Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Metod Medja

Razvoj ovojne knjižnice za OpenCL v jeziku C#

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Patricio Bulić

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode uporabiti, morda bo zapisal tudi ključno literaturo.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	application programming interface	programski vmesnik
AVX	advanced vector extensions	napredne vektorske razširitve
GC	garbage collector	zbiralec smeti
ICD	installable client driver	namestljiv gonilnik odjemalca
OpenCL	open computing language	odprt računalniški jezik

Povzetek

Naslov: Razvoj ovojne knjižnice za OpenCL v jeziku C#

Avtor: Metod Medja

OpenCL je ogrodje za programiranje in izvajanje programov na različnih vrstah procesnih elementov, kot so splošno namenski procesorji in grafične procesne enote. Za prevajanje in izvajanje programov ta ponuja programski vmesnik oziroma API. Tega je mogoče neposredno uporabljati v vseh jezikih, ki se prevajajo v strojno kodo, saj so tako prevedene tudi implementacije vmesnika. V jezikih, kot sta C# in Java pa ni tako, saj se njuni programi izvajajo v navideznem stroju. Ker OpenCL ne ponuja knjižnic za te navidezne stroje, je zanje treba narediti ovojni vmesnik, ki izpostavi funkcije narejene v strojni kodi kot funkcije v navideznem stroju. V diplomskem delu bom opisal izdelavo ovojne knjižnice za jezik C#, ki poleg izpostavljanja originalnih funkcij omogoča tudi objektno usmerjen pristop k uporabi vmesnika.

Ključne besede: opencl, csharp, knjiznica.

Abstract

Title: Development of an OpenCL wrapper in C#

Author: Metod Medja

OpenCL is a framework for programming and executing programs on several types of processing elements. Such devices include graphics processing units and general-purpose processors. Its API provides the means for building an executing such programs. It can be used directly in languages that compile to native code as the implementations are provided as native libraries. Languages such as C# and Java cannot use these libraries directly as they execute in a virtual machine. Because OpenCL does not provide implementations for virtual machines, they require a wrapper interface. Such an interface must expose native functions as functions inside the virtual machine. This bachelor's thesis is going to go over the implementation of a wrapper library for C# which exposes the native functions as well as provides an objective oriented approach for the API.

Keywords: opencl, csharp, library.

Poglavje 1

Uvod

Vzporedni sistemi in paralelno programiranje nista nova pojma. Poleg velikega števila algoritmov, ki jih je mogoče izvajati na več procesnih enotah, je veliko algoritmov zasnovanih prav za ta namen. Takšni algoritmi se znotraj namiznih in mobilnih programov navadno izvajajo s pomočjo niti. Vsaka nit opravi del celotnega dela algoritma. In ker se različne niti lahko izvajajo na različnih jedrih procesorja, lahko tako programi delo opravijo hitreje, kot bi ga navadno. A ustvarjanje štirih niti na štiri jedrnem procesorju ne pomeni, da se bo vsaka nit izvajala na svojem jedru. Na delitev niti med jedri navadno vpliva razvrščevalec operacijskega sistema [83]. Včasih tudi razvrščevalec vgrajen v navidezni stroj, ki ga uporablja programski jezik.

1.1 OpenCL

OpenCL je ogrodje, ki ga sestavlja programski jezik in programski vmesnik oziroma API. V OpenCL 1.X je programski jezik razširitev jezika C [88]. V njem je mogoče s konstrukti jezika C definirati programe za OpenCL. Pri tem imajo posebno vlogo funkcije, ki so definirane kot ščepci. Te predstavljajo vstopne točke programov OpenCL, vsak program jih lahko vsebuje nič ali več. Programi brez ščepcev so koristni kot knjižnice, ki jih lahko uporabljajo drugi programi. Zagon programa se vedno zgodi preko ščepca. Ti se

med svojim delovanjem izvajajo na več procesnih elementih, glede na to, s katerimi parametri je bil ščepec zagnan. Za razliko od večnitnega izvajanja funkcije, tu sočasnega izvajanja kode ne nadzoruje razvrščevalec operacijskega sistema. To omogoča razvijalcem več nadzora nad načinom izvajanja njihovih programov. In ker je OpenCL podprt na različnih vrstah naprav, lahko z njim razvijalci izkoristijo tudi druge vire procesorske moči, kot so grafične kartice.

Razvijalci programov, ki želijo bolje izkoristiti centralne procesne enote in grafične procesne enote lahko to naredijo s pomočjo OpenCL. A to ni vedno preprosto. OpenCL je v osnovi standard. Ta definira arhitekturo, model za izvajalno okolje in naprave, delovanje izvajalnika in jezik. Implementacije tega standarda navadno zagotavljajo proizvajalci strojne opreme, ki podpirajo OpenCL. Te so razvijalcem ponujene kot knjižnice, ki implementirajo programski vmesnik in vsebujejo izvajalnik ter prevajalnik za njihove naprave. To pomeni, da morajo biti razvijalci pozorni na več stvari. Ker OpenCL navadno ni prisoten ob privzeti namestitvi gonilnikov, je treba te priskrbeti posebej. In ker so implementacije odvisne od strojne opreme, je te nekoliko težje priskrbeti vnaprej. Druga težava je morebiten obstoj več implementacij na istem sistemu. V tem primeru morajo razvijalci previdno izbrati pravo, oziroma jih naložiti več. Tretja težava je uporaba teh knjižnic v jezikih, ki ne morejo neposredno uvoziti knjižnic, ki so prevedene v strojno kodo.

Prvi dve težavi nista pomembni za to diplomsko nalogo. OpenCL namreč definira zasnovo okolja oziroma platforme. Implementacije proizvajalcev strojne opreme navadno izpostavijo le eno okolje, ki omogoča delo z njihovimi napravami. To omogoča obstoj združenih implementacij, ki omogočajo delo s strojno opremo več proizvajalcev. Ena izmed teh implementacij je Khronosov nalagalnik ICD [77]. Ta je na voljo kot implementacija OpenCL, ki poišče in naloži druge implementacije. Delo s to implementacijo omogoča uporabo strojne opreme več proizvajalcev brez dodatnega dela razvijalca.

To diplomsko delo se osredotoča na tretjo težavo. Torej uporabo OpenCL

v C#. Ker so implementacije OpenCL na voljo kot knjižnice, prevedene v strojno kodo, je za uporabo teh v jezikih, ki se prevajajo v strojno kodo, kot so C, C++ in Rust, potreben uvoz zaglavne datoteke ter povezava prevedene knjižnice v program ob preverjanju, zagonu oziroma po potrebi. V jezikih, ki se izvajajo znotraj navideznega stroja ali se tolmačijo, je lahko uporaba bolj zapletena. Za zagon takih programov se na zažene program neposredno, temveč se zažene navidezni stroj oziroma izvajalnik, ki poskrbi za pravilno izvajanje programa [56]. To ne pomeni, da program ne more vsebovati ene izmed implementacij OpenCL. Prav tako to ne pomeni, da program ne more naložiti ene izmed implementacij. Pomeni le, da mora to storiti s pomočjo svojega izvajalnika. Izvajalnik oziroma navidezni stroj, ki ga uporablja C#, potrebuje za uporabo knjižnic, kot je OpenCL, dodatno delo. Standardni način nalaganja takih knjižnic zahteva posebne definicije prototipov metod z vnaprej določenim imenom knjižnic, iz katerih bodo te naložene [68].

1.2 Cilj

C# je dobro prepoznan jezik, katerega uporaba je s pomočjo Microsoftovega .NET Standard modela mogoča na številnih operacijskih sistemih, kot so Windows MacOS, Linux, Android in iOS. Jezik ponuja širok nabor zmogljivosti, ki naredijo pisanje kode hitro, preprosto in varno. Cilj diplomskega dela je razvoj knjižnice, ki omogoča uporabo OpenCL 1.2 v jeziku C# in ponuja:

1. podporo na operacijskih sistemih Windows, MacOS in Linux,
2. nalaganje ene ali več implementacij OpenCL,
3. izvajanje metod API, ki jih določa standard,
4. objektno usmerjen ovojni vmesnik,
5. podporo za asinhrono programiranje,
6. dodatno varnost pri uporabi generičnih funkcij,

7. dodatno varnost pri delu s ščepci,

8. poenostavljen vmesnik za delo s slikami,

9. varno deljenje pomnilnika brez dodatnega kopiranja podatkov.

Dodaten cilj diplomskega dela je objava knjižnice na spletnem portalu GitHub.

Poglavje 2

Načrt izdelave

Izdelava knjižnice potrebuje nekaj načrtovanja. Poleg izbire imena in imenskega prostora knjižnice, je potreben tudi načrt opravil, potrebnih za izdelavo. Ker mora knjižnica omogočati neposredno uporabo vseh s standardom določenih funkcij, je dobro tu tudi začeti. Podpora za izvajanje teh funkcij bo namreč potrebna ob dodajanju ovojnih razredov. Ker OpenCL v svojih funkcijah uporablja lastne podatkovne tipe, je pred tem seveda treba definirati tudi te. S tem se zaključi osnova, ki bi že lahko bila samostojna knjižnica. Ker bo število podatkovnih tipov ustvarjenih samo s tem delom precej veliko, je dobro te ustvariti v lastnem imenskem prostoru oziroma paketu. Po izdelavi osnovnega dela knjižnice je smiselno nadaljevati z objektno usmerjenim vmesnikom. Ta predstavlja naslednji večji del knjižnice. Vse ostale zahtevane zmogljivosti so namreč le razširitev tega.

Odločil sem se, da bo ime knjižnice OCL.NET. Microsoft priporoča, da se ime projekta oziroma korenskega imenskega prostora projekta začne z imenom podjetja [67]. To samostojni razvijalci navadno izpustijo, razen če projekt nastaja v okviru večje skupine projektov. Takrat se pogosto uporabi ime skupine projektov. Ime projekta in imenskega korenskega prostora bi lahko začel z lastnim imenom in se tako držal priporočil. A ker ta projekt ne nastaja kot del večje skupine projektov in bo odprtokoden, obstaja možnost, da bo projekt v prihodnosti prevzel nekdo drug. V tem primeru je uporaba

lastnega imena v imenu projekta nekoliko neprimerna.

Poleg okvirnega načrta je treba določiti tudi za katero različico in variacijo .NET bo knjižnica razvita. Medtem ko .NET Framework [70] predstavlja originalno in najobsežnejšo implementacijo ogrodja .NET, je Mono [1] odprtokodna različica tega, ki ni omejena na operacijski sistem Windows. A to sta le najstarejši implementaciji. Ena izmed implementacij je tudi .NET Core [69]. To je prva odprtokodna implementacija, ki jo je razvil Microsoft. Ta je podprta na več operacijskih sistemih, a ne vsebuje vgrajene podpore za izgradnjo uporabniških vmesnikov. Zaradi omejenih zahtev, ki jih ima taka knjižnica, bi bila lahko ta narejena v katerikoli izmed implementacij. A obstaja tudi druga možnost. Z željo po poenotenju vseh implementacij je Microsoft izdelal specifikacijo .NET Standard [73]. Ta z vsako različico določa seznam API, ki jih morajo vsebovati implementacije. In čeprav to ni mogoče pri izdelavi programov, je ob izdelavi knjižnice mogoče za ciljno implementacijo .NET ogrodja določiti .NET Standard. Knjižnice, ki to naredijo, so potem združljive z vsemi implementacijami .NET ogrodja, ki implementirajo .NET standard. Microsoft pri izbiri različice priporoča uporabo čim nižje. Torej prve, ki vsebuje vse potrebne zmogljivosti. S poznavanjem ogrodja je mogoče oceniti, kateri API so potrebni za izdelavo knjižnice. In po pregledu podprtosti, je najslabše podprt atribut, ki je potreben za uvoz knjižnic, ki so prevedene v strojno kodo. Ta je bil v .NET Standard vključen z različico 1.1 [34]. To pomeni, da ni smiselno izdelati projekta, ki bo ciljal na različico 1.0, ker ga bo hitro treba nadgraditi na različico 1.1. A ta ne omogoča nalaganja več implementacij knjižnice in določanja poti do knjižnice po zagonu programa. Rešitev te težave bo torej mogoče zahtevala višjo različico standarda.

Za preverjanje pravilnega delovanja na vseh treh operacijskih sistemih je najlažje uporabiti enega ali več računalnikov z več operacijskimi sistemi. Izpostavljanje grafičnih kartic in druge strojne opreme navideznemu računalniku je mogoče, a zahteva več dela. Zaradi slučajne dostopnosti se bo testiranje izvajalo na treh računalnikih. In sicer namiznem računalniku s sistemom

Windows in prenosnikoma z Linux in MacOS. Za deljenje kode se bo uporabil Git, saj je cilj diplomskega dela objava na portalu GitHub. Za razvoj knjižnice se bo uporabilo JetBrainsovo razvojno okolje Rider. Razvoj knjižnic in programov v C# je mogoč v kateremkoli urejevalniku, a razvojna okolja navadno olajšajo in pospešijo razvoj. In Rider za razliko od večine ostalih razvojnih okolij podpira tudi Linux.

Poglavje 3

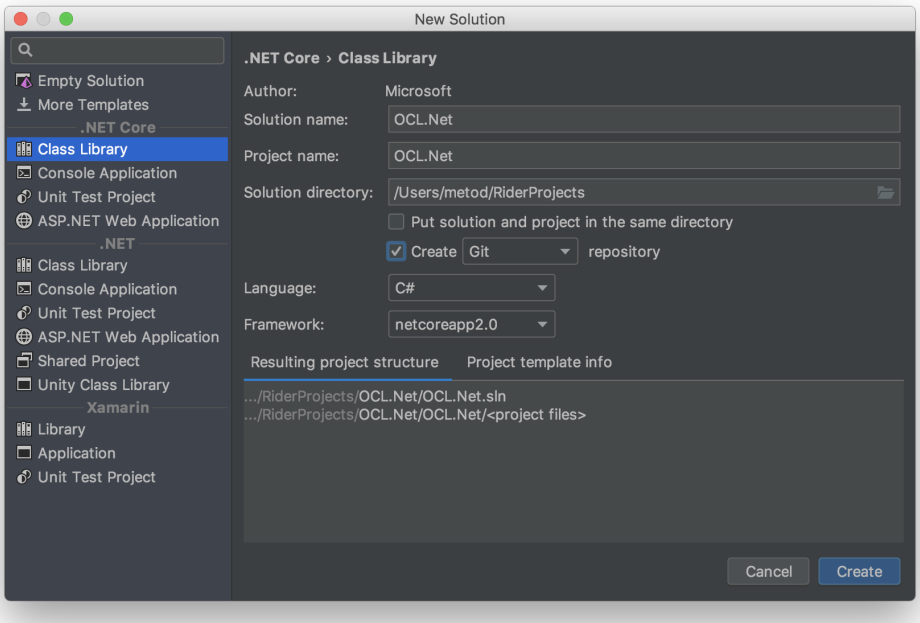
Osnovni del knjižnice

Najbolj osnovni del knjižnice je podpora za uporabo OpenCL API v C#. Ker se C# prevede v Microsoftov skupni zbirnik [95], se knjižnica lahko uporablja tudi v jezikih kot so Visual Basic, F# in C++ z razširitvijo CLI [71, 72].

Prvi korak potreben za implementacijo osnovnega dela je priprava podatkovnih struktur, ki jih OpenCL izpostavlja v svoji zaglavni datoteki. In jezik C, v katerem je spisana zaglavna datoteka, in C# omogočata definiranje struktur. Zato da se njihove vrednosti pravilno predstavljene v C# in v implementaciji OpenCL, je treba poskrbeti, da so njihova polja urejena v pravem vrstnem redu in da uporabljajo primerljive in enako velike podatkovne tipe [63, 62]. To je v C# mogoče doseči z dodatnimi atributi, ki se navedejo ob definiciji struktur. Podobno je potrebno narediti za vse vrste kazalcev na funkcije. Ti se uporabljajo za povratne klice in morajo biti v C# definirani kot delegati [33, 61]. In ker ima C# podporo za številne podatkovne tipe, je smiselno za konstantne vrednosti določiti tudi te. Pri tem je potrebna dodatna pozornost pri velikosti podatkovnih tipov, saj vsak številni podatkovni tip v ozadju uporablja enega izmed celoštevilskih podatkovnih tipov [37].

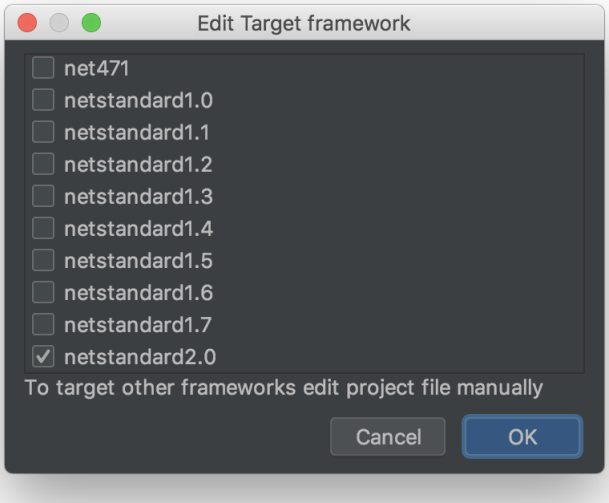
3.1 Izdelava projekta

V razvojnem okolju Rider je mogoče ustvarit veliko različnih C# projektov. Ker je projekt knjižnica za .NET Standard, je tega mogoče ustvariti kot .NET Core knjižnico, ki za ogrodje uporablja .NET Standard. Izbire različic .NET Standard v dialogu za ustvarjanje projekta omogočajo le različico 2.0. Končno stanje dialoga pred ustvarjanjem projekta je prikazano na sliki 3.1.



Slika 3.1: Dialog za ustvarjanje projekta.

Po ustvarjanju projekta je temu mogoče spremeniti različico .NET Standarda preko nastavitev projekta. V dialogu nastavitev se pod nastavitvami aplikacije nahaja možnost izbire ciljnega ogrodja. Tam je mogoča poleg prehoda iz .NET Standard na .NET Core in .NET Framework tudi izbira implementacije. Ob kliku na tipko za menjavo ciljnega ogrodja se odpre podmeni, ki je viden na sliki 3.2. Z odstranjevanjem izbire za .NET Standard 2.0 in izbiro .NET Standard 1.1 se spremeni ciljna različica na 1.1.



Slika 3.2: Dialog za izbiro ciljnih različic ogrodja.

3.2 Dodajanje podatkovnih tipov

Za boljšo organizacijo so priročni imenski prostori. Ker bo celoten osnovni del namenjen omogočanju uporabe implementacij OpenCL, se ta del knjižnice lahko ustvari v imenskem prostoru OCL.Net.Native. OCL.Net je imenski prostor projekta, Native je ime podprostora, v katerem bo ta del knjižnice. V C# je priporočeno, da so imena prostorov tesno vezana z datotečno organizacijo. Zato se za ustvarjanje podprostora ustvari direktorij. Ime Native je bilo izbrano, ker se knjižnicam, ki so prevedene v strojno kodo, v kontekstu C# programov in knjižnic v angleščini pogosto reče "native library". Pregled vseh struktur in različnih vrst konstant hitro razkrije, da jih je zelo veliko. Zato je dobro dodati dva podprostora znotraj Native. In sicer Structures in Enums. Dodajanje teh dveh podprostorov bo zmanjšalo število datotek, ki se nahajajo v direktoriju Native, in s tem pripomoglo preglednosti organizacije projekta.

3.3 Števni podatkovni tipi

Dodajanje števnih podatkovnih tipov je precej preprosto. Zaglavna datoteka namreč za vsako vrsto številskih podatkov definira svoj podatkovni tip in makroje s konstantami. Za vsakega izmed teh tipov je treba narediti števni tip, ki uporablja enakovreden celoštevilski podatkovni tip. Ker poimenovanje števnih tipov ni pomembno, se lahko njih in njihove vrednosti preimenuje v skladu z Microsoftovimi priporočili za poimenovanje [49]. Pri števnih tipih, ki predstavljajo zastavice, je smiselno uporabiti atribut `Flags` [45].

Iskanje vseh možnih vrednosti števnih tipov je preprost, a dolgotrajen postopek. Težave predstavljajo zgolj tipi, katerih celoštevilski podatkovni tip bi moral biti `intptr_t`. Ta podatkovni tip uporabljata `cl_device_partition_property` in `cl_context_properties`. Čeprav ima C# ustreznico tega tipa, se ta ne more uporabiti kot osnovni tip števnega podatkovnega tipa. Ker bo knjižnica uporabljena na 32 in 64-bitnih računalnikih, je največja možna velikost tega podatkovnega tip 64 bitov. Zato je najbližja ustreznica podatkovni tip `long`, ki je velik 64 bitov in je predznačen. A ker je podatkovni tip na 32-bitnih arhitekturah navadno velik 32 bitov, morajo biti vse vnaprej določene vrednosti temu primerno manjše. Zato jih je mogoče shraniti tudi v podatkovni tip `int`. Ker vse funkcije OpenCL, ki sprejemajo ta dva tipa, to počnejo preko kazalcev, kjer funkcije pričakujejo zaporedje vrednosti teh dveh podatkovnih tipov, števil in kazalcev [15, 12, 13], to nekoliko omeji izbiro podatkovnega tipa. Zaradi tega bodo morale biti te funkcije definirane kot da sprejemajo polje tipa `IntPtr`. A ker je preko API te vrednosti mogoče tudi prebrati, je smiselno, da se zanje uporabi podatkovni tip `long`, saj je s tem zagotovljeno, da bo lahko hranil vse možne vrednosti.

3.4 Strukture

Večina podatkovnih tipov, ki jih definira OpenCL, je definiranih kot poimenovanje kazalca ene izmed internih struktur. Za take strukture je mogoče

uporabiti podatkovni tip `IntPtr`. A se s tem izgubi ločevanje različnih vrst vrednosti preko podatkovnega tipa. To se lahko reši tako, da se zanje naredi strukture, ki vsebujejo le eno nespremenljivo lastnost. In sicer lastnost tipa `IntPtr`, saj ta podatkovni tip ustreza velikosti kazalca arhitekture, na kateri se knjižnica izvaja. Primer strukture za `cl_device_id` je razviden na sliki 3.3.

```
namespace OCL.Net.Native.Structures
{
    [StructLayout(LayoutKind.Sequential)]
    [SuppressMessage("ReSharper", "ConvertToAutoProperty")]
    public struct DeviceId
    {
        public IntPtr Handle => _handle;
        private readonly IntPtr _handle;
    }
}
```

Slika 3.3: Dialog za izbiro ciljnih različic ogrodja.

Struktura se začne z atributom, ki določa njeno urejenost. Ta določa, da so polja v strukturi v pomnilniku shranjena v istem vrstnem redu, kot so določena v kodi. Drugi atribut onemogoči opozorilo po izboljšavi. Struktura ima namreč dve lastnosti, ki se ju da združiti. Prva se v kontekstu C# navadno imenuje lastnost, druga polje [81]. V tem primeru je mogoče te dve lastnosti zamenjati z definicijo `public readonly IntPtr Handle;` ali `public IntPtr Handle { get; }`. Prvi pristop ni popolnoma enakovreden, saj taka lastnost ne more izpolniti zahtev vmesnika, če ta potrebuje lastnost tipa `IntPtr` z imenom `Handle` [55]. Drugi pristop je enakovreden, a je lahko težaven v primeru, ko struktura vsebuje kombinacijo obeh vrst lastnosti. Napredne lastnosti, ki so definirane kot metode, namreč včasih potrebujejo dodatne navadne lastnosti. Mesto, kjer so te lastnosti definirane, ni strogo

določeno, zato sme prevajalnik te ustvariti na neprimernem mestu in s tem spremeniti ureditev strukture v pomnilniku.

Kljub temu da se le tipa za naprave in okolja v zaglavni datoteki končata z `_id`, ima zaključevanje imen vseh teh tipov z `Id` svoj smisel. Ker bo knjižnica vsebovala ovojne razrede za vse te strukture, se s tem lahko hitro loči med strukturami, ki ustrezajo API, in razredi, ki omogočajo objektno usmerjen pristop.

Poleg vseh potrebnih struktur knjižnica vsebuje še dve dodatni strukturi. OpenCL namreč definira le eno strukturo za pomnilniške objekte `cl_mem`, a so ti v resnici ali slike ali medpomnilniki. Razlog za enoten podatkovni tip je verjetno ta, da so nekatere funkcije skupne slikam in medpomnilnikom. V C# je mogoče s pomočjo besede `implicit` definirati samodejne pretvorbe med podatkovnimi tipi [53, 41]. To omogoča definiranje treh struktur. Eno za pomnilniške objekte, drugo za slike in tretjo za medpomnilnike. S pomočjo implicitne pretvorbe slik in medpomnilnikov v pomnilniške objekte je mogoče v skupnih funkcijah uporabiti vse tri tipe struktur. Ker operacija zgolj spreminja podatkovni tip in ne vrednost, je smiselno dodati tudi eksplicitni pretvorbi iz pomnilniških objektov v slike in medpomnilnike.

Poleg teh podatkovnih tipov OpenCL definira še tri strukture. `cl_buffer_region`, `cl_image_format` in `cl_image_desc`. Te so uporabljene pri delu s pomnilniškimi objekti in so v strukture v C# predelane tako, da se za vsako izmed lastnosti izdela preprosta C# lastnost. Pri tem si morajo te slediti v pravem vrstnem redu in uporabljati primeren podatkovni tip. Od teh struktur je bolj nenavadna le struktura `cl_image_desc`. Ta je splošno namenska, saj se uporablja tako za eno, dve in tridimenzionalne slike ter polja slik [25]. Uporabo strukture je mogoče poenostaviti z dodatnimi statičnimi metodami, ki olajšajo nove strukture. Ta struktura, tako kot `cl_buffer_region`, uporablja podatkovni tip `size_t`, ki se pogosto uporablja za naslavljanje elementov v polju. Temu primerno se v obeh primerih z njim zgolj opisuje velikost polj. Najbolj primeren podatkovni tip v tem primeru je `UIntPtr`, saj se velikost tega prav tako prilagaja glede na arhitekturo procesorja. Tipa `uintptr_t` in

`size_t` sicer po definiciji nista enaka, a bosta enaka v večini primerov uporabe knjižnice. Ker je podatkovni tip `int` najpogosteje uporabljen številski podatkovni tip v jeziku C# in je tudi privzeti podatkovni tip za naslavljanje polj, je smiselno dodati metode, ki omogočajo delo z njim. Podatkovni tip `ulong` ni uporabljen tako pogosto. A je primeren kot alternativa tipu `UIntPtr`, ki je precej neroden za uporabo.

3.5 Delegati

OpenCL API poleg številnih podatkovnih tipov v svojih funkcijah sprejema tudi kazalce na funkcije. C# ustreznica teh so delegati. Ti so lahko popolnoma anonimni, generični [2, 46] ali tipizirani. Ker le tipizirani omogočajo določanje imen parametrov, je v imenskem podprostoru Native dodana datoteka Delegates, ki vsebuje definicije za vse vrste kazalcev funkcij.

3.6 Funkcije

Po določanju podatkovnih tipov, je za uporabo katerekoli izmed implementacij potrebno dodati prototipe za vse funkcije. Te je priporočeno dodati z definicijami zunanjih statičnih metod z atributom `DllImport`. A ta pristop ne omogoča nalaganja več implementacij in zahteva, da je ime knjižnice poznano ob času prevajanja. Zaradi teh dveh razlogov je bolj primerna uporaba drugačnega pristopa. S pomočjo `DllImport` je namreč mogoče naložiti knjižnice kot so `dl` in `kernel32`. S pomočjo teh je nato mogoče naložiti dodatne knjižnice dinamično [36]. Po pridobivanju kazalca na primerni simbol v knjižnici je mogoče tega spremeniti v anonimno metodo s pomočjo metode `GetDelegateForFunctionPointer` [59]. Drugi način uporabe metod iz naloženih knjižnic je preko ukaza vmesnega jezika, ki ga uporablja C#. Ukaz `calli` namreč omogoča izvajanje klicev v strojni kodi [75].

Odprtokodna knjižnica `AdvanceDLSupport` (oziroma `AdvancedDLSupport`) omogoča dinamično nalaganje drugih knjižnic in podpira oba načina klica-

nja funkcij [4]. Ker uporablja licenco "GNU Lesser General Public License, Version 3 (LGPLv3)", se sme uporabiti v tem projektu. Največja omejitev te knjižnice je uporaba različice 2.0 .NET Standarda. To pomeni, da mora biti projekt nadgrajen na to različico. Ta knjižnica, tako kot Micorsoftov priporočeni pristop, potrebuje prototipe funkcij, ki bodo uvožene. Pristop te knjižnice zahteva, da so ti prototipi določeni kot metode vmesnika.

Pri definiranju metod v vmesniku je treba uporabiti primerne podatkovne tipe in ali ohraniti poimenovanje metod ali pravo poimenovanje navesti s pomočjo dodatnih atributov. Kazalce je mogoče obravnavati kot polja ter kot `out` in `ref` parametre struktur. Oba parametra omogočata podajanje vrednosti preko referenc [78, 82]. Mogoče jima je določiti tudi nekaj dodatnih atributov. Atribut `MarshalAs` omogoča spreminjanje načina prenosa podatkov ob klicih [58]. Ker zna knjižnica sama določiti najbolj primeren način prenosa, dodajanje tega atributa navadno ni potrebno. Uporaba atributa je navadno potrebna le v primerih, ko se knjižnica odloči za napačen način prenosa nizov. Poleg tega atributa je mogoče atributom dodati tudi atributa `In` in `Out`. Knjižnica prisotnosti teh dveh atributov ne upošteva. Njuna uporaba pomaga uporabnikom knjižnice kot namig.

3.7 Nalaganje implementacij

Po nadgradnji projekta na .NET Standard 2.0 in dodajanju knjižnice `Advance DLSupport` preko upravljavca paketov NuGet, je treba to uporabiti za ustvarjanje instanc. Ta ustvarjanje instanc omogoča z razredom `NativeLibrary Builder`. A pred uporabo tega, je potrebno poznati poti oziroma imena datotek implementacij OpenCL. Ker to poteka za vsak operacijski sistem nekoliko drugače, se lahko za to delo definira vmesnik in se ga implementira za vsak operacijski sistem. Poleg tega se lahko doda tudi implementacija, ki zgolj posreduje vse dostope do instance najbolj primerne implementacije vmesnika. Ker mora ta vmesnik znati iskati knjižnico po imenu, v vseh primernih direktorijih, bo s tem opravljal vse, kar zahteva vmesnik `ILibrary`

`PathResolver` knjižnice `AdvanceDLSupport`. Zato lahko novi vmesnik deduje njega. Glavna naloga novega vmesnika je še vedno iskanje poti oziroma imen datotek implementacij OpenCL. MacOS vedno ponuja svojo implementacijo na vnaprej določeni poti, zato je zanj določanje poti precej preprosto. In ker namestitev gonilnikov grafičnih kartic Nvidia na Windows in Linux nakazuje na to, da ta namesti tudi nalagalnik ICD, zna biti dovolj, da se na teh dveh sistemih poskuša najti to implementacijo. A ker nič od tega ni zagotovljeno, je dobro dodati možnost dodajanja direktorijev za iskanje implementacij in imen dodatek teh implementacij.

Zadnji del nalaganja teh implementacij je nekaj poenostavitev uporabe. Ker je cilj knjižnice olajšanje razvoja, je dobro dodati možnost samodejnega nalaganja implementacij. To se lahko naredi s pomočjo statičnega razreda, ki hrani privzet seznam naloženih knjižnic. Ker ni nujno, da bo knjižnica kadarkoli uporabljena, je dobro te naložiti ob prvem dostopu. A ker lahko do lastnosti dostopa več niti hkrati, je treba okoli lastnosti dodati tudi zaklepanje. S pomočjo zaklepanja z dvojnim preverjanjem je mogoče zaklepanje preskočiti v primeru, ko so privzete knjižnice že naložene [35]. Znotraj statičnega razreda je potrebno le iskanje in nalaganje implementacij OpenCL. To je v celoti izvedljivo s pomočjo knjižnice `AdvanceDLSupport` in iskalcev implementacij OpenCL. Priročen način izvedbe tega je, da se za iskanje in nalaganje uporabi privzeta instanca vmesnika za iskanje, ki je spremenljiva. Tako lahko uporabnik spremeni privzeti način nalaganja pred uporabo knjižnice. Prav tako se noben del knjižnice ne zanaša na to, da se privzete naložene implementacije ne spreminjajo, zato se statični lastnosti, ki hrani instance implementacij OpenCL, lahko omogoči spreminjanje vrednosti. Nazadnje je dobro dodati zastavico, ki določa, ali se uporabijo vse ali le prva najdena implementacija. V primeru uporabe nalagalnika ICD je namreč mogoče naložiti več implementacij, ki izpostavijo isto okolje.

3.8 Obvladovanje napak

Za razliko od C#, C ne pozna izjem. Zato vse funkcije OpenCL, ki lahko spodletijo, na nek način vračajo dodatno številsko vrednost. Negativne vrednosti zaznamujejo napake ob izvedbi funkcije. Obravnavanje napak je mogoče nekoliko poenostaviti s pomočjo razširitvene metode [43]. Z njimi je mogoče definirati dodatno metodo na številnem tipu `ErrorCode`, ki definira vse napake. V primeru negativne vrednosti rezultata, lahko ta sprožiti izjemo. Boljši pristop bi bil dodajanje različic metod, ki to naredijo samodejno. A ta oteži vzdrževanje.

3.9 Rezultat

Z osnovnim delom knjižnice je mogoča uporaba OpenCL v jeziku C#. S pomočjo kode na sliki 3.4 je mogoče izpisati imena vseh naprav, ki podpirajo OpenCL.

```
const DeviceInfo di = DeviceInfo.DeviceName;
const DeviceType dt = DeviceType.All;
var p = default(PlatformId);

foreach (var lib in OpenCl.Libraries)
{
  lib.clGetDeviceIDs(p, dt, 0, null, out var n).HandleError();
  var dl = new DeviceId[n];
  lib.clGetDeviceIDs(p, dt, n, dl, out _).HandleError();

  foreach (var d in dl)
  {
    lib.clGetDeviceInfo(
      d, di, UIntPtr.Zero, IntPtr.Zero,
      out var s).HandleError();

    var buf = Marshal.AllocHGlobal((int) s);
    lib.clGetDeviceInfo(d, di, s, buf, out _).HandleError();

    Console.WriteLine(Marshal.PtrToStringUTF8(buf));
    Marshal.FreeHGlobal(buf);
  }
}
```

Slika 3.4: Koda za izpis imen vseh naprav OpenCL.

Poglavje 4

Razširitev knjižnice

Naslednji večji del knjižnice je podpora za objektno usmerjeno programiranje. Za večino struktur je smiselno dodati razred, kjer so funkcije OpenCL, ki se sklicujejo na pripadajočo strukturo, predstavljene kot metode razreda. Možne so tudi druge izboljšave in dopolnitve:

1. poenostavljanje uporabe zapletenih funkcij z večkratnimi definicijami metod,

2. ohranjanje informacije o podatkovnem tipu elementov pomnilniških objektov s pomočjo generičnih parametrov,

3. ločevanje eno, dve in tridimenzionalnih slik s pomočjo dodatnih razredov,

4. varna uporaba C# polj znotraj pomnilniških objektov brez dodatnega kopiranja,

5. varen dostop do preslikanih delov pomnilniških objektov,

6. podpora za asinhrono programiranje,

7. preverjanje argumentov ščepcev s pomočjo metapodatkov.

4.1 Okolja in naprave

Z dodajanjem ovojnih razredov je mogoče poenostaviti programski vmesnik in omogočiti objektno usmerjen pristop uporabe knjižnice. Cilj tega je narediti uporabniški vmesnik bolj domač razvijalcem, ki so vajeni jezika C#. Podatkovna tipa `IntPtr` in `UIntPtr` sta v C# zelo redko uporabljena, saj sta prvotno namenjena komunikaciji z drugimi jeziki in delom s strojno kodo [57, 89]. To pomeni, da je uporaba teh v prototipih funkcij OpenC API, čeprav pravilna, neprijazna za razvijalce. Ker je najpogosteje uporabljen celoštevilski podatkovni tip v C# `int`, je dobro dodati možnost njegove uporabe. Pretvorbo iz `int` v pravilni podatkovni tip bosta namreč morala narediti ali knjižnica ali uporabnik knjižnice. Če se ta naredi v knjižnici, se s tem le poenostavi njena uporaba. A to ne pomeni, da je `int` vedno najboljša izbira. Nekatere izmed funkcij OpenCL namreč sprejemajo podatkovni tip `UIntPtr` ravno zato, ker sta `int` in `uint` lahko v nekaterih primerih premajhna. Za te metode mora ostati način podajanja večjih vrednosti. V teh primerih se lahko metode definira večkrat, enkrat z uporabo podatkovnega tipa `int` in ponovno z uporabo drugega podatkovnega tipa. Drugi podatkovni tip je lahko kar `UIntPtr`, saj je to tudi želeni podatkovni tip. Lahko pa je tudi `ulong`, saj je v C# ta podatkovni tip vedno velik, vsaj toliko kot `UIntPtr`. To pomeni, da je ta sicer lahko večji, a uporaba tega zgolj premakne napako ob pretvorbi iz uporabnikove kode v kodo knjižnice.

Poleg uvajanja alternativnih podatkovnih tipov lahko knjižnica uvede tudi uporabo številnih podatkovnih tipov za podatkovna tipa `cl_device_partition_property` in `cl_context_properties`. Za oba sta definirana številna tipa, a ta nista uporabljena, saj sta v zaglavni datoteki OpenCL oba definirana kot `intptr_t`. Ker C# ne omogoča definiranja številnih tipov, ki za osnovni podatkovni tip uporabljajo `IntPtr`, uporabljata `long`. In ta ni vedno enako velik kot `intptr_t`. Metode ovojnih razredov lahko sprejemajo številni tip in ju pretvorijo naknadno.

Izboljšave, ki jih prinesejo ovojni razredi, ne pridejo brez dodatne cene. Ker knjižnica ne onemogoča neposrednega dostopa do API s pomočjo struk-

tur, to pomeni, da morajo biti ovojni razredi ali striktno ločeni od struktur ali delovati skupaj. Pristop z ločevanjem je precej preprost, uporabniku mora biti zgolj onemogočen dostop do struktur OpenCL. A ta pristop drastično omeji razširjanje knjižnice. Drug pristop je omogočanje kombiniranega načina uporabe knjižnice. Bistvo tega pristopa je dodajanje načina za pretvarjanje med vrednostmi struktur in instancami razredov. To je tudi največja težava tega pristopa. Pridobivanje strukture iz razreda je preprosto, saj mora razred vedno poznati vrednost, s katero dela. Obnavljanje razreda iz vrednosti strukture je bolj zapleteno. Strukture, kot so okolja, naprave, konteksti in dogodki namreč držijo le kazalec. Zato mora biti povezava med temi kazalci in instancami razredov narejena naknadno. Najlažji način izvedbe tega je s slovarjem. A če slovar hrani reference na instance razredov, to pomeni, da teh instanc ne more počistiti zbiralec smeti [47]. Če bi bile strukture spremenjene v razrede, bi za to lahko uporabil razred `ConditionalWeakTable`, ki s pomočjo šibkih referenc poveže dve instanci razredov [30, 39]. Druga možna rešitev je uporaba slovarja, ki kot vrednosti hrani šibke reference. Uporaba teh omogoča samodejno brisanje instanc, kljub temu da so shranjene v slovarju [94]. S tem je rešena težava preprečevanja brisanja instanc, a ne brisanja zapisov iz slovarja. Ker so ključi takega slovarja strukture, se te nikoli ne pobrišejo, zato zapisi v slovarju ostanejo za vedno. To težavo je mogoče delno rešiti z ročnim brisanjem zapisov, kadarkoli katera izmed instanc šibkih referenc ne obstaja več in ob ročnem odstranjevanju instanc, pri katerih se štejejo reference.

Število vrstic kode, potrebnih za dodajanje teh preslikav, je mogoče zmanjšati s ponovno uporabo kode. To je mogoče doseči z dodajanjem abstraktnega razreda, ki se uporabi kot osnova vseh ostalih razredov. Ta ima generični parameter, ki določa vrsto strukture, ki se uporablja za identifikacijo. Nepravilno uporabo razreda je mogoče omejiti, če se vrednost generičnega parametra omeji na strukture in instance novega vmesnika, ki se ga lahko doda na vse primerne strukture [98].

Pridobivanje seznama okolij in naprav je bolj preprosto. OpenCL defi-

nira dve funkciji za obe operaciji. Ti dve funkciji, kot vse ostale, ki vračajo več vrednosti, omogočata dva načina klicanja. Prvi je z medpomnilnikom in velikostjo medpomnilnika, drugi brez medpomnilnika in z velikostjo medpomnilnika enako nič. Drugi način pri vseh metodah preskoči izvajanje in preko reference vrne velikost podatkov oziroma število vrednosti, ki bi se drugače zapisale v medpomnilnik. Edini način klicanja teh metod, ki zagotavlja, da bodo prebrane vse vrednosti, je, da se funkcijo najprej pokliče na drugi način, pridobi primeren medpomnilnik in ponovi klic s tem medpomnilnikom. Ta način klicanja je bolj zapleten, a ga lahko reši kar knjižnica. Za klicanje teh funkcij mora knjižnica uporabiti eno izmed referenc na naložene implementacije OpenCL. Omogočanje ročnega določanja teh je smiselno za napredne primere uporabe. A uporabniki knjižnice tega najverjetneje ne bodo potrebovali vedno, zato je smiselno metode za nalaganje definirati večkrat, ker različice, ki instanc ne sprejemajo, uporabijo privzete. Kar ni smiselno, je podajanje instance, ko se s pomočjo okolja nalagajo naprave, saj morajo biti te naložene s pomočjo iste instance, kot jo uporablja okolje. To pomeni, da morajo instance okolij hraniti referenco na instance implementacij OpenCL. In iz podobnih razlogov to velja tudi za vse ostale razrede. Zato je lastnost za implementacijo najbolje definirati v skupnem osnovnem razredu.

Ker morajo metode, ki nalagajo okolja in naprave vračati instance razredov in ne vrednosti struktur. Saj brez ustvarjanja instanc teh ni mogoče povezati s pripadajočimi vrednostmi struktur, kar bi onemogočilo pretvorbo iz vrednosti struktur v instance razredov. Ročno ustvarjanje instanc za navadnega uporabnika knjižnice ni priporočeno, saj je ta postopek zaradi hranjenja povezav nekoliko bolj zapleten. Zato je pametno narediti konstruktor zaseben. Možna je tudi uporaba zaščitenega konstruktorja, a je veljavnost te rešitve omejena z možnostjo razširjanja oziroma dedovanja od razreda. Dedovanje od razredov, katerih instance ustvarja knjižnica, ni priročno. Knjižnica bi se za pravilno uporabo izpeljanih razredov morala zavedati njihovega obstoja in poznati pravilen način ustvarjanja le-teh. Prvi del težave je mogoče rešiti s knjižnico, drugi ni rešljiv tako enostavno. Zato je smiselno preprečiti

dedovanje teh razredov in uvesti alternativne pristope dopolnjevanja razredov. En alternativni pristop je spreminjanje vidljivosti konstruktorja oziroma ene izmed zasebnih metod, ki kličejo konstruktor, na interno, ter to metodo narediti javno v drugem razredu. S tem je uporabniku knjižnice ponovno omogočen klic konstruktorja, a je ta bolj zapleten. To pomaga pri preprečevanju napačne uporabe knjižnice, medtem ko ohranja nekaj izmed možnosti dopolnjevanja, v primeru, da bi uporabnik želel naložiti okolja ali naprave na drugačen način.

Razreda za okolja in naprave nista namenjanja le držanju vrednosti struktur in lažjemu nalaganju le-teh. Tako okolja, naprave in druge strukture omogočajo poizvedovanje po dodanih informacijah, kot so imena, zmogljivosti in nastavitve. Pridobivanje teh vrednosti ni najbolj preprosto, saj za vsako strukturo obstaja po ena funkcija, ki omogoča pridobivanje vseh informacij. Katera informacija bo pridobljena, se nadzoruje s parametrom. Ti parametri so v knjižnici definirani kot številni podatkovni tipi, zato je njihov podatkovni tip mogoče dodati kot nov generični parameter skupnega osnovnega razreda. Prav tako je postopek branja pri vseh podatkovnih tipih enak, spremeni se le funkcija, ki jo je treba poklicati. Zato se v osnovnem razredu definira abstraktna metoda, ki s pomočjo vrednosti številnega tipa prebere vrednost informacije. Funkcije za branje informacij vrednost zapišejo na mesto, na katero kaže kazalec. To pomeni, da je mogoče branje ene vrednosti ali polja vrednosti obravnavati enako, saj je branje ene vrednosti enako branju polja z eno vrednostjo. Poseben način branja potrebuje le branje niza. Nizi v C# za vsak znak uporabljajo dva bajta, medtem ko OpenCL uporablja ASCII oziroma UTF-8. To pomeni, da lahko posamezen znak uporablja več kot 1 bajt, a to le v primeru, ko je to potrebno [92]. Zaradi tega je potrebno nize pridobljene iz OpenCL obravnavati po en bajt naenkrat, torej s pomočjo podatkovnega tipa `byte`. Posledično to pomeni, da bi neposredna pretvorba takega niza v niz v jeziku C# spremenila pomen besedila ob prvem znaku, ki ne uporablja dveh bajtov. Pravilno pretvorbo besedila je mogoče izvesti s pomočjo razreda `UTF8Encoding`, ki omogoča pretvarjanje polja bajtov v niz

[93].

Pridobivanje informacij ima še eno dodatno oviro. Metodo je zapleteno narediti generično, torej da deluje samodejno, glede na podatkovni tip podan kot generični argument. C# omogoča izdelovanje polja poljubne velikosti z generičnimi parametri. Torej ob ustvarjanju polj ni treba poznati točenega podatkovnega tipa. A spreminjanje tega v kazalec in preprečevanje premikanja s strani GC ni preprosto. Prvo težavo predstavljajo števni podatkovni tipi. .NET pozna koncept, ki ga v angleščini poimenujejo "blittable". Podatkovni tipi, ki imajo to lastnost, imajo enako pomnilniško predstavitev v in izven .NET [8]. To lastnost ima večina osnovnih podatkovnih tipov, kot so `byte`, `int`, `long` in drugi. To lastnost imajo tudi vse strukture, ki so definirane z atributom `StructLayout`, saj ta določa njihovo pomnilniško ureditev [85]. A ta atribut ne deluje na števnih tipih, njihova pomnilniška ureditev pa je samodejna. In ker ni določena vnaprej, števni tipi nimajo te lastnosti. To pomeni, da pretvorba teh vrednosti v kazalce in nazaj, s pomočjo bolj pogosto uporabljenih metod, kot je `Marshal.PtrToStructure`, ni mogoča [64]. Drugo težavo predstavlja fiksiranje podatkov. Stavek `fixed` namreč ne deluje s spremenljivkami, katerih podatkovni tip sestavljajo generični podatkovni tipi [44]. V C# poleg tega obstaja še en način fiksiranja. In sicer s pomočjo metode `GCHandle.Alloc`. A ta ne deluje s podatkovnimi tipi, ki nimajo lastnosti "blittable"[48]. Težave kot je ta rešujeta strukturi `Span` in `Memory`. Namenjeni sta namreč bolj varni uporabi pomnilnika, ki ga ne upravlja GC [84, 66]. Strukturi držita zaporedje vrednosti, ki je lahko podano kot polje ali kazalec, in med drugim omogočata tudi reinterpretacijo. Preverjanja pred reinterpretacijo s pomočjo teh dveh struktur dovoljujejo uporabo uporabo števnih tipov. Zato je z njima mogoče interpretirati zaporedje različnih podatkovnih tipov v zaporedje bajtov. To zaporedje bajtov je nato mogoče fiksirati s pomočjo stavka `fixed`. Ker strukturi `Span` in `Memory` nista na voljo v .NET Standard 2.0 in bosta dodani z različico 2.1, ki ni dokončana, je potrebno zanju uvoziti knjižnico `System.Memory`. Različica 2.1 bo namreč zgolj vključila to knjižnico. Za pravilno delovanje te je potrebna tudi uporaba C#

različice 7.2. To je mogoče izbrati v istem dialogu kot ciljno različico ogrodja .NET.

Drug način reševanja te težave je nadgradnja na C# 7.3. Nadgradnja na to različico omogoča uporabo generične omejitve `unmanaged` [97]. S pomočjo te omejitve je mogoče zagotoviti, da je generični parameter primeren za uporabo v stavku `fixed`. To pomeni, da je z njim po dodajanju omejitve mogoče fiksirati polja generičnih tipov, ki ustrezajo tej omejitvi. In za razliko od lastnosti "blittable", številni podatkovni tipi ustrezajo tej omejitvi.

Naprave dodatno omogočajo tudi deljenje. Deljenje naprav je v OpenCL mogoče preko ene metode, katere uporaba ni trivialna. Ta omogoča enakomerno deljenje, deljenje po skupinah in deljenje po sorodnosti domene [15]. To je v ovojnem razredu naprave mogoče poenostaviti z metodami za vsak način deljenja. Te metode nato po pravilih specifikacije izdelajo primerno zaporedje lastnosti.

Zadnji del razreda za naprave je podpora za spreminjanje števila referenc. OpenCL s pomočjo teh določi, kaj mora neko vrednost pobrisati iz pomnilnika. Vse strukture razen okolja imajo namreč po dve funkciji, s katerimi se lahko poveča in pomanjša število referenc. Funkcija za naprave deluje le za naprave, ki so bile ustvarjene z deljenjem, a klic te z neveljavno napravo, ne vrne napake, temveč preprosto ne naredi ničesar [27]. Ker razred za naprave ni edini podatkovni tip, ki bo potreboval to logiko, se ta nahaja v novem abstraktnem razredu, ki deduje od skupnega abstraktnega razreda, in mu doda par metod. Novi razred definira tudi dve abstraktni metodi, ki morata povečati in pomanjšati število referenc za njuno instanco. Nadzorovanje števila referenc je mogoče narediti na dva načina. Prvi je, da razred hrani število referenc, ki so bile narejene preko njega. Drugi je, da razred sproti poizveduje po trenutnem številu referenc preko primerne informacije podatkovnega tipa. Pomembnost dveh implementacij se najbolj pozna pri metodi `Dispose` vmesnika `IDisposable`, katerega implementacija je priporočena za vse podatkovne tipe, ki držijo vire, ki jih ne nadzoruje GC [52]. V prvem primeru je lahko ta metoda implementirana tako, da pobriše le lastne reference.

V drugem primeru to ni mogoče, metoda `Dispose` mora biti omejena na eno ali vse reference. Ker brisanje le ene reference ne zagotavlja brisanja objekta, kar ni v skladu z namenom metode `Dispose`, in brisanje vseh referenc lahko vpliva na uporabnike strukture, ki ne uporabljajo ovojnega razreda, je bolj smiseln prvi pristop. Vse tri metode, torej metoda za večanje števila referenc, manjšanje števila referenc in brisanje vseh referenc, morajo biti varne za uporabo z več niti sočasno. A ta varnost mora biti izpuščena, če brisanje vseh referenc sproži destruktor razreda. Ta klic se namreč izvede na niti GC, kjer ne sme priti do zaklepanja. Prav tako ta klic ne sme spodleteti, saj izjeme na niti GC sesujejo program.

Rezultat implementacije razredov za okolja in razrede je poenostavljen način uporabe naprav. Primer izpisa imen vseh naprav iz prejšnjega poglavja 3.4 je z novimi razredi mogoče narediti na bolj enostaven način. Kot je razvidno iz slike 4.1.

```
foreach (var device in Device.QueryDevices(DeviceType.All))
{
    Console.WriteLine(device.Name);
}
```

Slika 4.1: Objektno usmerjena koda za izpis imen vseh naprav OpenCL.

4.2 Konteksti

S pomočjo naprav je mogoče v OpenCL definirati kontekste. Ti se uporabljajo za upravljanje ukaznih vrst, pomnilnika, programov in ščepcev [12]. Za ustvarjanje konteksta je treba podati eno ali več naprav istega okolja. Kljub temu da knjižnica omogoča sočasno uporabo različnih implementacij OpenCL, morajo vse naprave pripadati isti implementaciji, saj se ustvarjanje konteksta vedno izvede s pomočjo ene izmed implementacij. Podajanje naprave napačne implementaciji bo v najboljšem primeru vrnilo napako `CL_`

`INVALID_DEVICE` in v najslabšem primeru uporabilo napačno napravo. Kontekstom je mogoče ob ustvarjanju podati tudi okolje. Ker je to samodejno določeno na podlagi uporabljenih naprav, je ta lastnost uporabna zgolj kot dodatno preverjanje, da so bile uporabljene pravilne naprave.

Ker je kontekst mogoče ustvariti z več napravami, metoda za ustvarjanje na spada v razred za naprave, vsaj ne kot metoda instanc razreda. Bolj primerno mesto za to je v razredu kontekstov, kot statična metoda. Ustvarjanje konteksta je mogoče tudi znotraj konstruktorja. A ker ustvarjanje konteksta lahko spodleti in ne obsega več kot le pripravo nove instance, to delo presega naloge konstruktorja [54].

Najtežji del implementacije razreda za kontekste je omogočanje spremljanja napak. Ob ustvarjanju kontekstov je namreč mogoče podati kazalec na funkcijo, ki jo implementacije OpenCL kličejo v primeru napak. Spremljanje teh napak je koristno, saj implementacije poleg napak včasih sporočajo tudi opozorila. Metoda, ki je podana kot kazalec, je lahko statična ali vezana na določeno instanco. A uporaba metode, ki je vezana na instanco, ima svoje težave. Če OpenCL pokliče funkcijo za tem, ko je instanca pobrisana iz pomnilnika, to sesuje program, saj kazalec na metodo ni več veljaven. Uporaba statične metode te težave nima. A ker funkcija ne vsebuje konteksta, klica brez dodatnega dela ni mogoče posredovati pravi instanci razreda za kontekste. Reševanje težave iskanja pravega konteksta je mogoče z uporabo dodatnega parametra `user_data`. Ta se lahko poda ob ustvarjanju konteksta in je uporabljen ob vsakem povratnem klicu. Vrednost tega parametra ne more biti struktura konteksta, saj ta ne more obstajati pred ustvarjanjem. To pomeni, da je treba za ta namen uvesti dodaten identifikator. Ker je podatkovni tip parametra `user_data` `intptr_t`, bo ta navadno velik vsaj 32 bitov, torej toliko kot `int` v C#. Zato je lahko ta identifikator ustvarjen kot zaporedno število. V primeru ustvarjanja zelo velikega števila kontekstov obstaja možnost izrabe vrednosti podatkovnega tipa, a ker ustvarjanje kontekstov ni pogosta operacija, ta težava ni pomembna. Povezovanje tega identifikatorja z instanco razreda je mogoče narediti na isti način, kot povezavo vrednosti

struktur s pripadajočimi instancami razredov za preslikavo med podatkovnimi tipi OpenCL in ovojnimi razredi. Ker je ustvarjanje instanc razredov za omogočanje razširitev mogoče preko pomožnega razreda, mora ta podpirati tudi delo s temi identifikatorji.

S pomočjo ustvarjenega ovojnega razreda za kontekste je mogoče te uporabiti na način, ki ga prikazuje slika 4.2.

```
using (var context = Context.Create(Device.QueryDevices()))
{
    Console.WriteLine(
        $"Kontekst uporablja {context.NumDevices} naprav.");
}
```

Slika 4.2: Primer uporabe ovojnega razreda za kontekste.

4.3 Izvajalne vrste in dogodki

Ker je delo z izvajalnimi vrstami precej enostavno, je tudi izdelava ovojnega razreda precej enostavna. Izvajalne vrste je mogoče ustvariti z napravo in kontekstom. Naprava, ki je podana ob ustvarjanju, more biti del konteksta, v katerem se uporablja. To preverjajo implementacije OpenCL, zato tega zaradi enostavnosti knjižnice ni potrebno dodatno preverjati. Treba je zgolj preveriti, ali kontekst in naprava uporabljata isto implementacijo OpenCL. Ustvarjanje je mogoče nekoliko poenostaviti z dodajanjem dodatne definicije metode za ustvarjanje, ki sprejem le napravo in zanjo samodejno naredi kontekst. Vrstam je ob ustvarjanju mogoče določiti tudi dve lastnosti [11]. Prva omogoča izvajanje izven vrstnega reda dodajanja opravil, kar je v bolj naprednih primerih uporabe lahko izredno koristno, a naredi uporabo bolj zapleteno, saj je s tem za določanje vrstnega reda izvajanja opravil, potrebno te ustvariti s seznamom opravil, na katera morajo počakati. Druga lastnost omogoča profiliranje. Ko je ta lastnost omogočena vsebujejo vsi dogodki

informacije o njihovem izvajanju. Obe je mogoče vklopiti ob ustvarjanju izvajalne vrste preko dodatnih parametrov.

Po dodajanju ovojnega razreda za izvajalne vrste je mogoče dodati razrede za dogodke. OpenCL API omogoča pridobivanje informacij in čakanje na dogodke. Da je z njimi mogoče delati asinhrono, omogoča tudi dodajanje poslušalcev, torej kazalcev na funkcije za povratne klice. In ker uporabljajo dogodki štetje referenc, jih je, kot večino podatkovnih struktur OpenCL, mogoče zadržati in sprostiti. Za dogodke, ki so ustvarjeni na izvajalni vrsti, ki ima vklopljeno profiliranje, je mogoče pridobiti tudi časovne žige različnih dogodkov. Poleg vsega tega je dogodke mogoče tudi ustvarjati. Prvi način ustvarjanje dogodkov je preko funkcij `clEnqueueMarkerWithWaitList` in `clEnqueueBarrierWithWaitList`. Z njima je mogoče ustvariti in dodati v izvajalno vrsto dogodke, ki so končani, ko se zaključijo vsi dogodki izvajalne vrste oziroma vsi podani dogodki. Specifikaciji metod ne zahtevata, da vsi dogodki pripadajo isti izvajalni vrsti [20, 16]. Razlika med funkcijama je, da dogodek, ki je ustvarjen s prvo, ne preprečuje izvajanja opravil, ustvarjenih za dogodkom. Razlika med metodama ima torej vpliv le, ko je dogodek dodan na vrsto, ki omogoča izvajanje opravil izven vrstnega reda.

Drugi način ustvarjanja dogodkov so uporabniški dogodki. Ti dogodki ne pripadajo vrsti, temveč zgolj okolju. Za razliko od dogodkov, ki jih uporabljajo implementacije OpenCL, je pri teh mogoče nastavljanje stanja dogodka. Metoda za nastavljanje stanja sprejme katerikoli dogodek, a deluje le na uporabniških dogodkih [29]. To je dober razlog za implementacijo le-teh s posebnim razredom. Stanje, ki ga je mogoče nastaviti, je lahko le zaključeno stanje ali koda napake.

Potrebna sta torej dva razreda. Osnovni razred za vse dogodke, ki bo omogočal pridobivanje informacij, čakanje in prijavljanje na povratne klice ter razred za uporabniške dogodke, ki bo dedoval od tega in omogočal zaključevanje dogodka. S tem se omogoči objektno usmerjen način uporabe knjižnice. S pomočjo povratnih klicev je mogoče tudi dogodkovno vodeno programiranje [40]. A ta način ni najbolj primeren za jezik C#. Različica

5 je namreč uvedla nov pristop k asinhronemu programiranju s pomočjo ključnih besed `async` in `await` [6]. Z različico 7.0 je C# dobil podporo za čakanje (s pomočjo `await`) na poljubne podatkovne tipe [96, 5, 42]. Z dodajanjem metode `GetAwaiter`, ki vrne instanco razreda ali strukturo, ki vsebuje lastnost `IsCompleted`, metodo `GetResult` in implementira ali vmesnik `INotifyCompletion` ali vmesnik `ICriticalNotifyCompletion`, je mogoče prevajalnik naučiti, kako uporabiti poljuben podatkovni tip znotraj avtomata, ki se uporablja za izvajanje asinhronih metod. V primeru vseh dogodkov OpenCL metodi `GetResult` ni potrebno vračati ničesar, saj dogodki ne proizvajajo rezultatov in temu primerno nimajo generičnih parametrov. To pomeni, da mora metoda `GetResult` zgolj čakati, da se dogodek zaključi, če je v stanju teka. In to OpenCL podpira. Za implementacijo obeh vmesnikov je potrebno le dodajanje metod, preko katerih se je mogoče prijaviti za povratne klice. In tudi to je podprto s strani OpenCL. Edini manjkajoči del implementacije je prijavljanje na povratni klic. Za razliko od povratnega klica konteksta, dogodki omogočajo prijavljanje po ustvarjanju dogodka in ob klicu funkcij podajo vrednost strukture dogodka. Ker so, tako kot kontekсti, lahko tudi dogodki pobrisani pred izvedbo povratnega klica, je tudi pri njih potrebno za povratni klic uporabiti statično metodo. Posredovanje pravemu dogodku je v tem primeru zaradi podane vrednosti dogodka mogoče brez dodatnih identifikatorjev. To pomeni, da je težava z iskanjem prave instance razreda rešena trivialno. Edina težava je iskanje pravega delegata. Tej težavi se je mogoče izogniti tako, da seznam poslušalcev hrani razred. Zadnji dodatek, ki naredi dogodke bolj sorodne opravilom .NET, ki se navadno uporabljajo skupaj s ključno besedo `await`, je dodajanje samodejnega pošiljanja dogodkov v izvajalno vrsto.

Poleg seznama poslušalcev dogodka lahko razred hrani tudi nekaj drugih stanj. In sicer logično vrednost, ki določa, če je dogodek zaključen, kodo napake dogodka in logično vrednost, ki določa, če je bil dogodek odposlan na napravo, ki ga mora izvesti. Dostop do metod, ki spreminjajo ta stanja, mora biti sinhroniziran. A dostop do metod, ki to stanje le berejo, ni ve-

dno potreben. Tak primer predstavlja metoda `GetResult`, ki mora čakati na zaključek dogodka. Ta namreč lahko čakanje preskoči, če je dogodek zaključen. Preverjanje stanja bi lahko bilo sinhronizirano. A ker je čakanje zaključenega dogodka dovoljeno, je to sinhronizacijo mogoče izpustiti, saj je pogosto nepotrebna. Izogibanje cene zaklepanja je pomembno, saj je edini dober razlog za ohranjanje ločenega stanja o dogodkih, kljub temu da to ponuja OpenCL, hitrejši dostop do teh podatkov. In ker je to stanje le kopija stanja OpenCL, je treba skrbeti, da se ta z njim tudi ujema. To pomeni, da morajo vsi dogodki, katerih zaključka ni sprožila knjižnica, spremljati stanje dogodka preko povratnega klica. Tudi če teh dogodkov ne spremlja nihče. Ena izmed prednosti spremljanja stanja dogodkov je poznavanje napake v primeru, da je dogodek zaključen neuspešno. Težava s čakanjem na dogodke je, da vse funkcije podpirajo delo z več dogodki. Zaradi tega, tudi če uporabnik funkcije čaka le na en dogodek, te funkcije vrnejo splošno napako vrste dogodkov in ne napake, zaradi katere dogodek ni uspel. Ta način delovanja je ohranjen v statičnih metodah za čakanje na več dogodkov, a spremenjen na metodi instance, ki čaka le na en dogodek. Ta lahko namreč izkoristi ta podatek in vrne pravo napako.

Ker so nekateri dogodki ustvarjeni kot odziv na operacijo nad pomnilniškim objektom, je lahko uporabniku knjižnice olajšana uporaba tako, da se tem dogodkom doda umeten rezultat. Primer take operacije je branje medpomnilnika. Navadno bi moral uporabnik sprožiti branje, s čakanjem na dogodek ali izvajalno vrsto počakati, da se zaključi. Podatki so namreč prebrani šele, ko se branje zaključi. Če bi metoda za branje vrnila dogodek, bi bil lahko rezultat tega dogodka polje, v katero so se prebrale vrednosti. V tem primeru lahko uporabnik uporabi `await` in s tem iz dogodka pridobi prebrano polje. Ta vzorec uporabe ima izreden pomen. Brez možnosti čakanja na rezultat bi moral imeti uporabnik knjižnice pred klicem metode shranjeno referenco na polje, v katerem bo rezultat. Z možnostjo čakanja, lahko to polje ustvari metoda samostojno. Za dodajanje podpore za dogodke z rezultati je dodan nov razred, ki deduje od originalnega razreda za dogodke. Ta ima generični

parameter, ki določa podatkovni tip rezultata in hrani vrednost, ki bo podana kot rezultat. Da ta razred pravilno vrne rezultat, kadar je uporabljen z `await`, je treba zanj na novo implementirati metodo `GetAwaiter`.

Uporabnost in združljivost z opravili oziroma razredom `Task`, ki je del orgodja .NET, je mogoče izboljšati z metodami za pretvarjanje med dogodki in opravili. Metoda za ustvarjanje opravila iz dogodka je lahko narejena s pomočjo razreda `TaskCompletionSource`. Ta razred omogoča predstavitev zunanjih asinhronih operacij in ustvarjanje objektov, torej instance razreda `Task`, preko katerih je mogoče spremljati te operacije [86, 87]. Drugo stran teh pretvorb predstavlja metoda za pretvarjanja opravila v dogodek. To je narejeno z ustvarjanjem uporabniškega dogodka, ki je zaključen, ko se dokonča opravilo.

Zadnja izboljšava je samodejno sproščanje dogodkov. Možnost čakanja na dogodke pomaga narediti knjižnico bolj prijazno za uporabnike. A ta poenostavitev hkrati poenostavi tudi eno izmed pogostih napak uporabe OpenCL. Objekti, ki so ustvarjeni tekom izvajanja programa morajo biti pobrisani. Ker mehanizem za brisanje teh objektov temelji na štetju referenc, mora uporabnik vsak objekt, ki je bil ustvarjen kot posledica njegove uporabe vmesnika, tudi sprostiti. Pri dogodkih se uporabniki temu lahko izognejo tako, da jih ne uporabljajo. A ker mora knjižnica ponuditi način čakanja na asinhrone operacije in ker je priporočen način dela z asinhronimi operacijami v C# uporaba `async` in `await`, mora knjižnica zato uporabiti dogodke. Te lahko uporabi uporabnik knjižnice, a to ni nujno. Uporabnik jih v primeru uporabe vrste z zaporednim izvajanjem lahko popolnoma izpusti. Čakanje na zaključek zadnjega dogodka lahko namreč doseže tako, da čaka le na zadnji dogodek. In ker dogodkov ni ustvaril uporabnik knjižnice, jih v primeru, da jih ne uporabi, ni dolžen sprostiti. To je mogoče rešiti z dodajanjem podpore samodejnega sproščanja dogodkov po njihovem zaključku. Po tem ostane le težava s skrivanjem te zmogljivosti. Uporabnik knjižnice, ki vidi, da je ime podatkovnega tipa `Event`, ne bo vedel, da bo ta mogoče sproščen samodejno, če ne prebere dokumentacije. To lahko pripelje do podobne težave, le da so

zdaj dogodki nezaželeno sproščeni. Novo težavo so lahko odpravi z uvajanjem ločenega podatkovnega tipa. Ta lahko ali deduje od razreda za dogodke ali uporabi kompozicijo. Če od njega deduje, lahko uporabnik ta dogodek pomotoma shrani v spremenljivko tipa `Event` in s tem izgubi to informacijo. Če uporabi kompozicijo in hrani referenco do originalnega dogodka, to ni mogoče. Novemu razredu se doda tudi metoda za pretvorbo začasnega dogodka v pravi dogodek, nekaj izmed metod iz originalnega razreda za dogodke, ki so smiselne, tudi ko je dogodek začasen, in metode za implicitno pretvarjanje v originalni razred dogodkov in strukturo dogodkov. Pretvorba v strukturo dogodkov je uporabna, ko uporabnik knjižnice želi izvesti nekaj, kar ni podprto, oziroma dogodek uporabiti izven knjižnice. V tem primeru je mogoče, da želi delati z začasnim dogodkom, zato je dovolj vrniti strukturo, ki jo hrani originalni dogodek. Pretvorba v originalni podatkovni tip dogodka je nekoliko drugačna. V primeru, ko uporabnik knjižnice pride do tega, so mu omogočene operacije, ki niso smiselne za začasne dogodke. Zato je smiselno v primeru take pretvorbe onemogočiti samodejno sproščanje dogodka.

S pomočjo ustvarjenih ovojnih razredov za izvajalne vrste in dogodke je mogoče te uporabiti na način, ki ga prikazuje slika 4.3.

```
public static async Task Main(string[] args)
{
  var device = Device.QueryDevices(DeviceType.Gpu).First();

  using (var queue = CommandQueue.Create(device, profiling: true))
  using (var userEvent = UserEvent.Create(queue))
  using (var @event = Event.WhenAll(queue, blocking: false, userEvent))
  {
    var task = Task.Run(() => WaitForEvent(@event));

    await Task.Delay(1000);
    userEvent.SetCompleted();
    await task;
  }
}

private static async Task WaitForEvent(Event @event)
{
  Console.WriteLine("Waiting for event");
  await @event;
  Console.WriteLine("Event completed with status {0}", @event.Status);
}
```

Slika 4.3: Primer uporabe dogodkov.

4.4 Delo s pomnilnikom

Ena izmed težav uporabe knjižnic, ki ne tečejo znotraj izvajalnika .NET ogrodja, je deljenje pomnilnika. Zlasti zahtevno je podajanje referenc na vrednosti, kar je seveda potrebno za prenos polj. To je zahtevno, ker znotraj izvajalnika pomnilnik upravlja GC. In ta lahko poljubno premika vrednosti v pomnilniku, saj so dostopi do teh znotraj izvajalnika izvedeni preko posebnih referenc in ne pomnilniških naslovov. Ko se vrednosti, s katerimi upravlja GC, deli z deli programa, ki se ne izvajajo znotraj izvajalnika, ti za dostop potrebujejo pomnilniški naslov. A ker lahko GC premika vrednosti po pomnilniku, se pomnilniški naslovi vrednosti lahko spremenijo. En način reševanja te težave je s fiksiranjem vrednosti. S tem se lahko prepreči premikanje vrednosti [32]. A dolgotrajna nepremičnost vrednosti lahko povzroči težave, kot so razdrobitev pomnilnika [79]. Ker je medpomnilnike in slike OpenCL mogoče ustvariti z vnaprej pripravljenim pomnilnikom in ker ti pomnilniški objekti lahko obstajajo dalj časa, ima lahko ustvarjanje teh s pomočjo .NET polj negativen vpliv na hitrost izvajanja programa. Uporabnik knjižnice se temu lahko izogne tako, da podatke za uporabo z medpomnilniki in slikami pripravi v pomnilniku, katerega ne upravlja GC. A je to precej zahtevno v primerjavi z uporabo polj. Dodajanje podpore za ta način uporabe zahteva tudi dodajanje novega načina uporabe knjižnice. Dobra rešitev za to težavo bi uporabniku omogočala podajanje reference na poljubno zaporedje vrednosti in omogočala samodejno fiksiranje, če je to potrebno. To težavo rešuje struktura `Memory` [66]. Ta namreč predstavlja poljubno zaporedje vrednosti v pomnilniku. Z njo je mogoče delati s polji, deli polj in vsemi pomnilniškimi strukturami, ki implementirajo razred `MemoryManager` [65]. To pomeni, da je mogoče razširiti strukturo `Memory` in ji dodati podporo za nove podatkovne strukture. Za implementacijo razreda `MemoryManager` je potrebno poskrbeti le, da je do elementov v podatkovni strukturi mogoče dostopati preko polja, reference ali kazalca, ter da je mogoče podatkovno strukturo fiksirati. Zaradi teh omejitev je struktura `Memory` popolna za ta primer uporabe, saj zagotavljajo vse potrebno za varno podajanje podatkovne strukture OpenCL.

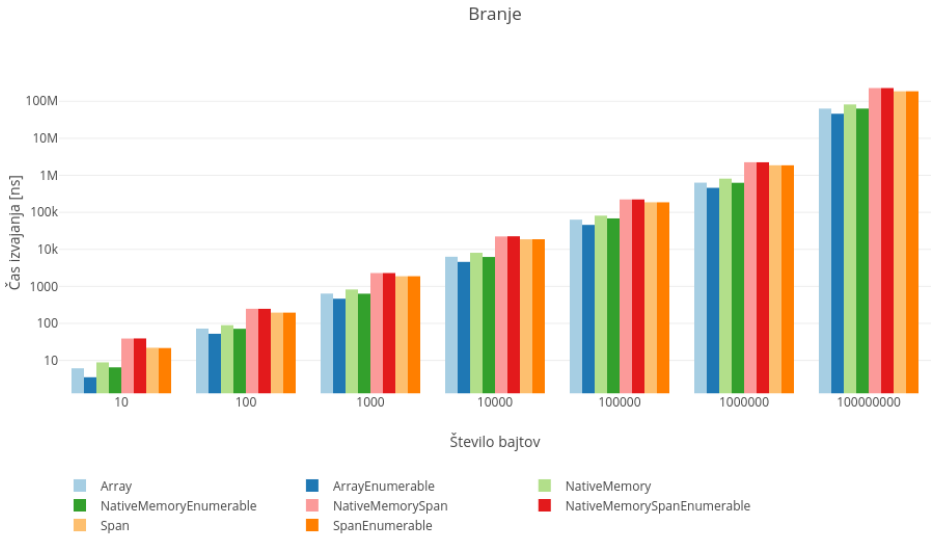
Uporaba strukture `Memory` torej omogoča uporabniku knjižnice, da tej poda pomnilnik, ki ni pod nadzorom GC in s tem omogoča način uporabe, ki ne škoduje GC. Ena izmed bolj popularnih knjižnic, ki omogoča delo z nenadzorovanim pomnilnikom je jemalloc.NET. A ta žal ni bila posodobljena že dalj časa in ne deluje z najnovejšimi različicami implementacij .NET, saj so te spremenile poimenovanje razreda `OwnedMemory` v `MemoryManager`.

Za namen knjižnice je sicer dovolj, da je njen uporabnik zgolj obveščen o težavah, v primeru uporabe navadnega polja. A je bolje od tega ponuditi implementacijo razreda `MemoryManager`, ki uporablja vgrajene metode za delo z nenadzorovanim pomnilnikom. Čeprav je implementacija takega razreda precej enostavna, to ni popolnoma trivialna naloga. Ta razred namreč deluje kot alternativa poljem in polja so v .NET izredno optimizirana. V knjižnici se razred imenuje `NativeMemory`. Da bo razred uporabljen kot alternativa poljem, mora poleg hitrosti zagotoviti tudi uporabnost. Za zagotavljanje hitrosti je potrebno profiliranje. To pokaže, kateri del razreda je najpočasnejši. Poleg merjenja časa izvajanja posameznih delov kode je mogoče tudi merjenje časa izvajanja različnih operacij novega razreda v primerjavi s polji. To lahko pokaže očitno počasnejše operacije in pomaga poiskati počasne dele implementacije. Razred sicer nima veliko različnih operacij. Najbolj pomembni operaciji sta namreč branje in pisanje vrednosti. In čeprav sta ti operaciji precej enostavni, ju je zelo preprosto narediti precej počasnejši od operacij na poljih. Prvi pomembni dejavnik je način dostopa. Ker razred dela s pomnilnikom, ki ga na nadzoruje GC, dostopi do njega ne potekajo preko reference GC, temveč kazalca. V C# je mogoče s kazalci delati preko posebnih metod ali neposredno. Neposredna uporaba uporablja posebne ukaze za indirektno nalaganje [74], a je za to zahtevana uporaba besede `unsafe` [90]. Prednost neposrednega dostopanja preko kazalcev je izogibanje dodatnih klicev. Drugi pomembni dejavnik je preverjanje dostopov. C# prevajalnik je včasih namreč sposoben ugotoviti, ali so vsi dostopi do polja veljavni pred zagonom programa. To je razvidno iz prevedene vmesne kode, saj v teh primerih koda ne preverja indeksov. Isto je mogoče doseči v razredu `NativeMemory` z

odstranjevanjem teh preverjanj. A se s tem izgubi nekaj varnosti uporabe. Za to je v knjižnici na voljo tudi implementacija `SafeNativeMemory`, ki je identična `NativeMemory`, le da ta preverja indekse. Nazadnje so pomembne še metode, preko katerih potekajo dostopi. .NET vmesni jezik namreč pozna več ukazov za klicanje metod. Glavna sta ukaza `call` in `callvirt`. Prevajalnik uporabi ukaz `callvirt` ob klicih navideznih metod in ob klicih metod referenčnih tipov, kadar je mogoče, da se bo klic izvršil na ničli referenci [76]. Najzanesljivejši način preprečevanja preverjanja ničle reference je s spremembo podatkovnega tipa. Podatkovni tipi, definirani kot strukture, se namreč nahajajo na skladu in niso shranjene preko reference. Prav tako njihove metode ne morejo biti navidezne. Zato prevajalnik za njih uporabi ukaz `call`. Čeprav mora biti podatkovni tip `NativeMemory` razred, saj je tako definiran `MemoryManager`, je lahko podatkovni tip, ki se uporablja za štetje, struktura.

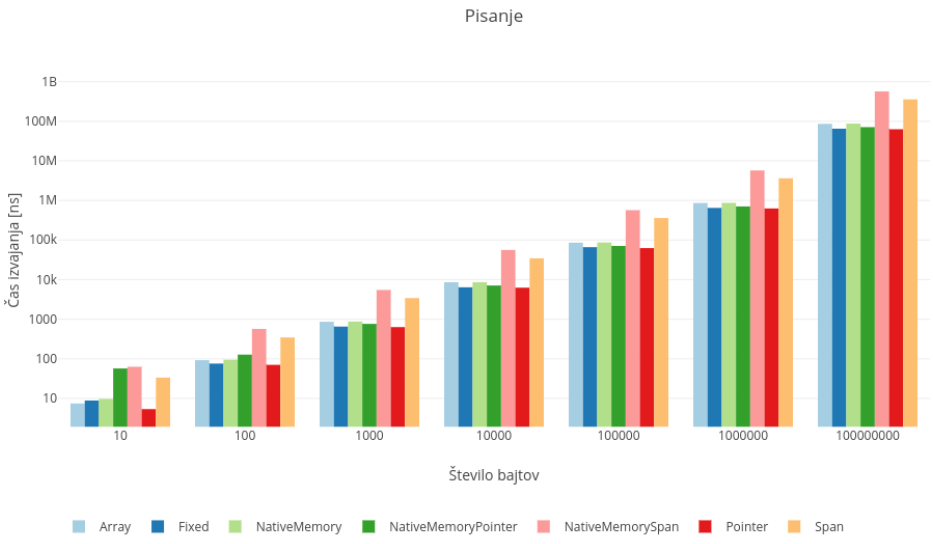
Hitrost dostopov razreda `NativeMemory` je s pomočjo teh izboljšav primerljiva s hitrostjo dostopa do polj. To naredi razred primeren kot alternativo poljem. V nekaterih primerih je uporaba tega razreda lahko celo hitrejša. Njegova glavna prednost in hkrati pomanjkljivost je namreč, da ustvarjanje nove instance ne nastavi vseh vrednosti elementov na privzete vrednosti. Polja v C# so po ustvarjanju namreč vedno nastavljena na ničlo vrednost danega podatkovnega tipa. In čeprav je to nastavljanje precej hitro, to še vedno upočasni ustvarjanje polja. Zaradi tega se lahko `NativeMemory` primerja s polji, kljub temu, da si ne more pomagati s hitrejšimi alokacijami pomnilnika, ki jih lahko pod določenimi pogoji ponudi GC.

Slika 4.4 primerja čase ustvarjanja in branja vseh elementov različnih podatkovnih struktur različnih velikosti. Primerjani so časi ustvarjanja in branja iz polja, `NativeMemory`, `NativeMemory` preko strukture `Span` in nenadzorovanega pomnilnika preko strukture `Span`. Za vsakega izmed načinov branja je izmerjen tudi čas s pomočjo zanke `foreach`. Ker zanka `foreach` za svoje delovanje uporablja vmesnik `IEnumerable`, so ti časi poimenovani s pripono `Enumerable`.



Slika 4.4: Povprečni časi branja iz sekvenčnih pomnilnikov.

Slika 4.5 primerja čase ustvarjanja in pisanja na vsako mesto različnih podatkovnih struktur. Primerjani so časi ustvarjanja in pisanja v polje, fiksirano polje, `NativeMemory`, `NativeMemory` preko kazalca, `NativeMemory` preko strukture `Span`, kazalca na nenadzorovan pomnilnik in v nenadzorovan pomnilnik preko strukture `Span`.



Slika 4.5: Povprečni časi pisanja v sekvenčne pomnilnike.

Da bi bil novi razred bolj uporaben, se lahko na njem implementira vmesnik `IEnumerable`. S tem se omogoči uporaba razširitvenih metod, ki jih definira statični razred `Enumerable` in vgrajenih poizvedb LINQ [38, 50]. A pomemben del razširitvenih metod razreda `Enumerable` so tudi metode za grajenje zbirk iz enumeratorjev. Za omogočanje izgradnje zbirke `NativeMemory` iz poljubnega enumeratorja je potrebno definirati novo razširitveno metodo za vmesnik `IEnumerable`. Za razliko od grajenja kolekcij s spremenljivim številom elementov, kot sta `List` in `Set`, je gradnja zbirke `NativeMemory` bolj podobna gradnji polja. Velikost pomnilnika, ki ga zasedata ti strukturi, je namreč konstantna. To pomeni, da mora metoda `ToArray` izdelati polje točno določene velikosti, čeprav ta morda ni znana vnaprej. Ob gradnji ima namreč dostop le do enumeratorja, ki ga ni vedno mogoče prevrteti nazaj. Zato ni vedno mogoče najprej prešteti elemente, izdelati polje in nato vstaviti elemente v polje. Prav tako to ni vedno učinkovito, saj lahko enumerator vsebuje kompleksne preobrazbe, ki trajajo dalj časa. Enaka težava doleti ustvarjanje zbirke `NativeMemory`. V primerih, ko je mogoče brez dodatnega dela preveriti končno število elementov, je mogoče to optimizirati. A v primerih, ko to ni mogoče, je potrebno podatke najprej shraniti v vmesno zbirko. Ta vmesna zbirka je lahko seznam oziroma zbirka zaporednih vrednosti spremenljive dolžine. Taka zbirka v C# že obstaja in se imenuje `List`. A ta ni nujno najbolj učinkovita. Za svoje delovanje namreč uporablja polje, katerega velikost se samodejno podvoji, kadarkoli uporabnik dodaja nove elemente v polno zbirko. Težava v tem pristopu je, ko je elementov že veliko. Večanje majhnega polja je precej preprosta operacija, saj lahko ta v primeru, ko je v pomnilniku za poljem dovolj neuporabljenega prostora, uspe brez kopiranja vrednosti. In čeprav je večanje večjega polja v resnici ista operacija, mora ta z večjo verjetnostjo ustvariti popolnoma novo večje polje in vanj skopirati vse vrednosti originalnega polja. Ko je velikost polja zelo velika, lahko čas večanja polja postane nezanemarljiv. En način reševanje te težave predstavlja uporaba polj, ki držijo druga polja. Glavno polje je manjše in hrani le reference na druga polja, ta držijo podatke. Torej namesto da se

hranijo vse vrednosti v enem polju, se hranijo v dodatnih poljih. Ko je zadnje izmed teh polj polno, se ustvari novo, dvakrat večje polje in vstavljanje elementov se nadaljuje v tem polju. Če glavno polje postane premajhno, se njegova velikost poveča s pomočjo kopiranja. Glavna prednost tega pristopa je da se število elementov v glavnem polju povečuje približno s korenom števila elementov. Če ima prvo polje z elementi dolžino 64, mora glavno polje za hranjenje milijon elementov držati le 14 elementov. In če se glavno polje povečuje z večkratnikom 2 in ima ob ustvarjanju strukture dolžino 4, se mora to povečati le dvakrat. Slabost tega pristopa je zapletenost vstavljanja in brisanja elementov na poljubnih mestih. A tega podatkovna struktura, katere edina naloga je začasno hranjenje elementov, ki so vedno vstavljeni na konec zbirke, ne potrebuje. Podobna rešitev se lahko doseže tudi z uporabo povezanega seznama namesto polja za glavni seznam. Uporaba povezanega seznama za hranjenje elementov ni dobra rešitev. Ta podatkovna struktura je namreč namenjena začasni hrambi elementov enumeratorja in kopiranju teh elementov v `NativeMemory`. Ker so podatki v `NativeMemory` shranjeni sekvenčno, je mogoče podatke vanj kopirati v blokih. A za kopiranje blokov podatkov ni dovolj, da je sekvenčno urejen le končni pomnilnik, temveč tudi izvorni pomnilnik. Zato je najprimernejša struktura za hranjenje elementov polje.

Ustvarjeni razred `NativeMemory` omogoča uporabo nenadzorovanega pomnilnika na način, ki ga prikazuje slika 4.6.

```
using (var memory = new NativeMemory<int>(0x60000000))
{
  Console.WriteLine("Allocated {0} bytes of native memory",
    memory.Size);

  for (var i = 0L; i < memory.LongLength; i++)
    memory[i] = unchecked((int) i);

  Console.WriteLine("Wrote {0} values into native memory",
    memory.LongLength);

  var sum = 0D;

  for (var i = 0L; i < memory.LongLength; i++)
    sum += memory[i];

  Console.WriteLine("Sum of all values {0}", sum);
}
```

Slika 4.6: Primer pisanja in branja pomnilnika z razredom `NativeMemory`.

4.5 Pomnilniški objekti

OpenCL API razlikuje med dvema vrstama pomnilniških objektov. Čeprav definira le eno podatkovno strukturo za pomnilniške objekte, definira ločene funkcije za medpomnilnike in slike. Ločene metode so potrebne zaradi načina predstavitve podatkov. Medtem ko so medpomnilniki zgolj zaporedje vrednosti, so slike predstavljene kot zaporedje slikovnih elementov. Slikovni elementi so lahko shranjeni na različne načine. Vsako sliko namreč določa podatkovna struktura `cl_image_format` [26]. Ta določa kanale in podatkovni tip slikovnih elementov vsake slike. To pomeni, da je interpretacija in velikost vsakega od slikovnih elementov lahko drugačna med različnimi slikami. Poleg tega so za razliko od medpomnilnikov slikovni elementi urejeni neprekinjeno le znotraj posamezne vrstice. V večdimenzionalnih slikah in enodimenzionalnih poljih slik so vrstice namreč dolge točno določeno število bajtov. To število določa vrednost `image_row_pitch` v podatkovni strukturi `cl_image_desc` in mora biti vsaj velikost slikovnih elementov v bajtih pomnožena s številom slikovnih elementov v vrstici [25]. Podobno velja za dvodimenzionalne rezine slike v primeru tridimenzionalnih slik in polj dvodimenzionalnih slik. Velikost vrstic in dvodimenzionalnih rezin slike je v marsikaterem primeru lahko kar najmanjša, ki lahko hrani celotno vrstico oziroma dvodimenzionalno rezino slike. A to ni vedno mogoče, saj je včasih treba vrstice in dvodimenzionalne rezine poravnati na določeno število bajtov. En primer tega je delo z razredom `Bitmap`, ki ga ponuja .NET Framework. Ta zahteva, da so vrstice slike vredno poravnane na štiri bajte [7]. V tem primeru lastnosti `Stride` ne določa le velikosti vrstice v bajtih, temveč tudi orientacijo slike. V primeru, da je slika orientirana od zgoraj navzdol in je `Stride` pozitivna vrednost, je mogoče sliko pridobljeno iz razreda `Bitmap` uporabiti z OpenCL neposredno. V nasprotnem primeru je potrebno sliko ali obrniti ali podati obrnjeno na glavo, tako da se za velikost vrstice uporabi absolutna vrednost lastnosti `Stride`.

OpenCL do neke mere ločuje tudi različne vrste slik. Te so namreč lahko enodimenzionalne, dvodimenzionalne, tridimenzionalne, polja enodimenzio-

nalnih slik in polja dvodimenzionalnih slik. Čeprav se za vse uporabljajo iste metode, je uporabe teh metod različna glede na vrsto slike. Ta se najbolje pozna na funkciji `clEnqueueCopyImage`, saj ta deluje drugače glede na vrste slike, med katerimi se kopira vsebina [17].

Knjižnica zato definira več vrst ovojnih razredov. En abstrakten razred za pomnilniške objekte, ki omogoča operacije, ki so identične za vse vrste pomnilniških objektov. Definira tudi dva razreda za medpomnilnike. Prvi ni omejen z generičnim parametrom in omogoča operacije, ki so neodvisne od vsebine medpomnilnika, ter razred z generičnim parametrom, ki omogoča vse operacije nad medpomnilniki in pri tem upošteva velikost elementov medpomnilnika. Nazadnje knjižnica definira razrede za slike. Podobno kot pri medpomnilnikih se tudi tu razredi delijo na te, ki nimajo generičnega parametra in niso odvisni od vsebine, ter na te, ki od njih dedujejo, imajo generični parameter in omogočajo tudi operacije, ki so odvisne od vsebine slike. Ti razredi so definirani za vse vrste slik.

Prvi funkciji, ki jih morajo podpirati ti razredi, sta funkciji za ustvarjanje medpomnilnikov in slik. Čeprav OpenCL definira le dve funkciji, eno za ustvarjanje medpomnilnikov in drugo za ustvarjanje slik, je mogoče pomnilniške objekte ustvariti na različne načine. Če ob ustvarjanju pomnilniških objektov ni podana nobena zastavica, OpenCL ves pomnilnik za pomnilniški objekt dodeli sam. Ob ustvarjanju je mogoče implementaciji OpenCL podati tudi kazalec na pomnilniški prostor, iz katerega OpenCL kopira začetne vrednosti pomnilniškega objekta. Nazadnje je mogoče ob ustvarjanju podati kazalec in od implementacije zahtevati, da za hranjenje pomnilniškega objekta, ko se nahaja v pomnilniku gostitelja, uporabi pomnilnik, na katerega kaže kazalec. A to ne pomeni, da se mora pomnilniški objekt vedno nahajati na želenem delu pomnilnika, saj lahko implementacija OpenCL predpomni celoten ali le del pomnilniškega objekta na poljubnem mestu v pomnilniku [10]. Da se pomnilniški objekt oziroma del pomnilniškega objekta nahaja tam, je zagotovljeno le, če je ta preslikan v pomnilniški prostor gostitelja [19]. Kadar je pomnilniški objekt ustvarjen tako, da implementacija dodeli pomnilnik, je

mogoče od nje zahtevati, da dodeli pomnilnik iz gostitelju dosegljivega naslovnega prostora. OpenCL sicer to mora narediti ob preslikavi pomnilnika, a specifikacija ne zahteva, da mora pomnilnik, v katerega se naredi preslikava obstajati po koncu preslikave. S to zastavico je mogoče omejiti število kopij pomnilnika ob preslikavah. Nazadnje je ne glede na način ustvarjanja pomnilnika s pomočjo zastavic mogoče omejiti branje ali pisanje s strani gostitelja ali naprav. S temi zastavicami je mogoče implementaciji OpenCL namigniti, kaj se bo dogajalo s pomnilniškim objektom in preprečiti nepotrebna kopiranja. Če iz nekega pomnilniškega objekta gostitelj le bere in naprave le pišejo, lahko implementacija preskoči kopiranje vrednosti iz gostiteljevega pomnilnika v pomnilnik naprave, ko se položaj pomnilniškega objekta premakne od gostitelja ali ene izmed naprav na drugo napravo. Na vseh razredih pomnilniških objektov so zato definirane različne metode za ustvarjanje pomnilniških objektov. Prva je `Create`, ki ustvari pomnilniški objekt tako, da implementacija dodeli potreben prostor. Pri tem je mogoče določiti tudi, ali naj bo pomnilnik dodeljen v naslovnem prostoru, ki je dosegljiv gostitelju. A je pri tem zastavica preimenovana v `usePinnedMemory`, saj dodeljevanje v naslovnem prostoru pomnilnika ni onemogočeno brez te zastavice, njena prisotnost namreč zahteva le, da je ta prostor nepremičen. Izraz `pinned` je bil izbran, ker opisuje glavni pomen te zastavice in se pojavlja na različnih forumih. Metoda `Create` na razredih za slike ne sprejme velikosti vrstic in velikosti dvodimenzionalnih rezin slike, saj OpenCL ne omogoča določanja teh, kadar ni podan kazalec na pomnilniški prostor, ki se mora uporabiti za kopiranje začetnih vrednosti ali hranjenje slike znotraj pomnilniškega prostora gostitelja. Metoda `From` omogoča ustvarjanje pomnilniškega objekta, za katerega prostor dodeli implementacija OpenCL, a se njene vrednosti kopirajo iz podane strukture tipa `Span`. Ker `Span` ni edini splošni način predstavitve zaporednega pomnilnika, obstaja tudi metoda `FromMemory`, ki je identična prejšnji, a omogoča uporabo strukture `Memory`. Zadnja pomembna metoda se imenuje `For`. Ta namesto strukture `Span` ali strukture `Memory` prejme implementacijo razreda `MemoryManager` in ustvari pomnilniški objekt, ki se

v pomnilniku gostitelja hrani znotraj pomnilnika, ki ga nadzoruje `Memory Manager`. Dodatna metoda `ForMemory` naredi enako, le da prejme strukturo `Memory`. Za razliko od metode `For`, ta ne omogoča uporabo strukture `Span`, saj je to mogoče fiksirati le za čas izvajanje metode. Ločeni metodi za `Memory Manager` in `Memory` obstajata zaradi manjše razlike v upravljanju pomnilnika. Metoda `For` namreč uporabi metodo `Dispose` podane implementacije `Memory Manager` za sproščanje pomnilnika po uničenju pomnilniškega objekta. Vse metode ustvarjanja slik morajo prejeti dovolj podatkov za ustvarjanje struktur `cl_image_format` in `cl_image_desc`. Za strukturo `cl_image_desc` je dovolj, da metode prejmejo dimenzije slik, neobvezni velikosti vrstice in dvodimenzionalne rezine slike. Za strukturo `cl_image_format` sta potrebni ureditev kanalov in podatkovni tip kanalov slike. Ker razredi za slike določajo podatkovni tip elementov s pomočjo generičnega parametra, je za nekatere podatkovne tipe mogoče določiti vrednost te strukture zgolj glede na ureditev kanalov in način dostopa do podatkov znotraj gonilnikov, normalizirano ali ne, je za te metode mogoče dodati tudi različico, ki zahteva le ta dva podatka.

Obstaja še en poseben način ustvarjanja enodimenzionalnih slik. In sicer s pomočjo medpomnilnika. OpenCL loči tako vrsto slike, a to ne pomeni, da je uporabljanje take slike drugačno od uporabe navadne enodimenzionalne slike. Zato se lahko za tako sliko uporabi obstoječi razred za enodimenzionalne slike. Potrebuje le novo metodo za ustvarjanje, ki se imenuje `ForBuffer`.

Za branje in pisanje pomnilniških objektov so potrebne različne metode. Poleg preprostega branja in pisanje, ki se razlikuje med različnimi vrstami pomnilniški objektov, OpenCL omogoča tudi branje eno, dve ali tridimenzionalnih regij slik. Medtem ko sta položaj in velikost regije branja ali pisanja slik vedno podani v slikovnih elementih, OpenCL odmik in dolžino pri branju medpomnilnikov sprejme v bajtih. Ker razred za medpomnilnike pozna podatkovni tip elementov, lahko te metode ti dve vrednosti popravita primerno. Vse te metode omogočajo asinhron in sinhron način izvajanja njihove operacije. Glavna prednost sinhronega načina je, da je preprostejša za uporabo,

saj ni potrebe po ustvarjanju dogodkov in spremljanju njihovega stanja. A ker knjižnica omogoča precej enostavno delo z dogodki, ta potreba tu ni precej velika, saj lahko uporabnik vedno sinhrono počaka na dogodek. Ker knjižnica podpira dogodke, ki se počistijo sami, uporabniku ni treba zanje skrbeti ročno, kar pomeni, da je edini negativni način izključevanja možnosti sinhrone uporabe knjižnice začasno ustvarjanje dogodkov. V primeru pisanja morajo metode zato sprejeti strukturo `Memory`, saj mora biti njena vsebina fiksirana dlje, kot se izvaja metoda. V primeru branja so stvari malo bolj zapletene. Ker knjižnica definira dogodke, ki lahko držijo vrednost, je smiselno iz metod za branje vrniti tak dogodek. Na ta način lahko knjižnica skrije pomnilnik, v katerem bo shranjen rezultat, dokler dogodek ni izveden do konca. To pomaga preprečiti kakšno napako, a ima svojo ceno. Vrednost v dogodku mora imeti namreč točno določen podatkovni tip, in čeprav je mogoče veliko število podatkovnih tipov, ki podatke hranijo zaporedno, pretvoriti v strukturo `Memory`, to spremeni podatkovni tip. Zato obstaja več različic metod za branje. Prva nabora metod delata in vračata dogodke, ki vsebujejo navadna polja. Ena izmed različic polje ustvari sama, druga prejme polje, ki že obstaja. Tretji nabor različic omogoča branje v strukturo `Memory`, ki jo mora podati uporabnik. Četrti nabor različic omogoča branje v podatkovni tip `NativeMemory`, ki je definiran v tej knjižnici. Ta nabor različic `Native Memory` ustvari samodejno. Zadnji nabor različic metod za branje omogoča branje v poljuben podatkovni tip, ki deduje od razreda `MemoryManager`. Ta nabor različic v dogodku vrača vrednost, ki je istega tipa kot podan pomnilnik. To pomeni, da je z njimi mogoče uporabiti tudi `NativeMemory`, ne da se izgubi informacija o pravem podatkovnem tipu pomnilnika, v katerega se bere medpomnilnik ali slika.

Pomnilniške objekte je mogoče tudi kopirati v druge pomnilniške objekte. OpenCL za to definira pet metod. Eno za kopiranje iz medpomnilnika v medpomnilnik, drugo za kopiranje regije medpomnilnika v regijo drugega medpomnilnika, tretjo za kopiranje medpomnilnika v sliko, četrto za kopiranje slike v medpomnilnik in peto za kopiranje slike v sliko. Izmed vseh metod za

delo s pomnilniškimi objekti tu najbolje pride do izraza smiselnost različnih različic metod. OpenCL pri delu s pomnilniškimi objekti namreč ne zahteva velikosti vrstic in dvodimenzionalnih rezin. Te vrednosti so lahko nastavljene na nič in v tem primeru jih OpenCL določi samodejno. A način določanja teh vrednosti vedno upošteva le velikosti regij operacij in ne položaja regij. To pomeni, da je velikost vrstice v bajtih poračunana kot produkt širine regije in velikosti slikovnih elementov. A to v primerih, ko se operacije izvajajo nad medpomnilniki, ni pravilno, če ima regija odmik. Zato lahko obstajajo različice metod, ki berejo, pišejo in kopirajo celotne pomnilniške objekte in regije pomnilniških objektov, in ne potrebujejo teh velikosti. A ne morejo obstajati različice metod, ki omogočajo branje, pisanje ali kopiranje regij z odmikom, ki ne potrebujejo teh velikosti. Pomanjkanje obstoja takih različic metod je pomembno, ker funkcije kot je `clEnqueueReadBufferRect` po specifikaciji ne zaznavajo te napake [22].

Poleg preprostega branja, pisanja in kopiranja pomnilniških objektov je te mogoče tudi zapolniti z določeno vrednostjo. V primeru medpomnilnikov je to vzorec, ki se ponavlja od podanega odmika do želene dolžine. V primeru slik je to barva. A način podajanja te barve ni najbolj enostaven, funkcija namreč sprejme barvo preko kazalca tipa `void`. Če je podatkovni tip kanalov slike normaliziran, mora kazalec kazati na štiri zaporedna števila s plavajočo vejico. Njihov razpon je odvisen od tega, ali je podatkovni tip kanala predznačen ali ni predznačen. Če postavni tip kanalov slike ni normaliziran, mora kazalec kazati na štiri zaporedna števila tipa `int`, katerega vrednost mora biti v razponu podatkovnega tipa kanala slike. In nazadnje, če je podatkovni tip kanala `half` ali `float`, torej eno izmed podatkovnih tipov s plavajočo vejico, standard ne določa razpona vrednosti, le da mora biti ta podana kot število s plavajočo vejico [26, 18]. Ker je lahko to za uporabnika knjižnice nekoliko zapleteno, ta omogoča dva drugačna načina uporabe. Prvi omogoča podajanje vrednosti vseh štirih kanalov kot dvojno natančnih števil s plavajočo vejico, drugi s pomočjo strukture `Color`, ki hrani vrednosti vseh štirih komponent kot bajte. V obeh primerih se vrednosti normalizirajo glede

na podatkovni tip kanalov slike in pretvorijo ter shranijo v pravilen podatkovni tip samodejno. V primeru, da je podatkovni tip kanalov slike `half` ali `float`, so vrednosti pretvorjene vrednosti s plavajočo vejico v razponu med nič in ena, saj standard ne določa, kako morata biti podana vrednosti. Bolj natančno, če so vrednosti kanalov podane kot dvojno natančna števila s plavajočo vejico, je edina sprememba oženje podatkovnega tipa.

Med operacije pomnilniških objektov spadajo tudi preslikave. Preslikavo je mogoče izvesti s funkcijama `clEnqueueMapImage` in `clEnqueueMapBuffer`. Tako kot ostale operacije nad pomnilniškimi objekti je tudi ti dve mogoče izvesti asinhrono. Pri preslikavah je pomembno tudi, da jih je treba končati. Temu je namenjena funkcija `clEnqueueUnmapMemObject`, ki jo je prav tako mogoče izvesti asinhrono. Ker je ustvarjanje preslikave asinhrono bodo metode, ki izvajajo to operacijo, vrnile začasen dogodek. Vrednost tega dogodka bi bila lahko enostavno podatkovnega tipa `IntPtr` ali celo navaden kazalec. A C# ni najbolj prijazen jezik za delo s kazalci, zato je bolj primerno vrniti podatkovno strukturo, ki jo lahko uporabnik knjižnice uporabi kot polje. Razred `NativeMemory` vsebuje veliko logike, ki bi bila uporabna tudi v teh podatkovnih strukturah, zato je vsa koda razreda, ki ni vezana na dodeljevanje in sproščanje pomnilnika ter ustvarjanje instance iz drugih podatkovnih struktur, lahko premaknjena v osnovni razred. Tega je nato mogoče uporabiti kot osnovo za vse nove podatkovne strukture, ki omogočajo preprosto branje in pisanje iz preslikanih medpomnilnikov in različnih vrst slik. Metoda za končanje preslikave se lahko nahaja tako na novih podatkovnih strukturah kot na razredih pomnilniških objektov. Ker je končanje preslikave zelo pomembno, ni slaba ideja, da je na voljo na obeh mestih, saj se s tem zmanjša verjetnost, da bo to pozabil uporabnik knjižnice. K temu se lahko pripomore tudi s samodejnim končanjem preslikave, ko je razred za dostop do preslikanega pomnilnika pobrisan s pomočjo metode `Dispose`.

Ena izmed zadnjih operacij pomnilniških objektov je selitev pomnilnika. Z ukazom `clEnqueueMigrateMemObjects` je mogoče pomnilnik seliti v pomnilnik gostitelja ali na napravo [21]. Selitev h gostitelju kopira vsebino

pomnilniškega objekta v pomnilnik gostitelja in zabeleži, da se pomnilnik nahaja pri gostitelju. To pomeni, da bo ob naslednji uporabi pomnilniškega objekta na eni izmed naprav, njegova vsebina kopirana nazaj na napravo. Selitev na napravo je nekoliko drugačna in ima temu primerno ime zastavice. Njen namen je selitev pomnilniškega objekta na napravo, ki jo definira izvajalna vrsta, s katero se izvaja operacija. A ta selitev ne kopira vsebine želene naprave, temveč le spremeni položaj pomnilniškega objekta. To pomeni, da bo kakršnakoli sprememba, ki ni bila nikoli preslikana na napravo, po tej selitvi izgubljena.

Zadnja operacija pomnilniških objektov je registracija povratnega klica ob uničenju pomnilniškega objekta. Kazalec funkcije, ki je podan za povratni klic, kot pri dogodkih, prejme kot eno izmed vrednosti strukturo pomnilniškega objekta OpenCL [28]. To naredi uporabo te funkcije precej enostavno. A ker je pomembno, da uporabnik tekom povratnega klica ne izvede določenih metod OpenCL in ne zadržuje izvajanja predolgo, je bolje uporabniku izpostaviti drugačen način registracije povratnih klicev. Podobno kot pri kontekstih je možno tega izpostaviti preko dogodkov. A pri tem velja ena razlika. Če uporabnik pomnilniškega objekta ne ustvari s pomočjo implementacije razreda `MemoryManager`, razredi pomnilniških objektov nimajo potrebe po registraciji povratnega klica. V tem primeru mora biti ta narejen naknadno. To je mogoče s pomočjo spreminjanja metode za prijavljanje na dogodek, ki jo navadno C# prevajalnik ustvari samodejno [3]. Pri prijavljanju povratnega klica je nazadnje potrebno poskrbeti le za dve dodatni stvari. In sicer varno prijavljanje na povratni klic v primeru dodajanja poslušalcev iz več niti in izvajanje vseh povratnih klicev na ločeni niti. Povratni klici namreč niso znotraj nadzora knjižnice in bi lahko blokirali eno izmed niti, ki jih uporablja implementacija OpenCL.

Slika 4.7 prikazuje primer ustvarjanja slike s pomočjo knjižnice ImageShapr. Knjižnica za hranjenje slikovnih elementov privzeto uporablja strukturo `Rgba32`. Ovojni razred `Image` te strukture ne pozna, a jo lahko uporabi, če je podan format slike. Če bi knjižnica vrnila preprosto zaporedje bajtov,

bi bilo dovolj podati zaporedje kanalov, saj je to dovolj, da ovojni razred ugotovi podatkovne tipe kanalov na podlagi podatkovnega tipa zaporedja.

```
public static void Main(string[] args)
{
  var device = Device.QueryDevices(DeviceType.Gpu).First();

  using (var queue = CommandQueue.Create(device))
  using (var image = LoadRgba32Image(queue, "image.jpg"))
  {

  }
}

private static Image2D<Rgba32> LoadRgba32Image(
  Context context, string path)
{
  var format = new ImageFormat(
    ImageChannelOrder.Rgba, ImageChannelType.UnsignedInt8);

  using (var image = SixLabors.ImageSharp.Image.Load(path))
  {
    return Image2D.From(context, image.GetPixelSpan(),
      format, image.Width, image.Height);
  }
}
```

Slika 4.7: Primer ustvarjanja slike prebrane z knjižnico `ImageSharp`.

4.6 Programi in ščepci

Vse operacije, ki jih OpenCL izvaja na napravah, morajo biti del programa. Program OpenCL lahko vsebuje izvorno kodo in prevedene programe za različne naprave. OpenCL API omogoča ustvarjanje programov iz izvorne kode in prevedenih programov, ki so ob ustvarjanju programa podani preko kazalca na zaporedje bajtov. Ovojni razred za programe mora zato podpirati oba načina ustvarjanja. Podpirati mora tudi prevajanje in povezovanje programov ter izgradnjo programov, torej je kombinirano operacijo prevajanja in povezovanja. Zlasti prevajanje, saj je to potrebno za uporabo programov ustvarjenjih iz izvorne kode. OpenCL omogoča asinhrono prevajanje in povezovanje programov. A za razliko od večine ostalih funkcij, te ne uporabljajo dogodkov. Namesto dogodkov sprejmejo kazalec na funkcijo, preko katere obvestijo, ko se operacija zaključi. Tudi ovojni razred za dogodke uporablja povratni klic, a to ne pomeni, da se lahko to asinhrono operacijo predstavi s tem razredom, saj ne ovija nobenega dogodka, ki bi se ga lahko podalo drugim metodam oziroma funkcijam, ki jih metode ovijajo. Za omogočanje asinhronih operacij je torej potreben nov razred. Ker vse operacije za povratni klic uporabljajo kazalce na funkcije, ki sprejmejo enake parametre, in je preverjanje stanja operacije vedno izvedeno preko klica iste metode, je dovolj le en razred. V primeru, da operacija uspe, lahko ovojni razred izdela izjemo, ki lahko vsebuje besedilni opis napake. Ta je lažje razumljiv, če razred pozna vrsto operacije, zato je smiselno definirati številni podatkovni tip vrst operacij. Standard določa, da je povratni klic lahko izveden asinhrono, kar pomeni, da to ni garantirano. Standard zahteva le, da je izveden, ko je operacija zaključena [9]. To pomeni, da se klic lahko izvede, preden je instanca ovojnega razreda, ki spremlja operacijo, zaključena. To je rešljivo s preverjanjem stanja operacije po izdelavi instance. A ker je klic lahko izveden asinhrono, torej po koncu operacije, je mogoče, da to preverjanje zazna, da je operacija zaključena, preden se izvede asinhroni povratni klic. V tem primeru ga je treba ignorirati. In nazadnje, to pomeni, da se lahko preverjanje stanja izdelave dogaja sočasno, na več nitih. Težavo je seveda mogoče rešiti

tudi tako, da se za povratni klic uporabi metodo instance, ki je vezana na instanco ovojnega razreda, ki spremlja operacijo. A ta rešitev lahko sesuje program, v primeru, da je povratni klic izveden po izbrisu instance ovojnega razreda [51, 60].

Znotraj izvajalnega okolja OpenCL je ščepec kombinacija kazalca na funkcijo in argumentov [14]. To je pomembno zaradi načina izvajanja ščepcev, ki ga uporablja OpenCL. Ščepcu se vrednosti argumentov nastavi, preden se v izvajalno vrsto doda zahtevo po izvajanju. Zaradi tega je za izvajanje ščepca potrebnih več klicev. Ena izmed slabosti tega pristopa izhaja iz načina nastavljanja argumentov. Vse argumente se nastavi tako, da se implementaciji poda kazalec na vrednost in velikost le-teh vrednosti, kjer je kazalec tipizan kot `void*`. To pomeni, da je precej enostavno podati argument napačnega podatkovnega tipa. Edina obramba pred to napako je možnost poizvedovanja po podatkovnem tipu argumentov [24]. To je seveda mogoče le, če je bil program preveden z opcijo `-cl-kernel-arg-info`. A ker prevajanje programa omogoča knjižnica, je lahko ta opcija dodana samodejno. S pomočjo te informacije je namreč mogoče narediti dve stvari. Preverjanje podatkovnega tipa argumenta in pretvorba vrednosti v pravi podatkovni tip. Preverjanje se sliši precej enostavno in tudi je. V večni primerov je primerjanje obeh podatkovnih tipov dovolj za uspešno preprečevanje uporabe napačnega podatkovnega tipa. A to ne deluje vedno. C# namreč ne pozna vektorjev, kar pomeni, da brez dodajanja novih struktur to ni mogoče. Poleg tega implementacije OpenCL včasih ne povejo točnega podatkovnega tipa. V teh primerih lahko tako preverjanje spodleti in uporabnik knjižnice izgubi možnost nastavljanja parametra. To je mogoče rešiti z dodatnim preverjanjem velikosti obeh podatkovnih tipov. Če je vsaj eden izmed podanih podatkovnih tipov preprost številski tip in drugi eden izmed podatkovnih tipov OpenCL. Pretvorbe pa so nekoliko bolj enostavne. Knjižnica mora poskrbeti le za pravilno pretvorbo preprostih vrednosti. Pri tem je nekaj pretvorb mogoče napisati ročno in s tem ohraniti znanje o podatkovnem tipu rezultata, ostale pa je mogoče izvesti s pomočjo metode `ChangeType`, ki zna pretvarjati vrednosti med različnimi

podatkovnimi tipi, ki implementirajo vmesnik `IConvertible` [31].

Po nastavljanju vrednosti argumentov ostane le izvajanje ščepcev. OpenCL omogoča tri načine izvajanja ščepcev. Prva dva sta povezana z instancami ščepcev, katerih ovojni razred opisuje to poglavje. Zadnji je povezan z izvajanjem funkcij znotraj gostiteljevega programa. Tega načina izvajanja ni preprosto poenostaviti, saj je za vsak tak ščepec treba definirati novo podatkovno strukturo. Če to naredi knjižnica, se to zgodi po prevajanju programa, kar pomeni, da uporabnik knjižnice ne more uporabiti podatkovne strukture med pisanjem programa. Veliko lažje je uporabniku omogočiti uporabi prvih dveh funkcij. Prva omogoča izvajanje ščepca s poljubnimi odmiki, velikostmi in številom izvajalnih skupin. Medtem ko druga omogoča popolnoma isto, le da je število dimenzij vedno enako ena, odmik nič in velikost ena [23]. Vmesnik OpenCL je tu mogoče poenostaviti zgolj z dodatnimi metodami, ki omogočajo zaganjanje ščepcev enodimenzionalno in z le eno delavno skupino, torej tako, da je lokalna velikost enaka globalni. Zlasti prva izmed teh dveh dodatnih metod lahko izkoristi dodane omejitve načina klicanja, da prepreči nepotrebno ustvarjanje polja, saj je namesto polja velikosti mogoče v tem primeru metodi za izvajanje ščepca podati kazalec na vrednost na skladu. In nazadnje vse te metode lahko pomagajo zaščititi uporabnika tako, da preverijo, če so vsa polja velikosti in odmikov prave velikosti, torej natanko eno vrednost za vsako dimenzijo izvajanja ščepca.

Slika 4.8 prikazuje primer poganjanja ščepca ki izračuna vsoto števil. Pri izvajanju tega ščepca je zelo enostavno pozabiti podatkovne tipe argumentov. Ščepec v spodnjem primeru sprejme zaporedje vrednosti tipa `cl_int` in dolžino zaporedja tipa `cl_ulong`. Kljub temu, da je drugi argument podan kot 32-bitna vrednost in ne 64-bitna vrednost, knjižnica s pomočjo informacij o argumentih zazna to napako in ščepcu posreduje 64-bitno vrednost. Če bi ščepcu poskusili za prvi argument podati zaporedje števil s plavajočo vejico, bi to sprožilo izjemo.

```
public static async Task Main(string[] args)
{
  var device = Device.QueryDevices(DeviceType.Gpu).First();
  var input = Enumerable.Range(0, 10000).ToNativeMemory();

  using (var queue = CommandQueue.Create(device))
  using (var program = await BuildProgram(queue, "kernel.cl"))
  using (var kernel = program.CreateKernel("sum"))
  using (var buffer = Buffer.For(queue, input))
  {
    kernel.Arguments[0].Set(buffer);
    kernel.Arguments[1].Set(buffer.Length);

    await kernel.Enqueue(queue, device.MaxComputeUnits);

    using (var mappedMemory = await buffer.Map(
      queue, 0, 1, MapFlags.MapRead))
    {
      Console.WriteLine("Result: {0}", mappedMemory[0]);
    }
  }
}

private static async Task<Program> BuildProgram(
  Context context, string path)
{
  var program = await Program.FromFile(context, path);
  return await program.Build();
}
```

Slika 4.8: Primer poganjanja ščepca.

Poglavje 5

Praktični primer

Za preverjanje uspešnosti izdelave knjižnice je bilo implementirano matrično množenje. Kljub preprosti naravi tega opravila, ima naivna implementacija matričnega množenja časovno zahtevnost $O(n^3)$. To pomeni, da je množenje večjih matrik precej počasno. Napisanih je bilo 8 programov za množenje matrik. In sicer:

1. serijska C++ implementacija,
2. serijska C++ implementacija, ki uporablja AVX,
3. serijska C# implementacija,
4. serijska C# implementacija, ki uporablja AVX,
5. vzporedna C# implementacija,
6. vzporedna C# implementacija, ki uporablja AVX,
7. C++ OpenCL implementacija,
8. C# OpenCL implementacija, ki uporablja knjižnico OCL.NET.

Vsi programi sprejmejo tri argumente. Prvi in drugi argument predstavljata poti do binarnih datotek, vsebujoč velikosti in vrednosti matrik, ki jih bodo programi množili. Tretji argument predstavlja pot, na katero bodo

programi zapisali rezultat množenja. Programi s serijskimi implementacijami množijo matrike s pomočjo treh zank, medtem ko programi z vzporednimi implementacijami množijo matrike s pomočjo ene porazdeljene zanke [80]. C++ program, ki uporablja AVX, to uporablja neposredno s klicem funkcij, ki so definirane v `immintrin.h`, medtem ko C# program za to uporablja eno izmed novosti v novejših različicah izvajalnega okolja .NET [91]. Obe OpenCL implementaciji uporabljata isti ščepec. Le-ta vse tri matrike sprejme kot enokanalne dvodimenzionalne slike. Izvorna koda ščepca je prikazana na sliki 5.1.

Ker je cilj izdelave teh programov primerjanje preprostosti in učinkovitosti različnih implementacij, je zaželeno, da program 8 množi matrike s podobno učinkovitostjo, kot jih množi program 7. Prav tako je zaželeno, da je program 8 učinkovitejši od ostalih C# programov, zlasti pri množenju večjih matrik. Programa 1 in 2 sta C++ različici programov 3 in 4. Napisana sta bila za določanje učinkovitosti izvajalnika .NET. Glavni del programa 8 je več kot dvakrat krajši od glavnega dela programa 7 in zajema le 32 vrstic kode. Izvorna koda glavne metode programa 8 je prikazana na sliki 5.2. Program 8 poleg krajše izvorne kode ponuja tudi dodatno varnost, saj program preveri podatkovne tipe argumentov ščepcev in vrže izjemo, preden se zažene ščepec. Prav tako je poenostavljeno sproščanje virov. Še posebej pa uporaba dogodkov za asinhrono izvajanje operacij, saj za razbijanje glavne metode na začetni del in povratne klice, poskrbi že prevajalnik.

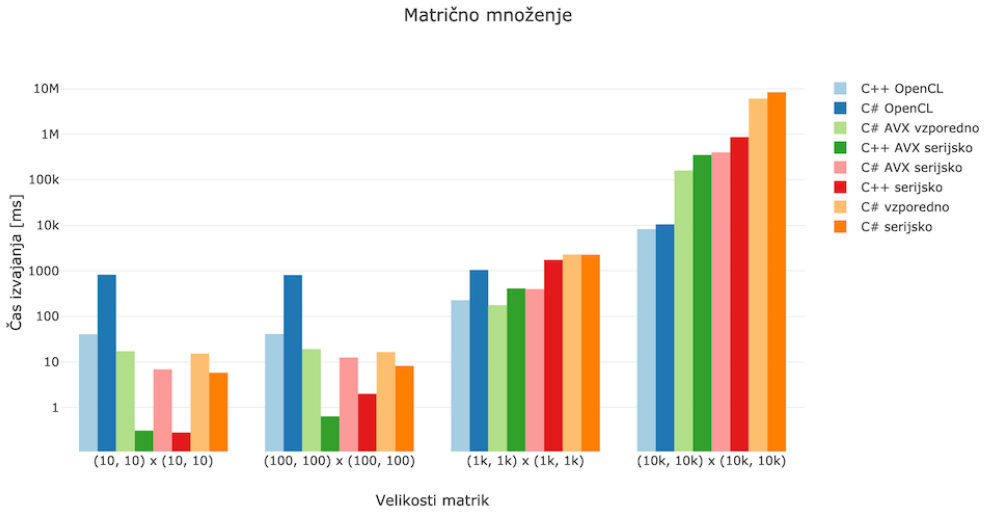
Povprečni časi izvajanja programov z matrikami štirih različnih velikosti so prikazani na sliki 5.3. Iz slike je razvidno, da uporaba knjižnice OCL.NET upočasni zagon programa za dobrih 800 milisekund. Zaradi tega je program precej počasen pri množenju prvih treh parov matrik. Ker na ta čas ne vpliva velikost matrik, je množenje največjega para s tem programom precej bolj učinkovito. Čas izvajanja pri tem paru matrik je namreč primerljiv s časom izvajanja programa 7 in mnogo hitrejše od izvajalnega časa vseh ostalih programov, kar pokaže, da se ohrani pohitritev, ki ga prinese uporaba OpenCL.

```
const sampler_t m_sampler =
    CLK_NORMALIZED_COORDS_FALSE |
    CLK_ADDRESS_NONE |
    CLK_FILTER_NEAREST;

__kernel void matrix_multiply(
    __read_only image2d_t left,
    __read_only image2d_t right,
    __write_only image2d_t result) {
  int x = get_global_id(0);
  int y = get_global_id(1);
  int width = get_image_width(result);
  int height = get_image_height(result);

  if (x < width && y < height) {
    int length = get_image_width(left);
    float4 sum = (float4)(0);

    for (int i = 0; i < length; i++) {
      float4 l = read_imagef(left, m_sampler, (int2)(i, y));
      float4 r = read_imagef(right, m_sampler, (int2)(x, i));
      sum += l * r;
    }

    write_imagef(result, (int2)(x, y), sum);
  }
}
```

Slika 5.1: Ščepec za množenje matrik.

```
var first = Matrix.Read(args[0]);
var second = Matrix.Read(args[1]);
var result = Matrix.Create(second.Width, first.Height);

var device = Device.QueryDevices(DeviceType.Gpu).First();

using (var queue = CommandQueue.Create(device))
using (var program = await BuildProgram(queue, "kernel.cl"))
using (var kernel = program.CreateKernel("matrix_multiply"))
using (var firstImage = CreateInputImage(queue, first))
using (var secondImage = CreateInputImage(queue, second))
using (var resultImage = CreateOutputImage(queue, result))
{
  var wgs = kernel.WorkGroups[device].WorkGroupSize;
  var ls = (ulong) Math.Sqrt(wgs);
  var countX = ((ulong) result.Width - 1) / ls + 1;
  var countY = ((ulong) result.Height - 1) / ls + 1;
  var localSizes = new[] {ls, ls};
  var globalSizes = new[] {ls * countX, ls * countY};

  kernel.Arguments[0].Set(firstImage);
  kernel.Arguments[1].Set(secondImage);
  kernel.Arguments[2].Set(resultImage);

  kernel.Enqueue(queue, globalSizes, localSizes);
  await resultImage.ReadIntoArray(queue, result.Values);
}

result.Write(args[2]);
```

Slika 5.2: Glavna metoda programa 8.



Slika 5.3: Povprečni časi množenja dveh matrik.

Poglavje 6

Zaključek

Dokončana knjižnica omogoča izvajanje vseh funkcij OpenCL in omogoča lažjo uporabo OpenCL s pomočjo objektno usmerjenih ovojnih razredov. A kljub preprosti naravi večine zahtev knjižnice, njena izdelava ni bila popolnoma enostavna.

Jezik C# ponuja veliko izboljšav nad starejšimi jeziki, kot sta C ali C++, in izdelava knjižnice, ki naredi uporabo OpenCL čim bolj domačo uporabnikom, ki so vajeni teh izboljšav in različnih vzorcev, ki se pojavljajo v ogrodju .NET in ostalih .NET knjižnicah, je lahko zapletena. Izredna pogostost podatkovnega tipa `int` pomeni, da bodo uporabniki, ki knjižnico integrirajo v obstoječe programe, pogosto delali prav s tem podatkovnim tipom. Če je od uporabnika zahtevano pretvarjanje vrednosti v drugi podatkovni tip, vsakič ko uporablja knjižnico, to lahko ustvari občutek nepripadnosti. Dodano težavnost predstavljajo izboljšave, kot je asinhron model programiranja. OpenCL omogoča asinhrono delo s knjižnico, ki je v C# lahko precej poenostavljeno. A to zahteva dodatne klice funkcij OpenCL. Izbira katerih funkcij in kdaj se izvajajo, mora biti čim boljša, saj vsak dodaten klic upočasni delovanje.

Najbolj zahteven del izdelave knjižnice je bilo testiranje. OpenCL ima mnogo implementacij, vsaka je narejena drugače in ima svoje težave. Nvidia ne omogoča poizvedovanja po napravah brez določanja okolja ter ima

težave s prevajanjem ščepcev, ki beležijo. Intel na MacOS Mojave nepravilno pripravi interne podatkovne strukture, kar povzroči sesutje programa, če ta poskuša prebrati izvorno kodo povezanega programa ali binarno vsebino oziroma imena ščepcev programa, ki ni preveden. Prav tako različne implementacije izvajajo operacije, ki niso definirane zelo natančno, na različne načine. Zaradi tega je bilo potrebno knjižnico testirati na različnih napravah, z različnimi operacijskimi sistemi. A tudi testiranje s tremi različnimi napravami ne zagotavlja pravilnega delovanja na vseh napravah. Zaradi tega bi knjižnici izredno pripomogli testi, s katerimi bi bilo lahko testiranje knjižnice v veliki meri avtomatizirano.

Knjižnico bi bilo mogoče izboljšati tudi s podporo podatkovnega tipa `half` in vektorskih podatkovnih tipov, za katere ogrodje .NET nima ustreznikov, kar pomeni, da jih mora uporabnik knjižnice definirati sam. Ker je namen knjižnice obdelava podatkov, so seveda smiselne tudi optimizacije, saj se bo veliko uporabnikov OpenCL, zanjo odločilo ravno zaradi hitrosti. In nazadnje je vedno mogoče izboljšati vmesnik knjižnice. Trenutni vmesnik ima nekaj dodatnih metod, ki omogočajo lažje izvajanje bolj pogostih operacij. In verjetno je smiselno dodati še kakšno. A to hkrati poveča število vseh metod, in teh je veliko. Zato se knjižnico nedvomno lahko poenostavi z manjšanjem števila metod. Zlasti z odstranjevanjem katere izmed manj smiselnih dodatnih metod. Dober način manjšanja površine vmesnika bi bil omejevanje števila podatkovnih tipov, ki jih sprejme knjižnica. A take spremembe pokvarijo združljivost knjižnice s starejšimi različicami. Zato jih je dobro uveljavljati le ob večjih spremembah različice knjižnice.

Literatura

[1] About Mono. Dosegljivo: https://www.mono-project.com/docs/about-mono/. [Dostopano 9. 7. 2018].

[2] Action delegate. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.action?view=netstandard-1.1. [Dostopano 10. 7. 2018].

[3] add (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/add. [Dostopano 24. 8. 2018].

[4] AdvanceDLSupport. Dosegljivo: https://github.com/Firwood-Software/AdvanceDLSupport. [Dostopano 11. 7. 2018].

[5] Async task types in C#. Dosegljivo: https://github.com/dotnet/roslyn/blob/master/docs/features/task-types.md. [Dostopano 14. 7. 2018].

[6] Asynchronous programming with async and await (C#). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/index. [Dostopano 14. 7. 2018].

[7] BitmapData.Stride property. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.drawing.imaging.bitmapdata.stride?view=netframework-4.7.2. [Dostopano 24. 8. 2018].

[8] Blittable and non-blittable types. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types. [Dostopano 12. 7. 2018].

[9] clBuildProgram. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clBuildProgram.html. [Dostopano 27. 10. 2018].

[10] clCreateBuffer. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateBuffer.html. [Dostopano 24. 8. 2018].

[11] clCreateCommandQueue. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateCommandQueue.html. [Dostopano 13. 7. 2018].

[12] clCreateContext. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateContext.html. [Dostopano 10. 7. 2018].

[13] clCreateContextFromType. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateContextFromType.html. [Dostopano 10. 7. 2018].

[14] clCreateKernel. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateKernel.html. [Dostopano 28. 10. 2018].

[15] clCreateSubDevices. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateSubDevices.html. [Dostopano 10. 7. 2018].

[16] clEnqueueBarrierWithWaitList. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueBarrierWithWaitList.html. [Dostopano 14. 7. 2018].

[17] clEnqueueCopyImage. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueCopyImage.html. [Dostopano 24. 8. 2018].

[18] clEnqueueFillImage. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueFillImage.html. [Dostopano 24. 8. 2018].

[19] clEnqueueMapBuffer. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueMapBuffer.html. [Dostopano 24. 8. 2018].

[20] clEnqueueMarkerWithWaitList. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueMarkerWithWaitList.html. [Dostopano 14. 7. 2018].

[21] clEnqueueMigrateMemObjects. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueMigrateMemObjects.html. [Dostopano 24. 8. 2018].

[22] clEnqueueReadBufferRect. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueReadBufferRect.html. [Dostopano 24. 8. 2018].

[23] clEnqueueTask. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clEnqueueTask.html. [Dostopano 28. 10. 2018].

[24] clGetKernelArgInfo. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clGetKernelArgInfo.html. [Dostopano 28. 10. 2018].

[25] cl_image_desc. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/cl_image_desc.html. [Dostopano 10. 7. 2018].

[26] cl_image_format. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/cl_image_format.html. [Dostopano 16. 9. 2018].

[27] clReleaseDevice. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clReleaseDevice.html. [Dostopano 12. 7. 2018].

[28] clSetMemObjectDestructorCallback. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clSetMemObjectDestructorCallback.html. [Dostopano 24. 8. 2018].

[29] clSetUserEventStatus. Dosegljivo: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clSetUserEventStatus.html. [Dostopano 14. 7. 2018].

[30] ConditionalWeakTable<TKey,TValue> class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.conditionalweaktable-2?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[31] Convert.ChangeType method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.convert.changetype?view=netcore-2.1. [Dostopano 28. 10. 2018].

[32] Copying and pinning. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/framework/interop/copying-and-pinning. [Dostopano 23. 8. 2018].

[33] Delegates. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/. [Dostopano 10. 7. 2018].

[34] DllImportAttribute class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.dllimportattribute?view=netstandard-1.1. [Dostopano 9. 7. 2018].

[35] Double-checked locking. Dosegljivo: https://en.wikipedia.org/wiki/Double-checked_locking. [Dostopano 11. 7. 2018].

[36] Dynamic loading. Dosegljivo: https://en.wikipedia.org/wiki/Dynamic_loading. [Dostopano 11. 7. 2018].

[37] enum (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/enum. [Dostopano 10. 7. 2018].

[38] Enumerable class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=netstandard-2.0. [Dostopano 24. 8. 2018].

[39] Ephemeron. Dosegljivo: https://en.wikipedia.org/wiki/Ephemeron. [Dostopano 12. 7. 2018].

[40] Event-driven programming. Dosegljivo: https://en.wikipedia.org/wiki/Event-driven_programming. [Dostopano 14. 7. 2018].

[41] explicit (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/explicit. [Dostopano 10. 7. 2018].

[42] Extending the async methods in C#. Dosegljivo: https://blogs.msdn.microsoft.com/seteplia/2018/01/11/extending-the-async-methods-in-c/, 2018. [Dostopano 14. 7. 2018].

[43] Extension methods (C# programming guide). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods. [Dostopano 11. 7. 2018].

[44] fixed statement (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/fixed-statement. [Dostopano 12. 7. 2018].

[45] FlagsAttribute class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions. [Dostopano 10. 7. 2018].

[46] Func<TResult> delegate. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.func-1?view=netstandard-1.1. [Dostopano 10. 7. 2018].

[47] Fundamentals of garbage collection. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals. [Dostopano 12. 7. 2018].

[48] GCHandle.Alloc method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.gchandle.alloc?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[49] General naming conventions. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions. [Dostopano 10. 7. 2018].

[50] Getting started with LINQ in C#. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/getting-started-with-linq. [Dostopano 24. 8. 2018].

[51] How to: Marshal callbacks and delegates by using C++ interop. Dosegljivo: https://docs.microsoft.com/en-us/cpp/dotnet/how-to-marshal-callbacks-and-delegates-by-using-cpp-interop?view=vs-2017. [Dostopano 27. 10. 2018].

[52] IDisposable interface. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[53] implicit (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/implicit. [Dostopano 10. 7. 2018].

[54] Instance constructors (C# programming guide). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors. [Dostopano 13. 7. 2018].

[55] Interfaces. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/. [Dostopano 10. 7. 2018].

[56] Interpreter. Dosegljivo: https://en.wikipedia.org/wiki/Interpreter_(computing). [Dostopano 9. 7. 2018].

[57] IntPtr. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.intptr?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[58] MarshalAsAttribute class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshalasattribute?view=netstandard-2.0. [Dostopano 11. 7. 2018].

[59] Marshal.GetDelegateForFunctionPointer method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.getdelegateforfunctionpointer?view=netstandard-1.1. [Dostopano 11. 7. 2018].

[60] Marshal.GetFunctionPointerForDelegate method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.getfunctionpointerfordelegate?view=netcore-2.1. [Dostopano 27. 10. 2018].

[61] Marshaling a delegate as a callback method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/framework/interop/

marshaling-a-delegate-as-a-callback-method. [Dostopano 10. 7. 2018].

[62] Marshaling classes, structures, and unions. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/framework/interop/marshaling-classes-structures-and-unions. [Dostopano 10. 7. 2018].

[63] Marshalling. Dosegljivo: https://en.wikipedia.org/wiki/Marshalling_(computer_science). [Dostopano 10. 7. 2018].

[64] Marshal.PtrToStructure method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.ptrtostructure?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[65] MemoryManager<T> class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.buffers.memorymanager-1?view=netcore-2.1. [Dostopano 24. 8. 2018].

[66] Memory<T> struct. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.memory-1?view=netcore-2.1. [Dostopano 12. 7. 2018].

[67] Names of namespaces. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces. [Dostopano 10. 7. 2018].

[68] Native interoperability. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/native-interop. [Dostopano 9. 7. 2018].

[69] .NET Core guide. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/core/. [Dostopano 9. 7. 2018].

[70] .NET Framework guide. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/framework/. [Dostopano 9. 7. 2018].

[71] .NET programming languages. Dosegljivo: https://www.microsoft.com/net/learn/languages. [Dostopano 9. 7. 2018].

[72] .NET programming with C++/CLI (Visual C++). Dosegljivo: https://docs.microsoft.com/en-us/cpp/dotnet/dotnet-programming-with-cpp-cli-visual-cpp. [Dostopano 10. 7. 2018].

[73] .NET Standard. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/net-standard. [Dostopano 9. 7. 2018].

[74] OpCodes class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes?view=netstandard-2.0. [Dostopano 24. 8. 2018].

[75] OpCodes.Calli field. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.calli?view=netstandard-1.1. [Dostopano 11. 7. 2018].

[76] OpCodes.Callvirt field. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.callvirt?view=netstandard-2.0. [Dostopano 24. 8. 2018].

[77] OpenCL ICD extension. Dosegljivo: https://www.khronos.org/registry/OpenCL/extensions/khr/cl_khr_icd.txt, 2010. [Dostopano 9. 7. 2018].

[78] out parameter modifier (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier. [Dostopano 11. 7. 2018].

[79] OutOfMemoryException and pinning. Dosegljivo: https://blogs.msdn.microsoft.com/yunjin/2004/01/27/outofmemoryexception-and-pinning/, 2004. [Dostopano 23. 8. 2018].

[80] Parallel.For method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for?view=netcore-2.1. [Dostopano 12. 2. 2019].

[81] Properties. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties. [Dostopano 8. 9. 2018].

[82] ref (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref. [Dostopano 11. 7. 2018].

[83] Scheduling. Dosegljivo: https://en.wikipedia.org/wiki/Scheduling_(computing). [Dostopano 9. 7. 2018].

[84] Span<T> struct. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.span-1?view=netcore-2.1. [Dostopano 12. 7. 2018].

[85] StructLayoutAttribute class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.structlayoutattribute?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[86] TaskCompletionSource<TResult> class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1?view=netstandard-2.0. [Dostopano 14. 7. 2018].

[87] The nature of TaskCompletionSource<TResult>. Dosegljivo: https://blogs.msdn.microsoft.com/pfxteam/2009/06/02/the-nature-of-taskcompletionsourcetresult/, 2009. [Dostopano 14. 7. 2018].

[88] The OpenCL specification. Dosegljivo: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, 2012. [Dostopano 9. 7. 2018].

[89] UIntPtr. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.uintptr?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[90] unsafe (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unsafe. [Dostopano 24. 8. 2018].

[91] Using .NET hardware intrinsics API to accelerate machine learning scenarios. Dosegljivo: https://blogs.msdn.microsoft.com/dotnet/2018/10/10/using-net-hardware-intrinsics-api-to-accelerate-machine-learning-scenarios/. [Dostopano 12. 2. 2019].

[92] UTF-8, a transformation format of ISO 10646. Dosegljivo: http://www.ietf.org/rfc/rfc3629.txt, 2003. [Dostopano 12. 7. 2018].

[93] UTF8Encoding.GetString(Byte[], Int32, Int32) method. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.text.utf8encoding.getstring?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[94] WeakReference<T> class. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/api/system.weakreference-1?view=netstandard-2.0. [Dostopano 12. 7. 2018].

[95] What is "managed code"? Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/managed-code. [Dostopano 10. 7. 2018].

[96] What's new in C# 7.0. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7. [Dostopano 14. 7. 2018].

[97] What's new in C# 7.3. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7-3. [Dostopano 14. 7. 2018].

[98] where (generic type constraint) (C# reference). Dosegljivo: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint. [Dostopano 12. 7. 2018].