VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Profilování paralelních aplikací

Profiling of Parallel Applications

2018 Bc. Jakub Beránek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Beránek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Profilování paralelních aplikací
Profiling of Parallel Applications

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementace nástroje pro profilování paralelních aplikací. Cílovou architekturou bude NVIDIA CUDA. Nad rámec možností již existujících nástrojů by se nově navržená aplikace měla zaměřit spíše na detekci a vyhodnocování časových a prostorových přístupů skupin vláken do paměti s důrazem na následnou vizualizaci výsledků.

1. Seznámení se s problematikou profilování paralelních aplikací, přehled existujících nástrojů a jejich možností.
2. Návrh a implementace nástroje pro profilování paralelní aplikace.
3. Implementace jednoduchých ukázek ve vybrané technologii s ohledem na slabá místa při práci s globální a sdílenou pamětí.
4. Provedení testů navržených ukázek za pomoci nově vytvořeného nástroje, vyhodnocení testů.
5. Shrnutí dosažených výsledků.

Seznam doporučené odborné literatury:

[1] Edward Angel: Interactive Computer Graphics, ISBN-10:032153586 (2009)
[2] Jason Standers, Edward Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming, ISBN-10: 0131387685, 2010

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2017
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Chtěl bych poděkovat svému vedoucímu, Ing. Petru Gajdošovi, Ph.D., za předání odborných znalostí o platformě CUDA a pomoc při tvorbě této práce.

Abstrakt

Tato práce se zabývá vizualizací paměťových přístupů programů na platformě CUDA. Je v ní prezentován profilovací nástroj, který instrumentuje CUDA programy a generuje z nich záznam paměťových přístupů. Ty jsou poté prezentovány uživateli pro odhalení možných paměťových optimalizací. První část práce popisuje metody profilování programů a technologie použité pro tvorbu profilovacího nástroje. Poté následuje popis existujících nástrojů pro profilování aplikací pro grafické karty. Třetí část se zabývá návrhem a implementací profilovacího nástroje a webové aplikace pro vizualizaci zaznamenaných přístupů.

Klíčová slova: vizualizace paměťových přístupů, profilování, instrumentace, paralelní aplikace, CUDA, LLVM

Abstract

This thesis deals with visualising memory accesses of CUDA programs. It presents a profiling tool that instruments CUDA programs and generates their memory access traces. These can then be visualised to exploit memory access optimization opportunities. First part of the thesis introduces application profiling methods and technologies used to create the profiling tool. The second part describes existing tools that allow profiling graphics processing units. The last part discusses design and implementation of the profiling tool and introduces a web application that visualises the stored memory traces.

Key Words: memory access visualization, profiling, instrumentation, parallel applications, CUDA, LLVM

Obsah

Seznam použitých zkratek a symbolů

CAS	–	Compare And Swap
CPU	–	Central Processing Unit
CSS	–	Cascading Style Sheets
DMA	–	Direct Memory Access
DOM	–	Document Object Model
GPU	–	Graphics Processing Unit
GPGPU	–	General-Purpose computing on Graphics Processing Units
HTML	–	Hypertext Markup Language
IDE	–	Integrated Development Environment
IR	–	Intermediate Representation
JIT	–	Just-In-Time
JSON	–	JavaScript Object Notation
NUMA	–	Non-Uniform Memory Access
ODR	–	One Definition Rule
OOP	–	Object-Oriented Programming
PCI	–	Peripheral Component Interconnect
PTX	–	Parallel Thread Execution
RTTI	–	Run-Time Type Information
SSA	–	Static Single Assignment
SIMT	–	Single Instruction, Multiple Threads
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Už desítky let platí, že frekvence počítačových procesorů je mnohonásobně větší než frekvence jejich operační paměti. Tato nerovnováha může způsobit výrazné zpomalení aplikace, pokud procesor neustále jen čeká, než k němu dorazí požadovaná data z paměti. Procesory z tohoto důvodu už dlouhou dobu obsahují malé a rychlé cache paměti. Ty ale samy o sobě nestačí. Aby byl jejich potenciál plně využit, musí být program navržen pro efektivní využití cache. Lze například ovlivnit zarovnání a velikost použitých datových struktur s ohledem na velikost řádek cache nebo používat speciální instrukce procesoru pro práci s cache pamětí (například přednačítání nebo netemporální zápisy). Obecně řečeno je nutno zajistit, aby přístupy do paměti měly dobrou časovou a prostorovou lokalitu - to je jednou z nejdůležitějších podmínek nutných pro maximální výkon výpočtu. Aby však mohl programátor s pamětí takto pracovat, potřebuje k tomu vhodné nástroje.

Pro efektivní využití paměti je nutno použít programovací jazyk, který s ní dovoluje manipulovat přímo, což často není možné v moderních jazycích vysoké úrovně. Umožňují to pouze jazyky, které mají blízko k hardwaru počítače, jako je například *C* a *C++*. Pomocí nich lze velmi detailně ovládat, jakým způsobem bude procesor využívat operační paměť, což je také jeden z důvodů, proč jsou často používány pro výkonnostní výpočty.

Kromě vhodného programovacího jazyka je pro optimalizaci paměťových přístupů nutná i zpětná vazba o tom, co se za běhu programu s pamětí děje. Existuje mnoho profilovacích nástrojů, které měří dobu výpočtu strávenou v jednotlivých funkcích [1], analyzují alokovanou paměť procesu [2] nebo dokonce simulují cache procesoru [3]. Pokročilejší nástroje dokáží k jednotlivým řádkům kódu dodat informaci o tom, jestli byl přístup do paměti efektivní nebo ne (tj. jestli přistupoval do hlavní paměti nebo do některé z cache pamětí) [4].

Tyto nástroje umožňují naměřit výkon aplikace a po provedené změně v kódu tak pomáhají určit, jestli změna ovlivnila výkon a jestli vedla ke zkrácení nebo prodloužení doby výpočtu. Naměřené informace o výkonu však neposkytují návod k tomu, jakou změnu v aplikaci udělat, aby běžela rychleji. Při komplexitě dnešních překladačů (popř. běhových prostředí) může být velmi složité poznat, jak změna v kódu ovlivní práci s pamětí a samotný výkon aplikace. To lze zjistit zaznamenáním informací z reálného běhu programu - po odstranění abstrakcí kódu a aplikování optimalizací překladačem. Tyto údaje poté mohou sloužit k pochopení toho, jak aplikace s pamětí pracuje a kde jsou případná úzká hrdla.

Jednou z oblastí, kde je kladen velký důraz na efektivní práci s pamětí a maximální výkon, jsou výpočty prováděné na grafických kartách. Ty byly sice původně určené výhradně pro grafické aplikace, nicméně později vznikla programovatelná rozhraní, která na grafických kartách umožnila provádět libovolné výpočty. Díky tomu se začaly hojně používat pro výkonnostní výpočty, například pro fyzikální simulace nebo strojové učení. Grafické karty mají jinou architekturu než klasické procesory. Skládají se až z tisícovek jader, která jsou sice méně výkonná než jádra procesoru, ale díky svému počtu zvládnou paralelně zpracovat obrovské množství dat.

O to více je u těchto karet důležitý efektivní přístup do paměti, o kterou soupeří spousta vláken zároveň. Jednotlivé generace architektur grafických karet také mohou vyžadovat specifické vzory přístupu do paměti pro dosažení maximální efektivity.

Tato práce si klade za cíl vytvořit nástroj, který umožní zaznamenat paměťové přístupy konkrétního běhu aplikace. Nástroj by měl být zaměřen na paralelní aplikace pro grafické karty, u kterých je efektivní práce s pamětí velmi důležitá. Zároveň pro ně ale existuje pouze zlomek profilovacích nástrojů ve srovnání s nástroji pro klasické procesory. Rozhraní pro grafické karty totiž často nejsou otevřená ani zpětně kompatibilní a existující profilovací nástroje je tak velmi obtížné použít s jejich aktuální verzí. Nástroj vytvářený v této práci by měl využít platformy LLVM tak, aby bylo snadné ho upravit pro nové verze těchto grafických rozhraní. Dále by měl být schopný zobrazit naměřená data v intuitivní podobě programátorovi. Tato vizualizace by měla umožnit pochopit kam, kdy a jak aplikace přistupuje do paměti a použít tato data k zefektivnění těchto přístupů. Nástroj by tak byl užitečný pro poskytnutí přehledu o paměťových operacích na grafické kartě.

Implementace profilovacího nástroje je zaměřena na platformu Linux, jazyk C/C++ a architekturu grafických karet CUDA, popsané principy jsou však aplikovatelné univerzálně. V této práci jsou používány výrazy CPU (Central Processing Unit) a GPU (Graphics Processing Unit). Pokud není uvedeno jinak, tak CPU označuje procesor počítače, resp. kód, který na něm běží, obdobně GPU označuje grafický akcelerátor, resp. kód, který na ní běží.

Kapitola 2 poskytuje přehled o metodách používaných k profilování aplikací a technologiích použitých v této práci. Dále v kapitole 3 následuje popis existujících nástrojů pro profilování paralelních aplikací. Kapitola 4 popisuje návrh a implementaci profilovacího nástroje a následující kapitola 5 vývoj vizualizační aplikace. Poté následuje zhodnocení dosažených výsledků a popis možných rozšíření nástroje.

2 Technologie

Tato kapitola popisuje technologie, knihovny a principy použité při tvorbě profilovacího nástroje. Nejprve je popsán překladač LLVM, poté platforma pro výpočty na grafických kartách CUDA a následně metody používané pro profilování aplikací.

2.1 LLVM

LLVM je sada knihoven, které spolu tvoří infrastrukturu pro tvorbu překladačů a programovacích jazyků [5]. Původně bylo LLVM vyvinuto jako open-source, rozšiřitelný a modulární překladač pro jazyky *C* a *C++*, který měl tvořil alternativu k monolitickému překladači GCC. Později se z něj stala univerzální platforma pro optimalizaci, analýzu, instrumentaci a generování programů. Poskytuje funkcionalitu od zpracování zdrojového kódu až po vygenerování výsledných spustitelných souborů. LLVM se skládá ze sady frontendů, backendů a virtuální instrukční sady nazvané LLVM IR.

Frontend

Frontend je nástroj pro zpracování programu ve formě programovacího jazyka. Je zodpovědný za kontrolu (a někdy i automatickou opravu) syntaktických a sémantických chyb v kódu. Je naimplementován buď jako služba, kterou lze využít v integrovaných programovacích prostředích anebo přímo jako součást překladače. Frontend nejprve provádí lexikální analýzu kódu a následně z něj vybuduje abstraktní syntaktický strom, nad kterým lze provádět analýzy. Poté z tohoto stromu vygeneruje instrukce ve virtuální instrukční sadě LLVM IR. Každá z těchto fází překladu je poskytována ve formě knihovny, což usnadňuje tvorbu frontendu pro nový programovací jazyk.

Backend

Backend přijímá na vstupu LLVM IR instrukce, obvykle převzaté z frontendu. Jeho úkolem je vygenerovat z této virtuální instrukční sady spustitelný program určený pro konkrétní procesor, resp. instrukční sadu. Virtuální instrukce v backendu procházejí sadou transformací. Nejprve dochází k výběru instrukcí, kdy jsou jednotlivé IR instrukce nebo sady instrukcí transformovány do odpovídajících instrukcí pro cílovou architekturu. Následně je zvoleno co nejoptimálnější pořadí instrukcí pro maximální využití superskalárních architektur a minimalizaci výpadků v pipeline procesoru. Poté backend aplikuje běžné optimalizace (například peephole optimization [6] nebo strength reduction [7]) a naalokuje registry pro instrukce. Ve finální části překladu dochází ke konverzi instrukcí buď do jazyka symbolických adres nebo přímo do strojových instrukcí pro zvolený procesor. Kromě přímého překladu do strojových instrukcí lze LLVM backend použít také pro dynamický překlad instrukcí za běhu programu (JIT překlad). Toho lze využít třeba pro optimalizaci kódu, jehož optimální vlastnosti nejsou v době překladu známé, nebo

pro emulaci chybějících hardwarových funkcí grafických akcelerátorů. Stejně jako u frontendu poskytuje LLVM sadu rozhraní pro zjednodušení tvorby backendu pro novou instrukční sadu.

LLVM IR

LLVM IR je instrukční sada pro virtuální procesor, takže není závislá na konkrétní hardwarové architektuře. Její instrukce jsou vždy v tzv. SSA formě [8, 9], u které je zaručeno, že do každé proměnné je zapsána hodnota právě jednou. Tato forma je běžně používána pro reprezentaci imperativních programovacích jazyků, protože usnadňuje analýzu programu a aplikování optimalizací[1]. LLVM poskytuje rozhraní pro optimalizaci, analýzu a modifikaci IR instrukcí ve formě tzv. průchodů. Průchod je funkce, která na vstupu přijímá IR modul. Ten může libovolně upravit a poté jej vrátí k dalšímu zpracování. IR modul obsahuje deklarace a definice funkcí, globálních proměnných a datových typů, ladící údaje a údaje o architektuře, pro kterou je modul určen. Při překladu jazyka C modul přibližně odpovídá jednotce překladu, obvykle tedy jednomu souboru s implementací, který je překládán do formy objektového souboru. LLVM při překladu IR moduly postupně předává průchodům, které si lze navolit. LLVM obsahuje velké množství vestavěných průchodů, které jsou vždy zaměřené na jednu konkrétní věc z oblasti analýzy (počítání instrukcí, sestavení grafu volání funkcí, analýza překrytí ukazatelů) nebo transformace (eliminace nedosažitelného kódu, propagace konstant, rozbalování cyklů) kódu. Při použití nejagresivnější optimalizace (-O3) pro překlad C/C++ programů se v LLVM používá několik desítek těchto průchodů pro optimalizaci kódu. Lze také jednoduše vytvořit vlastní průchody, čehož bylo využito pro tvorbu profilovacího nástroje v této práci.

Každý typ IR instrukce je v kódu reprezentován třídou. Existují tak třídy pro instrukci načtení hodnoty z paměti, sečtení dvou čísel nebo paměťové bariéry. Vzhledem k tomu, kolik různých architektur a programovacích jazyků LLVM podporuje, jsou jednotlivé třídy značně obsáhlé a obecné. Třídy instrukcí tvoří hierarchii s jedním společným předkem, třídou `Value`. Kromě instrukcí z této třídy dědí i hodnoty nacházející se v kódu, například čísla nebo ukazatele. Všechny prvky vyskytující se v IR tak lze reprezentovat tímto abstraktním rozhraním. To umožňuje zpracovávat jednotlivé instrukce a hodnoty polymorfně, což velmi usnadňuje psaní nástrojů a průchodů, které s IR pracují. Pro některé účely je však nutné zjistit konkrétní typ daného objektu. Kvůli toho LLVM obsahuje vlastní dynamický typový systém, který je alternativou k standardnímu systému C++ RTTI a umožňuje za běhu aplikace zjišťovat typ objektů.

Instrukce jsou uspořádány do tzv. základních bloků (basic block), což je sekvence instrukcí bez větvení, s právě jedním vstupním a výstupním bodem. Toto logické uspořádání usnadňuje analýzu instrukcí, protože v rámci jednoho bloku je zaručeno, že nedojde ke skoku mimo daný blok. Funkce jsou poté tvořeny skupinami těchto základních bloků. LLVM nabízí rozhraní pro procházení i modifikaci všech výskytů dané instrukce nebo hodnoty v rámci bloku či funkce, což usnadňuje modifikaci a instrumentaci kódu.

[1] Pro přechodnou reprezentaci funkcionálních jazyků je obvykle používán tzv. continuation-passing style [10]

Komponenty LLVM jsou modulární a je snadné je použít jako knihovnu v jiném programu. Pro vytvoření nového programovacího jazyka tak stačí napsat frontend pro zpracování syntaxe a veškeré optimalizace a generování kódu pro různé architektury poskytne LLVM bez nutnosti jakékoliv další práce. Stejně tak pokud vznikne nová hardwarová architektura, stačí vytvořit nový backend a poté pro ni lze psát aplikace v libovolném programovacím jazyce, který LLVM podporuje. Toto mimo jiné velmi urychluje experimenty s vývojem nových programovacích jazyků, umožňuje vytvářet nástroje pro automatickou analýzu zdrojového i strojového kódu a jednoduše testovat nové optimalizace. LLVM podporuje velké množství programovacích jazyků (například C, C++, Rust, Swift, D, C#, Haskell, Java, Kotlin, Python) i architektur a backendů (například ARM, MIPS, PTX, PowerPC, SPARC, x86, x64, WebAssembly). Nejvíce vyvíjeným frontendem je state-of-the-art překladač jazyků C a C++ *Clang*, který je svou funkcionalitou srovnatelný s běžně používaným překladačem *GCC*. Mimo jiné podporuje i překlad programů napsaných pro platformu CUDA.

2.2 CUDA

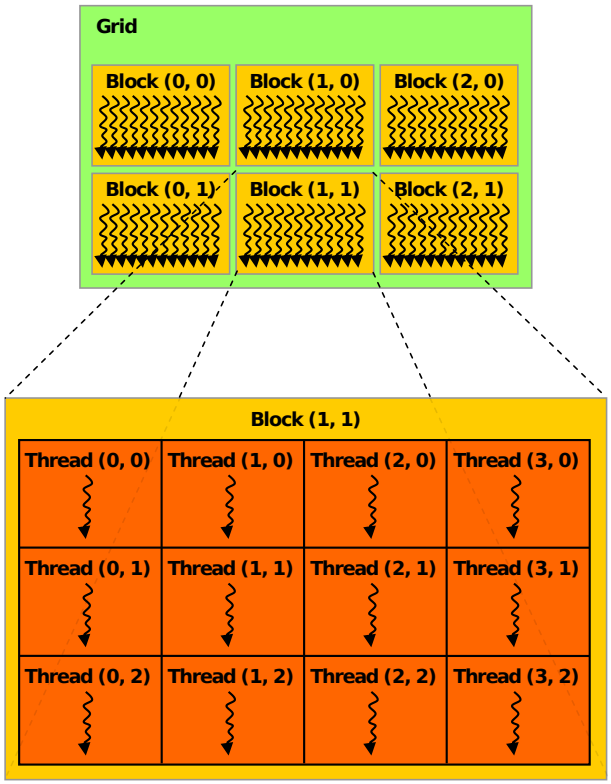
Grafické karty byly původně určené výhradně pro hardwarovou akceleraci 3D scén (hlavně ve videohrách a modelovacích nástrojích). Díky své architektuře využívající velké množství výpočetních jader však začaly být brzy využívány také pro obecné výpočetní aplikace (například fyzikální, chemické, biologické či mechanické simulace), kde jejich použití mohlo významně zlepšit výkon programů. Grafické karty ovšem uměly řešit pouze problémy v doméně počítačové grafiky, jiné výpočty tedy musely být převedeny do jazyka trojúhelníků a textur. Tento způsob programování byl náročný a neumožňoval naplno využít potenciál grafických akcelerátorů. Z tohoto důvodu přišla v roce 2007 společnost Nvidia s platformou *CUDA* [11], která umožňuje efektivně provádět obecné výpočty na grafických kartách této společnosti. CUDA se skládá z hardwarové architektury GPU, rozšíření jazyka C, které je použito k psaní kódu pro GPU a také ze sady knihoven mj. pro lineární algebru, zpracování signálů a strojové učení, které jsou ručně optimalizovány pro efektivní využití dané architektury. CUDA byla jedním z prvních rozhraní pro tzv. GPGPU[2], díky rozšíření známého jazyka C je velmi jednoduché ji použít a Nvidia jí poskytuje stálou podporu. V roce 2009 vznikl otevřený standard OpenCL, který je narozdíl od platformy CUDA kompatibilní i s klasickými procesory a grafickými kartami od jiných společností, než je Nvidia. Jeho vývoj je však pomalejší kvůli nutnosti udržovat kompatibilitu mezi všemi podporovanými platformami.

CUDA akcelerátory mohou spouštět až tisíce vláken, která provádí výpočty paralelně. Jednotlivá vlákna jsou organizována do trojrozměrných skupin (tzv. bloků). Ty jsou dále seskupeny do trojrozměrné mřížky [12]. Každé vlákno tak tvoří pomyslný bod v šestirozměrném prostoru. Ukázku výpočetní mřížky si lze prohlédnout na obrázku 1. Výpočet na grafické kartě je spouštěn pomocí funkcí, které se označují jako kernely. Při spuštění kernelu je nutné určit, jaké rozměry

[2]Obecné výpočty na grafických akcelerátorech

bude mít výpočetní mřížka a bloky vláken. Toto rozdělení je užitečné při mapování výpočtu na dvourozměrné a trojrozměrné úlohy, není ale nutné jej využívat. Vlákna mají při běhu k dispozici informaci o jejich pozici ve výpočetní mřížce, což lze použít například k indexování do pole, textury, matice nebo jiné vícerozměrné datové struktury. Instrukce programu jsou vždy vykonávány v tzv. warpech, což jsou skupiny 32 jader, které jsou vždy součástí jednoho bloku. Celý warp vždy vykonává stejnou instrukci nad různými daty, což lze chápat jako obdobu vektorových instrukcí na CPU. Tento způsob vykonávání kódu je označován jako SIMT[3]. Pokud GPU kód obsahuje podmínky, které rozdělí programový čítač warpu v závislosti na datech, tak dojde k tzv. divergenci warpu. Obě dvě větve podmínky jsou poté postupně vykonány celým warpem. Vlákna provádějící větev, která pro ně není splněna, jsou neaktivní. Efekt instrukcí provedených těmito vlákny je anulován pomocí vymaskování registrů. Takto desynchronizovaný warp samozřejmě není tak efektivní, jako když se provádí stejná instrukce všemi vlákny, je tedy žádoucí divergenci v kódu minimalizovat. Warpy jsou prováděny na multiprocesorech (streaming multiprocessor) grafické karty. Ty obsahují plánovače warpů, které plánují vykonávání warpů podle vytížení procesoru a paměťových závislostí. Nejnovější generace CUDA karet Volta umožňuje plánovat provádění instrukcí na úrovni jednotlivých vláken[4], ne celých warpů, což zmírňuje problém divergence warpů.

[3]Single instruction, multiple threads
[4]ITS - independent thread scheduling



Obrázek 1: Výpočetní mřížka CUDA vláken

Zdroj: Nvidia Corporation [12]

Grafické akcelerátory mají svou vlastní operační paměť, obvykle je tedy nutné zkopírovat data z CPU na GPU před výpočtem a poté zase zkopírovat výsledek zpátky v opačném směru. Tento proces často tvoří úzké hrdlo aplikací, novější generace karet proto umožňují překrývat výpočet s kopírováním dat nebo přímo přistupovat k paměti procesoru z grafické karty. V paměti procesoru jsou data namapována do virtuálního adresního prostoru po jednotlivých stránkách. Při přesunu dat na GPU přes sběrnici PCI by tak mohlo dojít ke stránkovací chybě (page fault), například pokud je daná stránka odložena ve swapu na disku nebo pokud k ní zatím nebylo přistoupeno. Proto dochází před samotným kopírováním přesun dat do tzv. page-locked (pinned) paměti, ve které k tomuto dojít nemůže a grafická karta tak může použít mechanismus přímého přístupu k paměti (DMA) a zkopírovat data bez asistence procesoru. Novější verze platformy CUDA se snaží minimalizovat rozdíl mezi prací s CPU a GPU pamětí a usnadnit tak programátorům práci. To umožňuje mechanismus Unified Virtual Addressing, který u 64-bitových aplikací sjednocuje paměť CPU a GPU do jednoho společného virtuálního adresního prostoru. Díky tomu lze přistupovat k paměti procesoru přímo z grafické karty bez nutnosti speciální alokace dat (tyto přístupy ale samozřejmě mají kvůli nutnosti přenosu dat přes PCI sběrnici velkou odezvu). Tento koncept byl dále rozšířen pomocí Unified Memory, která automaticky migruje stránky mezi procesorem a grafickou kartou a zefektivňuje tak práci se sjednoceným adresním prostorem. Stejnou

datovou strukturu tak lze používat na CPU i GPU a navíc při přístupu z grafické karty budou data přesunuty do její paměti, což značně urychlí odezvu následujících přístupů.

CUDA zařízení mohou přistupovat k 1, 2, 4 nebo 8 bytům paměti v rámci jedné instrukce. Adresy přístupů musí být zarovnané pro danou velikost přístupu, protože CUDA nepodporuje nezarovnané paměťové přístupy. Efektivita přístupů se odvíjí od toho, k jakému typu paměti je přistupováno a jaký vzor přístupu je použit. Níže následuje přehled jednotlivých typů paměti (resp. pohledů na paměť) CUDA akcelerátorů.

Globální paměť je hlavní paměť grafické karty, která má řádově jednotky GiB a přístup do ní je relativně pomalý. Globální přístupy z jednotlivých vláken v rámci warpu jsou seskupeny pomocí transakcí, kdy je načteno 32, 64 nebo 128 bytů (podle velikosti řádky cache). Je výhodné načítat paměť z po sobě jdoucích adres, aby těchto transakcí vznikalo co nejméně. Zároveň je vhodné, aby byly přístupy zarovnané na úroveň řádků cache (obvykle 128 bytů pro L1 a 32 bytů pro L2 cache). I kdyby totiž vlákna přistupovala k paměti sekvenčně, pokud bude rozsah paměti zasahovat do dvou řádků cache, budou vytvořeny dvě paměťové transakce. Na novějších CUDA architekturách jsou hodnoty načítané z globální paměti udržovány pouze v L2 cache [12], u starších verzí se ukládaly i do L1 cache.

Lokální paměť je používána pro velké lokální proměnné a pole, pro které už nestačí velikost registrů. Data lokální paměti jsou umístěna v globální paměti (používá se pro ně ale rozdílná cache strategie), můžou tedy mít také značnou odezvu a malou průchodnost.

Sdílená paměť je rychlá a malá paměť umístěná blízko k výpočetním jádrům na čipu multiprocesoru. Tato paměť je sdílená v rámci jednoho výpočetního bloku. Její velikost se pohybuje v desítkách KiB pro každý blok. Paměťové přístupy k sdílené paměti jsou prováděny pomocí paměťových modulů (memory bank), které jsou rozděleny po 4 nebo 8 bytech a zpravidla je jich stejně jako vláken ve warpu, tedy 32. Modul, který bude použit pro paměťový přístup, je určen podle adresy, ke které se přistupuje. Pokud při paměťovém přístupu každé vlákno použije jiný modul, proběhnou všechny přístupy paralelně. Pokud ale dojde k současnému přístupu více vláken pomocí stejného modulu, dojde ke konfliktu (tzv. bank conflict), což způsobí serializaci přístupů a tedy i zpomalení paměťové operace.

Texturovací paměť je speciální cache nad globální pamětí, která je optimalizována pro 2D prostorové přístupy, čehož bývá často využíváno právě u práci s grafickými texturami. Tato paměť je pouze pro čtení.

Konstantní paměť je také cache nad globální pamětí, která je optimalizována pro čtení malého počtu konstantních dat.

Registry slouží ke stejnému účelu jako u procesoru, tj. k uchování malého množství pracovních dat, které jsou často využívány a je k nim poskytován velmi rychlý přístup. Jednotlivé bloky na CUDA zařízeních zpravidla obsahují tisíce registrů, které jsou sdíleny mezi aktivními vlákny.

Pro efektivní přístupy do paměti na grafické kartě je důležité, aby byly přístupy zarovnané a ideálně blízko v paměti u sebe. Pokud dvě vlákna načítají data z dvou po sobě jdoucích adres, tak může dojít ke sloučení těchto přístupů do jedné efektivní paměťové transakce. Pokud dvě vlákna načítají data z dvou vzdálených míst ve (virtuální) paměti, tak musí transakce vzniknout dvě. Z tohoto důvodu se může vyplatit používat pro kernely jinou reprezentaci dat než pro CPU. Problém efektivity přístupu do paměti může nastat například u polí objektů (array of structures), které jsou běžné v objektově orientovaných jazycích. Kernely často prochází všechny objekty v poli, ale modifikují nebo čtou pouze několik vybraných atributů. V této situaci dochází k tomu, že i když se čte z paměti sekvenčně, tak jednotlivé hodnoty jsou v paměti relativně daleko od sebe, protože mezi nimi leží všechny ostatní atributy daného objektu. Tento problém nenastává pouze u grafických akcelerátorů, ale také u CPU, protože při přednačítání (prefetch) do cache se zbytečně načítají atributy, které nejsou využity. Pro vyřešení tohoto problému lze pole struktur převést na tzv. strukturu polí (structure of arrays). Při tomto převodu je pro každý atribut objektu vytvořeno samostatné pole, které umožňuje přistupovat k danému atributu sekvenčně v paměti. Pokud tak bude kernel číst pouze jeden atribut objektu, bude přístup k němu efektivní. Obdobný princip je například řádkové versus sloupcové uložení matic, vektorů či tabulek databáze. Detaily toho, jak vlákna do paměti přistupují, nemusí být zřejmé ze zdrojového kódu programu. Právě zobrazení těchto přístupů do paměti je tak jednou z motivací pro tvorbu nástroje vyvíjeného v této práci.

CUDA silně využívá paralelismu, obsahuje tedy i mechanismy pro synchronizaci. Jednou z nejpoužívanějších konstrukcí pro synchronizaci je instrukce *syncthreads*, která slouží jako bariéra pro všechny warpy v rámci bloku. Pokud vlákno provede tuto instrukci, tak se zablokuje, dokud ji neprovedou i všechna ostatní vlákna v bloku. Je tak nutné zajistit, aby tato instrukce byla provedena opravdu všemi vlákny bloku, jinak může dojít k uzamčení (deadlock). Velmi užitečným synchronizačním mechanismem jsou také atomické instrukce. Ty fungují obdobně jako atomické instrukce procesoru - umožňují načíst, modifikovat a zpětně zapsat několik bytů v jedné atomické transakci, takže nemůže dojít k přečtení nekonzistentních dat. Mezi podporované atomické operace patří např. inkrementace, sčítání, výběr maxima, výměna hodnoty a CAS[5]. Původně byly tyto instrukce podporovány pouze pro 32bitová celá čísla, v novějších verzích platformy CUDA už je ale lze provádět i na 64bitových číslech s plovoucí řádovou čárkou a to dokonce i v globální paměti.

CUDA programy jsou psány v syntaktickém rozšíření jazyka C++, CUDA C. To poskytuje speciální syntaxi pro spouštění kernelů, pomocí které lze navolit, na kolika vláknech kernel poběží a jakou strukturu bude mít výpočetní mřížka. Příklad spuštění kernelu si lze prohlédnout ve výpisu 1.

Kernely musí být v kódu označeny pomocí atributu `global`. Lze v nich používat pouze funkce přeložené pro grafickou kartu, které jsou označeny atributem `device` a v závislosti na použitém

[5]Compare and swap je instrukce, která porovná hodnotu v paměti s referenční hodnotou a pokud se rovnají, tak do paměti atomicky zapíše zadanou hodnotu. Často se používá pro implementaci ostatních atomických instrukcí.

překladači nemohou využívat některé vlastnosti C++ (např. výjimky nebo funkcionalitu z nejnovějších C++ standardů). CUDA programy lze přeložit buď proprietárním překladačem *nvcc* od Nvidie nebo překladačem Clang. Překladač nvcc sice stejně jako Clang vnitřně používá LLVM IR pro reprezentaci kódu, nicméně zdrojový kód tohoto překladače není zveřejněn a jeho vnitřní funkcionalita není zdokumentována. Proto je obtížné jej využít k automatizované modifikaci CUDA programů. Clang naproti tomu umožňuje využít celou LLVM infrastrukturu pro analýzu a modifikaci CUDA kódu a dovoluje používat nejnovější C++ standardy při psaní GPU kódu, nicméně nepodporuje zatím všechny CUDA funkce a optimalizace dostupné v nvcc.

```
// spuštění kernelu s 16 bloky, v každém bloku bude 32 vláken
addKernel<<<16, 32>>>(param1, param2);
```

Výpis 1: Spuštění kernelu

Při překladu CUDA programů je překládán zvlášť kód pro CPU a GPU. Kód pro procesor je přeložen klasickým způsobem do formy objektových souborů. GPU kód je přeložen do PTX, zpětně i dopředně kompatibilní instrukční sady pro CUDA karty. V kernelech je možné přímo používat PTX instrukce, což je ekvivalentní použití jazyka symbolických instrukcí (assembly) pro CPU v C kódu. Při spuštění aplikace driver grafické karty přeloží PTX kód pro architekturu použité grafické karty. Tento JIT překlad samozřejmě prodlužuje dobu spuštění aplikace, proto lze také přímo při překladu programu kromě PTX vygenerovat i kód pro zvolené architektury karet a přiložit ho ke spustitelnému souboru. Tím vzniká tzv. fatbinary - spustitelný soubor obsahující kód pro více architektur nebo instrukčních sad. PTX je uložen ve formě dat v objektovém souboru, který se poté klasickým způsobem přilinkuje ke zbytku aplikace. Od verze 5 CUDA podporuje tzv. separátní překlad GPU kódu, kdy se každý soubor s kódem pro grafickou kartu přeloží do přemístitelného objektového souboru, který je poté přilinkován k výsledné aplikaci. Bez separátního překladu musí být všechny proměnné a funkce použité v GPU kódu ve stejném souboru (resp. ve stejné jednotce překladu).

2.3 Metody profilování

Pro profilování aplikace je nutné získat data o jejím běhu, a to ideálně tak, aby to samotný běh aplikace ovlivnilo co nejméně. Základními způsoby pro sběr těchto dat jsou vzorkování a instrumentace.

2.3.1 Vzorkování

Při výkonnostním profilování se často používá vzorkování (tzv. *sampling*). Jedná se o statistickou metodu, kdy se program v pravidelných intervalech (například co 10 milisekund) přerušuje a při každém přerušení se uloží informace o jeho současném stavu (zásobník volaných funkcí, stav registrů, čítače procesoru). Poté lze zpětně analyzovat v které funkci program strávil nejvíce času nebo které instrukce v jednotlivých metodách trvaly nejdéle.

Naměřené vzorky lze vizualizovat různými způsoby. Jednou z možností je tzv. flame chart [13], který zobrazuje zaznamenané zásobníkové rámce. Dalším způsobem vizualizace je tzv. call graph, pomocí kterého lze vidět, které funkce se navzájem volaly a kolik času bylo v jednotlivých funkcích stráveno. Více o vzorkování a grafech vzájemného volání se lze dozvědět v [1].

Vzorkování má výhodu v tom, že téměř neovlivňuje výkonnostní charakteristiky sledovaného programu a není složité jej naimplementovat. Jedná se však o stochastický vzorkovací proces, takže nelze zachytit veškeré důležité informace o běhu programu. Z tohoto důvodu není tato metoda vhodná pro zachycení paměťových přístupů. Vzorkování je používáno například v nástrojích VTune [4] nebo CodeXL [14].

2.3.2 Instrumentace

Další z možností, jak získat informace o běhu aplikace, je instrumentace. Ta spočívá v (obvykle automatizovaném) umístění kódu na vybraných místech programu (například v prologu či epilogu funkce). Tento kód zaznamenává informace za běhu programu nebo chování programu sám modifikuje. Kromě profilování lze tuto techniku použít například k analýze korektnosti programů [15]. Výhoda instrumentace spočívá v tom, že umožňuje zachytit veškeré výskyty sledovaných událostí, na rozdíl od náhodného vzorkování. Nevýhodou je fakt, že pokud je instrumentační kód volán často a provádí složité operace, tak může značně zpomalit provádění programu a zároveň zkreslit naměřená data. Kdyby například instrumentace zaznamenávala informace o každé provedené instrukce, tak cena instrumentace bude mnohonásobně vyšší než samotné provádění programu. Takto použitá instrumentace poté měří spíše dobu svého provádění než dobu provádění programu. Účelem profilovacího nástroje vyvíjeného v této práci však není měřit dobu výpočtu programu, ale získat data o přístupech. Pro zaznamenání paměťových operací je tento přístup nutný a proto je také v nástroji použit.

Instrumentaci lze rozdělit do dvou kategorií - statická a dynamická. Dynamická instrumentace modifikuje kód programu až za jeho běhu a provádí se tedy přímo na instrukcích programu. Výhoda dynamického přístupu je v tom, že si vystačí se spustitelným souborem a nevyžaduje pro instrumentaci překlad aplikace (což je užitečné hlavně v případě, kdy není původní zdrojový kód k dispozici). Nicméně je relativně obtížné ji naimplementovat. Kromě čtení a generování instrukcí je nutné taky částečně duplikovat funkci překladače, protože naivní přidání nebo úprava instrukcí v programu může způsobit jeho pád nebo nežádoucí změnu chování. Instrumentační nástroje obvykle nejprve převedou instrukce programu do abstraktního formátu, který umožňuje jednoduchou modifikaci. Jakmile se do kódu přidají potřebné instrumentační funkce, tak dojde k zpětnému překladu na strojové instrukce tak, aby program zůstal validní. Dynamickou instrumentaci využívá například Valgrind [16] nebo Pin [17].

Statická instrumentace probíhá před spuštěním programu a lze ji provádět na zdrojovém kódu, přechodné reprezentaci programu nebo už na výsledném spustitelném souboru s instrukcemi. Rozdíly mezi těmito způsoby jsou hlavně v tom, kolik informací o původním programu dokáží

získat a zda pro instrumentaci vyžadují překlad programu. Níže následuje popis jednotlivých reprezentací spolu s výhodami a nevýhodami jejich instrumentace.

Instrumentace zdrojového kódu

Zdrojový kód poskytuje vysokou úrovni abstrakce a obsahuje veškeré informace o zápisu programu. Díky tomu, že je reprezentován textem, tak pro velmi jednoduchou instrumentaci lze použít i prosté regulární výrazy. Kvůli jeho vysoké abstrakci nicméně nelze instrumentací zachytit nízkoúrovňové detaily o chování programu (například jednotlivé přístupy do paměti nebo práci s registry procesoru). Nevýhodou je také nutnost přeložit celý program po instrumentaci. Původní zdrojový kód musí být před instrumentací zálohován, aby nebyl znehodnocen. Popřípadě je nutné upravit překlad programu tak, aby vytvářel spustitelný soubor z již instrumentovaného zdrojového kódu.

Instrumentace přechodného formátu

Přechodný formát je používán pro popis programu v překladačích. Usnadňuje analýzu programu bez nutnosti starat se o syntaxi konkrétního programovacího jazyka a o detaily cílové architektury. Obvykle je tvořen sadou instrukcí pro virtuální stroj, které jsou ve finální fázi překladu převedeny na strojové instrukce pro zvolenou instrukční sadu. Příkladem tohoto formátu je již zmíněný LLVM IR. Jelikož IR vzniká ze zdrojového kódu, obsahuje metadata s kompletními informacemi o programu, zároveň je ale na mnohem nižší úrovni abstrakce než zdrojový kód a umožňuje tak analyzovat a instrumentovat jednotlivé instrukce programu. Tvoří tak užitečný kompromis mezi instrumentací zdrojového kódu a strojových instrukcí. Jeho nevýhodou je nutnost překladu programu při instrumentaci a také absence univerzálního standardu (každý překladač má obvykle svůj vlastní formát). Překladače také nemusí poskytovat rozhraní pro instrumentaci svého IR formátu.

Instrumentace strojových instrukcí

Strojové instrukce postrádají část informací z původního zdrojového kódu, umožňují ale instrumentovat a analyzovat veškeré detaily chování programu. Na rozdíl od předchozích způsobů reprezentace nevyžadují opětovný překlad programu po instrumentaci. Instrumentovat přímo instrukce je nicméně obtížné ze stejných důvodů jako u dynamické instrumentace – je třeba načíst instrukce do abstraktního formátu a poté je převést zpátky do instrukcí procesoru pro zaručení korektnosti programu modifikovaného instrumentací.

3 Existující nástroje

Tato kapitola popisuje existující profilovací a instrumentační nástroje pro platformu CUDA a dále zmiňuje nástroje určené pro vizualizaci paměťových přístupů.

3.1 Profilovací nástroje

Pro grafické karty existuje pouze zlomek profilovacích nástrojů ve srovnání s nástroji dostupnými pro profilování procesorů. U obou platforem však platí, že nejpokročilejší nástroje jsou vyvíjeny samotným výrobcem daného hardwaru (Intel, Nvidia, AMD) [4] [18] [14]. Mikroarchitektury moderních procesorů jsou velmi složité a často nejsou veřejně známy veškeré jejich detaily. Pouze jejich výrobce tak může vytvořit nástroj, který z nich dokáže vyčíst maximum informací. V této kapitole jsou popsány profilovací nástroje pro platformu CUDA.

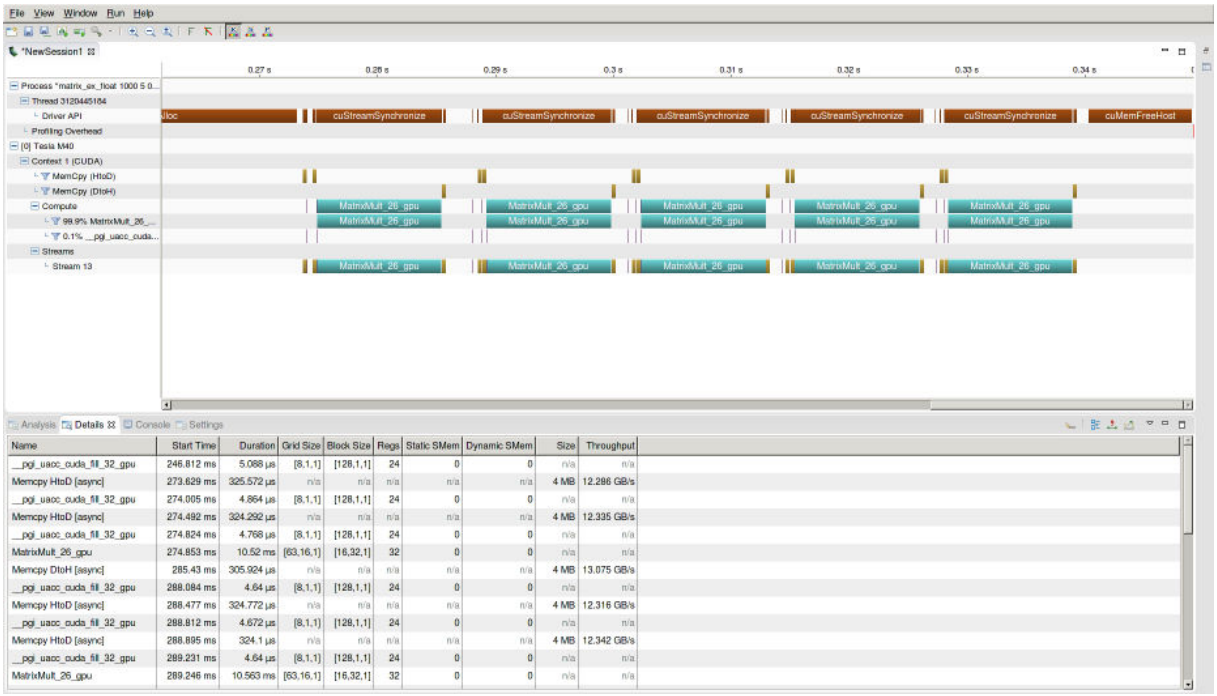
3.1.1 Nsight

Nsight je vývojové prostředí integrované v programech Eclipse a Visual Studio, které usnadňuje vývoj, profilování a ladění programů pro grafické karty od Nvidie. Je vyvíjené přímo společností Nvidia, takže dokáže zjistit velké množství údajů o provádění CUDA kódu pomocí používání nezdokumentovaných funkcí. Nabízí pokročilé ladící nástroje, například krokování v kernelech, zobrazení registrů jednotlivých vláken, rozložení vláken mezi warpy, kontrolu chybných přístupů do paměti aj. Umožňuje také velice detailně profilovat provádění GPU kódu díky vestavěnému nástroji s názvem Visual Profiler.

Rozhraní tohoto nástroje si lze prohlédnout na obrázku 2. Po spuštění CUDA programu se v nástroji zobrazí časová osa, ze které lze vyčíst dobu trvání paměťových přenosů mezi procesorem a grafickou kartou a také dobu provádění jednotlivých kernelů. Díky této vizualizaci lze například určit, jestli by dávalo smysl překrýt přenos paměti s výpočtem. Kromě ní však nástroj nabízí i velmi detailní profilování kódu. Nástroj sám postupně vede programátora k informacím o programu, které ho zajímají. Nejprve se program spustí a zobrazí se vytížení jednotlivých kernelů. Poté je programátor vyzván k tomu, aby si vybral kernel, který ho zajímá, a při příštím spuštění už se profilování zaměřuje na něj. Analýza kernelu pak nabízí zkoumání propustnosti a odezvy paměti nebo aritmetických výpočtů. V nástroji si lze také prohlédnout zdrojový kód kernelu s vyznačenými řádky, které zabíraly nejvíce času výpočtu, např. kvůli tomu, že instrukce čekaly na data z paměti.

Tento nástroj poskytuje detailní analýzu běhu programu, která je nepostradatelná pro získání maximálního výkonu z grafických karet. Všechny údaje nabízené tímto nástrojem jsou však agregované. Jsou k dispozici údaje o počtu nebo poměru událostí v programu, nelze z něj ale získat informace o konkrétních přístupech do paměti. Z nástroje nelze přímo vyčíst, jaké přístupové vzory do paměti vlákna používala nebo kde přesně může docházet k paměťovým konfliktům.

Visual Profiler se snaží mít co nejmenší vliv na výkon programu, proto ani takto detailní údaje zaznamenávat efektivně nemůže.



Obrázek 2: Časová osa kernelů ve Visual Profileru

Zdroj: `https://www.microway.com/hpc-tech-tips/accelerating-code`

3.1.2 nvprof

Nvprof je nástroj bez grafického prostředí určený pro profilování CUDA programů. Pomocí vzorkování umožňuje rychle zjistit, které funkce v CUDA programech zabírají nejdelší čas výpočtu. Kromě profilování GPU může zároveň vzorkovat i CPU kód a poskytnout tak ucelený pohled na výkon aplikace. Data naměřená tímto programem lze vizualizovat ve Visual Profileru, čehož lze využít například pro profilování aplikací v prostředích bez grafického rozhraní.

3.1.3 CUPTI

CUPTI je programovatelné rozhraní pro získávání profilovacích údajů a čtení výkonnostních čítačů z CUDA akcelerátorů. Jedná se o základní profilovací rozhraní, které využívají profilovací nástroje a knihovny (například PAPI, TAU nebo VampirTrace [19, 20]). Celkem nabízí čtyři typy rozhraní - Activity, Callback, Event a Metric. Activity rozhraní umožňuje získávat asynchronně záznamy o konkrétních operacích provedených na GPU. Lze například zjistit, kolikrát byl proveden konkrétní zápis do paměti, jak probíhaly přenosy paměti nebo kolikrát a s jakými parametry byl spuštěn nějaký kernel. Rozhraní Callback umožňuje nadefinovat si funkci, která bude zavolána při spuštění zvolené funkce CUDA rozhraní. Lze tak například provést libovolný

kód při každém kopírování nebo alokaci paměti na grafické kartě. Rozhraní Event umožňuje zaznamenat čítače vestavěné v hardwaru grafických karet. Tyto čítače obsahují například informace o počtu provedených instrukcí, poměru požadované a skutečné paměťové odezvy pro jednotlivé typy paměti nebo počet úspěšných přístupů do cache paměti. Tyto údaje pak lze v agregované podobě číst pomocí Metric rozhraní.

CUPTI sice umožňuje zaznamenat spoustu informací, které by byly užitečné pro nástroj vyvíjený v této práci, tyto údaje jsou ale stále agregované. Pro detailní vizualizaci konkrétních paměťových přístupů je potřeba mít data o každém přístupu zvlášť, a tento způsob sběru dat CUPTI nepodporuje.

3.2 Instrumentační nástroje

Existuje několik robustních nástrojů, které umožňují instrumentovat klasické CPU programy [16, 17, 21]. Instrumentace aplikací pro grafické karty však není tak rozšířená. Pro platformu CUDA existují nástroje SASSI [22], Lynx [23] a Panoptes [24].

SASSI

SASSI je nástroj pro instrumentaci CUDA kódu, který obsahuje vlastní verzi překladače PTX (zdrojový kód překladače není zveřejněn). Umožňuje spouštět uživatelem dodaný kód v C++ při různých událostech v CUDA programu (provedení instrukce, zavolání funkce). Události obsahují informaci o prováděném kódu, např. pro paměťový přístup lze zjistit adresu čtení či zápisu, umístění instrukce ve zdrojovém kódu aj. Tato funkcionalita by byla dostatečná k zaznamenání paměťových přístupů, nicméně SASSI už není aktivně vyvíjeno. Podporuje pouze CUDA 7 z roku 2015 a pro překlad vyžaduje velmi staré verze překladačů. Z důvodu uzavřeného zdrojového kódu není možné kód jednoduše modifikovat pro novější verze.

Lynx

Lynx je součástí frameworku Ocelot [25], který umožňuje překládat CUDA kód pro běh na platformách, které jinak nejsou kompatibilní s platformou CUDA (CPU a grafické karty AMD). Lynx používá komponenty Ocelotu pro zpracování a překlad PTX kódu, který tak lze dynamicky instrumentovat. Stejně jako SASSI už Lynx ani Ocelot nejsou aktivně vyvíjeny (poslední verze, kterou podporovaly, je CUDA 5) a nelze je použít pro instrumentování moderních CUDA programů.

Panoptes

Panoptes za běhu aplikace zachytává CUDA volání a dynamicky modifikuje chování programu. K tomu využívá parser PTX. Při spuštění kernelu načte PTX kód, upraví ho a upravený kód poté pošle dále CUDA driveru k překladu. Autor tohoto nástroje jej využil k implementaci kontroly korektnosti paměťových přístupů, inspirované nástrojem Memcheck. Podobná funkcionalita byla

později přidána přímo do CUDA frameworku společností Nvidia. Tento nástroj již dlouho není aktivně vyvíjen a nepodporuje veškeré instrukce PTX.

Z příkladu uvedených nástrojů je zřejmé, že udržovat instrumentační nástroje pro grafické karty je velmi obtížný úkol. Architektura a programovací rozhraní grafických karet se často mění a Nvidia neposkytuje žádné otevřené zpětně kompatibilní rozhraní pro práci se svým překladačem. Jediným udržitelným řešením je tak v současné době použití překladače LLVM, který překlad CUDA programů podporuje a je aktivně vyvíjen širokou komunitou společností i jednotlivců.

3.3 Vizualizační nástroje

Vizualizace přístupů do paměti nebývá běžnou součástí profilovacích nástrojů. Ty obvykle vizualizují komunikaci mezi procesy [19, 20], čas strávený v jednotlivých funkcích [13] nebo dynamické alokace paměti [26, 27]. Analýza přístupů do pamětí zatím našla své uplatnění v reverzním inženýrství a také u NUMA architektur. Následuje popis několika nástrojů, které se věnují vizualizaci přístupů do paměti.

Tracectory

Tracectory [28] je nástroj, který analyzuje záznamy běhu Windows aplikací generované ladícím nástrojem OllyDbg. Z těchto záznamů poté vytváří jednoduchý 2D graf, kde je na horizontální ose adresa v paměti a na vertikální ose čas. Ukázku tohoto grafu si lze prohlédnout na obrázku 8. Díky této vizualizaci lze jednoduše rozpoznat časté vzory přístupu do paměti. Například instrukce, které čtou či zapisují z po sobě jdoucích adres na grafu vytvoří diagonálu. Naopak náhodné přístupy do paměti, které jsou často neefektivní, na grafu vytvoří mřížku bez zjevného řádu. Dále lze v tomto nástroji zjistit, jaká hodnota byla uložena v paměti v danou chvíli. Motivací pro tvorbu tohoto nástroje byla reverzní analýza kódu. Díky vizualizaci paměťových přístupů lze totiž analyzovat například chování obfuskovaných programů.

Memview

Memview [29] vizualizuje mapu paměti běžícího procesu v reálném čase. Zobrazuje jednotlivá čtení a zápisy v paměti a umožňuje škálovat zobrazení od celého adresního prostoru až po jednotlivé byty. K zaznamenání přístupů do paměti využívá instrumentační framework Valgrind. Na obrázku 9 lze vidět mapu paměti běžícího procesu. Z obrázku je patrný jeden z problémů vizualizace paměti - k dispozici je ohromné množství dat, ve kterém může být složité se vyznat.

Memory Trace Visualizer

Memory Trace Visualizer [30] je nástroj, který vizualizuje zaznamenanou sekvenci paměťových operací běhu programu. Při vizualizaci probíhá simulace cache paměti, což umožňuje analyzovat přibližnou efektivitu jejího využití. Pro usnadnění orientace v paměti lze zvolit oblast paměti,

která bude zobrazena. Oblasti lze poté interpretovat různým způsobem, například jako klasické jednorozměrné pole nebo jako dvourozměrnou matici. Pro získání paměťových přístupů je použit nástroj CHUD[6] od společnosti Apple.

NUMA architektury

Několik publikací [31, 32] se věnuje také analýze přístupů do paměti u NUMA architektur. Ty mají operační paměť rozdělenou na několik modulů a přístupy do jednotlivých modulů nejsou stejně efektivní ze všech jader či soketů. Je proto užitečné vizualizovat informace o tom, do jaké části paměti jednotlivá vlákna přistupují. U těchto architektur se analyzuje hlavně to, které vlákno poprvé přistoupilo k dané paměťové stránce (first-touch). To poté totiž ovlivňuje, na kterém paměťovém modulu bude stránka fyzicky naalokována. Poté je také důležité zjistit, jestli některá vlákna často nesahají do vzdálených paměťových modulů, což zpomaluje odezvu dané paměťové operace.

V průběhu psaní této práce se nepodařilo dohledat žádný nástroj, který by umožňoval instrumentaci CUDA programů za účelem zaznamenávání paměťových přístupů a jejich následné vizualizace.

[6] Computer Hardware Understanding Development

4 Profilovací nástroj

Profilovací nástroj vytvářený v této práci se skládá ze dvou částí. První, která se stará o sběr dat za běhu programu, je popsána v této kapitole. Druhá část, která naměřená data vizualizuje uživateli, je popsána v kapitole 5.

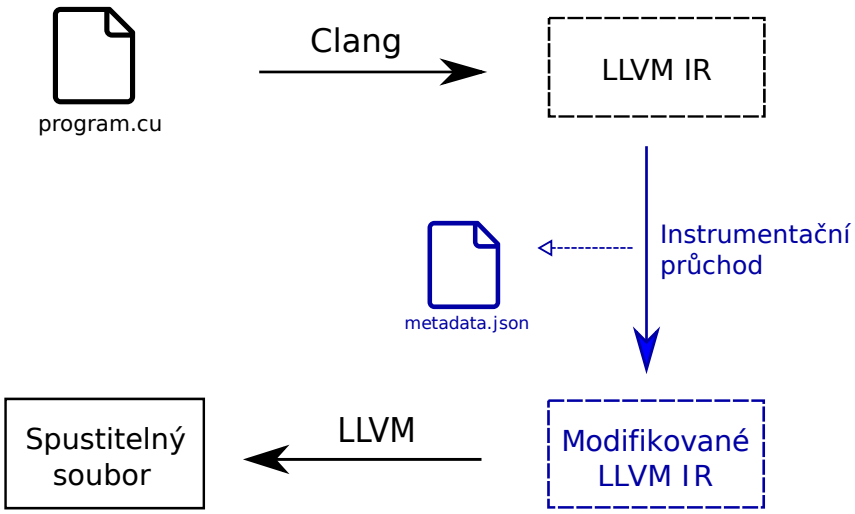
4.1 Specifikace požadavků

Cílem této práce bylo vytvořit nástroj, který by umožnil zaznamenat paměťové přístupy grafické karty v CUDA aplikaci. Naměřená data by měla obsahovat detailní informace o paměťovém přístupu a umožnit identifikovat vlákno, které přístup provedlo. Pokud budou údaje k dispozici, měly by se také uchovávat ladící informace o umístění instrukcí ve zdrojovém kódu a datovém typu proměnných. Pro účely vizualizace je také nutné, aby nástroj zaznamenával paměťové alokace na grafické kartě a umožnil přiřadit je ke konkrétním přístupům do paměti. Nástroj by měl být jednoduše použitelný a neměl by vyžadovat provádění velkého počtu změn ve zdrojovém kódu aplikace. Jelikož nástroj není určen k měření doby výpočtu programu a zaznamenává velké množství dat, tak se počítá s tím, že může znatelně ovlivnit výkon programu. Profilovaný program by ale neměl být nástrojem zpomalen natolik, aby se nedal použít alespoň pro malé testovací instance určené k experimentování s optimalizováním daného programu.

4.2 Architektura

Profilovací nástroj je tvořen LLVM průchodem a knihovnou pro zaznamenávání paměťových přístupů za běhu aplikace. Pro implementaci obou komponent byl zvolen jazyk *C++*. V tomto jazyce je napsané LLVM i programy, pro jejichž profilování je nástroj určený, jedná se tak o logickou volbu pro tvorbu průchodu. C++ navíc umožňuje pracovat s pamětí počítače na nízké úrovni, což je nutné pro získání detailních informací o přístupech grafické karty do paměti za běhu aplikace. Průchod nástroje se zakomponuje do běžného překladu CUDA aplikace pomocí překladače Clang. Pro použití průchodu stačí přidat několik přepínačů při překladu programu a v souborech s kernely vložit hlavičkový soubor profilovacího nástroje. Jako profilovací metoda byla zvolena instrumentace, která na rozdíl od vzorkování umožňuje zaznamenat veškeré přístupy vyskytující se ve zdrojovém kódu programu. Instrumentace sice zpomalí běh programu, nicméně měření doby výpočtu není účelem tohoto nástroje, autor práce to tedy považuje za rozumný kompromis.

Průběh překladu s instrumentací je znázorněn na obrázku 3. Černě vyznačené části znázorňují standardní průběh překladu, modré části zobrazují instrumentaci a soubory, které jsou při ní vytvářené (přerušované šipky). Překladač Clang nejprve přeloží zdrojový kód do formátu LLVM IR. Poté průchod profilovacího nástroje přidá do IR kód, který bude za běhu zaznamenávat paměťové přístupy. Při provádění průchodu se vygenerují soubory s metadaty, které obsahují mapování na původní zdrojový kód (čísla řádků, datové typy) pro jednotlivé kernely. Takto



Obrázek 3: Architektura instrumentace

modifikovaný program se poté předá do backendu LLVM, který z něj vygeneruje spustitelný soubor.

Strukturu běhu instrumentovaného programu si lze prohlédnout na obrázku 4. K programu je přilinkována knihovna pro sběr přístupů a knihovna pro zaznamenávání alokací. Instrukce vložené instrumentací volají funkce z knihovny pro sběr přístupů a zaznamenávají přístupy a alokace. Dodatečné alokace, které nešlo instrumentovat, jsou zachyceny knihovnou pro alokace, zpracovány a předány knihovně pro sběr přístupů. Po každém spuštění kernelu se vygeneruje soubor s paměťovými přístupy daného kernelu. Před ukončením programu je ještě vytvořen soubor s údaji o běhu celé aplikace. Všechny tyto soubory poté tvoří vstup pro vizualizační aplikaci.

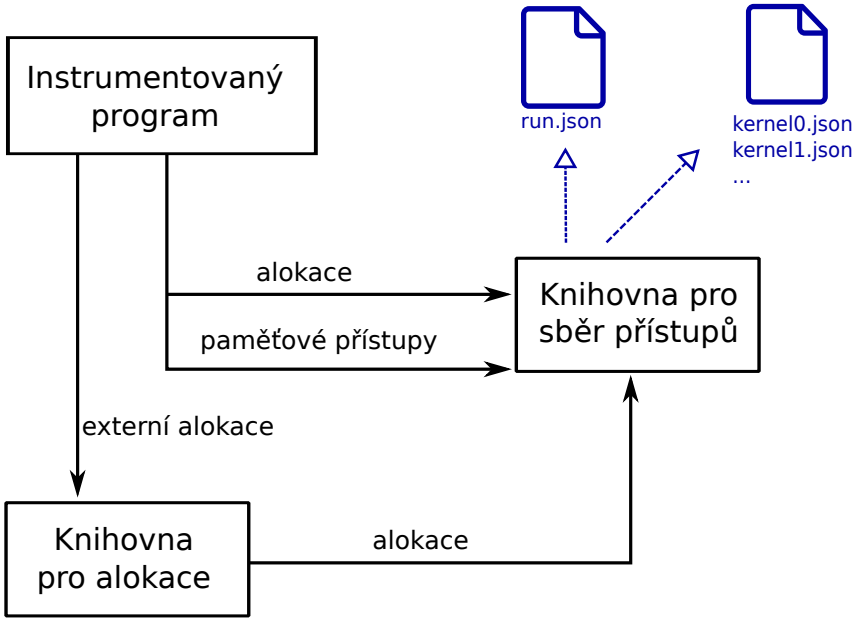
Profilovací nástroj se skládá z několika izolovaných částí, které spolupracují na zaznamenávání paměťových přístupů. Tyto komponenty jsou detailně popsány v následujících podkapitolách.

4.3 Instrumentační průchod

Instrumentační průchod se stará o vložení volání profilovacích funkcí do instrumentovaného programu. Na svém vstupu obdrží kompilační jednotky programu (obvykle jeden implementační zdrojový soubor tvoří jednu kompilační jednotku) ve formě LLVM IR modulů. CUDA zdrojové soubory (`.cu`) jsou rozděleny na dva IR moduly. Jeden obsahuje CPU instrukce, které inicializují parametry kernelů a spouští je. Druhý obsahuje samotný GPU kód kernelů, který je spouštěn na grafické kartě. Nástroj instrumentuje oba dva typy modulů.

Instrumentace CPU kódu

V modulu s CPU kódem se vyhledají dva typy událostí – spuštění kernelu a alokace paměti. Před spuštěním kernelu je nutné nachystat pole přístupů na grafické kartě a po ukončení kernelu



Obrázek 4: Běh instrumentovaného programu

tato data zkopírovat na procesor a vypsat do souboru. Informace o paměťových alokacích jsou užitečné, aby bylo možné odvodit, k jakému typu paměti jednotlivá CUDA vlákna přistupovala.

Spuštění kernelu je ve zdrojovém kódu reprezentováno voláním funkce. Clang toto volání při překladu do IR rozdělí na volání několika CUDA funkcí, které nejprve nastaví parametry pro kernel a poté jej spustí pomocí funkce `cudaLaunch`. Právě tato funkce je v IR vyhledána. Těsně před ni se vloží volání funkce, která vytvoří za běhu programu kontext pro spouštěný kernel. Za ni se vloží volání funkce, která zaznamenaná data zkopíruje z grafické karty a dále zpracuje. Tyto funkce jsou detailněji popsány v sekci 4.4. Textová reprezentace instrumentovaných instrukcí v LLVM IR je zobrazena ve výpisu 2. Výpis 3 zobrazuje, jak by přibližně instrumentované spuštění kernelu mohlo vypadat ve zdrojovém kódu (řádky s šedým pozadím značí kód vložený instrumentací).

```
%7 = alloca %"struct.KernelContext"
call void @_cu_initKernelContext(%"struct.KernelContext"* %7,
    [7 x i8]* @__cu_ProfileGlobal_0)
call void @__cu_kernelStart(%"struct.KernelContext"* %7)
%8 = call i32 @cudaLaunch(i8* bitcast (void (i32*)* @_Z6kernelPi to i8*))
call void @__cu_kernelEnd(%"struct.KernelContext"* %7)
call void @__cu_disposeKernelContext(%"struct.KernelContext"* %7)
```

Výpis 2: Instrumentace spuštění kernelu v LLVM IR

```
KernelContext ctx = __cu_initKernelContext("kernel");
__cu_kernelStart(&ctx);
kernel<<<128, 256>>>(cudaPtr);
__cu_kernelEnd(&ctx);
__cu_disposeKernelContext(&ctx);
```

Výpis 3: Instrumentace spuštění kernelu v C++

Dále se při instrumentaci vyhledají funkce pro alokaci a dealokaci GPU paměti. Těch CUDA obsahuje několik desítek, stěžejní jsou ale dvě základní funkce `cudaMalloc` (alokace paměti) a `cudaFree` (uvolnění paměti). Jiné funkce nástroj nevyhledává, v případě potřeby ale lze podporu speciálních alokačních funkcí do nástroje přidat. Před alokační funkce jsou vložena volání funkcí, které za běhu programu tyto alokace zaznamenají. Jsou jim předány údaje o velikosti alokované paměti a ladící údaje ve formě datového typu a umístění volání alokační funkce ve zdrojovém kódu. Po každém spuštění kernelu se vyhledají všechny paměťové alokace, které jsou v danou chvíli aktivní a jsou vypsány do souboru spolu s paměťovými přístupy provedenými daným kernelem.

Při vkládání IR instrukcí, které volají funkce, je nutné jim předávat parametry. Základní datové typy, které lze uložit do registru (čísla nebo ukazatele) je předat snadné. Pro zaznamenání ladících informací je nicméně nutné předávat i řetězce. Ty jsou v jazyce C obvykle reprezentovány pomocí ukazatele na pole znaků ukončené znakem s hodnotou 0. Instrumentační průchod tak musí pro předání každého řetězce nejprve vytvořit globální proměnnou, která bude obsahovat znaky daného řetězce, a poté funkci předat ukazatel na tuto proměnnou. Pro zamezení jmenných kolizí je před název takto vytvořených proměnných přidán prefix a na konec jejich názvu je přidáno číslo, které se po vytvoření každého řetězce inkrementuje. Proměnné jsou navíc vytvořeny se soukromou vazbou (private linkage), díky čemuž nejsou viditelné z ostatních modulů. Aby se předešlo zbytečnému plýtvání místa, tak se při instrumentaci hodnoty řetězců uchovávají v cache. Pokud vznikne požadavek na předání řetězce, který už byl jednou vytvořen, tak se nevytváří nová globální proměnná a pouze se předá ukazatel na již existující proměnnou.

Ukázku části instrumentačního nástroje, která instrumentuje volání funkce `cudaMalloc`, si lze prohlédnout ve výpisu 4. Za volání `cudaMalloc` je vloženo volání funkce `__cu_malloc`, která je naimplementovaná v knihovně pro sběr přístupů. `cudaMalloc` přijímá jako první parametr adresu paměti, do které zapíše adresu vytvořené alokace. Tato adresa je spolu s velikostí alokace, datového typu alokované paměti, názvu proměnné, do které se adresa ukládá a umístění ve zdrojovém kódu předána vkládané funkci. Za běhu programu jsou pak tyto údaje zaznamenány do seznamu alokací.

Instrumentují se pouze moduly, v jejichž zdrojovém kódu je vložen hlavičkový soubor nástroje. Tyto moduly jsou rozpoznány podle toho, že v nich jsou nadefinovány funkce z tohoto hlavičkového souboru. Při instrumentaci je ale samozřejmě nutné tyto funkce vynechat, jinak by

```
void MemoryAlloc::handleCudaMalloc(CallInst* call)
{
    auto emitter = this->context.createEmitter(call->getNextNode());

    Value* addressLoad = emitter.getBuilder().CreateLoad(call->getOperand(0));

    auto& values = this->context.getValues();
    Type* valueType = getMallocValueType(call);
    std::string typeStr = this->context.getTypes().stringify(valueType);

    auto debug = getCudaMallocDebug(call);
    emitter.malloc(
        addressLoad, // adresa
        call->getOperand(1), // velikost alokace
        values.int64(valueType->getPrimitiveSizeInBits() / 8), // velikost typu
        values.createGlobalCString(typeStr), // název typu
        values.createGlobalCString(debug.getName()), // název proměnné
        values.createGlobalCString(getFullPath(debug)) // umístění ve zdrojovém kódu
    );
}
```

Výpis 4: Instrumentace funkce `cudaMalloc`

došlo k přetečení zásobníku kvůli vloženým voláním, které by volaly rekurzivně samy sebe. Pro každou funkci se tak nejprve kontroluje, jestli její název neobsahuje prefix nástroje. Pokud ano, tak je při instrumentaci ignorována.

Instrumentace GPU kódu

V kódu pro grafickou kartu se nejprve vyhledají všechny kernely. Ty lze detekovat pomocí atributu, který nastavuje Clang při překladu zdrojového kódu. Na začátek každého kernelu je přidán inicializační kód, který vždy provede pouze první vlákno z výpočetní mřížky (vlákno na pozici 0.0.0.0.0.0). Tento kód zaznamená informace o využití sdílené paměti a rozměrech výpočetní mřížky pro dané spuštění kernelu. Za inicializační blok je do kódu vložena synchronizační instrukce, aby ostatní vlákna počkala na konec inicializace.

Dále jsou v kernelu nalezeny všechny přístupy do paměti (čtení i zápis). Z těch jsou vybrány přístupy do globální nebo sdílené paměti grafické karty. Přístupy do registrů a k lokálním proměnným obvykle netvoří úzká hrdla aplikací a zbytečně by zahlcovaly vizualizaci. Instrumentování těchto přístupů lze případně volitelně zapnout. Přístupy do sdílené paměti jsou detekovány pomocí jednoduché heuristiky. Pro daný přístup se analyzuje, odkud načítá nebo kam zapisuje data (přes ukazatel, přístupem do struktury, pomocí lokální proměnné atd.). Tento postup je aplikován rekurzivně, dokud se nedojde k bodu, který již nelze dále analyzovat (například ukazatel, u kterého není znám adresní prostor). Pokud na konci tohoto řetězce je proměnná naalokovaná ve sdílené paměti, tak je přístup označen za přístup do sdílené paměti. LLVM obsahuje vestavě-

```
__global__ void kernel(int* data)
{
    __shared__ int sharedMem[10];
    if (__cu_isFirstThread()) {
        __cu_markSharedBuffer(/* adresa */ sharedMem,
            /* velikost pole */ sizeof(sharedMem),
            /* velikost pruku pole */ sizeof(sharedMem[0]),
            /* id datového typu */ 0
        );
        __cu_storeDimensions(); // uložení rozměrů mřížky
    }
    __syncthreads(); // synchronizace vláken

    __cu_load(/* adresa */ data + threadIdx.x, /* velikost */ sizeof(int),
        /* adresní prostor - globální paměť */ 0,
        /* id datového typu */ 0,
        /* lokace v kódu */ 0, /* hodnota */ data[threadIdx.x]
    );
    __cu_store(sharedMem + threadIdx.x, sizeof(int),
        /* adresní prostor - sdílená paměť */ 1,
        0, 1, sharedMem[threadIdx.x]
    );
    sharedMem[threadIdx.x] = data[threadIdx.x];
}
```

Výpis 5: Ukázka instrumentovaného kernelu

nou podporu pro pokročilejší verzi této heuristiky, která analyzuje ukazatele, tato analýza však přestala fungovat pro kernely v nových verzích LLVM.

Před zápis, resp. za načtení je vloženo zavolání funkce, která za běhu daný přístup zaznamená. Popis těchto funkcí je v sekci 4.5. Zápis lze instrumentovat před jeho provedením, protože je známá hodnota, která se zapisuje. Načtení je instrumentováno až poté, co se provede, aby šlo zaznamenat hodnotu, která byla načtena.

Ve výpisu 5 je znázorněno, jak by mohl vypadat instrumentovaný kernel ve zdrojovém kódu. Na začátku kernelu dochází k zaznamenání údajů o sdílené paměti a rozměrech výpočetní mřížky prvním vláknem. Poté dochází k zaznamenání načtení z globální paměti a zápisu do sdílené paměti.

Instrumentovány jsou opět pouze ty CUDA soubory, do kterých byl vložen hlavičkový soubor nástroje `CuprRuntime.h`. Tímto lze určit, které kernely se mají profilovat na úrovni zdrojových souborů. Dále lze filtrovat profilované kernely pomocí regulárního výrazu. Parametry ovlivňující instrumentaci jsou uvedeny v sekci 4.7.

Při instrumentaci se pro každý instrumentovaný kernel vytvoří soubor s metadaty. Tento soubor obsahuje mapovací tabulku datových typů a umístění v kódu pro jednotlivé paměťové alokace a paměťové přístupy. Tyto ladící údaje jsou reprezentovány řetězci, které však nelze

na grafickou kartu předat tak, jak to bylo popsáno výše při popisu instrumentace CPU kódu. Jednak by řetězce zbytečně plýtvaly místem na grafické kartě, a také by se pro každý z nich musela speciálně alokovat paměť. Z toho důvodu jsou na grafickou kartu datové typy a umístění v kódu předávány v podobě čísel. Mapovací tabulku v tomto souboru poté lze použít pro zpětné získání názvů datových typů a umístění ve zdrojovém kódu. Kromě ladících metadat se do tohoto souboru ukládá také obsah zdrojového souboru s daným kernelem. Díky tomu lze při vizualizaci filtrovat přístupy do paměti podle jejich umístění ve zdrojovém kódu.

Místo generování souboru pro každý kernel by bylo uživatelsky přívětivější, kdyby se metadata vložila přímo do instrumentovaného programu, který by je mohl do jednoho souboru zapsat až při svém spuštění. Instrumentační nástroj by mohl sesbírat veškeré informace o programu a do jeho vstupního bodu vygenerovat kód, který by metadata vypsal do souboru. LLVM nicméně instrumentačnímu průchodu předkládá jednotlivé moduly po jednom, zcela izolovaně a bez jasně určeného pořadí. Mohlo by se tak stát, že modul se vstupním bodem programu přijde dříve než budou zpracovány všechny ostatní moduly a tím pádem by nebyla k dispozici veškerá metadata. Z tohoto důvodu je nutné generovat soubory s metadaty zvlášť po instrumentaci každého modulu.

Po spuštění instrumentovaného programu se v jeho pracovním adresáři vytvoří složka, jejíž název se skládá z prefixu *cupr* a časového razítka s časem spuštění programu. V této složce se za běhu programu vytváří soubory pro jednotlivá spuštění kernelu (pro každé spuštění kernelu je vygenerován jeden soubor s paměťovými přístupy). Zároveň se do této složky nakopírují veškerá metadata kernelů, která jsou nalezena v pracovní složce profilovaného programu, aby poté šlo celou složku jednoduše nahrát do vizualizační aplikace.

Výhodou této formy statické instrumentace je, že jsou k dispozici údaje ze zdrojového kódu a lze tak použít ladící údaje o datových typech a umístění instrukcí v kódu. Tyto údaje za běhu aplikace už totiž nejsou zcela k dispozici. Nevýhodou tohoto přístupu je, že instrumentovat lze pouze moduly, který jsou překládány, nelze tedy upravit již přeložené externí knihovny. To má za následek, že pokud by například došlo k alokaci CUDA paměti v externí knihovně, tak by tato alokace nebyla zaznamenána, protože by její volání nebylo instrumentované. Z tohoto důvodu byla pro účely profilovacího nástroje vytvořena jednoduchá knihovna, která dokáže sledovat CUDA alokace za běhu aplikace. Zaznamená se tak alespoň část informací o externích alokacích. Tato knihovna je popsaná v sekci 4.6. Ke spuštění kernelu samozřejmě může také dojít v externí knihovně. V tom případě je ale pravděpodobné, že autorem kernelu není uživatel nástroje a nemá tedy smysl profilovat v něm paměťové přístupy.

4.4 Sběr dat na CPU

Profilovací nástroj obsahuje knihovnu pro správu alokací grafické karty a formátovaný výpis zaznamenaných přístupů. Jak bylo zmíněno v sekci 4.3, před každým spuštěním kernelu je volána funkce z této knihovny, která inicializuje kontext kernelu. Kontext obsahuje informace o kernelu (jméno, rozměry výpočetní mřížky), naměřená data o paměťových přístupech a alokacích sdílené

paměti a také časovač pro měření doby výpočtu kernelu. Při jeho inicializaci se zapne časovač a na grafické kartě se naalokuje paměť pro ukládání informací o paměťových přístupech. Po ukončení kernelu se zaznamenaná data přenesou z grafické karty do paměti procesoru a poté jsou zformátována a vypsána do souboru. Synchronní kopírování dat z GPU po každém spuštění kernelu způsobí, že v profilovaném programu nebude možné překrývat spouštění více kernelů naráz. To teoreticky může ovlivnit chování programu, pokud tuto funkcionalitu využívá, protože serializací kernelů může dojít ke zpomalení výpočtu. Kernely jsou serializovány, aby se na grafické kartě nemuselo vytvářet více bufferů pro paměťové přístupy. Ty jsou velké a na kartě by mohla dojít paměť. Navíc by bylo komplikované určovat, který buffer použít, kvůli omezení externích proměnných, které je popsáno v následující sekci.

Kromě správy kontextů si knihovna také udržuje seznam všech alokací na grafické kartě. Pro danou alokaci se zaznamenává její adresa a velikost, ladící údaje (pokud jsou k dispozici) a stav - po uvolnění paměti se daná alokace označí jako neplatná. Díky tomu by šlo i detekovat přístupy do nevalidní paměti, to ale není motivací vyvíjeného nástroje. Pro tuto funkcionalitu lze použít například nástroj CUDA Memcheck[7]. Údaje o alokacích jsou knihovně předávány buď pomocí funkcí vložených instrumentací nebo pomocí knihovny pro zaznamenávání alokací popsané v sekci 4.6.

Jakmile jsou přístupy zkopírovány z GPU na CPU, tak dojde k jejich shluknutí dle warpu. Každý paměťový přístup na grafické kartě je prováděn v rámci jednoho warpu. Když 32 vláken ve warpu provede paměťový přístup, tak většina údajů o daném přístupu bude těmito vlákny sdílená (typ přístupu, velikost, datový typ, lokace v kódu atd.). Lišit se budou pouze pozice jednotlivých vláken, adresy, ke kterým vlákna přistupovala a hodnoty, které četla nebo zapisovala. Z tohoto důvodu jsou přístupy zařazeny do skupin, které společné údaje sdílejí. Skupina daného přístupu je identifikována pomocí pozice vlákna ve výpočetním mřížce, identifikačního čísla warpu přístupu a časového razítka. Díky tomuto shlukování instrumentované programy generují menší soubory. Zároveň je potřeba i pro vizualizaci, je tak výhodnější ho udělat jednou přímo v instrumentovaném programu a ne pokaždé při načítání souborů v prohlížeči, kde shlukování může trvat dlouho.

4.5 Sběr dat na GPU

Pro účely zaznamenání paměťových přístupů je v nástroji poskytován hlavičkový soubor s několika GPU funkcemi. Ten musí být vložen do každého souboru s kernely, který chce uživatel instrumentovat. To je způsobeno tím, že překladač Clang nepodporuje tzv. *device linking* při CUDA kompilaci. Kvůli tomu nelze připojit k CUDA programu externí knihovnu s kódem pro grafickou kartu. Kód pro sběr dat na GPU tak musí být manuálně vložen do zdrojového kódu. Jedná se o jedinou úpravu kódu, kterou je nutné pro profilování provést. Z důvodu nemožnosti odkazovat se na externí proměnné a funkce musí být samotná implementace funkcí na straně

[7] `https://docs.nvidia.com/cuda/cuda-memcheck/index.html`

CPU i GPU přímo v hlavičkovém souboru. To znamená, že když se tento hlavičkový soubor vloží do více zdrojových souborů v rámci jednoho programu, tak dojde k chybě při překladu kvůli porušení tzv. ODR (pravidla jedné definice). Toto pravidlo je definováno v C++ standardu [33] a mimo jiné říká, že každá funkce musí být definovaná v celém programu maximálně jednou. Toto pravidlo se dá obejít použitím modifikátoru `inline`, pomocí kterého lze překladači slíbit, že všechny definice daného symbolu v programu jsou stejné a nevadí tak, když jich bude více. Použitá kombinace volání GPU funkcí vložených instrumentací a absence device linking v Clangu bohužel znemožňuje použití tohoto modifikátoru. Tento problém lze vyřešit třemi možnými způsoby. Prvním řešením by bylo doimplementování chybějící funkcionality do překladače Clang, to je ale mimo rámec této práce. Další možností by bylo překládat CUDA soubory manuálně do objektových souborů a poté je k sobě přilinkovat pomocí překladače *nvcc*, respektive jeho linkeru *nvlink*. To by ale pro uživatele nástroje znamenalo značnou komplikaci překladu programu. Poslední možností, která byla zvolena, je ignorování pravidla ODR, což lze zajistit pomocí přepínače překladače Clang (více je uvedeno v sekci B.1.3 v příloze). Chyby způsobené vícenásobnou definicí stejného symbolu nejsou časté a projevily by se při překladu bez profilování. Zároveň toto řešení neznamená komplikaci překladu aplikace, protože stačí přidat jeden přepínač. Autorovi práce tak tato možnost přišla jako nejlepší kompromis.

Na grafické kartě je potřeba při běhu kernelu zaznamenat každý paměťový přístup. Kernel vykonává až tisíce vláken najednou, je tedy potřeba zápisy přístupů synchronizovat. K tomu jsou využity atomické instrukce. Při každém zápisu vlákno provede atomicky operaci `fetch-and-add`, při které se atomicky inkrementuje hodnota proměnné a zároveň se vrátí její předchozí hodnota. Tato operace inkrementuje index v bufferu, do kterého se přístupy zapisují (alokace tohoto bufferu byla popsána v sekci 4.4). Na původní hodnotu indexu před inkrementací se v bufferu zapíše požadovaná hodnota. Stejný způsob zápisu do pole je použit i pro ukládání informací o proměnných ve sdílené paměti. Pro každý přístup se zaznamenává jeho typ (čtení, zápis), souřadnice vlákna ve výpočetní mřížce, identifikační číslo warpu, adresa a velikost přístupu, adresní prostor (lokální, globální, sdílená paměť) a čtená či zapisovaná hodnota. Pokud byly při instrumentaci k dispozici, tak se uloží také datový typ přístupu a umístění instrukce ve zdrojovém kódu. Poslední zaznamenanou informací je hodnota čítače hodinového signálu procesoru grafické karty. Tato hodnota neodpovídá reálnému času, takže nelze zjistit informace o absolutním čase přístupu. Každý multiprocesor má svůj vlastní čítač, takže ani nelze relativně porovnávat hodnoty mezi přístupy, protože jednotlivé warpy mohly mezi přístupy změnit multiprocesor, na kterém jsou vykonávány. Tato hodnota tak slouží pouze pro shlukování přístupů do skupin, které bylo popsáno v sekci 4.4. Všechna vlákna ve warpu vždy provedou přístup se stejným časovým razítkem, díky tomu lze určit, které přístupy byly provedeny v rámci jednoho warpu.

4.6 Zaznamenání alokací

Tato dynamická knihovna definuje signatury funkcí pro alokaci CUDA paměti a umožňuje tak provést kód při každém zavolání těchto alokačních funkcí. Pomocí mechanismu LD_PRELOAD[8] lze tuto knihovnu připojit ke CUDA programu a zachytávat pomocí ní alokace na grafické kartě. Informace o alokacích jsou pak předány knihovně pro sběr přístupů a následně je zavolána původní CUDA funkce, aby k alokaci paměti skutečně došlo. Tato knihovna byla vytvořena, aby se zachytily i alokace paměti v externích knihovnách, které nelze instrumentovat. Knihovna ale zachytává všechny alokace, tedy i ty instrumentované. Instrumentační průchod vkládá volání pro zaznamenání alokace až za samotné volání alokace, aby byla k dispozici adresa naalokované paměti. Mohlo by tak dojít k zaznamenání alokace dvakrát. Z tohoto důvodu se při zaznamenání nejprve zkontroluje, zda na dané adrese již neexistuje aktivní alokace. Pokud ano, tak jsou údaje o alokaci přepsány místo vytvoření nové alokace. Informace o alokacích na grafické kartě by se daly získat i pomocí rozhraní CUPTI. Pro účely tohoto nástroje však stačí velmi jednoduché zachytávání alokací. Autorovi práce přišlo jednodušší dopsat tuto funkcionalitu přímo do nástroje a nevyžadovat tak po uživatelovi využití externí závislosti.

4.7 Parametry

Profilovací nástroj lze ovlivnit pomocí parametrů, které se předávají buď při překladu anebo při spouštění instrumentovaného programu pomocí proměnných prostředí. Konkrétní názvy a výchozí hodnoty jednotlivých parametrů jsou uvedeny v sekci B.1.4 v příloze.

Instrumentování lokálních přístupů Při instrumentaci GPU kódu jsou ignorovány čtení a zápisy lokálních proměnných a argumentů funkce. Pokud se při překladu nastaví tento parametr, tak budou instrumentovány všechny dostupné přístupy v kernelech.

Filtrování kernelů podle názvu Tento parametr umožňuje nastavit při instrumentaci programu regulární výraz, který bude použit jako filtr pro kernely. Pokud název kernelu nebude rozpoznán zadaným regulárním výrazem, tak daný kernel nebude instrumentován[9].

Velikost bufferu na GPU Tento parametr ovlivňuje velikost bufferu, který je použit pro ukládání paměťových přístupů na GPU. Pokud kernel provede moc paměťových přístupů a volné místo v bufferu se vyčerpá, tak program vypíše na standardní výstup upozornění. Kapacitu bufferu pak lze zvýšit, což ale samozřejmě zmenší paměť grafické karty dostupnou pro samotný výpočet. Pokud by přístupů bylo moc a grafická karta by neměla dostatek paměti, lze nastavit použití bufferu, který bude v operační paměti procesoru a namapuje se pro použití z GPU. Tím se uvolní paměť na grafické kartě, kvůli komunikaci přes sběrnici a použití atomických operací však může dojít ke značnému zpomalení profilované aplikace.

[8] `http://man7.org/linux/man-pages/man8/ld.so.8.html`

[9] Stále však platí, že aby kernel byl instrumentován, tak musí být v jeho zdrojovém souboru vložen hlavičkový soubor `CuprRuntime.h`

```
message Dim3 {
    required int32 x = 1;
    required int32 y = 2;
    required int32 z = 3;
}

message MemoryAccess {
    required Dim3 threadIdx = 1;            // pozice vlákna v bloku
    required string address = 2;            // adresa přístupu
    required string value = 3;              // čtená/zapisovaná hodnota
}

message Warp {
    repeated MemoryAccess accesses = 1;     // přístupy
    required Dim3 blockIdx = 2;             // pozice warpu v mřížce
    required int32 warpId = 3;              // id warpu
    required int32 debugId = 4;             // index ladícího údaje
    required int32 size = 5;                // počet čtených/zapisovaných bytů
    required int32 kind = 6;                // typ přístupu (čtení/zápis)
    required int32 space = 7;               // adresní prostor
    required int32 typeIndex = 8;           // index datového typu
    required string timestamp = 9;          // časové razítko
}
```

Výpis 6: Protobuf schéma warpu

Formát vygenerovaných souborů Soubory s paměťovými přístupy se vytváří v textovém formátu JSON. Ten je jednoduše zpracovatelný ručně i strojově a tvoří dnes standard pro přenos dat v mnoha odvětvích. Vygenerované JSON soubory nejsou odsazené z důvodu zmenšení velikosti souboru. Volitelně je lze naformátovat pro lepší čitelnost (např. pro ladění generovaných dat). Kernely mohou generovat velké množství přístupů a serializace v textovém formátu je v tomto případě neefektivní. Proto je nabízena i alternativa ve formě binární serializace. Otestováno bylo několik knihoven pro serializaci (Protocol Buffers[10], Cap'n Proto[11] a Flatbuffers[12]). Z nich byly vybrány knihovny Protocol Buffers (Protobuf) a Cap'n Proto. Obě knihovny umožňují serializaci dat v pevně stanoveném binárním formátu. Díky tomu jsou soubory vygenerované těmito knihovnami několikrát menší než v případě formátu JSON. Protocol Buffers byl zvolen, protože se jedná o běžně využívaný serializační formát a má dobrou podporu mezi programovacími jazyky. Cap'n Proto není tolik používaný, ale je několikrát rychlejší než Protobuf díky tomu, že používá stejné datové struktury pro reprezentaci dat v paměti i v přenosovém formátu. Obě knihovny jsou nepovinné, profilovací nástroj tedy lze přeložit i bez nich (v tom případě lze použít pouze JSON). Vizualizační komponenta podporuje všechny tři formáty. Definici datové struktury warpu ve formátu Protobuf si lze prohlédnout ve výpisu 6. Některé hodnoty (jako například časové razítko nebo adresa paměti) jsou z důvodu kompatibility s formátem JSON uloženy jako řetězec, JSON (potažmo ani Javascript) totiž nepodporuje 64bitová celá čísla.

Komprese dat Pokud jsou vygenerované soubory moc velké, lze použít kompresi s kódováním *gzip*. Ke kompresi je využita knihovna zlib[13], která je opět nepovinná pro překlad nástroje. Kompresi lze libovolně kombinovat se všemi výstupními formáty.

Paralelizace formátování Formátování a serializace velkého počtu paměťových přístupů je náročná, proto je tato činnost v knihovně pro sběr dat na CPU paralelizována. Vnitřně se vytváří *threadpool* (skupina vláken), která zpracovává žádosti o formátování přístupů. Pokud by instrumentovaná aplikace sama využívala více vláken a paralelní formátování by ji zpomalovalo nebo by paralelní zpracování vyžadovalo velké množství paměti, tak lze tuto funkcionalitu vypnout.

[10] `https://developers.google.com/protocol-buffers`
[11] `https://capnproto.org/`
[12] `https://google.github.io/flatbuffers/`
[13] `https://www.zlib.net`

5 Vizualizační aplikace

Naměření dat o běhu aplikace je prvním nutným krokem při profilování, tato data je však také potřeba zobrazit uživateli v intuitivní podobě. Počet paměťových přístupů provedených programy může běžně dosahovat řádu milionů či miliard. Takto velké množství dat je potřeba zpracovat, umožnit v něm vyhledávat a filtrovat a poskytnout na něj různé typy pohledů. Z tohoto důvodu byla v rámci této práce kromě profilovacího nástroje vytvořena také vizualizační aplikace.

5.1 Specifikace požadavků

Aplikace by měla umět načítat data ve všech formátech, které profilovací nástroj generuje. Aby se dalo v naměřených datech snadněji orientovat, měla by umožňovat operace jako je vyhledávání, filtrování a shlukování. Na data by mělo jít nahlížet více pohledy, hlavní je ale zobrazit adresní prostor aplikace a umožnit spojit jednotlivé přístupy vláken s adresou v paměti. Pomocí této vizualizace by měly jít sledovat vzory přístupů vláken do paměti. Stejně jako u profilovacího nástroje by měla být vizualizační aplikace jednoduše použitelná.

5.2 Architektura

Údaje získané instrumentací jsou vizualizovány pomocí webové aplikace. Tato forma prezentace byla zvolena, protože je nezávislá na operačním systému, nabízí jednoduché rozhraní pro vykreslování vektorové i rastrové grafiky a usnadňuje rychlé prototypování. Aplikace je zároveň pojata jako tzv. Single-page application, kde veškerá logika a směrování probíhá na straně klienta v prohlížeči. Díky tomu lze web sestavit do statických HTML souborů. Poté k jeho lokálnímu spuštění ani není potřeba spouštět webový server, stačí aplikaci otevřít v internetovém prohlížeči. Webové aplikace sice zatím nedosahují stejného výkonu jako nativní aplikace, nicméně pro potřeby vizualizace v této práci je jejich výkon dostačující.

Aplikace je psána v jazyce TypeScript [34]. Jedná se o nástavbu jazyka JavaScript, který je jediným běžně podporovaným jazykem, který lze využít přímo v prohlížeči na straně klienta. TypeScript přidává k JavaScriptu statický typový systém, což snižuje riziko určitých chyb v kódu a zároveň slouží k jeho dokumentaci. Podporuje také syntaxi nejnovějších ECMAScript standardů[14], které usnadňují vývoj. Překladač TypeScriptu generuje čistý JavaScriptový kód, který je kompatibilní i se staršími verzemi prohlížečů, jeho použití tedy nijak nesnižuje univerzálnost výsledné aplikace. Nevýhodou TypeScriptu je problematická spolupráce s knihovnami, které neobsahují typové definice. V posledních letech však už většina nejčastěji používaných knihoven typové definice nabízí a jejich použití v TypeScriptu je tedy snadné.

Uživatelské rozhraní aplikace je vytvořeno pomocí knihovny React. Ta umožňuje deklarativně vytvářet komplexní hierarchii uživatelských komponent. Úlohou komponenty je přijmout

[14]ECMAScript je standard pro skriptovací jazyky, JavaScript je jednou z jeho implementací

vlastnosti (*props*) a na základě nich vykreslit uživatelské rozhraní pomocí metody `render`. Z té lze vrátit deklarativní popis vzhledu ve formě minijazyka JSX[15], který kombinuje JavaScript se syntaxí inspirovanou XML. To vede k přehlednějšímu zápisu vzhledu komponenty než při použití textových šablon, které jsou běžné v jiných knihovnách. Zároveň lze při vytváření vzhledu komponenty využívat veškerých konstrukcí programovacího jazyka, což v šablonách nejde a dle autorova názoru je to jedno z velkých omezení textových šablon.

React komponenty fungují na jednoduchém principu jednosměrného toku dat. Data se předávají z rodičů potomkům a události naopak proudí od potomků k rodičům. Pokud chce rodič reagovat na událost vyvolanou v potomkovi, tak mu předá v *props* funkci, kterou potomek při dané události zavolá. Díky této filozofii je jednoduché pochopit, jak proudí data v aplikaci. Každá komponenta má kromě vlastností i vnitřní stav, ve kterém si může uchovávat data a předávat je potomkům. Pokud komponenta nemá vnitřní stav a její grafická reprezentace závisí čistě na vstupních vlastnostech, je nazývána čistou komponentou a lze ji reprezentovat jednou funkcí (jinak se komponenty obvykle definují jako třídy).

React pro efektivní vykreslování komponent používá techniku zvanou Virtual DOM. Při každé změně stavu komponenty nebo při vyvolání uživatelské události v aplikaci dojde ke překreslení celého stromu komponent. Každá komponenta může překreslení vynechat (například pokud se nezměnily její vstupní vlastnosti). Pokud k překreslení komponenty dojde, tak jsou JSX výrazy vytvořené její metodou render převedeny na strom JavaScriptových objektů. React porovná tento strom se stromem vytvořeným při minulém překreslení a vygeneruje minimální množinu změn, kterou je nutné provést na výstupním grafickém zařízení pro vykreslení nového stavu. Tímto zařízením může být například DOM[16] v internetovém prohlížeči nebo strom komponent v nativní mobilní aplikaci. Pokud jedna komponenta například změní jeden atribut v HTML tagu své reprezentace, tak React sice provede překreslení celé komponenty, ale pro provedení změny pouze zavolá jednu funkci, která změní v DOM prohlížeče daný atribut. Díky této metodě je React velmi efektivní a zvládne vykreslit tisícovky komponent na stránce. Tato technika je umožněna díky tomu, že vzhled komponent je definován deklarativně, metoda render tedy při každém zavolání vrátí kompletní reprezentaci vzhledu komponenty. Tím se React liší od většiny ostatních knihoven, kde se běžně prvky uživatelského rozhraní vytvoří pouze jednou a pak se v závislosti na událostech mění jejich atributy. Nejnovější verze Reactu experimentují s vlastní implementací uživatelských vláken využívajících kooperativního multitaskingu. Díky tomu lze překreslovat strom komponent asynchronně a prioritizovat důležité události.

Ukázku jednoduché React komponenty napsané v jazyce TypeScript si lze prohlédnout ve výpisu 7. Pomocí typového systému má tato komponenta jasně definované rozhraní vlastností, které přijímá, a rozhraní svého vnitřního stavu. Komponenta obsahuje tlačítko, které po stisknutí inkrementuje vnitřní čítač a vypisuje jeho současný stav.

[15]Pokud je JSX použit s jazykem TypeScript, tak je označován názvem TSX
[16]Strom objektů v prohlížeči

```
interface Props
{
    increment: number;
}
interface State
{
    counter: number;
}

class Counter extends Component<Props, Readonly<State>> {
    readonly state: State = {
        counter: 0
    };

    render() {
        return (
            <Button
                onClick={this.increaseCounter}
                text={this.state.counter.toString()} />
        );
    }
    increaseCounter = () => {
        this.setState(state => ({
            counter: state.counter + this.props.increment
        }));
    }
}
```

Výpis 7: Ukázka React komponenty

Pro správu stavu aplikace byla zvolena architektura Flux (konkrétně její implementace pomocí knihovny Redux). Flux byl navrhnut firmou Facebook jako řešení pro neustále narůstající komplexitu webových aplikací. Jeho základní myšlenkou je reprezentovat veškerý stav aplikace v jednom objektu, který se nazývá *store*. Veškeré změny tohoto stavu musí proběhnout pouze vyvoláním předem definovaných akcí. Stav se nesmí měnit přímo, lze ho upravit pouze tak, že se nahradí novým objektem, který může mít některé atributy upravené. Díky tomu lze rychle poznat, které atributy se změnily, protože stačí porovnat adresu objektů a ne jejich obsah (tzv. shallow compare). Tento princip pochází z funkcionálního programování, kde obvykle nelze měnit datové struktury přímo, ale lze je pouze nahradit novou verzí (jsou tzv. immutable). Tato architektura je velmi odlišná od objektově orientovaného paradigma, kde je stav aplikace obvykle distribuován mezi velké množství objektů, jejichž stav se může kdykoliv změnit. To někdy způsobuje velmi obtížně nalezitelné chyby a může způsobit nekonzistentní stav aplikace.

Redux nabízí kompromis. Při jeho použití je nutné striktně dodržovat pravidla neměnného stavu aplikace, což může vyžadovat více kódu než při vytvoření stejné funkcionality pomocí OOP. Pokud ale jsou jeho pravidla dodržena, tak lze pro každou změnu stavu aplikace vysledovat, která akce ji vyvolala. To usnadňuje ladění programu, umožňuje cestování historií stavů programu (tzv. time-travel debugging) a také usnadňuje reprodukci uživatelských chyb. Pokud se zaznamená sekvence akcí uživatele, tak přehráním této sekvence vývojář může replikovat stav, do kterého se daný uživatel dostal.

Vytváření nového stavu v reakci na akci je v Reduxu prováděno pomocí funkce nazvané *reducer*. Stav aplikace je uložen ve slovníku, kde každý klíč odpovídá jednomu reduceru (lze je i vnořovat). Když se v aplikaci vyvolá akce, tak každý reducer v závislosti na ni buď vrátí původní stav odpovídající jeho klíči, nebo vytvoří stav nový. Kdyby aplikace uměla pouze vyvolávat akce a měnit svůj stav, tak by v mnoha případech nebyla moc užitečná. V aplikacích je obvykle žádoucí provádět funkce, které mají tzv. vedlejší efekt. Ty se projeví tak, že nemění stav aplikace, ale typicky komunikují s externím rozhraním. Může jít například o zápis do souboru nebo o vyslání síťového požadavku. Pro vytváření vedlejších efektů lze v Reduxu použít několik přístupů. Ten asi nejjednodušší je vytvořit funkci, která provede vedlejší efekty a v závislosti na nich vyvolá požadované akce. Například při provedení síťového požadavku by se nejprve vygenerovala akce značící, že požadavek byl vyslán. Do stavu aplikace by se tak uložila informace o tom, že požadavek byl vytvořen a při překreslení by se například mohla objevit informace, že probíhá načítaní dat. Jakmile se požadavek dokončí, tak se opět vyvolá akce značící buď úspěšné nebo neúspěšné dokončení požadavku. Aby se toto obstarávání akcí nemuselo provádět manuálně, tak byla pro vývoj aplikace použita knihovna `redux-observable`, která umožňuje reagovat na proud akcí v aplikaci a jednoduše na ně reagovat a případně vytvářet nové akce. Tato knihovna je založena na reaktivním programovacím paradigmatu (implementovaném knihovnou `RxJs`). To umožňuje snadnou práci s asynchronními událostmi a proudy dat, což se hodí právě pro vytváření vedlejších efektů.

Redux je samostatná knihovna, ale velmi často je používána jako datové úložiště pro knihovnu

React. Při použití Reduxu je důležitý stav aplikace uložen v něm, a při každé změně tohoto stavu dojde k překreslení stromu React komponent. Ve vnitřním stavu jednotlivých komponent by v tomto případě neměly být uloženy doménové informace o stavu aplikace, ale pouze přechodné informace, například vizuální stav komponenty. Toto pravidlo je nicméně někdy nutné porušit z důvodu optimalizace vykreslování.

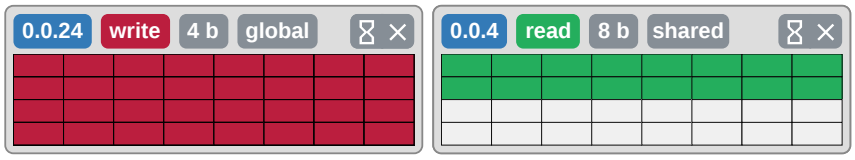
JavaScript obsahuje pouze jeden datový typ pro čísla. Vnitřně je reprezentovaný buď jako 32bitové celé číslo anebo jako 64bitové číslo s plovoucí řadovou čárkou ve formátu IEEE 754. Kvůli toho nelze v JavaScriptu přesně reprezentovat celá čísla mimo rozsah $-(2^{53}-1)$ až $2^{53}-1$. To je problémové pro vizualizaci paměťových přístupů, jejichž adresy jsou dnes již obvykle tvořené 64bitovými čísly. Z tohoto důvodu jsou pro reprezentaci adres v aplikaci použita čísla s neomezenou přesností (pomocí knihovny `big-integer`).

5.3 Rozhraní aplikace

Prvním krokem, který je potřeba v aplikaci udělat, je načíst soubory vygenerované profilovacím nástrojem. Ty se vytvářejí při každém spuštění profilované aplikace do nové složky, stačí tak obvykle tuto složku otevřít a načíst všechny soubory v ní. Pokud ale profilovaný program není spuštěn ze stejné složky, ve které byl přeložen, tak ve vygenerované složce mohou chybět metadata s ladícími údaji kernelů. V tom případě je nutné je dohledat a načíst do aplikace, jinak nebude možné naměřená data vizualizovat. Soubory s paměťovými přístupy mohou být velké, proto jejich načítání v aplikaci probíhá pomocí tzv. Web workerů. Jedná se o rozhraní v prohlížečích, které dovoluje spouštět JavaScriptový kód izolovaně od hlavního okna aplikace, tento výpočet se poté spouští v jiném vlákně. Díky tomu lze načíst i velké soubory do aplikace bez toho, aby se zablokovalo uživatelské rozhraní. Do aplikace lze nahrávat soubory vygenerované ve formátu JSON, Protobuf i Cap'n Proto, a to i ve zkomprimované verzi. Při načtení souborů se provede základní validování jejich obsahu.

Jakmile jsou soubory nahrány, tak se zobrazí časová osa obsahující jednotlivá spuštění kernelů. Z těch je nutné si zvolit kernel, jehož přístupy budou vizualizovány. V rámci kernelu jsou zobrazeny jednotlivé paměťové přístupy warpů. Těch může být velmi velké množství, proto je nutné z nich vyfiltrovat pouze ty warpy, které uživatele zajímají. Filtrovat lze buď podle pozice bloku warpu, podle pozice warpu ve zdrojovém kódu nebo podle typu přístupu. Ukázku zvolených warpů si lze prohlédnout na obrázku 5. Na obrázku jsou dva warpy, každý s 32 vlákny. První warp má pozici ve výpočetní mřížce 0.0.24 a provádí zápis 4 bytů do globální paměti. Jednotlivá vlákna warpu jsou reprezentována obdélníky v mřížce. Zápis provádějí ta vlákna, která jsou obarvená, v tomto případě všechna. Druhý warp je na pozici 0.0.4 a čte 8 bytů ze sdílené paměti. U tohoto warpu byla aktivní (a tedy prováděla čtení) pouze polovina vláken. Pro každý warp se dá pomocí tlačítka vybrat všechny přístupy stejného warpu, které v daném kernelu proběhly.

Jakmile si uživatel zvolí požadované warpy, tak si pro ně může vybrat jeden ze tří různých pohledů. Prvním pohledem je tabulka paměťových konfliktů. Pokud zvolené warpy přistoupily ke



Obrázek 5: Přístupy vláken ve warpu

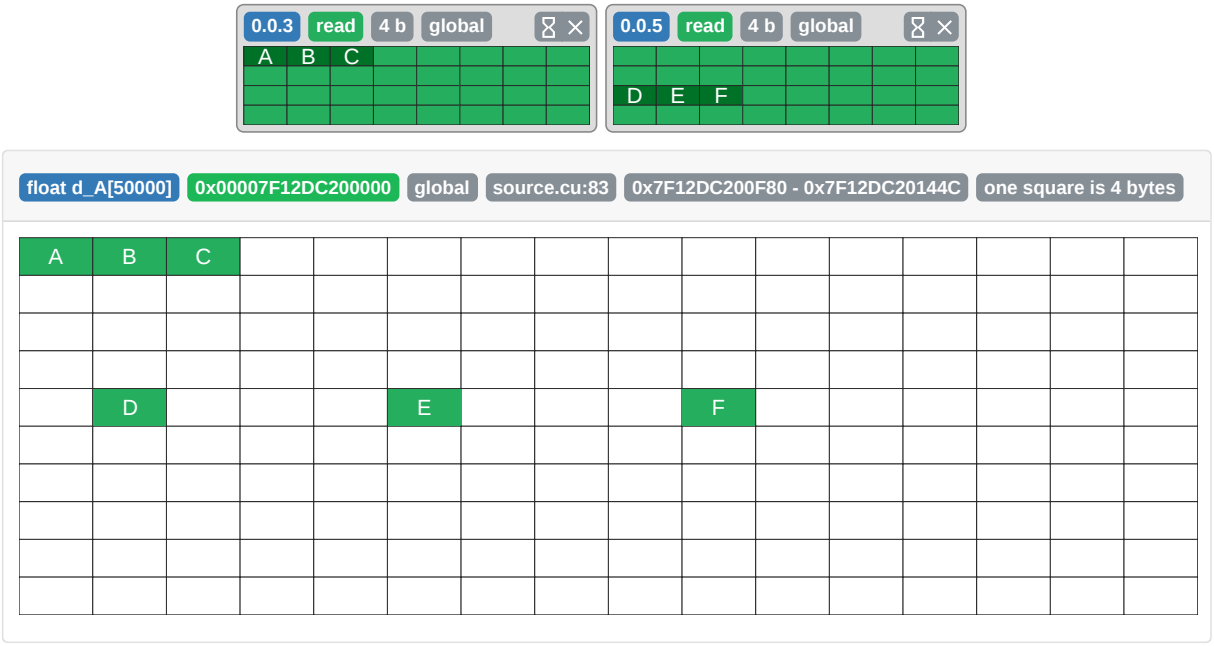
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	20	21	22	23	24	25	26	27	28	29	30	31
0.0.0	0.0.1	0.0.2	0.0.3	0.0.4	0.0.5	0.0.6	0.0.7	0.0.8	0.0.9	0.0.10	0.0.11	0.0.12	0.0.13	0.0.14	0.0.15	0.0.20	0.0.21	0.0.22	0.0.23	0.0.24	0.0.25	0.0.26	0.0.27	0.0.28	0.0.29	0.0.30	0.0.31
0.0.16	0.0.17	0.0.18	0.0.19																								

Obrázek 6: Konflikt při přístupu do sdílené paměti

stejnému úseku paměti, tak se tyto přístupy vypíšou do tabulky. Pro každý přístup je zobrazena rozsah adres, ve kterých ke konfliktu došlo, a souřadnice vláken, které na danou adresu přistupovaly. Údaje o kolizích v paměti jsou vypočteny tak, že se vytvoří mapa paměti na úrovni bytů. Pro každou paměťovou operaci zvolených warpů se do mapy zapíše, ke kterým bytům paměti přistupovala. Na konci jsou pak všechny ovlivněné byty zkontrolovány a pokud k nějakému bytu přistoupil více než jeden warp, tak je tato skutečnost nahlášena jako konflikt. Vzhledem k tomu, že nelze získat přesné informace o časování přístupu, tak je na uživatelovi, aby tato data správně interpretoval a vyhodnotil, zda konflikt mohl opravdu nastat.

Další poskytovanou vizualizací je tabulka konfliktů pro moduly sdílené paměti. Tento pohled má smysl pouze pokud je vybraný právě jeden warp, protože konflikty při přístupu do sdílené paměti mohou nastat pouze v kontextu určitého warpu. Pro každý modul je v tabulce zobrazen jeden sloupec (modulů je 32, jeden pro každé vlákno ve warpu). V daném sloupci se zobrazují všechny přístupy do daného modulu v rámci zvoleného warpu. Pokud tak je v nějakém sloupci více než jeden řádek, tak to označuje konflikt při přístupu k modulu sdílené paměti. Paměťový modul použitý pro přístup je dán adresou, ke které se přistupuje. Přiřazení přístupu k modulu je tak dáno jednoduchým výpočtem. Moduly mají velikost 4 byty, takže se adresa paměti nejprve vydělí hodnotou 4, aby se získal virtuální index modulu. Tento index se poté vydělí hodnotou 32 a zbytek po tomto dělení udává modul použitý pro daný přístup do sdílené paměti. Velikost modulu může být u novějších CUDA architektur i 8 bytů. Skutečná velikost modulu je před každým spuštěním kernelu zaznamenána a vizualizační aplikace při zjišťování konfliktů sdílené paměti tuto hodnotu respektuje. Ukázka konfliktu v modulech sdílené paměti je na obrázku 6. První čtyři sloupce jsou označeny červeně, protože u těchto modulů došlo ke konfliktu. Do nultého modulu přistoupila vlákna na pozici 0.0.0 a 0.0.16, do prvního vlákna na pozici 0.0.1 a 0.0.17 atd.

Třetím pohledem na naměřená data je zobrazení mapy paměťového prostoru aplikace. Pro zvolené warpy se zobrazí všechny alokace na grafické kartě, které byly v době vykonávání daných warpů aktivní. U alokací je zobrazena deklarace proměnné v jazyce C , do které byla alokace uložena, adresa a adresní prostor a umístění ve zdrojovém kódu. Po najetí myši na vlákno warpu se zobrazí blok paměti, ke kterému vlákno přistupovalo. Lze si zobrazit i zpětné mapování, po



Obrázek 7: Zobrazení paměťových přístupů v adresním prostoru

najetí myši na blok paměti se zvýrazní vlákno, které k němu přistupovalo. Na vlákno lze kliknout a trvale zvýraznit jeho paměťový přístup. Zvýrazněné přístupy jsou pojmenované písmenem abecedy. Takto lze procházet jednotlivá vlákna a sledovat, jak přistupují k paměti, jestli vlákna blízko u sebe přistupují k paměti sekvenčně nebo jestli jsou mezi jejich přístupy mezery. Ukázku paměťové alokace a zvýrazněných přístupů si lze prohlédnout na obrázku 7. Na obrázku jsou zobrazeny dva warpy a některé paměťové buňky pole s padesáti tisíci čísly, do kterého warpy přistupovaly. Jedna buňka na obrázku reprezentuje 4 byty paměti. Vlákna prvního warpu přistupovala k paměti sekvenčně – sousedící vlákna četla sousedící buňky paměti. Naproti tomu u druhého warpu docházelo k tzv. *strided* přístupům, protože buňky paměti čtené sousedícími vlákny mezi sebou měly 16 bytovou mezeru. Tato situace může nastat například pokud vlákna čtou pouze jeden atribut ze struktury uložené v poli struktur. V tomto případě by daná struktura měla velikost 24 bytů. Tento typ přístupu, který může být neefektivní, lze pomocí vizualizace paměťového prostoru snadno odhalit.

6 Testování dosažených výsledků

Instrumentační nástroj zaznamenává veškeré dostupné přístupy do paměti na grafické kartě a pro jeho použití stačí přidat několik přepínačů překladači a vložit hlavičkový soubor do zdrojového kódu. Tímto nástroj splňuje dva hlavní požadavky, které na něho byly kladeny. Pro praktické použití nástroje je ovšem důležitý i jeho vliv na dobu výpočtu programu a velikost souborů, které generuje. Obě vlastnosti by mohly způsobovat problémy, pokud by instrumentovaný program běžel moc dlouho nebo pokud by generoval nepřiměřené množství dat. V této kapitole je nejprve otestován vliv instrumentace na výkon programu a poté je otestována vizualizační aplikace.

6.1 Vliv instrumentace na výkon

Vliv instrumentace na dobu překladu a běhu aplikace byl otestován na vzorových programech, které jsou dodávány s instalací programovacího balíčku CUDA. U jednotlivých programů byla naměřena doba výpočtu a velikost vygenerovaných souborů s paměťovými přístupy, a to pro všechny kombinace výstupních formátů (JSON, Cap'n Proto, Protobuf a jejich komprimované verze). Programy byly testovány s optimalizacemi a zapnutým generováním ladících údajů, které jsou nezbytné pro interpretaci naměřených dat. Každé měření bylo opakováno desetkrát, výsledný čas je dán průměrem z těchto opakovaných měření. Experimenty byly provedeny na počítači s procesorem Intel Core i7–7700HQ (2,8 GHz) a grafickou kartou Nvidia GeForce GTX 1050Ti. Formátování a serializace paměťových přístupů byla paralelizována na 8 vláknech. Programy byly přeloženy překladačem Clang 4.0. Kód, který prováděl měření, a měřené programy jsou k dispozici v repozitáři projektu. Testovány byly následující programy.

vectorAdd sečte dva vstupní vektory a uloží výsledek do třetího vektoru. Testovaný program sčítal vektory s 50 tisíci čísly. Tento program byl zvolen jako jednoduchá ukázka sekvenčního přístupu vláken k paměti.

simpleGL generuje rastrové obrázky, ve kterých se pohybuje vlna. Výpočet se provádí na obrázku o rozměrech 256x256 pixelů. Při testování nebyl měřen čas vykreslování obrázku, ale pouze jeho vytvoření kernelem na grafické kartě. Za běhu tohoto programu je kernel generující obrázek spuštěn celkem padesátkrát. Tento program byl zvolen jako ukázka zápisu paměti do dvojrozměrného bufferu.

mandelbrot iterativně počítá Mandelbrotovu množinu. Celkem se provede 10 iterací výpočtu této množiny. Tento program často přistupuje k paměti a generuje velké množství přístupů. Byl tak zvolen pro otestování zátěže formátování výstupu.

reduction počítá součet vektoru čísel pomocí paralelní redukce. V testované konfiguraci se sčítá 65 tisíc 64bitových čísel s plovoucí řadovou čárkou. Tento program byl zvolen, aby otestoval schopnost nástroje zvládnout spuštění velkého množství kernelů a zaznamenání přístupů do sdílené paměti.

Tabulka 1: Doba překladu instrumentovaných aplikací

Program	**Bez instrumentace** [s]	**S instrumentací** [s]
vectorAdd	1.35	1.83
simpleGL	2.47	2.91
mandelbrot	4.12	4.81
reduction	4.13	9.5

Tabulka 2: Doba výpočtu instrumentovaných aplikací

Program	**BEZ** [s]	**JS** [s]	**JSK** [s]	**PR** [s]	**PRK** [s]	**CP** [s]	**CPK** [s]
vectorAdd	0.02	0.36	0.46	0.36	0.46	0.26	0.36
simpleGL	0.003	4.1	6.7	4.6	6.1	1.9	3.6
mandelbrot	0.1	16.7	31.7	19.7[a]	28.1[a]	6.5	13.2
reduction	0.12	43	57	21.4	25.9	12.5	16

[a]Formátování proběhlo na 4 vláknech kvůli nedostatku operační paměti

V tabulce 1 je zaznamenána doba překladu programů bez použití instrumentace a s instrumentací. Instrumentace měla měřitelný, avšak u většiny testovaných programů zanedbatelný vliv na dobu překladu. Nejvíce byl ovlivněn program `reduction`, který obsahuje šest kernelů, z nichž některé jsou šablonované. Dohromady všechny jejich kombinace vytváří celkem 132 různých kernelů, které musí projít instrumentací, což se projevilo na času překladu.

V tabulce 2 jsou naměřené časy výpočtu instrumentovaných programů. Jako reference pro srovnání slouží doba výpočtu bez použití instrumentace (BEZ). Dále byly měřeny konfigurace, které zapisovaly data ve formátech JSON (JS), komprimovaný JSON (JSK), Protobuf (PR), komprimovaný Protobuf (PRK), Cap'n Proto (CP) a komprimovaný Cap'n Proto (CPK).

I přes atomické zápisy přístupů do bufferu na grafické karty běžely instrumentované kernely relativně rychle, zpomalení bylo zhruba 1x-10x vzhledem k původnímu programu. Nejvíce času instrumentované aplikace trávily zpracováním naměřených dat a jejich serializací. Naměřená data by šlo rychle zapsat na disk v binární podobě, potom by ale bylo složité je zpracovat, z toho důvodu probíhá předzpracování do běžného serializačního formátu už v rámci běhu programu. Z naměřených dat je zřejmé, že čas provádění je silně závislý na použitém formátu dat. Paměťové přístupy můžou generovat až stovky milionů záznamů a naformátovat takto velké množství dat je výpočetně náročné. Dalším úzkým hrdlem je kopírování paměťových přístupů z GPU na CPU.

Tabulka 3 obsahuje velikosti souborů s paměťovými přístupy pro jednotlivé konfigurace. Hodnota v tabulce udává průměrnou velikost souboru s paměťovými přístupy vygenerovaného pro jedno spuštění kernelu. Velikosti jsou udány v jednotkách mebibytů. V posledním sloupci je celkový počet paměťových přístupů provedených daným programem a počet kernelů vykonaných programem. Nejmenší soubory generuje Protobuf a Cap'n Proto, protože se jedná o binární

Tabulka 3: Velikost souborů vygenerovaných instrumentací

Program	JS	JSK	PR	PRK	CP	CPK	počet přístupů/spuštění
vectorAdd	14.3	0.8	7.4	0.6	8.5	0.6	150000/1
simpleGL	24.6	1	12.8	0.9	15.1	1	262144/50
mandelbrot	372.6	16	192.4	12.4	227.5	14.4	4036113/10
reduction	11.9	0.38	5.8	0.27	6.7	0.39	33301468/301

formáty. Pokud se ale použije komprese, tak jsou velikosti výsledných souborů srovnatelné mezi jednotlivými formáty. Komprese ovšem zhruba dvakrát prodlužuje čas serializace.

Serializace do formátu JSON trvá srovnatelně dlouho jako serializace do formátu Protobuf, ale generuje znatelně větší soubory. Při použití komprese u ní lze zmenšit velikost vygenerovaných souborů až řádově, za cenu zpomalení serializace. Formát JSON je vhodný pouze pro malé kernely. Nejrychleji probíhá serializace do formátu Cap'n Proto.

Instrumentované programy dle očekávání běží několikrát pomaleji než původní programy. Nicméně pro sledování vzorů přístupů do paměti stačí instrumentovat daný program na malém, testovacím vstupu, který půjde i s instrumentací vyřešit v rozumném čase. Po optimalizaci přístupů na daném vstupu pak lze otestovat, jak změna pomohla na větších vstupech, už bez použití instrumentace. Dle názoru autora práce tak toto zpomalení netvoří limitující faktor pro použití instrumentačního nástroje.

6.2 Vizualizační aplikace

Vizualizační aplikace umožňuje načítat soubory s paměťovými přístupy ve všech podporovaných formátech. Přístupy jsou vizualizovány ve formě warpů, paměťové mapy adresního prostoru a tabulky konfliktů paměťových modulů. Pomocí zobrazení warpů a alokací lze intuitivně sledovat vzory přístupů jednotlivých vláken do paměti, což bylo hlavním účelem této vizualizační aplikace. Její nevýhodou je, že naměřené údaje je nutné správně interpretovat. Uživatel musí rozumět, jak jsou vlákna uspořádány do warpů a jak provádí paměťové přístupy, aby mohl tuto vizualizaci využít k optimalizaci kernelu.

U vizualizační aplikace byl měřen čas načtení souborů s paměťovými přístupy. Konfigurace formátů jsou totožné jako v předchozí sekci. Načtení se skládalo z přečtení souboru z disku (testování probíhalo na lokálním serveru, takže zde neprobíhaly síťové požadavky), případné dekomprimace a převodu do Javascriptových objektů. Naměřené hodnoty si lze prohlédnout v tabulce 4. Nejrychleji se ve většině případů načítaly soubory ve formátu JSON, protože se jedná o reprezentaci Javascriptových objektů, kterou prohlížeče podporují nativně. Jeho soubory jsou však velmi velké, a kvůli toho bylo u programu `mandelbrot` rychlejší načtení ve formátu Protobuf. Cap'n Proto nemá dobrou podporu pro Javascript, jeho Javascriptová implementace dokázala v prohlížeči načíst pouze soubory z programu `vectorAdd`. Načítání je zpomalené kvůli

Tabulka 4: Doba načtení souborů s přístupy ve vizualizační aplikaci

Program	**JS** [s]	**JSK** [s]	**PR** [s]	**PRK** [s]	**CP** [s]	**CPK** [s]
vectorAdd	0.5	0.8	0.7	0.9	3.3	3.5
simpleGL	0.9	1.8	1.2	1.4	-	-
mandelbrot	23	20	17	20	-	-
reduction	1.5	2.1	1.9	2.6	-	-

serializaci posílaných dat do Web workeru a zpátky z něj, nicméně použití Web workeru je nutné pro zachování responzivity aplikace.

7 Závěr

V této práci byly popsány existující nástroje pro ladění, profilování a instrumentaci CUDA programů, obecné principy profilování a technologie CUDA a LLVM. Pro účely profilování aplikací na grafické kartě byl vytvořen instrumentační nástroj, který pomocí automatizovaného vkládání instrukcí do CUDA programů umožňuje zaznamenávat za běhu programu paměťové operace na grafické kartě. Zaznamenané údaje obsahují detailní informace o vláknu provádějícím přístup i o samotném přístupu. Kromě toho také nástroj zaznamenává paměťové alokace na grafické kartě. Veškeré zaznamenané údaje jsou poté uloženy do souborů ve formátech JSON, Protobuf nebo Cap'n Proto. Otestováním nástroje bylo zjištěno, že sice zpomaluje dobu výpočtu, ale ne natolik, aby to omezovalo jeho použití. Nástroj je jednoduché použít k instrumentaci existujícího kódu, stačí přidat několik přepínačů překladači a do zdrojového souboru s kernelem vložit jeden hlavičkový soubor. Soubory generované instrumentovanými programy mají jasně popsaný formát a jsou jednoduše strojově zpracovatelné. Lze je tak použít pro vytvoření nových nástrojů určených pro vizualizaci profilování CUDA programů.

Vytvořený profilovací nástroj tak splňuje požadavky, které na něj byly kladeny. Zároveň dokazuje, že pomocí překladače Clang lze jednoduše instrumentovat programy na grafických kartách. Nástroj byl během svého vývoje upraven pro podporu dvou nových verzí překladače Clang (5 a 6) a tyto úpravy nevyžadovaly velké změny v kódu. V průběhu tvorby této práce vyšly také nové verze platformy CUDA. Verze 9 je už podporována v LLVM, verze 9.1 bude podporována budoucími verzemi Clangu a bude ji tedy podporovat i vytvořený profilovací nástroj. Díky použití frameworku LLVM je jednodušší psát udržitelné instrumentační nástroje. Právě špatná udržitelnost způsobená uzavřeností platformy CUDA způsobuje rychlé zastarávání ostatních nástrojů pro instrumentaci CUDA kódu.

Dále byla v práci vytvořena webová aplikace, která využívá soubory vygenerované instrumentovaným programem k vizualizaci paměťových přístupů. Pomocí této aplikace lze zobrazit a vyhledávat jednotlivé warpy a vlákna, které prováděly paměťové přístupy. Pro zvolené warpy pak lze zobrazit paměťové konflikty přístupů k modulům sdílené paměti a přístupům ke globální paměti. Dále lze v aplikaci pro každý paměťový přístup zobrazit jeho umístění v adresním prostoru aplikace a sledovat tak vzory přístupů sousedních vláken ve warpu. Tento pohled na paměťové přístupy usnadňuje intuitivní představu o tom, jak vlákna pracují s pamětí a může tak programátorovi pomoct tyto přístupy optimalizovat.

Instrumentační nástroj obsahuje funkční základ pro automatizovanou modifikaci CUDA programů a je snadné jej rozšířit o nové funkce. Lze jej tak použít jako vstupní bod pro vytváření nových instrumentačních nástrojů pro aplikace grafických karet. Nástroj by šel rozvíjet do šířky, například přidáním podpory pro více CUDA alokačních funkcí nebo instrumentací konkrétních PTX instrukcí, které by poté šlo vizualizovat. Alternativou k současnému způsobu instrumentace by mohla být instrumentace zdrojového kódu, která by odstranila závislost nástroje na překladači Clang. Ta by však nemohla zachytit detaily o nízkoúrovňových detailech programu.

Nástroj by mohl být rozšířen o zaznamenávání paměťových přístupů CPU. To by vyžadovalo drobné změny formátu generovaných souborů a nový pohled na přístupy ve webové aplikaci. Zaznamenání přístupů na procesoru by šlo provést například pomocí již zmíněných nástrojů Pin [17] nebo Valgrind [16]. Nástroj by poté sloužil jako univerzání vizualizátor paměťových přístupů jak na procesoru, tak na grafické kartě. Vizualizační aplikace by mohla být rozšířena o podporu automatické detekce problémových situací, jako jsou například konflikty při přístupu do sdílené paměti nebo vzdálené přístupy sousedních vláken do paměti. Tato detekce by usnadnila hledání výkonnostních problémů, které teď musí uživatel vyhledávat manuálně.

Instrumentace může být velmi užitečným nástrojem pro získávání dat o běhu programu, a to i u grafických karet, kde instrumentace není moc rozšířená. Uzavřenost platformy CUDA bohužel komplikuje tvorbu nástrojů pro grafické karty od Nvidie. S rozvíjející se podporou CUDA programů v překladači Clang se to snad v budoucnu zlepší. Díky ní můžou vznikat nové profilovací nástroje, které usnadní optimalizaci paralelních programů pro grafické karty a které půjde snadno udržovat.

Literatura

1. GRAHAM, Susan L.; KESSLER, Peter B.; MCKUSICK, Marshall K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* 1982, roč. 17, č. 6, s. 120–126. ISSN 0362-1340. Dostupné z DOI: 10.1145/872726.806987.
2. *Massif: a heap profiler*. Dostupné také z: http://valgrind.org/docs/manual/ms-manual.html.
3. NETHERCOTE, Nicholas. *Dynamic Binary Analysis and Instrumentation*. United Kingdom, 2004. Disertační práce. Computer Laboratory, University of Cambridge.
4. *Intel® VTune™ Amplifier*. Dostupné také z: https://software.intel.com/en-us/intel-vtune-amplifier-xe.
5. LATTNER, Chris; ADVE, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. Palo Alto, California: IEEE Computer Society, 2004, s. 75–. CGO '04. ISBN 0-7695-2102-9. Dostupné také z: http://dl.acm.org/citation.cfm?id=977395.977673.
6. MCKEEMAN, W. M. Peephole Optimization. *Commun. ACM*. 1965, roč. 8, č. 7, s. 443–444. ISSN 0001-0782. Dostupné z DOI: 10.1145/364995.365000.
7. COOPER, Keith D.; SIMPSON, L. Taylor; VICK, Christopher A. Operator strength reduction. *ACM Trans. Program. Lang. Syst.* 2001, roč. 23, s. 603–625.
8. ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Global Value Numbers and Redundant Computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA: ACM, 1988, s. 12–27. POPL '88. ISBN 0-89791-252-7. Dostupné z DOI: 10.1145/73560.73562.
9. CYTRON, Ron; FERRANTE, Jeanne; ROSEN, Barry K.; WEGMAN, Mark N.; ZADECK, F. Kenneth. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 1991, roč. 13, č. 4, s. 451–490. ISSN 0164-0925. Dostupné z DOI: 10.1145/115372.115320.
10. KELSEY, Richard A. A correspondence between continuation passing style and static single assignment form. In: *ACM SIGPLAN Notices*. 1995, sv. 30, s. 13–22. Č. 3.
11. NICKOLLS, John; BUCK, Ian; GARLAND, Michael; SKADRON, Kevin. Scalable Parallel Programming with CUDA. *Queue*. 2008, roč. 6, č. 2, s. 40–53. ISSN 1542-7730. Dostupné z DOI: 10.1145/1365490.1365500.
12. *NVIDIA Cuda C Programming Guide*. Dostupné také z: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

13. GREGG, Brendan. *CPU Flame Graphs*. Dostupné také z: http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

14. *CodeXL*. Dostupné také z: https://github.com/GPUOpen-Tools/CodeXL.

15. NETHERCOTE, Nicholas; SEWARD, Julian. How to shadow every byte of memory used by a program. In: *Proceedings of the 3rd international conference on Virtual execution environments*. 2007, s. 65–74.

16. NETHERCOTE, Nicholas; SEWARD, Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 2007, roč. 42, č. 6, s. 89–100. ISSN 0362-1340. Dostupné z DOI: 10.1145/1273442.1250746.

17. LUK, Chi-Keung; COHN, Robert; MUTH, Robert; PATIL, Harish; KLAUSER, Artur; LOWNEY, Geoff; WALLACE, Steven; REDDI, Vijay Janapa; HAZELWOOD, Kim. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, IL, USA: ACM, 2005, s. 190–200. PLDI '05. ISBN 1-59593-056-6. Dostupné z DOI: 10.1145/1065010.1065034.

18. *NVIDIA® Nsight™*. Dostupné také z: https://www.nvidia.com/object/nsight.html.

19. SHENDE, Sameer S.; MALONY, Allen D. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 2006, roč. 20, č. 2, s. 287–311. ISSN 1094-3420. Dostupné z DOI: 10.1177/1094342006064482.

20. KNÜPFER, Andreas; BRUNST, Holger; DOLESCHAL, Jens; JURENZ, Matthias; LIEBER, Matthias; MICKLER, Holger; MÜLLER, Matthias S.; NAGEL, Wolfgang E. The Vampir Performance Analysis Tool-Set. In: RESCH, Michael; KELLER, Rainer; HIMMLER, Valentin; KRAMMER, Bettina; SCHULZ, Alexander (ed.). *Tools for High Performance Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 139–155. ISBN 978-3-540-68564-7.

21. BRUENING, Derek; GARNETT, Timothy; AMARASINGHE, Saman. An Infrastructure for Adaptive Dynamic Optimization. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. San Francisco, California, USA: IEEE Computer Society, 2003, s. 265–275. CGO '03. ISBN 0-7695-1913-X. Dostupné také z: http://dl.acm.org/citation.cfm?id=776261.776290.

22. STEPHENSON, M.; HARI, S. K. S.; LEE, Y.; EBRAHIMI, E.; JOHNSON, D. R.; NELLANS, D.; O'CONNOR, M.; KECKLER, S. W. Flexible software profiling of GPU architectures. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, s. 185–197. ISSN 1063-6897. Dostupné z DOI: 10.1145/2749469.2750375.

23. FAROOQUI, N.; KERR, A.; EISENHAUER, G.; SCHWAN, K.; YALAMANCHILI, S. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In: *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 2012, s. 58–67. Dostupné z DOI: 10.1109/ISPASS.2012.6189206.

24. KENNELLY, Chris. Dostupné také z: https://github.com/ckennelly/panoptes.

25. FAROOQUI, Naila; KERR, Andrew; DIAMOS, Gregory; YALAMANCHILI, S.; SCHWAN, K. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. Newport Beach, California: ACM, 2011, 9:1–9:9. GPGPU-4. ISBN 978-1-4503-0569-3. Dostupné z DOI: 10.1145/1964179.1964192.

26. PRINTEZIS, Tony; JONES, Richard E. GCspy: an adaptable heap visualisation framework. In: *OOPSLA*. 2002.

27. MORETA, S.; TELEA, A. Visualizing Dynamic Memory Allocations. In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2007, s. 31–38. Dostupné z DOI: 10.1109/VISSOF.2007.4290697.

28. *Tracectory*. Dostupné také z: https://bitbucket.org/oebeling/tracectory/wiki/Home.

29. *Memview*. Dostupné také z: https://github.com/ajclinto/memview.

30. CHOUDHURY, A. N. M. Imroz; POTTER, Kristin C.; PARKER, Steven G. Interactive Visualization for Memory Reference Traces. In: *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*. Eindhoven, The Netherlands: The Eurographs Association & John Wiley & Sons, Ltd., 2008, s. 815–822. EuroVis'08. Dostupné z DOI: 10.1111/j.1467-8659.2008.01212.x.

31. WEYERS, Benjamin; TERBOVEN, Christian; SCHMIDL, Dirk; HERBER, Joachim; KUHLEN, Torsten W.; MÜLLER, Matthias S.; HENTSCHEL, Bernd. Visualization of Memory Access Behavior on Hierarchical NUMA Architectures. In: *Proceedings of the First Workshop on Visual Performance Analysis*. New Orleans, Louisiana: IEEE Press, 2014, s. 42–49. VPA '14. ISBN 978-1-4799-7058-2. Dostupné z DOI: 10.1109/VPA.2014.12.

32. BENIAMINE, David; DIENER, Matthias; HUARD, Guillaume; NAVAUX, Philippe O. A. TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures. In: *Proceedings of the 2Nd Workshop on Visual Performance Analysis*. Austin, Texas: ACM, 2015, 1:1–1:9. VPA '15. ISBN 978-1-4503-4013-7. Dostupné z DOI: 10.1145/2835238.2835239.

33. *Programming languages - C++*. Geneva, CH, 2017. Standard. International Organization for Standardization.

34. BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. Understanding TypeScript. In: *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*. Ed. JONES, Richard. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, s. 257–281. ISBN 978-3-662-44202-9. Dostupné z DOI: 10.1007/978-3-662-44202-9_11.

A Struktura projektu

Projekt, který je přiložen na DVD, má následující adresářovou strukturu.

- `/`
  - `collection` ... zdrojové kódy profilovacího nástroje
    - `benchmark` ... měření výkonu
    - `include`
      - `CuprRuntime.h` ... hlavičkový soubor určený pro vložení do instrumentovaného kódu
    - `instrument` ... instrumentační průchod
    - `runtime` ... knihovna pro sběr profilovaných dat
    - `runtimetracker` ... knihovna pro zaznamenávání alokací
    - `tests` ... testy
    - `samples` ... příklady profilovaných programů s naměřenými daty
  - `schema` ... Schémata generovaných souborů
  - `thesis`
    - `thesis.pdf` ... PDF soubor s textem této práce
  - `visualization` ... vizualizační aplikace

Projekt je verzován pomocí verzovacího systému git. Repozitář projektu je dostupný online na této adrese: `https://github.com/kobzol/cuda-profile`. Webové vizualizační rozhraní je možné si vyzkoušet zde: `https://kobzol.github.io/cuda-profile`.

B Instalace projektu

B.1 Profilovací nástroj

Profilovací nástroj byl primárně vyvíjen pro Linuxové distribuce založené na Debianu, instalační instrukce jsou tedy zaměřené na ně. Jeho kód se nachází ve složce `collection`, následující příkazy předpokládají spouštění z této složky.

B.1.1 Závislosti

Projekt lze sestavit pomocí programu *CMake*[17], který ze zdrojového kódu generuje soubory pro konkrétní sestavovací systém (např. *make*). Před sestavením projektu je nutné nainstalovat závislosti CMake a LLVM. Instrumentační nástroj by měl být kompatibilní s LLVM 4, 5 i 6 a s rozhraním CUDA ve verzi 8 a 9. Podpora pro CUDA 9.1 je v době tvorby této práce stále ve vývoji. Mezi verzí CUDA 8 a 9 vznikly relativně velké změny ve spolupráci rozhraní CUDA s Clangem, doporučená a nejvíce otestovaná kombinace verzí je pro použití profilovacího nástroje je LLVM 4 a CUDA 8. Pro rychlou instalaci potřebných závislostí lze použít následující příkaz.

```
$ apt-get install build-essential cmake llvm-4.0-dev clang-4.0
```

Pro použití komprese a knihoven Protobuf či Cap'n Proto je potřeba doinstalovat tyto balíčky. Aplikace používá Protobuf ve verzi 2 a Cap'n Proto ve verzi 0.6.

```
$ apt-get install libprotobuf-dev protobuf-compiler capnproto libcapnp-dev zlib1g-dev
```

Testy jsou napsány v Pythonu 2.7 a jsou určené pro testovací knihovnu *pytest*[18]. Pro spuštění testů je nutné doinstalovat balíčky Python, pytest a dále nainstalovat Protobuf a Cap'n Proto pro Python.

```
$ apt-get install python python-pip python-pytest
$ pip install protobuf pycapnp
```

B.1.2 Sestavení

Pro sestavení projektu do složky *build* lze použít tyto příkazy:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=RELEASE ..
$ make -j
```

[17] `https://cmake.org`
[18] `https://docs.pytest.org/en/latest`

Po sestavení lze v kořenovém adresáři projektu spustit testy pomocí tohoto příkazu:

```
$ py.test tests
```

Pro zjednodušení instalace profilovacího nástroje je v repozitáři k dispozici konfigurační soubor pro *Docker*[19], který obsahuje všechny potřebné závislosti. Jelikož profilovací nástroj vyžaduje CUDA grafickou kartu pro své spuštění, musí se Docker kontejner spustit s podporou běhového prostředí CUDA. Pro sestavení projektu v Dockeru a následné spuštění testů lze použít tyto příkazy v kořenovém adresáři projektu:

```
$ apt-get install nvidia-docker2
$ docker build -t profiler .
$ docker run --runtime=nvidia profiler
```

Kontejner je možné zapnout v interaktivním režimu a použít jej k instrumentaci a spuštění CUDA programů.

B.1.3 Použití

Pro každý kernel, který se má instrumentovat, je nutné vložit do jeho zdrojového `.cu` souboru hlavičkový soubor `CuprRuntime.h` ze složky `include`.

```
#include <CuprRuntime.h>
```

Aby šlo takto soubor vložit, tak musí být složka `include` přidána do seznamu cest ke hlavičkovým souborům v překladači.[20]

Instrumentovaný program se poté musí přeložit pomocí překladače Clang. Při překladu je nutné přilinkovat knihovny profilovacího nástroje. Následující skript přeloží zdrojové soubory a provede instrumentaci profilovacím nástrojem. Skript předpokládá, že v proměnné prostředí CUPR_BUILD_DIR je absolutní cesta k adresáři se sestaveným profilovacím nástrojem (tedy např. adresář `build`, viz výše).

[19] `https://www.docker.com`

[20] Nestačí zkopírovat hlavičkový soubor do jiného projektu, protože se odkazuje na jiné soubory profilovacího nástroje pomocí relativních cest

```
$ clang++ -std=c++14 --cuda-gpu-arch=sm_30 \
      -I/usr/local/cuda/include -L/usr/local/cuda/lib64 \
      -I${CUPR_BUILD_DIR}/include \
      -L${CUPR_BUILD_DIR}/runtime \
      -Xclang -load -Xclang ${CUPR_BUILD_DIR}/instrument/libinstrument.so \
      -lruntime \
      -z muldefs \
      -lcudart \
      -xcuda \
      <zdrojové soubory> -o cuda
```

Přepínač `-z muldefs` je nutný pouze pokud je soubor `CuprRuntime.h` vložen do více než jednoho souboru s kernely. Tento přepínač ignoruje výskyt více definic stejné deklarace a je nutný z důvodu omezení Clangu popsaného v sekci 4.5.

Níže je pro úplnost ukázka minimálního souboru pro instrumentaci a sestavení CUDA programu překladačem Clang pomocí CMake.

```
cmake_minimum_required(VERSION 3.4)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_EXTENSIONS OFF)

set(CUPR_SRC_DIR <repozitář projektu>)
set(CUPR_BUILD_DIR <složka se sestaveným projektem>)

find_package(CUDA REQUIRED)

set(CMAKE_CXX_COMPILER clang++)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -g --cuda-gpu-arch=sm_30 -Xclang -load \
-Xclang ${CUPR_BUILD_DIR}/instrument/libinstrument.so")

set_source_files_properties(kernel.cu PROPERTIES LANGUAGE CXX)

include_directories(${CUDA_INCLUDE_DIRS})
link_directories("${CUDA_TOOLKIT_ROOT_DIR}/lib64")
link_directories("${CUPR_BUILD_DIR}/runtime")

add_executable(profiled main.cpp kernel.cu)
target_link_libraries(profiled cudart runtime)
target_include_directories(profiled PRIVATE ${CUPR_SRC_DIR}/include)
```

Instrumentovaný program potřebuje k běhu dynamickou knihovnu profilovacího nástroje, je tedy nutné buď vytvořit symbolický odkaz ke knihovně pomocí programu `ldconfig`[21] nebo použít proměnnou prostředí LD_LIBRARY_PATH[22] pro určení umístění knihovny. Následuje

[21] `http://man7.org/linux/man-pages/man8/ldconfig.8.html`
[22] `http://man7.org/linux/man-pages/man8/ld.so.8.html`

kompletní ukázka spuštění programu i s použitím knihovny pro zaznamenání dynamických alokací a předání parametrů.

```
$ LD_PRELOAD=${CUPR_BUILD_DIR}/runtimetracker/libruntimetracker.so \
  LD_LIBRARY_PATH=${CUPR_BUILD_DIR}/runtime \
  FORMAT=PROTOBUF \
  COMPRESS=1 \
  ./cuda
```

Po ukončení programu se v jeho pracovním adresáři vytvoří složka se zaznamenanými údaji. Ty lze poté zobrazit ve vizualizační aplikaci.

B.1.4 Parametry

V tabulce 5 jsou vypsány názvy, datové typy a implicitní hodnoty parametrů, kterými lze ovlivnit instrumentaci a běh profilovaných programů. Veškeré parametry se předávají pomocí proměnných prostředí. Pokud je v tabulce uveden kontext *překladač*, tak se parametr předává při překladu programu překladači Clang, kontext *program* značí předání parametru při spouštění instrumentovaného programu. U parametrů typu `bool` se pravdivá hodnota zadává jako 1 a nepravdivá hodnota jako 0.

Tabulka 5: Parametry profilovacího nástroje

Název	Typ	Implicitně	Kontext	Popis
INSTRUMENT_LOCALS	bool	vypnuto	překladač	Instrumentace lokálních přístupů
KERNEL_REGEX	řetězec		překladač	Regulární výraz pro filtrování kernelů
BUFFER_SIZE	číslo	1000000	program	Velikost bufferu pro přístupy [počet přístupů]
PRETTIFY	bool	vypnuto	program	Odsazení pro lepší čitelnost JSONu
COMPRESS	bool	vypnuto	program	Gzip komprimace vygenerovaných souborů
FORMAT	řetězec	JSON	program	Serializační formát (PROTOBUF, CAPNP, JSON)
HOST_MEMORY	bool	vypnuto	program	Použití paměti CPU pro ukládání přístupů
THREAD_POOL	bool	zapnuto	program	Paralelizace formátování přístupů

B.1.5 Formát generovaných souborů

Při instrumentaci a spouštění instrumentovaných programů se generují tři typy souborů. Jejich specifikace je umístěna v kořenovém adresáři projektu ve složce `schema`. Soubor s informacemi o běhu celé aplikace má název `run.json` a vytváří se při každém spuštění instrumentovaného programu. Dalším typem souboru je soubor s metadaty, který je vždy pojmenován `<nazev>.metadata.json`, kde `nazev` je název kernelu, jehož metadata soubor obsahuje. Tento soubor je vytvořen pro každý instrumentovaný kernel při překladu aplikace. Posledním a nejdůležitějším typem souboru je soubor s paměťovými přístupy. Ten se vytváří za běhu programu při každém spuštění instrumentovaného kernelu.

B.2 Webová aplikace

Vizualizační aplikace se nachází ve složce `dashboard`, následující příkazy předpokládají spouštění z této složky.

B.2.1 Závislosti

Pro instalaci a spuštění vizualizační webové aplikace je nutné nainstalovat běhové prostředí Javascriptu *NodeJS*[23] alespoň ve verzi 8 a jeho správce balíčků *npm*[24]. Závislosti projektu lze nainstalovat tímto příkazem:

```
$ npm install
```

B.2.2 Použití

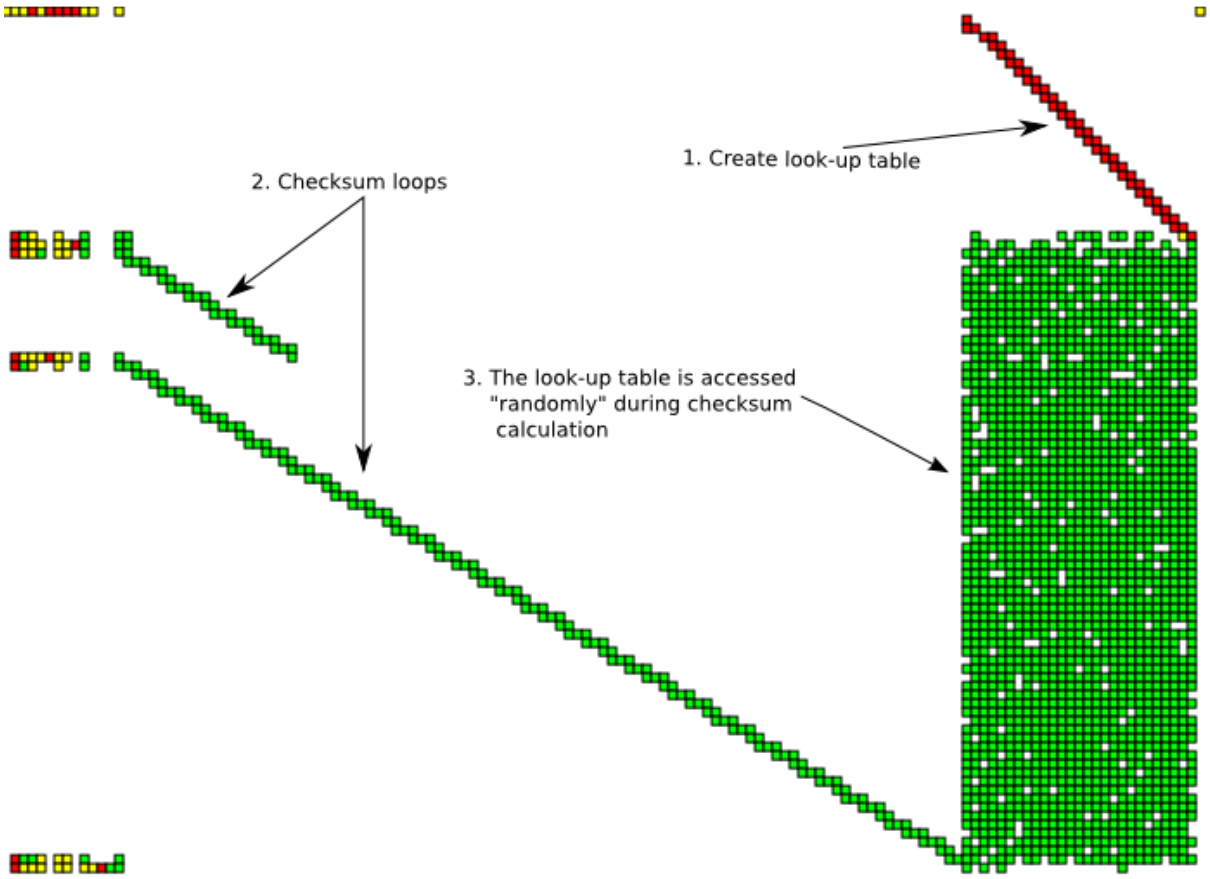
Aplikaci je poté možno spustit buď přímo ve vývojářském režimu nebo ji lze sestavit do statických HTML, CSS a Javascript souborů a poté použít libovolný webový server k servírování aplikace ze složky `build`:

```
$ npm run start # vývojářský režim
$ npm run build # sestavení do statických souborů
```

Aplikace při spuštění ve vývojářském režimu běží na portu 3000, lze se na ni tedy dostat v prohlížeči pomocí adresy `http://localhost:3000`. Webovou aplikaci si lze prohlédnout také na této adrese: `https://kobzol.github.io/cuda-profile`.

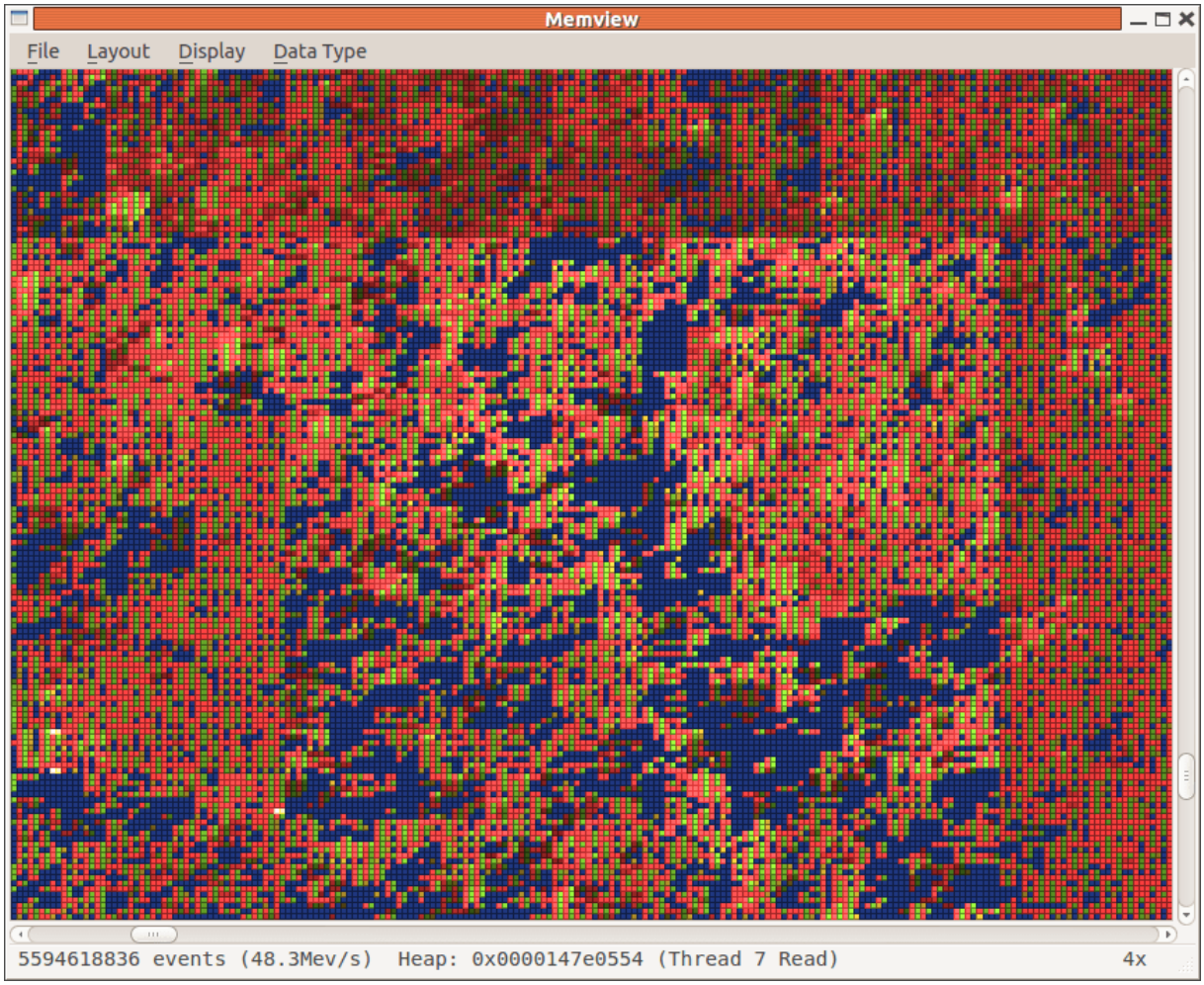
[23] `https://nodejs.org`
[24] `https://www.npmjs.com`

C Ukázky vizualizace paměťových přístupů



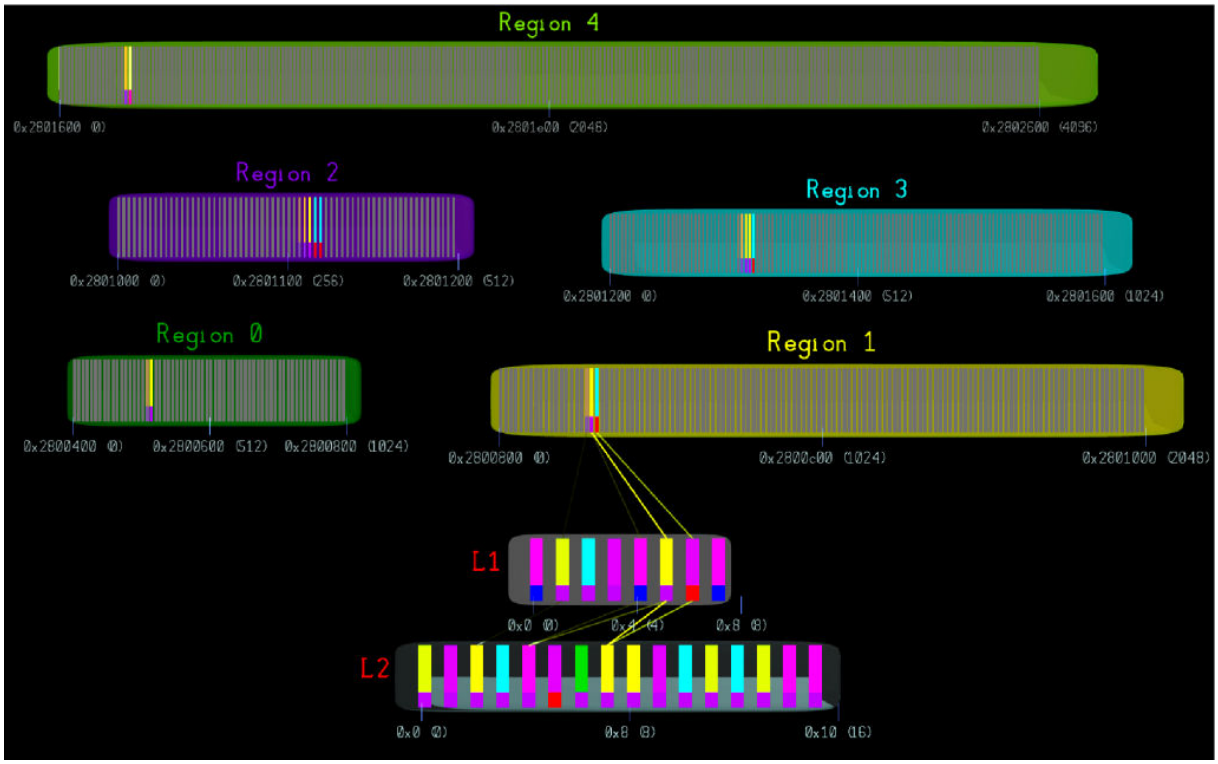
Obrázek 8: Vizualizace paměťových přístupů v nástroji Tracectory

Zdroj: `https://bling.kapsi.fi/blog/x86-memory-access-visualization.html`



Obrázek 9: Vizualizace paměťových přístupů v nástroji Memview

Zdroj: `https://github.com/ajclinto/memview`



Obrázek 10: Vizualizace paměťových přístupů a cache v nástroji Memory Trace Visualiser

Zdroj: A. N. M. Imroz Choudhury, 2008 [30]