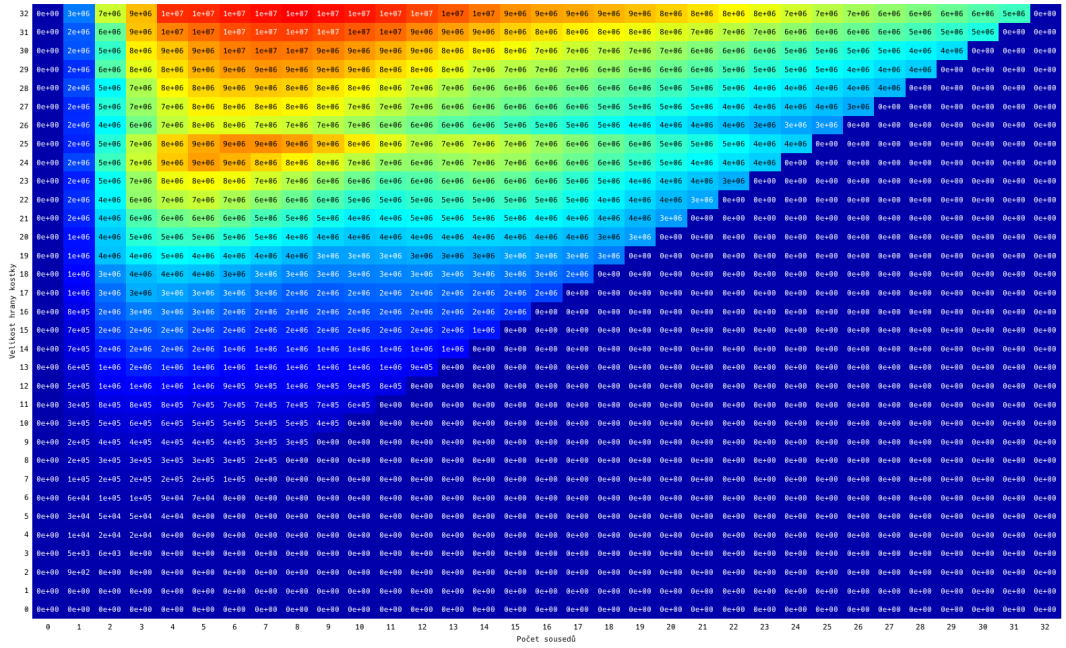
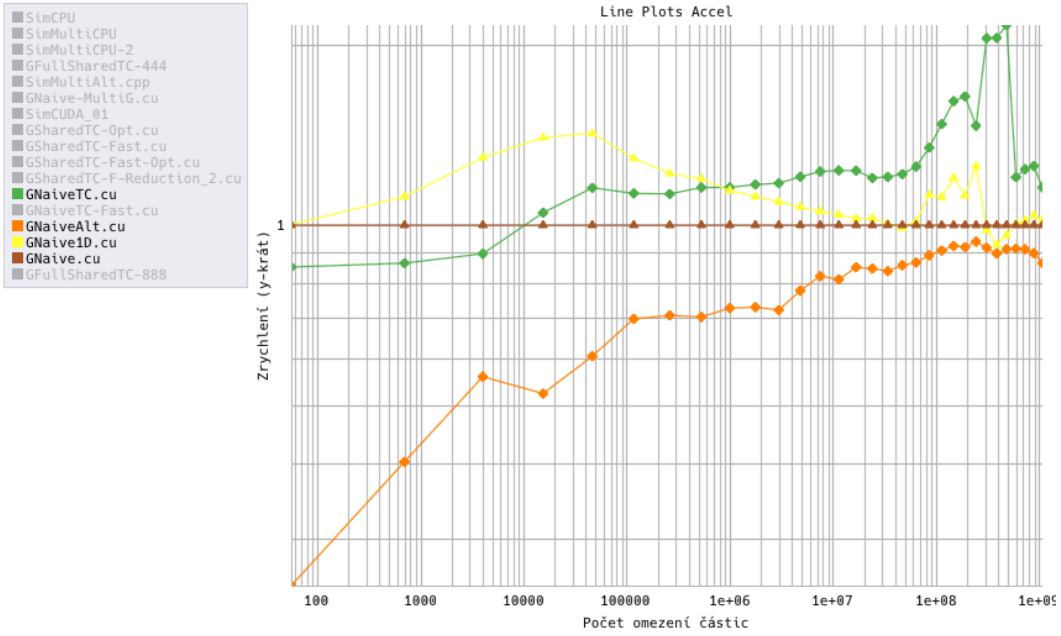
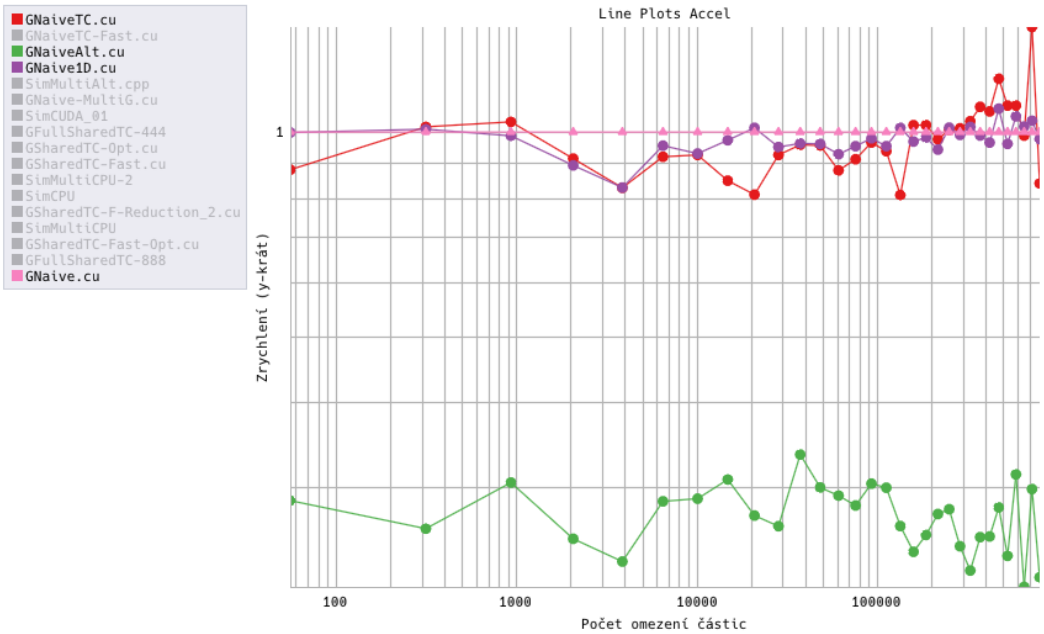


Figure D.3: Mapa GNaive (SC-GPU)



(a) S=H-1



(b) S=1

Figure D.4: Zrychlení 2 (SC-GPU) (log)

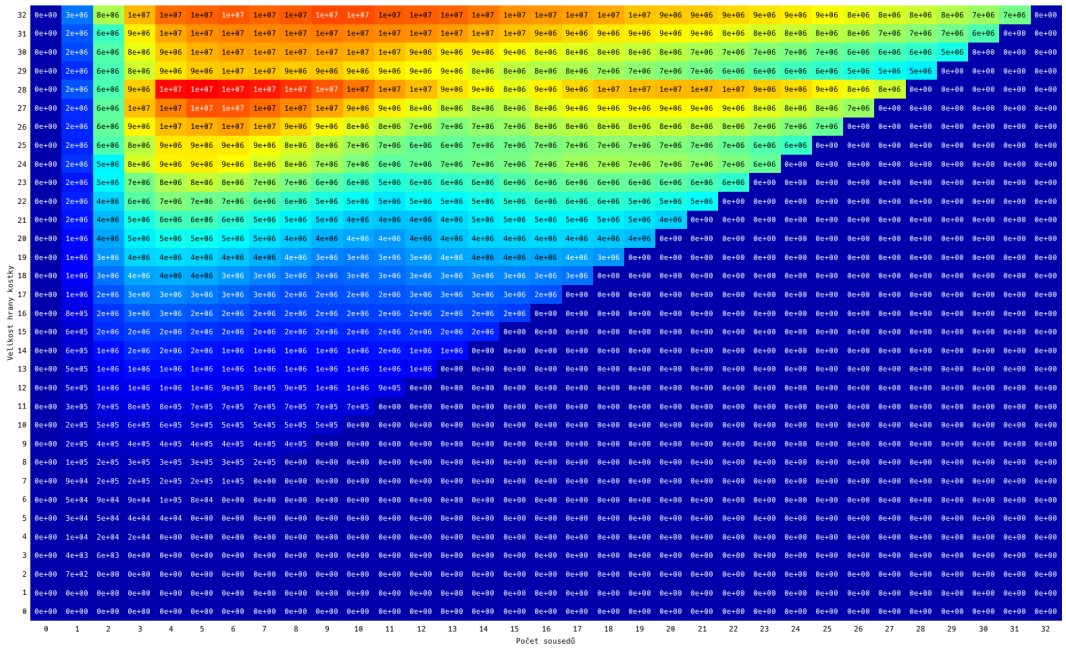


Figure D.5: Mapa GNaiveTC (SC-GPU)

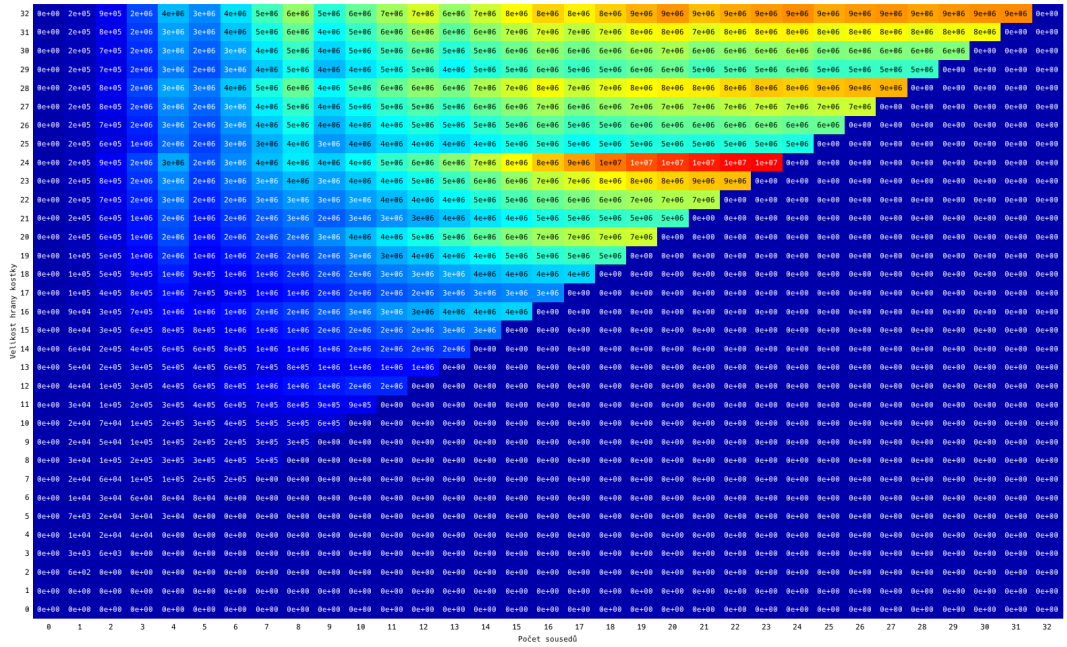
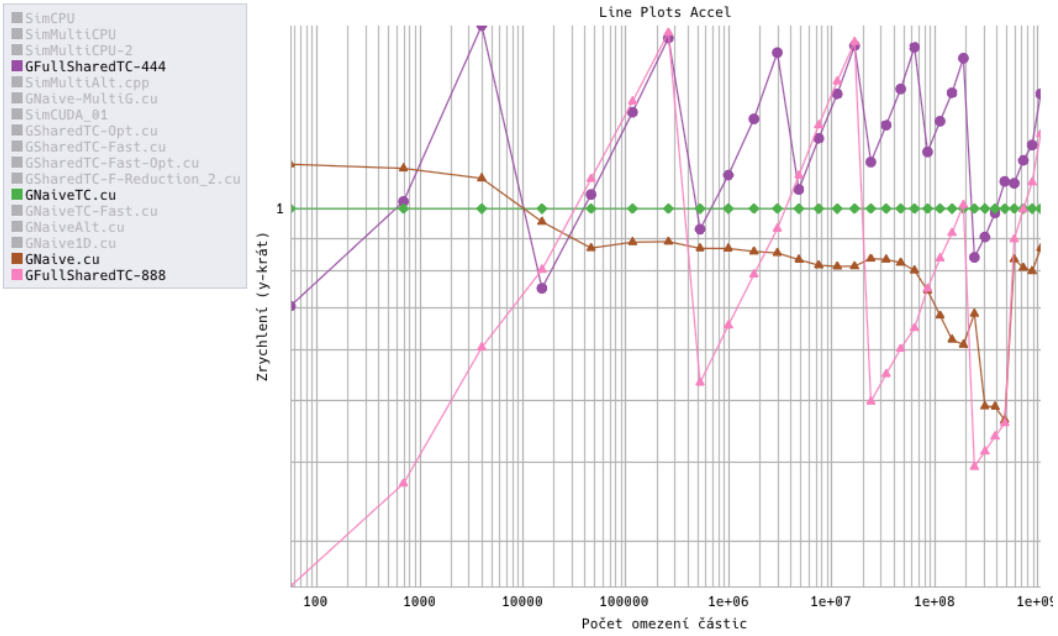
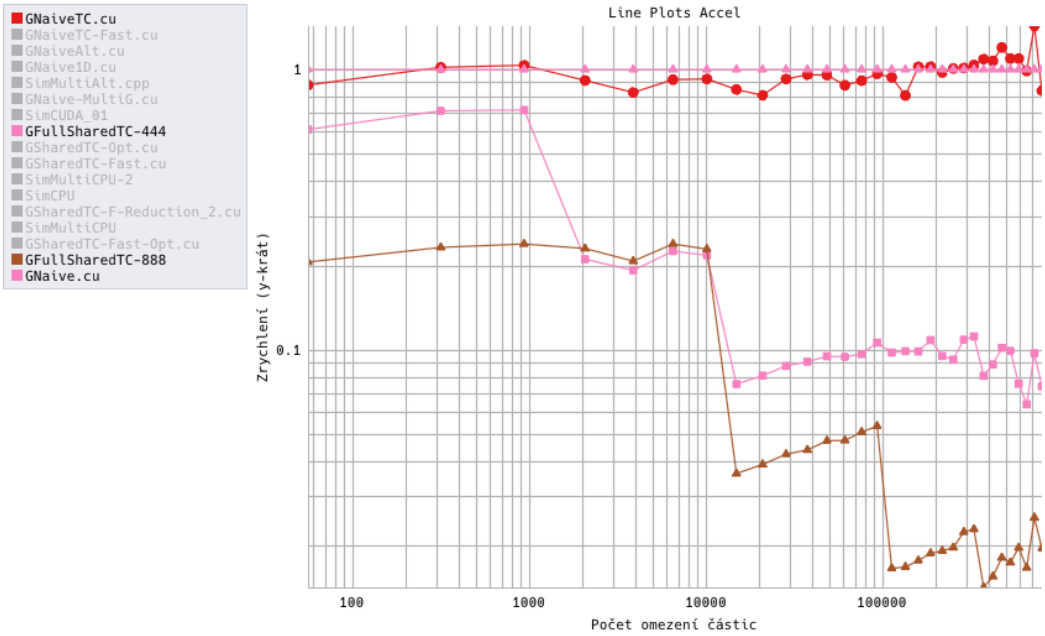


Figure D.6: Mapa GShared-4 (SC-GPU)

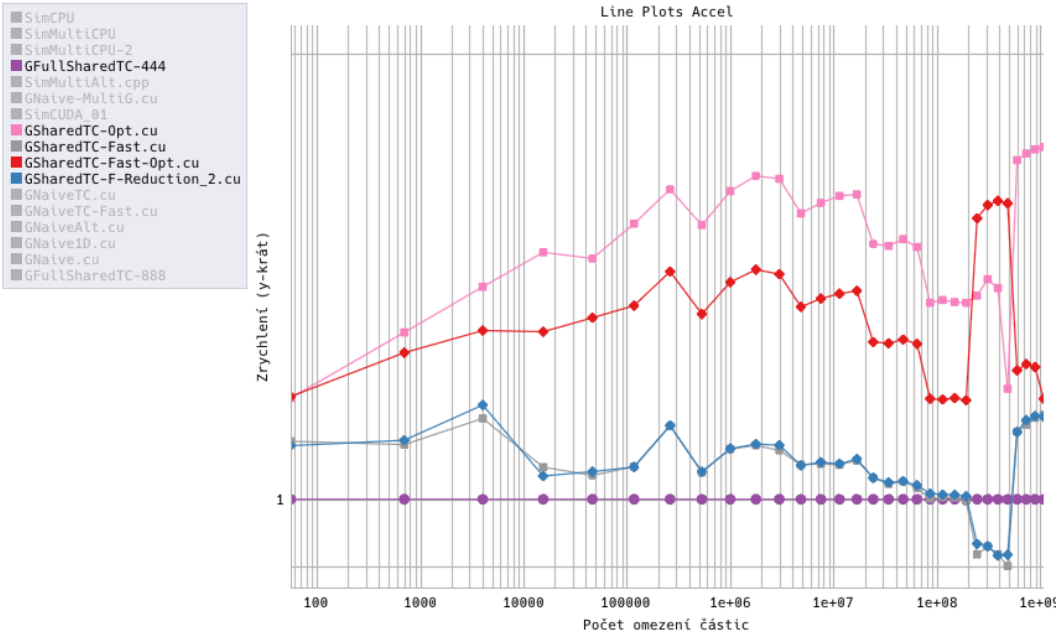


(a) S=H-1

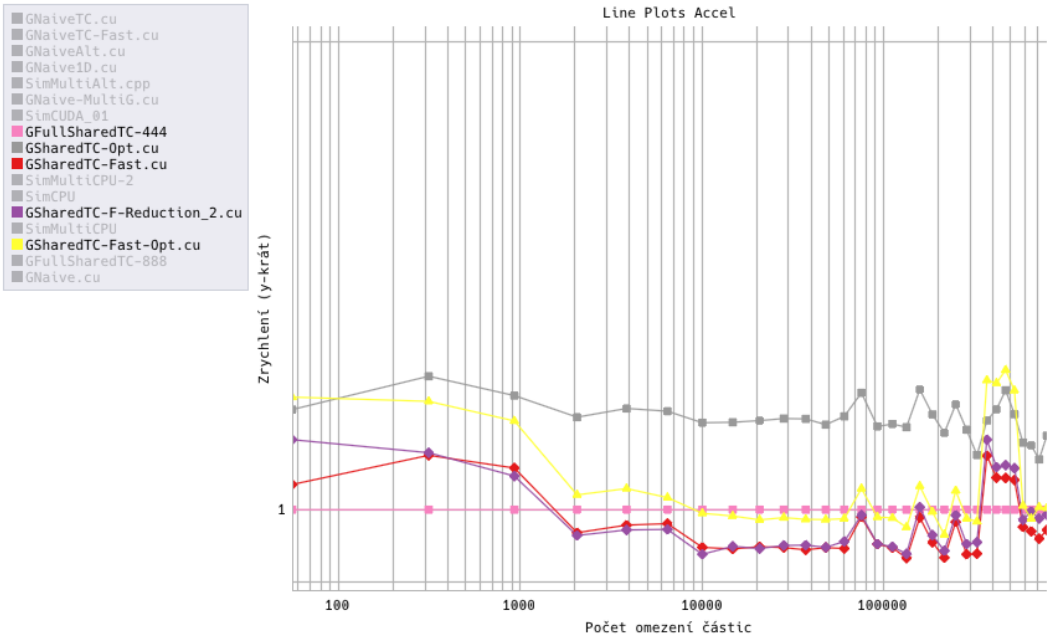


(b) S=1

Figure D.7: Zrychlení 3 (SC-GPU) (log)



(a) S=H-1



(b) S=1

Figure D.8: Zrychlení 4 (SC-GPU) (log)

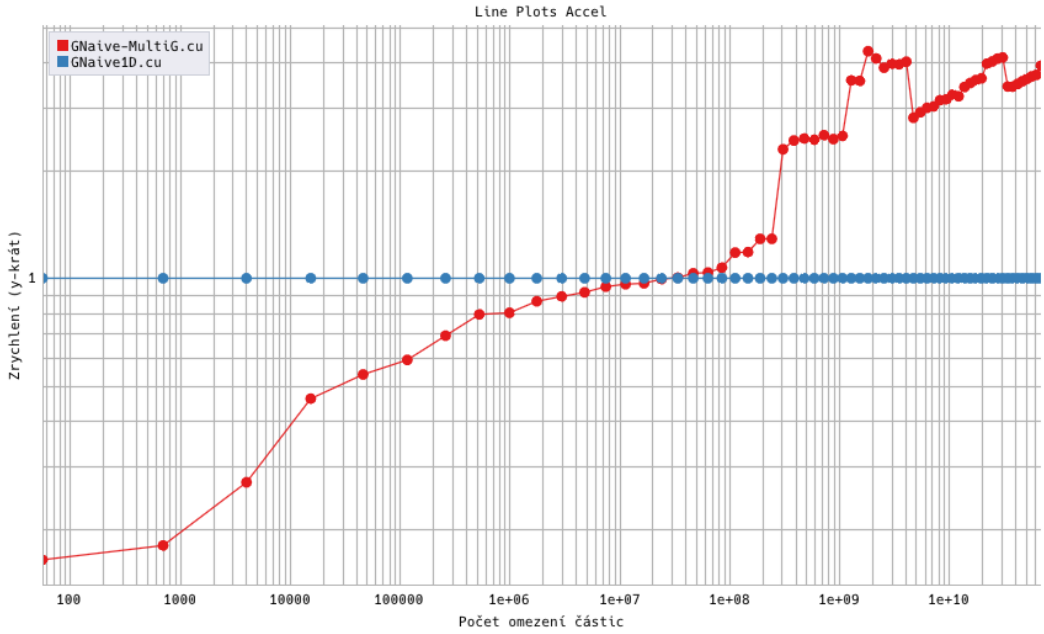


Figure D.9: Zrychlení 5 (SC-GPU) (log)

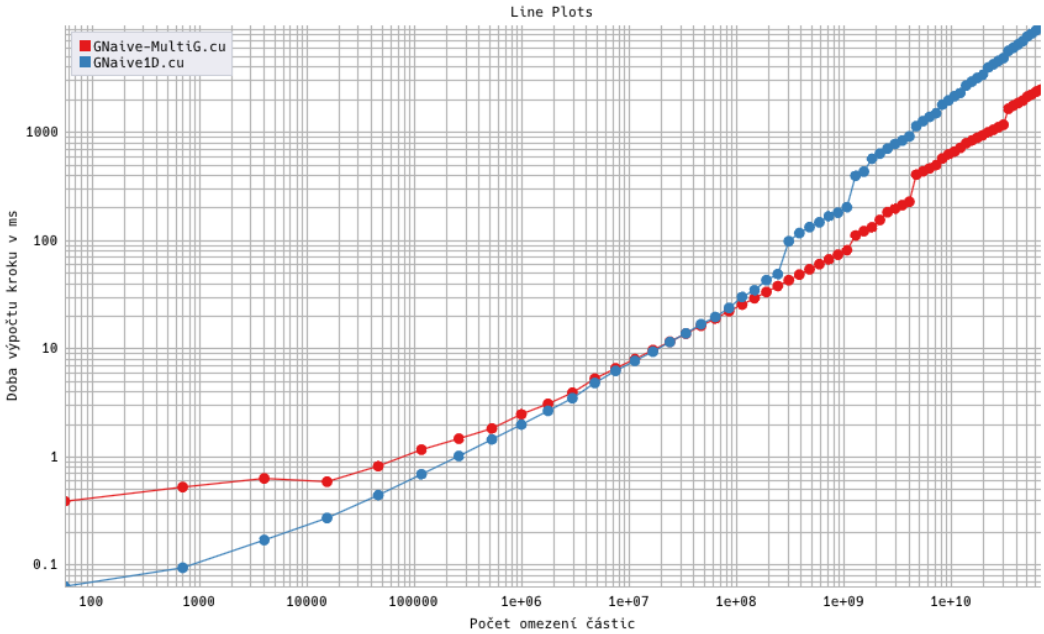
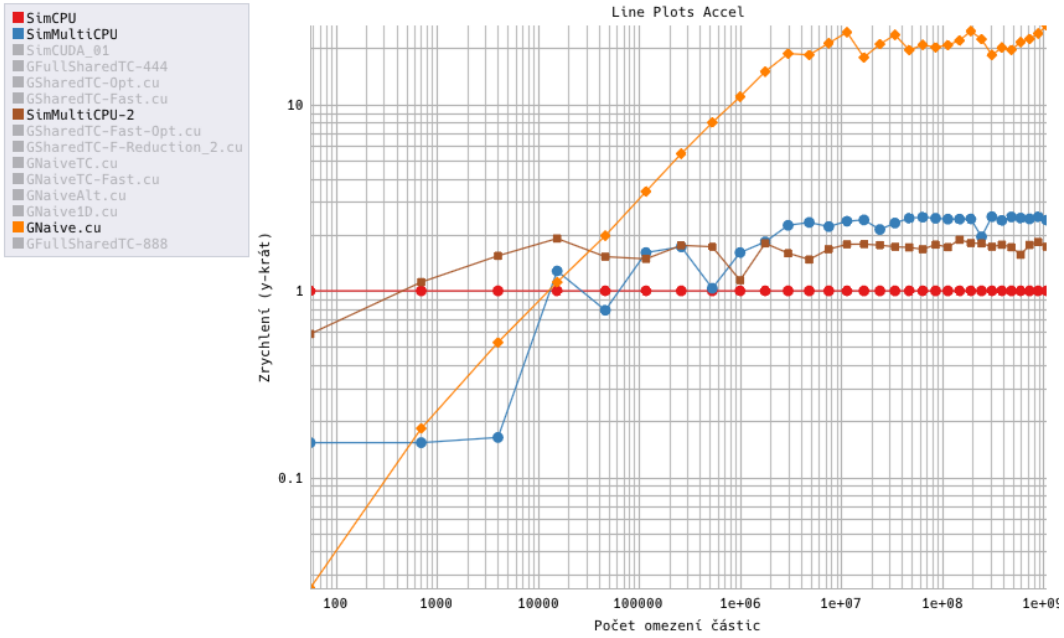


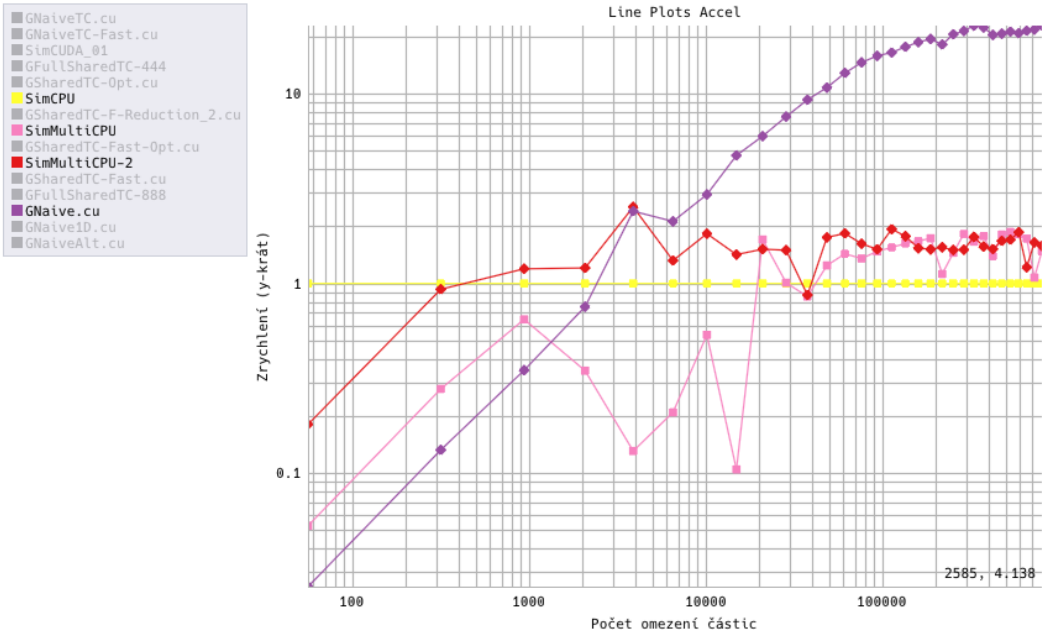
Figure D.10: Rychlost 5 (SC-GPU) (log)

Příloha E

Stařec

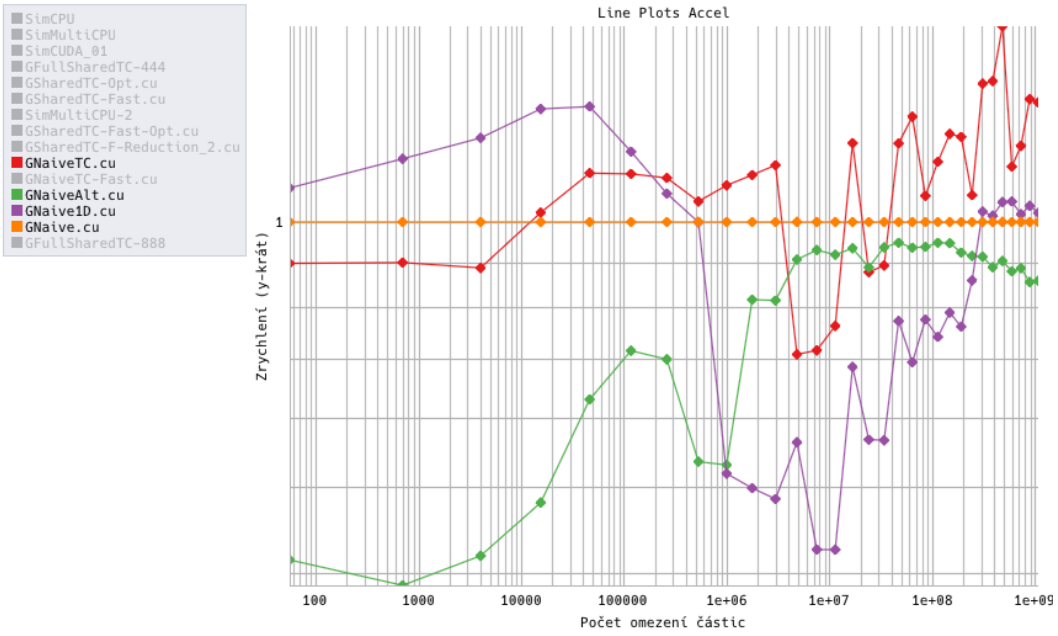


(a) S=H-1

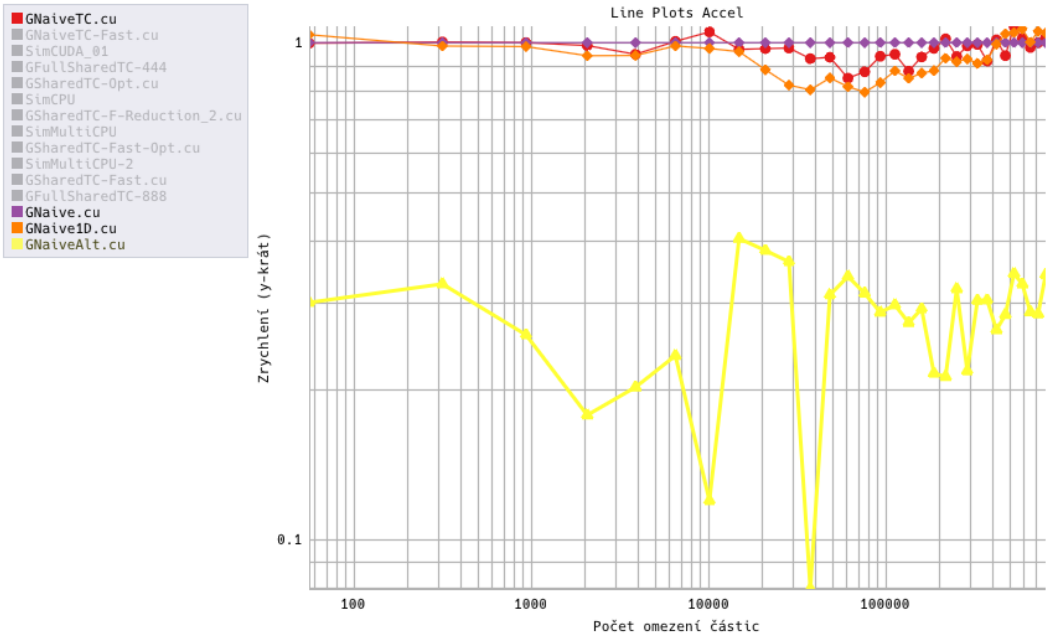


(b) S=1

Figure E.1: Zrychlení (Stařec) (log)

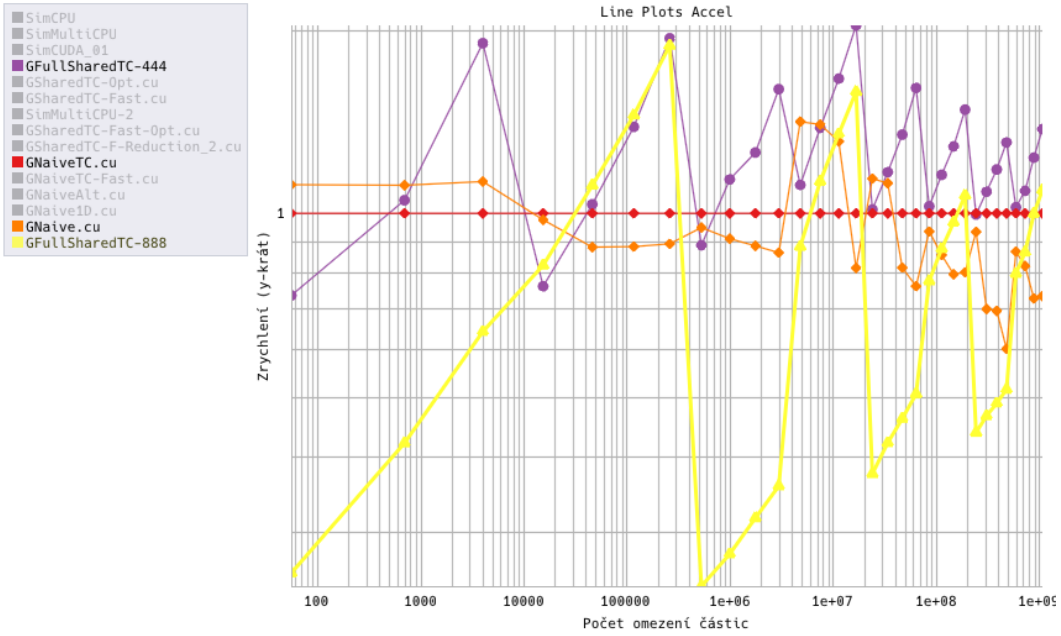


(a) S=H-1

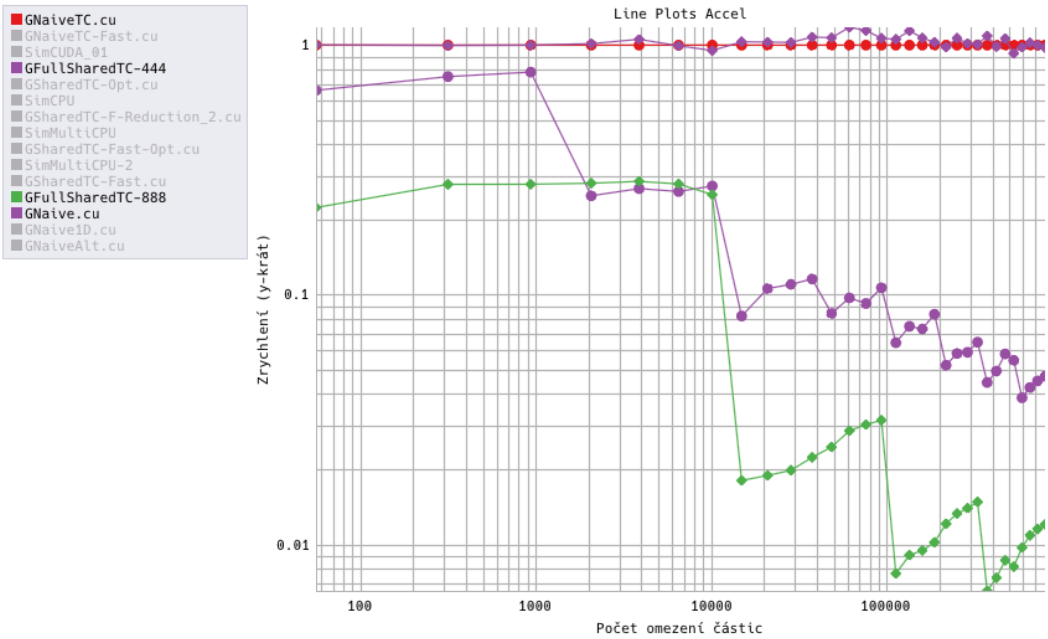


(b) S=1

Figure E.2: Zrychlení 2 (Stařec) (log)

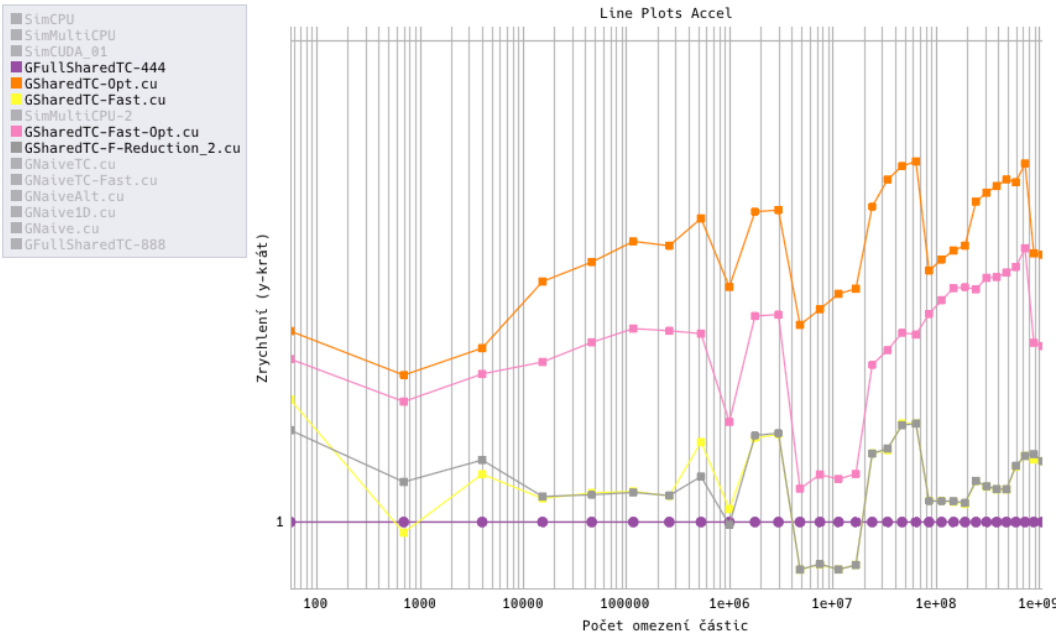


(a) S=H-1

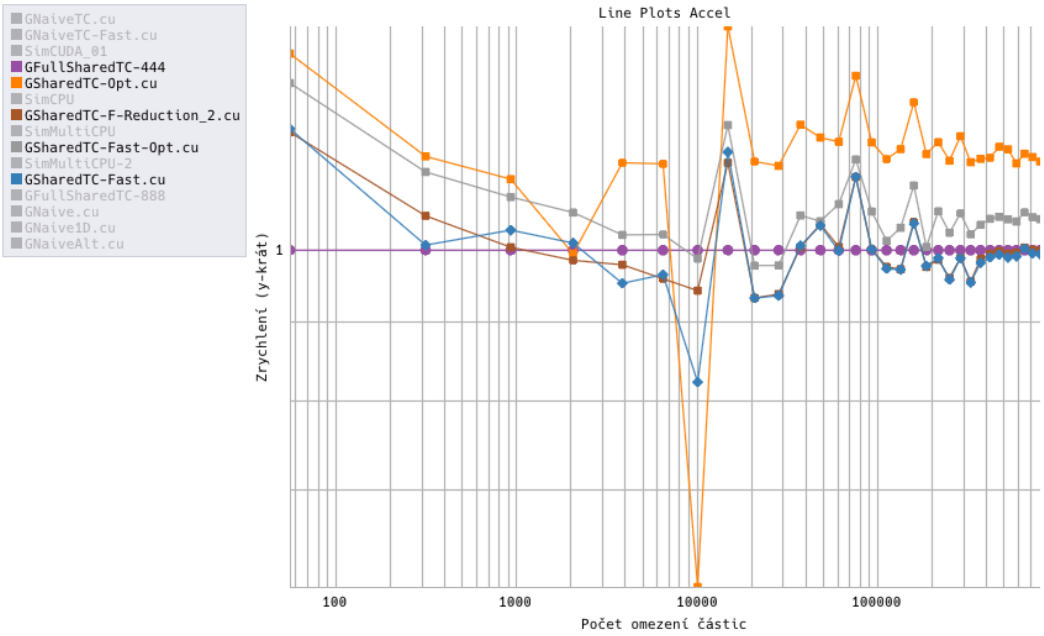


(b) S=1

Figure E.3: Zrychlení 3 (Stařec) (log)



(a) S=H-1



(b) S=1

Figure E.4: Zrychlení 4 (Stařec) (log)