VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

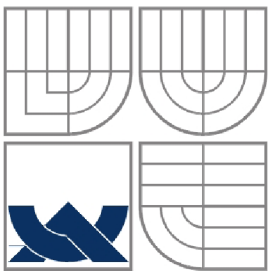
PŘEKLADAČ JAZYKA C# DO JAZYKA NVIDIA CUDA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

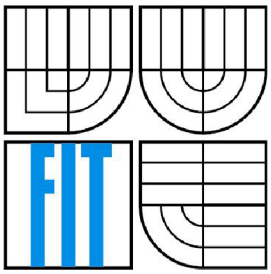
AUTOR PRÁCE
AUTHOR

BC. JIŘÍ ZAJÍC

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEKLADAČ JAZYKA C# DO JAZYKA NVIDIA CUDA

COMPILER FROM C# LANGUAGE TO NVIDIA CUDA LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. JIŘÍ ZAJÍC

VEDOUCÍ PRÁCE
SUPERVISOR

ING. PETER JURNEČKA

BRNO 2012

Abstrakt

Tato diplomová práce je zaměřena na akceleraci výpočtů na grafické kartě NVidia pomocí technologie CUDA s implementací na platformě .NET. Problém je řešen jako překladač jazyka C# do jazyka NVidia CUDA s využitím výrazových schopností jazyka C#, jenž přináší větší míru abstrakce při zachování stejné sémantiky akcí. Aplikace je implementována v jazyce C# s využitím open-source knihovny NRefactory.

Abstract

This master's thesis is focused on GPU accelerated calculations on NVidia graphics card. CUDA technology is used and converted to implementation on a .NET platform. The problem is solved as a compiler from C# programing language to NVidia CUDA language with expression atrributes of C# language that preserves the same semantics of actions. Application is implemented in C# programing language and uses NRefactory, the open-source library.

Klíčová slova

CUDA, C#, .NET, NVidia, překladač, NRefactory, Microsoft, MS Visual Studio

Keywords

CUDA, C#, .NET, NVidia, compiler, NRefactory, Microsoft, MS Visual Studio

Citace

Jiří Zajíc: Překladač jazyka C# do jazyka NVidia CUDA, diplomová práce, Brno, FIT VUT v Brně, 2012

Překladač jazyka C# do jazyka NVidia CUDA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Petera Jurnečky, Ing. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jiří Zajíc
23.5.2012

Poděkování

Děkuji svému vedoucímu Ing. Peteru Jurnečkovi za odbornou pomoc, kterou mi poskytoval během celé práce na této diplomové práci.

Obsah

1 Úvod

V moderním pojetí programování se stále více aplikací směřuje do paralelních výpočtů, ať už formou vláken či celých procesů. V okamžiku, kdy byl v počítači přítomen pouze jeden procesor to nemělo tak velký smysl jako dnes, kdy se vyrábí více-jádrové procesory a grafické čipy obsahují několik desítek či stovek výpočetních jednotek.

V okamžiku, kdy se výpočetní síla grafické karty stala vyšší než centrálního procesoru, tlak na přesun náročných výpočtů (ne-grafického charakteru) na grafickou kartu vzrostl. Touto cestou se vydala společnost NVidia vyrábějící grafické čipy, která navrhla technologii pro provádění těchto výpočtů s názvem CUDA. Ta používá pro implementaci algoritmů upravený jazyk C s názvem CUDA C.

S vědomím, že se potýkáme při programování se stále větší mírou abstrakce, se vyvíjí programovací jazyky, které toto paradigma zohledňují. Příchod objektově-orientovaných jazyků byl předznamenáním tohoto trendu a zohlednila jej práce společnosti Microsoft, která vyvíjí .NET Framework, který vedle vysokoúrovňových programovacích jazyků jako C# nebo Visual Basic .NET, poskytuje vyšší úroveň abstrakce i v dalších aplikacích, ať je to například odstínění běhu aplikace virtuálním strojem nebo automatická správa paměti.

Tato diplomová práce se zabývá překladem jazyka C# do jazyka NVidia CUDA. Teoretický základ akcelerace výpočtů na grafické kartě je popsán ve druhé kapitole. Zde je prezentován především v podobě technologie NVidia CUDA, popisu jejích částí, prezentace průběhu výpočtu a výčtu participujících komponent.

Technologie, které byly použity při implementaci projektu jsou popsány ve třetí kapitole. Je zde podán rozbor technologie .NET z pohledu architektury, princip překladu zdrojových kódů a zevrubný popis jazyka C#. Dále je zde přiblížena syntaxe a zvláštnosti implementačního jazyka pro technologii CUDA s názvem CUDA C. Kapitolu uzavírá popis knihovny NRefactory využité pro implementaci překladače.

Čtvrtá kapitola přibližuje existující řešení pro danou problematiku. Popisuje možnosti jednotlivých přístupů v podobě teoretických postupů i již implementovaných projektů. Zaznamenány jsou výhody jednotlivých přístupů i omezení daných řešení.

Nejrozsáhlejší obsahem je pátá kapitola popisující návrh řešení celé aplikace. Kapitola obsahuje návrh celé architektury, rozdělení na jednotlivé projekty. Jsou zde popsány jednotlivé komponenty systému včetně obrazové dokumentace. Součástí kapitoly je popis návrhového vzoru MVVM, který byl použit při návrhu grafického rozhraní.

Praktický výstup této práce je zaznamenán v šesté kapitole, která je zaměřena na implementaci aplikace. Výsledkem diplomové práce je aplikace, ve které je možné upravovat, překládat a spouštět projekty v jazyce C#. Tato kapitola rozebírá její části i omezení.

Implementovaná aplikace byla otestována sadou referenčních příkladů. Popis testovacích aplikací je přiblížen v kapitole sedmé, která zahrnuje vybrané příklady, které byly použity k ověření funkčnosti aplikace.

Zhodnocení projektu přináší kapitola 8. Zde je shrnuta práce na celé práci s výčtem nedostatků a pozitiv, doplněna hodnocením aplikace. Součástí je nástin možných rozšíření a možnosti dalšího postupu.

2 Akcelerace výpočtů na grafické kartě

S rozvojem herního průmyslu rostly i požadavky na výkon grafických karet, což vyústilo v situaci, kdy výkon grafické karty předčil výkon centrálního processoru. Tato situace nastala na přelomu let 2002 a 2003, kdy grafické karty GeForce FX 5800 a GeForce FX 5950 Ultra od firmy NVidia předčily tehdy osazované procesory Intel typu Pentium 4 (s frekvencí 2,4 GHz). Vedlejším efektem bylo představení technologií CUDA od Nvidie, ATI Stream od společnosti AMD/ATI a standardu OpenCL (Open Computing Language) od konsorcia Khronos (tvořené společnostmi AMD, Intel, NVidia a další), které řeší přesun náročných výpočtů na grafickou kartu. Informace pro tuto kapitolu byly čerpány z pramenů [5], [11], [12], [15] a [24].

2.1 CUDA

Technologie CUDA (Computer Unified Device Architecture) je kompatibilní se standardem OpenCL a DirectCompute od společnosti Microsoft. Byla představena v roce 2006 společně s architekturou G80. 2. února 2007 bylo vydáno první SDK 1.0 pro karty NVidia Tesla, založené na této architektuře. Téhož roku byl vydán patch na verzi 1.1 pro podporu karet série GeForce 8. V roce 2008 bylo současně s architekturou G200 vydáno i SDK 2.0. Velkým skokem bylo vydání SDK verze 3.0 (2010) s nativní podporou výpočtů v plovoucí řádové čárce s dvojnásobnou přesností, ukazatelů na funkce a podporou rekurze. V současnosti je aktuální verze 4.0 (květen 2011) s unifikací paměťových prostorů a podporou výpočtů současně na několika grafických kartách najednou.

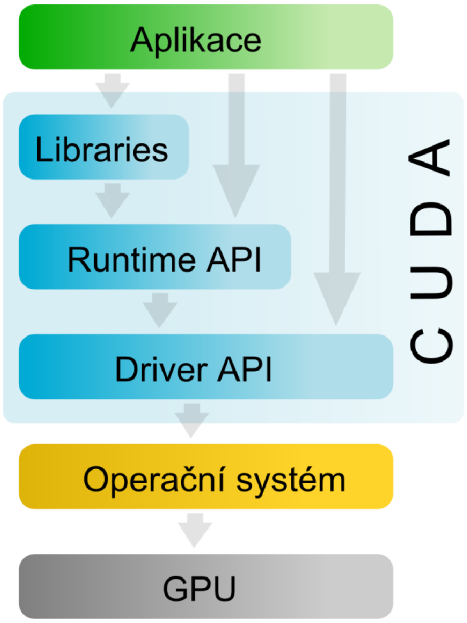
Nativní podpora pro technologii CUDA je v podobě jazyků C, C++, ale existuje množství knihoven třetích stran, například pro Python, Javu, Fortran nebo Haskell. Podporovány jsou operační systémy Windows, Linux a Mac OS X.

2.1.1 Architektura

Z pohledu GPU je zařízení rozděleno na několik multiprocesorů a globální paměť, která slouží pro kopírování dat z RAM na GPU a naopak. Každý multiprocesor obsahuje několik procesorů, které jsou řízeny instrukční jednotkou. Procesory (obvykle 32-bitový procesor SIMD) mají přístup ke svým vyhrazeným registrům, dále k paměti, která je sdílená všemi procesory, a k rychlým vyrovnávacím pamětem pro uložení konstant a textur. Každý procesor má také přístup ke globální paměti celé grafické karty.

Typický běh CUDA aplikace probíhá tak, že se inicializuje zařízení, nakopírují se na něj hodnoty, nad těmito hodnotami se provede výpočet a ze zařízení se překopíruje výsledek, který se zpracuje už normálním CPU.

CUDA aplikace může přistupovat ke GPU trojím způsobem. Nejnižší úroveň představuje CUDA Driver API. Ten zajišťuje funkce zahrnující správu paměti (např. kopírování dat z a na zařízení), správu zařízení (informace o zařízení, připravenost GPU, atd.), podporu pro OpenGL a Direct3D, správu řízení aplikace a další. Služby CUDA Driver API využívá CUDA Runtime API. To zahrnuje správu vláken, kontrolu chyb, nadstavbové funkce pro správu paměti, zařízení a řízení, a také správu textur, OpenGL a Direct3D. Nejvýše postavené jsou CUDA knihovny zajišťující matematické operace – CUBLAS (Basic Linear Algebra Subprograms) nebo CUFFT (rychlá Furierova transformace). Architekturu ilustruje následující obrázek [1]:



Ilustrace 1: Architektura CUDA [vlastní]

2.1.2 Omezení

Největším omezením je dostupnost aplikačního rozhraní CUDA pouze pro grafické karty od společnosti NVidia, pro ostatní karty je třeba využít jinou technologii. Další nevýhodou je fakt, že díky kopírování dat mezi operační pamětí a pamětí GPU může dojít k takovému celkovému zpomalení, že výsledná aplikace je pomalejší, než by byla aplikace implementovaná pouze na CPU. Z hlediska programování je velmi nepříjemná skutečnost, že validní C/C++ kód může být označen překladačem za nevalidní díky optimalizačním technikám, které kompilátor využívá kvůli omezeným zdrojům.

3 Použité technologie

Pro implementaci aplikace jsem se rozhodl využít .NET platformu díky tomu, že s ní mám zkušenosti v rámci svého zaměstnání a jevila se mi tak jako nejlepší možnost. Přispěla k tomu i velká dostupnost mnoha nástrojů a návodů.

První podkapitola se věnuje technologii Microsoft .NET Framework, na které je postaveno řešení celého projektu. Text kapitoly popisuje její běhové prostředí a základní knihovny, které byly využity pro implementaci. Jako implementační jazyk byl zvolen C#, kterému je rovněž věnována jedna podkapitola. Při práci na této části byly využíty zdroje [1] - [4] a [6] .

Další podkapitola se věnuje důležitému souboru knihoven s názvem NRefactory. V rámci podkapitoly je představeno využití a nejdůležitější jmenné prostory knihovny. Doplňující informace a materiály o tomto projektu jsou dostupné v [17], [18].

Cílový jazyk, tedy CUDA C, je popsán v poslední podkapitole tohoto bloku. Ta přibližuje nejdůležitější aspekty tohoto jazyka a princip překladu zdrojového kódu psaného v tomto jazyce. Kapitola vychází ze znalostí nabytých v kapitole 2.1.1.

3.1 Microsoft .NET Framework

.NET Framework je softwarová komponenta, součást systému MS Windows vyvinutá společností Microsoft, jež slouží pro vývoj aplikací a webových služeb. Základním požadavkem je zejména poskytování objektově orientovaného prostředí pro vývoj softwaru, který může být uložen a spuštěn lokálně, nebo spuštěn lokálně, ale distribuován prostřednictím sítě i spuštěn vzdáleně. Framework klade důraz na minimalizací problémů s nasazením a verzováním výsledného produktu. Odstíněním virtuálního stroje dosahuje bezpečného spouštění aplikací.

Mezi hlavní přednosti frameworku patří snaha o sjednocení vývoje pro desktopové stanice, mobilní zařízení a vestavěná zařízení zároveň. Nejrozšířenější je platforma Microsoft .NET Framework, která je určená pro osobní počítače s verzí operačního systému MS Windows (verze 1.0 byla dostupná už od Windows NT). Druhou skupinu tvoří Microsoft .NET Compact Framework pro mobilní zařízení s operačním systémem Windows Mobile, využívající knihovny jádra normálního .NET Frameworku designované tak, aby zabíraly méně místa. Poslední skupinu tvoří nejvíce omezený .NET Micro Framework, jenž je určen jen pro vestavěná zařízení. Skupinou stojící mimo výše zmíněné je Mono, což je open-source platforma, která poskytuje běhové prostředí .NET pro systémy Linux a MAC OS X. Jádrem frameworku jsou dvě hlavní komponenty: Common Language Runtime a .NET Framework Class Library.

3.1.1 Virtuální stroj

CLR (Common Language Runtime) zajišťuje nejdůležitější funkce .NET Frameworku. Je to zejména činnost virtuálního stroje, pomocí kterého jsou překládány, spouštěny a řízeny veškeré programy psané v programovacích jazycích v rámci .NET.

Zdrojový kód napsaný v některém z jazyků v rámci platformy (z nejznámějších jsou to C#, Visual Basic, Visual C++, F# nebo J#) je přeložen nejprve svým překladačem do tzv. CIL kódu. Tím skončí překlad zdrojového kódu v době kompilace. CIL kód je pak v době běhu aplikace překládán pomocí CLR do nativního kódu. Tento proces je znám jako *just-in-time* (JIT) překlad.

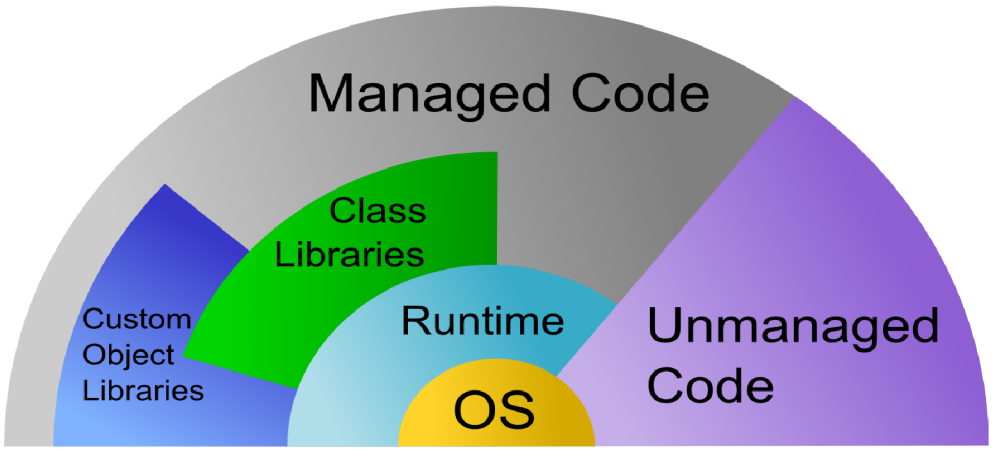
CLR dále zajišťuje například typovou kontrolu a automatickou správu paměti, implementovanou jako třífázový *garbage-collector*. Odchytávání výjimek, vzniklých během běhu aplikace, je taktéž řízeno pomocí CLR.

3.1.2 Knihovny tříd

Knihovna jádra .NET Frameworku zahrnuje širokou paletu objektově-orientovaných typů, které jsou integrovány s CLR. Je to zejména kolekce základních typů (*Object*, *String*, *Char*, *Int32* atd.) nebo strukturované typy jako kolekce nebo slovníky. Knihovny tříd poskytují nástroje:

- Pro tvorbu konzolových aplikací,
- Třídní systém pro Windows Forms aplikace (prostředí pro tvorbu formulářových aplikací),
- Windows Presentation Foundation (WPF) systém (pro aplikace s bohatým grafickým uživatelským rozhraním),
- ASP.NET aplikace (nástroje pro tvorbu webových aplikací), webové služby a jiné.

Vztah mezi jádrem operačního systému, CLR, knihovnami a aplikacemi přibližuje obrázek [2]:



Ilustrace 2: Vztah komponent .NET frameworku se zdrojovým kódem [vlastní]

3.1.3 Programovací jazyk C#

Pro implementaci aplikace byl zvolen jazyk C#. Důvodem bylo, kromě znalosti jazyka, několik skutečností, zejména ale fakt, že poskytuje široké možnosti, co se týče načítání a parsování zdrojových kódů a díky integraci C# a vývojového prostředí Visual Studio .NET je vývoj v něm rychlý a pohodlný.

Jazyk byl původně pojmenován SMC (Simple Managed C) jako vedlejší produkt vývoje .NET frameworku. Ve svých počátcích představoval pouze sadu objektově orientovaných knihoven v jazyku podobnému základnímu jazyku C. Vývoj jazyka byl oficiálně oznámen na konferenci v roce 2000 u příležitosti představení projektu .NET. Základy jazyka položil projektový architekt Microsoftu Anders Hejlsberg, který navrhl systém CLR. Ten se stal základem pro C# 1.0, vydaný Microsoftem v lednu 2002 a byl oficiálně standardizován normou ISO/EIC 23270 v dubnu 2003.

C# vychází z jazyka C++. Proti němu nepodporuje globální funkce či proměnné, vše musí být definováno uvnitř tříd. Všechny typy jsou děděny ze základního typu *Object* a všechny implementují funkci *ToString()*. C# nepodporuje konverzi celočíselné hodnoty na hodnotu typu boolean při použití v podmíněném příkazu a cyklech. Díky automatické správě paměti nelze explicitně alokovat a dealokovat místo v paměti, vše řeší *garbage-collector*. C# nepodporuje vícenásobnou dědičnost, i když jedna třída může implementovat více rozhraní.

Verze 2.0 zahrnula genericitu typů, anonymní metody a iterátory. S verzí 3.0, vydanou v srpnu 2007, přišly funkcionální prvky C# v podobě systému LINQ, lambda metod a anonymních tříd. Poslední verzí C# je 4.0, vydaná v dubnu 2010. Největšími přínosy jsou nově udělaný *Task* systém, který implementuje zjednodušenou kontrolu nad vlákny, a dynamické bindování do grafického rozhraní (využívané ve Windows Forms a WPF aplikacích). Tato verze byla použita k implementaci aplikace.

3.2 NRefactory

NRefactory je open-source knihovna napsaná v jazyce C#, která slouží pro syntaktickou a sémantickou analýzu zdrojových kódů psaných na platformě .NET. Knihovnu naprogramoval Daniel Grunwald a její vývoj byl zahájen v září 2009, kdy byla vydána verze 1.0. Knihovna podporovala abstraktní syntaktický strom, který mohl reprezentovat jakoukoli konstrukci jazyka C# nebo VisualBasic .NET. Obsahovala větší možnost reprezentace konstrukcí než vestavěné typy, které obsahuje .NET Framework ve jmenném prostoru *System.CodeDom*. Vývoj knihovny pokračoval do verze 4, kde byla odladěna většina interních chyb. V srpnu 2011 byla vydána NRefactory 5, která

byla kompletně přepsána. Byla přidána sémantická analýza, která je však momentálně dostupná jen pro jazyk C#.

Verze 5 je postavena na .NET Frameworku verze 4.0. Je závislá na překladači *Mono.Cecil* verze 0.9.5, což je open-source knihovna, která slouží pro překlad a generování programů, převeditelných do CIL formátu (viz kapitola 4.1.1). NRefactory obsahuje modifikovanou kopii tzv. *msc*, což je Mono překladač C#.

Knihovna obsahuje 8 jmenných prostorů, z nichž byly využity tyto dva:

- *NRefactory Core*
  - *TypeSystem* – obsahuje jazykově nezávislou reprezentaci typového systému .NET,
  - *TypeSystem.Implementation* – implementace rozhraní typových systémů,
  - *Semantics* – třídy reprezentující sémantiku jazykových elementů,
  - *PatternMatching* – třídy pro rozpoznávání vzorů v abstraktních syntaktických stromech pro C# a Visual Basic .NET,
  - *Documentation* – dokumentace pro Nrefactory,
  - *Util* – pomocné třídy.
- *NRefactory.CSharp*
  - obsahuje abstraktní syntaktický strom pro C#,
  - *Completion* – *IntelliSence* pro C#,
  - *Resolver* – sémantická kontrola výrazů,
  - *Analysis* – sémantická kontrola celých zdrojových kódů,
  - *TypeSystem* – implementace typového systému pro jazyk C#.

Další jmenné prostory jsou:

- *VB* – syntaktický strom pro Visual Basic .NET,
- *Tests* – unit testy a další kontroly pro ověřování funkčnosti Nrefactory,
- *ConsistencyCheck* – kontrola na reference uvnitř Nrefactory,
- *Demo*, *GtkDemo*, *VB.Tests* – další projekty, které slouží pro ukázku aplikace či její testování.

3.3 Jazyk CUDA C

Technologie CUDA využívá podmnožinu jazyka C/C++ s názvem CUDA C. Pro pochopení smyslu implementace CUDA aplikací je nutné znát architekturu technologie CUDA, popsanou v kapitole 2.2.1. Jazyk CUDA C je rozšířením ANSI C. Jeho syntaxe je téměř totožná se standartním jazykem C. Navíc obsahuje klíčová slova, která slouží pro označení sémantiky částí zdrojového kódu určeného pro běh na grafické kartě. Jeho součástí je i nová syntaxe ve volání určitých funkcí.

3.3.1 Kernel funkce

Hlavním stavebním kamenem CUDA C je tzv. kernel, což je definice funkce na grafické kartě. Tato funkce je uvozena klíčovým slovem *__global__*, její návratová hodnota je vždy *void* a je spuštěna současně v jednotlivých vláknech a blocích. Spuštění probíhá po skupinách nazvaných *warp*. *Warp* je skupina 32 vláken, která jsou spuštěna současně. Identifikace vlákna je dána vestavěnou proměnnou *threadIdx*, což je 3-složkový vektor (pro zjednodušenou práci s maticemi). Identifikace bloku je rovněž třísložkový vektor s názvem *blockIdx*. V obou případech je možné použít jen jednu nebo i více složek daného vektoru.

K volání kernel funkce se přidává konfigurace spuštění (execution configuration), což představuje syntaktický zápis *<<<X, Y, Z>>>*, kde X představuje počet bloků, ve kterých se má daná funkce spustit a Y zaznamenává počet vláken v každém bloku, ve kterých se má daná funkce spustit. Modifikátor Z určuje velikost sdílené paměti pro vlákna (udáváno v bytech).

Práce s pamětí v CUDA C je velmi podobná práci v klasickém jazyku C. Pro alokování místa na globální paměti GPU je třeba zavolat funkci *cudaMalloc* (*cudaFree* při uvolnění), analogicky s jazykem C. Pro provedení výpočtu je nejprve potřeba překopírovat proměnné na zařízení, k čemuž sloučí funkce *cudaMemcpy*, která slouží i pro kopírování opačným směrem.

CUDA C poskytuje nástroje pro synchronizaci vláken, které ale může být provedeno jen v rámci jednoho bloku. Je také přítomno odchytávání chyb, které se ale nechová jako konstrukce *try-catch* známé z vyšších programovacích jazyků, ale programátor je nucen sám zjišťovat, zda nedošlo při běhu na zařízení k chybě.

3.3.2 Překlad CUDA C

Překlad CUDA C aplikace probíhá podobně, jako v případě překladu zdrojového kódu standardního jazyka C. Překladač rozdělí zdrojový kód na části psané v čistém C/C++ a funkce psané pro zařízení. Standardní zdrojové kódy jsou přeloženy do podoby připravené pro linkování. Kód pro zařízení je nejprve přeložen do PTX kódu (Parallel Thread Execution). Ten je pak přeložen pomocí překladače pro dané GPU podle cílové architektury. Výsledný kód je pak slinkován dohromady s přeloženými kódy psanými ve standardním C (C++). Přilinkovány jsou též knihovny CUDArt (CUDA Runtime) a CUDA Core Library, nutné k běhu aplikace.

4 Existující řešení

Pro problematiku akcelerace výpočtů na grafické kartě existuje několik řešení, které spojují výpočty na GPU s .NET frameworkem. Tyto projekty byly nápomocny pro návrh aplikace jako inspirace pro přístup k řešení jednotlivých problémů. Některé nápady byly využity, ale koncepčně mají všechna řešení chyby, kterým se návrh aplikace snaží vyhnout.

4.1 Portování C++ knihovny

Tento přístup není nijak zakódován ve formě dostupné knihovny, jedná se o přístup, jak lze řešit akceleraci výpočtů na grafické kartě pomocí technologie CUDA. Předpokladem pro toto řešení je programování aplikací v jazyce C++, nejedná se tedy o implementaci aplikací na .NET platformě.

Princip je následující: Nejprve je naprogramován projekt v C++, který využívá přímo vestavěné funkce CUDA dostupné z knihoven pro C++. Tento projekt je následně slinkován do dynamické knihovny. Při programování v C# jsou následně pomocí konstrukce *DLLImport* portovány funkce přímo z oné *nemanagované* (viz obrázek [2]) knihovny. V rámci řešení byl tento postup využit pro překlad částí zdrojového kódu určeného pro běh na zařízení a následného portování do kódu napsaného v C#, který jej volá. Další informace o tomto přístupu lze nalézt v [10], [26].

4.2 CUDAfy

Tento projekt je již plnohodnotným řešením využívajícím pouze platformu .NET. Je vyvíjen společností Hybrid DSP Systems se sídlem v Nizozemí od května 2011. V základu se jedná o sadu knihoven, které umožňují .NET emulaci volání kernel funkce, podporují několik datových typů, které lze použít k výpočtu na grafické kartě, definici kernel metod, kostant a struktur, a použití asynchronních operací.

Z pohledu programování je přístup knihovny CUDAfy podobný tomu, jaký je použit v návrhu diplomové práce, ale nezohledňuje abstraktní vyjadřovací schopnosti jazyka C#. Jedná se skutečně jen o obal nad jazykem C#, který konvertuje konstrukce C# do odpovídající syntaxe jazyka CUDA C jedna ku jedné. Zajímavostí je že, tento projekt využívá též knihovnu NRefactory pro syntaktickou analýzu jazyka C#.

V návrhu diplomové práce lze najít inspiraci tímto projektem v použití vlastních atributů jako prostředku pro značkování kernel funkcí. Další informace o tomto projektu lze získat v [13] a [14].

4.3 GPU.NET

Projekt společnosti TidePowerd, která byla založena v roce 2009 na University of Alabama, představuje plnohodnotný nástroj pro programování akcelerovaných výpočtů v jazyce C#, jejich překlad do mezikódu, který je následně podle typu grafické karty přeložen buď do PTX kódu pro grafické karty NVidia, Stream IL pro karty od AMD/ATI nebo dalších.

Programování pro GPU.NET vypadá velmi podobě jako v případě CUDAfy, s tím rozdílem, že GPU.NET přistupuje ke spouštění kernel funkcí v podobě zvláštní třídy *Launcher*. Tento přístup je ale neintuitivní v tom, že si programátor sám musí hlídat, zda má nastavenu konfiguraci pro spuštění, která se musí explicitně definovat před voláním kernel funkce. Naopak výhodu proti projektu CUDAfy představuje zapouzdření volání alokace paměti a kopírování proměnných na zařízení do čistého volání kernel funkce, přičemž programátor nemusí práci s pamětí řešit. Znaky tohoto přístupu v upravené podobě jsou rovněž převedeny i do návrhu diplomové práce. Více informací o tomto řešení lze nalézt na adrese [16], [17].

5 Návrh řešení

Tato kapitola je zaměřena na popis návrhu řešení celé aplikace překladače. Zahajuje ji přiblížení motivace k řešení tohoto projektu, která byla výchozím bodem pro návrh celé aplikace. A to zejména z pohledu uživatele, který programuje v jazycích s vysokou mírou abstrakce. Následující podkapitola přibližuje největší problémy, se kterými bylo nutné při návrhu aplikace počítat, a to hlavně srovnání obou participujících jazyků do stejné úrovně, která by byla nejlepším možným kompromisem výrazových schopností obou jazyků.

Po tomto teoretickém úvodu následují podkapitoly, které jsou už plně zaměřeny na konkrétní návrh řešení. Z pohledu hierarchie je to na nejvyšší úrovni návrh celé architektury aplikace. Ta je rozdělena do čtyř projektů: knihovna pro export sémantických konstrukcí specifických pro jazyk CUDA C do jazyka C#, dále překladač C# kódu, který zajišťuje syntaktickou a sémantickou kontrolu. Třetím projektem je knihovna generátoru, která poskytuje prostředky pro překlad C# do CUDA C. Projektem, který řídí celý proces generování, překladu a spuštění, je interpret výsledků popsaný poslední podkapitolou. Výklad celého návrhu je provázen obrazovou dokumentací, která má za úkol přiblížit nejasné, či důležité části projektů.

5.1 Motivace

Aplikace je koncipována jako výukový systém pro začínající programátory, kteří mají zkušenost s jazykem C# a chtějí si vyzkoušet práci na paralelních výpočtech s využitím grafické karty jako prostředku pro urychlení výpočtů. Díky využití jazyka C#, tedy jazyka s vysokou mírou abstrakce, byl kladen zřetel na zapouzdření některých částí jazyka CUDA C, u kterých by nastavená abstrakce neohrozila výpočetní sílu jazyka CUDA C.

Pro příklad lze vzít volání kernel funkce. Režie spojená s volání funkce na zařízení je v případě jazyka CUDA C velká. Nejprve je třeba alokovat globální paměť na GPU a pak pomocí funkce *cudaMemcpy* nakopírovat operandy výpočtu do globální paměti grafické karty. Pak lze teprve provést výpočet. Následně je potřeba opět přenést výsledek z paměti GPU do operační paměti a uvolnit paměť na GPU.

Všechny tyto problémy mohou být abstrahovány do takové míry, že lze specifikovat pouze která funkce se má na zařízení zavolat, přižemž jsou jí do parametrů předány operandy, počet bloků, vláken a velikost sdílené paměti. A pokud je očekáván nějaký výsledek, přímo jej lze přiřadit jako výsledek volané funkce. Tato konstrukce v jazyce C# by se pak při překladu rozložila do zmíněných konstrukcí jazyka CUDA C, přičemž by byla zachována stejná výpočetní síla obou aplikací při vyšší míře abstrakce.

Podobný postup jako ve výše zmíněním případě je dále aplikovatelný například na odchytávání chyb, převodem z ručního programování zachytávání chyb v případě CUDA C, na systém zpracovávající výjimky při implementaci v jazyce C#.

5.2 Problémy k řešení

Pro psaní CUDA aplikací v jazyce C# bylo potřeba kompletně navrhnout schéma překladu daného kódu a jeho spuštění tak, aby nebyla ohrožena výpočetní síla jazyka CUDA C, ale i tak, aby bylo možné zohlednit výrazové prostředky jazyka C#. Ze strany jazyka CUDA C bylo potřeba dodat volání vestavěných funkcí pro kopírování proměnných z a na grafickou kartu, dále vestavěné proměnné pro identifikaci bloku a vlákna, synchronizace paměti apod.

Převod ze strany jazyka C# má dvě části. Jednak převod syntaxe, která je v obou jazycích v základu totožná. Druhým problémem je převod tříd a jmenných prostorů do prostředí jazyka C. Problém u převodu syntaxe nastává v okamžiku, kdy je potřeba alokovat paměť. C# pracuje s automatickou správou paměti a tudíž problémy s alokováním a dealokováním paměti nemusí programátor řešit. Naproti tomu práce s pamětí u jazyka C (potažmo CUDA C) vyžaduje ruční implementaci alokace a dealokace. Bylo tedy potřeba v okamžiku, kdy dochází k práci se strukturovanými typy, řešit i problémy s pamětí.

Pro převod tříd a jmenných prostorů byla použita stejná technika, a to rozlišování názvu každého člena pomocí prefixu. Pro příklad: funkce *A()* ze třídy *CLASS* a jmenného prostoru *NSPACE* je po převodu do jazyka CUDA C zaznamenána jako *NSPACE_CLASS_A()*. Sémantická kontrola tohoto řešení musí obsahovat kontrolu typů, definice funkce ve třídě, deklarace třídy ve jmenném prostoru a podobně.

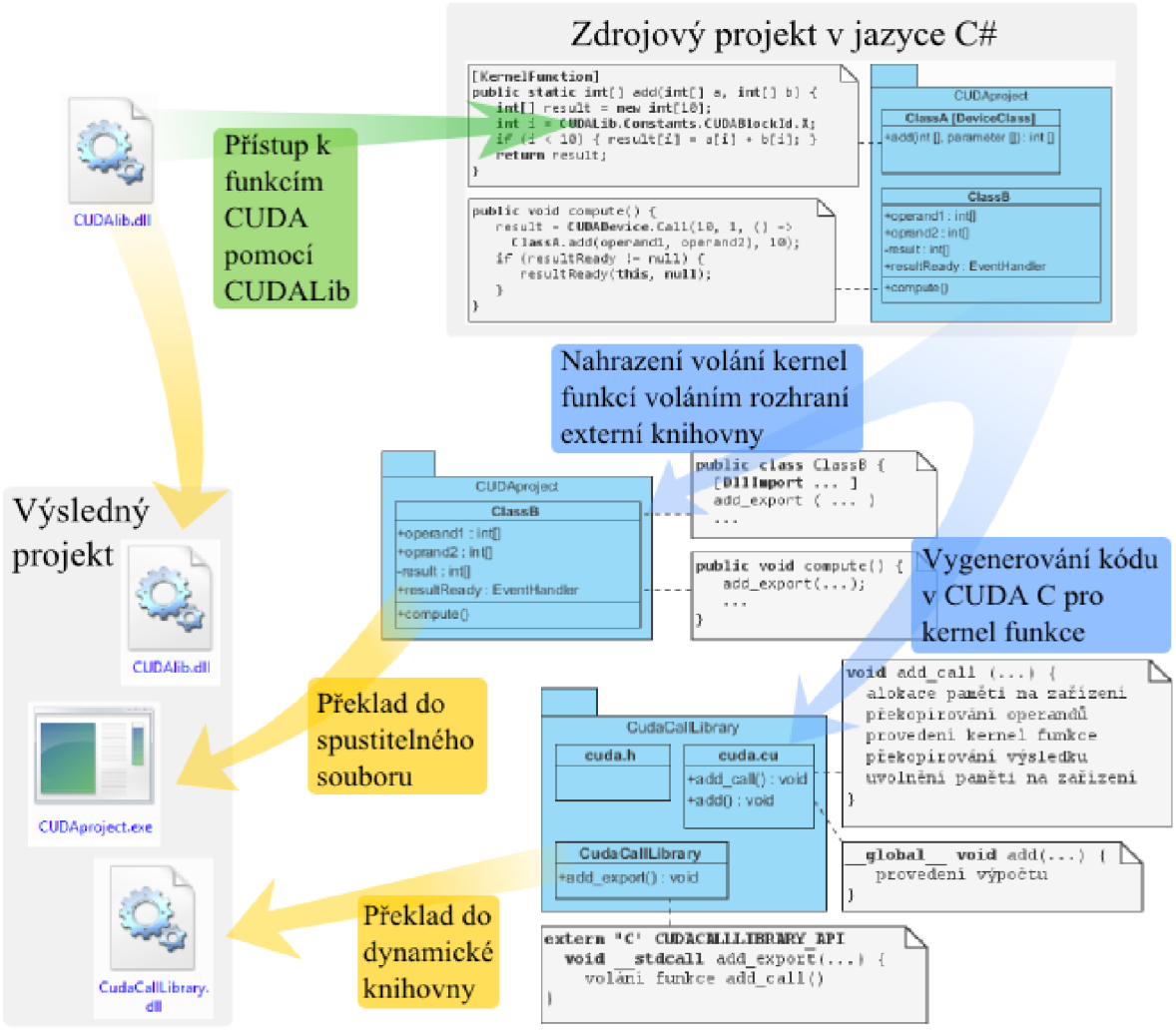
5.3 Architektura aplikace

Předpokládaný postup při práci s aplikací je následující – uživatel (programátor) napíše projekt v jazyce C#. Do zvláštní třídy, která je patřičně označena, jsou umístěny patřičně označené funkce, které jsou určeny pro běh na zařízení. Implementace této třídy probíhá za pomoci výrazových prostředků, které mají stejnou sémantiku jako v jazyce CUDA C. Nad touto třídou proběhne syntaktická a sémantická kontrola pro jazyk C#. Pokud proběhne tato analýza v pořádku, je kód rozparsován na úroveň jednotlivých instrukcí. Pro jednotlivé instrukce je vygenerován v kontextu zdrojového kódu jejich ekvivalent v CUDA C. Ten se přeloží překladačem pro CUDA C do zvláštní dynamické knihovny. Odkazy na tuto knihovnu jsou vygenerovány do míst, odkud jsou volány funkce

na zařízení. Kód určený pro běh na standartním CPU je přeložen překladačem pro C#. Dynamická knihovna s funkcemi pro zařízení je přiložena k aplikaci. Pokud všechny kroky proběhnou v pořádku, je program spuštěn s prezentací výsledků. Pokud v kterékoli části převodu dojde k chybě, je tato nahlášena, prezentována a běh převodu se zastaví.

S vědomím výše zmíněných problémů s převodem, byla aplikace rozdělena do následujících čtyř projektů, které spolu vzájemně spolupracují – knihovna (CUDALib), která poskytuje vestavěné proměnné a funkce CUDA C převedené do implementace v C#, překladač zdrojového kódu v C#, který analyzuje chyby a předá syntaktický strom generátoru. Ten jej převede odpovídající části zdrojového kódu do jazyka CUDA C, upraví zdrojový kód v C# tak, aby byl s tímto kódem kompatibilní a výsledek pošle projektu, který při úspěšnosti všech překladů provede spuštění výsledné aplikace. V rámci návrhu je tento projekt označen jako interpret, ale jeho funkce je nejen interpretace výsledků, ale i řízení celého procesu analýzy, generování, překladu a spuštění aplikace.

Princip fungování aplikace přibližuje následující obrázek [3]:



Ilustrace 3: Princip fungování aplikace [vlastní]

5.4 Návrh knihovny CUDALib

Knihovna CUDALib poskytuje ekvivalentní funkce k vestavěným funkcím CUDA C s vyšší mírou abstrakce při zachování stejné sémantiky a výpočetní síly. V rámci projektování aplikace bylo potřeba vyřešit problémy s označením definice kernel funkce, voláním kernel funkce, základními vestavěnými proměnnými a přístupem ke sdílené paměti. Knihovna byla rozdělena do čtyř modulů:

- modul pro volání kernel funkcí,
- modul pro značkování funkcí,
- modul pro práci s vestavěnnými proměnnými a funkcemi,
- modul pro podporu sdílené paměti.

5.4.1 Volání kernel funkcí

Jednou ze zvláštností CUDA C je notace konfigurace pro spuštění (<<<*PočetBloků, PočetVláken*>>>). Tuto konstrukci bylo nutné obejít zavedením voláním kernel funkce přes obalující funkci. Idea volání vypadá následovně:

```
datovy_typ vysledek =
   zavolejKernelFunkci(pocetBloku, pocetVlaken, kernelFunkce);
```

Pro převedení této notace do jazyka C# je možné využít generické funkce, které podle typu návratové hodnoty *kernelFunkce* vrací stejný návratový typ. Samotná definice *zavolejKernelFunkci* by pak zajišťovala jen syntaktický podklad, který by bylo možné použít pro volání *kernelFunkce* s definovaným počtem bloků a vláken. V okamžiku rozpoznání volání funkce *zavolejKernelFunkci* by se pak při generování rozhodlo, jakým způsobem se daná konstrukce vygeneruje.

5.4.2 Definice kernel funkce

V jazyce CUDA C jsou označeny kernel funkce pomocí klíčového slova *__global__* a mají návratovou hodnotu *void*. Vzhledem k tomu, že není možné dodefinovávat v C# klíčová slova, bylo nutné tento problém vyřešit jinak.

Jazyk C# disponuje systémem pro definování vlastních atributů, čehož bylo možné využít. Každá funkce *kernel* je označena tímto atributem. V rámci návrhu je označen tento atribut *KernelFunnction*. Pro tyto funkce musí být zavedeny zvláštní postupy. Vzhledem k tomu,

že návratová hodnota *kernel* funkce v CUDA C je *void*, je nutné toto omezení obejít a zavést například přidání parametru pro uložení návratové hodnoty při generování.

Na zařízení mohou být prováděny nejen kernel funkce, ale i normální funkce, které mohou vracet standardní návratovou hodnotu. V jazyce CUDA C jsou tyto funkce označeny podobně jako kernel funkce, pouze klíčové slovo je *__device__*. Tyto funkce nejsou určeny pro volání z hostovaného kódu, mohou být pouze volány z *kernel* funkcí, nebo z jiných *device* funkcí. Pro jejich rozpoznání v rámci návrhu aplikace jsou tyto funkce označeny atributem *DeviceFunction*.

Aplikace počítá s existencí třídy, ve které jsou funkce určené pro běh na zařízení shromážděny. Tato třída je rovněž označena zvláštním atributem (pro návrh je pojmenován *DeviceClass*) pro zvýšení čitelnosti a přehlednosti kódu. Zároveň je tento atribut použit pro hledání kernel funkcí při překladu zdrojového kódu určeného pro běh na zařízení.

5.4.3 Práce s vestavěnými atributy a funkcemi

CUDA C používá v rámci definic funkcí určených pro běh na zařízení vestavěné identifikátory pro označení výpočetní jednotky. Jsou to tří-složkové vektory: *blockIdx* pro identifikaci bloku, *blockDim* pro označní dimenze bloku a *threadIdx* pro identifikaci vlákna. Pro práci s těmito proměnnými v rámci aplikace je nutné zavést jejich ekvivalenty v jazyce C#.

Podobně jako v případě volání *kernel* funkcí jsou reprezentovány tyto proměnné jen syntaktickou konstrukcí, která se při generování rozvine do ekvivalentní formy v CUDA C. V C# lze nejlepšího chování dosáhnout pomocí statických tříd a v nich definovaných statických proměnných. Ty nemají žádný smysl jako datová úložiště, ale pouze jako reprezentace daných proměnných, proto jsou definovány jako konstantní.

Stejným způsobem je definována funkce pro synchronizaci vláken. V jazyce CUDA C se tato funkce jmenuje *__syncthreads()*. V rámci knihovny je definován jmenný prostor Synchro pro podporu volání vestavěných funkcí. Ten obsahuje třídu, jež poskytuje statickou funkci, která reprezentuje volání funkce *__syncthreads*(). Toto řešení umožňuje rozšíření množiny podporovaných vestavěných funkcí bez změny struktury knihovny.

5.4.4 Práce se sdílenou pamětí

Při běhu aplikace na zařízení lze využít paměť sdílenou mezi výpočetními jednotkami. Pro označení přístupu do sdílené paměti v používá CUDA C dvojici klíčových slov *shared __extern__*, které jsou následovány datovým typem, identifikátorem a dvěma hranatými závorkami pro označení definice pole.

Podobně jako při návrhu volání kernel funkce, je i v tomto případě při převodu využito generických vlastností jazyka C#. Zde v podobě generické třídy, jejíž typový parametr se při překladu do CUDA C převede na datový typ, který bude využit pro přístup do sdílené paměti

5.5 C# překladač

Pro parsování C# kódu je použita externí komponenta NRefactory, která je určena pro parsování zdrojových kódu napsaných na platformě .NET. Využitím této komponenty se návrh syntaktické a sémantické analýzy zjednodušil na úroveň ošetření chybových výstupů, které NRefactory používá.

Překladač byl navrhnut jako třída, která zapouzdřuje volání funkcí NRefactory a vytváří jednotné rozhraní pro parsování zdrojového kódu. Obsahuje metody pro:

- načítání obsahu zdrojového souboru do interní proměnné,
- spuštění parsování nad daným zdrojovým souborem,
- získání příznaku úspěchu parsování, který je datového typu *boolean,*
- získání souhrnu případných chyb,
- vrácení abstraktního syntaktického stromu jako výsledku parsování.

5.6 Generátor kódu v jazyce CUDA C

Projekt generátoru CUDA C tvoří hlavní jádro celé aplikace. Slouží jednak pro generování kódu určeného pro běh na zařízení, ale také k analýze syntaktického stromu dodaného projektem překladače.

Pracuje ve dvou fázích. V první fázi vyhledává kód určený pro běh na zařízení. Tento kód oddělí od ostatního a vygeneruje pro něj odpovídající ekvivalent v CUDA C. Jelikož je tento kód při překladu celé aplikace přesunut do externí knihovny, je potřeba vygenerovat přístupy ke *kernel* funkcím, a to vlastní funkci funkcí volání, hlavičkovou funkci, která ji identifikuje a rozhraní, přes které je tato funkce volána z externího projektu. Nedílnou součástí je i vygenerování kódu v C#, přes který je volána ona externí knihovna s *kernel* funkcemi.

Ve druhé fázi se hledají volání *kernel* funkcí. Ve třídách, ve kterých jsou tato volání nalezena jsou přigenerována volání externí knihovny a vlastní volání kernel funkce je nahrazeno voláním rozhraní knihovny přes konstrukci *DLLImport*.

Kvůli zapouzdření veškerých generativních akcí tvoří rozhraní celého generátoru s okolím pouze jedna třída. V rámci návrhu je pojmenována *Generator*. Zajišťuje volání vnitřních tříd, ale celý proces generování neřídí, pouze spouští akce požadované interpretem a prezentuje případné chyby a varování.

Další částí projektu tvoří jednotlivé generátory instrukcí. Podporovány jsou:

- definice funkcí, tříd a jmenných prostorů,
- podmíněná instrukce *if*, cykly *do, while, for* a instrukce *return,*
- deklarace proměnných,
- výrazy – přiřazovací, binární a unární operace, indexovací výrazy a volání funkcí.

Jelikož generovací akce jsou zapouzdřeny uvnitř projektu, je jejich činnost potřeba řídit vnitřně. Výsledkem generování by už měly být jen hotové, vygenerované a upravené zdrojové kódy nebo příznak neúspěchu. Proces generování je řízen distribuovaně z třetí části projektu, kterou tvoří manažerské třídy.

Pomocnou částí projektu jsou modelové třídy. Ty slouží jen jako datové schránky pro ukládání informací o průběhu generování. Jsou to i jednotky sdružující výsledky generování nebo modely pro fyzické úložiště zdrojových kódů.

Důležitou částí generátoru je tabulka symbolů. Ta slouží pro kontrolu definice identifikátorů v blocích ale zároveň jako úložiště informací o proměnných a jejich datových typech, včetně jejich reprezentace v CUDA C a v C#. Tuto část projektu tvoří modely pro reprezentaci úrovní zdrojového kódu a jejich částí (např. modely pro jmenné prostory, třídy a části tříd).

5.6.1 Jádro generátoru

Generování výsledného kódu je rozprostřeno do všech tříd, které reprezentují instrukce. Celý proces ale řídí instance třídy *ProcessManager*. Pokyn k začátku generování udává třída *Generator*. Následný průběh je však zcela v režii *ProcessManageru*.

Princip generování je následující: Nejprve jsou načteny všechny zdrojové kódy, které mohou obsahovat volání nebo definice funkcí určených pro běh na zařízení. Tyto zdrojové kódy jsou uloženy do modelů, které obsahují jejich syntaktické stromy a cestu k fyzickému souboru, která je později využita k uložení změn, pokud zdrojový kód obsahuje volání *kernel* funkcí.

Dalším krokem je nalezení funkcí určených pro běh na zařízení. Ty jsou umístěny ve třídě, která je označena atributem *DeviceClass*. Pokud je tato třída nalezena, jsou analyzovány funkce, zda se jedná o *kernel* nebo *device* funkce. K nalezeným funkcím jsou vytvořeny generační jednotky, ve kterých se vygeneruje kód v CUDA C pro funkci, její volání, hlavičku, rozhraní pro přístup z jiné dynamické knihovny a volání tohoto rozhraní v jazyce C#.

V posledním kroku jsou hledány volání kernel funkcí ve zdrojovém kódu a nahrazovány voláním rozhraní knihovny s *kernel* funkcemi. Využity jsou tu volání vytvořené v předchozím kroku. Poté následuje nastavení výsledků generování, a to hlášení o chybách, varování, zachycené záznamy o průběhu generování a výsledky v podobě vygenerovaných jednotek určených pro překlad CUDA překladačem a zbylého projektu přeloženého standardním překladačem.

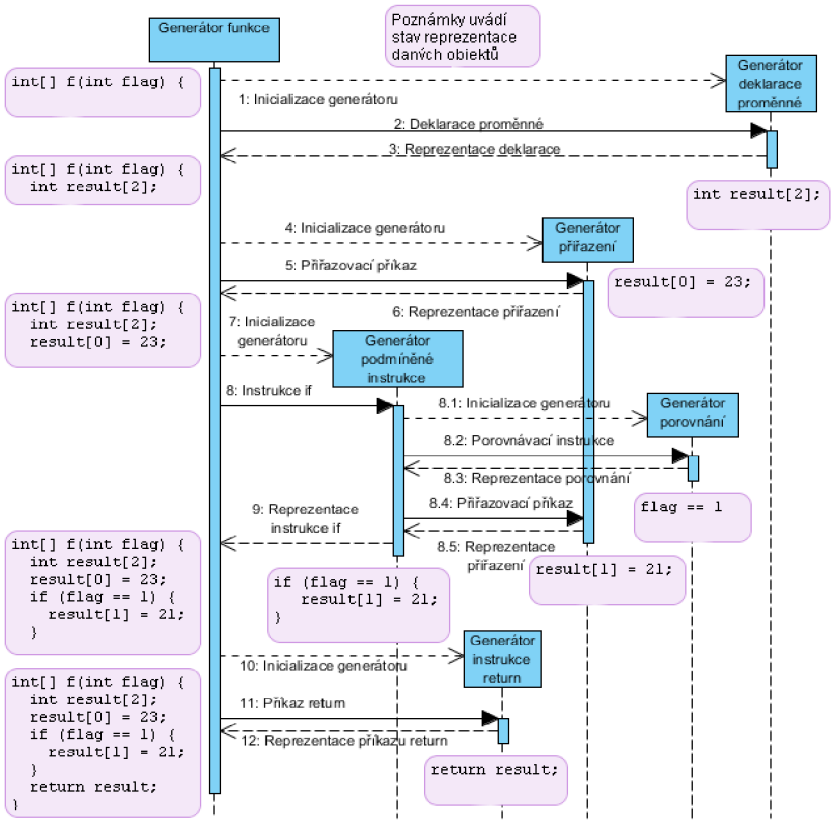
5.6.2 Průběh generování

Při generování do CUDA C jsou privátní i veřejné proměnné třídy prohlášeny za globální. Instrukce definice proměnné je předána danému generátoru a ten vytvoří její reprezentaci. Stejný generátor je použit i pro definice lokálních proměnných. Výsledná hodnota je přidána na konec reprezentace celého zdrojového souboru.

Definice funkce je předána generátoru. Ten nejprve vytvoří reprezentaci hlavičky funkce včetně parametrů. Následně je pro každou instrukci zavolán její generátor, který vrátí odpovídající kód v CUDA C. Reprezentace instrukcí jsou uloženy do reprezentace defince funkce a tato hodnota je zase předána třídě *ProcessManager*, která ji přidá k celému zdrojovému kódu v CUDA C. Tato hierarchická struktura funguje pro všechny zanořené bloky. Generování je demonstrováno na funkci:

```
int[] f(int flag)
{
    int[] result = new int[2];
    result[0] = 23;
    if (flag == 1) { result[1] = 21; }
    return result;
}
```

Průběh generování je znázorněn sekvenčním diagramem na obrázku [4].

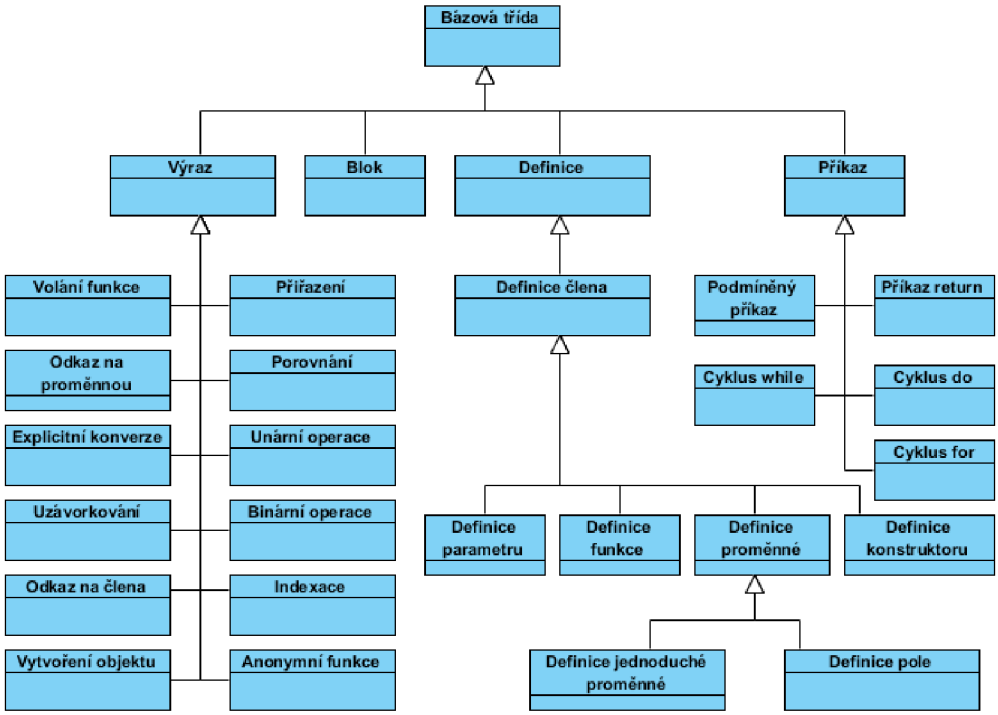


Ilustrace 4: Demonstrace generování funkce [vlastní]

5.6.3 Generátorové třídy

Pro každý typ instrukce v C# existuje vlastní generátorová třída vyjma těch, které nejsou podporovány. Rozdělení do unikátních tříd je dáno strukturou knihovny NRefactory, která ctí tento model a jejíž části syntaktického stromu jsou použity ve třídách pro udržení reprezentace v jazyce C#.

Všechny třídy, které slouží jako generátory, dědí z abstraktní bázové třídy. Ta jednak obsahuje definici proměnné pro uložení reprezentace instrukce vygenerované v CUDA C a dále abstraktní metodu, která v poděděných třídách slouží jako konvertor ze C# do CUDA C. Všechny třídy dále obsahují vlastní interní proměnnou pro uložení původní C# konstrukce uloženou v datovém typu příslušném k dané konstrukci, tak jak jej reprezentuje knihovna NRefactory.



Ilustrace 5: Konceptuální diagram tříd generátorů [vlastní]

Generátory vytvářejí hierarchickou strukturu. Nejvýše je položena již zmíněná bázová třída. Třídy, které z ní dědí, slouží jako obecné objekty pro reprezentaci skupiny instrukcí. Skupiny jsou rozděleny na výrazové generátory, definiční generátory a generátory pro kontrolní bloky jako podmíněné instrukce, cykly, bloky kódu nebo instrukce *return*.

Další skupinou, která sice patří do hierarchické struktury generátorů, ale negeneruje celé příkazy jsou generátory datových typů. Základní dělení je na jednoduché a složené typy. Pro jednoduché datové typy se provádí přímá konverze podle klíčového slova. Pro složené typy (jsou

podporovány pouze deklarace polí) se analyzuje počet dimenzí daného pole a jejich velikost. Generování typu je pak rozděleno na vygenerování klíčového slova typu a velikosti pole. Generování klíčového slova pro datový typ je řízeno manažerem datových typů, popsaného v kapitole 5.6.4.

Skupinou stojící mimo hierarchickou strukturu jsou pomocné třídy, které slouží pro generování volání funkcí. Pokud se jedná o volání funkce definované ve zdrojovém kódu, proběhne normální generování, tak jako u všech ostatních generátorů. Výjimku tvoří třída pro generování volání kernel funkcí pojmenovaná *CudaDevice* a třída *CudaMembers* pro generování vestavěných proměnných a funkcí jazyka CUDA C. Třída *CudaDevice* obsahuje metody, které generují dílčí složky průběhu volání funkce na zařízení, což je alokace paměti na zařízení, kopírování hodnot z a na zařízení a uvolnění paměti. Tyto funkce jsou dále využívány nejdůležitější částí této třídy, což je zpracování volání kernel funkcí popsaném v kapitole 5.4.1, která využívána ve druhé fázi pro generování volání kernel funkce. Třída tedy obsahuje veřejnou funkci, která přijímá parametry: počet bloků, počet vláken, velikost sdílené paměti v bytech, definice hlavičky kernel funkce a velikost výsledku. Výsledkem této funkce je vygenerovaná posloupnost příkazů, které alokují paměť na zařízení, překopírují hodnoty proměnných, zavolají kernel funkci s předaným počtem bloků a vláken, překopírují výsledek zpět a uvolní na zařízení paměť.

5.6.4 Generátorové manažery

Manažeři jsou pomocné třídy pro generování. Zajišťují funkce pro správu CUDA C identifikátorů, správu datových typů, správu referencí aktuálního zdrojového kódu nebo řízení průběhu generování popsaného v kapitole 5.6.1. Všechny třídy popsané v této kapitole jsou navrženy podle návrhového vzoru jedináček.

Manažer identifikátorů pro CUDA C ve spolupráci s manažerem referencí se stará o rozpoznání identifikátorů definovaných v knihovně CUDALib, které mají být nahrazeny voláním vestavěné CUDA funkce nebo proměnné. Její funkce jsou rozděleny do dvou skupin: ty které slouží pro rozpoznání identifikátorů a ty, které slouží pro generování příslušné konstrukce. Druhá skupina funkcí využívá k vlastnímu generování funkce ze třídy *CudaMembers*.

O kontrolu a převody datových typů se stará další manažerská třída. Jelikož C# má například jen pro pojmenování datového typu *Integer* čtyři možné tvary, bylo nutné sjednotit deklaraci daného typu a vyskytnul-li by se jakýkoliv z tvarů, tak jej převést na reprezentaci podle jazyka CUDA C. Pro implementaci zdrojových kódů běžících na zařízení jsou povoleny ordinální datové typy a typ *float* s plovoucí řádovou čárkou.

Manažer kernel funkcí je další třída typu jedináček, která zajišťuje kontrolu nad definovanými kernel funkcemi. V okamžiku kdy je rozpoznána ve zdrojovém kódu definice funkce, automaticky se

kontroluje, zda je tato funkce opatřena atributem, který by ji deklaroval jako funkci určenou pro běh na zařízení. Pokud tomu tak je, funce je uložena manažerem do interní enumerované proměnné v podobě generovací CUDA jednotky popsané následující kapitolou. Pokud je při generování rozpoznáno volání kernel funkce, musí být nejprve zkontrolováno, zda taková funkce existuje, což je právě manažer, který tuto informaci poskytuje. Tato třída poskytuje zároveň funkce pro export vygenerovaných kernel funkcí, jejich volání, rozhraní pro jejich volání z externí knihovny a hlavičky, k čemuž využívá modelů generovacích jednotek.

Proces generování monitoruje manažer záznamů. Před zahájením generovacího procesu je vymazána jeho interní paměť, která sdružuje informace o hlášených problémech. Po skončení je to tato třída, ze které se získávají informace o průběhu generování. Součástí manažeru je enumerátor, který určuje závažnost hlášení v podobě informace, varování nebo chyby.

Každý zdrojový kód v C# obsahuje reference na další objekty, které jsou v kódu použity. Pro tyto informace existuje také manažer, který sbírá odkazy na jmenné prostory na začátku každého zdrojového kódu. Nalezený odkaz nejprve rozdělí na jednotlivá zanoření do cílového prostoru podle tečkové notace. Toto pole řetězců následně ukládá do veřejného listu, který je využíván například manažerem identifikátorů pro CUDA C. Poskytuje funkci, která umožňuje mapování daného parametru v podobě pole řetězců na existující jmenný prostor. Výsledkem je příznak nalezení daného jmenného prostoru.

Poslední manažerskou třídou je manažer pro tabulku symbolů. Ten poskytuje okamžitý přístup k aktuálně zpracovávanému jmennému prostoru a aktuálně zpracovávané třídě. Poskytuje též rozhraní pro nastavení a ukončení zpracovávání nového jmenného prostoru.

5.6.5 Pomocné modely

Nejmenší část projektu tvoří pomocné modely pro reprezentaci různých typů dat v průběhu generování. Kromě modelu generovací jednotky slouží tyto třídy jen jako datová úložiště bez vnitřní logiky. První je model reprezentující výsledek překladu zdrojového kódu, který obsahuje kompilační jednotku NRefactory se syntaktickým stromem, přístupovou proměnnou k fyzickému umístění zdrojového kódu a řetězec pro uložení výsledného kódu v jazyce C# po úpravách spojených s voláním kernel funkcí. Další je notifikační model, který slouží manažeru záznamů k uchování informací o průběhu generování. Obsahuje proměnné pro uložení zprávy, souboru kde bylo hlášení vygenerováno, řádku a znaku.

Nejdůležitější je model generovací jednotky pro jazyk CUDA C. Ten sdružuje informace o nalezených kernel metodách. Jsou v něm uloženy informace o třídě, ve které byla funkce nalezena, o hlavičce funkce, jejích parametrech a návratové hodnotě. Dále poskytuje funkce, které k uložené kernel metodě vygenerují kód v C# pro definici rozhraní k externí knihovně a volání tohoto rozhraní.

Dále vygenerují kód v CUDA C pro hlavičku volání kernel funkce, volání samotné a funkci fungující jako rozhraní, ale ze strany knihovny. Pro samotnou kernel funkci zde též existuje exportovací funkce, ale ta zabaluje už jen volání generátorové třídy s odpovídajícími parametry.

5.6.6 Tabulka symbolů

Zajišťování kontroly platnosti proměnných a sdružování informací o nich zajišťuje tabulka symbolů. Ta obsahuje datové modely pro ukládání jednotlivých typů dat do hierarchické struktury. Datové modely se dělí na tři skupiny: modely pro abstraktní reprezentaci hierarchické struktury (tabulka symbolů), modely pro reprezentaci části struktury zdrojového kódu v jazyce C# (jako třída, jmenný prostor atd.) a modely pro reprezentaci datového typu (jednoduchý *integer*, pole atd.).

Všechny modely z poslední skupiny jsou odvozeny z abstraktní bázové třídy, která poskytuje uložení reprezentace typu v jazyce C# a uložení základního typu, k čemuž využívá manažer pro správu typů.

Základem modelů pro reprezentaci struktury je opět abstraktní bázová třída. Obsahuje proměnné pro uložení jména, typu a úrovně zanoření. Typ symbolu může být: proměnná, metoda, parametr, třída a jmenný prostor. Pro reprezentaci každého typu existuje zvláštní třída. Model funkce navíc obsahuje informace o parametrech, návratovém typu a příznak, zda se jedná o kernel funkci. Tuto informaci využívá manažer pro správu kernel funkcí. Modely pro reprezentaci třídy a jmenného prostoru pak obsahují odkazy na své tabulky symbolů.

Nejvýše stojí modely pro reprezentaci hierarchické struktury. Základem je model pro definici bloku. Používá se uvnitř tříd a slouží pro ukládání identifikátorů na jednotlivých úrovních zanoření (např. pro definici funkce, definice cyklu vevnitř funkce, definice podmíněného příkazu vevnitř cyklu atd.). Agregačně výš stojí model pro reprezentaci tabulky symbolů na úrovni třídy. Obsahuje zásobník modelů pro definici bloku a zabaluje volání funkcí modelu pro definici bloku stojícího v zásobníku nejvýše. Rovněž umožňuje startovat a ukončovat platnosti jednotlivých bloků. Hierarchicky nejvýše stojí model pro reprezentaci jmenného prostoru, který obsahuje kolekci modelů tříd, jež jsou v něm definovány

5.7 Interpret výsledků

Vstupní bod celé aplikace tvoří interpret výsledků. Jeho návrh je zaměřen na jednoduchost a použitelnost uživatelského rozhraní. Aplikace by měla sloužit jako jednoduché vývojové prostředí, ve kterém je možné otevírat projekty, upravovat zdrojové kódy jednoduchou formou a překládat a spouštět hotové projekty.

Vzhledem k implementaci v C# bylo rozhodnuto implementovat uživatelské rozhraní pomocí technologie WPF, jelikož nabízí velkou varietu možností jak nadefinovat grafické rozhraní. Projekt je rozdělen do několika modulů. Vzhledem k tomu, že bylo využito návrhového vzoru MVVM, byl projekt rozdělen na datové modely, pohledy a modely pro pohledy.

5.7.1 Návrhový vzor MVVM

Návrhový vzor MVVM (Model – View – ViewModel) je určen pro aplikace programované v technologii WPF (Windows Presentation Foundation). Základem je oddělení aplikační logiky od uživatelského rozhraní, čehož je dosaženo striktním vymezením určenosti jednotlivých částí. Obecně je platná konvence, že je přínosnější definovat více jednoduchých tříd, které agregační vazbou tvoří hierarchickou strukturu, než mít méně tříd, které dělají více věcí. Této konvence se hojně využívá ve všech třech úrovních MVVM a je podporována například formou vnořování pohledů nebo navazováním dat na uživatelské rozhraní do zanořených modelů formou tzv. *bindování*.

První je model, který obsahuje referenci na zdroj dat. Může to být obalovací třída nad voláním webové služby, databázová entita nebo jakýkoliv další typ datového zdroje. Tyto třídy jsou velmi jednoduché a neobsahují žádnou aplikační logiku.

Nejrozsáhleší jsou modely pro pohledy (ViewModel v konvenci MVVM). Spojují dohromady pohledy a datové modely, provádí nad daty operace a prezentují hotová data do uživatelského rozhraní. Je vyžadováno, aby datové modely neměly žádnou referenci na pohledové modely, a ty zase aby neměly referenci na uživatelské rozhraní. Dodržení tohoto schématu usnadňuje změny uživatelského rozhraní i výměny zdroje dat.

Prezentační vrstvu tvoří pohledy (View). Programovacím jazykem pro grafické rozhraní je jazyk XAML (Extensible Application Markup Language) založený na XML. Je možné psát rozhraní i pomocí jazyka C#, ale přehlednost zdrojového kódu upřednostňuje XAML. Rovněž se upřednostňuje odklon od událostmi řízeného programování a využívají se spíše automatické změny obsahu pohledu použitím modelu, který implementuje rozhraní *INotifyPropertyChanged.* V případě akčních prvků je upřednostňována implementace příkazů a chování (třídy implementující rozhraní *ICommand* nebo potomci třídy *Behavior*). Další informace o tomto návrhovém vzoru lze nalézt v [21], [22], [23] a [27]

5.7.2 Návrh uživatelského rozhraní

Při návrhu uživatelského rozhraní byl kladen důraz na jednoduchost a použitelnost. Důležitý je i prvek toho, aby bylo prostředí uživateli automaticky známé. Rozvržením i barevným schématem bylo tedy rozhraní inspirováno produktem MS Visual Studio 2010, což je nejrozšířenější vývojové

prostředí pro implementaci aplikací v jazyce C#. Základní dělení je na panel s prezentací stromové struktury otevřeného projektu, panel pro editaci zdrojových kódů a panel pro prezentaci chybových či varovných hlášení. Všechny modely pro pohledy jsou odvozeny od bázové třídy, která imlementuje rozhraní *INotifyPropertyChanged*. Tato třída poskytuje další funkce pro vynucení obnovení uživatelského rozhraní a naplnění novými daty,

Základem je samotné okno aplikace, jež tvoří podklad, do kterého se vkládají další pohledy. To určuje rozdělení rozhraní do panelů. Modelem pro tento pohled je řídící třída, které zaštituje akce celého uživatelského rozhraní. Obsahuje modely pro další pohledy (pro všechny panely) a určuje který panel je viditelný, který zdrojový kód je otevřen atd.

Model pro prezentaci otevřeného projektu pracuje se stromovou strukturou. K jejímu naplnění využívá datové modely, odvozené od třídy, která poskytuje vlastnosti pro uložení jména a příznaku, zda je položka stromu příslušná k tomuto modelu vybrána. Každý tento model reprezentuje soubor nebo adresář otevřeného projektu. Tyto modely jsou pak uspořádány do kolekce, která je předána jako datový kontext grafické komponentě, které je zobrazí v nadefinované formě. Součástí projektového modelu je podpora otevírání, ukládání, překladu a spuštění projektů.

Každý zdrojový kód projektu může být otevřen pro editaci. K tomu je využíván středový panel, který obsahuje záložky s jednotlivými otevřenými soubory. Model pro pohled pro zobrazení zdrojového kódu je velmi jednoduchý. Obsahuje reprezentaci souboru v podobě jeho modelu použitého ve stromové struktuře řetězec, ve kterém je uložen jeho obsah. Součástí je i podpora zavírání a ukládání záložky s pohledem s tímto modelem.

Výsledky překladů jsou prezentovány v panelu na pravé straně. Ten obsahuje záložky pro prezentaci chyb, varování a hlášení o průběhu, a to jak formou listu jednotlivých hlášení i prostého textu. Všechny pohledy, které se vyskytují v tomto panelu mají za model třídu, která zaštiťuje jak hlášení ve formě listu nebo textu. Způsob prezentace hlášení je předán v konstruktoru a po celou dobu života třídy se nemění. Při zobrazení hlášení formou listu využívá datového modelu, který obsahuje zprávu, soubor ke kterému zpráva náleží a v případě, že je tato informace známa, tak i řádek a znak (pro případ prezentace chyb v překladu).

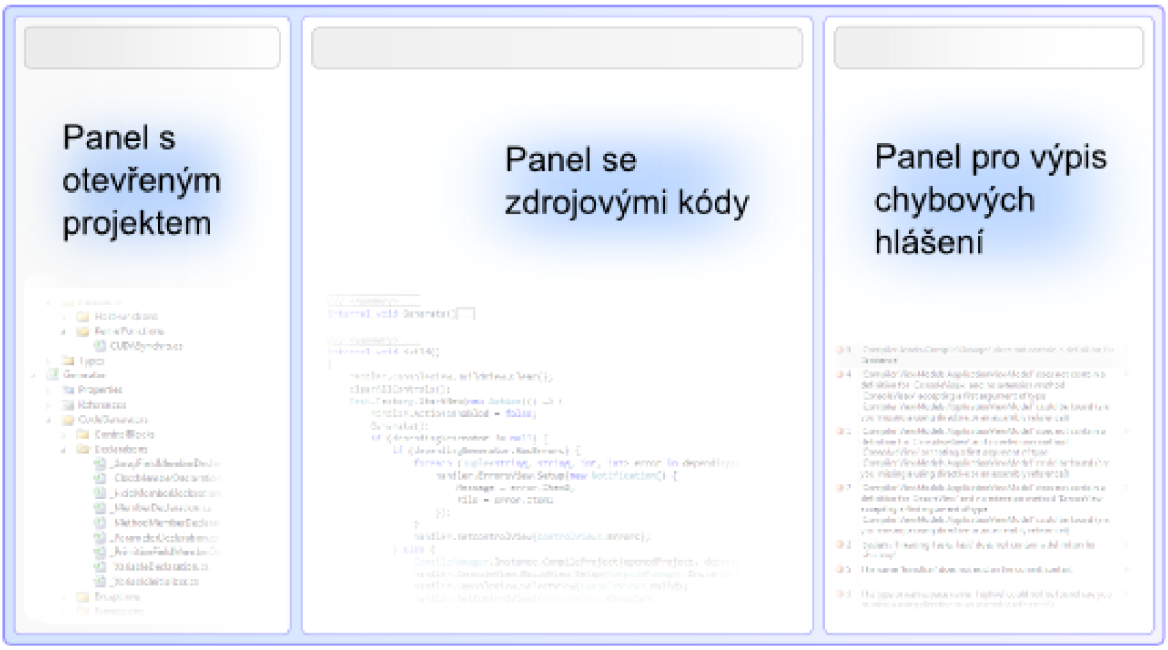
5.7.3 Jádro interpretu

Podobně jako v případě projektu generátoru tvoří hlavní motor projektu manažerské třídy navržené podle vzoru jedináček. Jsou určeny pro správu akcí nad generováním, překladem a spuštěním aplikace. Všechny dědí z abstraktní bázové třídy, která poskytuje rozhraní pro získání chyb, varování nebo průběhu jednotlivých fází.

Prvním je manažer pro správu generování. K zadanému projektu (ve formě modelu ze stromové struktury) vygeneruje pomocí projektu generátoru výsledné zdrojové soubory. Zároveň převezme od projektu generátoru chybová a varovná hlášení.

Tuto třídu využívá další manažer, a to ten, který má na starosti řízení překladu projektu. Navenek má pouze jednu veřejnou metodu, pomocí které přeloží knihovnu s funkcemi určenými pro běh na zařízení a nakopíruje ji k výslednému projektu, který rovněž přeloží. Funkce tohoto manažeru je volána z pohledového modelu pro prezentaci projektu, který nejprve nechá nagenerovat zdrojové kódy jedním manažerem a následně je nechá přeložit druhým manažerem jako reakci na stisk tlačítka pro překlad.

Ze stejného modelu je využíván i poslední manažer, který se stará o spuštění projektu. Manažer pouze obdrží projektový model ze stromu, získá cestu ke spustitelnému souboru přeloženého projektu a spustí jej. Stejně jako v minulých dvou případech analyzuje případná chybová hlášení a předává je modelům pro zobrazení chyb.

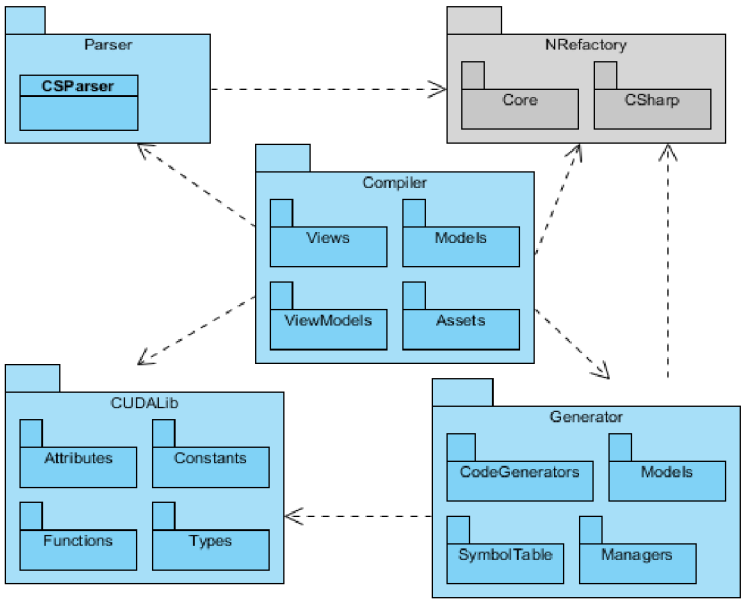


Ilustrace 6: Návrh uživatelského rozhraní [vlastní]

6 Implementace aplikace

Tato kapitola popisuje postupy a techniky, které byly použity při programování aplikace. Je zde přiblížena architektura aplikace z hlediska implementačního jazyka. Vzhledem k tomu, že byla aplikace vyvíjena v jazyce C#, bylo pro implementaci použito integrované vývojové prostředí Visual Studio .NET Framework 2010 Enterprise Edition od společnosti Microsoft.

Při implementování aplikace bylo vycházeno z návrhu popsaného v minulé kapitole. Aplikace se tedy dělí do čtyř projektů. Celou aplikaci tvoří jeden Microsoft Studio Solution, který obsahuje tři třídní knihovny – *CUDALib*, *Generator*, *Parser* a projekt zajišťující grafické rozhraní pro uživatele – *Compiler*. Všechny projekty kromě knihovny *CUDALib* dále referencují dvě externí dynamické knihovny *NRefactory Core* a *NRefactory CSharp*. Situaci dokumentuje diagram balíčků na obrázku číslo [7].



Ilustrace 7: Architektura aplikace popsaná diagramem balíčků [vlastní]

6.1 Knihovna CUDALib

Knihovna CUDALib podle návrhu slouží jako poskytovatel vestavěných konstrukcí, které poskytuje jazyk CUDA C, ale nejsou v základu podporovány jazykem C#. Knihovna tedy podporuje definice CUDA atributů, vestavěných funkcí, proměnných a konstant v syntaxi jazyka C#. Zvláštností všech tříd, které se nacházejí v této knihovně je, že jejich funkce ani vlastnosti neobsahují žádný výkonný

kód, veškeré proměnné či funkce slouží jen jako syntaktické konstrukce a využívají se jen k reprezentaci konstrukce, která je dále zpracovávana generátorem.

Visual Studio projekt knihovny CUDALib je implementován jako knihovna tříd (*Class Library*). Nereferencuje žádnou další knihovnu kromě základních knihoven .NET Frameworku. Všechny jeho třídy jsou veřejné a projekt může vystupovat i jako samostatná knihovna. Základní jmenný prostor CUDALib obsahuje čtyři další jmenné prostory – *CUDALib.Attributes*, *CUDALib.Constants, CUDALib.Functions* a *CUDALib.Types*. Žádný z těchto podprostorů nevyužívá jiný, jsou zcela samostatné.

Nejmenší ze všech je jmenný prostor *CUDALib.Constants*, který obsahuje jednu třídu. Ta je statická a nese název *CUDAConst*. Obsahuje pouze dvě veřejné konstanty typu *int* a to *BlockIdx* a *ThreadIdx*, které slouží pro reprezentaci stejnojmenných vestavěných proměnných v jazyce CUDA C.

6.1.1 Definice vlastních atributů

Vlastní atributy, které se používají pro označování funkcí, jsou definovány v tomto jmenném prostoru. Pro definice atributů bylo použito odvození od základní třídy *Attribute*, která se používá k označování konstrukcí v jazyce C#. Od této třídy dědí třídy *DeviceFunctionAttribute*, a *HostFunctionAttribute*. Pro obě tyto třídy je definováno omezení použití pomocí systémového atributu *System.AttributeUsage(System.AttributeTargets.Method)* tak, že jsou tyto atributy aplikovatelné pouze na funkce. Obě třídy mají veřejný bezparametrický konstruktor a nemají žádné privátní proměnné ani metody.

6.1.2 Syntaxe volání kernel funkce

Jmenný prostor *CUDALib.Functions* obsahuje třídu *CUDADevice*. Ta neobsahuje konstruktor a není ani instanciovatelná, jelikož je definována jako abstraktní. Obsahuje veřejnou, statickou funkci s názvem *Call*, sloužící k reprezentaci volání kernel funkce z kódu. Její definice vypadá následovně :

```
public static TResult Call<TResult>
    (int blockCount, int threadCount, Expression<Func<TResult>> deviceFunction)
{    return default(TResult);    }
```

Jednoduchý příklad jejího volání může vypadat takto:

```
[KernelFunction]
int[] getBlockIdx() {    return new int[10];    }

public void main()
{
   int[] result; result = CUDADevice.Call(10, 5, () => getBlockIdx(), 10);
}
```

Tato reprezentace znamená žádost o spuštění kernel funkce *getBlockIdx()* v deseti blocích a pěti vláknech bez žádného vstupního parametru s návratovou typu *int*. Funkce využívá objekt typu *Expression* k identifikaci volání funkce, která má návratovou hodnotu typu *TResult*, který se zároveň tak stává návratovým typem samotné funkce *Call*. Pokud je tato syntaktická konstrukce rozpoznána generátorem, je nagenerováno odpovídající volání v CUDA C, včetně alokace paměti a kopírování proměnných z a na zařízení, blíže popsané v kapitole 6.3. Posledním parametrem je velikost pole, které se vrací jako výsledek. Funkce je dále přetížena o jeden další parametr, kterým se udává velikost sdílené paměti v *bytech*.

6.1.3 Přístup ke sdílené paměti

Další generické vlastnosti jazyka C# byly využity při implementaci konstrukce pro přístup ke sdílené paměti. Lze použít generickou třídu, jejíž typový parametr může nabývat hodnot ordinálních datových typů nebo typů s plovoucí řádovou čárkou. Tato třída byla nazvána *Shared* a nachází se ve jmenném prostoru *CUDALib.Types*.

Vzhledem k tomu, že pro přístup ke sdílené paměti v CUDA C se používají pole, byl přetížen indexovací operátor pro tuto třídu, a to tak, že pro uživatele vrací hodnotu uloženou v datovém typu, daným typovým parametrem třídy *Shared* na indexu dané celočíselnou hodnotou. Implementačně jsou přetíženy metody *get* a *set* vlastnosti *this[int i]*. Metoda *get* vrací základní hodnotu odvozenou od datového typu. Metoda *set* má prázdné tělo.

6.2 Knihovna C# překladače

Překladač je v rámci implementace prototypu nejjednodušší projekt. Obsahuje jedinou třídu, která nese název *CSParser*. Zde je hlavní využití knihovny NRefactory, neboť knihovna překladače v podstatě jen zaštiťuje volání překladu, prováděné knihovnou NRefactory. V konstruktoru třídy *CSParser* se vytváří instance třídy *CSharpParser*, definované ve jmenném prostoru *ICSharpCode.NRefactory.CSharp*, která se ukládá do privátní proměnné třídy *CSParser*.

Nejdůležitější funkce je *Parse(string name)* bez návratové hodnoty, která načte obsah souboru daný cestou k souboru předanou v parametru. Pak zavolá instanci NRefactory *CSharpParseru* a jeho funkci *Parse()* nad daným souborem. Výsledek ukládá do privátní proměnné typu *CompilationUnit*. Ta je přístupná pomocí veřejné vlastnosti, která má definovaný pouze veřejný *getter*. Aby bylo možné prezentovat případná chybová hlášení vzniklé při překladu, jsou nadefinovány další vlastnosti pro získání chyb a varování a dvě příznakové vlastnosti, které udávají zda se při překladu vyskytla chyba či varování. Varování i chyby nejsou prezentovány pomocí modelu, ale datovým typem n-tice:

Tuple<string, string, int, int>, kde první řetězec ukládá zprávu, druhý soubor, ve kterém došlo k chybě, celočíselná hodnota na třetí pozici udává řádek a čtvrtá znak.

6.3 Projekt Generátoru CUDA C kódu

Knihovna *Generator* představuje nejrozsáhlejší projekt celé aplikace. Hierarchicky nejvyšší jmenný prostor *Generator* je rozdělen do čtyř podprostorů *CodeGenerators, Managers, Models* a *SymbolTable*. Samotný jmenný prostor *Generator* obsahuje jedinou veřejnou třídu z celého projektu, jenž se jmenuje *CUDACGenerator*.

Tato třída tvoří rozhraní projektu s okolím. Všechny veřejné funkce, které tato třída poskytuje jsou pouze zabalená volání vnitřních tříd bez jakékoli vlastní logiky. Jsou to zejména funkce pro reset vnitřního stavu knihovny před zahájením dalšího generování, funkce pro přidání další jednotky v podobě syntaktického stromu knihovny NRefactory a nebo funkce pro zahájení procesu generování. Všechny tyto funkce využívají rozhraní manažeru pro řízení průběhu generování (třída *ProcessManager*). Součástí třídy CUDACGenerator jsou, stejně jako v případě překladače jazyka C#, vlastnosti, které prezentují pro interpret průběh překladu v podobě listů chyb. Použitý je stejný datový typ n-tice se stejnou sémantikou jednotlivých částí.

6.3.1 Řízení generovacích akcí

Vnitřní řízení generovacích akcí mají na starosti manažerské třídy, implementované podle návrhového vzoru jedináček. Hlavní třídou, která řídí celý proces je ProcessManager, i když na generování se podílejí významnou měrou všechny třídy.

ProcessManager poskytuje veřejné funkce, které jsou využívány rozhraním projektu, třídou *CUDACGenerator*. Z pohledu posloupnosti akcí při generování je první funkcí *Reset()*. Ta volá všechny manažerské třídy a volá na nich funkci se stejným názvem i sémanikou – uvedení manažeru do stavu pro start nového generování. Zároveň vnitřně tato funkce inicializuje list pro plnění generovacími jednotkami v podobě modelů třídy *CompilationGeneratorUnit*. Tento list je naplněn další funkcí volanou z *CUDACGeneratoru* a to *IncludeUnit*.

Po naplnění všech jednotek generovaného projektu je generovací proces zahájen stejnojmennou funkcí *Process()* a má dvě fáze. To jaká fáze právě probíhá je dáno nastavením vlastnosti *Status*, která odkazuje na proměnnou enumerovaného typu *ProcessStatus* a může nabývat hodnot *GenerateCudaC* nebo *UpdateCSCode*. Podle nastavení této vlastnosti se mění chování některých generátorů.

V první fázi (*GenerateCudaC*) se hledají všechny kernel funkce. Implementačně to znamená volání funkce *FindAndProcessCudaDefinitions* na generátorové třídě *_Namespace*. Funkce je volána na všech nalezených jmenných prostorech. Jejím úkolem je pro všechny třídy v daném jmenném

prostoru detekovat, zda se jedná o třídu s atributem *DeviceClass*. Pokud je tento atribut nalezen přichází na řadu další manažer a to *DeviceFunctionsManager*. Jeho úkolem je sbírat nalezené funkce určené pro běh na zařízení. K tomu využívá vnitřní list modelů třídy *CudaMethodGeneratorUnit* pro reprezentaci funkce vygenerované v CUDA C. K nalezené funkci se ukládá specifikace třídy, ve které byla nalezena, specifikace parametrů a návratového typu, vygenerovaný kód v CUDA C a typ funkce (může být kernel nebo obyčejná funkce určená pro běh na zařízení daná atributem *DeviceFunction*).

Po nalezení všech funkcí určených pro grafickou kartu nastává druhá fáze, kdy se hledají volání nalezených kernel funkcí. Z *ProcessManageru* je spouštěna pro každý jmenný prostor funkce *FindAndProcessCudaCalls*. Ty hledají skrz všechny třídy a jejich funkce přiřazovací příkazy, na jejichž pravé straně se vyskytuje volání funkce *CUDADevice.Call()* indikující volání kernel funkce. Ve třídách, kde jsou tato volání nalezena, se přigeneruje konstrukce *DLLImport* pro definici rozhraní s externí knihovnou, do které budou vloženy kernel funkce (detailně popsáno v kapitole 6.3.4), a konstrukce pro volání kernel funkce je nahrazena voláním rozhraní (viz kapitola 6.3.2). Upravený C# kód je uložen do modelu *CompilationGeneratorUnit*, který odpovídá aktuálně zpracovávané jednotce. Tyto jednotky jsou pak použity pro export C# kódu do *CUDACGeneratoru* a následně až do projektu interpretu, který je zpracuje.

6.3.2 Generátorové třídy

Implementace generátorů je vázána návrhem popsaným v kapitole 5.6.2. Byla dodržena struktura tříd, zaznamenaná konceptuálním diagramem tříd na obrázku [5]. Hierarchicky nejvýš je postavena třída *_Base*, ze které dědí všechny generátory, protože obsahuje definice řetězce *representation*, který je využíván jako textová reprezentace dané C# konstrukce pro každý generátor. Dále je zde deklarace abstraktní metody *process()* s návratovou hodnotou datového typu *string*, která je v dceřiných třídách určena pro analýzu a zpracování dané instrukce. Součástí je i potlačení standardní definice funkce *ToString()*, která je pro každý generátor upravena tak, aby vracela obsah proměnné *representation*.

Jmenný prostor generátorů je dále dělen na další podprostory, které zahrnují generátory pro určité oblasti konstrukcí jazyka C#. Jsou to tedy prostory pro kontrolní bloky a příkaz *return* (*ControlBlocks*), deklarace funkcí a proměnných (*Declarations*), prostor zahrnující syntaxi výrazů (*Expressions*), zvláštní prostor pro definice volání funkcí (*Invocations*) a mimo stojící prostor pro reprezentaci datových typů (*Types*).

Každý z těchto prostorů kromě *Invocations* obsahuje abstraktní třídu, která zaštiťuje volání kterékoliv konstrukce z daného prostoru. Například pro prostor *ControlBlocks* existuje abstraktní třída *_Statement*. Ta poskytuje statickou funkci *ProcessStatement(Statement statement)*, kde datový typ

Statement je obecný typ NRefactory pro reprezentaci příkazu. V této funkci je podle typu instance parametru rozhodnuto, který generátor se vytvoří a zpracuje daný příkaz. Funkce instancuje generátory pro převod tříd, jmenných prostorů, cyklů, bloků příkazů a podmíněného příkazu.

Stejným způsobem vystupují abstraktní třídy *_Expression* (ze jmenného prostoru *Expressions*, se statickou funkcí *ProcessExpression(Expression expression)*) a *_Type* (ze jmenného prostoru *Types*, se statickou funkcí *ProcessType(Type type)*). V rámci aplikace je rozděleno generování výrazů do tříd pro převod přiřazovacího příkazu, binární operace, použití identifikátoru, indexovací operátor, odkaz na objekt a jeho vlastnosti či funkce, vytváření objektu, použití konstanty nebo provedení unární operace.

Ze schématu implementace tříd zpracovávajících výrazy vystupuje generátor pro přiřazovací příkaz, jelikož v rámci omezení aplikace je to jediná konstrukce, která dovoluje volání kernel funkce a přiřazení výsledné hodnoty do proměnné s výsledkem. Podle nastavení proměnné Status ze třídy ProcessManager se chová rozdílně. V případě, že probíhá fáze hledání a generování CUDA C kódu, je tento generátor zapínán pouze v tehdy, kdy se jedná o přiřazovací příkaz uvnitř kernel nebo device funkce. Konstrukce v jazyce C# se tedy převede odpovídajícím způsobem na CUDA C, což zahrnuje volání *_Expression.ProcessExpression()* pro každou stranu přiřazovacího příkazu, jelikož se může jednat o vnořené výrazy. V druhém případě je nejprve kontrolována pravá strana příkazu, zda se nejedná o volání funkce *CudaDevice.Call()*. K této analýze se používá manažerská třída *CudaIdentifiersManager*. Ta obsahuje syntaktické zápisy všech konstrukcí z knihovny CUDALib a porovnává je s nacházenými identifikátory. Porovnávací funkce této třídy se dále využívají například v generátorech pro reference na objekty nebo volání funkce. Pokud je tedy nalezeno volání kernel funkce, tak je tato část syntaktického stromu vyjmuta a nahrazena objektem typu *InvocationExpression,* který zajišťuje volání rozhraní externí knihovny (viz kapitola 6.3.4)

Všechny ostatní generátory ze všech podprostorů mají jinak stejnou strukturu – veřejný konstruktor, který má za parametr instanci datového typu reprezentující danou konstrukci pro potřeby knihovny NRefactory, dále privátní proměnnou, jež udržuje tuto konstrukci, a potlačení funkce *process()*, ve které každá třída provede překlad své instrukce v C# do CUDA C.

6.3.3 Generování kernel funkce

Generátory obsažené v prostoru *Invocations* stojí mimo hierarchickou strukturu generátorových tříd. Nejvýznamějším z nich je *CudaDevice*. Ten slouží pro generování funkcí spojených s grafickou kartou. Jsou to funkce *GenerateKernelCall*, *GenerateCudaMalloc*, *GenerateCudaMemcpy* a *GenerateCudaFree*. Tři poslední slouží jen pro generování jednořádkových volání stejnojmenných funkcí s patřičnými parametry. Nejrozsáhlejší funkcí je *GenerateCudaCall*. Ta přijímá počet bloků

a vláken, jméno kernel funkce, list jejích parametrů a název proměnné, do které se má uložit výsledek. Postup generování do CUDA C je následující:

- Pro každý parametr kernel funkce se vygeneruje deklarace pomocné proměnné, která bude sloužit pro uchování obsahu parametru v globální paměti grafické karty. Pro tyto proměnné se vygeneruje funkce *cudaMalloc*, která zajistí alokaci paměti na zařízení. Následně je vygenerována funkce *cudaMemcpy*, která překopíruje obsah parametru do nově deklarované proměnné na zařízení.
- Vygeneruje se pomocná proměnná pro vrácení výsledku, pro kterou se taktéž alokuje místo na zařízení.
- Volání kernel funkce se generuje s nově vytvořenými proměnnými a přidanou proměnnou pro vrácení výsledku. Vše v syntaxi CUDA C, takže s konfigurací pro spuštění, tedy s počtem bloků a vláken v notaci tří ostrých závorek na obou stranách.
- Pak je přidána fukce *cudaMemcpy*, která vykopíruje obsah pomocné proměnné pro vrácení výsledku do skutečné proměnné, předané v parametru funkce *GenerateCudaCall*.
- Generování končí připojením funkce *cudaFree* pro všechny pomocné proměnné – obsahy všech parametrů a výsledku.

Příklad generování lze nalézt v příloze [1].

6.3.4 Modelové třídy

Jednotky, které se používají pro uschování informací pro potřeby generování jsou umístěny do jmenného prostoru *Generator.Models*. Je zde také umístěna třída *Notification*, jež slouží jako datová schránka pro hlášení chyb, varování. Používá ji další z manažerů, který slouží pro zachycování průběhu generování, s názvem *LogManager*. Po ukončení generování je list modelů *Notification* obsahující zaznamenané události převeden na list n-tic *Tuple<string,string,int,int>* (popsaný v kapitole 6.3), který je použit dále třídou *CUDACGenerator* pro export těchto dat pro projekt interpretu.

Dalším, významějším, modelem je *CompilationGeneratorUnit*. Tato třída představuje jeden zdrojový kód projektu v jazyce C#. V konstruktoru této třídy je předán abstraktní syntaktický strom, který byl vytvořen projektem překladače v podobě datového typu *NRefactory – CompilationUnit*. Při druhé fázi generování, je do vlastnosti *ResultSourceCode* uložen upravený kód v jazyce C#, kde jsou vytvořeny definice *DLLImport* a volání kernel funkcí jsou nahrazena voláním tohoto rozhraní. Po ukončení generovacího procesu jsou tyto jednotky využity opět rozhraním třídy *CUDACGenerator* k exportu upravených zdrojových kódů pro interpret formou datového typu *Dictionary<string, string>*, kde první řetězec je cesta ke zdrojovému souboru a druhý obsahuje

překopírovaný obsah proměnné *ResultSourceCode* z odpovídající jednotky. Cesta ke zdrojovému souboru je ve třídě *CompilationGeneratorUnit* obsažena též.

Nejvýznamějším modelem je *CudaMethodGeneratorUnit,* který uzavírá celý generovací proces kernel a device funkcí. Je to jednotka, která se používá pro uložení nalezených kernel funkcí. To je však její méně významná funkce. Tou významnější je sjednocení všech částí kódu, které je potřeba vygenerovat pro kernel funkci, pod jedinou třídou, ze které se pak exportují. Třída obsahuje funkce:

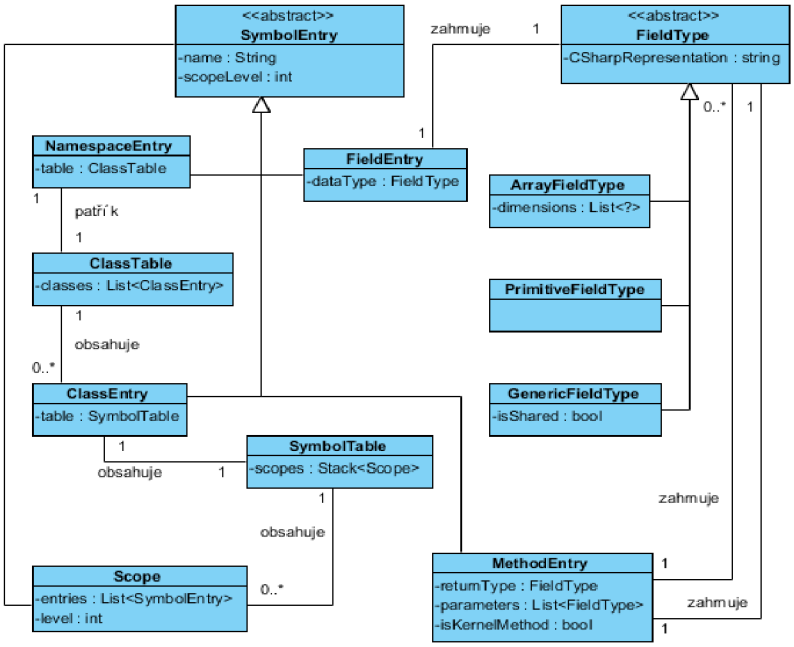
- *GenerateDLLImport* – vygeneruje konstrukci *DLLImport* ve formě řetězce v jazyce C#, jež se při exportu třídy, ve které se volání kernelu nachází, přidá pod hlavičku třídy.
- *GenerateDLLImportCall* – generuje volání konstrukce *DLLImport* v jazyce C# v závislosti na předaných parametrech funkce, konfigurace pro spuštění a identifikátoru výsledku.
- *GenerateKernelCall* – generuje volání kernel funkce, které je použito pro volání kernelu na straně externí knihovny, tato funkce využívá generátoru *CudaDevice* a její funkci *GenerateKernelCall()*. Vygenerovaný obsah obaluje do volací funkce.
- *GenerateHeader* – volací funkce musí být přístupná třídě, která zajišťuje komunikaci knihovny s okolím. K tomu je potřeba vygenerovat hlavičku do zvláštního souboru, k čemuž se používá tato exportovací funkce.
- *GenerateApi* – generuje rozhraní pro volání kernel funkce ze strany externí knihovny.

Všechny tyto funkce jsou použity pro export manažerem *DeviceFunctionManager*. Export probíhá tak, že se prochází všechny uložené jednotky a do jednoho řetězce se exportují všechny hlavičky, do dalšího všechny volání a kernel funkce a do dalšího api funkce (v manažeru jsou to funkce *ExportKernelsWithCalls(), ExportKernelExports()* a *ExportKernelHeaders()*). Manažerské funkce jsou vystaveny do třídy *CUDACGenerator*, aby mohly být exportovány do projektu interpretu.

6.3.5 Tabulka symbolů

Pomocnou strukturu pro ukládání informací o nalezených identifikátorech tvoří tabulka symbolů situovaná do jmenného prostoru *Generator.SymbolTable*. Třídy v této části generátoru se dělí do 3 skupin.

První skupinu tvoří datové třídy pro uložení reprezentace datového typu (ve smyslu primitivní proměnná, pole atd.). Třídy z této skupiny jsou odvozeny od abstraktní třídy *FieldType*. Ta obsahuje abstraktní vlastnost *CSharpRepresentation* pro uložení reprezentace datového typu v jazyce C#. Zároveň obsahuje konstruktor, jenž musí být volán ze všech dceřiných tříd a obsahuje uložení základního datového typu *(int, float* atd.) do vnitřní proměnné. Odvozené třídy slouží k uložení primitivního typu, pole (má navíc list, který udržuje informace o dimenzích pole) a generického typu (používá se při generování konstrukce *SharedArray<T>* pro přístup do sdílené paměti).



Ilustrace 8: Diagram tříd pro tabulku symbolů [vlastní]

Další skupinou jsou třídy odvozené od abstraktní bázové třídy *SymbolEntry*. Tyto modely slouží pro uchování informací o jednotlivých typech identifikátorů – pro proměnnou, metodu, třídu nebo jmenný prostor. Třídy využívají agregační vazby na třídy založené na typu *FieldType*. U proměnných se používají k uchování jejich datového typu. U metod pak k uložení typu návratové hodnoty a parametrů. Vztahy mezi třídami jsou zobrazeny v diagramu tříd na obrázku [8].

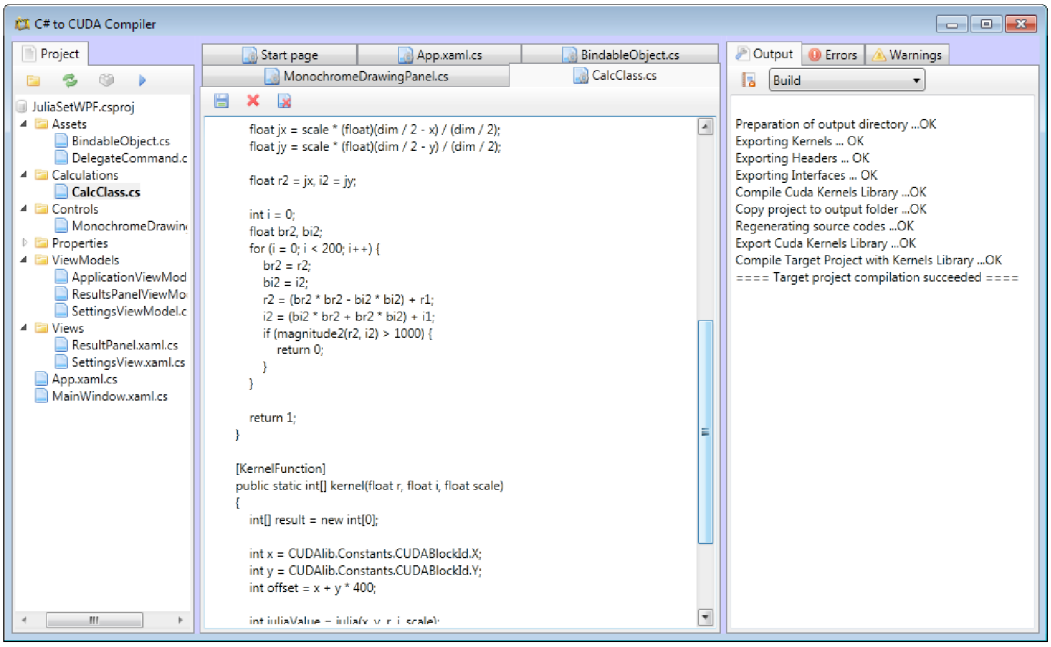
Hierachickou strukturu z pohledu vnoření úseků kódu završuje třetí skupina modelů. Zde je na nejnižší úrovni třída *Scope*. Ta se používá pro rozlišení zanoření bloků kódu uvnitř tříd a obsahuje list tříd poděděných od *SymbolEntry*, které představují seznam identifikátorů platných pro danou úroveň. Zásobník těchto bloků obsahuje instance třídy *SymbolTable*, která se používá pro ukládání všech identifikátorů dané třídy. Na nejvyšší úrovni je třída *ClassTable*, jež obsahuje list všech tříd (a jejich identifikátorů) ve jmenném prostoru, ke kterému tato tabulka náleží.

Pro urychlení přístupu k úrovním tabulky symbolů se využívá manažerské třídy *SymbolTableManager,* která udržuje všechny zpracované jmenné prostory formou slovníku. Třída poskytuje odkazy na aktuálně zpracovávaný jmenný prostor a aktuálně zpracovávanou třídu. Součástí je i nastavení počátku a konce platnosti zpracovávaného jmenného prostoru. Zároveň je zde jako, ve všech manažerech, funkce *Reset(),* jež v případě této třídy smazává obsah slovníku s uloženými jmennými prostory.

6.4 Uživatelské rozhraní

Projekt interpretu výsledů figuruje mezi ostatními projekty celé aplikace jako rozhraní s uživatelem. Je postaven jako aplikace typu WPF. Jeho základem tedy není funkce Main, ale definice aplikace v syntaxi jazyka XAML, která je načítána ze souboru App.xaml s nezbytným podporujícím kódem v *App.xaml.cs*. Obsahuje jen odkaz na definici hlavního okna, čímž je soubor *MainWindow.xaml*.

V projektu je dodrženo schéma návrhu popsaného kapitole 5.7, zejména pak je kladen důraz na implementaci grafického rozhraní pomocí návrhového vzoru MVVM. Projekt tedy obsahuje jmenné prostory pro datové modely (Compiler.Models), modely pro pohledy (Compiler.ViewModels) a samotné pohledy (Compiler.Views). Součástí jsou ji nezbytné manažerské a pomocné třídy umístěné ve jmenném prostoru Compiler.Assets. Použité grafické prvky se pak nacházejí ve složce se zdroji (Resources). Výsledné uživatelské rozhraní je zobrazeno na ilustraci [9].



Ilustrace 9: Uživatelské rozhraní projektu interpretu [vlastní]

6.4.1 Hlavní okno aplikace

Okno aplikace vystupuje ze zavedené adresářové struktury, není umístěno do složky s pohledy, ale je v kořenovém adresáři. Má ovšem svůj model, třídu ApplicationViewModel, která zároveň tvoří jádro uživatelského rozhraní a řídí odpovídající volání na uživatelské akce.

Ten obsahuje odkazy na jednotlivé modely pro pohledy, které jsou použity v hlavním okně. Jsou modely pro pohled projektů (*ProjectViewModel*), jednotlivé modely pro otevřené zdrojové kódy dané kolekcí, která poskytuje události pro změny kolekce (*ObservableCollection<TabViewModel>),* model pro konzoli hlášení (*ConsoleViewModel*) a modely pro výpis chyb a varování se stejným typem *NotificationViewModel*. Hlavní model obsahuje také funkce pro zobrazení dialogů pro otevření projektu a otevření souboru projektu, jelikož to vyžaduje součinnost modelu projektu a otevření nové záložky do kolekce otevřených zdrojových kódů. Zároveň je zde funkce *SetControlView()* pro nastavení aktuálně otevřené kontrolní záložky (myšleno chyby, varování nebo výpis konzole).

Zdrojový kód pro hlavní okno obsahuje rozdělení aplikačního okna na tři části – modul pro prezentaci projektu, modul pro editaci zdrojových kódů a modul pro prezentaci chyb a varování. Každý tento modul tvoří WPF komponenta *TabControl* ze systémového jmenného prostoru *System.Windows.Controls*. V případě modulu pro prezentaci projektu obsahuje záložku, s odkazem na komponentu *<views:ProjectView />*, s definovaným datovým kontextem, který odkazuje přes konstrukci binding do hlavního modelu na instanci třídy *ProjectViewModel.* Grafické rozhraní pro editaci zdrojových textů otevřeného projektu je dáno datovými šablonami (v jazyce C# *DataTemplate),* které jsou namapovány do hlavního modelu na zmíněnou kolekci modelů *TabViewModel.*

Tato třída je základní třídou pro dva další modely – *StartPageViewModel* a *SourceCodeViewModel.* Šablona pro určitou záložku se vybírá pomocí datového typu pohledu, k tomu slouží *PageTemplatesSelector,* definovaný ve jmenném prostoru *Compiler.Assets*. Šablona úvodní strany obsahuje pouze nabídku akcí při otevření aplikace, jako je otevření projektu. Šablona zdrojového kódu je velmi jednoduchá a nabízí v podstatě jen komponentu *TextField,* pro editaci zdrojového kódu a nástrojovou lištu, která umožňuje uložení, otevření zdrojového kódu atd. Tyto tlačítka jsou nabindovány do modelu typu *SourceCodeViewModel,* který pro odchytávání uživatelských akcí nad tlačítky využívá koncept delegátních příkazů (stejný koncept je využit i u tlačítek v komponentě projektu a jejího model typu *ProjectViewModel*).

Tento koncept se využívá místo klasické události tlačítka *OnClick,* protože poskytuje možnost ovládání přístupnosti tlačítka z modelu. Vynechání režie, která souvisí s vytvořením objektu události, taktéž přineslo zrychlení. V projektu je tento koncept použit ve třídě *DelegateCommand* (jmenný prostor *Compiler.Assets*), která implementuje rozhraní *ICommand.* Toto rozhraní obsahuje hlavně funkce *Execute()* a *CanExecute().* Do konstruktoru třídy jsou předány odkazy na akce, které se mají provést, když je stisknuto tlačítko a když se ověřuje, zda je tlačítko možné stisknout – tyto akce jsou pak namapovány na zmíněné dvě funkce z rozhraní *ICommand.*

6.4.2 Zobrazení, překlad a spuštění projektu

Šablona pro zobrazení otevřeného projektu obsahuje definici komponenty pro zobrazení stromové struktury (*TreeView*). Datový kontext této komponenty je namapován do modelu *ProjectViewModel* na kolekci objektů *TreeItemModel.* Tato abstraktní třída je základem pro všechny modely, které slouží k zobrazení stromové struktury projektu. Přímo z třídy *TreeItemModel* dědí dva modely: *ExpandableTreeItemModel* a *ProjectFileModel.* Druhý jmenovaný slouží k zobrazení listů stromu v podobě jednotlivých zdrojových souborů a kromě jména a cesty, jež jsou definovány už v základní třídě, poskytují pouze informaci o příponě souboru. *ExpandableTreeItemModel* slouží k zobrazení uzlů stromu, k tomu obsahuje kolekci dětských uzlů opět typu TreeItemModel. Jsou z něj poděděny dvě dceřiné třídy: *ProjectModel* (slouží jako kořenový uzel projektu) a *FolderModel* (zobrazení vnitřních složek v projektu).

Jádrem projektu interpretu je datový kontext tlačítek na liště šablony pro projekt. Jsou to tlačítka *Build* a *Execute,* které jsou navázány na delegátní příkazy modelu *ProjectViewModel.* První má jako akci pro kliknutí na tlačítko definovánu funkci *Build().* Ta využívá nový způsob přístupu k multivláknovému programování, zavedený poprvé v .NET Frameworku 4 a to systém *Task.* Ten zaručuje bezpečné chování vláken, zachytávání výjimek, rušení vláken a příjem chyby v proměnné typu *CancellationToken* a zamezení pádu aplikací díky výjimce typu *ThreadAbortException.* Pomocí konstrukce *Task.Factory.StartNew()* je tedy spuštěno nové vlákno, na kterém se spojují všechny tři části této aplikace popsané v minulých kapitolách.

Jako první jsou vygenerovány zdrojové kódy v CUDA C a odpovídajícím způsobem upraveny závisející kódy v jazyce C#. K tomu je využívána třída *GeneratorManager* a její funkce *Process*(). Jí je v argumentu předán celý strom projektu v podobě modelu ProjectModel, který obsahuje odkazy na všechny zdrojové kódy projektu. Funkce odfiltruje všechy jiné soubory než v jazyce C# a uloží je do listu. Třída obsahuje vnitřní proměnné typu *CUDACGenerator* a *CSParser,* tedy rozhraní projektů generátoru a překladače jazyka C#. Generátor nastaví do výchozího stavu zavoláním jeho funkce *Reset().* Pak prochází zdrojové kódy projektu a nad soubory s jazykem C# zavolá funkci překladače *Parse*(), čímž vytvoří do jeho vnitřní proměnné syntaktický strom daného zdrojového souboru. Ten je pak předán generátoru funkcí *IncludeUnit().* Po načtení je na generátoru zavolána funkce *Process(),* čímž se provede vše potřebné (viz kapitola 6.3.1). Případné chyby a varování se ukládají do vnitřních proměnných v podobě listů modelu *Notification* (obsahuje informace o zprávě, souboru, řádku a znaku, kde došlo k chybě). Generátor zpřístupňující výsledná data je pak předán dalšímu manažeru, a to třídě *CompileManager.*

Ten se stará o kopírování částí projektu na správná místa při překladu projektu. Jeho součástí je odkaz na připravený projekt v jazyce C++, který je použit jako šablona pro externí knihovnu, do které

budou umístěny funkce určené pro běh na zařízení. Projekt je umístěn ve složce *Resources/CudaCallLibrary.* Překlad projektu je pak rozdělen na části:

- Vyčištění cílové složky - v projektu je vytvořena, případně vyčištěna složka s názvem *Output,* ve které bude umístěn výsledný projekt.
- Export kernel funkcí – na generátoru je zavolána funkce *GetKernels(),* která vyexportuje všechny *device* a *kernel* funkce i s jejími voláními v CUDA C ve formě řetězce (generátor volá svůj manažer *DeviceFunctionsManager* a jeho funkci *ExportKernelsWithCalls(),* kapitola 6.3.4). Tento řetězec je předán funkci, která jej nakopíruje do souboru *cuda.cu* projektu externí knihovny.
- Export hlaviček pro volání kernel funkcí – podobně jako výše s tím rozdílem, že se volá funkce generátorová funkce *GetHeaders()* a obsah je nakopírován do souboru *cuda.h.*
- Export rozhraní – podobně jako výše, volá se funkce *GetKernelExports().* Obsah je přidán do souboru *CudaCallLibrary.cpp,* předtím je mu však ještě předřazen zdrojový kód s definicí aplikačního rozhraní celé knihovny, který je uložen ve zvláštním souboru ve složce *Resources.*
- Překlad externí knihovny – využitím překladače *msbuild* je přeložena externí knihovna. Projekt použitý jako šablona má upravenou konfiguraci, aby zahrnul základní systémové knihovny CUDA C, a byl přeložen pod architekturou V80 s nastavením překladače pro CUDA 4.0.
- Úprava C# souborů – otevřený projekt je nakopírován do složky *Output.* Na generátoru je zavolána funkce *GetUnits(),* která vrátí upravené zdrojové soubory v jazyce C#. Podle nastavených cest jsou soubory výsledného projektu nahrazeny těmi upravenými.
- Nakopírování externí knihovny – přeložená dynamická knihovna s kernel funkcemi je nakopírována do adresáře, kde bude veden překlad výsledného projektu.
- Překlad projektu – projekt je přeložen pomocí překladače *msbuild.*

Po úspěšném překladu je zpřístupněna funkce pro spuštění. Ta využívá poslední manažer projektu interpretu – třídu *ExecuteManager*. Té je do funkce *RunOuputProject()* předán v parametru opět kořen otevřeného projektu v podobě modelu *ProjectModel.* Je vytvořen nová instance systémové třídy *Process*, které je do konfigurace předáno jméno a cesta ke spustitelnému souboru výsledného projektu. Zavoláním funkce *Start()* se proces spustí.

6.4.3 Hlášení chyb

Každý z manažerů prezentuje nalezené chyby a varování v podobě listu modelu *Notification*. Tyto listy jsou předány hlavnímu modelu (instance třídy *ApplicationViewModel*), který podle závažnosti

zobrazí určitý pohled. Pokud se v hlášení vyskytují chyby, zobrazí záložku pohledu ErrorsView, pokud se vyskytují varování, zobrazí záložku pohledu *WarningsView* a pokud vše dopadlo v pořádku, je zobrazena záložka *ConsoleView*, která informuje pouze o průběhu.

Obě záložky pro hlášení chyb a varování využívají stejný typ pohledu, a to třídu *NotificationViewModel*, jež vystavuje nalezené problémy formou kolekce modelů *Notification*, které jsou zde importovány z manažerů. Tato třída rovněž může prezentovat hlášení textovou formou, a to pomocí proměnné *Output*. Toho využívá *ConsoleViewModel* (model pro pohled *ConsoleView*), který obsahuje tři proměnné typu *NotificationViewModel* a prezentuje průběh generování, překladu a spuštění.

6.5 Testování aplikace

Aplikace byla podrobena zevrubnému testování, zaměřeného na ověření funkčnosti jednotlivých částí systému. Testování spočívalo ve spouštění sady referenčních příkladů a ověřování výsledků. Příklady byly brány z knihy Cuda by Example[9] od Jasona Sanderse.

Referenční příklady byly vybrány z částí knihy, jejichž tématika je pokryta v implementovaném řešení. Zahrnuje základní práci s definicí funkcí určených pro běh na zařízení a techniku jejich volání. Testováno bylo zejména implementování algoritmů v základní syntaxi jazyka C#, definice proměnných, použití cyklů a podmíněných příkazů. Součástí bylo i testování pokročilých technik jako přístup ke sdílené paměti nebo synchronizace vláken. Zároveň byla aplikace otestována ve tvorbě rozsáhlejších aplikací, jejichž součástí je grafické rozhraní.

Aplikace byla testována na několika desítkách aplikací. Skupiny testovaných projektů jsou popsány v podobě tří referenčních projektů z knihy Cuda by Example. Jsou to nejjednoduší aplikace typu součtu vektorů demonstrujících základní použití, rozsáhlejší aplikace s grafickým rozhraním, které využívají technologie CUDA a aplikace s podporou pokročilých konstrukcí.

6.5.1 Součet vektorů

Prvním referenčním příkladem je součet dvou vektorů (v knize je uveden pod kapitolou 4.2.1). Jedná se o nejjednodušší použití technologie CUDA, které lze implementovat. Příklad je určen hlavně pro demonstraci použití aplikace a programování paralelních algoritmů s použitím knihovny CUDALib.

Projekt *VectorSum*, který zahrnuje testovací aplikaci, obsahuje pouze 2 zdrojové soubory. Prvním z nich je vstupní bod projektu, třída *Program* obsahující funkci *Main*. V ní jsou inicializovány dva celočíselné vektory jako operandy a výsledek. Následuje volání *kernel* funkce *AddVectors()* pomocí konstrukce *CUDADevice.Call*() s nastavenou konfigurací pro spuštění v deseti blocích, každý s jedním vláknem. Pro tento příklad není inicializována žádná sdílená paměť. Funkci ukončuje výpis výsledků do konzole s čekáním na uživatelský vstup.

Druhým souborem je třída určená pro běh na zařízení s názvem *CalcClass*. Obsahuje jedinou *kernel* funkci, a to již zmíněnou *AddVectors* (patřičně opatřenou atributem *KernelFunction*), která přijímá dvě celočíselné pole a vrací výsledek uložený rovněž do celočíselného pole. Po inicializaci výsledku je do lokální proměnné uložen identifikátor aktuálního bloku pomocí reference konstanty

CUDABlockId.X z knihovny *CUDALib*. Tato proměnná je použita k indexaci obou vektorů, jejichž součet je uložen na stejný index do proměnné výsledku.

Projekt je velmi jednoduchý, ale demostruje základní způsoby práce s technologií CUDA a knihovnou *CUDALib*. Překlad tohoto projektu v interpretu má za následek správné vytvoření výsledného projektu ve složce *Output*. Po jeho spuštění jsou vektory správně sečteny a je zobrazen výsledek.

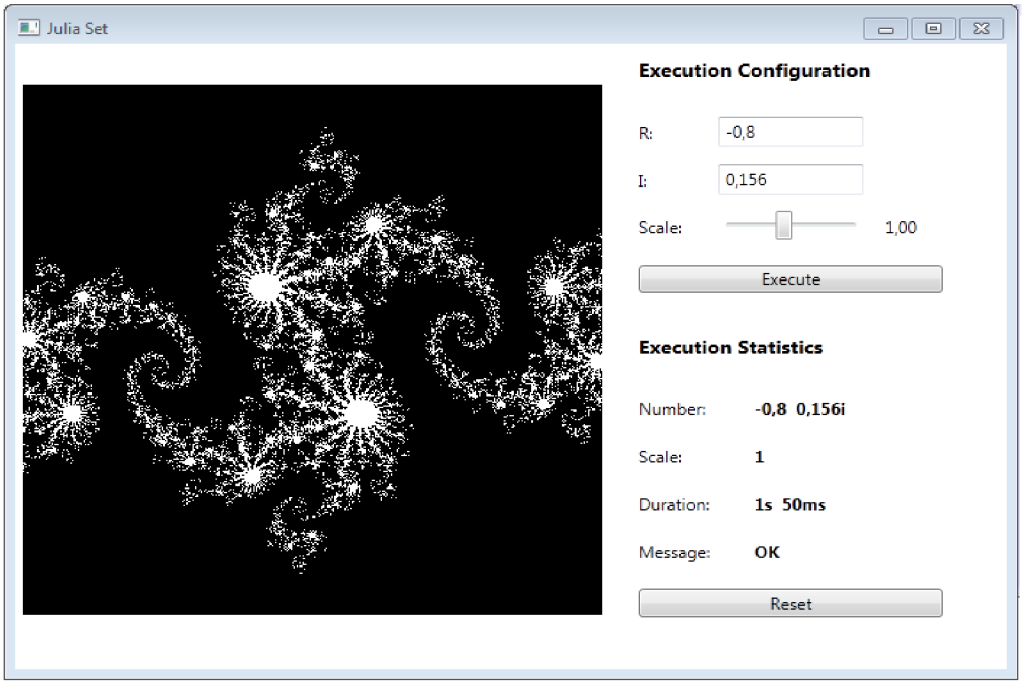
6.5.2 Výpočet Juliovy množiny

Dalším projektem je implementace příkladu na výpočet Juliovy množiny z kapitoly 4.2.2. Juliova množina je množina všech bodů z v komplexní rovině, pro které posloupnost

$$z_{n+1} = z_n^2 + c,$$

kde c je libovolné komplexní číslo, nediverguje. Hranice této množiny tvoří tzv. fraktál, což je geometrický objekt, který je soběpodobný (v jakémkoli měřítku má stále stejný motiv, který se opakuje) a je generován opakovaným použitím jednoduchých pravidel.

Tento projekt je zaměřen na demonstraci spojení technologie CUDA s grafickým rozhraním implementovaným ve WPF. Demonstruje změnu tvaru Juliovy množiny v závislosti na měnících se parametrech daných reálnou a imaginární složkou komplexního čísla. Při implementaci byl opět využit návrhový vzor MVVM.



Ilustrace 10: GUI aplikace pro výpočet Juliovy množiny [vlastní]

Aplikace je rozdělena do několika jmenných prostorů:

- *Assets* – pomocné třídy pro definice příkazů a načítání dat,
- *Calculations* – obsahuje třídu *CalcClass*, která sdružuje *kernel* funkce,
- *Controls* – komponenty uživatelského rozhraní, v tomto případě panel, který umožňuje rychlé vykreslování Juliovy množiny,
- *ViewModels* – modely pro pohledy,
- *Views* – pohledy uživatelského rozhraní.

Třída *CalcClass* obsahuje funkci *kernel*(), která přijímá tři parametry typu *float*: *r* (reálná složka komplexního čísla), *i* (imaginární složka) a *scale* (faktor přiblížení Juliovy množiny). Vzhledem k tomu, že funkce vrací pole čísel, reprezentující náležitost bodu do množiny, je vypočítán index, na kterém bude každá jednotka pracovat pomocí souřadnice x a souřadnice y bloku, která je navíc vynásobena s velikostí mřížky dané proměnnou *CUDAGridDim.X*. Pro výpočet hodnoty na tomto indexu volá každý blok funkci *julia*(), která pomocí souřadnic x a y daného bodu předaných v parametrech vypočítává náležitost bodu do Juliovy množiny.

```
[DeviceFunction]
public static int julia(int x, int y, float r1, float i1, float scale)
{
     float r2 = scale * (float)(dim / 2 - x) / (dim / 2);
     float i2 = scale * (float)(dim / 2 - y) / (dim / 2);
     float br2, bi2;
     for (int i = 0; i < 200; i++) {
         br2 = r2; bi2 = i2;
         r2 = (br2 * br2 - bi2 * bi2) + r1;
         i2 = (bi2 * br2 + br2 * bi2) + i1;
         if ((r2 * r2 + i2 * i2) > 1000) {  return 0; }
     }
     return 1;
}
```

Z definice funkce *julia*() vyplývá, že souřadnice x a z jsou použity k inicializaci komplexního čísla z ze vzorce uvedeného výše popsaného reálnou složkou (*r2*) a imaginární složkou (*i2*). Argumenty *r1* a *i1* představují komplexní číslo c. V několika desítkách iterací je následně rozhodnuto, zda číslo z patří do Juliovy množiny, pokud jeho velikost překračuje nastavený práh.

Vše je prezentováno pomocí uživatelského rozhraní, které je složeno z dvou pohledů: *ResultPanel*, který obsahuje komponentu pro vykreslení Juliovy množiny, a *SettingsView*, jenž obsahuje textová pole a posuvník pro editaci komplexního čísla, které vstupuje do výpočtu.

6.5.3 Demonstrace sdílení paměti a synchronizace vláken

Projektem, který představuje pokročilou práci s pamětí a vlákny, je výpočet skalárního součinu z kapitoly 5.3.1. Demonstruje použití objektu *SharedArray<T>* pro přístup k paměti sdílené mezi vlákny v rámci bloku a použití objektu *CUDASynchro* pro synchronizaci těchto vláken. Tvoří ho dvě třídy – vstupní třída *Program* s metodou *Main* a třída *CalcClass* pro uchování kernel funkcí.

Součástí třídy *Program* je funkce *sumSquares(),* která slouží pro ověření výsledku. Její zápis byl vytvořen podle teorie uzavřené diferenciální formy, která vektorovým polím přiřazuje skalární funkci. Je volána z funkce *Main* a její výsledek je uložen do lokální proměnné *sum*. Ve funkci *Main* jsou nejprve inicializovány vstupní operandy, v tomto případě n-rozměrné vektory. Prvky prvního vektoru jsou inicializovány indexem, na kterém prvek leží, prvky druhého vektoru pak dvojnásobkem jeho indexu. Po nastavení počtu bloků, vláken a velikosti sdílené paměti je zavolána funkce *CUDADevice.Call*(), jenž volá *kernel* funkci *dot*(). Ta vrací skalární součin vstupních vektorů na nultém indexu pole výsledku. Tato hodnota je porovnána s obsahem proměnné *sum* a výsledek je zobrazen. Pro dvacet testovaných hodnot se výsledky obou funkcí vždy shodovaly.

V kernel funkci *dot*() je po inicializaci výsledné proměnné *result* definován přístup ke sdílené paměti pomocí proměnné *cache*, jež je datového typu *SharedArray<float>*. Nastaven je též index pro přístup do sdílené paměti (proměnná *cacheIndex* je inicializována identifikátorem vlákna). V první fázi výpočtu si každé vlákno spočítá mezisoučet prvků operandů a tento výsledek následně uloží do sdílené paměti na index daný proměnnou *cacheIndex*. V tomto okamžiku je potřeba zajistit, aby všechna vlákna spočítala svůj mezisoučet před dalším pokračováním algoritmu. To je garantováno použitím funkce *CUDASynchro.SynchronizeThreads(),* která je přeložena do CUDA C instrukce *__syncthreads()*. Tato konstrukce zajistí, že všechna vlákna počkají na tomto místě dokud probíhá v některém z nich nějaká činnost. Po uložení všech mezivýsledků je provedena redukce mezisoučtů na jeden výsledek pomocí cyklu while. Řídící proměnná cyklu je inicializována číslem, které představuje polovinu velikosti vstupního vektoru a v každé iteraci cyklu je toto číslo děleno dvěma. Situaci ilustruje následující úryvek zdrojového kódu:

```
int i = CUDABlockDim.X / 2;
while (i != 0) {
   if (cacheIndex < i) {
        cache[cacheIndex] = cache[cacheIndex] + cache[cacheIndex + i];
    }
    CUDASynchro.SynchronizeThreads();
    i = i / 2;
}
```

Všechna vlákna s nastavenou proměnou *cacheIndex* menší, než je polovina vstupního pole, vypočítají svou novou hodnotu jako součet svého prvku a prvku na pozici *cacheIndex* plus polovina

vstupního pole. Následně jsou vlákna synchronizována, aby byly zapsány všechny mezivýsledky. V dalším kroku se využívá již jen první poloviny těchto vláken, jelikož další již nejsou potřeba. V závěru algoritmu je tedy výsledek uložen ve sdílené paměti na pozici 0 a tato hodnota je předána jako výsledek celé funkce.

7 Závěr

Technologie CUDA mě zaujala z mnoha pohledů. Tím největším se stala rychlost výpočtu proti centrálnímu procesoru. Architektura aplikace, způsoby implementace akcelerace, ale i samotné principy fungování grafických karet pro mě byly velkou novinkou a přínosem, který si odnáším z tohoto projektu.

V úvodu práce jsem popsal teoretické zázemí, které je nutné vědět pro práci na akcelerování výpočtů na grafické kartě. S těmito znalostmi v kombinaci s principy existujících řešení jsem sestavil návrh aplikace, který byl nejlepším možným kompilátem všech aspektů, které jsem při práci zohlednil. Využíval jsem jeden z nejmodernějších programovacích jazyků současnosti, ve kterém jsem se hodně zdokonalil.

Výsledná aplikace je koncipována jako výukový systém pro programátory, kteří pracují pod platformou .NET a chtějí akcelerovat své algoritmy pomocí grafické karty. K akceleraci je použita technologie CUDA, zejména pak její možnosti paralelizace, využití sdílené paměti a synchronizace vláken. Součástí výsledné aplikace je interpret výsledků, který příjemnou a intuitivní formou umožňuje upravovat, překládat a spouštět projekty implementované v jazyce C#.

Aplikace podporuje omezenou množinu datových typů, které je možné použít pro implementaci algoritmů na grafické kartě. Též některé validní algoritmy mohou být vyhodnoceny jako chybné, díky optimalizačním technikám, které používá překladač jazyka CUDA C. V mnoha případech jsou omezení dána možnostmi, které podporuje technologie CUDA pro implementaci programů, zejména kvůli omezeným zdrojům.

Pro další práci na projektu by bylo vhodné rozšířit podporované datové typy o strukturované typy. Též interpret výsledků by mohl dostát změn, zejména implementování modulu pro překlad zdrojových kódů určených pro běh na grafické kartě, například do aplikace MS Visual Studio. Možným vylepšením by též byla optimalizace vnitřní struktury generujících a překládajících komponent.

Implementované řešení bylo oceněno druhým místem na soutěži Student EEICT 2012 v kategorii Informační systémy. Celkově je výsledná aplikace použitelným prostředkem pro automatický překlad zdrojových kódů napsaných v jazyce C# do prostředí NVidia CUDA.

Literatura

[1] *.NET Framework Conceptual Overview* [online]. 2012. [cit. 28.12.2011]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/zw4w595w.aspx>

[2] *.NET Compact Framework* [online]. 2012 [cit. 27.12.2011]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/netframework/aa497273>

[4] *.NET Micro Framework* [online]. 2012 [cit 27.12.2011]. Dostupné z WWW: <http://www.microsoft.com/en-us/netmf/default.aspx>

[5] GREVER, Steve. *ATI Stream vs. NVIDIA CUDA* [online]. 7.8.2009 [cit. 27.12.2011]. Dostupné z WWW: <http://www.pcper.com/reviews/Graphics-Cards/ATI-Stream-vs-NVIDIA-CUDA-GPGPU-computing-battle-royale>

[6] WATSON, Ben. *C# 4.0 řešení praktických programátorských úloh*. [cit. 5.1.2011] Brno : Zoner Press, 2010. ISBN 978-80-7413-094-6

[7] FARBER, Rob. *CUDA Application Design and Development*. [cit. 30.12.2011] ISBN 978-0123884268. Morgan Kaufmann, 14.1.2011

[8] *CUDA Architecture Overview* [online]. 2009 [cit. 30.12.2012]. Dostupné z WWW: <http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf>

[9] SANDERS, Jason. KANDROT, Edward. *CUDA by example: an introduction to general-purpose GPU programming*. [cit. 30.12.2011] ISBN 978-0-13-138768-3 Library of Congress Cataloging-in-Publication Data

[10] WOZNIAK, Rafal. *CUDA Integration with C#* [online]. 25.1.2010 [cit. 4.1.2012]. Dostupné z WWW: <http://www.c-sharpcorner.com/uploadfile/rafaelwo/cuda-integration-with-C-Sharp/>

[11] FATICA, Massimiliano. *CUDA Libraries* [online]. 5.5.2009 [cit. 29.12. 2011]. Dostupé z WWW: <http://www.gpgpu.org/static/sc2007/SC07_CUDA_3_Libraries.pdf>

[12] *CUDA Programming Model Overview* [online]. 2008 [cit. 30.12.2012]. Dostupné z WWW: <http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf>

[13] KOPP, Nick. *CUDAfy .NET* [online]. 14.12.2011 [cit. 4.1.2012]. Dostupné z WWW: <http://cudafy.codeplex.com/>

[14] *CUDAfy .NET* [online]. 2012 [cit. 14.2.2012]. Dostupné z WWW: <http://www.hybriddsp.com/Products/CUDAfyNET.aspx>

[15] *GPU Accelerated Computing* [online]. 2008 [cit. 27.12.2011]. Dostupné z WWW: <http://www.gadgetadvisor.com/computer-hardware/gpu-computing>

[16] *GPU.NET* [online]. 2011 [cit. 4.1.2011]. Dostupné z WWW: <http://www.tidepowerd.com/product>

[17] *GPU.NET v2.2 API Documentation* [online]. 2012 [cit. 5.1.2012]. Dostupné z WWW: <http://www.tidepowerd.com/gpu-net/api/latest>

[18] GRUNWALD, Daniel. *NRefactory* [online]. 19.10.2011 [cit. 3.1.2012]. Dostupné z WWW: <http://wiki.sharpdevelop.net/NRefactory.ashx>

[19] *NRefactory 5.0.0.6* [online]. 19.10.2011 [cit. 3.1.2012]. Dostupné z WWW: <http://nuget.org/packages/ICSharpCode.NRefactory>

[20] *Introduction to the .NET Framework* [online]. 2008 [cit. 30.12.2012]. Dostupné z WWW: <http://media.wiley.com/product_data/excerpt/46/04712359/0471235946.pdf>

[21] SMITH, Josh. *MVVM Foundation* [online]. 16.2.2010 [cit. 4.5.2012]. Dostupné z WWW: <http://mvvmfoundation.codeplex.com/>

[22] DAJBYCH, Václav. *MVVM: Model-View-ViewModel* [online]. 21.4.2009 [cit. 4.5.2012]. Dostupné z WWW: <http://dajbych.net/model-view-viewmodel>

[23] MANOLOV, Ivo. *Model-View-ViewModel (MVVM) Applications: General Introduction* [online]. 17.3.2012 [cit. 4.5.2012]. Dostupné z WWW: <http://blogs.msdn.com/b/ivo_manolov/archive/2012/03/17/10284665.aspx>

[24] *NVIDIA CUDA Library Documentation 3.0* [online]. 2012 [cit. 29.12.2011]. Dostupné z WWW: <http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/online/index.html>

[25] *Overview of the .NET Framework* [online]. 2008 [cit. 30.12.2012]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/a4t23ktk.aspx>

[26] *Platform invoke tutorial* [online]. 2012 [cit. 4.1.2012]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/aa288468%28v=vs.71%29.aspx>

[27] SMITH, Josh. *WPF Applications With The Model-View-ViewModel Design Pattern* [online]. únor 2009 [cit. 4.5.2012]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

Seznam příloh

Příloha 1.

Demonstrace generování kernel funkce

Příloha 2.

CD obsahující:

- zdrojové kódy aplikace
- technická zpráva v elektronické podobě
- ukázkové aplikace
- návod na zprovoznění aplikace

Příloha 1

Demonstrace generování kernel funkce

Zdrojový kód v C#:

```
class Program
{
   static void Main(string[] args)
   {
      int[] result = new int[10];
      int[] vector1 = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int[] vector2 = new int[] { 5, 4, 0, 8, 6, 1, 2, 6, 3, 7 };
      result = CUDADevice.Call(10, 1,() => CalcClass.AddVectors(vector1,vector2), 10);
      for (int i=0; i<10; i++) {
          Console.WriteLine(vector1[i] + " + " + vector2[i] + " = " + result[i]);
      }
      Console.ReadKey();
   }
}


[DeviceClass]
class CalcClass {
   [KernelFunction]
   public static int[] AddVectors(int[] a, int[] b)
   {
      int[] result = new int[10];
      int i = CUDAlib.Constants.CUDABlockId.X;
      if (i < 10) {
          result[i] = a[i] + b[i];
      }
      return result;
   }
}
```

Výsledný kód v C#:

```
class Program
{
   [System.Runtime.InteropServices.DllImport("CudaCallLibrary2.dll",
       CallingConvention=System.Runtime.InteropServices.CallingConvention.StdCall)]
   public static extern void AddVectors_export(int[] a, int[] b, int[] AddVectors_return_result, int
        blockCountX, int blockCountY, int blockCountZ, int threadCountX, int threadCountY, int
        threadCountZ, int sharedMemorySize, int length);

   static void Main (string[] args)
   {
      int[] result = new int[10];
      int[] vector1 = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int[] vector2 = new int[] { 5, 4, 0, 8, 6, 1, 2, 6, 3, 7 };
      AddVectors_export (vector1, vector2, result, 10, 1, 1, 1, 1, 1, 0, 10);
      for (int i = 0; i < 10; i++) {
           Console.WriteLine (vector1 [i] + " + " + vector2 [i] + " = " + result [i]);
      }
      Console.ReadKey ();
   }
}
```

Obsah knihovny CudaCallLibrary:

cuda.cu

```
#include "stdafx.h"
#include "CudaCallLibrary2.h"
#include <stdio.h>


__global__ void AddVectors(int * a, int * b, int * AddVectors_return) {
   int i = blockIdx.x;
   if (i<10) {
      AddVectors_return[i] = a[i]+b[i];
   } else {}
   return;
}

void AddVectors_call (int* a,int* b,int* AddVectors_call_return, int blockCountX, int
         blockCountY, int blockCountZ, int threadCountX, int threadCountY, int
         threadCountZ, int sharedMemorySize, int length) {
    int* a_device;
    cudaMalloc( (void**) &a_device, sizeof(int) * length);
    cudaMemcpy(a_device,a,sizeof(int) * length, cudaMemcpyHostToDevice);

    int* b_device;
    cudaMalloc( (void**) &b_device, sizeof(int) * length);
    cudaMemcpy(b_device,b,sizeof(int) * length, cudaMemcpyHostToDevice);

    int* AddVectors_return;
    cudaMalloc( (void**) &AddVectors_return, sizeof(int) * length);
    dim3 blockGrid(blockCountX,blockCountY,blockCountZ);
    dim3 threadGrid(threadCountX,threadCountY,threadCountZ);

    AddVectors<<<blockGrid, threadGrid,sharedMemorySize>>>
         (a_device,b_device,AddVectors_return);

    cudaMemcpy(AddVectors_call_return,AddVectors_return,sizeof(int)*length,
         cudaMemcpyDeviceToHost);

    cudaFree(a_device);
    cudaFree(b_device);
    cudaFree(AddVectors_return);
}
```

cuda.h

```
void AddVectors_call (int* a,int* b,int* AddVectors_return_call_return, int blockCountX, int
          blockCountY, int blockCountZ, int threadCountX, int threadCountY, int threadCountZ,
          int sharedMemorySize, int length);
```

CudaCallLibrary.cpp

```
#include "stdafx.h"
#include "cuda.h"


.. definice rozhrani ..

extern "C" CUDACALLLIBRARY2_API void __stdcall AddVectors_export(int* a,int* b,int*
      AddVectors_return_export_return, int blockCountX, int blockCountY, int blockCountZ, int
      threadCountX, int threadCountY, int threadCountZ, int sharedMemorySize, int length) {
    AddVectors_call(a, b, AddVectors_return_export_return, blockCountX, blockCountY,
        blockCountZ, threadCountX, threadCountY, threadCountZ, sharedMemorySize, length);
}
```