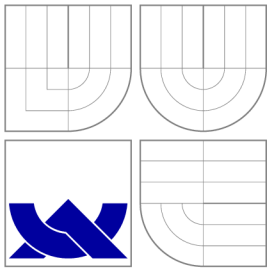
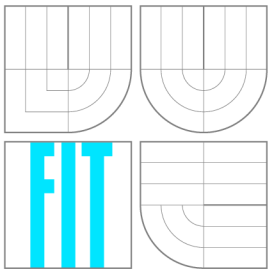


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KONSTRUKCE KD STROMU NA GPU

BUILDING KD TREE ON GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB BAJZA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOZEF KOBRTEK

BRNO 2016

Zadání diplomové práce/17937/2015/xbajza00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Bajza Jakub, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Konstrukce kD stromu na GPU**
Building kD Tree on GPU
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte knihovny OpenGL a nVidia CUDA.
2. Prostudujte algoritmy pro konstrukci kD stromů, zaměřte se na jejich implementaci na grafické kartě.
3. Navrhněte aplikaci využívající kD stromy implementované v rozhraní CUDA a OpenGL.
4. Aplikaci naimplementujte a demonstrujte na testovací scéně s benchmarkem.
5. Vykonejte měření na různých grafických kartách, porovnejte s existujícím řešením.
6. Zhodnoťte dosažené výsledky.

Literatura:

- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kobrtek Jozef, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Táto diplomová práca sa zaoberá konštrukciou akceleračných štruktúr typu kD strom a následnou ich následnou paralelizáciou pomocou GPU. Na začiatku je čitateľ oboznámený s platformou CUDA pre paralelné programovanie. Ide o popis všeobecných princípov ako aj špecifických vlastností, využitých v rámci tejto práce. Potom je čitateľ uvedený do problematiky akceleračných štruktúr pre sledovanie lúčov. Tieto štruktúry sú opísané a akceleračná štruktúra pre kD strom a jej varianty sú popísané do detailov. Následne je rozobraná analýza zvoleného variantu kD stromu a sú prezentované možné problémy a úskalia pri jej paralelnej implementácii. V rámci popisu implementácie je zahrnutý krátky popis CPU variantu a detailné popisy jednotlivých CUDA kernelov. Sekcia o testovaní prináša výsledky implementácie vo forme zrovnania CPU a GPU implementácie, ako aj vyhodnotenie naplnenie metriky stanovenej počas návrhu. V závere je obsiahnuté zhrnutie dosiahnutých cieľov a výsledkov nasledované popisom možných budúcich vylepšení na implementácii.

Abstract

This term project addresses the construction of kD tree acceleration structures and parallelization of this construction using GPU. At the beginning, there is an introduction of the reader into CUDA platform for parallel programming. There is a decription of generic principles as well as specific features that will be used in this thesis. Following that the reader is put into the issue of acceleration structures for Ray tracing. These structures are described and the kD tree acceleration structure and its variants are portrayed in detail. After that the analysis of chosen kD tree variant is broken down and the problems and issuse of its parallel implementation are adressed. As a part of implementation discription, there is a short descripton of CPU variant and detailed specifications of the CUDA kernels. The testing section brings the results of implementation in form of CPU vs GPU comparison, as well as evaluation of how much the metric set in design was fulfilled. In the end there is a summary of achieved goals and results followed by possible future improvements for the implementation.

Kľúčové slová

kD strom, SAH, GPU, GPGPU, CUDA, paralelizácia, sledovánie lúčov

Keywords

kD tree, SAH, GPU, GPGPU, CUDA, parallelization, ray tracing

Citácia

BAJZA, Jakub. *Konstrukce kD stromu na GPU*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kobrtek Jozef.

Konstrukce kD stromu na GPU

Prehlásenie

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Jozefa Kobrtka a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Bajza
24. mája 2016

Poďakovanie

Ďakujem pánovi Ing. Jozefovi Kobrtkovi za odbornú pomoc pri riešení semestrálneho projektu.

Obsah

Kapitola 1

Úvod

Počítačová grafika je v dnešnej dobe veľmi rozšírená pre jej široké spektrum využitia. Veľkou oblasťou počítačovej grafiky je 3d zobrazenie scén a objektov. Táto oblasť sa zaoberá postupmi ako efektívne a realisticky zobraziť scénu a ako ju uchovať. Jednou z populárnych metód zobrazenia 3d scén je sledovanie lúčov. Táto metóda je však výpočtovo náročná. Pre zrýchlenie výpočtov sa začali používať efektívne spôsoby ukladania priestorových dát. Jednou z najpoužívanejších štruktúr, používaných pre ukladanie priestorových dát, je k-d strom. Problémom týchto dátových štruktúr je ich náročná konštrukcia. Jedným zo spôsobov, ako urýchliť konštrukciu, je vytvorenie paralelnej implementácie. Existuje veľké množstvo prác, zaoberajúcich sa paralelnou implementáciou priechodu takýmito dátovými štruktúrami, ale len malé množstvo, zaoberajúce sa paralelizáciou ich konštrukcie. Cieľom tejto práce je ukázať, že paralelná implementácia konštrukcie k-d stromu môže priniesť dostatočné pozitívne výsledky a teda sa jej vyplatí venovať.

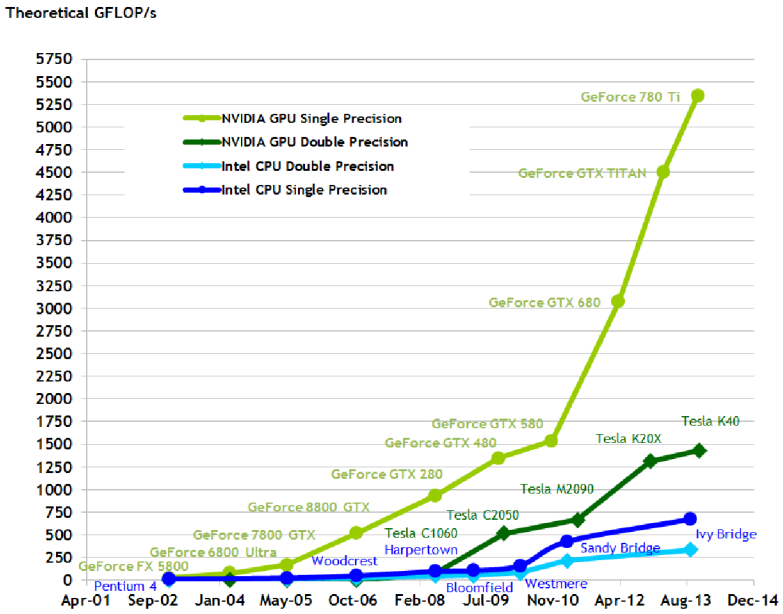
Táto práca sa zaoberá dátovou štruktúrou k-d strom, používanou pre techniku sledovania lúčov a jej následnou paralelizáciou na viacerých špecializovaných procesoroch na grafickej karte pomocou platformy CUDA. Kapitola 2 poskytuje informácie o použitej paralelizačnej platforme CUDA, jej štruktúre, vlastnostiach a obmedzeniach. Jej cieľom je priblížiť prvky platformy CUDA, ktoré budú využité neskôr v rámci tejto práce. V kapitole 3 sa nachádza popis prístupov ku akceleračným dátovým štruktúram, slúžiacim pre uchovávanie priestorových dát. Ide o skrátený popis v dnešnej dobe najpoužívanejších dátových štruktúr pre reprezentáciu trojrozmerných dát. Ku koncu kapitoly sa tiež nachádza podrobnejší popis nami zvolenej dátovej štruktúry k-d strom. Kapitola 4 sa venuje návrhu CPU a GPU variantov aplikácií. Obsahuje popis problematických miest a aspektov, ktorým je potrebné sa pri paralelnej implementácii venovať. Kapitola 5 popisuje implementáciu oboch variantov aplikácie. Ťažisko je v tomto prípade na GPU implementácii a popise jednotlivých CUDA kernelov. Kapitola 6 obsahuje špecifikáciu testovania a jeho výsledky. V prvej časti sa nachádza vyhodnotenie metriky naplnenia multiprocesorov. Následne v druhej časti sa nachádza porovnanie CPU a GPU implementácií so zdôvodnením nedostatkov. Záver obsahuje zhrnutie dosiahnutých výsledkov a smery, v ktorých by bolo možné prácu ďalej zdokonalovať.

Kapitola 2

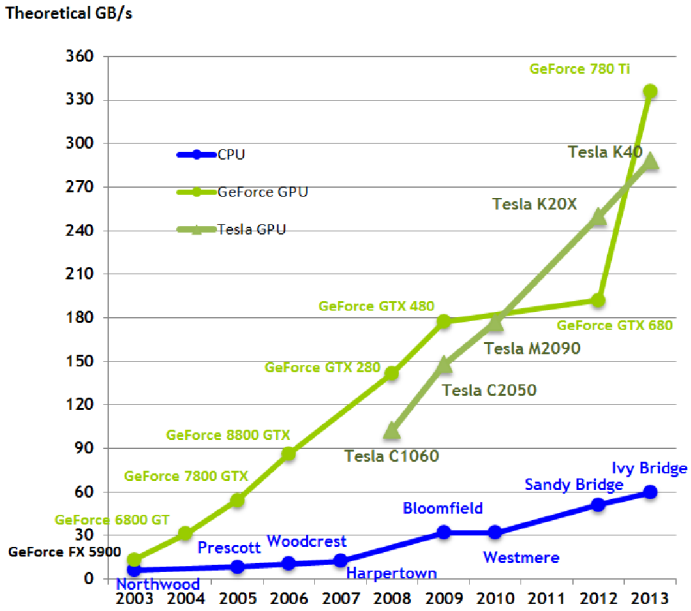
CUDA

CUDA je platforma pre paralelné programovanie, programové aplikačné rozhranie (API), a zároveň tiež programovací model vyvinutý firmou NVIDIA. CUDA umožňuje, pri správnej analýze a aplikácii, dramatické navýšenie výpočtového výkonu s využitím všeobecne použiteľných grafických kariet GPGPU (*General-purpose graphics processing units*). GPGPU označuje možnosť použitia GPU čipov, ktoré majú bežne na starosti špecializované grafické výpočty, na výpočty, na ktoré sa bežne používa CPU [12][10]. Použitím viacerých grafických kariet na jednom počítači alebo väčšieho počtu grafických čipov, je možné ďalej sparalelizovať už aj tak paralelnú podstatu spracovávania na grafickej karte [11]. Okrem toho, aj jednoduchý GPU-CPU framework poskytuje výhody, ktoré riešenie s viacerými CPU neposkytuje, čo vyplýva z špecializácie jednotlivých čipov [17].

Ako bolo spomenuté vyššie GPU čipy umožňujú, pri správnom použití, značne skrátiť dobu výpočtu. Akého urýchlenia sa dosiahne však závisí okrem konkrétnej aplikácie na výkone použitého GPU čipu. Za posledných niekoľko rokov nastal nárast výkonu GPU voči CPU ako ukazujú obrázky 2.1 a 2.2.

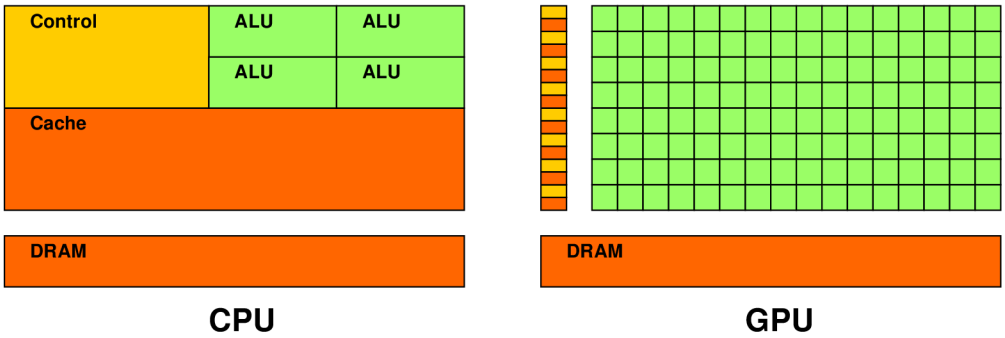


Obr. 2.1: Operácie s plávajúcou rádovou čiarkou za sekundu pre CPU a GPU [25].



Obr. 2.2: Širka prenosového pásma pre prístup do pamäte z CPU a GPU [25].

Dôvodom nezrovnalosti vo výpočtoch s plávajúcou rádovou čiarkou medzi CPU a GPU (viď 2.1) je špecializácia pre náročné, vysoko paralelné výpočty - presne to čo je potrebné v zobrazovaní - a preto sú GPU navrhnuté tak, že viac tranzistorov je venovaných na spracovávanie dát naproti ukladaniu dát do medzi-pamäte *cache*, ako je zobrazené na obrázku 2.3. Presnejšie, GPU sú navrhnuté tak, aby dobre zvládali problémy, ktoré môžu byť vyjadriteľné ako výpočty s dátovým paralelizmom (Jeden program vykonávaný na väčšom množstve dátových prvkov paralelne) a vysokým pomerom aritmetických ku pamäťovým operáciám. Keďže sa pre každý prvok vykonáva rovnaký program, nie sú také požiadavky na reguláciu toku riadenia a je možné skryť latenciu prístupu do pamäti cez veľké množstvo aritmetických operácií, naproti veľkým pamätiam *cache*. Paralelné spracovanie dát mapuje dátové prvky na paralelne pracujúce vlákna. V 3D zobrazení sa tento prístup využíva na spracovanie súborov pixelov a bodov. Táto kapitola bola vypracovaná na základe zdrojov [24, 25, 27].



Obr. 2.3: GPU dedikuje viac tranzistorov na spracovávanie dát [25].

2.1 CUDA Programovací model

Keďže paralelizmus dnešných GPU rastie s Moorovým zákonom [1], je nutné aby sme vyvíjali software, ktorý transparentne škáluje svoj paralelizmus aby využil narastajúci počet jadier procesorov.

Paralelný programovací model, ktorý CUDA prináša, je navrhnutý tak, aby prekonal túto výzvu, zatiaľ čo si zachová nízku krivku učenia pre programátorov oboznámených so štandardnými programovacími jazykmi ako napríklad jazyk C.

Jadrom tohto programovacieho modelu sú tri kľúčové abstrakcie:

- hierarchia skupín vlákien
- zdieľaná pamäť
- barierová synchronizácia

Tieto abstrakcie sú programátorovi sprístupnené ako minimálna kolekcia rozšírení jazyka, čo umožňuje ich jednoducho integrovať do existujúceho ale aj nového kódu.

Spomínané abstrakcie prinášajú jemno-zrnný dátový paralelizmus a paralelizmus vlákien, vnorený do hrubo-zrnného dátového a úlohového paralelizmu. Vedú programátora k tomu aby rozdelil problém na hrubé pod-problémy, ktoré môžu byť riešené nezávisle v paralelných blokoch vlákien. Každý pod problém je potom ešte delený na menšie časti tak, aby mohol byť riešený spoluprácou paralelne bežiacich vlákien v jednom bloku.

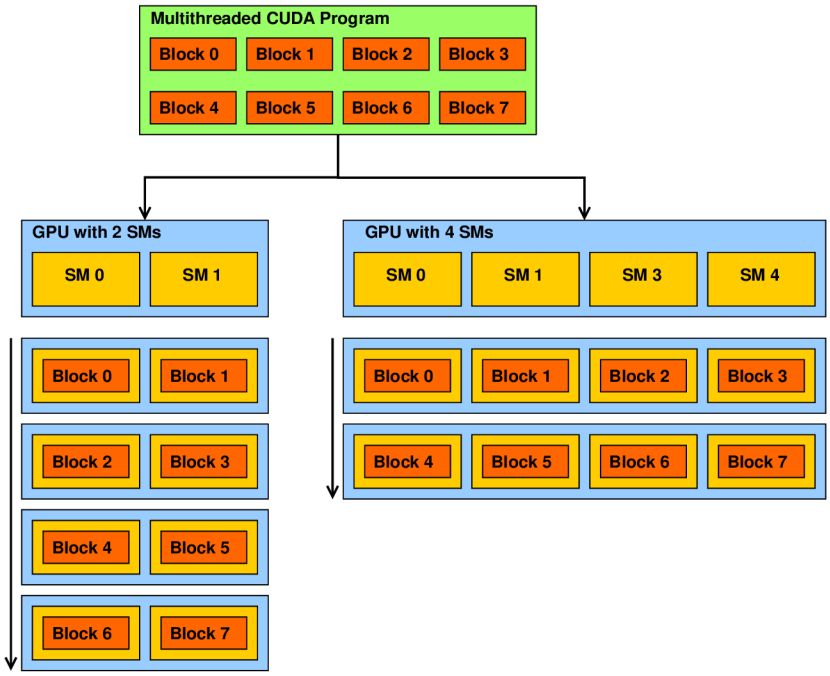
Toto rozdelenie zachováva vyjadrovacie možnosti jazyka tým, že dovoľuje vláknam spolupracovať na riešení pod-problémov zatiaľ čo umožňuje automatickú škálovateľnosť. Každý blok vlákien môže byť skutočne naplánovaný na ktoromkoľvek dostupnom multiprocesore v rámci GPU, v akomkoľvek poradí, paralelne alebo sekvenčne, tak aby skompilovaný CUDA program mohol byť spustený na akomkoľvek počte multiprocesorov, ako je ukázné na obrázku 2.4 a len systém spravujúci zariadenie musí vedieť fyzický počet multiprocesorov [25].

2.2 CUDA dôležité princípy

2.2.1 Kernel

CUDA C rozširuje jazyk C a čiastočne C++ (CUDA neovláda napríklad používanie objektov s metódami) tým, že pridáva programátorovi možnosť definovať funkcie, vo formáte jazyka C, zvané *kernely*. Keď je kernel funkcia spustená, tak dôjde k jej vykonaniu N-krát paralelne, naproti bežnému vykonaniu jeden krát. Každá inštancia kernelu je spracovávaná v rámci jedného z N spustených CUDA vlákien. Kernel funkciu je pri jej vytváraní nutné špecificky zadefinovať. Pri volaní funkcie kernelu je nutné určiť počet spustení kernelu definované pomocou veľkosti bloku a veľkosti mriežky. Každé vlákno, ktoré spúšťa kernel, dostane pridelené unikátne *identifikačné číslo vlákna* (threadIdx). Nepodobne všetky vlákna v rámci jedného bloku zdieľajú identifikačné číslo bloku (blockIdx), ktoré je medzi blokmi unikátne. ThreadIdx a blockIdx sú pre vlákna prístupné počas celého behu krenelu. Sú využívané algoritmami pre definíciu prístupu do pamäti a delenie práce medzi vlákna. Obrázok 2.5 ukazuje porovnanie volania krenelu oproti volaniu bežnej funkcie jazyka C [25].

[1] https://en.wikipedia.org/wiki/Moore's_law



Obr. 2.4: Automatická škálovatelnosť na platforme CUDA. Automatická distribúcia blokov medzi SM (Streaming Multiprocessor) na základe počtu SM [25].

2.2.2 Hierarchia vlákien

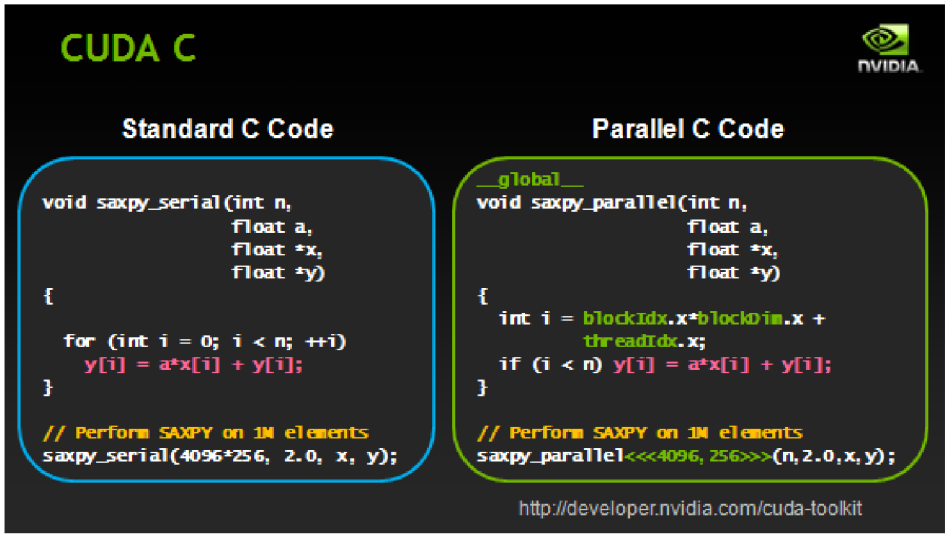
Premenná threadIdx je 3-zložkový vektor, ktorý umožnuje identifikovať vlákna pomocou jedno-/dvoj-/troj-rozmerného indexu vlákna. Vlákna zoskupené dokopy tvoria jedno-/dvoj-/troj-rozmerný *blok vlákien* (thread block). Toto poskytuje prirodzený spôsob ako spustiť výpočet naprieč prvkami tvoriacimi vektor, maticu alebo objem.

Maximálny počet vlákien na jeden blok je obmedzený. Je to z toho dôvodu, aby sa všetky vlákna v jednom bloku mohli nachádzať na jednom procesorovom jadre a zdieľať jeho obmedzené pamäťové zdroje. Na aktuálnych GPU je počet vlákien na jeden blok obmedzený na 1024. Napriek tomu je možné spustiť kernel na viacerých rovnako veľkých blokoch vlákien. Bloky sú, podobne ako vlákna, organizované do jedno-/dvoj-/troj-rozmernej *mriežky blokov vlákien* (grid of thread blocks) ako je zobrazené na obrázku 2.6. Počet blokov vlákien v mriežke je väčšinou daný veľkosťou spracovávaných dát alebo počtom procesorov v systéme, ktorý môže značne prekročiť.

Od blokov vlákien sa vyžaduje, aby ich bolo možné vykonávať nezávisle (paralelne alebo sériovo) a v ľubovolnom poradí. Táto požiadavka na nezávislosť dovoľuje blokom vlákien, aby boli plánované v ľubovolnom poradí na ľubovolnom počte jadier ako je ilustrované na obrázku 2.4. Toto dovoľuje programátorom písať kód, ktorý škáluje s počtom jadier.

Vlákna v rámci jedného bloku môžu spolupracovať prostredníctvom zdieľania dát cez *zdieľanú pamäť* (shared memory)). Prístup do tejto pamäti je ale nutné synchronizovať. Presnejšie je možné špecifikovať synchronizačné body pomocou bariér.

Bariéra funguje tak, že pred ňou všetky vlákna v bloku musia čakať predtým, než je ktorékoľvek vlákno pustené ďalej. Pre efektívnu kooperáciu sa očakáva, že zdieľaná pamäť má nízku odozvu a je blízko pri jadre procesora (podobne ako L1 cache). Podobne sa od funkcie bariérovej funkcie očakáva že bude nenáročná [25].



Obr. 2.5: Porovnanie kódu v C a v CUDA [26].

2.2.3 Hierarchia pamäti

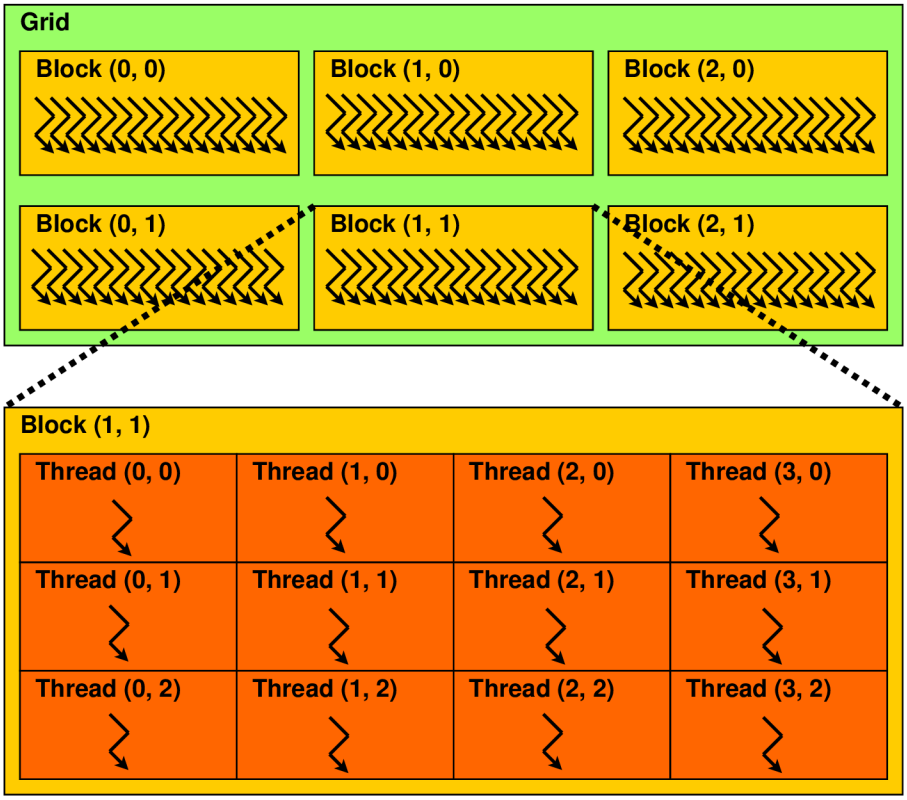
CUDA vlákna dokážu pristúpiť na dáta z viacerých pamäťových priestorov počas ich vykonávania (obr. 2.7). Každé vlákno má svoju privátnu lokálnu pamäť. Každý blok vlákien má spoločnú zdieľanú pamäť viditeľnú všetkým vláknam v rámci daného bloku. Táto zdieľaná pamäť má rovnakú životnosť ako samotný blok. Všetky vlákna majú prístup do rovnakej globálnej pamäti. Vlákna si taktiež medzi seba delia registre SM. Okrem toho existujú dve pamäťové priestory, slúžiace len na čítanie, dostupné pre všetky vlákna: pamäť konštánt a pamäť textúr. Globálna pamäť, pamäť konštánt a pamäť textúr sú optimalizované na rôzne použitia. Pamäť textúr poskytuje aj iné módy adresovania ako aj filtrovanie dát, pre niektoré špecifické dátové formáty. Globálna pamäť, pamäť konštánt a pamäť textúr zostávajú uchované medzi spusteniami kernelu v rámci jednej aplikácie [25].

2.2.4 Heterogénne programovanie

Ako ilustruje obrázok 2.8, programovací model CUDA predpokladá, že CUDA vlákna sú vykonávané na fyzicky separátnych zariadeniach, ktoré pracujú ako koprocesor ku hostiteľovi, ktorý vykonáva program v jazyku C. Ďalej sa predpokladá, že si obe strany(hostiteľ a zariadenie) udržujú vlastný separátny pamäťový priestor v DRAM. Tieto priestory sa nazývajú *pamäť hostiteľa* (host) a *pamäť zariadenia* (device). Program maá teda na starosti pamäť globálnu, konštánt a textúr, ktoré sú viditelné kernelom cez volania *CUDA runtime*. Toto zahŕňa aj alokáciu a dealokáciu a presun dát medzi hostiteľom a zaradením [25].

2.2.5 Compute Capability

Compute capability (výpočtové schopnosti) zariadenia je reprezentovaná číslom verzie, tiež niekedy zvané *"SM verzia"* ("SM version"). Pomáha identifikovať, ktoré funkcie hardware podporuje. Aplikácie sú schopné túto informáciu zistiť za behu. Toto číslo sa skladá z veľkej (X) a malej (Y) verzie v tvare X.Y. Zariadenie s rovnakým číslom veľkej verzie spadajú pod rovnakú architektúru (Maxwell, Kepler, Fermi ...) [25].



Obr. 2.6: Mriežka blokov [25].

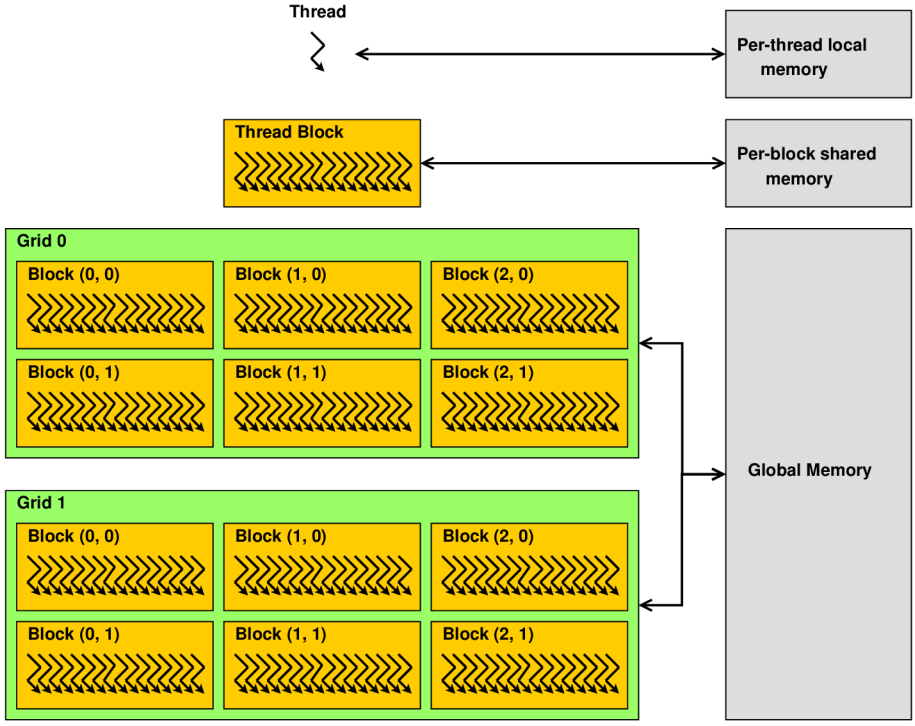
2.3 CUDA dôležité súčasti

CUDA platforma poskytuje programátorom dve API, ktoré s ňou umožňujú pracovať: Runtime API a Driver API. Runtime API je postavené nad nízko úrovňovým API v jazyku C, CUDA driver API. Driver API poskytuje programátorovi vyššiu úroveň kontroly nad aplikáciou tým, že sprístupňuje ďalšie koncepty ako: CUDA context, CUDA moduly. CUDA contex je analógia procesov pre zariadenie a CUDA modul je obdobou dynamicky pripojenej knižnice. Väčšina aplikácií ale nepotrebuje pracovať na nízkej úrovni a teda si vystačí s Runtime API.

2.3.1 Pamäť zariadenia

Ak bolo spomínané v sekcii o Heterogénnom programovaní (viď. 2.2.4), programovací model CUDA predpokladá, že systém je zložený z hostiteľa a zariadenia, kde obaja majú svoj vlastný oddelený pamäťový priestor. Kernely operujú z pamäti zariadenia, takže runtime poskytuje funkcie na alokáciu, dealokáciu a kopírovanie pamäti na zariadení. Okrem toho poskytuje runtime funkcie na presun dát medzi hostiteľskou pamäťou a pamäťou zariadenia.

Lineárna pamäť existuje na zariadení v 40-bitovom adresnom priestore, takže separátne alokované entity sa môžu navzájom na seba odkazovať pomocou ukazovateľov [24].



Obr. 2.7: Hierarchia pamäti [25].

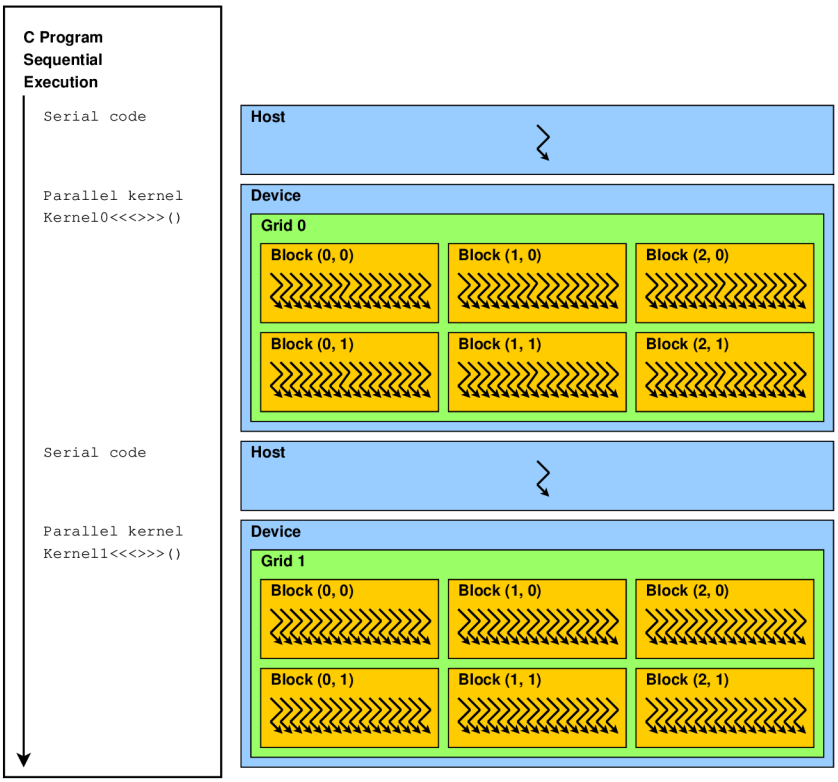
2.3.2 Asynchrónne paralelné vykonávanie

CUDA sprístupňuje nasledujúce operácie ako nezávislé úlohy, ktoré môžu byt vykonávané naraz:

- Výpočet na hostiteľskej strane.
- Výpočet na strane zariadenia.
- Presun dát z hostiteľskej pamäti do pamäti zariadenia.
- Presun dát z pamäti zariadenia do hostiteľskej pamäti.
- Presun dát v rámci pamäti daného zariadenia.
- Presun dát medzi zariadeniami.

Paralelné vykonávanie medzi hostiteľom a zariadením

Paralelné vykonávanie medzi hostiteľom a zariadením je umožnené pomocou asynchrónnych knižničných funkcií, ktoré navrátia kontrolu hostiteľskému vláknu pred tým, než zariadenie dokončí požadovanú úlohu. Pri používaní asynchrónnych volaní môžu byť mnohé operácie na zariadení zaradené do fronty a spracované CUDA ovládačom, až keď sú požadované zdroje dostupné. Tento prístup pomáha odľahčiť záťaž spravovania zariadenia z hostiteľského vlákna, aby toto vlákno mohlo vykonávať iné úlohy. Následujúce operácie na zariadení sú asynchrónne z pohľadu hostiteľa:



Obr. 2.8: Heterogénne programovanie [25].

- Spustenie krenelu.
- kopírovanie v rámci pamäti jedného zariadenia.
- kopírovanie z hostiteľskej pamäti do pamäti zariadenia, kde blok kopírovaných dát je do 64 KB.
- kopírovanie pamäti vykonávané funkciou so sufixom Async.
- Funkcie pre hromadné nadstavovanie hodnôt v pamäti.

Programátor môže globálne zakázať asynchrónne spúšťanie kernelov pre všetky CUDA aplikácie bežiace na systéme. Táto funkcia by sa však mala využívať len na účely odstraňovania chýb.

Spustenia kernelu sú synchrónne v prípade, že hardwarové počítadlá sú zberané pre profiler (Nsight, Visual Profiler) pokiaľ nie je povolené paralelné profilovanie kernelov. Asynchrónne kopírovanie pamäti sa tiež stáva synchrónnym ak zahŕňa hostiteľskú pamäť, ktorá nie je viazaná na stránku [24].

Paralelné vykonávanie kernelov

Niektoré zariadenia s compute capability 2.x a vyššou sú schopné spúšťať viac kernelov naraz. Aplikácie sú schopné si túto funkcionalitu overiť cez vnútorný atribút zariadenia. Maximálny počet naraz spustených kernelov závisí taktiež od compute capability zariadenia ako ukazuje tabuľka 2.1.

compute capability	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximálny počet naraz spustených kernelov	16	16	4	32	32	32	32	16

Tabuľka 2.1: Technická špacifkácia podľa compute capability [25].

Kernel z jedného CUDA kontextu nemôže byť vykonávaný zároveň s kernelom z iného CUDA kontextu. U Kernelov, ktoré využívajú veľa textúr alebo veľké množstvá lokálnej pamäti, je menej pravdepodobné, že budú vykonávané naraz s inými kernelmi. Stále totiž platí, že všetky vlákna, teda aj vlákna iných kernelov, sa delia o zdroje patriace SM [24].

Prelínanie presunov dát a vykonávania kernelov

Niektoré zariadenia sú schopné vykonávať asynchrónny presun dát na alebo z GPU popri vykonávaní kernelu. Ak presun dát zahŕňa hostiteľskú pamäť, táto pamäť musí byť hostiteľská pamäť viazaná na stránku. Taktiež je možné kopírovať v rámci zariadenia popri vykonávaní kernelu a/alebo s kopírovaním z alebo na zariadenie. Presuny dát v rámci zariadenia sú iniciované použitím štandartných funkcií na kopírovanie dát kde zdroj a ciel sa oba nachádzajú na rovnakom zariadení.

Presuny dát naraz

Niektoré zariadenia s compute capability 2.x a vyššou sú schopné prelínať kopírovanie do a zo zariadenia. Ak presuny dát zahŕňa hostiteľskú pamäť, táto pamäť musí byť hostiteľská pamäť viazaná na stránku.

2.3.3 Jednotný virtuálny adresný priestor

Pokiaľ je aplikácia spúšťaná ako 64-bitový proces, hostiteľ a všetky zariadenia používajú jeden spoločný adresový priestor pre compute capability 2.0 a vyššiu. Všetky alokácie hostiteľskej pamäti cez CUDA API volania a všetky alokácie pamäti na zariadení sú v rámci tohto virtuálneho rozsahu adries. Ako dôsledok:

- Lokácia ktorejkoľvek pamäte na hostiteľskej strane alokovanej pomocou CUDA, alebo na ktoromkoľvek zo zariadení, ktoré používajú jednotný adresný priestor, môže byť zistená z hodnoty ukazovateľa použitím.
- Keď dochádza ku kopírovaniu z alebo do pamäte ktoréhokoľvek zo zariadení, ktoré používa jednotný adresný priestor, je možné zistiť umiestnenie pomocou parametra predaného špecifickej funkcii. Toto funguje pre hostiteľské ukazovatele, ktoré neboli alokované cez CUDA alokáciu, pokiaľ dané aktívne zariadenie používa jednotné adresovanie.
- Alokácie pomocou `cudaHostAlloc()` sú automaticky neprenositeľné medzi všetkými zariadeniami, ktoré používajú jednotný adresný priestor, a ukazovatele získané cez `cudaHostAlloc()` môžu byť použité priamo z vnútra kernelov bežiacich na týchto zariadeniach.

2.4 Driver API

Nízko úrovňové imperatívne API založené na *popisovačoch* (handle). Na väčšinu objektov sa odkazuje pomocou netransparentných popisovačov, ktoré môžu byť predané funkciám aby bolo možné manipulovať dané objekty. Driver API je potrebné explicitne inicializovať predtým, než sa zavolá ktorákoľvek funkcia z tohto API. Následne je nutné vytvoriť CUDA kontext. V rámci CUDA kontextu, sú kernely explicitne načítavané ako PTX alebo binárne objekty. Kernely napísané v jazyku C musia teda byť kompilované separátne do PTX alebo binárnych objektov. Aplikácia, ktorá chce bežať na zariadeniach s architektúrami, ktoré ešte niesu dostupné, musí načítavať PTX a nie binárny kód. Toto je spôsobené tým, že binárny kód je špecifický pre každú architektúru a teda nekompatibilný s budúcimi architektúrami. Naproti tomu PTX kód sa kompiluje do binárneho kódu pri načítavaní ovládačom zariadenia.

2.4.1 Kontext

CUDA kontext je analógia ku CPU procesom. Všetky zdroje a akcie vykonávané v rámci driver API sú zabalené vnútri CUDA kontextu a systém automaticky upratuje tieto zdroje keď je kontext zrušený. Okrem objektov má každý kontext svoj vlastný adresný priestor. Z toho vyplýva, že sa hodnoty CUDA ukazovateľov z rôznych kontextov odkazujú na rôzne miesta v pamäti. Hostiteľské vlákno môže mať len jeden kontext zariadenia ako aktuálny. V momente keď je kontext vytvorený, tak sa stáva aktuálnym pre volajúce hostiteľské vlákno.

Každé hostiteľské vlákno má zásobník aktuálnych kontextov. Je možné vložiť nový kontext na vrch zásobníka. Podobne je možné odstrániť kontext zo zásobníka a tým daný kontext odpojiť od hostiteľského vlákna. Kontext je potom "volne plávajúciä môže byť priradený ľubovolnému hostiteľskému vláknu. Pri odstránení kontextu zo zásobníka sa tiež obnoví predchádzajúci kontext ak nejaký na zásobníku existuje. Každý kontext si uchováva počítadlo použití. Pri vytvorení kontextu sa jeho počítadlo nastavený na 1. Pri naviazaní alebo uvoľnení kontextu sa tiež upraví jeho počítadlo. Kontext je zrušený, keď počítalo použití klesne na 0 alebo ak je kontext explicitne zničený. Použitie počítadiel umožňuje prepájať funkčnosť s kódom od tretej strany pracujúcim v rovnakom kontexte [27].

2.4.2 Modul

Moduly sú dynamicky načítavané baličky dát a kódu pre zariadenie, podobné dll súborom vo Windows, ktoré sú výstupom NVCC kompilátoru. Modul exportuje nasledujúce prvky ako rozhranie:

- symboly
- funkcie
- globálne premenné
- textúry a povrchy

Všetky spomínané prvky sú udržiavané v rámci rozsahu modulu, a vystavené navonok ako body na prepojenie so zvyškom aplikácie. Týmto prístupom je umožnené aby mohli moduli napísané tretími stranami spolupracovať v rámci rovnakého CUDA kontextu [27].

Kapitola 3

Dátové štruktúry pre reprezentáciu priestorových dát

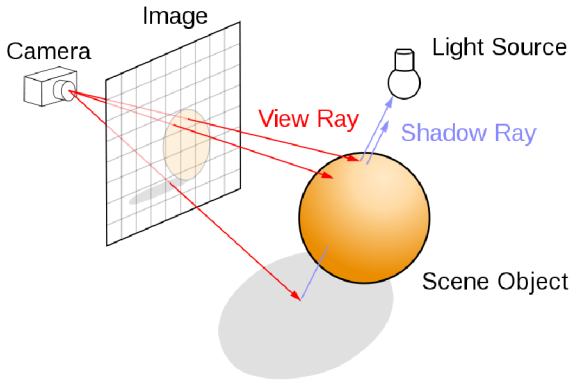
Táto kapitola začína stručným popisom zobrazovacej techniky sledovania lúčov (viď. 3.1), ktorou sa táto práca priamo zaoberá. Ďalej nasledujú sekcie pojednávajúce o akceleračných dátových štruktúrach využívaných pre 3d zobrazovacie techniky ako: hierarchia obalových telies (viď. 3.2.1), oktálový strom (viď. 3.2.3) a k-d strom (viď. 3.2.4). Na koniec je detailnejšie rozobraná štruktúra k-d strom (viď. 3.3), na ktorej stojí jadro tejto práce, a to konkrétnejšie jeden z variantov konštrukcie štruktúry k-d strom, prevzatý z práce [22].

3.1 Sledovanie lúčov

Sledovanie lúčov je 3d zobrazovacia technika, ktorý vytvára výsledný obraz sledovaním cesty svetla vrhnutého od pozorovateľa cez pixely v rovine obrazu do scény a simuláciou vplyvu ich stretnutia s virtuálnymi objektami v scéne. Patrí do skupiny algoritmov pre výpočet *globálneho osvetlenia* (global illumination) scény, čo sú techniky, ktoré prinášajú realistickejšie zobrazenie oproti jednoduchej 2d rasterizácii, pretože berú do úvahy nie len svetlo prichádzajúce priamo zo zdrojov svetla ale aj svetlo z týchto zdrojov, ktoré bolo odrazené od rôznych povrchov v scéne.

Týmto prístupom je možné simulovať široké spektrum vizuálnych efektov ako odraz, refrakcia, rozptyl a disperzné fenomény. Sledovanie lúčov je technika, ktorá dokáže produkovať vysoký stupeň vizuálneho realizmu, ale podobne za cenu vysokej výpočtovej náročnosti. Toto spôsobuje, že sledovanie lúčov je vhodné pre aplikácie, kde je možné si obraz dopredu predrenderovať nezávisle na potrebnom čase. Toto platí napríklad v prípade statických obrázkov alebo vizuálnych efektov vo filme a televízii. Sledovanie lúčov je ale menej vhodné pre aplikácie bežiace v reálnom čase, ako sú napríklad počítačové hry alebo simulátory, pretože v nich je dôležitá rýchlosť zobrazenia a responzívnosť aplikácie [7, 23].

Tento rýchlostný problém sa snažia zmeniť niektoré prístupy pre efektívnejšiu prácu s dátami scény. Značne skrátený čas výpočtu môže napríklad priniesť použitie vhodnej dátovej štruktúry pre reprezentáciu priestorových dát. Tieto dátové štruktúry sú vytvorené so zameraním na rýchly prístup ku priestorovým dátam. Čo je však nutné často riešiť je ich efektívna výstavba/prestavba pre aplikácie s meniacimi sa scénami. Postupom času bolo vyvinutých niekoľko dátových štruktúr s vhodnými vlastnosťami pre použitie s technikou sledovania lúčov, ktoré budú rozobrané v ďalších sekciách.



Obr. 3.1: Sledovanie lúčov buduje obraz pomocou vysielania lúčov do scény [1].

Popis algoritmu

Hlavnou výhodou sledovania lúčov je teda zobrazenie, ktoré je bližšie foto-realizmu vďaka prirodzeným vlastnostiam tejto techniky. Principiálne samotné sledovanie lúčov funguje tak, že sa vyšle lúč z pomyselného oka pozorovateľa cez zvolený pixel vo virtuálnej obrazovke a spočíta sa farba objektu, ktorý je vidieť cez tento pixel (obrázok 3.1). Scéna je zložená z objektov, ktoré boli zadefinované zvonka a sú uložené vo vhodných dátových štruktúrach (viď. 3.2) [1].

Typicky musí byť každý lúč testovaný na prienik s podmnožinu všetkých objektov v scéne. Keď je nájdený najbližší objekt v trajektórii lúča, algoritmus spočíta prichádzajúce svetlo v bode prieniku, preskúma materiálové vlastnosti zvoleného objektu, a zahrnie vplyv materiálu do výpočtu aby získal výslednú farbu pixelu a uhol odrazu pre odrazený/é lúč(e). Niektoré varianty algoritmu a reflexné alebo priesvitné materiály vyžadujú viac lúčov vrhnutých do scény z bodu prieniku.

Posielať lúče z oka do scény naproti prírode, ktorá posiela lúče zo scény do oka, sa môže zdať zvláštne. Dôvodom je to, že použitím tohto prístupu, čo do počtu, ušetríme niekoľko rádov vrhnutých lúčov a teda sme niekoľko násobne efektívnejší. Keďže valná väčšina lúčov vrhnutých zo svetla nedopadne do oka pozorovateľa, plytval by tento prístup veľké množstvo zdrojov na výpočty, ktoré pre nijakým spôsobom výsledok neovplyvnia.

Z toho dôvodu sa výpočet zjednodušuje tak, že predpokladáme, že daný lúč pretína pohľadové teleso. K ukončeniu sledovania lúča dochádza po maximálnom počte odrazov alebo po určitej prejdenej vzdialenosti bez prieniku s telesom. Následne sa z dáta zozbieraných lúčom/ami vypočíta výsledná farba pixelu [23, 21, 18].

3.2 Akceleračné dátové štruktúry pre sledovanie lúčov

Ako už bolo spomínané vyššie, akceleračné dátové štruktúry pre sledovanie lúčov sú vytvorené s ohľadom na rýchly prístup k dátam. Problémami týchto dátových štruktúr býva práve ich výstavba a prestavba v dynamických scénach, ktorá spotrebuje značné množstvo zdrojov. Keďže ide o podobné výpočty nad veľkým množstvom dát, naskytuje sa príležitosť využiť pre výstavbu týchto štruktúr paralelizmu.

[1] https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

3.2.1 Hierarchia obalových telies

Hierarchia obalových telies je štruktúra, ktorá využíva jednoduchých geometrických tvarov na obaľovanie častí scény, pretože je rýchlejšie počítať priesečník geometricky jednoduchým telesom. Nasleduje definícia obalového telesa za, ktorou sa nachádza detailnejší popis samotnej dátovej štruktúry.

Obalové telesá

V rámci počítačovej grafiky a výpočtovej geometrie, je obalové teleso, pre množinu objektov, také najmenšie priestorové teleso, pre ktoré platí, že plne obsahuje zjednotenie všetkých objektov z danej množiny. Obalové telesá sa využívajú na zvýšenie efektivity geometrických operácií, použitím jednoduchých objemových telies na obaľovanie zložitejších objektov. Pri jednoduchších telesách je totiž menej náročné zistiť ich priesečník s ľubovolným iným objektom [13].

Obalové telesá sa najčastejšie používajú na urýchľovanie rôznych typov testov nad priestorovými dátami (napr. Test na ne-prázdnosť prieniku dvoch telies).

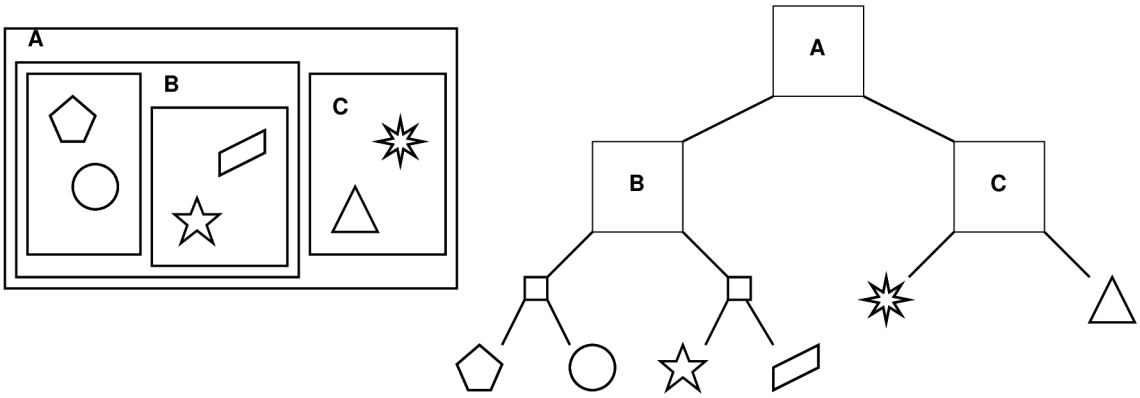
Pri sledovaní lúčov (viď. 3.1) sa obalové telesá používajú pre rýchle testovanie na prienik lúčov s objektami. U mnohých iných algoritmov sa využívajú na testy viditeľnosti, tzn. čo patrí do výhľadového telesa, alebo na kolízne testy. Pokiaľ lúč alebo pohľadové teleso nepretínajú obalové teleso, nemôžu pretínať ani žiaden z objektov obsiahnutých v rámci daného obalového telesa. Priesečníky, ktoré sú získané týmto výpočtom, sú použité ako zoznam objektov, ktoré treba vykresliť.

Testovanie obalového telesa na prienik býva typicky niekoľkonásobne rýchlejšie než testovanie objektu samotného, čo je spôsobené jednoduchosťou geometrie. Dôvodom je štruktúra objektu, ktorá je poväčšine tvorená polygónmi alebo dátovými štruktúrami, ktoré sú následne redukované na približnú polygonálnu reprezentáciu. V prípade, že sa obalové teleso ani nenachádza v pohľadovom telese, je plytvaním testovať každý polygón na prienik s pohľadovým telesom.

Pre získanie obalových telies komplexných objektov sa využíva rozklad objektov/scén na menšie časti pomocou grafu scény alebo špecifickejšie hierarchie obalových telies. Základnou myšlienkou tohto prístupu je rozložiť scénu do stromovitej štruktúry, kde koreňový uzol obsahuje celú scénu a každý potomok jej pod-časť. Najbežnejšími obalovými telesami sú obalové gule a obalové kvádre. U obalových kádrov rozlišujeme *osovo rovnobežné obalové kvádre* (axis-aligned bounding box (AABB)) a *orientované obalové kvádre* (oriented bounding box (OBB)). Výhodou AABB je jednoduchší výpočet priesečníkov s inými objektami a nevýhodou je že pri rotácii objektu je nutné obalové teleso prepočítať [13].

Popis hierarchie obalových telies

Hierarchia obalových telies (Bounding Volume Hierarchy - BVH) je stromová štruktúra vytvorená nad množinou geometrických objektov. Všetky geometrické objekty, ktoré tvoria listové uzly stromu, sú obalené obalovými telesami. Tieto uzly sú potom zoskupované do malých skupín a obalované väčšími obalovými telesami. A tieto obalové telesá sú opäť rekurzívne zoskupené a obalené ďalšou úrovňou obalových telies. Nakoniec získame koreňový uzol s jedným obalovým telesom obsahujúcim celú scénu (obrázok 3.2). Hierarchie obalových telies sú často využívané pre podporu operácií nad množinami geometrických telies, ako napríklad detekcia kolízií alebo sledovanie lúčov (viď. 3.1).



Obr. 3.2: Príklad hierarchie obalových telies s obdĺžnikmi ako obalovými telesami.
Vľavo: príklad priestorového rozdelenia telies v 2D.
Vpravo: Korešpondujúca hierarchia obalových telies k priestoru vľavo [2].

Napriek tomu, že obaľovanie objektov do obalových telies nám umožňuje zjednodušiť testy nad množinami geometrických objektov, stále vykonávame rovnaké množstvo porovnaní. Usporiadaním týchto obalových telies do hierarchie získame možnosť zredukovať počet porovnaní a tým znížiť časovú náročnosť na logaritmickú. Vďaka tejto hierarchii, pokiaľ pri testovaní priesečníkov dôjde k negatívnemu výsledku, môžeme vynechať prehladávanie celého podstromu.

Bežným problémom pri konštrukcii hierarchie obalových telies je vyhodnotenie dvoch protichodných cieľov. Na jednu stranu potrebujeme mať čo najjednoduchšie obalové telesá, aby sme ušetrili pamäť a zjednodušili testy na priesečníky a vzdialenosť. Na druhú stranu by sme chceli mať obalové telesá, ktoré čo najtesnejšie obopínajú v sebe obsahované objekty, aby sme pri testoch na priesečníky dostávali čo najmenej "planých poplachov"na prienik s objektom [13].

3.2.2 Binárne delenie priestoru

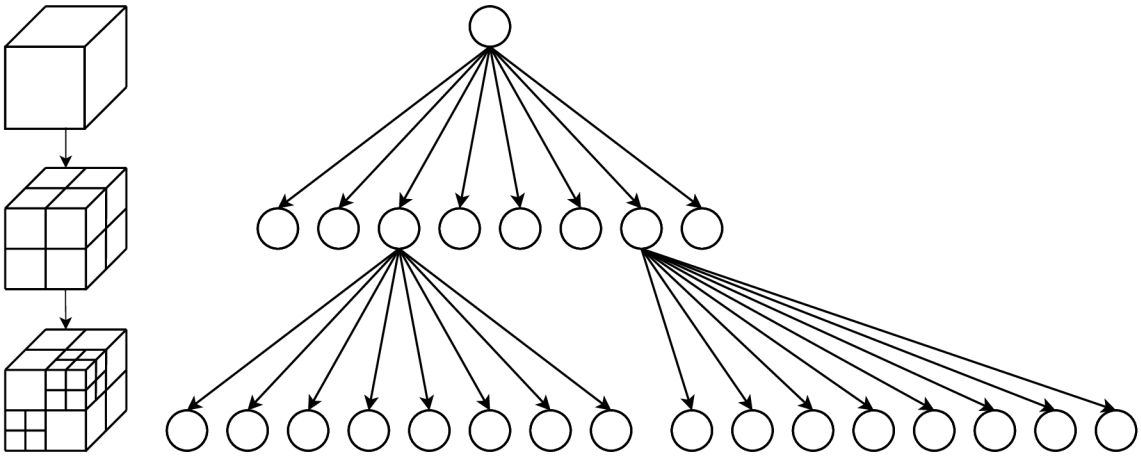
Binárne delenie priestoru (Binary space partitioning (BSP)) je metóda rekurzívneho delenia priestoru do konvexných množín podľa deliacich hyper-rovín. Výsledkom tohto prístupu je stromová štruktúra nazývaná BSP strom, ktorá obsahuje reprezentáciu všetkých objektov v scéne.

Binárne delenie priestoru bolo vyvinuté špecificky pre účely reprezentácie priestorových dát. Stromová štruktúra BSP dovoľuje efektívne sprístupnenie priestorových informácií o objektoch v scéne, čo je užitočné pre vykresľovanie. Jednu z týchto vlastností je napríklad usporiadanie objektov odpredu dozadu podľa vzdialenosti od pozorovateľa, čo pomáha pri testoch na to, ktoré teleso spôsobilo kolíziu s vrhnutým lúčom. Medzi využitia týchto štruktúr patri operácie nad geometrickými útvarmi, čo sa zužitkováva pri modelovaní v CAD softwaroch, detekcii kolízií v robotike a 3d hrách a pri priechode scénou technikou sledovania lúčov [20, 9].

[2] https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

3.2.3 Oktálový strom

Oktálový strom (Octree) je stromová dátová štruktúra, ktorej každý vnútorný uzol má presne 8 potomkov. Oktálový strom sú používané pre rekurzívne delenie 3d priestoru do ôsmich oktantov. Ku deleniu dochádza v každej dimenzii hyper-rovinami na "polovice", preto 8 potomkov. V 2d priestore sa využíva variant kvadrárny strom. Názov je odvodený z delenia priestoru troma vzájomne kolmými rovinami na osem rovnako veľkých častí = oktantov (obrázok 3.3). Oktálové stromy sa prevažne využívajú 3d počítačovej grafike a to hlavne v počítačových hrách.



Obr. 3.3: Vľavo: Rekurzívne delenie priestoru na oktanty.
Vpravo: Korešpondujúci oktárny strom [3].

Každý nelistový uzol oktálového stromu dalej delí svoju časť priestoru do ôsmych oktantov. U *oktálového stromu s bodovými regiónmi* (point region (PR) octree) uchováva každý uzol priestorový bod, ktorý je "stredom"pre ďalšie delenie v tomto uzle. Tento bod určuje následné rozdelenie priestoru do oktantov. Pri *oktálovom strome založenom na maticiach* (matrix based (MX) octree) je deliaci bod implicitne umiestnený do centra priestoru, ktorý uzol reprezentuje. Koreňový uzol PR oktálového stromu môže reprezentovať nekonečný priestor, naproti tomu koreňový uzol MX oktálového stromu musí reprezentovať konečný ohraničený priestor, aby bolo možné správne definovať jeho centrum. Oktálové stromy, na rozdiel od k-d stromov nevykonávajú delenie priestoru podľa dimenzií ale podľa centrálneho bodu. Okrem toho sú k-d stromy vždy binárne (pretože používajú rovnaké binárne delenie vo všetkých dimenziách), čo nie je pravda o oktálových stromoch [15, 8].

3.2.4 K-d strom

K-d strom (skratka pre k-dimenzionálny strom) je stromová dátová štruktúra pre reprezentáciu priestorových dát v k-dimenzionálnom priestore, založená na hierarchickom delení priestoru. V skutočnosti ide len o nadstavbu binárnych stromov pre troj a viac dimenzionálne priestory. Tieto dátové štruktúry sú vhodné pre aplikácie, ktoré potrebujú vykonávať výpočty na základe priestorových informácií, ako napríklad prehľadávanie určitého rozsahu objemu alebo vyhľadávanie najbližšieho susedného bodu/útvaru [2].

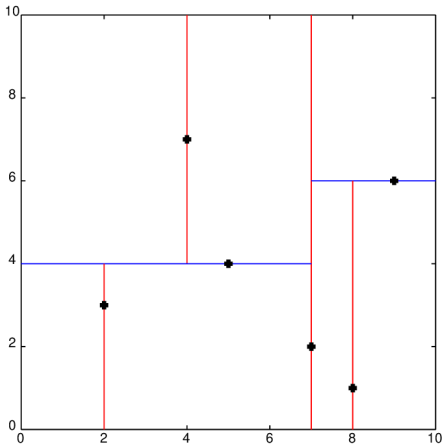
[3] https://en.wikipedia.org/wiki/Octree

K-d strom je binárny strom, ktorého každý uzol je priestorový bod definovaný k rozmeroch. Každý bod, ktorý nie je listom, definuje deliacu hyper-plochu, ktorá rozdelí priestor na dva pod-priestory. Body "naľavo"(z nižšou hodnotou súradnice v danej dimenzii ako súradnica deliacej plochy) od tejto hyper-plochy sa nachádzajú v ľavom pod-strome daného uzlu a obdobne body "napravo"(z vyššou hodnotou súradnice v danej dimenzii ako súradnica deliacej plochy) sa nachádzajú v pravom pod-strome [2].

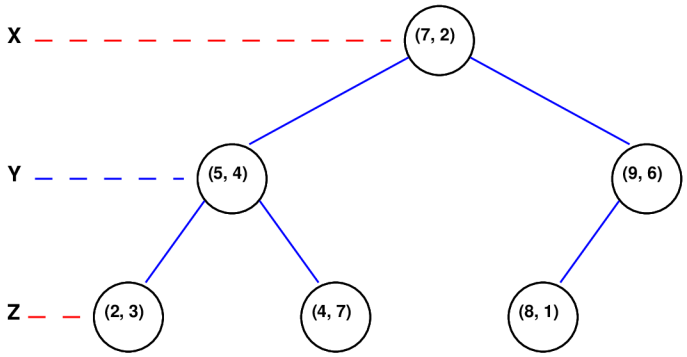
Existuje mnoho spôsobov ako zvoliť deliacu hyper-plochu, a tým pádom aj mnoho spôsobov ako vybudovať k-d strom. Kanonický spôsob výstavby k-d stromu má nasledujúce obmedzenia:

- Pri priechode stromom nadol dochádza pri výbere deliacej hyper-plochy ku cykleniu cez jednotlivé osi. (Napr. koreňový uzol má hyper-plochu kolmú na os x, jeho potomkovia kolmú na os y, ich potomkovia na os z a nasledujúca úroveň opäť na os x).
- Body sú do stromu vkladané tak, že sa z pomedzi bodov vkladaných do pod-stromu vyberie deliaci bod (napr. pomocou mediánu) s ohľadom na polohu týchto bodov na osi, ktorá je kolmá voči deliacej hyper-ploche.

Tento prístup produkuje vyvážené k-d stromy, v ktorých každý listový uzol je približne rovnako vzdialený od koreňového uzla. Vyvážené stromy však nemusia byť optimálnym riešením pre konkrétne aplikácie [3]. Obrázky 3.4a a 3.4b ukazujú príklad konštrukcie k-d stromu.



(a) Delenie k-d stromu pre množinu bodov: ((2,3), (5,4), (9,6), (4,7), (8,1), (7,2)) [?].



(b) Výsledný k-d strom [?].

Obr. 3.4: Ukážka konštrukcie k-d stromu.

Časová náročnosť

Výstavba statického k-d stromu z množiny n bodov má nasledujúcu časovú náročnosť:

- $O\big(n \cdot log^2(n)\big)$ v prípade, že sme pri hľadaní mediánu použili triediaci algoritmus s náročnosťou $O\big(n \cdot log(n)\big)$ ako napríklad *triedenie hromadou* (Heapsort) alebo *triedenie zlučovaním* (Mergesort)

- $O\big(n \cdot log(n)\big)$ v prípade, že sme pri výbere mediánu použili *algoritmus medián mediánov* [4] s časovou zložitosťou $O\big(n\big)$
- $O\big(k \cdot n \cdot log(n)\big)$ ak množina n bodov je dopredu zoradená, pre každú z dimenzií pomocou triedenia so zložitosťou $O\big(n \cdot log(n)\big)$ ako triedenie hromadou a triedenie zlučovaním, predtým než sa začne budovať samotný k-d strom.

Vloženie nového bodu do vyváženého k-d stromu ma zložitosť $O\big(log(n)\big)$.
Odstránenie bodu z vyváženého k-d stromu ma zložitosť $O\big(log(n)\big)$.
Vyhľadanie jedného najbližšieho suseda vo vyváženom k-d strome s náhodne distribuovanými bodmi ma priemernú zložitosť $O\big(log(n)\big)$ [4, 6, 5].

Body v listoch

Existujú ale aj iné možnosti definície k-d stromu napríklad tak, že iba listové uzly obsahujú body zo vstupnej množiny. Tento typ k-d stromov umožňuje množstvo iných mechanizmov na voľbu deliacej hyper-plochy, než delenie mediánom. *Pravidlo stredného bodu* (midpoint splitting rule) vyberie stred najdlhšej osi prehľadávaného pod-priestoru, nezávisle na priestorovom rozdelení bodov. Tento prístup zaručuje pomer rozdelenia prinajhoršom 2:1 ale výsledná hĺbka ja závislá na distribúcii bodov. Variant nazývaný *kĺzavý stredný bod* (sliding-midpoint), použije stredný bod iba ak by výsledok obsahoval body na oboch stranách delenia. Inak delí podla bodu najbližšie ku stredu [3].

Priestorové objekty

Ďalšou z možných modifikácií štruktúry k-d strom je uchovávať v rámci stromu trojuholníky alebo hyper-obdĺžniky namiesto bodov. Pri prehľadávaní pod-priestoru sú vo výsledku očakávané všetky útvary, ktoré pretínajú daný pod-priestor. Pri výstavbe stromu je nutné využiť prístup, keď sú všetky objekty (napr. trojuholníky) umiestnené v listoch [3].

Implicitné k-d stromy

Implicitný k-d strom je typ k-d stromu, ktorý je definovaný implicitne nad rektilineárnou mriežkou (rovnobežná mriežka kde je priestor rozdelený na bunky nepravidelnej veľkosti). Pozícia a orientácia deliacich hyper-plôch nie je daná explicitne, ale implicitne pomocou zvolenej rekurzívnej deliacej funkcie, definovanej nad hyper-obdĺžnikmi patriacimi uzlom stromu. Deliaca hyper-plocha každého uzla musí byť umiestnená na rovnakom mieste ako niektorá z plôch mriežky, čím delí mriežku uzla na dve pod-mriežky.

Min/max k-d stromy

Min/max k-d strom si v každom zo svojich uzlov uchováva hodnotu minima a maxima. Ide o minimálnu a maximálnu súradnicu v danej dimenzii pre zvolený uzol. Minimum/maximum uzla je minimum/maximum z rovnakých hodnôt jeho potomkov. U listov sú tieto hodnoty získané priamo zo súradníc objektov.

[4] https://en.wikipedia.org/wiki/Median_of_medians

3.3 Zvolený variant k-d stromu

Ako je spomínané v predošlej časti, k-d stromy majú mnoho variantov podľa toho s akými dátami chceme pracovať a aké vlastnosti požadujeme. Táto práca zakladá predovšetkým na prácach, ktoré napísali I. Wald a V. Havran [22, 14]. Podľa týchto materiálov bol zvolený variant k-d stromu, ktorý pracuje nad trojuholníkmi a má dáta uložené len v listových a nie vo vnútorných uzloch. Konštrukcia funguje obdobne ako u klasického k-d stromu spomínaného v 3.3, až na niekoľko odlišností.

3.3.1 Heuristika postavená na celkovom povrchu telesa

Prvou odlišnosťou je spôsob výberu deliacej hyper-plochy. Pre výber deliacej hyper-plochy sa používa *heuristika povrchu* (surface area heuristic (SAH)) [22]. SAH je heuristika postavená na celkovom povrchu telesa. SAH sa použije pri rozdeľovaní priestoru V na odhad ceny delenia podľa ceny oboch pod-priestorov (V_L,V_R) závislej na ich povrchu a počte trojuholníkov (N_L,N_R), ktoré sa v nich po delení nachádzajú. SAH podá odhad ceny pri priechode takto vytvoreným uzlom. Pre správne fungovanie vyžaduje SAH splnenie niekoľkých predpokladov:

1. Lúče sú nekonečné priamky uniformne rozložené v priestore.
2. Cena jedného kroku priechodu stromom a cena výpočtu priesečníku s trojuholníkom sú známe a sú to (K_T,K_I).
3. Cena výpočtu priesečníku s N trojuholníkmi je približne $N \cdot K_I$, teda lineárne závislá od počtu trojuholníkov.

S využitím týchto predpokladov pri danej konfigurácii platí, že pravdepodobnosť, že lúč, ktorý preťal voxel V, má pravdepodobnosť P že pretne sub-voxel $V_{sub} \subset V$:

$$P_{[V_{sub}|V]} = \frac{SA(V_{sub})}{SA(V)} \tag{3.1}$$

kde $SA(V)$ je povrch voxelu V.

Očakávaná cena $C_V(p)$ pre danú plochu p je potom jeden krok priechodu stromom a očakávaná cena pretnutia lúča a rozdelených voxelov (V_L,V_R):

$$C_V(p) = K_T + P_{[V_l|V]} \cdot C(V_l) + P_{[V_r|V]} \cdot C(V_r) \tag{3.2}$$

Analogicky je potom cena celého stromu T:

$$C(T) = \sum_{n \in nodes} \frac{SA(V_n)}{SA(V_S)} \cdot K_T + \sum_{l \in leaves} \frac{SA(V_l)}{SA(V_S)} \cdot K_I \tag{3.3}$$

kde V_S je *osovo rovnobežné obalové teleso* (axis-aligned bounding box (AABB)) celej scény S. Najlepší k-d strom T pre scénu S by bol taký pre ktorý nadobudne výsledok tejto rovnica čo najmenšiu hodnotu. Počet možných stromov však drasticky narastá s veľkosťou scény a nájsť globálne optimálne riešenie je v dnešných dňoch na netriviálnych scénach príliš náročné.

Namiesto globálneho optimálneho riešenia použijeme lokálne hladné prehľadávanie. To znamená, že cena delenia V pomocou p je spočítaná ako keby potomkovia V boli listové uzly.

$$C_V(p) \approx K_T + P_{[V_L|V]} \cdot |T_L| \cdot K_I + P_{[V_R|V]} \cdot |T_R| \cdot K_I \tag{3.4}$$

$$C(T) = K_T + K_I \cdot \left(\frac{SA(V_L)}{SA(V)} \cdot |T_L| + \frac{SA(V_R)}{SA(V)} \cdot |T_R| \right) \tag{3.5}$$

Týmto dôjde ku hrubému zjednodušeniu a výsledok býva prehnaný oproti skutočnej cene, keďže je pravdepodobné, že sa T_L a T_R ešte budú deliť. V praxi ale tento odhad funguje dobre pretože funguje konzistentne lepši než iné prístupy, ktoré sú lepšie v určitých špecifických prípadoch. Táto vlastnosť je jedným z dôvodov prečo bol pre túto prácu zvolený práve tento typ k-d stromu, pretože dosahuje dobrých výsledkov nezávisle na scéne [22].

3.3.2 Kritérium pre ukončenie konštrukcie k-d stromu

Okrem toho že je SAH metódou pre odhadovanie ceny potencionálnych kandidátov na delenie priestoru, je SAH schopný určiť kedy ukončiť rekurzívne delenie priestoru. Keďže cenu listového uzla sme schopný vyjadriť ako $C_{asLeaf} = K_I * |T|$, tak vieme povedať, že sa nám ďalšie delenie neoplatí ak cena najlepšieho delenia je vyššia ako cena nedelenia:

$$Terminate(V,T) = \begin{cases} true & ; min_p C_V(p) > K_I|T| \\ false & ; otherwise \end{cases} \tag{3.6}$$

Táto lokálna aproximácia sa dokáže jednoducho zaseknúť v lokálnom minime: keďže *lokálne hladný* (localy greedy) SAH nadceňuje $C_V(p)$, môže dôjsť k ukončeniu delenia, aj keď správna cena by znamenala ďalšie delenie. K tomuto fenoménu dochádza hlavne, keď je potrebné oddeliť ploché bunky po bokoch ne-plochej bunky. Pri nesprávne zvolených parametroch, alebo pri použití iných funkcií pre určenie ceny, môže byť rekurzia ukončená predčasne. Tento problém by bolo možné vyriešiť napevno napísaným kódom, to však nie je generické riešenie [22, 14].

3.3.3 Modifikácie a rozšírenie

V skutočnosti je väčšina predpokladov použitých pri odvodení rovnice 3.4 prinajmenšom pochybných:

1. Lúče väčšinou neprecheprejdú cez neprázdne voxely bez kolízie.
2. Hustota lúčov vrhnutých do scény väčšinou nie je uniformná.
3. Cena delenia na ľavú a pravú "polovicu"by nemala byť lineárna a obe strany a všetky listové uzly by mali mať konštantný faktor simulujúci jeden krok priechodu stromom.

Napriek tomu v praxi jednoduchý SAH ako je vysvetlený vyššie - lokálne hladný výber deliacej plochy s lineárnym odhadom ceny listových uzlov a automatickým kritériom pre ukončenie rekurzie - je často tým najlepším riešením a len niekoľko známych modifikácií prináša konzistentne lepšie výsledky. Spomedzi nich je najbežnejšou modifikáciou uprednostňovať delenie, ktoré orezáva kusy scény obsahujúce len prázdny priestor. Toho je možné dosiahnuť upraveným cenovej funkcie tak, že v prípade, že N_L alebo N_R sú nulové, je výsledná cena upravená o konštantný faktor [22]:

$$\lambda(p) = \begin{cases} 80\% & ; |T_L| = 0 \vee |T_R| = 0 \\ 1 & ; otherwise \end{cases} \tag{3.7}$$

Aby sa zabránilo zaseknutiu v lokálnom minime pri použití automatického kritéria pre ukončenie rekurzie, bolo zistené, že pomáha pokračovať v delení niekoľko krokov po tom, čo bolo splnené kritérium. Problémom však býva správne vyhodnotenie pre generické scény. Okrem toho niektoré implementácie využívajú namiesto ukončovania rekurzie na základe ceny, ukončenie v určitej predom definovanej maximálnej hĺbke. Týmto prístupom je možné ušetriť značné množstvo pamäti [22, 14].

3.3.4 Výber deliacej plochy

V predchádzajúcich sekciách bola popísaná cenová funkcia pre odhad ceny delenia pre zvolenú deliacu plochu p. Predpokladom pre tento výpočet je, že hodnoty N_L a N_R a rozmery V_L a V_R sú známe. Keďže celkový počet možných deliacich plôch je nekonečný, je nutné zvoliť systematický prístup, ktorý preskúma len konečnú podmnožinu všetkých možností a z nich vyberie tú najlepšiu. Pre akýkoľvek pár plôch (p_0, p_1), medzi ktorými nedochádza ku zmene N_L a N_R, sa ocenenie $C(p)$ mení lineárne len v závislosti na súradnici x_p plochy p. Z toho vyplýva, že hodnota $C(p)$ môže dosahovať minimum len na miestach, kde dochádza ku zmene počtov N_L a N_R, a tieto miesta zodpovedajú konečnému množstvu deliacich plôch. Keďže cieľom použitia heuristiky SAH je odhaliť miesta, kde hodnota $C(p)$ dosahuje minima, budú predmetom záujmu kandidátne plochy, ktoré spĺňajú vyššie spomínané kritériá [22].

Jedným z primitívnych spôsobov ako zvoliť deliacu plochu je použitím 6 hraničných plôch tvoriacich AABB $B(t)$ daného pod-priestoru. Ide o veľmi jednoducho a rýchlo implementovateľný prístup, ktorý zaručí rýchle vybudovanie stromu. Problémom však je, že táto metóda je pomerne nepresná a môže dôjsť k tomu, že pri rozdeľovaní trojuholníkov skončí niektorý trojuholník nesprávne vo voxeli, s ktorým nemá žiaden priesečník. Intuitívnym riešením je na základe skúseností vytvoriť model pre testovanie na prekryv trojuholníkov a voxelov. V dôsledku problematického návrhu modelu, však tento test nemusí nepriniesť požadované výsledky. Pre malé voxely napríklad môže dôjsť k tomu, že je daný voxel plne obsiahnutý v $B(t)$, a teda nie je možné nájsť žiadnu kandidátnu plochu pre delenie. V záujme zachovania presnosti je vhodné najprv vykonať orezanie trojuholníka t podľa rozmerov obalujúceho voxelu V. Následne sú ako kandidátne plochy použité steny AABB $B(t \cap V)$ získaného orezaním trojuholníka t. Pri orezávaní je podstatné, dávať si pozor na špeciálne prípady ako sú napríklad "ploché bunky"(Bunky, ktorých šírka v niektorých súradniciach je nulová a teda nemajú žiaden objem.), aby nedošlo ku odrezaniu trojuholníka, ktorý patrí do vnútra takejto plochej bunky [22, 14].

3.3.5 Rozdelovanie trojuholníkov do N_L a N_R

Aby bolo možné vypočítať hodnotu 3.4 pre každú možnú deliacu plochu p, je potrebné aby boli známe počty N_L a N_R pre sub-voxely V_L a V_R. Pri výpočte ocenenia je podstatné si dávať pozor, podobne ako v predchádzajúcej sekcii, na ploché bunky. Pokiaľ sa v strede inak prázdneho voxelu nachádza trojuholník osovo rovnobežný s dvoma z troch osí, malo by podľa správnosti pri konštrukcii dôjsť ku dvom deleniam, ktoré postupne vygenerujú dva prázdne voxely a jeden plochý voxel obsahujúci len spomínaný trojuholník. Toto je ideálne riešenie ale vyžaduje špeciálny prístup pri výstavbe a priechode stromom. Pri výstavbe je nutné zabezpečiť, že trojuholníky ležiace vo vnútri plochej bunky sa do výsledku zarátajú, ale ne-rovnobežné trojuholníky, ktoré sa bunky len dotýkajú alebo ho pretínajú sa na výsledku neprejavia.

Ak by sa použil pre určovanie počtov N_L a N_R test na prekryv trojuolníka a voxelu, mohly by sa do výsledku zarátať aj trojuholníky, ktorých priesečník s voxelom je len hrana alebo bod. Podobne by mohlo dôjsť k tomu, že by sa trojuholníky ležiace v deliacej ploche p započítali do oboch sub-voxelov. Hlavným problémom v tomto prípade je neefektívnosť, ktorej je možné sa vyhnúť rozdelením množiny všetkých trojuholníkov T aktuálneho voxelu do troch podmnožín T_L, T_R, T_P. Ide o trojuholníky, ktoré majú ne-nulový prienik s $V_L \setminus p$, $V_R \setminus p$ a p.

$$T_L = \{t \in T | Area(t \cap (V_L \setminus p)) > 0\} \tag{3.8}$$

$$T_R = \{t \in T | Area(t \cap (V_R \setminus p)) > 0\} \tag{3.9}$$

$$T_P = \{t \in T | Area(t \cap p) > 0\} \tag{3.10}$$

Ak sú tieto hodnoty známe, je možné vyrátať hodnotu 3.4 a to tak, že jeden krát spojíme dokopy T_L a T_P a druhý krát T_R a T_P. Možnosť, ktorou dospejem ku nižšiemu výsledku, je hľadané ohodnotenie [22].

3.3.6 Konštrukcia k-d stromu

Pri konštrukcii k-d stromu budeme nasledovať jednoduchší variant algoritmu, ktorý má časovú zložitosť $O\big(n \cdot log^2(n)\big)$ podľa práce [22]. Tento variant budeme mierne modifikovať, aby bolo neskôr možné lepšie aplikovať postupy paralelizácie. Bolo by síce možné použitie rýchlejšieho a komplikovanejšieho variantu s časovou náročnosťou $O\big(n \cdot log(n)\big)$, nie je to ale nutné, pretože paralelizačné úpravy, ktoré budú popísané v ďalších kapitolách, je možné po minimálnych úpravách aplikovať na oba varianty a predmetom tejto práce nie je implementácia k-d stromu ako taká ale jej paralelizácia. Nasleduje popis kľúčových častí konštrukcie k-d stromu stávajúcich na predpokladoch z predchádzajúcich sekcií.

Algoritmus 1 Rekurzívna konštrukcia k-d stromu [22]

```
function RecBuild(triangles T, voxel V) return node
    if Terminate(T, V) then
        return new leaf node(T)
    (C_V(p), p, pside) = ChoosePlane(T, V) // Vyberá deliacu plochu p a pside
    (V_L, V_R) = SplitV(p, V) // Rozdelí voxel V na dva sub-voxely pomocu p
    (T_L, T_R) = SplitTriangles(T, p, pside) // p rozdelí T na dve podmnožiny
    return new node(p,RecBuild(T_L, V_L),RecBuild(T_R, V_R))
```

Z algoritmu 1 vidíme, že konštrukcia k-d stromu je rekurzívny algoritmus, ktorý sa skladá z niekoľkých krokov. Na začiatku je umiestnená podmienka ukončenia rekurzie, ktorá zabezpečuje, že sa algoritmus ukončí pri splnení zvolených predpokladov. Následne je nutné zvoliť deliacu plochu, podľa ktorej sa bude deliť priestor a trojuholníky. Toto má na starosti funkcia *ChoosePlane*, ktorá bude viac do hĺbky popísaná nižšie. Keď je zvolená deliaca plocha p, dôjde s jej pomocou k rozdeleniu voxelu V vymedzujúceho aktuálny priestor na dva sub-voxely V_L a V_R. Potom sú trojuholníky z množiny T rozdelené pomocou p do dvoch množín T_L a T_R korešpondujúcich s voxelmi V_L a V_R. Ako bolo spomínané vyššie, jeden trojuholník sa ale môže vo výsledku dostať do oboch množín v prípade, že pretínal deliacu plochu a teda zasahoval do oboch pod-priestorov. Pre oba pod-priestory V_L a V_R sa potom rekurzívne vyhodnocuje funkcia *RecBuild* [22].

Algoritmus 2 Cenové ohodnotenie danej konfigurácie pomocou SAH [22]

function C(P_L, P_R, N_L, N_R) **return** ($C_V(p)$)
  return $\lambda(p)\cdot(K_T + K_I\ (P_L \cdot N_L + P_R \cdot N_R))$

function SAH(p, V, N_L, N_R, N_P) **return** ($C_p, pside$)
  (V_L, V_R) = SplitV(p, V) // Rozdelí voxel V na dva sub-voxely pomocu p
  $P_L = \frac{SA(V_L)}{SA(V)}$; $P_R = \frac{SA(V_R)}{SA(V)}$
  $C_{pL} = C(P_L, P_R, N_L + N_P, N_R)$
  $C_{pR} = C(P_L, P_R, N_L, N_R + N_P)$
  if $C_{pL} < C_{pR}$ **then**
    return (C_{pL}, LEFT)
  else
    return (C_{pR}, RIGHT)

Pre konštrukciu k-d stromu je podstatná heuristika SAH (viď. 3.3.1), ktorá nám pomáha ohodnotiť možné deliace plochy podľa vhodnosti. Pre vypočítanie ocenenia pre zvolenú deliacu plochu p, je nutné najprv pomocou nej rozdeliť voxel V na dva sub-voxely. Následne sa vyrátajú pomery P_L a P_R celkovej plochy každého zo sub-voxelov oproti celkovej ploche pôvodného voxelu V. Tie sa potom použijú pri výpočte cien delenia C_{pL} a C_{pR} pre V_L a V_R. Ocenenie pre ľubovolný zo sub-voxelov sa počíta ako je spomínané v algoritme 2. Podstatnou súčasťou tohto výpočtu je správna práca s planárnymi trojuholníkmi. Ich počet je pri oceňovaní oboch sub-voxelov prirátaný ku príslušnému počtu trojuholníkov. Ocenenia oboch sub-voxelov sú potom porovnané a nižšie z nich je zvolené ako výsledné ocenenie C_p. Podobne je určený aj smer použitý pre prácu s planárnymi trojuholníkmi, ktorý zodpovedá tomu, ktoré z ocenení bolo zvolené ako výsledné.

Výpočet ceny pre daný voxel je možné ovplyvniť pomocou dopredu definovaných konštánt K_T a K_I. Pomocou parametra K_T je možné úpravou ceny priamo upraviť pravdepodobnosť, že dôjde k deleniu. Naproti tomu parameter K_I umožňuje upravovať cenu v závislosti na počte trojuholníkov v danom voxely [22].

Algoritmus 3 Voľba najlepšej deliacej plochy [22]

function ChoosePlane(T, V) **return** ($C_V(p)$, p, $pside$)
  initialize ($C_V(p) = inf, p =< empty >, pside =< empty >$)
  eventlist $E =< empty >$;
  for all k **in** $Dimensions$ **do**
    for all t **in** T **do**
      B = ClipTriangleToBox(t, V) // Vytvorí obalové teleso pre trojuholník t
      if B is planar **then**
        E.insert(e($B_{min}, k, |$))
      else
        E.insert(e($B_{min}, k, +$))
        E.insert(e($B_{max}, k, -$))
  // End of for
  Sort(E) // Sort events by defined rules
  ($C_V(p)$, p, $pside$) = FindPlane(N, V, E)

Predtým než dôjde ku samotnej voľbe deliacej plochy, je nutné uskutočniť niekoľko prípravných krokov. Ako ukazuje algoritmus 3, najprv dôjde ku inicializácii ceny, deliacej plochy a smeru delenia. Cenu je nutné inicializovať, aby bolo neskôr možné úspešne vykonať porovnanie s aktuálne vypočítanou cenou. Inicializáciu deliacej plochy a smeru delenia je možné využiť ku singalizácii, že nebola zvolená žiadna deliaca plocha (napr. nenašlo sa lepšie ohodnotenie / voxel neobsahoval žiadne trojuholníky). Pre voľbu najlepšej deliacej plochy je potrebné vytvoriť zoznam udalostí E. Každá udalosť e je tvorená pozíciou delenia, dimenziou delenia a typom udalosti. Typy udalostí sú koniec ($PLANE\_END, symbol : -$), planárny ($PLANE\_IN, symbol : |$) a začiatok ($PLANE\_START, symbol : +$). Do zoznamu udalostí E sú pre každú dimenziu a každý trojuholník postupne vkladané udalosti. Najprv je vždy z trojuholníka t pomocou aktuálneho voxelu V orezané jeho obalové teleso B. Pokiaľ je trojuholník t planárny v smere rovnobežnom s osou dimenzie k vloží sa do zoznamu jedna udalosť typu $|$. Inak sa do zoznamu E vložia udalosti pre začiatok a koniec trojuholníka t. Deliace pozície pre tieto udalosti sa získavajú z obalového telesa B a teda korešpondujú s hraničnými rovnami pre daný trojuholník t. Potom, čo je zoznam udalostí E naplnený, dôjde k jeho zoradeniu. Zoraďuje sa primárne podľa pozície delenia, sekundárne podľa dimenzie delenia a terciárne podľa typu udalosti, kde $END < IN < START$. Posledným krokom tejto časti je samotné hľadanie deliacej plochy [22].

Algoritmus 4 Nájdenie najlepšej deliacej plochy [22]

Predpoklady: zoznam E je zoradený podľa predom stanovených pravidiel
function FINDPLANE(N, V, E) **return** ($C_V(p)$, p, $pside$)
for all k **in** $Dimensions$ **do**
$N_{L,k} = 0, N_{P,k} = 0, N_{R,k} = N$
for $i = 0; i < |E|$; **do**
$pp = (E_{i,point}, E_{i,k}); p^+ = p^- = p^| = 0$
while $i < |E| \wedge E_{i,k} = pp_k \wedge E_{i,point} = pp_{point} \wedge E_{i,type} = -$ **do**
inc p^-; inc i
while $i < |E| \wedge E_{i,k} = pp_k \wedge E_{i,point} = pp_{point} \wedge E_{i,type} = |$ **do**
inc $p^|$; inc i
while $i < |E| \wedge E_{i,k} = pp_k \wedge E_{i,point} = pp_{point} \wedge E_{i,type} = +$ **do**
inc p^+; inc i
$N_{P,k} = p^|, N_{R,k}{}^\smile = p^|, N_{R,k}{}^\smile = p^-$
(C, $side$) = SAH(V, pp, $N_{L,k}$, $N_{P,k}$, $N_{R,k}$)
if $C < C_V(p)$ **then**
($C_V(p)$, p, $pside$) = (C, pp, $side$)
$N_{L,k} += p^+, N_{L,k} += p^|, N_{P,k} = 0$
return ($C_V(p)$, p, $pside$)

Algoritmus 4 popisuje agregáciu hodnôt potrebných pre výpočet ohodnotenia pomocou heuristiky SAH. Predpoklad zoradeného zoznamu udalostí nám umožňuje efektívnejšiu agregáciu hodnôt p^-, $p^|$ a p^+. Výpočet prebieha separátne pre jednotlivé dimenzie. Dôvodom je potreba sledovať hodnoty N_L, N_P a N_R pre počty trojuholníkov pre každú z dimenzií samostatne. Nasleduje priechod zoznamom udalostí, kde každá udalosť má možnosť ovplyvniť aktuálne rozdelene trojuholníkov. Najprv sa získa deliaca plocha pp prislúchajúca udalosti v zozname s indexom i. Taktiež sa inicializujú počítadlá pre jednotlivé typy udalostí p^-, $p^|$ a p^+. Potom dôjde ku agregácii počtov jednotlivých typov udalostí. Pokiaľ obsahuje udalosť

E_i deliacu plochu, ktorej dimenzia $E_{i,k}$ a deliaci bod $E_{i,point}$ sa zhodujú s dimenziou a deliacim bodom plochy pp, tak navýši počítadlo toho typu udalostí, ktorý sa zhoduje s typom udalosti $E_{i,type}$. Agregované hodnoty p^-, $p^|$ a p^+ sú využité pre úpravu aktuálnych hodnôt počtov trojuholníkov N_L, N_P a N_R. Tieto hodnoty sú vzápätí použité pre výpočet ohodnotenia pomocou heuristiky SAH. Výsledok sa porovnáva z predošlou najnižšou hodnotou a nižšia z hodnôt sa uchová pre ďalšie iterácie. Pre skončením aktuálnej iterácie je ešte nutné opäť modifikovať počty trojuholníkov, keďže niektoré agregované hodnoty sa prejavia až v ďalšej iterácii. Oproti tomuto algoritmu je možné vykonať úpravu a spojiť jednotlivé iterácie po dimenziách do jednej veľkej iterácie. Stále je však nutné samostatne si uchovávať informáciu o počtoch trojuholníkov pre každú z dimenzií [22].

Kapitola 4

Návrh

Pri návrhu paralelných aplikácií je nutné okrem všeobecne platných konceptov paralelného programovania zohľadniť aj špecifické atribúty zvolenej paralelnej platformy. Z toho dôvodu je podstatné pri implementácii jedného z variantov (CPU, GPU), pozerať aj na druhý variant. Táto kapitola rozoberá, akým spôsobom bola navrhnutá výsledná aplikácia, pre vývoj oboch variantov bok po boku, v sekcii 4.1. Sekcia 4.2 sa zaoberá tým, ktoré časti aplikácie je možné a vhodné paralelizovať a úpravami potrebnými pre správny chod GPU implementácie. O všeobecných konceptoch paralelného programovania a problémoch pri optimalizácii pojednáva v sekcii 4.2.2. Návrh oboch variantov bol podriadený vlastnostiam platformy CUDA popísaným v kapitole 2.

4.1 Spoločný návrh CPU a GPU variantov

Pokiaľ od testovania očakávame výsledky užitočné pre určenie efektivity zvolených optimalizácií je podstatné, aby sme na konci vždy porovnávali iba dve porovnateľné hodnoty. Z tohto dôvodu je potrebné hľadieť pri návrhu na oba varianty implementácie, aby nedošlo k tomu, že vo výsledku získame rýchlejšiu CPU implementáciu, lebo sme v nej vykonali úpravy, ktoré sme následne nezohľadnili v GPU implementácii. Za týmto účelom je CPU implementácia navrhnutá ako východzí bod pre GPU implementáciu. Znamená to teda, že nejde o najrýchlejší možný optimalizovaný variant konštrukcie k-d stromu, ale o vhodné východisko pre paralelnú implementáciu. Taktiež je ale potrebné počítať s tým, že GPU variant bude vyžadovať určitú réžiu na dátami, ktorá bude navyše oproti tomu čo bude v CPU variante.

Obe implementácie konštrukcie k-d stromu sú primárne založené na teórii spomínanej v pod-sekcii 3.3 a táto sekcia sa na uvedené popisy odkazuje. Implementácia CPU variantu sleduje túto predlohu a snaží sa zachovať aj značenie dôležitých prvkov aby bolo v implementácii možné jednoducho identifikovať kľúčové body. CPU implementácia využíva taktiež rovnakého členenia na logické sekcie vychádzajúce zo spomínanej teórie. Nejde však o jedna ku jednej kópiu predlohy, pretože bolo potrebné vykonať niekoľko úprav pre dosiahnutie lepších východzích vlastností. Najpodstatnejšie je však, že oproti GPU implementácii nie sú využité žiadne optimalizácie alebo efektívnejšie postupy výpočtov. Vďaka tomu môžeme CPU implementáciu brať ako referenčný bod a akékoľvek zrýchlenie získané z výsledkov poukazuje, čo sa týka efektivity, na čistý zisk. Ako je popísané v pod-sekcii 3.3.6, náš zvolený variant je miernou modifikáciou variantu z práce [22]. Spravidla ide o úpravy, ktoré približujú podobu CPU a GPU variantov bližšie k sebe.

Jednou z úprav, ktorú je potrebné explicitne spomenúť je úprava obmedzení na podmienku pre ukončenie rekurzívnej konštrukcie k-d stromu V prípade nášho variantu algoritmu budeme používať nasledujúce obmedzenia:

- Prvé obmedzenie: na maximálnu možnú hĺbku, ktorú strom počas delenia priestoru môže dosiahnuť. Toto obmedzenie nám umožňuje limitovať maximálne pamäťové nároky algoritmu za cenu ne-úplne rozdeleného výsledného stromu.
- Druhé obmedzenie: na efektívnosť delenia priestoru na základe parametrov K_T a K_I spomínaných vyššie, počte trojuholníkov T a rozmeroch voxelu V. Túto efektívnosť delenia nám presne vyjadruje cena C získaná pomocou heuristiky SAH popísanej v sekcii 3.3.1. Samotná podmienka je závislá na počte trojuholníkov a vyzerá nasledovne $C > K_I * |T|$. Táto podmienka by mala zabrániť zbytočnému deleniu, ktoré by prinieslo len minimálny alebo žiaden zisk.

4.2 Návrh GPU implementácie

Táto sekcia sa zaoberá časťami referenčného CPU variantu, ktoré zaberajú väčšinu času výpočtu a ich vhodnosťou pre paralelizáciu, ako aj špecifickými vlastnosťami CUDA platformy, na ktoré treba pri implementácii dávať pozor. GPU implementáciou sa v tomto prípade myslia len kernely platformy CUDA a úprava dát potrebná pred a po spustení týchto kernelov, pre ich správny beh. Zvyšok aplikácie je pre GPU variant znovu-použitý z CPU implementácie.

4.2.1 Časti aplikácie s potenciálom pre paralelizáciu

Pri prvotnej analýze CPU variantu bolo zistené, že väčšinu času bude tento variant venovať na výpočet jednotlivých úrovní delenia k-d stromu. Pretože ide o spracovávanie veľkého množstva rovnorodých dát, má tento výpočet potenciál pre paralelnú implementáciu. Túto časť je možné ďalej rozdeliť na menšie bloky s užším zameraním, ktoré je potom možné analyzovať samostatne. Nasleduje rozbor piatich krokov tejto časti a analýza ich vhodnosti pre paralelnú implementáciu:

I. Príprava zoznamu udalostí

Prvým krokom výpočtu, ktorý je pre každé delenie potrebné vykonať, je príprava zoznamu udalostí, z ktorého bude celý nasledujúci výpočet vychádzať. Vstupom pre tento krok je zoznam trojuholníkov T a obalové teleso V, ktoré dané trojuholníky obsahuje. Výstupom je zoznam udalostí naplnený podľa pravidiel stanovených v pod-sekcii 3.3.6. Pre každý trojuholník sú vyhodnotené jeho hranice vzhľadom na obalové teleso. Tie sú následne použité pri samotnej generácii udalostí do zoznamu. Z toho je vidieť, že tento krok vykonáva menší počet operácií nad veľkým množstvom dát, a teda má potenciál pre paralelnú implementáciu. Pre jednoduchosť je vhodné zlúčiť priechod jednotlivých dimenzií do jedného veľkého priechodu. Tento krok pomôže ušetriť inštrukcie a zjednodušiť prístup vlákien ku správnym dátovým položkám. Pre veľký počet prístupov do pamäti, ktoré tento krok vyžaduje, bude rýchlosť prístupu do pamäti jeho úzkym miestom.

II. Zoradenie zoznamu udalostí

Druhým krokom výpočtu, nasledujúcim prípravu zoznamu udalostí, je jeho zoradenie. Vstupom je naplnený zoznam udalostí E z prvého kroku. Výstupom je zoznam udalostí E zoradený podľa predom stanovených pravidiel. Pri zoraďovaní sa použijú 3 zoraďovacie kritériá:

- Primárne kritérium je pozícia trojuholníka t v zvolenej dimenzii k
- Sekundárne kritérium je dimenzia delenia k
- Terciárne kritérium je typ udalosti (tzn. koniec, stred, začiatok)

Zoraďovanie je bežným paralelným problémom a existuje množstvo paralelných implementácií rôznych typov radenia. Zoraďovanie je teda problémom vhodným pre paralelizáciu. V rámci tejto práce však paralelná implementácia zoraďovania nebude preskúmaná z dvoch dôvodov. Prvým dôvodom je už spomínané veľké množstvo existujúcich implementácií. Druhým podstatnejším dôvodom je, že pokiaľ by sme chceli CPU implementáciu dodatočne vymeniť za jej efektívnejší variant spomínaný v práci [22], museli by sme zoraďovanie prepracovávať, pretože by fungovalo na inom princípe. Možnosťou tiež bolo použiť niektorú z knižníc pre paralelné výpočty ako je napríklad `CUDA thrust`. Tento prístup však nemá z hľadiska tejto práce žiaden prínos. Ako bolo spomenuté vyššie, pre paralelizáciu sme si zvolili časti, ktoré sú rovnaké pre oba varianty.

III. Rozdelenie počtu trojuholníkov vzhľadom na deliacu plochu

Tretím krokom výpočtu je určenie počtov trojuholníkov vzhľadom na deliacu plochu. Vstupom je zoradený zoznam udalostí E. Výstupom sú počty trojuholníkov naľavo N_L, prelínajúcich N_P a napravo N_R od deliacej plochy. V pôvodnej predlohe sa vždy udržujú aktuálne hodnoty pre každú kombináciu pozície trojuholníka vzhľadom na deliacu plochu a dimenzie (tzn. celkom 9 hodnôt). Po našej úprave pre lepšiu paralelizáciu, ale udržujeme aktuálnu hodnotu pre každú kombináciu pozície trojuholníka vzhľadom na deliacu plochu a každú plochu. Narastie nám síce veľkosť uchovávaných dát, ale umožní nám to lepí paralelný prístup. Pre každú rôznu deliacu plochu sa teda vyráta trojica N_L, N_P a N_R, ktorá sa využije v ďalšom výpočte. Toho sa dosahuje postupným porovnávaním udalostí so zvolenou deliacou plochou. Z formy akou prebieha tento výpočet vyplýva, že je vhodný pre paralelizáciu. Problémom však je, že nedosahuje plnej efektivity CPU variantu. CPU variant totiž efektívne znovu-používa niektoré hodnoty a nepočíta ich nanovo. Výsledná rýchlosť teda bude závisieť v značnej miere aj na tom, či budeme schopný do výpočtu efektívne zapojiť dostatočný počet výpočtových jednotiek.

IV. SAH

Štvrtým krokom výpočtu je učenie hodnoty heuristiky SAH. Tento krok je úzko previazaný z tretím krokom. Vstupom je zoznam rôznych deliacich plôch *planes*, obalové teleso V, počty trojuholníkov N_L, N_P a N_R a parametre K_T a K_I. Výstupom je zoznam ocenení *costs* korešpondujúcich k príslušným deliacim plochám. Výpočet samotný obsahuje dva navzájom nezávislé prúdy, ktorých vnútorné inštrukcie sú ale vždy závislé na výsledku predchádzajúcej inštrukcie. V tomto prípade ide o výpočet, ktorý je náročný aritmeticky aj pamäťovo. Výpočet SAH je teda vhodný pre paralelizáciu ale je potrebné nájsť vyváženie medzi efektívnym prístupom do pamäti a skladbou výpočtových inštrukcií.

V. Redukcia hodnôt

Piatym a posledným krokom výpočtu je redukcia získaných hodnôt. Ak je bežné pri paralelných algoritmoch, výsledkom býva veľké množstvo paralelne získaných hodnôt. Aby sme na konci mali len jeden výsledok, je potrebné použiť metódu paralelnej redukcie. Vstupom je zoznam ocenení *costs* získaných pomocou heuristiky SAH. Výstupom je jedno najnižšie výsledné ocenenie, ktoré korešponduje s hľadanou deliacou plochou. Paralelná redukcia patrí medzi bežné metódy používané pri paralelizácii. Je teda vhodným prvkom pre paralelnú implementáciu. V prípade tejto implementácie sa budem snažiť aplikovať špecifické detaily platné pre tento konkrétny prípad.

4.2.2 Problémy návrhu paralelných aplikácií

Návrh a tvorba paralelných aplikácií vyžadujú oproti bežným neparalelným aplikáciám využitie nových prístupov. Existuje veľké množstvo spôsobov ako pareleliZovať ako aj miest kde je pralelizácia možná. Našim cieľom nie je prejsť ich všetky ale priblížiť tie, na ktoré určitým spôsobom sa zameriame. V tejto pod-sekcii sa nachádza popis niekoľkých prístupov, ktorých sa budeme držať. Detailnejší spôsob akým boli tieto postupy aplikované je popísaný v kapitole o implementácii 5.

I. Prístup do globálnej pamäti

Prvým z problémov, ktoré je pri každej paralelizácii na GPU potrebné riešiť je, efektívny prístup z GPU ku dátam poskytnutým zvonka. V prípade CUDA platformy sú tieto dáta uložené v globálnej pamäti. Je síce prístupná všetkým vláknam kernelu, ale prístupová doba je dlhá (400-800 cyklov) [16]. Pri prístupe do globálnej pamäti sa využíva tzv. *spájanie prístupu do pamäti* (coalescing). Prístup do globálnej pamäte je vyhodnocovaný po warpoch (32 vlákien). Pre efektívny prístup je vhodný čo najmenší počet čo najmenších transakcií. Proti neefektívnemu prístupu do pamäti pomáha zarovnanie počiatočnej adresy na násobok veľkosti segmentu. Segment je pamäť načítaná jednou transakčnou operáciou (32B, 64B, 128B). Dôležitý je tiež spôsob prístupu do pamäti, *prístupový vzor* (access pattern). Najlepšie výsledky sú dosiahnuté pri sekvenčnom prístupe, kde každé vlákno číta jednu dátovú položku z pamäti a tieto položky sú v pamäti uložené za sebou. Ak je *compute capability* verzie 1.2 alebo vyššej, nie je potrebné aby tieto vlákna pristupovali v sekvenčnom poradí, len aby každé pristúpilo ku jednej z adries v rámci daného bloku. Pokiaľ sa tieto pravidlá nedodržia, dochádza k tomu, že časť dátových položiek prenesená v rámci transakčných operácií nie je využitá, čo znižuje efektívnu priepustnosť zbernice. Taktiež je vhodné spracovávať väčšie množstvo položiek v jednom vlákne, pretože pri viacerých transakčných operáciách typu `LOAD` alebo `STORE`, je použité zreťazené spracovanie. Latencia prístupu sa obvykle pokrýva prepínaním vlákien (warpov) na multiprocesoroch. Všetky prístupy do globálnej pamäti idú cez L2 cache, ale nie je dobré sa na túto vlastnosť spoliehať pri nesprávnom prístupe do globálnej pamäti.

II. Pamäti v rámci kernelu

Práca s globálnou pamäťou nie je jediným pamäťovým problémom v rámci paralelného návrhu. Sústrediť sa treba tiež na použitie pamäti pre výpočty vo vnútri kernelu. Použitie globálnej pamäti pre vnútorné výpočty je príliš pomalé. Z toho dôvodu má každý multiprocesor k dispozícii pevne stanovený počet registrov a zdielanej pamäti. Tieto zdroje

sa rovnomerne rozdelia medzi všetky warpy a následne medzi vlákna v rámci jednotlivých warpov. Počet potrebných zdrojov pre jeden warp ovplyvňuje koľko warpov môže naraz bežať na jednom multiprocesore. Každé vlákno má svoje vlastné registre, ale zdieľanú pamäť spoločne zdieľa celý blok vlákien. Prideľovanie registrov má na starosti prekladač. Je vhodné snažiť sa ponechať prekladač aby mohol vložiť relevantné hodnoty do registrov čo je rýchlejšie ako prístup do zdielanej pamäti.

Zdielaná pamäť má pomerne nízku latenciu (len niekoľko cyklov) [16]. Okrem rýchlejšieho prístupu, je zdielaná pamäť lepšia aj pre nespájané prístupy do pamäti a rozosielanie rovnakých dát všetkým vláknam. Je možne nadstaviť rozdelenie pamäti medzi zdielanú pamäť a L1 cache. Kernely potrebujú viacero prístupov do pamäti a preto bolo ponechané pôvodné nastavenie: 48KB zdielaná pamäť / 16KB L1. Zdielaná pamäť je organizovaná po N bankoch o M bitoch (závislé na copute capability). Prístup do zdieľanej pamäti sa vyhodnocuje po warpoch (32 vlákien). Je dôležité vyhýbať sa tzv. „konfliktom bankov“ čo je prístup viacerých vlákien do jedného banku. V takomto prípade je prístup všetkých vlákien do tohto banku serializovaný. Výnimkou je situácia, keď všetky vlákna pol-warpu pristupujú na rovnakú adresu. Vtedy ide o rozosielanie rovnakých dát všetkým vláknam a vykoná sa len jedna inštrukcia pre získanie dát.

III. Priepustnosť inštrukcií

Celková priepustnosť inštrukcií je z časti závislá na konfigurácii spúšťania kernelov. Je potrebné spúšťať kernely nad dostatočným počtom vlákien aby sa efektívne využil výkon GPU. Množstvo vlákien v bloku je obmedzené na násobky 32, ale v závislosti na compute capability je možné určiť minimálne množstvo potrebných vlákien pre riešenie problémov ako sú oneskorenie a nedostatočné naplnenie. Pokiaľ niektorý z operandov inštrukcie, ktorú vlákno vykonáva nie je dostupný dôjde k zablokovaniu vlákna. Čítanie z pamäti nespôsobuje zablokovanie vlákien. Oneskorenie výpočtu vychádza na asi 18-22 cyklov [19]. Toto oneskorenie sa zakrýva prepínaním medzi rôznymi vláknami. Taktiež je ho možné skryť použitím navzájom nezávislých inštrukcií. Oneskorenie prístupu do pamäti závisí na spôsobe prístupu do pamäti a je úzko previazané s efektívnym naplnením multiprocesorv. Ďalším problémom priepustnosti býva divergencia vlákien. Pokiaľ sa viacero vlákien v rámci warpu rozhodne vykonávať rôzny kód, všetky tieto varianty sa serializujú. Rôzne warpy môže vykonávať rôzny kód bez vplyvu na výkonnosť.

IV. Efektívne naplnenie multiprocesorov

Efektívne naplnenie multiprocesorov (occupancy), je metrika, ktorá hovorí o efektívnom využití prostriedkov multiprocesorov. Ide o hodnotu udávanú v percentách, ktorá vyjadruje pomer počtu v danom momente skutočne vykonávaných vlákien N^{SM} oproti maximálnemu možnému počtu vykonávaných vlákien N^{SM}_{max} na multiprocesor:

$$Occupancy = \frac{N^{SM}}{N^{SM}_{max}} \tag{4.1}$$

Naplnenie je však limitované využitím zdrojov multiprocesorov. Ide napríklad o registre a zdielanú pamäť. Multiprocesor nemôže prideliť väčšie množstvo zdrojov než má, a teda môže naraz spracovávať len toľko warpov, koľko mu dovolia ich požiadavky na zdroje. Aj keď dosiahnuť 100% naplnenosť multiprocesorov nie je vždy možné, je dobré monitorovať obmedzenia konkrétnej aplikácie aby bolo množné odhaliť prípadné nedostatky.

Kapitola 5

Implementácia

Výsledná implementácia sa skladá z dvoch častí, CPU a GPU variantov. Implementácia CPU variantu z veľkej časti sleduje teóriu načrtnutú v kapitolách 3 a 4. Z toho dôvodu obsahuje sekcia CPU implementácie len ľahký popis a spomenuté použitie externých zdrojov a špecifických prvkov. Naproti tomu GPU implementácia popisuje riešenia problémov načrtnutých v kapitole 4 ako aj réžiu použitia CUDA kernelov pre jednotlivé časti aplikácie, ktoré boli paralelizované.

5.1 CPU Implementácia

Ako už bolo spomenuté skôr v tejto práci, väčšia časť CPU implementácie je znovu využívaná GPU implementáciou dokonca aj pri samotnej konštrukcii k-d stromu. Popis tejto implementácie sa teda bude zaoberať hlavne možnosťami jej parametrizácie a samotným riešením konštrukcie k-d stromu. Účelom tejto sekcie nie je poskytnúť detailný popis CPU implementácie ako takej.

5.1.1 Použité externé zdroje

Vzhľadom na to, že ide o rozsiahlejšiu aplikáciu, nie je cieľom implementovať každý prvok samostatne. Pre tieto účely boli využité niektoré externé zdroje riešiace špecifickú problematiku. Táto pod-sekcia poskytuje zoznam použitých externých zdrojov, krátky popis k čomu slúžia a ako boli v rámci projektu použité.

CUDA Samples hlavičkové súbory

CUDA Samples je súbor príkladov ako vytvárať CUDA aplikácie. Tieto príklady so sebou, ale taktiež nesú niekoľko hlavičkových súborov s funkciami pre uľahčenie práce programátorovi. Použitie týchto hlavičkových súborov spadá pod Cuda Samples EULA[1].

Konkrétne boli použité súbory `helper_cuda_drvapi.h` a `helper_string.h`. Zo súboru `helper_string.h` boli napríklad použité funkcie ako `getCmdLineArgumentValue()`, ktoré pomáhajú so spracovávaním parametrov príkazového riadku. `helper_string.h` je závislosťou pre použitie súboru `helper_cuda_drvapi.h`. Ten obsahuje napríklad makro pre pohodlnejšie spracovávanie chýb `__checkCudaErrors` a funkcie pre jednoduchšiu inicializáciu (`findCudaDeviceDRV`) a testovanie zariadenia (`_ConvertSMVer2CoresDRV`).

[1] http://docs.nvidia.com/cuda/eula/index.html

NVRTC

NVRTC je C++ knižnica, ktorá distribuovaná ako súčasť platformy CUDA. Jej účelom je preklad CUDA zdrojových kódov do formátu PTX za behu aplikácie. PTX (parallel thread execution) formát je zdrojový kód využívajúci špeciálnu inštrukčnú sadu pre GPU. Výsledný PTX „súbor“ je získaný vo forme reťazca jazyka C a môže byť ihneď použitý pre výpočty alebo uložený na disk pre neskoršie použitie.

Existuje možnosť nepoužiť NVRTC a preklad za behu vykonať zo separátneho procesu, ktorý spustí nvcc. Táto možnosť sa využívala, kým neexistovala podpora NVRTC a prináša nasledujúce nevýhody:

- Réžia prípravy pri tomto spôsobe kompilácie (napr. vytvorenie nového procesu a príprava a spustenie príkazu na príkazovom riadku) oproti použitiu NVRTC je zbytočne veľká.
- Koncový uživatelia sú nútení mať nainštalované nvcc a príbuzné nástroje, čo zhoršuje možnosti distribúcie takýchto aplikácií.

Použitie nvcc cez vedľajší proces má však aj jednu výhodu, ktorá ale bude možno s časom odstránená: Aktuálne NVRTC nepodporuje všetky možnosti parametrizácie prekladača, ktoré podporuje aktuálna verzia nvcc.

NVRTC teda umožňuje vziať predom pripravené CUDA súbory a za behu ich preložiť do formy PTX reťazca. Ten je potom možné načítať do samotného spúšťajúceho programu a zlinkovať s ďalšími modulmi, ktoré sme mohli získať prekladom vopred a/alebo za behu. Tento prístup prináša možnosť optimalizácií a urýchlenia, ktoré nie je možné dosiahnuť statickým dopredným prekladom.

Medzi podporované parametrizačné možnosti patrí: voľba cieľovej architektúry, voľba kompilácie relokovatelného kódu, ladiace informácie, obmedzenie maximálneho počtu registrov na funkciu, zachovávanie/nulovanie de-normalizovaných hodnôt pre plávajúcu rádovú čiarku, rýchla odmocnina/odmocnina so zaokrúhlením k najbližšiemu pre plávajúcu rádovú čiarku, rýchla delenie/delenie so zaokrúhlením k najbližšiemu pre plávajúcu rádovú čiarku, zlučovanie násobenia a sčítania do jednej operácie pre plávajúcu rádovú čiarku, definovanie makier preprocesora, definovanie cesty ku hlavičkovým súborom, povolenie podpory C++11.

Assimp

Assimp (celým menom Open Asset Import Library[2]), je C++ knižnica s otvoreným zdrojovým kódom slúžiaca pre načítavanie 3d modelov. Assimp podporuje široký výber bežne používaných formátov 3d modelov (napr. .obj, .3ds, .ply, .blend, .dxf, .fbx). Tieto formáty načítava uniformným spôsobom, čiže umožňuje pre užívateľa transparentnú prácu s dátami. Okrem toho umožňuje využiť aj rad funkcií pre dodatočnú úpravu načítaných dát do požadovanej podoby.

V rámci tejto práce bola knižnica Assimp použitá pre načítavanie 3d modelov, z ktorých má byť následne konštruovaný k-d strom. V aktuálnom stave podporuje aplikácia iba načítavanie modelov s jedným meshom. Čo sa týka úpravy dát, tak je využívaná triangulácia, pretože požadované vstupné dáta sú zoznam trojuholníkov. Zlúčenie duplicitných bodov je kozmetickou úpravou aby algoritmus nemusel prechádzať rovnaký bod viac krát.

[2] http://www.assimp.org/

5.1.2 Popis implementácie

CPU variant aplikácie bol implementovaný ako C++ projekt pre Microsoft Visual Studio. Aplikácia ako taká neobsahuje žiadne časti, ktoré by bránili prenositelnosti na iné platformy, ale jej funkčnosť nebola na iných platformách testovaná. Pri spustení dôjde najskôr ku spracovaniu parametrov a inicializácii potrebných štruktúr a CUDA platformy. Potom je spustená samotná konštrukcia k-d stromu.

Popis konštrukcie k-d stromu

Pri začiatku konštrukcie k-d stromu dôjde ako prvé k inicializácii. Okrem hodnôt, nad ktorými bude vykonávaný samotný výpočet, ako zoznam trojuholníkov T a obalové teleso V, sú inicializované aj rozmery mriežky pre spúšťanie kernelov na GPU (*threads_per_block*, *blocks_per_grid*), parametre pre výpočet cenovej funkcie SAH, K_T a K_I, maximálna hĺbka vnorenia *depth* a ukazovateľ voľby implementácie *is_cpu*.

Následne je spustená samotná rekurzia. Kontrola ukončovacieho kritéria je oproti teórii rozdelená na dve časti. Prvá časť sa nachádza na začiatku a kontroluje aktuálnu hĺbku v strome. Vďaka umiestneniu na začiatku nedôjde k zbytočnému výpočtu danej úrovne ak to nie je treba. Druhá časť sa nachádza pri konci z toho dôvodu, že ohodnotenie aktuálnej deliacej plochy vieme až po výpočte cenovej funkcie SAH. V tomto mieste implementácie dochádza ku voľbe medzi CPU a GPU variantom. Po tejto voľbe príde v CPU implementácii na rad príprava zoznamu udalostí, čo je implementované jednoduchým priechodom po trojuholníkoch pre jednotlivé dimenzie. Po príprave je zoznam udalostí zoradený. Radenie je identické pre oba varianty a využíva funkciu štandardnej C++ knižnice `std::sort()`. Potom sa prechádza ku hľadaniu deliacej plochy. Je vytvorený vektor deliacich plôch a k nemu korešpondujúce vektory počítadiel trojuholníkov. Tie sú pri priechode zoznamu udalostí naplnené navzájom rôznymi deliacimi plochami a adekvátnymi počtami trojuholníkov. Pre každú kombináciu deliacej plochy a počtov trojuholníkov je potom vyhodnotená cenová funkcia SAH a uložený výsledok. Deliaca plocha korešpondujúca k minimu z týchto výsledkov je hľadanou najlepšou deliacou plochou. Po nájdení deliacej plochy a kontrole druhej časti ukončovacieho kritéria, dôjde k rozdeleniu aktuálneho voxelu aj s trojuholníkmi na dva sub-voxeli. Pre každý sub-voxel je tento proces zopakovaný.

Možnosti parametrizácie

Vzhľadom na charakter aplikácie, ktorá bola vytvorená v rámci tejto práce, poskytuje jej rozhranie možnosť nadstaviť si viacero parametrov:

- `-verbose` zapína/vypína výpis informácií o GPU zariadení a parametroch spúšťania aplikácie.
- `-cpu` prepína medzi voľbou CPU a GPU implementácie.
- `-device=<integer>` možnosť manuálnej voľby GPU zariadenia.
- `-block=<integer>` možnosť nastavenia počtu vlákien na jeden blok. Veľkosť mriežky za dopočíta z parametrov GPU.
- `-grid=<integer>` možnosť nastavenia počtu blokov na mriežku. Funguje len ak bol manuálne nadstavený aj počet vlákien na blok.

- `-depth=<integer>` možnosť nastavenia maximálnej hĺbky pre k-d strom.
- `-traversal=<float>` možnosť nastavenia ceny kroku priechodu stromom.
- `-intersect=<float>` možnosť nastavenia ceny výpočtu priesečníku s trojuholníkom.
- `-model=<string>` cesta k súboru s 3d modelom, z ktorého sa vyrobí k-d strom.

Pokiaľ si žilovatel nedefinuje veľkosť bloku aj mriežky a tieto hodnoty nebudú pre dané zariadenie možné, bude veľkosť mriežky okresaná na najväčšiu hodnotu, ktorá už sa dá použiť.

5.2 GPU Implementácia

Ak už bolo spomenuté vyššie, táto sekcia popisuje iba časti aplikácie, ktoré boli upravované kvôli GPU implementácii. Sekcia je rozdelená podľa jednotlivých krokov výpočtu, ktoré prebehnú pri každej iterácii rekurzie. Pri každom kroku sa nachádza popis réžie okolo spúšťania krenelu ako aj popis samotného kernelu. Popisy kernelov stavajú na konceptoch návrhu paralelných aplikácií popísaných v pod-sekcii 4.2.2.

5.2.1 Príprava zoznamu udalostí

Prvý krok výpočtu naplní zoznam udalostí udalosťami, zodpovedajúcimi jednotlivým vstupným trojuholníkom. Maximálny možný počet udalostí je:

$$SIZE = 2 * 3 * sizeof(Event) * triangles.size() \tag{5.1}$$

Túto veľkosť je na rozdiel od CPU implementácie nutné na strane CPU vopred rezervovať. Zoznam udalostí nie je potrebné kopírovať na GPU a teda nie je žiadna réžia prenosu. Naproti tomu vstupný zoznam trojuholníkov, je okrem alokácie na GPU potrebné aj nakopírovať. Doba prenosu je priamo závislá od veľkosti zoznamu trojuholníkov ($SIZE$) a konkrétne hodnoty závisia na rýchlosti prenosu daného GPU. Pri výstupe je opäť réžia kopírovanie výsledného zoznamu udalostí ($SIZE$) späť na GPU.

Kernel ProcessEvents.cu

Tento kernel načíta trojuholník prislúchajúci jeho globálnemu indexu v rámci mriežky, ktorý pomocou obalového telesa V oreže. Výsledkom orezania je vytvorené nové obalové teleso, ktoré sa uloží do zdieľanej pamäti $sBox$. Potom na základe tohto nového obalového telesa naplní 3-6 udalostí prislúchajúcich jeho globálnemu indexu v rámci mriežky (1-2 udalosti pre jedno vlákno).

Prístup do globálnej pamäti je riešený pomocou cyklenia po $gridSize$. Každé vlákno načíta trojuholník korešpondujúci jeho globálnemu indexu vlákna v rámci mriežky i. Takto načítaný trojuholník je tvorený troma bodmi, z ktorých každý má tri súradnice. Vo výsledku to je `36B` prenesených dát na jedno vlákno. V tomto prípade nebolo využité žiadne zarovnanie, pretože pri zvolenom pamäťovom rozložení by to nepomohlo. Keďže ide o súvislý prístup do pamäti, kde žiadne prenesené dáta nie sú nevyužité. mali by požiadavky jedného bloku vyústiť na celkový prenos o deviatich `128B` transakciách (`4B` = float * 32 vlákien * 3 body * 3 súradnice).

Pamäti v rámci kernelu sú využívané k uloženiu pomocných hodnôt. Pomocné premenné, ktoré sú pamäťovo nenáročné a často sa do nich pristupuje, ako napríklad globálny index vlákna v rámci mriežky i a veľkosť mriežky $gridSize$, sú ponechané prekladaču aby si ich vložil do registrov. Naproti tomu pamäťovo náročnejšie obalové teleso V je uložené v zdielanej pamäti. Ide o kompromis medzi rýchlosťou a pamäťovou náročnosťou. Keďže každé vlákno pristupuje do zdieľanej pamäti len pomocou svojho indexu vlákna, ktorý je unikátny v rámci bloku, nedochádza ku žiadnym konfliktom bankov.

Priepustnosť inštrukcií je podobne ako naplnenie multiprocesorov podporená jednotným spúšťaním kernelov. Cyklus po $gridSize$ spôsobuje divergenciu len v prípade, že ide o posledný blok a teda divergencia v tomto prípade je nevyhnutná. Cyklus cez dimenzie je rovnaký pre všetky vlákna a nikdy nespôsobí divergenciu. Tento krenel obsahuje jeden podmienený blok `if-else`, ktorému sa bohužiaľ nevyhneme, ale jeho dopad je v podstate rovnaký, ako keby bol namiesto tohto bloku na rovnakom mieste sekvenčný kód z vnútra oboch častí bloku.

Efektívne naplnenie multiprocesorov je pokryté celkovým počtom spúšťaných vlákien. Vzhľadom na využitie registrov a zdieľanej pamäti však multiprocesory nebudú schopné spustiť maximálny počet warpov týchto krenelov naraz. Rýchly prístup do pamäti však v prípade tohto kernelu bude mať väčší dopad, keďže tento kernel neobsahuje veľké množstvo výpočtov.

5.2.2 Rozdelenie počtu trojuholníkov vzhľadom na deliacu plochu

Druhým krokom výpočtu je určenie počtov trojuholníkov vzhľadnom na pozíciu oproti deliacej ploche. Zoznam udalostí z predchádzajúceho kroku je po zoradení potrebné orezať o ne-inicializované udalosti na konci. Týmto získame zoznam udalostí identický s CPU implementáciou. Náročnosť tejto úpravy závisí od podielu osovo rovnobežných planárnych trojuholníkov v rámci scény (náročnosť je priamo úmerná ich počtu). Následne je vytvorený zoznam unikátnych deliacich plôch *planes* vychádzajúci zo zoznamu udalostí a 3 zoznamy počtov trojuholníkov N_L, N_P a N_R, ktoré majú rovnakú veľkosť ako zoznam deliacich plôch. Naplnenie zoznamu deliacich plôch je réžiou navyše a zhoršuje použitie tohto prístupu. Na GPU je potrebné kopírovať zoznam udalostí a zoznam deliacich plôch. Naspäť sa kopírujú všetky 3 zoznamy počtov trojuholníkov.

Kernel TriangleSplits.cu

Tento kernel najprv pre každé vlákno načíta prislúchajúcu deliacu plochu a udalosť. Následne prechádza každé vlákno v bloku po jednotlivých načítaných udalostiach a inkrementuje počítadlá korešpondujúce s deliacou plochou, ktorú načítalo.

Prístup do globálnej pamäti je riešený pomocou cyklenia po $gridSize$. A vnútorne dva krát po $BLOCK\_SIZE$. Každé vlákno načíta korešpondujúcu deliacu plochu a potom korešpondujúcu udalosť prislúchajúce jeho globálnemu indexu vlákna v rámci mriežky i. Vlákna pristupujú ku pamäťovým položkám sekvenčne a bez zarovnania. Všetky prenesené dáta sú využité. Pri načítaní deliacich plôch ide o `8B` (`4B` unsigned int, `4B` float) na vlákno, čo robí dokopy dve `128B` transakcie. U udalostí ide o `24B` (`8B` deliaca plocha, `12B` trojuholník, `4B` typ) na vlákno, čo vyústi na šesť `128B` transakcií.

Pamäti v rámci kernelu sú využívané k uloženiu pomocných hodnôt. Pomocné premenné, ktoré sú pamäťovo nenáročné a často sa do nich pristupuje, ako napríklad globálny index vlákna v rámci mriežky i a veľkosť mriežky $gridSize$ a premenné cyklov, sú ponechané prekladaču aby si ich vložil do registrov. Naproti tomu pamäťovo náročnejšie deliace

plochy a udalosti sú uložené v zdielanej pamäti. Ide o kompromis medzi rýchlosťou a pamäťovou náročnosťou. Keďže každé vlákno pristupuje do zdieľanej pamäti len pomocou svojho indexu vlákna, ktorý je unikátny v rámci bloku, nedochádza ku žiadnym konfliktom bankov. Okrem toho je pri prístupe ku udalostiam v najvnútornejšom cykle využité vlastnosti zdieľanej pamäti, kde ak všetky vlákna v rámci bloku pristupujú na jednu adresu zdieľanej pamäte, sú dané dáta rozoslané plošne celému bloku v rámci jednej operácie.

Priepustnosť inštrukcií je podobne ako naplnenie multiprocesorov podporená jednotným spúšťaním kernelov. Cyklus po *gridSize* spôsobuje divergenciu len v prípade, že ide o posledný blok a teda divergencia v tomto prípade je nevyhnutná. Podobne jednotlivé cykly po $BLOCK\_SIZE$ spôsobujú divergenciu len v prípade, že ide o posledný blok. Tento krenel obsahuje niekoľko podmienených blokov `if-else`, ktoré ale nemajú `else` alternatívu, čo znamená, že nedôjde ku žiadnemu sekvenčnému vykonávaniu.

Efektívne naplnenie multiprocesorov je pokryté celkovým počtom spúšťaných vlákien. Vzhľadom na využitie registrov a zdieľanej pamäti však multiprocesory nebudú schopné spustiť maximálny počet warpov týchto krenelov naraz. Väčší počet naraz spustených warpov by sa v prípade tohto krenelu vyplatil ale jeho nároky na zdroje nedovoľujú lepšie naplnenie multiprocesorov.

5.2.3 SAH

V treťom kroku sa nachádza vyhodnotenie cenovej funkcie SAH. Pre samotným výpočtom je potrebné vytvoriť pomocný vektory pre ohodnotenia *costs* o rovnakej veľkosti ako vektor *planes*. Na GPU je potrebné kopírovať zoznam zoznam deliacich plôch a všetky 3 zoznamy počtov trojuholníkov. Naspäť sa kopírujú len ohodnotenia získané funkciou SAH.

Kernel parallelSAH.cu

Tento kernel na začiatku pripraví pomocné parametre a obalové teleso. Nasleduje výpočet funkcie SAH. Kde sa po príprave obalových telies vyhodnotia priestorové podiely subvoxelov a potom sa učí cena delenia. Výsledky sa zapíšu do vektora ohodnotení.

Prístup do globálnej pamäti je riešený pomocou cyklenia po *gridSize*. Najprv sú jednorázovo pre každý blok, načítané pomocné parametre k_t a k_i a obalové teleso V. Každé vlákno načíta korešpondujúcu deliacu plochu a počty rozdelenia trojuholníkov vzhľadom na deliacu plochu N_L, N_P a N_R. Vlákna pristupujú ku pamäťovým položkám sekvenčne a bez zarovnania. Všetky prenesené dáta sú využité. Pri načítaní deliacich plôch ide o `8B` (`4B` unsigned int, `4B` float) na vlákno, čo robí dokopy dve `128B` transakcie. U počtov trojuholníkov ide o `4B` (unsigned int) na vlákno, čo vyústi na jednu `128B` transakciu na každý zo zoznamov.

Pamäti v rámci kernelu sú využívané k uloženiu pomocných hodnôt. Pomocné premenné, ktoré sú pamäťovo nenáročné a často sa do nich pristupuje, ako napríklad globálny index vlákna v rámci mriežky i a veľkosť mriežky *gridSize* a premenné cyklov, sú ponechané prekladaču aby si ich vložil do registrov. Naproti tomu pamäťovo náročnejšie obalové telesá sú uložené v zdielanej pamäti. Ide o kompromis medzi rýchlosťou a pamäťovou náročnosťou. Keďže každé vlákno pristupuje do zdieľanej pamäti len pomocou svojho indexu vlákna, ktorý je unikátny v rámci bloku, nedochádza ku žiadnym konfliktom bankov. V prípade celkového obalového telesa V je pri prístupe využitá vlastnosť zdieľanej pamäti, kde ak všetky vlákna v rámci bloku pristupujú na jednu adresu zdieľanej pamäte, sú dané dáta rozoslané plošne celému bloku v rámci jednej operácie.

Priepustnosť inštrukcií je podobne ako naplnenie multiprocesorov podporená jednotným spúšťaním kernelov. Cyklus po *gridSize* spôsobuje divergenciu len v prípade, že ide o posledný blok a teda divergencia v tomto prípade je nevyhnutná. Tento kernel neobsahuje žiadne iné podmienené bloky a teda nikde inde nedochádza ku divergencii.

Efektívne naplnenie multiprocesorov je pokryté celkovým počtom spúšťaných vlákien. Vzhľadom na využitie registrov a zdieľanej pamäti však multiprocesory nebudú schopné spustiť maximálny počet warpov týchto krenelov naraz. Rýchly prístup do pamäti však v prípade tohto kernelu bude mať väčší dopad, keďže tento kernel neobsahuje veľké množstvo výpočtov.

5.2.4 Redukcia hodnôt

Vo štvrtom a poslednom kroku výpočtu ide o zlúčenie hodnôt do jednej výslednej postupom paralelnej redukcie. Pre samotnou redukciou je potrebné naplniť vektor smerov prideľovania planárnych trojuholníkov *psides*. Náročnosť tejto operácie je úmerná počtu unikátnych deliacich plôch a jej nutnosť zhoršuje efektivitu výpočtu. Na GPU je kopírovaný iba zoznam vypočítaných cien. Kernel je spúšťaný dva krát. Medzi prvým a druhým spustením sú na CPU kopírované redukované ohodnotenia a indexy o veľkosti rovnej počtu blokov v mriežke $GRID\_SIZE$. Ako koncový výsledok je potom získaná jedno ohodnotenie a príslušný index.

ParallelReduction.cu

Tento kernel pre každé vlákno prečíta dve položky s ohodnotením a menšiu z nich si uloží. Pri tom si uloží aj ich index. Následne porovnáva dvojice hodnôt zo zmenšujúcim sa krokom až kým každému bloku nezostane len jedna hodnota.

Prístup do globálnej pamäti je riešený pomocou cyklenia po dvojnásobku *gridSize*. Je to z toho dôvodu, že každé vlákno pristupuje ku dvom pamäťovým položkám s ohodnoteniami. Každé z vlákien teda postupne načíta dve svoje položky s ohodnotením, ktoré porovnáva s aktuálne uchovávanou hodnotou. Vlákna pristupujú ku pamäťovým položkám sekvenčne a bez zarovnania. Všetky prenesené dáta sú využité. Pri každom načítaní ohodnotení ide o načítavanie `4B` (jeden float) na vlákno, čo robí dokopy jednu `128B` transakciu na každú z dvoch položiek. Pri zápise síce pristupuje do pamäti iba jedno z vlákien v každom bloku ale aj tak ide o jednu `128B` transakciu, ktorej efektívne využitie je nízke. To je spôsobené charakterom paralelnej redukcie.

Pamäti v rámci kernelu sú využívané k uloženiu pomocných hodnôt. Pomocné premenné, ktoré sú pamäťovo nenáročné a často sa do nich pristupuje, ako napríklad globálny index vlákna v rámci mriežky i a veľkosť mriežky *gridSize* a premenné cyklov, sú ponechané prekladaču aby si ich vložil do registrov. Ohodnotenia a indexy týchto ohodnotení sú načítané do zdieľanej pamäti. Umiestnenie do zdieľanej pamäti vyžaduje algoritmus paralelnej redukcie, aby bolo tieto hodnoty možné zdieľať v rámci bloku vlákien. Keďže každé vlákno pristupuje do zdieľanej pamäti len pomocou svojho indexu vlákna, ktorý je unikátny v rámci bloku, nedochádza ku žiadnym konfliktom bankov. Na konci sú dáta pre každý blok sústredené v jednej položke zdielanej pamäti.

Priepustnosť inštrukcií je podobne ako naplnenie multiprocesorov podporená jednotným spúšťaním kernelov. Cyklus po *gridSize* spôsobuje divergenciu len v prípade, že ide o posledný blok a teda divergencia v tomto prípade je nevyhnutná. Ostatné cykly spôsobujú divergenciu postupne a vždy odpadáva polovica vlákien s vyšším indexom čo zabezpečuje, že odpadávajú celé warpy a teda sa neplytvá výkonom. Tento krenel obsahuje niekoľko

podmienených blokov `if-else`, ktoré ale nemajú `else` alternatívu, čo znamená, že nedôjde ku žiadnemu sekvenčnému vykonávaniu.

Efektívne naplnenie multiprocesorov je pokryté celkovým počtom spúšťaných vlákien. Vzhľadom na využitie registrov a zdieľanej pamäti však multiprocesory nebudú schopné spustiť maximálny počet warpov týchto krenelov naraz. Rýchly prístup do pamäti však v prípade tohto kernelu bude mať väčší dopad, keďže tento kernel neobsahuje veľké množstvo výpočtov.

Kapitola 6

Testovanie

Ako bolo popísané v skorších kapitolách, celý návrh 4 a implementácia 5 oboch variantov aplikácie boli podriadené úmyslu testovať dosiahnuté výsledky. Za týmto účelom bola zvolená CPU implementácia ako referenčný bod pre GPU implementáciu. Cieľom tejto kapitoly teda bude poukázať na niektoré aspekty, ktoré sa podarilo dosiahnuť v rámci tejto práce. Prvá sekcia obsahuje vyhodnotenie úspešnosti efektívneho zaplnenia multiprocesorov pre jednotlivé kernely, spomínaného v rámci návrhu 4. Druhá sekcia obsahuje prezentáciu výsledkov merania a nadobudnuté urýchlenie oproti CPU variantu. V tretej sekcii je na základe dosiahnutých výsledkov navrhnutý postup pre efektívne využitie výpočtového výkonu pomocou prepínania medzi CPU a GPU implementáciu.

6.1 Efektívne naplnenie multiprocesorov

Efektívne naplnenie multiporcesorov je dôležitou metrikou, ktorá nám pomáha pri zistení výkonových obmedzení zariadenia. Ak sa pre daný kernel dosiahne optimálneho zaplnenia, sme schopný odhadnúť časovú náročnosť výpočtov pre rôzne množstvá vstupných dát (Bez réžie prípravy dát). Pokiaľ sme teda dospeli k optimálnemu naplneniu N warpov na jeden multiprocesor, grafický čip poskytuje M multiprocesorov a vykonanie jedného vlákna trvá D, tak sme schopný vyčísliť celkový čas T pre P vstupných trojuholníkov ako:

$$T = \left\lceil \frac{P}{32 * N * M} \right\rceil * D \tag{6.1}$$

Výsledná časová jednotka je závislá od časovej jednotky použitej pre parameter D. Ak už bolo spomínané, možnosti zaplnenia sú závislé na zdrojoch multiporcesorov využívaných jednotlivými blokmi. Čím viac zdieľanej pamäti alebo registrov daný blok potrebuje, tým menšie zaplnenie je možné dosiahnuť. Využitie zdieľanej pamäti a registrov nieje kumulatívne, ale obmedzenia z toho vyplývajúce sa odvodzujú podľa viac vyťaženého zdroja. K tomu aby sme boli schopný určiť množstvo jednotlivých zdrojov spotrebovaných pre každý blok daného kernelu, musel byť tento kernel preložený so špeciálnym parametrom `--ptxas-options=-v` pre rpekladač nvcc. Výstupom tohto príkazu je výpis všetkej použitej zdieľanej pamäti, lokálnej pamäti, pamäti konštánt a registrov, pre zvolený .cu súbor. Výstup tohto príkazu bol použitý ako základ grafov v nasledujúcej sekcii. V nasledujúcich pod-sekciách sú popísané jednotlivý kernely z hľadiska naplnenia multiprocesorov pomocou trojice grafov, vyjadrujúcich obmedzenia súvisiace s použitým algoritmom a architektúrov konkrétneho GPU. Všetky grafy (získané cez CUDA Occupancy calculator) pre jednotlivé kernely boli založené na zariadení s compute capability 2.1 .

6.1.1 Kernely ProcessEvents.cu a TriangleSplits.cu

Tieto dva kernely majú rovanké vstupná a teda aj výstupné hodnoty.

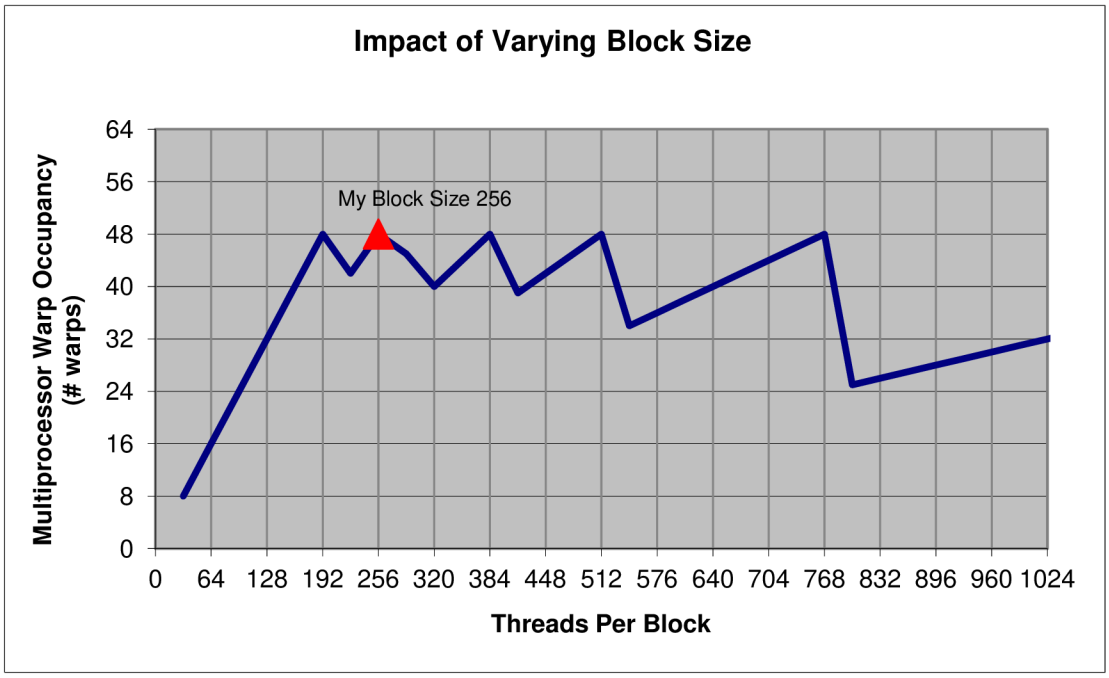
Výstup pre parameter prekladača `--ptxas-options=-v` :

- Maximálny pošet warpov na multiprocesor: 48
- Počet vlákien na blok: 256
- Počet registrov na blok: 20
- Počet zdieľanej pamäti na blok: 6144
- Naplnenie ($\frac{N}{48}$): 100%

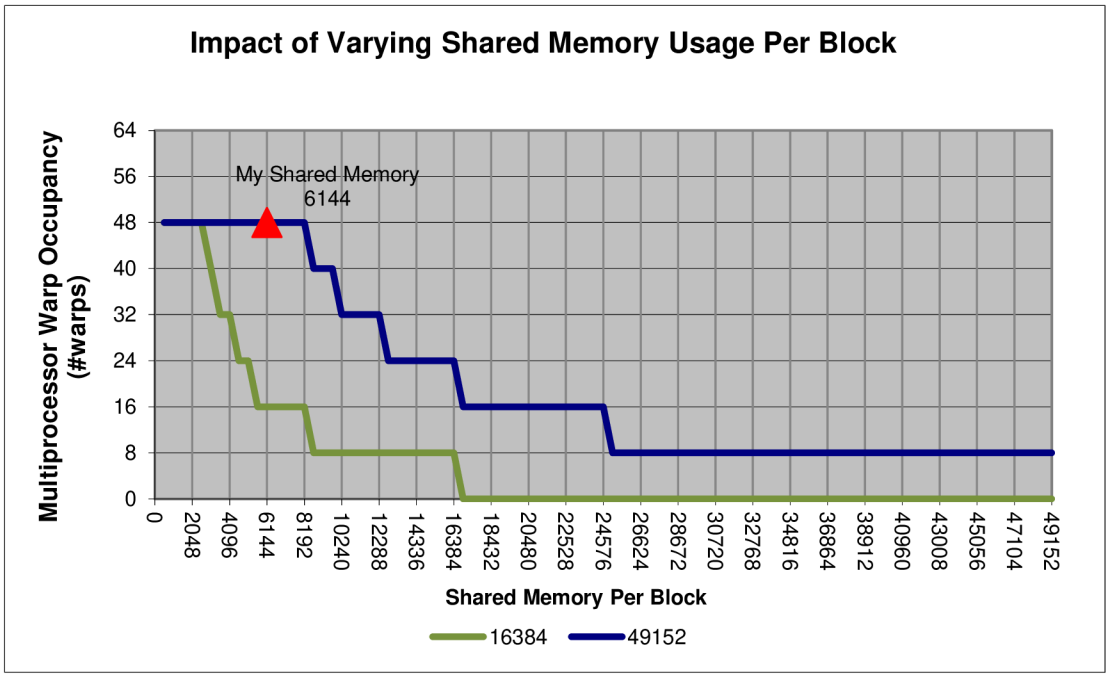
Krivka na obrázku 6.1 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili počet vlákien na blok. Je tu vidieť všeobecná charakteristika: pri voľbe veľkosti bloku 128 vlákien nie je dostatočný počet vlákien pre naplnenie multiprocesorov. Naproti tomu pri voľbe veľkosti bloku 1024 vlákien vidíme úpadok, ktorý má za následok narastajúca spotreba zdrojov multiprocesora so zvyšujúcim sa počtom vlákien na blok. Najvhodnejšími veľkosťami sú teda 256 a 512 vlákien na jeden blok.

Krivka na obrázku 6.2 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili množstvo využitej zdieľanej pamäti. Je vidieť, že pokiaľ tento kernel využije do 8192 Bajtov na blok, tak neutrpí žiadnu stratu na naplnení multiprocesorov. Taktiež je vidieť, že pri prepnutí na mód používajúci len 16384 Bajtovú zdieľanú pamäť by došlo ku značnému poklesu naplnenia multiprocesorov a preto táto zmena nie je vhodná.

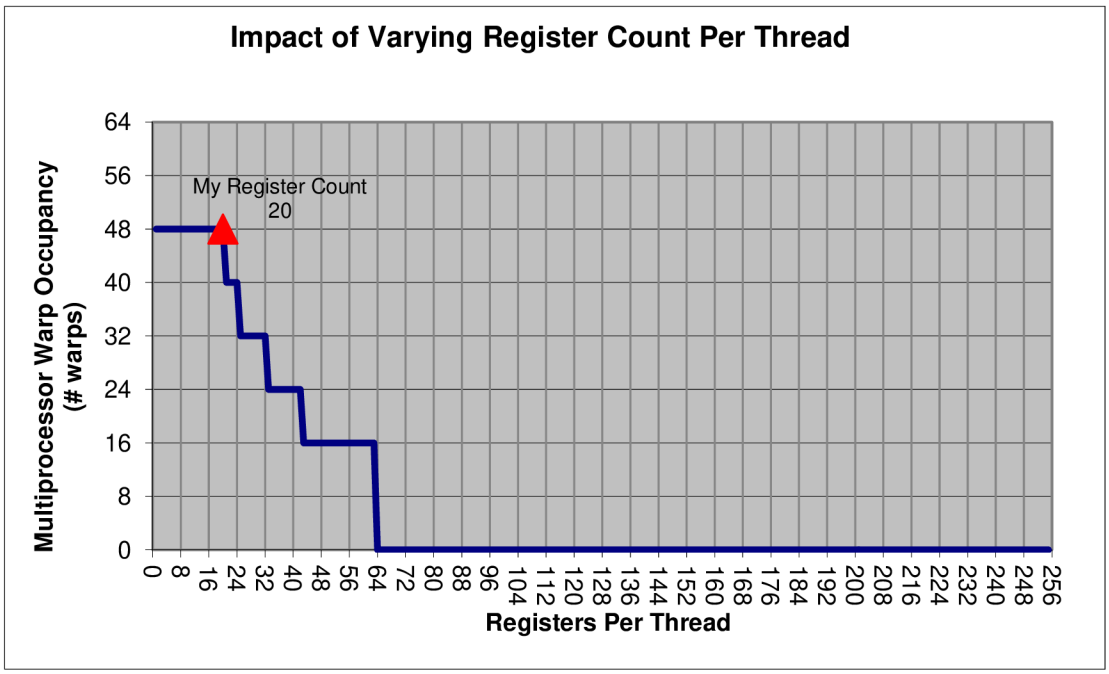
Krivka na obrázku 6.3 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili počet použitých registrov. Z grafu je vidieť, že pokiaľ by tento kernel potreboval na výpočet o 1 register viac, pokleslo by napleneni multiprocesorov o neclých 17%.



Obr. 6.1: Zaplnenie multiprocesorov pre rôzne veľkosti bloku



Obr. 6.2: Vplyv rôznych veľkostí použitej zdieľanej pamäti



Obr. 6.3: Vplyv rôzneho počtu použitých registrov

6.1.2 Kernel parallelSAH.cu

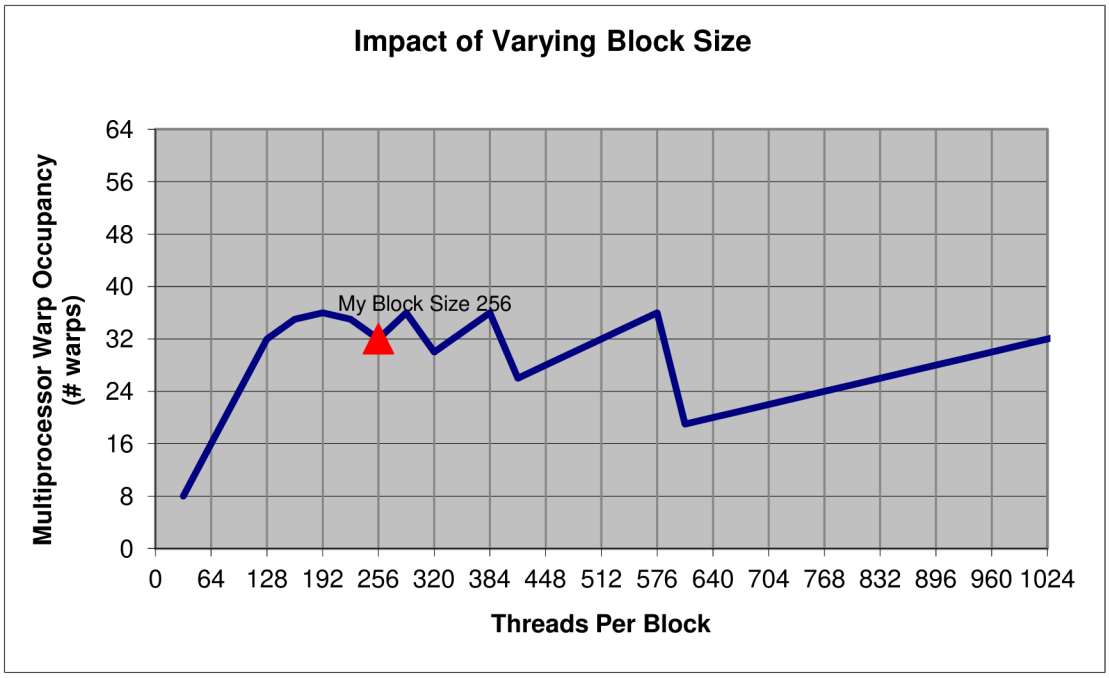
Výstup pre parameter prekladača `--ptxas-options=-v` :

- Maximálny pošet warpov na multiprocesor: 48
- Počet vlákien na blok: 256
- Počet registrov na blok: 27
- Počet zdieľanej pamäti na blok: 6144
- Naplnenie ($\frac{N}{48}$): 66,67%

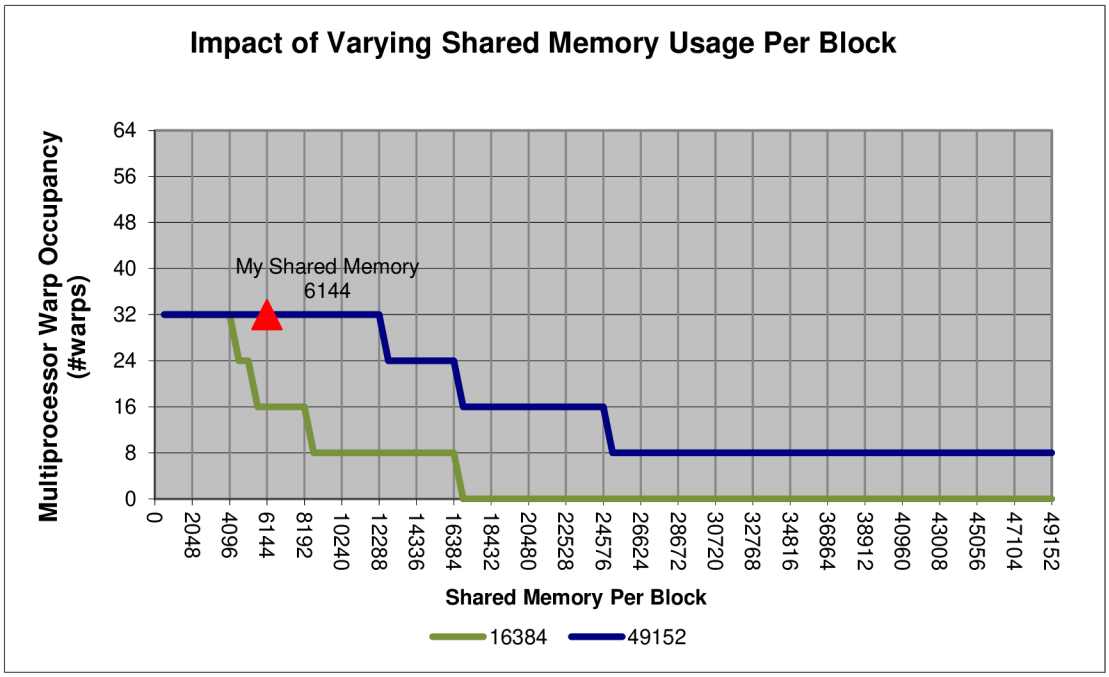
Krivka na obrázku 6.4 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili počet vlákien na blok. Naproti prvým dvom kernelom, pri veľkostiach bloku 128 a 1024 dostávame rovnaké naplnenie ako pri 256 a 512. Okrem toho sú tu ešte vyššie naplnenia na hodnotách, ktoré niesu mocninami čísla 2. To je spôsobené tým, že spotreba zdrojov obmedzila maximálne dosiahnuteľné naplnenie multiprocesorov.

Krivka na obrázku 6.5 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili množstvo využitej zdieľanej pamäti. Na rozdiel od prvých dvoch kernelov, by strata prepnutia na 16384 Bajtovú zdieľanú pamäť nemala až taký dopad. Pokiaľ by sme boli schopný znížiť využitie zdieľanej pamäti na 4096 na jeden blok, bolo by toto prepnutie bezstratové.

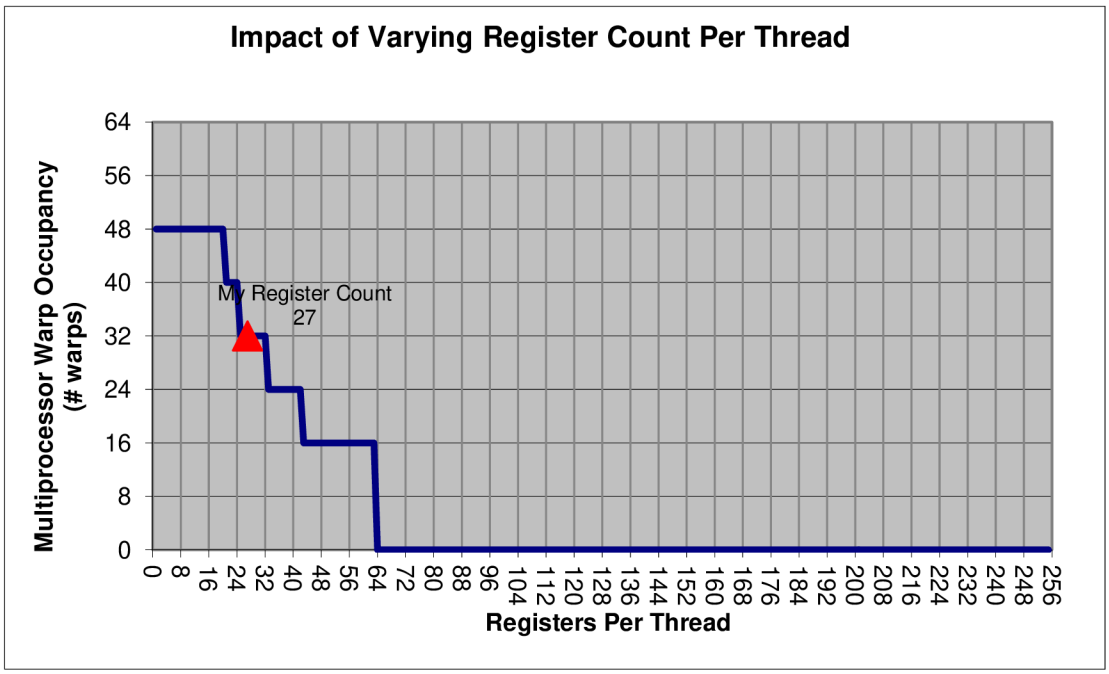
Krivka na obrázku 6.6 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili počet použitých registrov. Je vidieť, že najväčšiu stratu na naplnení multiprocesorov utrpel tento kernel na množstve potrebných registrov. Ak by sme boli schopní upraviť výpočet tak aby vyžadoval len 24 alebo 20 registrov, boli by sme schopný sa dostať na celkové naplnenie 83% respektíve 100%.



Obr. 6.4: Zaplnenie multiprocesorov pre rôzne veľkosti bloku



Obr. 6.5: Vplyv rôznych veľkostí použitej zdieľanej pamäti



Obr. 6.6: Vplyv rôzneho počtu použitých registrov

6.1.3 ParallelReduction.cu

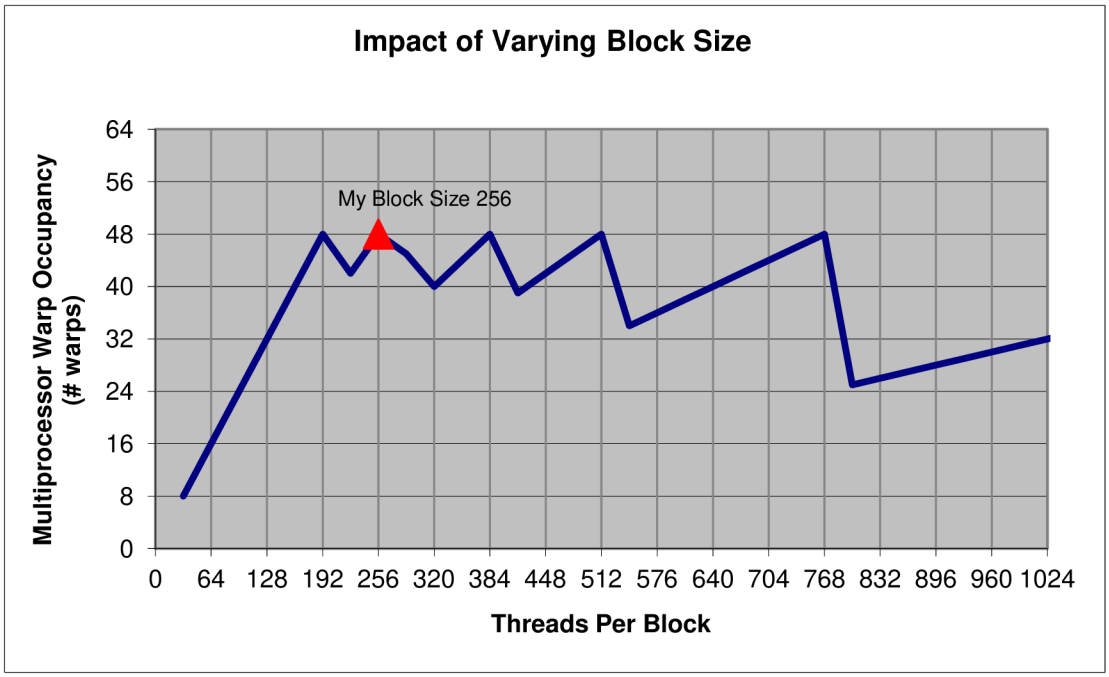
Výstup pre parameter prekladača `--ptxas-options=-v` :

- Maximálny pošet warpov na multiprocesor: 48
- Počet vlákien na blok: 256
- Počet registrov na blok: 12
- Počet zdieľanej pamäti na blok: 2048
- Naplnenie ($\frac{N}{48}$): 100%

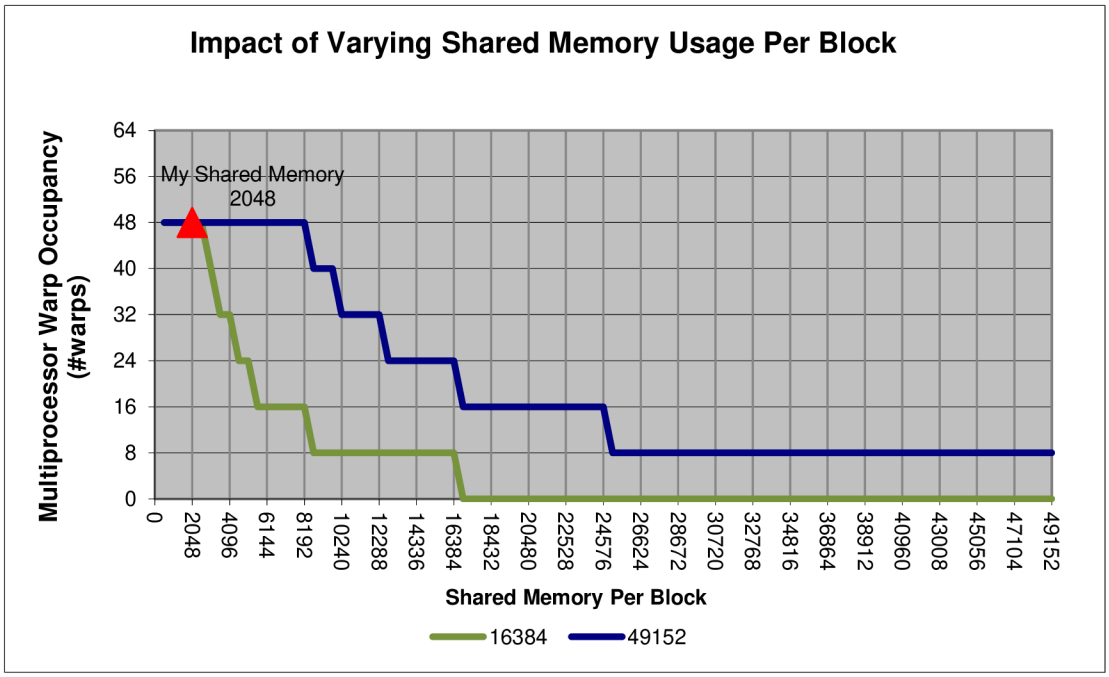
Krivka na obrázku 6.1 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili počet vlákien na blok. Podobne ako u prvých dvoch kerneleov je vidieť, že pri voľbe veľkosti bloku 128 vlákien nie je dostatočný počet vlákien pre naplnenie multiprocesorov a pri voľbe veľkosti bloku 1024 vlákien vidíme úpadok, ktorý má za následok narastajúca spotreba zdrojov multiprocesora so zvyšujúcim sa počtom vlákien na blok. Najvhodnejšími veľkosťami sú teda 256 a 512 vlákien na jeden blok.

Krivka na obrázku 6.2 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili množstvo využitej zdieľanej pamäti. Z grafu je u tohto kernelu vidieť, že spotreba zdieľanej pamäti je minimálna. Z togho vyplýva, že je možné prepnúť na mód používajúci len 16384 Bajtovú zdieľanú pamäť bez akejkoľvek straty na naplnení multiprocesorov.

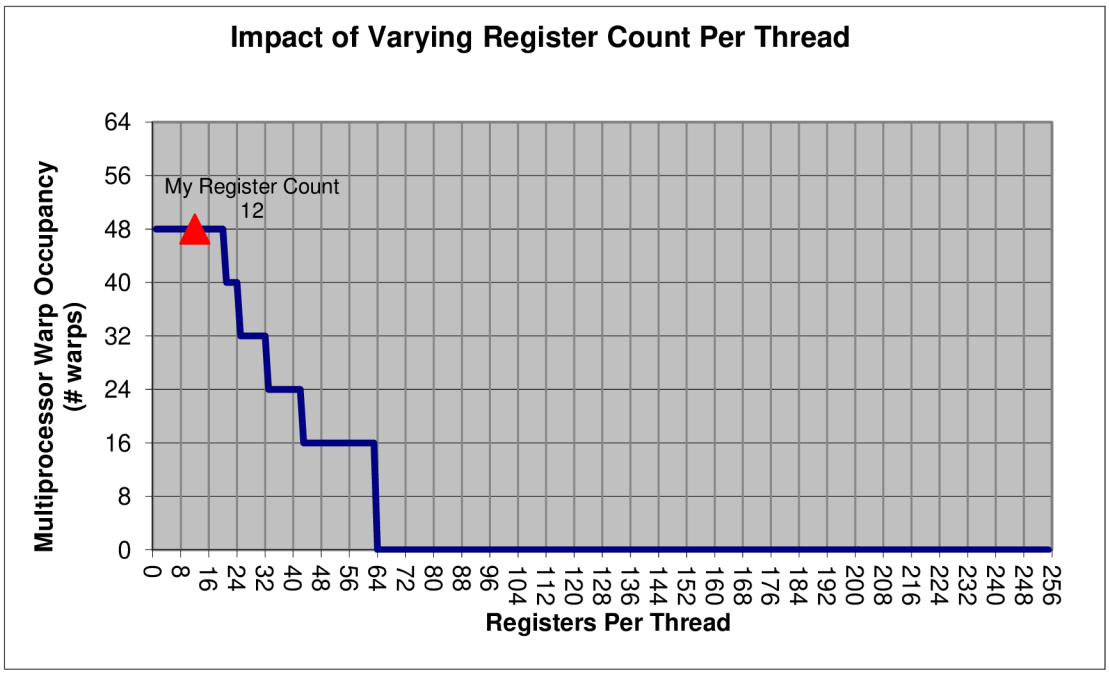
Krivka na obrázku 6.3 vyjadruje mieru naplnenia multiprocesorov v prípade, že by sme menili počet použitých registrov. Podobne ako u zdieľanej pamäti je pre tento kernel aj v prípade registrov ich spotreba minimálna. Nieje teda možné dosiahnuť znižovaním počtu použitých registrov lepšie naplnenie.



Obr. 6.7: Zaplnenie multiprocesorov pre rôzne veľkosti bloku



Obr. 6.8: Vplyv rôznych veľkostí použitej zdieľanej pamäti



Obr. 6.9: Vplyv rôzneho počtu použitých registrov

6.2 Výsledky merania

Ťažiskom tejto práce bolo navrhnúť paralelné varianty algoritmov za použitia platformy CUDA, ktoré umožnia rýchlejší výpočet nad rovnakými dátami. Tabuľka 6.1 ukazuje výsledky meraní, za účelom zrovnania CPU a GPU variantov implementácie. Merania boli vykonávané opakovane a výsledky sú získané spriemerovaním hodnôt viacerých meraní. CPU variant bol testovaný na procesore Intel Core i7-4790K 4.00GHz 4.00 GHz. GPU variant bol testovaný na grafickej karte Nvidia GeForce GTX 980. Ako bolo uvedené vyššie, rýchlosť výpočtu nie je závislá na vzťahu 6.1. Z toho dôvodu boli algoritmu predsunuté vstupné dáta o veľkosti 1000000 dátových položiek, aby bolo možné posudzovať efektivitu pri na veľkom množstve dát. Podobné platí aj pre nadstavenia veľkosti bloku a mriežky. Aplikácia bola navrhnutá tak, že pokiaľ je dosiahnuté maximálneho zaplnenia mutiprocesorov, tak spúšťanie väčšieho množstva vlákien alebo blokov výpočet neurýchli. Je to preto, lebo grafická karta vie naraz vykonávať len obmedzený počet vlákien a tento limit bol dosiahnutý.

Počas testovania bol odhalený problém v implementácii. Na niektorých grafických kartách pri niektorých konfiguráciách parametrov aplikácia zhavaruje. Z tohto dôvodu nebolo možné ju otestovať v takej miere ako bolo plánované. Podľa predpokladov spomínaných v predchádzajúcich kapitolách, ale vieme odvodiť výsledky aj z menšieho množstva nameraných dát nazbieraného z fungujúcich konfigurácií a kariet, takže nejde o fatálny problém. Tento problém nebol z časových dôvodov odstránený, keďže nebol pre testovanie kritický, čas sa venoval lepšiemu prepracovaniu kernelov.

Ako je vidieť, väčšina kernelov bola dobre navrhnutá a priniesla pozitívne výsledky. Druhý kernel (SpliTriangles.cu), ale priniesol značne negatívny výsledok. Problémov je v tomto prípade niekoľko. Prvým je nesprávny prístup do pamäti spôsobujúci zbytočne veľké množstvo transakcií. Druhým problémom sú tri vnorené cykly, ktoré znižujú efektivitu paralelnej implementácie. Tretí a najpodstatnejším problémom je divergencia v rámci vnorených cyklov. Nie je ale pravdepodobné, že odstránením týchto problémov sa dostaneme k efektívnej implementácii. Bolo by vhodné preskúmať možnosti iného spôsobu výpočtu, ktorý by bol následne paralelizovaný. V prípade prípade praktického využitia týchto algoritmov by samozrejme bolo možné aplikovať iba tie kernely, ktoré prinášajú najväčšie zrýchlenie.

	Kernel 1	Kernel 2	Kernel 3	Kernel 4
CPU	393.604492 ms	88.191711 ms	79.474022 ms	
GPU	5.388208 ms	2881.060303 ms	1.853156 ms	6.822299 ms
GPU bez réžie	0.392958 ms	-	1.145839 ms	1.297133 ms
Zrýchlenie	73.04x	0.03x	9.16x	

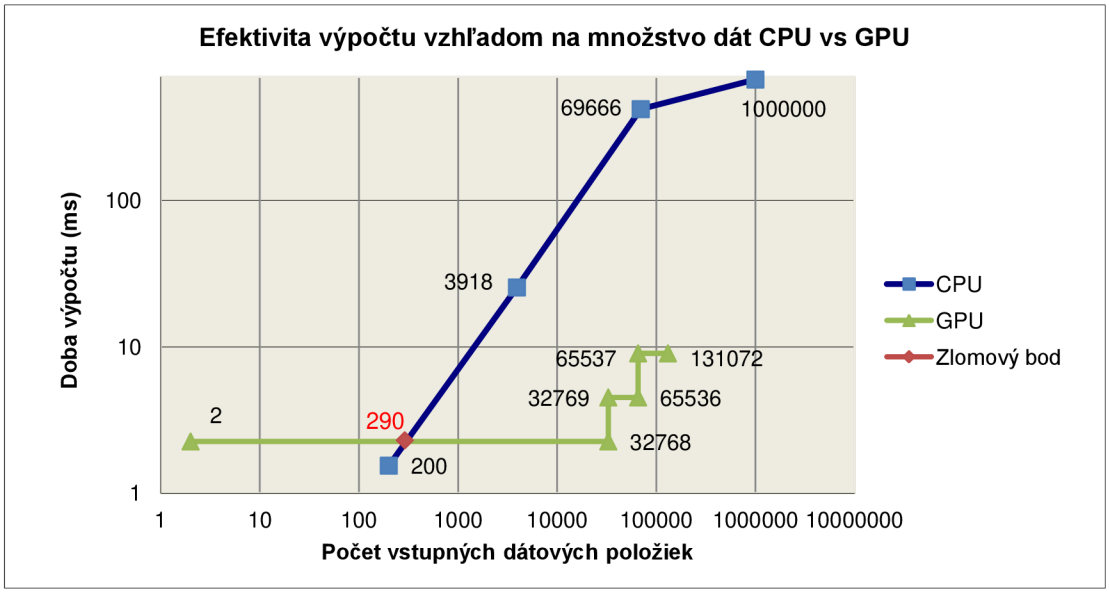
Tabuľka 6.1: Porovnanie výsledkov rýchlosti výpočtu CPU vs GPU.

Vytvoriť kernely zefektívňujúce výpočty pri konštrukcii k-d stromov sa teda vytvoriť podarilo, ale bolo by vhodné pre dosiahnutie maximálnej efektivity v tejto snahe pokračovať. Bolo by prínosné preskúmať obmedzenia zdrojov ako registre a zdieľaná pamäť a pokúsiť sa dosiahnuť minimálnych hodnôt. Podobne by stálo za to preskúmať

6.3 Prepínanie medzi CPU a GPU variantami

Ako bolo spomínané v predošlej sekcii, testovanie aplikácie prinieslo zväčša pozitívne výsledky. Je pravda, že tieto výsledky sú závislé na veľkosti vstupných dát podľa vzťahu 6.1. Z toho vyplýva, že existuje hodnota h počtu vstupných dátových položiek, od ktorej menšie množstvá vstupných dát je efektívnejšie počítať na CPU. Toto vyúsťuje do návrhu pre efektívnejšie využitie výpočtového výkonu a oboch variantov implementácie. Naším cieľom je čo najrýchlejšia implementácia. Toho dosiahneme tak, že pre všetky hodnoty nižšie ako h prepneme pri vyhodnocovaní z GPU na CPU implementáciu. Týmto využijeme najlepších výsledkov oboch variantov.

Nasledujúci graf ukazuje príklad porovnania rýchlostí oboch variantov nad rôznymi veľkosťami dát. Taktiež je na ňom vidieť zlomový bod h kde sa oplatí prepnúť medzi implementáciami.



Obr. 6.10: Porovnanie efektivity CPU a GPU

Kapitola 7

Záver

V rámci tejto práce bola podľa predlohy vytvorená referenčná CPU implementácia konštrukcie dátovej štruktúry k-d strom. Následne boli analyzované jej jednotlivé časti za účelom výberu vhodného základu pre paralelnú implementáciu. Bolo zvolených päť častí s perspektívou dobrej paralelnej implementácie. Týchto päť častí bolo analyzovaných z hľadiska predom stanovených paralelizačných prístupov na platforme CUDA. Následne boli zvolené štyri časti (spracovanie udalostí, delenie trojuholníkov, SAH a redukcia koncových hodnôt), ktoré boli implementované ako GPU varianty vo forme CUDA kernelov. Po dokončení implementácie sa prešlo na testovanie, ktoré ukázalo, že tri z kernelov spĺňali kritériá na metriku naplnenia multiprocesorov. Tieto tri kernely priniesli pozitívne výsledky z pohľadu rýchlosti pri porovnaní s referenčnou CPU implementáciou (Kernel 1 = 73.04x, Kernely 3 a 4 = 9.16x). Kernel 2 priniesol negatívne výsledky z dôvodu obmedzení algoritmu, z ktorého sa vychádzalo.

Táto práca síce priniesla pozitívne výsledky, ale to neznamená, že by zostávalo málo miesta na zlepšenie. Prvým krokom pre zlepšenie by bolo extrahovať kernely ako samostatné bloky a otestovať ich v separovanom prostredí. Zlepšenie by mohol priniesť aj prieskum kernelov na základe používaných zdrojov. Pokiaľ by sa zistilo, že je možné implementovať niektorý z kernelov tak, aby používal menšie množstvo zdrojov, mohlo by sa dosiahnuť ďalšieho zrýchlenia. V prípade kernelu, ktorý priniesol negatívne výsledky, by bolo vhodné preskúmať alternatívne algoritmy pre daný výpočet, aby sa našiel východiskový algoritmus s lepšími vlastnosťami pre paralelizáciu. V rámci testovania bola tiež prezentovaná myšlienka využitia prepínania CPU a GPU implementácie pre efektívnejší výpočet vzhľadom na veľkosť vstupu.

Literatúra

[1] Appel, A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), New York, NY, USA: ACM, 1968, s. 37–45.

[2] Bentley, J. L.: Multidimensional Binary Search Trees Used for Associative Searching. ročník 18, č. 9, 1975: s. 509–517, ISSN 0001-0782.

[3] Berg, M. d.; Cheong, O.; Kreveld, M. v.; aj.: *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 2008, ISBN 3540779736, 9783540779735, 95–95 s.

[4] Blum, M.; Floyd, R. W.; Pratt, V.; aj.: Time Bounds for Selection. *J. Comput. Syst. Sci.*, ročník 7, č. 4, Srpen 1973: s. 448–461, ISSN 0022-0000.

[5] Brown, R. A.: Building a Balanced k-d Tree in O(kn log n) Time. *CoRR*, ročník abs/1410.5420, 2014.

[6] Cormen, T. H.; Stein, C.; Rivest, R. L.; aj.: *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001, ISBN 0070131511.

[7] Dutre, P.; Bala, K.; Bekaert, P.; aj.: *Advanced Global Illumination*. AK Peters Ltd, 2006, ISBN 1568813074.

[8] Eberhardt, H.; Klumpp, V.; Hanebeck, U. D.: Density Trees for Efficient Nonlinear State Estimation. In *Proceedings of the 13th International Conference on Information Fusion (Fusion 2010)*, 2010.

[9] Fuchs, H.; Kedem, Z. M.; Naylor, B. F.: On Visible Surface Generation by a Priori Tree Structures. *SIGGRAPH Comput. Graph.*, ročník 14, č. 3, jul 1980: s. 124–133, ISSN 0097-8930.

[10] Fung, J.; Mann, S.: Computer vision signal processing on graphics processing units. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, ročník 5, May 2004, ISSN 1520-6149, s. V–93–6 vol.5.

[11] Fung, J.; Mann, S.: Using multiple graphics cards as a general purpose parallel computer: applications to computer vision. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, ročník 1, Aug 2004, ISSN 1051-4651, s. 805–808 Vol.1.

[12] Fung, J.; Tang, F.; Mann, S.: Mediated reality using computer graphics hardware for computer vision. In *Wearable Computers, 2002. (ISWC 2002). Proceedings. Sixth International Symposium on*, 2002, ISSN 1530-0811, s. 83–89.

[13] Gunther, J.; Popov, S.; Seidel, H.-P.; aj.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, Sept 2007, s. 113–118.

[14] Havran, V.; Bittner, J.: On Improving KD-Trees for Ray Shooting. *Journal of WSCG*, ročník 10, č. 1, 2002: s. 209–216.

[15] Laboratory, R. P. I. I. P.; Meagher, D.: *Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer.* 1980.
URL https://books.google.cz/books?id=CgRPOAAACAAJ

[16] Micikevicius, Paulius: Advanced CUDA C. GPU technology conference.
URL http://www.nvidia.com/content/GTC/documents/1029_GTC09.pdf

[17] Mittal, S.; Vetter, J. S.: A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, ročník 47, č. 4, Jul 2015: s. 69:1–69:35, ISSN 0360-0300.

[18] Nikodym, Tomas: Ray Tracing Algorithm For Interactive Applications. Bachelor's thesis.
URL https://dip.felk.cvut.cz/browse/pdfcache/nikodtom_2010bach.pdf

[19] Schroeder, Tim: CUDA OPTIMIZATIONS. ISC 2011 Tutorial.
URL http://www.nvidia.com/content/pdf/isc-2011/schroder.pdf

[20] Schumacher, R.; Division, A. F. H. R. L. T. R.: *Study for Applying Computer-generated Images to Visual Simulation.* AFHRL-TR, Air Force Human Resources Laboratory, Air Force Systems Command, 1969.

[21] Suffern, K.: *Ray Tracing from the Ground Up.* CRC Press, 2016, ISBN 9781498774703.
URL https://books.google.cz/books?id=_RDYCwAAQBAJ

[22] Wald, I.; Havran, V.: On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Interactive Ray Tracing 2006, IEEE Symposium on*, Sept 2006, s. 61–69.

[23] Whitted, T.: An Improved Illumination Model for Shaded Display. *Commun. ACM*, ročník 23, jun: s. 343–349, ISSN 0001-0782.

[24] WWW stránky: CUDA C Best Practices Guide [online]. [cit. 2016-01-01].
URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[25] WWW stránky: CUDA C Programming Guide [online]. [cit. 2016-01-01].
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[26] WWW stránky: CUDA parallel computing [online]. [cit. 2016-01-01].
URL http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html

[27] WWW stránky: NVIDIA CUDA Toolkit [online]. [cit. 2016-01-01].
URL https://developer.nvidia.com/cuda-toolkit