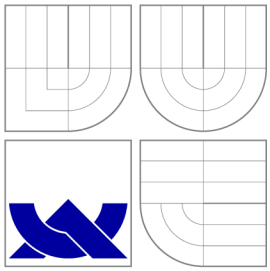
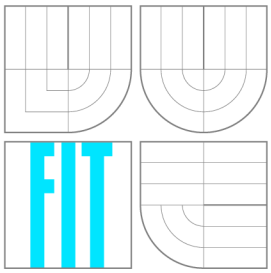


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD OPENCL APLIKACÍ PRO VESTAVĚNÉ SYSTÉMY

COMPILATION OF OPENCL APPLICATIONS FOR EMBEDDED SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL ŠNOBL

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2016

Zadání diplomové práce/18437/2015/xsnobl00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Šnobl Pavel, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Překlad OpenCL aplikací pro vestavěné systémy**
Compilation of OpenCL Applications for Embedded Systems
Kategorie: Překladače

Pokyny:

1. Seznamte se s kompilační platformou LLVM a programovacím jazykem OpenCL.
2. Seznamte se s dostupnými runtime knihovnami pro OpenCL. Nastudujte problematiku optimalizace OpenCL aplikací (kernelů), seznamte se s existujícími rozšířeními pro překladač LLVM umožňující překlad OpenCL kernelů.
3. Vyberte vhodnou runtime knihovnu pro OpenCL a rozšíření pro překlad OpenCL kernelů integrujte do překladače LLVM a otestujte.
4. Zaměřte se na optimalizace OpenCL kernelů pro VLIW architektury a architektury se SIMD instrukcemi. Vyberte vhodné optimalizace a integrujte je do překladače LLVM.
5. Na alespoň deseti aplikacích otestujte vytvořený překladač a porovnejte výkon s a bez přidaných optimalizací pro architektury se SIMD instrukcemi.
6. Zhodnoťte dosažené výsledky a navrhněte další rozšíření, která by mohla vést k vyššímu výkonu překládaných aplikací.

Literatura:

- R. Tsuchiyama, et al.: The OpenCL Programming Book, Fixstars Corporation; 1st edition (April 13, 2010).
- R. Banger, K. Bhattacharyya: OpenCL Programming by Example, Packt Publishing (December 23, 2013).

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

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

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

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

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Husár Adam, Ing., VCIT FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá podporou pro překlad a spouštění programů napsaných pomocí OpenCL frameworku na vestavěných systémech. OpenCL je systém pro programování heterogenních systémů, složených z procesorů, grafických akcelerátorů a dalších výpočetních zařízení. Využití má ovšem i na systémech skládajících se pouze z jedné výpočetní jednotky, kde umožňuje zápis paralelních programů (funkční a datový paralelismus) a práci s hierarchickým systémem pamětí. V rámci této práce jsou porovnány jednotlivé dostupné open source implementace OpenCL a následně je jedna vybraná integrována s překladačem LLVM. Tento překladač je generován v rámci sady nástrojů poskytovaných vývojovým prostředím pro tvorbu procesorů s aplikačně specifickou instrukční sadou zvaným Codasip Studio. Dále jsou navrženy a implementovány optimalizace pro architektury se SIMD instrukcemi a architektury typu VLIW. Výsledek je otestován a demonstrován na sadě testovacích aplikací.

Abstract

This master's thesis deals with the support for compilation and execution of programs written using OpenCL framework on embedded systems. OpenCL is a system for programming heterogeneous systems comprising processors, graphic accelerators and other computing devices. But it also finds usage on systems composed of just one computing unit, where it allows to write parallel programs (task and data parallelism) and work with hierarchical system of memories. In this thesis, various available open source OpenCL implementations are compared and one selected is then integrated into LLVM compiler infrastructure. This compiler is generated as a part of toolchain provided by application specific instruction set architecture processor developement environment called Codasip Studio. Designed and implemented are also optimizations for architectures with SIMD instructions and VLIW architectures. The result is tested and demonstrated on a set of testing applications.

Klíčová slova

OpenCL, LLVM, Codasip, SIMD, VLIW

Keywords

OpenCL, LLVM, Codasip, SIMD, VLIW

Citace

ŠNOBL, Pavel. *Překlad OpenCL aplikací pro vestavěné systémy.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Hruška Tomáš.

Překlad OpenCL aplikací pro vestavěné systémy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytl pan Ing. Adam Husár, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Šnobl
24. května 2016

Poděkování

Tímto bych chtěl poděkovat panu prof. Ing. Tomáši Hruškovi, CSc. za vedení této práce a panu Ing. Adamu Husárovi, PhD. za odbornou pomoc a cenné rady při jejím řešení.

Obsah

Kapitola 1

Úvod

Většina dnešních počítačů obsahuje kromě procesoru i další specializované výpočetní jednotky, mezi které nejčastěji patří grafická karta. Klasicky ovšem tyto jednotky nemůžeme přímo programovat a tak je využít pro obecné výpočty. Řešení nabízí speciální frameworky (softwarová prostředí nabízející určitou zabudovanou funkcionalitu) pro programování heterogenních výpočetních systémů jako je CUDA [25] od firmy Nvidia nebo OpenCL (Open Computing Language) [15] od Khronos Group [16], o kterém pojednává tato práce.

OpenCL díky své univerzálnosti umožňuje i programování klasických homogenních systémů, kde všechny výpočty probíhají na jediné výpočetní jednotce. V takovém případě je možné využít OpenCL pro explicitní vyjádření paralelismu např. na úrovni vláken nebo na úrovni dat (SIMD výpočty). Tím pádem se využití OpenCL stává zajímavým i z hlediska použití na vestavěných systémech, kde díky dnešní technologii jsou již tyto typy paralelismu dostupné a často přispívají k nemalému zvýšení výkonu a tím i odezvy systému, což jsou v případě vestavěných systémů často kritické vlastnosti. OpenCL rovněž umožňuje jednoduše pracovat s více různými paměťmi v rámci paměťové hierarchie zařízení.

Kapitola 2 slouží pro seznámení se standardem OpenCL. Kapitola 3 pojednává o kompilačním frameworku LLVM a o jeho schopnostech překladu OpenCL programů. Kapitola 4 seznamuje čtenáře s Codasip Studiem, které slouží pro návrh procesorů s aplikačně specifickou instrukční sadou, které jsou mimo jiné vhodné pro použití ve vestavěných systémech. Kapitola 5 se věnuje přehledu dostupných implementací OpenCL. Jsou zmíněny jak proprietární implementace, tak ty volně dostupné. Kapitola 6 popisuje více do hloubky `pocl` (Portable Computing Language), což je právě jedna z nejlepších volně dostupných implementací OpenCL. Kapitola 7 se se zaměřuje na popis procesu integrace `pocl` do překladače LLVM. Kapitoly 8 a 9 popisují návrh a implementaci optimalizací pro architektury se SIMD instrukcemi a architektury typu VLIW. Kapitola 10 se zabývá testováním vytvořeného překladače a porovnáním výkonu s a bez přidaných optimalizací. Poslední kapitolou je závěr, kde je uvedeno shrnutí této práce.

Tato diplomová práce navazuje na semestrální projekt, z něhož využívá kapitoly 1 až 5 a částečně kapitolu 6. Tyto kapitoly slouží hlavně pro popis teoretické části řešené problematiky.

Kapitola 2

OpenCL framework

OpenCL je framework pro vytváření programů spouštěných na heterogenních platformách složených z procesorů (CPU), grafických karet (GPU), digitálních signálových procesorů (DSP) a dalších procesorů. OpenCL definuje jazyk pro programování těchto zařízení a aplikační programová rozhraní (API) pro řízení platformy a spouštění programů na výpočetních zařízeních. OpenCL umožňuje paralelní výpočty pomocí datového a funkčního paralelismu. Jedná se o otevřený standard spravovaný neziskovým technologickým konsorciem Khronos Group.

Pomocí OpenCL může programátor napsat program, který následně poběží na široké škále systémů, od mobilních telefonů k superpočítačovým uzlům.

OpenCL poskytuje vysokou úroveň přenositelnosti i přes svoji relativní nízkoúrovňovost. To znamená, že programátor v OpenCL musí explicitně definovat platformu, její kontext a jak je práce rozvržena na různá zařízení [24].

Dobrá přenositelnost je dána abstraktním paměťovým a výpočetním modelem, problémem ovšem může být výkon takto přenesených aplikaci. Často je nutné aplikaci pro konkrétní platformu ručně odladit.

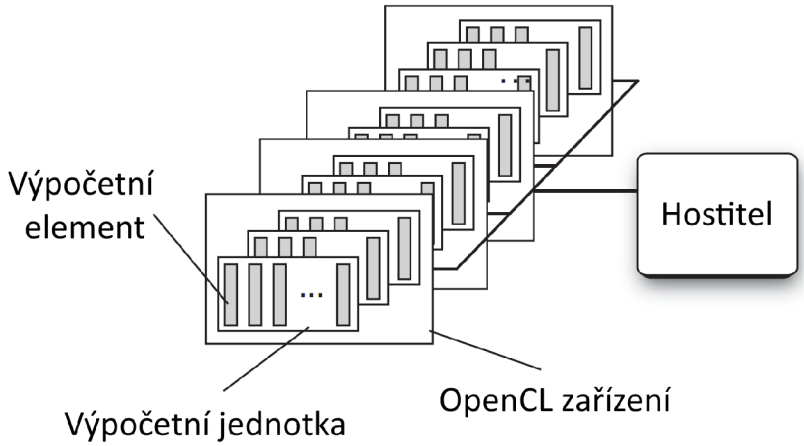
Aktuálním standardem OpenCL je verze 2.1, která je zpětně kompatibilní se staršími ale stále používanými standardy 1.1, 1.2 a 2.0. Rozsáhlou specifikaci těchto standardů ve formě PDF dokumentů lze nalézt na [14].

2.1 Hardwarový model

Hardwarový model OpenCL specifikuje vysokoúrovňovou reprezentaci heterogenní platformy použité pro OpenCL. Tento model je ukázán na obrázku 2.1. OpenCL platforma vždy obsahuje jediného hostitele (hlavní procesor), který interaguje s prostředím, tedy zajišťuje vstup a výstup dat a komunikuje s uživatelem programu [24].

Hostitel je připojen k jednomu nebo více OpenCL zařízením. Zařízení slouží k provádění samotného jádra výpočtu, proto se jim v OpenCL říká výpočetní zařízení. Tímto zařízením může být CPU, GPU, DSP nebo jakýkoli jiný procesor podporující OpenCL.

OpenCL zařízení jsou dále dělena na výpočetní jednotky (*compute units*), které jsou děleny na jeden nebo více výpočetních elementů (*processing elements*). Výpočet na zařízení provádí právě výpočetní elementy.



Obrázek 2.1: HW model OpenCL s jedním hostitelem a jedním nebo více výpočetními zařízeními [24]

2.2 Výpočetní model

OpenCL aplikace se skládá ze dvou částí: hostitelského programu a jednoho nebo více kernelů. Hostitelský program běží na hostitelském CPU. OpenCL nespecifikuje detaily toho, jak hostitelský program funguje, pouze jak interaguje s objekty definovanými v OpenCL.

Kernely běží na OpenCL zařízeních. Vykonávají hlavní část práce typické OpenCL aplikace. Kernely jsou většinou relativně jednoduché funkce, které transformují vstupní paměťové objekty na výstupní paměťové objekty. OpenCL definuje 2 typy kernelů:

- **OpenCL kernely**: funkce napsané v programovacím jazyku OpenCL C a kompilované OpenCL překladačem. Všechny implementace OpenCL musí podporovat OpenCL kernely.
- **Nativní kernely**: funkce vytvořené mimo OpenCL a používané z OpenCL pomocí ukazatelů na funkce. Tyto funkce mohou být např. definované v hostitelském zdrojovém kódu nebo exportované ze specializované knihovny. Schopnost spouštět nativní kernely je volitelnou součástí OpenCL standardu a sémantika nativních kernelů je závislá na konkrétní implementaci.

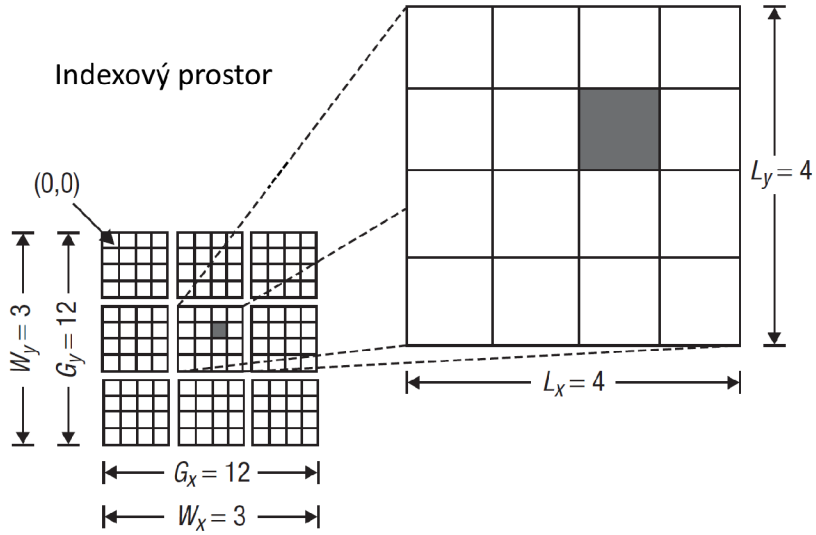
Kernely jsou definovány na hostitelské straně OpenCL aplikace. Hostitelský program určí, které kernely jsou určeny pro spuštění na OpenCL zařízení. OpenCL runtime poté vytvoří celočíselný indexový prostor a pro každý bod v tomto prostoru je spuštěna jedna instance kernelu. Instance spuštěného kernelu se nazývá work-item (česky *pracovní položka*, ovšem nepoužívá se) a je identifikována svými souřadnicemi v indexovém prostoru. Tyto souřadnice jsou globálním identifikátorem work-itemu.

Příkaz určující, které kernely se mají provést vytváří kolekci work-items, kde každý z nich provádí stejnou sekvenci instrukcí definovaných kernelem. I když je sekvence instrukcí všech work-items stejná, chování jednotlivých work-items se může lišit díky příkazům větvění uvnitř kódu nebo hodnotám vybraným na základě identifikátoru dané instance.

Work-items jsou organizovány do tzv. work-groups (česky *pracovní skupiny*, ale opět se nepoužívá). Work-groups umožňují dekompozici indexového prostoru na vyšší úrovni abstrakce. Všechny work-groups mají stejnou velikost v odpovídajících dimenzích a ta rovnoměrně rozděluje indexový prostor v každé dimenzi (viz obrázek 2.2). Work-groups mají

přiřazen unikátní identifikátor v rámci indexového prostoru a jednotlivé work-items unikátní lokální identifikátor v rámci své work-group. Každý work-item tak může být identifikován svým globálním identifikátorem nebo kombinací lokálního identifikátoru a identifikátoru své work-group.

Všechny work-items v rámci jedné work-group běží současně na výpočetních elementech jedné výpočetní jednotky. Konkrétní implementace může serializovat provádění kernelů a work-groups, OpenCL však zajišťuje, že všechny work-items v rámci work-group běží paralelně a sdílejí výpočetní prostředky zařízení [24]. Velikost work-groups specifikuje uživatel v rámci hostitelské aplikace, konkrétní implementace však může maximální velikost limitovat (třeba i na 1).



Obrázek 2.2: Rozdělení indexového prostoru na work-groups (W_x, W_y), každá work-group obsahuje $L_x * L_y$ work-items, celkový počet work-items je tak $G_x * G_y$ [24]

2.2.1 Paralelní provádění výpočtů

Současné vykonávání work-items v rámci work-group má za následek, že se vždy určitá část výpočtu kernelu provádí současně nad více různými daty. Existují 3 základní způsoby, jak může být tento paralelismus na úrovni hardwaru realizován:

- **Datový paralelismus**: SIMD (Single Instruction Multiple Data), případně SPMD (Single Program Multiple Data) výpočetní model. Všechny work-groups a work-items jsou vykonávány v rámci jednoho procesu a jednoho vlákna, jeden work-item odpovídá jednomu prvku vektoru. Počet paralelně vykonávatelných work-items (velikost jedné work-group) závisí na šířce vektorové jednotky a na datových typech použitých pro výpočty uvnitř kernelu.

- **Funkční paralelismus**: Jeden work-item odpovídá jednomu procesu nebo častěji vláknu běžícímu na výpočetním zařízení. Počet work-items v rámci work-group se může měnit v poměrně širokém rozmezí, existuje ale maximální množství vláken, která lze v hardwaru skutečně současně spustit.

- **Kombinovaný paralelismus**: Kombinace datového a funkčního paralelismu, tedy

více současně běžících vláken, každé vykonávající SIMD instrukce. Tento typ paralelismu, známý jako SIMT (Single Instruction Multiple Threads) je podobný jako v technologii CUDA určené pro grafické akcelerátory od firmy Nvidia.

2.3 Paměťový model

OpenCL definuje 4-úrovňovou paměťovou hierarchii pro výpočetní zařízení:

- **globální paměť**: je sdílená všemi výpočetními elementy, má ovšem obecně dlouhou přístupovou dobu. Tato paměť umožňuje čtení a zápis všemi work-items ve všech work-groups, čtení a zápisy mohou být cachovány v závislosti na schopnostech zařízení.
- **konstantní paměť**: menší, nižší latence, hostitelské CPU do ní může zapisovat, work-items pouze číst. Konstatní paměť je mapována do globální paměti, je tedy sdílena všemi work-items.
- **lokální paměť**: sdílená v rámci work-group, tato paměť může být použita pro proměnné sdílené všemi work-items v rámci jedné work-group.
- **privátní paměť**: viditelná jen v rámci jednoho work-item, většinou realizovaná pomocí registrů. Proměnné definované v privátní paměti nejsou viditelné z ostatních work-items a to ani v rámci stejné work-group.

Ne všechna zařízení musí implementovat každou úroveň této hierarchie rovněž v hardwaru, všechny úrovně mohou být dokonce mapovány do jediné společné paměti.

Zařízení mohou nebo nemusí sdílet paměť s hostitelským CPU. Aplikační programové rozhraní poskytuje prostředky pro přesuny dat mezi CPU a jednotlivými zařízeními.

2.4 Aplikační programové rozhraní

OpenCL poskytuje uživateli aplikační programové rozhraní (*Application Programming Interface*, API), které se skládá z následujících částí [24]:

- **OpenCL platform API**: definuje funkce používané hostitelským programem pro zjištění informací o dostupných OpenCL zařízeních a jejich schopnostech a rovněž funkce pro vytváření kontextu OpenCL aplikací. Hostitelská aplikace pomocí těchto funkcí zvolí jedno nebo více zařízení, na kterých budou probíhat výpočty a pomocí dotazů na jejich vlastnosti jim může tyto výpočty přizpůsobit na míru.
- **OpenCL runtime API**: pracuje s vytvořeným kontextem a vytváří fronty příkazů, které se mají provést na OpenCL zařízeních. Každému zařízení může být přiřazena jen jedna fronta, ovšem v rámci jednoho kontextu může současně existovat front více (jedna pro každé zařízení). Runtime API dále slouží pro vytváření paměťových objektů (bufferů), které následně mohou být nastaveny jako argumenty kernelů, pro překlad a spouštění kernelů, převzetí jejich výsledků a mnohé další. Většina funkcí OpenCL API je součástí právě runtime API.
- **Jazyk OpenCL C**: jazyk pro zápis OpenCL kernelů, popsaný v sekci 2.5.

Pro použití OpenCL je nutné v hostitelské aplikaci vložit hlavičkový soubor `CL/cl.h` (na OS firmy Apple `OpenCL/opencl.h`), dodávaný spolu s konkrétní implementací OpenCL a který obsahuje deklarace všech funkcí, konstant a datových typů tvořících OpenCL API. Samotné implementace funkcí se nachází v knihovně (nejčastěji dynamické, ale může být i statická), která musí být k aplikaci přilinkována parametrem `-lOpenCL`.

Kernely mohou být buďto zapsány přímo v hostitelském programu jako řetězec, což se většinou používá u krátkých a jednoduchých kernelů, nebo mohou být načteny z externího souboru. Ten může být buď klasicky textový anebo binární, obsahující částečně přeložený kód (např. LLVM IR bitkód). Načítání z externího souboru se většinou používá v případě složitých kernelů, kernelů potřebujících ke své činnosti pomocné funkce a podobně.

2.5 Jazyk OpenCL C

Programovací jazyk použitý pro psaní výpočetních kernelů je nazýván OpenCL C a je založen na C99 [10], ale je přizpůsoben pro potřeby programovacího modelu OpenCL. Paměťové buffery jsou uloženy v různých úrovních paměťové hierarchie a ukazatele jsou označeny kvantifikátorem regionu pomocí klíčových slov `__global`, `__constant`, `__local` a `__private`.

Příklad 2.5.1. Kernel implementující algoritmus pro násobení matice a vektoru v OpenCL C.

```
// Vynásobí A*x, výsledek uloží do y.
// Matice A je v paměti uložena po řádcích, prvek (i, j) je na
   pozici A[i*ncols+j].
__kernel void matvec(__global const float *A, __global const float
   *x, uint ncols, __global float *y)
{
    size_t i = get_global_id(0); // globální id, index řádku
    __global float const *a = &A[i*ncols]; // ukazatel na řádek
    float sum = 0.f; // akumulátor pro skalární součin
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

Místo toho, aby program určený pro výpočetní zařízení měl jednu hlavní funkci, označenou jako `main`, jsou OpenCL funkce označeny klíčovým slovem `__kernel`. To říká, že jsou vstupními body do programu spouštěného z hostitelského programu.

Ukazatele na funkce, bitová pole a pole s proměnnou délkou nejsou podporovány, rekurze je zakázána. Standardní knihovna jazyka C je nahrazena vlastní sadou standardních funkcí (zabudovaných, builtin), implementujících většinou různé datové operace [24].

Na příkladu 2.5.1 je ukázka jednoduchého kernelu pro násobení matice a vektoru. Kernel `matvec` spočítá při každém spuštění skalární součin jednoho řádku matice `A` a vektoru `x`. Pro rozšíření na plné násobení matice a vektoru spustí OpenCL runtime kernel pro každý řádek matice. Na hostitelské straně to provede funkce `clEnqueueNDRangeKernel`: jako argument dostane kernel, který se má vykonat, jeho argumenty a počet work-items odpovídající počtu řádků v matici `A`. Ukázka takového programu je na příkladu 2.5.2.

Programy napsané v OpenCL C jsou určeny k překládání za běhu hostitelského programu kvůli zajištění přenositelnosti OpenCL aplikací mezi různými výpočetními zařízení.

Příklad 2.5.2. Kód hostitelské aplikace ke kernelu z příkladu 2.5.1.

```
// vytvoření výpočetního kontextu pro GPU zařízení
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL,
    NULL, NULL);

// vytvoření fronty příkazů
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,NULL);
queue = clCreateCommandQueue(context, device_id, 0, NULL);

// vytvoření bufferů (matice, vstupního a výstupního vektoru)
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(float)*num_entries*num_entries,
    srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(float)*num_entries, srcx, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(float)*num_entries, NULL, NULL);

// vytvoření programu, jeho překlad a vytvoření kernelu
program = clCreateProgramWithSource(context,1,&matvec_kernel_src,
    NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "matvec", NULL);

// nastavení hodnot argumentů
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(int), &num_entries);
clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *)&memobjs[2]);

// nastavení počtu work-items a spuštění kernelu
global_work_size[0] = num_entries; // celkový počet work-items
local_work_size[0] = 64; // velikost jedné work-group
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size,
    local_work_size, 0, NULL, NULL);
```

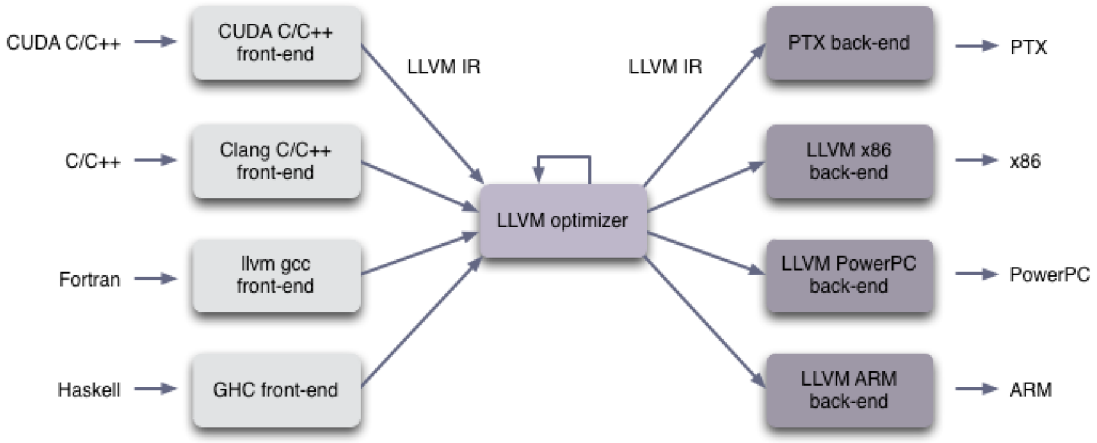
V následující kapitole bude ukázáno, jakým způsobem lze OpenCL aplikace napsané v C/C++ společně s kernely napsanými v OpenCL C překládat pomocí překladače Clang/LLVM.

Kapitola 3

Kompilační framework LLVM

LLVM [18] (nejedná se o zkratku ale plný název projektu) je překladový framework, kombinující modulární překladač a další nástroje potřebné pro překlad aplikací. K těmto nástrojům patří assembler, disassembler, linker a další. LLVM bylo vytvořeno za účelem podpory transparentních analýz a transformací po celou dobu života programu. K tomu poskytuje vysokoúrovňové informace důležité pro transformace při překladu, linkování, běhu a době nečinnosti mezi běhy. Původně se jednalo o výzkumný projekt na University of Illinois [19]. Framework je napsán v jazyce C++ a vlastním programovacím jazyce zvaném *TableGen*, který slouží pro zachycení informací o vlastnostech cílové architektury.

LLVM implementuje tzv. třífázový design, což v tomto případě znamená, že zde existuje přední část (*frontend*), prostřední část (*midend*) nebo také optimalizátor (*optimizer*) a zadní část (*backend*). Tyto tři části by na sobě měly být navzájem nezávislé. Hlavní výhodou třífázového designu je, že při potřebě podpory nového programovacího jazyka stačí upravit jen frontend a při potřebě podpory nové architektury stačí analogicky upravit jen backend. Není nutné vytvářit znovu celý překladač [17]. Schéma implementace tohoto designu v LLVM je na obrázku 3.1.



Obrázek 3.1: Implementace třífázového designu v LLVM [27]

Frontend zpracovává kód napsaný ve vstupním programovacím jazyce, provádí jeho lexikální, syntaktickou a sématickou analýzu, kontroluje jej na chyby a provádí jeho překlad do vnitřní reprezentace zvané LLVM Intermediate Representation (IR). Frontend by měl být

v ideálním případě nezávislý na architektuře, pro kterou je překlad prováděn, ovšem existují situace kdy je lepší tuto zásadu porušit (např. za účelem vytváření kvalitnějšího kódu). V LLVM existují implementace frontendu pro různé programovací jazyky, z nichž nejpoužívanější je `Clang`, frontend podporující jazyky C, C++ a Objective C [30].[1] Clang částečně porušuje nezávislost na architektuře, o které musí mít vždy dostupné určité informace v podobě tzv. *data-layout*u. Ten obsahuje informace o nativně podporovaných datových typech, velikosti ukazatele, zda se jedná o *little-endian* nebo *big-endian* architekturu apod.

Vstupem další části překladače – optimalizátoru – je IR vytvořený frontendem. Nad ním je prováděna řada tzv. optimalizačních průchodů spojených s pomocnými analýzami, jejichž společným cílem je vylepšit tento kód, což znamená urychlit vykonávání výsledného programu nebo co nejvíce zmenšit jeho velikost. To vše samozřejmě při zachování jeho sémantiky. V LLVM je optimalizátor reprezentován programem `opt`. Pomocí argumentů příkazové řádky je možné si vybrat, které průchody mají být provedeny, což lze provést jak jednotlivě, tak i hromadně použitím předdefinovaných skupin průchodů, majících společný cíl. V ideálním případě by měl být optimalizátor nezávislý jak na zdrojovém programovacím jazyce, tak na cílové architektuře, ovšem stejně jako v případě frontendu, i zde jsou potřeba pro zajištění efektivity určitých optimalizací informace např. o nativně podporovaných operacích na cílové architekuře. Některé základní optimalizace probíhají už při překladu zdrojového programu do LLVM IR ve frontendu a také backend může kód ještě dodatečně optimalizovat pro cílovou architekturu [30].

Backend překladače (někdy nazývaný generátor kódu), který je v LLVM reprezentován programem `llc`, dostane IR (buď od optimalizátoru nebo v případě vynechání optimalizací přímo od frontendu), který převede na nativní symbolický kód cílové architektury. Ten je pak pomocí assembleru dále převeden na objektový soubor a ten pomocí linkeru (program `ld`) spojen s dalšími objektovými soubory a je vytvořen výsledný spustitelný program [30].

3.1 LLVM Intermediate Representation

LLVM IR je způsob, jakým je reprezentován překládaný kód napříč jednotlivými částmi překladače. Má podobu nízkoúrovňové instrukční sady typu RISC, kde jednotlivé instrukce jsou tzv. v tříadresné formě. Takové instrukce mají dva operandy, nad kterými je provedena určitá operace a výsledek je uložen do třetího operandu. IR používá tzv. SSA (*Static Single Assignment*) formu, jejíž podstata spočívá v tom, že každá proměnná je přiřazena (vyskytuje se na levé straně příkazu) právě jednou. Tato forma zjednodušuje a vylepšuje výsledky různých optimalizací, generování cílového kódu a podobně [30].

IR podporuje lineární sekvence jednoduchých instrukcí jako jsou aritmetické, logické a porovnávací operace. Nechybí podpora pro návěští. Přes svůj vzhled podobný jazyku symbolických instrukcí umožňuje reprezentovat všechny funkce vyšších programovacích jazyků [30]. Může se vyskytovat celkem ve třech formách: klasická textová podoba čitelná člověkem, forma datových struktur přítomných v paměti (s touto formou vnitřně pracují jednotlivé nástroje) a na disk ukládaný binární kód, tzv. bitkód (vhodný např. pro rychlé načítání v rámci *Just-In-Time* kompilace). Všechny tři formy jsou navzájem ekvivalentní a převoditelné [17]. Jak vypadá IR v textové formě je ukázáno na příkladu 3.1.1.

Struktura programu se skládá z několika částí [21]: na nejvyšší úrovní je zde tzv. modul, což je překladová jednotka vstupního programu, typicky odpovídající jednomu vstupnímu zdrojovému souboru. Modul se skládá z funkcí, které opět typicky odpovídají jednotlivým

[1] Další implementací frontendu je např. GHC podporující jazyk Haskell [22].

funkcím příp. metodám ve vstupním programu. Obsah funkce je tvořen instrukcemi. Ty jsou rozděleny do tzv. základních bloků (*basic blocks*), což jsou nepřerušitelné posloupnosti instrukcí. To znamená, že základní blok má pevně definovaný vstup (návěští na jeho začátku) a výstup (instrukce skoku nebo návratu z podprogramu na jeho konci). Nelze tedy provést skok doprostřed základního bloku a stejně tak nelze zprostřed vyskočit ven. V modulu se rovněž vždy nachází tzv. metadata. To jsou data o datech, která jsou připojena k jednotlivým instrukcím a v nichž jsou uloženy dodatečné informace o kódu pro optimalizátor a generátor kódu. Využívat metadata může rovněž debugger, v případě překladu s ladicími informacemi se zde nachází např. čísla řádků a sloupců jednotlivých příkazů v původním programu [30].

Kód v IR dále obsahuje globální proměnné, *data-layout* cílové architektury, deklarace funkcí a další části, jejichž detailnější popis lze najít v manuálu [21].

Příklad 3.1.1. Ukázka překladu jednoduché funkce z jazyka C do LLVM IR.

```
unsigned mul(unsigned a, unsigned b) { // vynásobí dvě čísla
    return a * b;
}
```

```
define i32 @mul(i32 %a, i32 %b) { ; stejná funkce v LLVM IR
entry:
    %tmp1 = mul i32 %a, %b
    ret i32 %tmp1
}
```

3.2 OpenCL C frontend

Clang od verze 3.0 podporuje i kompilaci kódu napsaného v OpenCL C. Pro zapnutí této podpory je nutné překládat s parametrem `-x cl`, který frontendu řekne, že zdrojový soubor je napsán právě v OpenCL C (kód v C a C++ je detekován automaticky). Podpora se týká nových klíčových slov, datových typů (např. vektorové typy) a mírně odlišné sémantiky některých příkazů. Ukázka překladu je v příkladu 3.2.1.

Na příkladu je vidět, že Clang přeložil kernel stejně jako jakoukoli jinou funkci a nijak neřešil to, že se vlastně jedná jen o popis jedné výpočetní instance (jednoho work-item). Všechna volání funkcí (včetně bariéry) nechal tak jak jsou a nenahradil je odpovídající sémantickou akcí. To je úkolem až konkrétní použité implementace OpenCL, která takto přeložený OpenCL C kód dále zpracuje a transformuje, tedy vloží dodatečný kód zajišťující provedení kernelu pro všechny work-items a všechny work-groups, nahradí příkaz bariéry konkrétním kódem, který synchronizuje všechny work-items apod. Rovněž je vidět, že Clang na úrovni IR reprezentuje jednotlivé paměťové regiony (globální, lokální, ...) pomocí označení příslušných ukazatelů parametrem `addrspace(num)`. Ten říká, že daný ukazatel obsahuje adresu do určitého adresného prostoru. Adresné prostory jsou na úrovni IR očíslovány a každému OpenCL paměťovému regionu je přiděleno jiné číslo adresného prostoru (např. globální paměť je mapovaná do adresného prostoru číslo 1).

Příklad 3.2.1. Překlad jednoduchého kernelu z jazyka OpenCL C do LLVM IR pomocí Clangu.

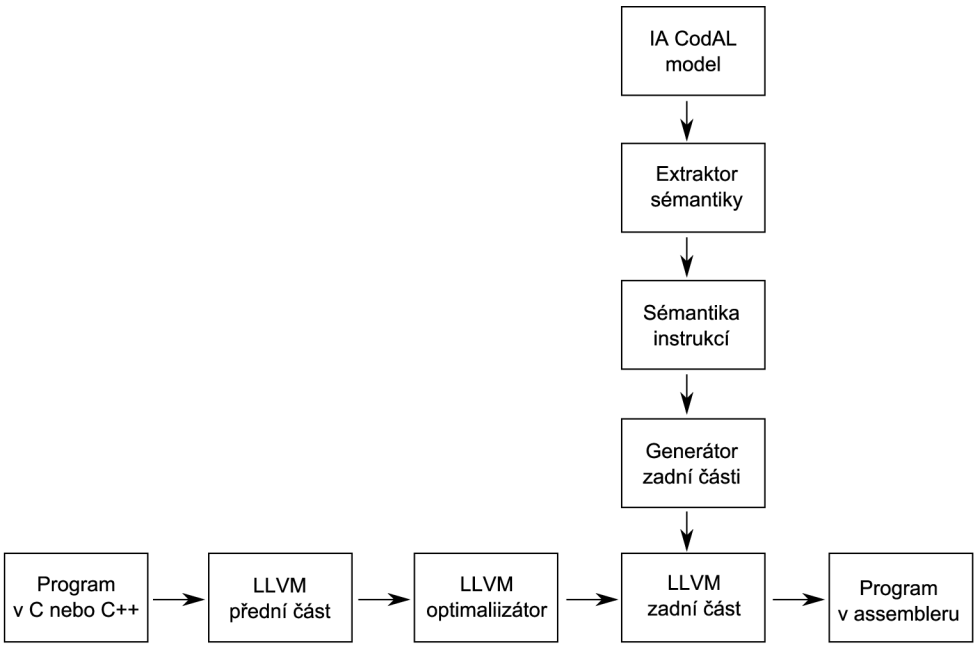
```
__kernel void demo(__global int *src, __global int *dst, int factor)
{
    int i = get_global_id(0);
    // bariéra, zde pouze na ukázku
    barrier(CLK_GLOBAL_MEM_FENCE);
    // vynásobí jeden prvek pole zadaným faktorem
    dst[i] = src[i] * factor;
}
```

```
define void @demo(i32 addrspace(1)* %src, i32 addrspace(1)* %dst,
   i32 %factor) {
entry:
    %call = tail call i32 @get_global_id(i32 0)
    tail call void @barrier(i32 1)
    %arrayidx = getelementptr inbounds i32, i32 addrspace(1)* %src,
       i32 %call
    %0 = load i32, i32* %arrayidx, align 4
    %mul = mul nsw i32 %0, %factor
    %arrayidx3 =getelementptr inbounds i32, i32 addrspace(1)* %dst,
       i32 %call
    store i32 %mul, i32* %arrayidx3, align 4
    ret void
}
```

Kapitola 4

Codasip Studio

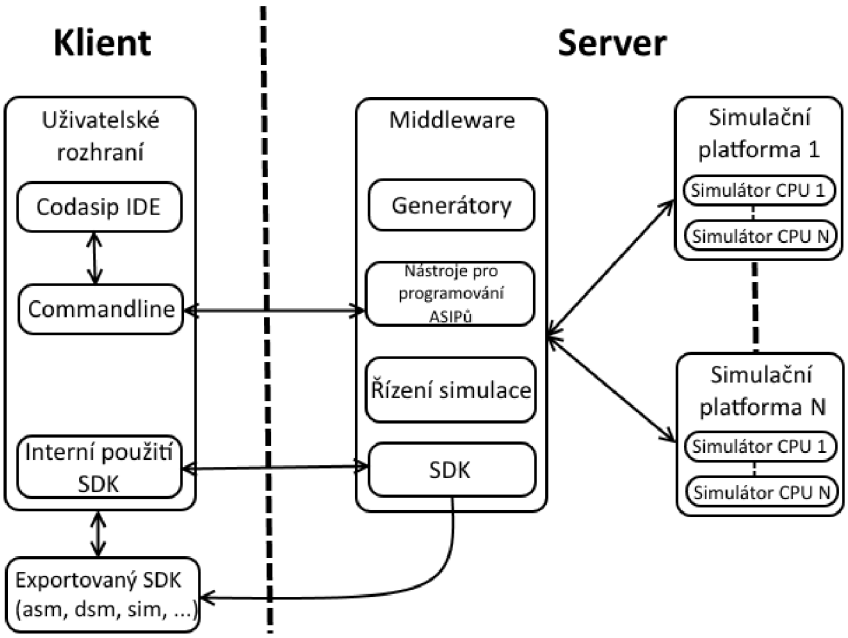
Codasip Studio je vysoce automatizované, plně integrované vývojové prostředí, pokrývající všechny aspekty návrhu procesorů s aplikačně specifickou instrukční sadou (ASIP) a platforem obsahujících kromě těchto procesorů i paměti, sběrnice a další součásti systémů na čipu. Současně umožňuje i tvorbu programů pro tyto procesory. Poskytuje plně automatické generování programovacích a simulačních nástrojů a generování syntetizovatelného RTL (*Register Transfer Level*) popisu architektury včetně funkční verifikace. Mezi konkrétní generované nástroje patří: assembler, disassembler, C/C++ překladač založený na LLVM a simulátor (několik typů) [3]. Schéma generování překladače a překlad programu je ukázáno na obrázku 4.1.



Obrázek 4.1: Schéma generování překladače v Codasip Studiu a překlad programu [30]

Z modelu procesoru v jazyce CodAL je extrahována sémantika instrukcí a z té je následně pomocí generátoru vygenerována zadní část překladače. Zdrojový kód programu je přeložen do IR a optimalizován pomocí společné přední části překladače a optimalizátoru. Poté jej vygenerovaná zadní část přeloží do instrukcí cílové architektury. Z této reprezentace pak již lze pomocí assembleru a linkeru vytvořit spustitelný program.

Codasip Studio se skládá ze tří vrstev znázorněných na obrázku 4.2: vrstvy uživatelského rozhraní, serverové vrstvy (také zvané middleware) a simulační vrstvy. Jednotlivé vrstvy spolu komunikují prostřednictvím TCP/IP protokolu, což umožňuje aby běžely na různých místech v síti. Navíc Codasip Studio umožňuje exportování nezávislé sady nástrojů pro vývoj software (*Software Development Kit* - SDK) [3].



Obrázek 4.2: Vrstvy Codasip Studia

Vrstva uživatelského rozhraní přijímá vstupy od uživatelů (jako je příkaz pro spuštění simulace) a zobrazuje důležité informace (jako je výsledek simulace). Skládá se ze dvou částí: první je Codasip IDE (*Integrated Development Environment*, integrované vývojové prostředí) postavené na platformě Eclipse a poskytující grafické uživatelské rozhraní; druhou, určenou spíše pro zkušenější uživatele, je Codasip Commandline, což je textové rozhraní ve formě příkazové řádky. Codasip Commandline lze použít pro skriptování, automatické testování a další pokročilou práci. Vrstva uživatelského rozhraní obousměrně komunikuje s middleware.

Serverová vrstva (nebo middleware) přijímá a zpracovává příkazy z vrstvy uživatelského rozhraní. Skládá se z [3]:

- generátorů sady vývojových nástrojů pro generování assembleru, překladače, simulátoru a dalších nástrojů potřebných pro vyvíjený ASIP,
- výsledných vygenerovaných nástrojů,
- důležitých aplikačních programových rozhraní: pro simulaci externích modelů a pro kosimulaci např. se SystemVerilog simulátory,
- generátoru pro generování prostředí pro funkční verifikaci a
- vysokoúrovňového syntézního nástroje pro tvorbu syntentizovatelného RTL z modelu ASIPu.

Serverová vrstva rovněž obsahuje funkce pro export SDK a pro správu distribuovaných simulací.

Pokud se jedná o příkaz, který dokáže middleware sám přímo vykonat (jako je generování SDK), tak to udělá a vrstvu uživatelského rozhraní informuje o výsledku. Pokud obdrží příkaz, který sám vykonat nedokáže (to se týká příkazů pro simulátor, např. nastavení breakpointu) přepošle jej odpovídajícímu simulátoru a ten jej provede. Middleware je rovněž zodpovědný za instalaci simulátoru do simulační vrstvy. Simulátory mohou být nainstalovány na jakékoli vhodné umístění v síti.

Simulační vrstva je řízena příkazy z middleware. Sestává z jedné nebo více simulačních platforem, kde každá obsahuje vlastní simulátory. Simulátorů je v platformě více v případě, že dochází k simulaci víceprocesorového systému, každý simulátor pak reprezentuje jeden procesor. Stejně jako spolu může komunikovat více procesorů uvnitř počítače, můžou spolu komunikovat i jednotlivé simulátory. Jeden middleware může současně řídit více simulačních platforem (více multiprocesorových systémů) [3].

Hlavním úkolem této práce je rozšířit automaticky generované nástroje o schopnost překládat a spouštět programy napsané v OpenCL a to pomocí integrace některé z volně dostupných implementací tohoto frameworku.

Kapitola 5

Dostupné implementace OpenCL

Implementací OpenCL existuje velké množství. Pro potřeby integrace do Codasip Studia je nutné zvolit takovou, která je open source (jsou k ní dostupné zdrojové kódy), její licenční podmínky umožňují její libovolné úpravy a následné používání i ke komerčním účelům bez nutnosti platit licenční poplatky.

5.1 Proprietární implementace

Mnoho výrobců hardwaru poskytuje podporu pro OpenCL jako součást ovladačů svých zařízení. Tyto implementace jsou vždy určeny pro konkrétní skupinu zařízení daného výrobce a nejsou k nim z pochopitelných důvodů dostupné zdrojové kódy. Rovněž licenční podmínky neumožňují jejich komerční využití třetí stranou. Z těchto důvodů není jejich integrace do Codasip Studia možná a zde budou zmíněny jen kvůli úplnosti.

Na druhou stranu je ovšem nutné zmínit, že tyto implementace jsou většinou vysoce kvalitní, což je dáno jejich profesionálním vývojem.

5.1.1 Intel

Na operačních systémech Windows a Mac OS je podpora pro OpenCL přímo součástí ovladačů pro grafické karty této firmy. Podpora se týká grafických čipů Intel HD a Iris integrovaných do procesorů 3. až 5. generace. Pro podporu vývoje v OpenCL Intel poskytuje sadů nástrojů zvanou Intel SDK for OpenCL Applications. Aktuální verze, podporující OpenCL 2.0, je Intel SDK for OpenCL Applications 2013 [8].

Na linuxových systémech lze využít projekt zvaný Beignet [9], viz dále.

5.1.2 AMD

Společnost AMD, jenž stojí i za grafickými kartami Radeon, rovněž nabízí vlastní implementaci OpenCL a sadu nástrojů pro vývoj aplikací nazvanou AMD APP SDK [1] (APP - *Accelerated Parallel Processing*). Podporovány jsou grafické karty od řady AMD Radeon HD 5x00 výše a rovněž AMD APU (*Accelerated Processing Unit*) řady Fusion.

V současné verzi nástrojů (verze 3.0), určené pro Windows i Linux, je podporováno OpenCL 2.0 a podpora pro nové OpenCL 2.1 je v plánu.

5.1.3 Další implementace

Mezi další existující proprietární implementace patří OpenCL Development Kit [7] od firmy IBM určený pro linuxové servery postavené na technologii IBM Power a také nástroje od firmy Nvidia. Ty jsou určeny pro grafické karty podporující technologii CUDA, ovšem podpora pro OpenCL je z pochopitelných důvodů limitována (jedná se v podstatě o konkurenční technologii), např. nejvyšší podporovaná verze OpenCL je 1.2.

5.2 Open source implementace

K implementaci použité pro začlenění do Codasip Studia musí být dostupné zdrojové kódy, aby ji bylo možné upravit pro potřeby běhu na vestavěných zařízeních, kde např. není dostupný runtime překladač. Takových, které by tyto podmínky splňovaly lze nalézt na internetu poměrně velké množství, ale bohužel většina z nich zatím není úplná v tom smyslu, že by implementovala kompletní standard OpenCL (libovolnou verzi). Zde jsou uvedeny ty, které jsou v tomto smyslu nejdál.

5.2.1 Beignet

Beignet je projekt Intelu pro podporu OpenCL na integrovaných grafických kartách Intel HD a Iris. Na Linuxu tato podpora není součástí ovladačů, ale je vyčleněna do samostaného open source projektu.

V současnosti Beignet podporuje kompletně OpenCL 1.2 a podpora pro standard 2.0 je ve vývoji.

Beignet je napsán v C a C++ a distribuován pod licencí LGPLv2.1+ (lesser GPL). Tato licence sice umožňuje komerční využití, ovšem jen za podmínky zachování této licence. Tedy v případě použití softwaru s touto licencí je nutné všem uživatelům poskytnout zdrojové kódy a to i v případě, kdy dojde k modifikacím původního kódu. To je ovšem pro využití v Codasip Studiu nevhodné.

5.2.2 pocl

Portable Computing Language (`pocl`, psáno malými písmeny) [11] cílí na to se stát open source implementací OpenCL, která může být snadno adaptována pro nová zařízení a to jak pro homogenní CPU, tak pro heterogenní GPU a akcelerátory.

`Pocl` používá Clang jako OpenCL C frontend a LLVM pro implementaci překladače kernelů (*kernel compiler*) a jako vrstvu umožňující přenositelnost. Díky tomu by mělo být snadné získat podporu pro OpenCL použitím `pocl` na jakémkoli zařízení, pro které existuje LLVM backend.

Cílem je dosáhnout lepší přenositelnosti výkonu použitím překladače kernelů, který může generovat work-group funkce, které mohou využít různé typy paralelního hardwaru: VLIW, superskalární, SIMD, SIMT, vícejádrové, vícevláknové a další.

Dalším účelem projektu je sloužit jako výzkumná platforma pro řešení problémů paralelního programování na heterogenních platformách.

`Pocl` je distribuováno pod MIT licencí, která umožňuje provádění libovolných úprav a následné komerční využívání a v současnosti implementuje valnou většinu OpenCL standardu verze 1.2. Podpora pro verzi 2.0 je v začátcích.

5.2.3 Clover

Clover [28] začal v červnu 2011 jako projekt na Google Summer of Code. Jeho cílem je poskytnout open source implementaci OpenCL použitelnou kýmkoli, kdo chce použít OpenCL nebo pro něj vyvíjet a to bez nutnosti používat proprietární ovladače nebo SDK.

V současnosti je podporováno pouze spouštění OpenCL programů v softwaru na hostitelském procesoru, ale je zde rozhraní, které by mělo umožnit budoucí použití i v heterogenních systémech (jako CPU-GPU).

Clover používá Clang/LLVM jako frontend pro kompilaci OpenCL kernelů a zdrojové kódy jsou distribuovány pod BSD licencí.

Podporováno je pouze OpenCL 1.1 a v současnosti se zdá, že vývoj Cloveru před několika lety ustal. To je problém, protože bez rozsáhlých zásahů do zdrojových kódu jej není možné integrovat do současných (a budoucích) verzí Clang/LLVM infrastruktury.

5.2.4 libclc

Libclc [29] je implementace OpenCL 1.1 knihovny používající Clang jako kernel compiler. Knihovna je distribuována pod licencemi BSD a MIT.

V tomto případě se ale jedná jen o builtin knihovnu obsahující zabudované funkce volatelné z kernelů. Funkce volatelné z hostitelského programu tato knihovna neobsahuje, stejně jako nějaké pokročilejší možnosti překladu kernelů. Pro účely integrace s nástroji Codasip Studia tak není vhodná, většina funkcionality by musela být dodatečně implementována.

Podpora pro OpenCL 1.2 je ve vývoji.

5.2.5 FreeOCL

FreeOCL [2] je CPU implementace OpenCL 1.2 pro Linux. Je dostupná pod licencí GNU LGPL v3 (lesser GPL). Narozdíl od mnoha jiných implementací používá vlastní jednoduchý C++ překladač místo LLVM/Clangu pro překlad OpenCL C kódu. To umožňuje použití běžných ladicích nástrojů pro debugování. Rovněž to přináší větší svobodu v tom, co lze v OpenCL C kódu dělat, protože je možné používat C/C++ standardní knihovnu.

Jak již bylo zmíněno, FreeOCL používá obdobnou licenci jako Beignet, což je z hlediska použití v Codasip Studiu problém. Další problém je, že projekt již pravděpodobně není aktivně vyvíjen, během posledního přibližně roku došlo jen ke sporadickým změnám a opravám. Rovněž zaměření jen na Linux a používání vlastního překladače je z hlediska úprav pro použití na vestavěných systémech a integrace do Codasip Studia problematické.

Kapitola 6

pocl - Portable Computing Language

Po zhodnocení všech dostupných open source implementací bylo zvoleno pro integraci do Codasip Studia `pocl`. Má ideální licenční podmínky, je to živý projekt, jehož vývoj stále probíhá a ze všech zkoumaných implementací nabízí nejlepší podporu OpenCL standardu. Velkou výhodou je snadná možnost přídávání podpory pro nová zařízení. V této kapitole je tak `pocl` popsáno detailněji.

6.1 Přehled

Na obrázku 6.1 je schéma jednotlivých částí, ze kterých se `pocl` skládá. Implementace je rozdělena do částí, které jsou přenositelné a spouštěné na hostitelském zařízení (hostitelská vrstva) a do částí, které implementují chování, které je specifické pro zařízení, na němž kernely poběží (vrstva zařízení). Vrstva zařízení zapouzdřuje prvky závislé na operačním systému a instrukční sadě jako je generování kódu pro cílové zařízení a režie spojená s vykonáváním kernelů.

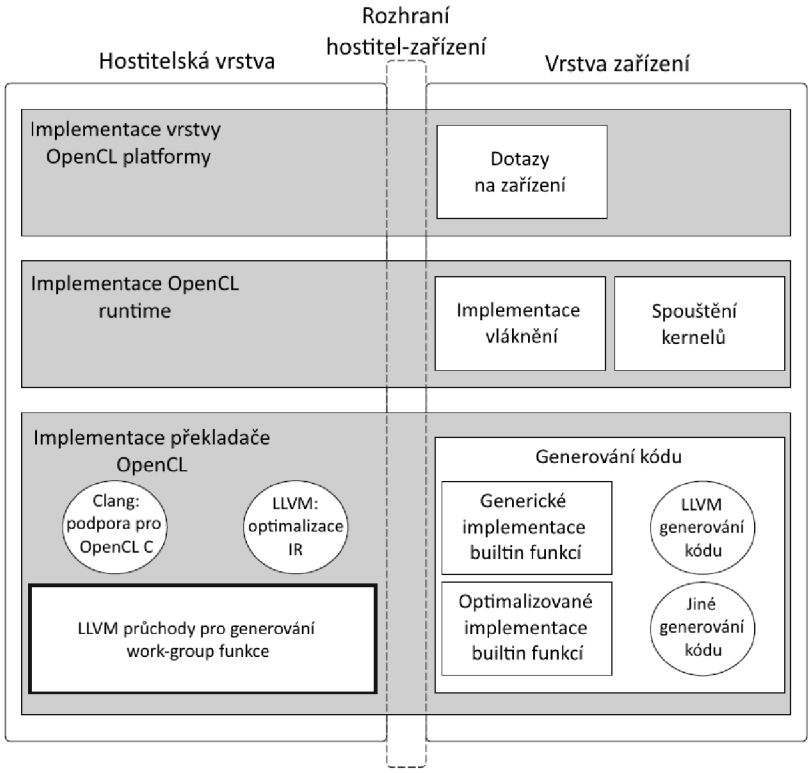
Většina implementace OpenCL API v `pocl` jsou generické implementace napsané v C, které volají vrstvu zařízení skrz generické rozhraní hostitel-zařízení pro části specifické pro dané zařízení. Například když se OpenCL program dotazuje na počet zařízení, tak `pocl` vrátí seznam podporovaných zařízení bez toho, aby bylo potřeba dělat něco závislého na konkrétním zařízení. Ovšem když se aplikace zeptá na velikost globální paměti určitého zařízení, je dotaz delegován do implementace daného zařízení.

Ve vrstvě zařízení je pro každé podporované zařízení implementována určitá funkcionalita, jako jsou na zařízení závislé části procesu kompilace kernelů, finální spuštění fronty příkazů zahrnující nahrání kernelu na zařízení a jeho spuštění, dotazování se na vlastnosti zařízení a podobně [12].

6.1.1 Pocl zařízení

`Pocl` obsahuje několik standardních implementací vrstvy zařízení, které mohou být přímo použity nebo mohou posloužit jako základ pro implementaci zařízení vlastního. Tyto standardní zařízení jsou [12]:

- **basic**
  Základní implementace, minimální příklad implementace CPU zařízení. Kernely jsou



Obrázek 6.1: Schéma pocl [12]

spouštěny sekvenčně bez využití vláken. Hostitel a zařízení jsou stejné fyzické zařízení.

- **pthread**
  Podobné jako *basic*, ovšem využívá POSIX vlákna pro spouštění více work-groups paralelně. Toto je příklad implementace vrstvy zařízení, která je schopná využít funkčního paralelismu na úrovni vláken.

- **ttasim**
  Experimentální implementace tzv. simulovaného heterogenního akcelerátoru. Tento ovladač simuluje modifikovatelné akcelerátory založené na principu *Transport-Triggered Architecture* (TTA), které mohou spouštět kernely. Procesory jsou simulovány pomocí volání simulátoru instrukční sady, který je součástí *TTA-based Co-design Environment* (TCE), což je framework pro vývoj TTE procesorů. Ovladač provádí správu pamětí zařízení na straně hostitele a řídí spouštění kernelů na zařízení.

- **cellspu**
  Další experimentální heterogenní akcelerátor. Tato implementace řídí tzv. synergistické výpočetní elementy (SPE) v heterogenní architektuře Cell, na které běží linuxový operační systém. Implementace používá knihovnu `libspe` jako rozhraní pro komunikaci s SPE.

6.2 Překladač kernelů

Jak již bylo zmíněno, `pocl` využívá pro překlad kódu napsaného v OpenCL C překladač LLVM, resp. jeho frontend Clang. To je ale jenom první fáze. Nad kódem v LLVM IR, který je výstupem Clangu, je provedena sada průchodů majících za úkol kód kernelu dále optimalizovat a upravit tak, aby mohl být spuštěn pro všechny work-groups a work-items.

Překlad probíhá v těchto krocích:

1. Překlad všech kernelů do LLVM bitkódu
   Kompilace kernelů v `pocl` využívá Clang jako OpenCL C frontend. Clang zpracuje popis kernelů a přeloží je do LLVM bitkódu. Výstupem Clangu jsou popisy kernel funkcí pro jediný work-item.

2. Přilinkování vestavěných funkcí
   Vestavěné funkce předkompilované do bitkódu jsou pomocí nástroje *llvm-link* (součást LLVM frameworku) na úrovni bitkódu slinkovány s přeloženými kernely.

3. Vytvoření work-group funkcí
   Funkce vykonávající jedinou instanci kernelu (jediný work-item) je zkonvertována na funkci, která vykonává kernel pro všechny instance v lokálním prostoru. Tento krok se provádí pro zařízení, která nedokáží automaticky spustit více instancí kernelu z popisu instance jedné. To zahrnuje běžné procesory, které nejsou optimalizovány pro vykonávání programů typu SPMD. Na druhou stranu GPU architektury (s datovými cestami typu SIMT a SIMD) toto často umí a dokáží se postarat o paralelní vykonání jednotlivých instancí kernelu pomocí svého plánovacího hardwaru.
   Tento krok je proveden, když je z hostitelského programu zavolána funkce pro spuštění kernelu (`clEnqueueNDRangeKernel` nebo `clEnqueueTask`). Až tehdy jsou totiž známy množství dat, nad kterým bude výpočet probíhat a velikost work-groups (tedy počet paralelně vykonávaných instancí). Znalost těchto informace je nutná pro vytvoření funkcí, které budou výpočet zadaným způsobem provádět.

4. Generování cílového kódu
   Funkce provádějící jednotlivé instance kernelů jsou spolu s dalšími pomocnými funkcemi a datovými strukturami přeloženy z LLVM IR do strojového kódu cílové architektury. Tento krok je tedy závislý na architektuře (předchozí kroky nebyly) a každé zařízení, které `pocl` podporuje musí mít jeho vlastní implementaci. Výsledkem je dynamicky načítaný objekt, který provádí funkce kernelů nad všemi daty zadaným způsobem.

6.2.1 Pomocné průchody a transformace

Nad LLVM IR je nutné provést množství různých transformací a průchodů, aby mohly být správně vytvořeny work-group funkce a bylo možné jednotlivé instance kernelu provádět paralelně. Nejdůležitější z těchto transformací jsou tyto:

- **Flatten**
  Tato transformace provede úplné inlinování (vložení kódu funkcí na místo jejich volání) všech funkcí volaných z kernelu. Kernel pak neobsahuje žádná volání, která by jinak bránila např. automatické vektorizaci a různým optimalizacím. Konkrétně dojde k přidání atributu `AlwaysInLine` ke všem dětským funkcím kernelu (funkcím z kernelu volaným, a to i nepřímo) a poté spuštění standardního optimalizačního průchodu

`-always-inline`, který provede samotné inlinování.
Volání některých funkcí (např. matematických jako je `sin` nebo `cos`) ovšem lze vektorizovat a taková volání tedy tento transformační průchod zachovává.

- **PHIsToAllocas**
  Konvertuje všechny ϕ uzly[1] na instrukce `alloca` aby bylo možné vkládat kód pro obnovení kontextu na začátek bodů, kde dochází ke spojení paralelního datového toku (tzv. *join points*). To je potřeba, protože ϕ uzly mohou být jen na začátku základních bloků a v některých případech je potřeba obnovit proměnné pocházející z jiného paralelního regionu (načíst je z pole kontextů v paměti) použité ve ϕ uzlech i jinde.
  Tato transformace je podobná LLVM průchodu `-reg2mem`, ovšem týká se jen ϕ uzlů.

- **AllocasToEntry**
  Tento průchod může být využit platformami, které nepodporují, aby dynamické objekty na zásobníku přesouvaly všechny zásobníkové alokace do vstupního základního bloku funkce. Tato transformace provede tento přesun už na úrovni LLVM IR.

- **AutomaticLocals**
  Transformace provádějící konverzi automatických (vytvořených na zásobníku) lokálních bufferů na argumenty kernelu. To je z důvodu vynucení stejného zacházení s oběma typy lokálních bufferů existujících v OpenCL: předávaných jako argumenty a instanciovaných v kernelu.

- **TargetAddressSpaces**
  Vnitřně používá `pocl` fixní čísla adresných prostorů pro rozlišení jednotlivých paměťových regionů OpenCL. To znamená, že Clang generuje LLVM IR, které používá tato čísla a to i pro platformy, které mají ve skutečnosti jenom jediný, plochý adresný prostor. To je z důvodu rozlišení ukazatelů do jednotlivých paměťových regionů a rovněž pro asistenci alias analýze (různé adresné prostory jsou oddělené, takže přístupy do nich se nepřekrývají).
  TargetAddressSpaces je průchod, který tato falešná čísla konvertuje na ta, která jsou očekávána cílovou platformou. Tento průchod může být i vynechám v případě, že backend pro danou platformu může takovou konverzi udělat sám. Nicméně některé optimalizační průchody v LLVM můžou mít s takto falešně očíslovanými adresnými prostory problém (včetně vektorizátorů) a tak je lepší, když jsou již na úrovni IR zkonvertovnány na skutečné, aby se těmto problémům zabránilo a optimalizace mohly být efektivnější.

- **WorkitemAliasAnalyzer**
  Dodává alias analyzátoru informace specifické pro OpenCL. V současnosti využívá faktu, že přístupy do paměti ze dvou různých instancí kernelu se nemohou překrývat v rámci stejného paralelního regionu a že paměťové regiony OpenCL jsou oddělené (přístupy do různých paměťových regionů nemohou vést k přístupu na stejné místo paměti).

[1] ϕ uzel SSA formy je instrukce provádějící výběr hodnoty na základě předcházejícího základního bloku

6.3 Runtime knihovna

Druhou důležitou součástí `pocl` je runtime knihovna, obsahující všechny OpenCL funkce volatelné z hostitelského programu. Patří sem např. funkce `clBuildProgram` pro překlad kernelů za běhu programu nebo `clEnqueueNDRangeKernel` pro spouštění kernelů.

Celá runtime knihovna je napsána v jazyce C z důvodu zajištění její funkčnosti na co nejširší skupině možných zařízení. Z tohoto důvodu se upustilo od implementace v C++, která by nemohla být plnohodnotně spuštěna např. na zařízeních nepodporujících zpracování výjimek, virtuálních metod apod.

Každá jednotlivá funkce je umístěna ve vlastním zdrojovém souboru a překládána odděleně, z výsledných objektových souborů a několika dalších pomocných modulů (obsahujících např. pomocné funkce pro správu alokované paměti) je následně sestavena buď dynamická nebo statická knihovna, případně obě. To, jaký typ knihovny bude vytvořen si lze zvolit při konfiguraci `pocl`, standardně je vytvářena pouze dynamická verze.

Runtime knihovna je přilinkována k hostitelské OpenCL aplikaci buď při spuštění (dynamická knihovna) nebo již při překladu (statická knihovna).

6.4 Knihovna builtin funkcí

Builtin (zabudované, vestavěné) jsou takové funkce, které lze volat z kernelů bez toho, aby bylo nutné tyto funkce někde deklarovat nebo definovat. Patří sem hlavně různé matematické operace - trigonometrické, zaokrouhlovací, speciální funkce pro práci s čísly v plovoucí řádové čárce a další. Jsou zde i implementace funkcí pro zjišťování velikosti work-group, jejich počtu nebo pro získání globálního a lokálního identifikátoru instance kernelu.

Stejně jako u runtime knihovny, i zde je každá jednotlivá funkce implementována ve vlastním zdrojovém souboru, funkce ovšem nejsou napsány v jazyce C, nýbrž v OpenCL C. Tyto soubory jsou přeloženy jen do LLVM bitkódu a jednotlivé moduly jsou pak slinkovány programem `llvm-link` do jednoho modulu rovněž v bitkódu. Při překladu OpenCL kernelů je k nim tato knihovna přilinkována opět pomocí `llvm-link`, ovšem jsou z ní vybrány jen ty funkce, které jsou v kernelech skutečně použity. Ty jsou následně inlinovány na místo jejich volání pomocí průchodu Flatten, popsaného v sekci 6.2.1.

6.5 Knihovna pomocných funkcí

Poslední knihovnou je `poclu` (`pocl` *utility library*), která obsahuje různé užitečné pomocné funkce použitelné z hostitelské aplikace. Nachází se zde např. funkce pro práci s datovými strukturami, funkce provádějící převody mezi *little endian* a *big endian* uspořádáním bytů v proměnných, funkce obalující různá volání OpenCL runtime knihovny a další. `Poclu` obsahuje i softwarovou implementaci operací nad datovým typem half float (16-bitové floating point číslo).

Funkce obsažené v této knihovně nejsou součástí standardu a autoři `pocl` je vytvořili jen za účelem zjednodušení zápisu často používaných konstrukcí. Většina testů dodávaných se zdrojovými soubory `pocl` knihovnu `poclu` využívá.

Kapitola 7

Integrace pocl do překladače LLVM

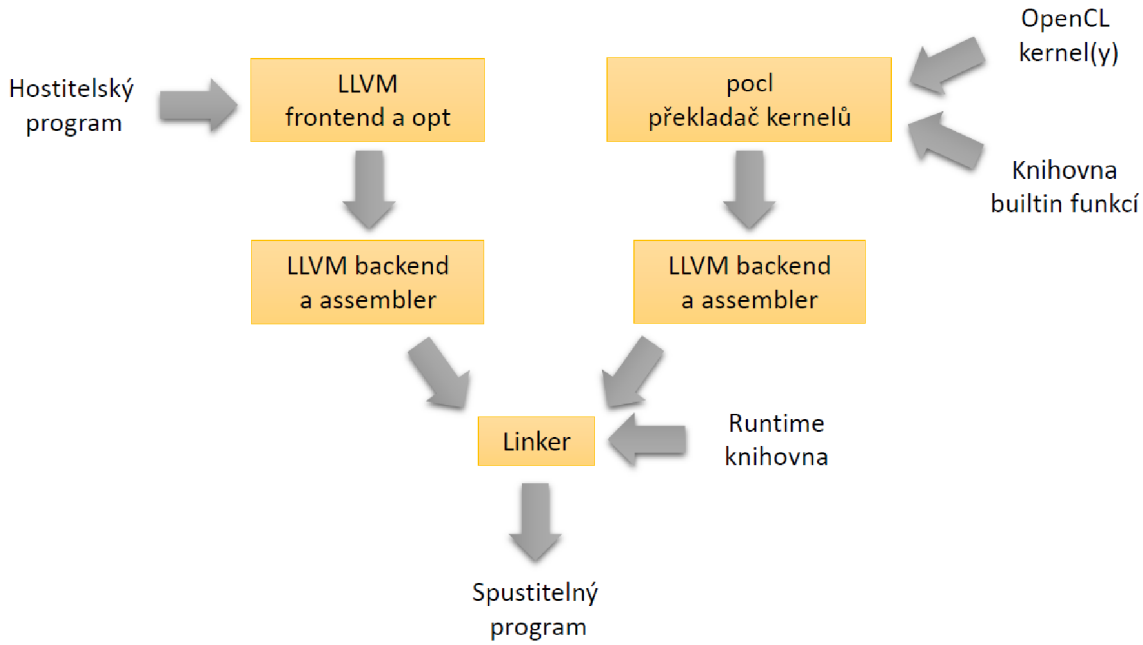
Pro úspěšný překlad a spouštění OpenCL aplikací pomocí nástrojů poskytovaných Codasip Studiem bylo nejprve nutné `pocl` integrovat do automaticky generovaného překladače. Dále bylo nutné najít způsob, jakým budou OpenCL aplikace (a zvláště jejich kernely) na cílových architekturách překládány a spouštěny. Zde bylo potřeba se vyrovnat s faktem, že zatímco OpenCL standard definuje, že jednotlivé kernely aplikace jsou zkompilovány za běhu s nastavením dostupným právě v té době (tzv. *Just-In-Time* překlad), v tomto případě nic takového možné není, vygenerovaný překladač dodatečnou kompilaci za běhu nepodporuje. Tato kapitola tak popisuje způsob řešení těchto problémů.

7.1 Překlad pocl

Jednotlivé části, které v `pocl` tvoří překladač kernelů, jsou koncipovány jako optimalizační průchody pro optimalizátor (program `opt`). Optimalizátor umí tyto průchody dynamicky načíst a spustit, čehož právě `pocl` využívá. Standardně (tedy podle návodu od tvůrců `pocl`) jsou všechny průchody tvořící překladač kernelů přeloženy do objektových souborů, společně slinkovány a je vytvořena dynamická knihovna. Ta je pak pomocí parametru `-load` optimalizátorem načtena a pomocí dalších parametrů jsou zvoleny konkrétní průchody, které budou nad vstupním souborem spuštěny.

Pro vývojáře překladače v Codasipu je tento postup (nutnost odděleného překladu LLVM/Clangu a `pocl`) poněkud nepraktický. Proto bylo rozhodnuto `pocl` integrovat přímo mezi zdrojové kódy samotného LLVM a překládat vše dohromady. Adresář `lib/llvmopencl` z `pocl` obsahující průchody překladače kernelů tak byl umístěn do adresáře `lib/Codasip` ve verzi LLVM používané v Codasip Studiu. Původní soubory `Makefile` byly odstraněny a nahrazeny jediným souborem `CMakelist.txt` obsahujícím seznam překládáných souborů (průchodů) a jméno nového adresáře bylo přidáno do nadřazeného `CMakelistu`. Průchody jsou tak přeloženy automaticky pokaždé, když je přeloženo LLVM, což usnadňuje vývojářům práci. Bylo rozhodnuto, že bude zachován koncept překladače kernelů coby dynamické knihovny; průchody tedy nejsou součástí `optu`, ale musí k němu být dynamicky přilinkovány. Tímto způsobem nedojde ke zbytečnému zvětšení velikosti tohoto programu pro uživatele, kteří podporu pro OpenCL nepotřebují.

7.2 Statický překlad kernelů



Obrázek 7.1: Překlad OpenCL aplikace se statickým překladem kernelů

Jak již bylo řečeno v sekci 7.1, průchody tvořící překladač kernelů byly vyčleněny do samostatné dynamické knihovny. Pro konkrétní architekturu je vždy rovněž přeložena knihovna builtin funkcí do podoby samostaného modulu v LLVM bitkódu. Pro plně statický překlad všech částí OpenCL aplikace včetně kernelů, a tím umožnění použití na vestavěných zařízeních, byl implementován následující postup:

1. **Vyčlenění kernelů do samostatných souborů**
   Kód kernelů v jazyce OpenCL C je oddělen od kódu hostitelské aplikace a vložen do samostatného souboru. V jednom souboru se může nacházet více kernelů nebo může být vytvořen samostaný soubor pro každý kernel, oba přístupy lze rovněž kombinovat (více souborů a v každém více kernelů). Do souboru s kernelem je rovněž nutné přidat všechny uživatelské funkce z něj volané. Pokud je použito rozdělení kernelů do více souborů je nutné každý z nich překládat zvlášť.

2. **Překlad kernelů do bitkódu**
   Soubor obsahující kernely je frontendem překladače (Clangem) přeložen do bitkódu (soubor `.bc`). Pomocí přepínače `-ffake-address-space-map` je frontendu řečeno, aby různé paměťové regiony použité v kernelech reprezentoval v bitkódu pomocí různě očíslovaných adresných prostorů. Konkrétní mapování je uvedeno v tabulce 7.1. Výchozí adresný prostor (při neuvedení žádného specifikátoru) má číslo 0. Rovněž je použit přepínač `-cl-kernel-arg-info`, který způsobí, že Clang uloží informace o argumentech kernelů (jejich názvy, datové typy a paměťové regiony) do metadat výsledného modulu.

Paměťový region OpenCL	Adresný prostor LLVM
`__global`	1
`__local`	2
`__constant`	3
`__private`	4

Tabulka 7.1: Mapování paměťových regionů na adresné prostory

3. **Vygenerování hlavičkového souboru**
   Pomocí upravené verze průchodu `generate-header`, který je součástí překladače kernelů, je vygenerován hlavičkový soubor jazyka C obsahující informace o vlastnostech překládaných kernelů ve formě konstant a maker preprocesoru. Tento hlavičkový soubor obsahuje pro každý kernel následující informace:

   - počet argumentů a pro každý boolovské hodnoty, vyjadřující zda se jedná o ukazatel, lokální argument, argument typu `image` nebo typu `sampler`
   - požadovaná velikost lokálních work-group v dimenzích x, y a z nastavená pomocí atributu `reqd_work_group_size` uvedeného před kernelem (více o používání tohoto atributu v kapitolách 8 a 9)
   - počet lokálních proměnných a jejich velikosti v bajtech
   - informace získané z metadat vygenerových Clangem v předchozím kroku: paměťový region každého argumentu, zda se jedná o argument pouze pro čtení, pouze pro zápis nebo obojí; zda je argument označen některým z klíčových slov `const`, `restrict` nebo `volatile`, název datového typu a název samotného argumentu

   Jak takový vygenerovaný hlavičkový soubor vypadá je ukázáno na příkladu 7.2.1.

   Příklad 7.2.1. Vygenerovaný hlavičkový soubor pro kernel z příkladu 3.2.1

```
#define _cldemo_NUM_ARGS 3
#define _cldemo_ARG_IS_POINTER {1, 1, 0}
#define _cldemo_ARG_IS_LOCAL {0, 0, 0}
#define _cldemo_ARG_IS_IMAGE {0, 0, 0}
#define _cldemo_ARG_IS_SAMPLER {0, 0, 0}
#define _cldemo_REQD_WG_SIZE {1, 1, 1}
#define _cldemo_NUM_LOCALS 0
#define _cldemo_LOCAL_SIZE {}
#define _cldemo_HAS_ARG_METADATA 31
#define _cldemo_ARG_ADDR_QUALIFIER {4507, 4507, 4510}
#define _cldemo_ARG_ACCESS_QUALIFIER {4515, 4515, 4515}
#define _cldemo_ARG_TYPE_QUALIFIER {0, 0, 0}
#define _cldemo_ARG_TYPE_NAME {"int*", "int*", "int"}
#define _cldemo_ARG_NAME {"src", "dst", "factor"}
```

4. **Přilinkování builtin knihovny**
   Soubor s kernely přeložený do bitkódu je na této úrovni slinkován s knihovnou builtin funkcí programem `llvm-link`. Vznikne tak jediný modul, který obsahuje kernely, uživatelské funkce a všechny builtin funkce (včetně těch, které žádný z kernelů nepoužívá).

5. **Spuštění překladače kernelů**
   V tomto kroku dojde ke spuštění všech průchodů, které tvoří překladač kernelů, nad modulem vytvořeným v předchozím bodě. Mezi spuštěnými průchody jsou jak všechny popsané v sekci 6.2.1, tak i další pomocné a optimalizační. Spuštěny jsou rovněž standardní optimalizace přítomné v LLVM sdružené pod přepínačem `-O3`. V rámci tohoto kroku tak dojde např. k implementaci bariér, inlinování všech funkcí volaných z kernelů (kromě matematických) a nahrazení falešných čísel adresných prostorů čísly platnými na cílové architektuře. Všechny nepoužité funkce, včetně funkcí, které byly nyní nainlinovány, jsou z modulu odstraněny.

   Příklad 7.2.2. Pomocný kód pro spuštění kernelu pro každý work-item (varianta, kdy pouze dimenze x má velikost > 1, nejčastější případ)

```
int num_groups_y, num_groups_z;
const int local_size_x = local_work_size[0];
const int global_size_x = global_work_size[0];
const int num_groups_x = global_size_x / local_size_x;

pocl_context context;
context.work_dim = work_dim;
context.global_offset[0] = 0;
context.global_offset[1] = 0;
context.global_offset[2] = 0;

context.num_groups[0] = num_groups_x;
context.num_groups[1] = 1;
context.num_groups[2] = 1;
context.group_id[1] = 0;
context.group_id[2] = 0;

// run kernel for all work-groups
for (unsigned gid_x = first_gidx; gid_x <= last_gidx; gid_x++) {
    context.group_id[0] = gid_x;
    __opencl_launch_wg(args, &context);
}
```

6. **Vygenerování pomocného kódu**
   Následně dojde k vygenerování kódu v jazyce C/C++, který je připojen k již dříve vygenerovanému hlavičkovému souboru a který má dvě hlavní funkce:

   - **spuštění kernelu**: kód, který zajišťuje provedení kernelu pro každý work-item v globálním indexovém prostoru. Tento kód je volán z runtime knihovny při požadavku na spuštění kernelu (runtime knihovna nevolá kernel přímo). Část tohoto kódu (případ, kdy work-groups mají velikost větší než jedna pouze v x-ové dimenzi) je ukázána na příkladu 7.2.2. V případě, že i velikost v y-ové resp. z-ové dimezi jsou větší než jedna, obsahuje pomocný kód jednu, resp. dvě dodatečné vnější smyčky. Velikosti v jednotlivých dimenzích jsou spočteny za běhu z celkového množství dat zadaného uživatelem v hostitelské aplikaci a z hodnot zadaných atributem `reqd_work_group_size`.
   - **registrace kernelu**: kód, který při spuštění programu zaregistruje kernel do globálních datových struktur runtime knihovny. Ta tak od začátku zná všechny

kernely tvořící aplikaci a může např. kontrolovat datové typy hodnot argumentů, jejich počet, odpovídat na uživatelské dotazy na vlastnosti kernelů a podobně. Součástí zaregistrovaných informací o kernelu je rovněž adresa funkce, kterou je potřeba zavolat pro spuštění kernelu. Využívá se zde menšího triku, že při startu programu dojde automaticky k zavolání konstruktoru všech globálních C++ objektů. Registrační kód je tak realizován jako globální objekt, v jehož konstruktoru dojde k naplnění datové struktury všemi informacemi o kernelu a k zavolání funkce `_register_opencl_kernel` (ukázáno na příkladu 7.2.3), která byla implementována uvnitř runtime knihovny. Při startu programu, ještě před zavoláním funkce `main`, pak dojde k provedení tohoto konstruktoru a tím i registraci kernelu.

7. **Překlad pomocného kódu a slinkování s kernely**
   Pomocný kód je přeložen do bitkódu, zoptimalizován a následně programem `llvm-link` slinkován s modulem obsahujícím kernely do výsledného `.bc` souboru.

Příklad 7.2.3. Kód registračního objektu pro registraci kernelu při startu programu

```
class cldemo_kernel {
public:
    cldemo_kernel() {
        kernel_obj.name = "cldemo";
        kernel_obj.call = __opencl_trampoline_mt_cldemo;
        kernel_obj.num_args = _cldemo_NUM_ARGS;
        kernel_obj.num_locals = _cldemo_NUM_LOCALS;
        kernel_obj.has_arg_metadata = _cldemo_HAS_ARG_METADATA;
        ...

        _register_opencl_kernel(&kernel_obj);
    }
private:
    _OpenCLKernel kernel_obj;
} _kernel_initializer_obj_cldemo;
```

Všechny výše popsané kroky byly implementovány v podobě skriptu v jazyce Python pojmenovaném `opencl-kernel-compiler`, který je součástí sady nástrojů vygenerované pomocí Codasip Studia. Hostitelská část aplikace se přeloží překladačem klasickým způsobem spolu s modulem v bitkódu, který je výsledkem tohoto skriptu a pomocí argumentu `-lOpenCL` slinkuje s runtime knihovnou. Schéma překladu typické OpenCL aplikace se statickým překladem kernelů je na obrázku 7.1 a seznam potřebných příkazů včetně spuštění výsledného programu je v příkladu 7.2.4.

Příklad 7.2.4. Příkazy pro překlad a spuštění typické OpenCL aplikace pomocí vygenerovaných nástrojů

```
~/export/urisc/bin/urisc-opencl-kernel-compiler cldemo.cl
~/export/urisc/bin/urisc-clang cldemo.c cldemo.bc -o cldemo.xexe -O3
    -lsim -lOpenCL
~/export/urisc/bin/urisc-isimulator.ia -r cldemo.xexe
```

7.3 Úpravy runtime knihovny

Pro použití s rekonfigurovatelným překladačem, generovaným pro konkrétní architekturu pomocí Codasip Studia, bylo nutné runtime knihovnu `pocl` upravit. Runtime knihovna se spolu s knihovnou builtin funkcí musí přeložit vždy pro konkrétní architekturu, na které OpenCL aplikace poběží. Ze zdrojových kódu tak bylo zapotřebí odstranit všechny konstrukce, které nejsou generovaným překladačem podporovány. Z konstrukcí použitých v `pocl` sem patří podpora posixových vláken (tzv. POSIX Threads, hlavičkový soubor `pthread.h`), s nimi souvisejících atomických zámku a načítání dynamických knihoven za běhu programu pro realizaci *Just-In-Time* překladu kernelů. Vlákna jsou v runtime knihovně implementovány pro umožnění vykonávání uživatelských příkazů a požadavků na pozadí.

Všechna použití zámků tak byla nahrazena pomocí maker prázdnými příkazy a všechna místa, kde dochází k vytváření dodatečných vláken byla kompletně odstraněna. Celý kód runtime knihovny je tak vykonáván jediným hlavním vláknem. To má vliv hlavně na neblokující příkazy jako je `clEnqueueWriteBuffer`, který umožňuje provést neblokující zápis do zápisového bufferu. Standardně by bylo spuštěno nové vlákno, které tuto operace vykoná na pozadí, zatímco pokračuje provádění dalších příkazů v hostitelském programu. Při použití jen jednoho vlákna dojde k uložení příkazu do fronty příkazů, pokračování vykonávání hostitelského programu a provedení neblokujícího příkazu až později. K tomu může dojít buď implicitně (nějaký další blokující příkaz používající výsledek neblokujícího) nebo explicitně voláním funkce `clFinish`. Takové chování je stále v souladu se standardem OpenCL.

Do runtime knihovny bylo rovněž přidáno nové zařízení (viz 6.1.1) pojmenované jednoduše `codasip`. To bylo založeno na zařízení `basic`, které ze všech implementací vrstvy zařízení v `pocl` nejvíce odpovídá typickému procesoru modelovaného pomocí Codasip Studia. Nejdůležitější změnou oproti zařízení `basic` je propojení se staticky přeloženými kernely, kdy ve funkci `pocl_codasip_run` dojde k přípravě argumentů a následnému zavolání funkce zaregistrované jako vstupní bod kernelu (registrace ukázána na příkladu 7.2.3). Tento kód provádějící spuštění kernelu je v příkladu 7.3.1.

Příklad 7.3.1. Spuštění staticky přeloženého kernelu pomocí zaregistrované vstupní funkce

```
// get currently used kernel
struct _OpenCLKernel *kernel_impl = kernel->impl;
size_t local_work_size[3];
size_t global_work_size[3];

// local work group size (number of parallel work-items)
local_work_size[0] = cmd->command.run.local_x;
local_work_size[1] = cmd->command.run.local_y;
local_work_size[2] = cmd->command.run.local_z;

// global work group size (total number of work items)
global_work_size[0] = pc->num_groups[0] * local_work_size[0];
global_work_size[1] = pc->num_groups[1] * local_work_size[1];
global_work_size[2] = pc->num_groups[2] * local_work_size[2];

// run kernel
kernel_impl->call(arguments, sizes, pc->work_dim,
    local_work_size, global_work_size);
```

Kvůli statickému překladu kernelů musela být výrazně upravena i implementace funkce `clEnqueueNDRangeKernel`, která se používá v hostitelské aplikaci pro spuštění kernelu.

U klasického překladu kernelu za běhu aplikace se právě touto funkcí nastavuje velikost lokální work-group (mimo jiné), v našem případě musela být ovšem zadána už dříve atributem `reqd_work_group_size` (používání tohoto atributu je rozebráno v kapitolách 8 a 9). Do funkce `clEnqueueNDRangeKernel` tak byla přidána kontrola zda hodnota předaná této funkci odpovídá té, uvedené u samotného kernelu. V případě nesouladu je uživatel upozorněn varovnou hláškou a jsou použity hodnoty, se kterými byl kernel přeložen. V případě nespecifikování požadované velikosti (standard toto dovoluje) jsou správné hodnoty zvoleny automaticky. Standardně implementace této funkce v `pocl` obsahuje výpočet ideální velikost lokální work-group v případě jejího nezadání, dále přikazy pro překlad kernelu pomocí *Just-in-time* kompilace a generování pomocného kódu. Tyto části se vzhledem ke statickému překladu kernelů staly nepotřebnými a byly proto z implementace odstraněny.

Samotné spuštění kernelu je delegováno na implementaci vrstvy zařízení a její funkci `pocl_device_run`. To znamená, že i přesto, že se implementované spouštění staticky přeložených kernelů velmi liší od standardních způsobů přítomných v `pocl`, nemuselo ve funkci `clEnqueueNDRangeKernel` dojít k žádným změnám a veškerá funcionalita byla implementována ve funkci `pocl_codasip_run`.

7.4 Náhrada průchodu TargetAddressSpaces

Při testování bylo zjištěno, že průchod `TargetAddressSpaces`, o němž byla řeč v 6.2.1, v některých případech selhává. Problém mu činí složitější vícenásobná pole ukazatelů, struktury obsahující pole ukazatelů a různé kombinace těchto datových stuktur. To uznávají i samotní autoři `pocl`, kteří tyto problémy zmiňují ve zdrojových kódech průchodu. Při pokusech o opravu těchto chyb se ukázalo, že by musela být přepracována většina průchodu, neboť jeho současná koncepce neumožňuje některé chyby plnohodnotně opravit. Bylo proto rozhodnuto implementovat vlastní verzi tohoto průchodu, která nebude výše zmíněnými nedostatky trpět a navíc bude umožňovat širší možnosti změn adresných prostorů použitých v překládaném programu. Sem patří třeba možnost nastavovat konkrétní mapování paměťových regionů na adresné prostory architektury dynamicky při spuštění překladu (v průchodu `TargetAddressSpaces` je toto mapování napevno nastaveno v kódu), což se hodí právě pro rekonfigurovatelný překladač generovaný pomocí Codasip Studia. O této funkci bude dále řeč v sekci 10.4.2.

Příklad 7.4.1. Program z příkladu 3.2.1 po zploštění adresných prostorů

```
define void @demo(i32* %src, i32* %dst, i32 %factor) {
entry:
    %call = tail call i32 @get_global_id(i32 0)
    tail call void @barrier(i32 1)
    %arrayidx = getelementptr inbounds i32, i32* %src, i32 %call
    %0 = load i32, i32* %arrayidx, align 4
    %mul = mul nsw i32 %0, %factor
    %arrayidx3 = getelementptr inbounds i32, i32* %dst, i32 %call
    store i32 %mul, i32* %arrayidx3, align 4
    ret void
}
```

Hlavním režimem, ve kterém tento průchod pracuje, se nazývá `flatten` a provádí tzv. „zploštění“ adresných prostorů. To znamená, že dojde k upravení všech ukazatelů tak, že

začnou ukazovat do výchozího adresného prostoru č. 0. Na příkladu 7.4.1 je ukázán výsledek zploštění kódu z příkladu 3.2.1, adresný prostor 0 není v kódu narozdíl od ostatních, s vyššími čísly, nijak vyznačen.

Průchod pracuje v následujících krocích :

- **ošetření globálních proměnných**
  Postupně projde všechny globální proměnné a nastaví jim správný adresný prostor. Rovněž zkontroluje jejich inicializátory a pokud se jedná o ukazatele, upraví i je.

- **ošetření funkcí**
  Projde všechny kernely, uživatelské a builtin funkce a zkontroluje a případně upraví adresné prostory jejich argumentů. Změní i návratový typ funkce pokud se jedná o ukazatel do nesprávného adresného prostoru.

- **ošetření instrukcí `alloca`**
  Instrukce typu `alloca` alokuje místo pro proměnnou na zásobníku aktuálně prováděné funkce. Alokovaný objekt je vždy umístěn v generickém adresném prostoru (adresný prostor číslo 0), LLVM tedy nepodporuje více zásobníků. V režimu `flatten` nemá tento krok uplatnění, protože všechny objekty jsou automaticky naalokovány ve správném adresném prostoru.

- **ošetření ostatních instrukcí**
  Všechny ostatní "obyčejné"instrukce se zkontrolují a pokud mají jako operandy nebo výsledek ukazatele (i nepřímo, např. jako součást struktury), jsou tyto upraveny na ukazatele do správného adresného prostoru.

- **ošetření instrukcí přetypování**
  Zvláštní pozornost musí být věnována instrukcím provádějícím přetypování beze změny velikosti. To provádí instrukce `bitcast` (přetypování libovolného primitivního objektu na jiný typ beze změny bitů) a `addrspacecast` (přetypování ukazatele do jednoho adresného prostoru na ukazatel do jiného). Při úpravách adresných prostorů může dojít ke vzniku instrukcí `bitcast`, kde zdrojový a cílový typ nemají stejný adresný prostor nebo instrukcí `addrspacecast`, kde zdroj a cíl naopak stejný adresný prostor mají. Oba tyto případy jsou v IR nelegální a je tak nutné je detekovat a případně jednu instrukci nahradit druhou.

- **ošetření volání intrinsic funkcí**
  Tento krok se týká intrinsic funkcí pro práci s pamětí, patří sem `memcpy` (kopírování úseku paměti), `memmove` (kopírování úseku paměti, zdroj a cíl se mohou překrývat) a `memset` (nastavení úseku paměti na určitou hodnotu). V LLVM existují různé verze těchto funkcí pro práci s různými adresnými prostory. Všechna volání intrinsic funkcí je tak třeba nahradit za verze pracující se správnými ukazateli.

- **kontrola chyb**
  Nakonec dojde ke zkontrolování, zda během úpravy adresných prostorů nedošlo ke vzniku chyb resp. nevalidního IR. Hlavním problémem může být vznik instrukcí volání funkcí, kde předávané ukazatele ukazují do jiného adresného prostoru než udává prototyp volané funkce. Pokud tato situace nastane, značí to chybu ve vstupním programu a uživateli je vypsána chybová hláška. Obdobně jako v případě ošetření instrukcí `alloca` se tento krok netýká režimu `flatten`, kde díky tomu, že všechny ukazatele ukazují do shodného adresného prostoru, nemůže k nekompatibilitám dojít.

Kapitola 8

Optimalizace pro architektury se SIMD instrukcemi

Jak již bylo zmíněno, všechny work-items v rámci work-group jsou dle OpenCL standardu vykonávány paralelně. V klasickém případě (např. u grafických karet) to znamená spuštění odpovídajícího počtu procesů, resp. vláken. Jak ale dosáhnout paralelismu u vestavěných systému, kde často ani jedna z těchto technik není dostupná? Jednou z možností, popsanou v této kapitole, je využití SIMD instrukcí.

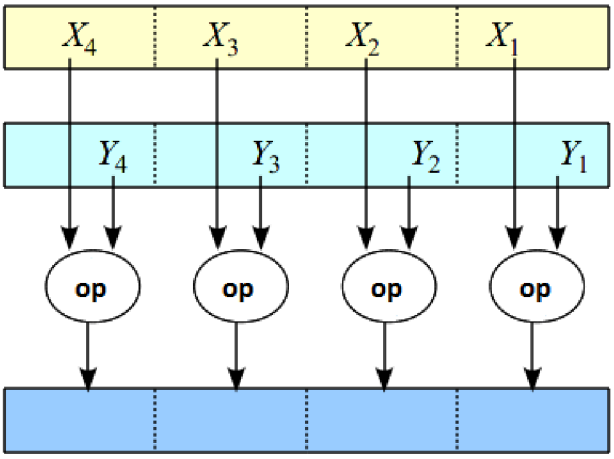
8.1 Instrukce typu SIMD

Instrukce typu SIMD (*Single Instruction*, *Multiple Data*, jedna instrukce, více dat) tvoří důležité výkonnostní rozšíření dnešních procesorů. Využívají širokých datových cest a funkčních jednotek moderních procesorů pro současné provádění jedné operace nad více datovými elementy, zvanými zabalené datové prvky (*packed data elements*). To jsou relativně krátké vektory uložené v paměti nebo registrech [5]. Klasické, skalární instrukce považují jednotlivé operandy za nedělitelný celek, skalár. Operandy SIMD instrukce jsou ovšem interpretovány jako vektor několika (zpravidla mocnina dvou) dílčích hodnot. Odtud pochází označení těchto instrukcí jako vektorové [30]. Jak taková instrukce pracuje je ukázáno na obrázku 8.1.

Tato technologie se v procesorech poprvé objevila v 70. letech, nicméně masivnějšího rozšíření se dočkala až o dvacet let později, kdy firma Intel zavedla do svých procesorů Pentium multimediální instrukční rozšíření MMX. Od té doby se používání SIMD instrukcí výrazně rozšířilo a podporují ho téměř všechny dnes používané typy procesorů [30]. Za všechny lze jmenovat instrukční rozšíření SSE a AVX použité v procesorech firem Intel a AMD nebo NEON od firmy ARM [23].

Existují SIMD instrukce pro přesuny dat, aritmetické a logické výpočty, porovnávání a všechny ostatní typy operací, prováděné klasickými skalárními instrukcemi. Ve většině případů bývají dostupné i speciální, pro vektorové instrukce specifické operace, jako je vložení prvku do vektoru nebo změna pořadí prvků uvnitř vektoru. SIMD instrukce mohou obecně pracovat nad hodnotami celočíselnými i s plovoucí nebo pevnou řádovou čárkou. Některé operace ve vektorové podobě neexistují, což se týká zejména těch, které provádí změnu řídicího toku programu, jako jsou přímé a nepřímé skoky nebo volání funkce [30].

V současné době jsou SIMD instrukce běžně podporovány i na vestavěných systémech a rovněž při návrhu procesorů v Codasip Studiu lze tyto instrukce popsat v jazyce CodAL.



Obrázek 8.1: Schéma funkce typické SIMD instrukce [30]

Vygenerovaný překladač pro takový procesor dokáže tyto instrukce automaticky využívat, ať už díky přímému zápisu těchto operací ve vstupním programu nebo pomocí automatického převodu skalárního kódu na vektorový prováděného autovektoračními průchody uvnitř optimalizátoru překladače.

Na úrovni LLVM IR jsou vektorové operace podporovány, většina instrukcí může mít jako operandy kromě skalárních typů i jejich vektory. Ukázka funkce provádějící součet dvou vektorů na úrovni IR je na příkladu 8.1.1.

Příklad 8.1.1. Funkce v LLVM IR provádějící součet dvou vektorů

```
define <4 x i32> @add(<4 x i32> %a, <4 x i32> %b) {
entry:
    %tmp1 = add <4 x i32> %a, %b
    ret <4 x i32> %tmp1
}
```

8.2 Paralelní vykonávání work-items

Jako nejvhodnější optimalizace pro architektury se SIMD instrukcemi bylo zvoleno využití těchto instrukcí pro paralelní provádění jednotlivých work-items. Bylo navrženo, že použití této optimalizace bude probíhat následujícím způsobem:

1. uživatel specifikuje velikost lokálních work-groups staticky ve zdrojovém kódu pomocí atributu `reqd_work_group_size` připojeného ke kernelu,
2. atribut je automaticky zpracován Clangem a do modulu uložen v podobě metadat,
3. během překladu kernelu je tato informace využita k úpravě kódu kernelu tak, že jednotlivé operace, z nichž se skládá, jsou prováděny vektorovými instrukcemi pracujícími paralelně nad daným počtem prvků; současně dojde k implementaci bariér,
4. opakovaným prováděním vektorizovaného kernelu se dosáhne vykonání jeho funkce pro každý work-item v globálním indexovém prostoru s paralelním vykonáním work-items v rámci jednotlivých lokálních work-groups.

Atribut `reqd_work_group_size` umí Clang zpracovat automaticky bez potřeby dodatečných úprav, protože je obsažen v OpenCL standardu. Při použití parametru příkazové řádky `-cl-kernel-arg-info` převede informace v něm obsažené do podoby metadat, která lze poté dále zpracovat.

8.2.1 Implementace v pocl

Pro realizaci bodu 3 z předchozího schématu byl překladač kernelů `pocl` rozšířen a byl do něj přidán nový průchod nazvaný `WorkItemLoops`. Ten projde všechny kernely a zkontroluje velikost jejich lokálních work-groups (velikost v jednotlivých dimenzích se dozví z metadat modulu jak je popsáno výše). V kódu kernelu identifikuje tzv. paralelní regiony, což jsou základní bloky, tvořící kód mezi bariérami. Bariéry jsou v `pocl` na úrovni IR reprezentovány voláními funkce `pocl.barrier` a lze tak identifikovat úseky kódu, které mají být provedeny před dosažením bariéry a které po něm. Pokud je velikost lokální work-group v některé dimenzi vyšší než jedna, vytvoří průchod okolo každého paralelního regionu obdobu smyčky typu `for` s počtem iterací rovným velikosti v dané dimenzi. Tímto způsobem může okolo každého paralelního regionu vytvořit až tři vzájemně vnořené smyčky jak je ukázáno na příkladu 8.2.1, nejvnitřnější smyčka odpovídá dimenzi x. Kernel v příkladu neobsahuje žádnou bariéru, uvnitř jedné sady smyček je tak umístěn celý jeho kód.

Příklad 8.2.1. Ukázka činnosti průchodu `WorkItemLoops`
Kernel se staticky definovanou velikostí lokálních work-groups v jednotlivých dimenzích (x = 4, y = 3, z = 6):

```
__kernel
__attribute__((reqd_work_group_size(4, 3, 6)))
void simple_demo(__global int *src, __global int *dst, int factor)
{
    int id = get_local_id(1);
    dst[id] = src[id] * factor;
}
```

Odpovídající kód vygenerovaný průchodem `WorkItemsLoops` znázorněný na úrovni jazyka C (průchod ve skutečnosti pracuje nad IR):

```
for (int i = 0; i < 6; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 4; k++) {
            int id = j;
            dst[id] = src[id] * factor;
        }
    }
}
```

V tomto průchodu dojde i k implementaci funkce `get_local_id`, která vrací číslo daného work-item v rámci lokální work-group v zadané dimenzi. Tato hodnota přímo odpovídá hodnotě řídicí proměnné odpovídající smyčky.

V případě, že by kernel vypadal tak, jak bylo ukázáno na příkladu 3.2.1, tedy obsahoval by bariéru, tvořily by příkazy před bariérou jiný paralelní region než příkazy po ní. Všechny work-items v lokální work-group musí v takovém případě dorazit k bariéře před tím, než budou moci pokračovat dále. Pokud by byl použit stejný postup jako výše, tedy celý kód kernelu uvnitř jedné smyčky (resp. jedné sady smyček), tak by první work-item provedl celý

kód, po něm druhý atd. To je ovšem špatně, první work-item může začít vykonávat kód po bariéře až poté, co kód před ní provedli všichni ostatní. Oba paralelní regiony tvořící v tomto případě kernel tak musí být umístěny do samostatných cyklů jak je ukázáno na příkladu 8.2.2. Tím zároveň dojde k implementaci bariéry, protože druhá smyčka nemůže začít být prováděna před tím, než první smyčka skončí.

Je zde zároveň vidět i problém, který realizace pomocí smyček přináší. Tím je existence proměnných, používaných z více paralelních regionů. V tomto jednoduchém příkladu dojde ještě před bariérou k získání identifikátoru work-itemu a za bariérou je tato hodnota použita pro indexování polí předaných argumenty. Při pouhém umístění obou příkazů do oddělených smyček by bez dodatečného ošetření došlo k tomu, že jednotlivé work-itemy si budou navzájem přepisovat lokální proměnnou, do které je tato hodnota uložena a program pak nebude fungovat správně. Je proto nutné provést tzv. uložení kontextu v prvním paralelním regionu a jeho obnovení ve druhém, jak je ukázáno na druhé části příkladu 8.2.2. Tuto dodatečnou režii je nutné vložit pro každou proměnnou, která je používána napříč paralelními regiony. Průchod tak musí nejprve všechny takové proměnné identifikovat a poté, při vytváření smyček okolo paralelních regionů, vygenerovat i odpovídající kód pro ukládání a obnovování kontextu. Na úrovni IR se toto provede pomocí vložení instrukce `alloca` alokující na zásobníku pole s počtem položek rovným počtu proměnných, které je nutné předávat mezi paralelními regiony. Následně instrukce `store` a `load` provádějí ukládání proměnných do takto vytvořeného kontextu a jejich zpětné načítání.

Příklad 8.2.2. Kernel z příkladu 3.2.1 při použití atributu `reqd_work_group_size(4,1,1)` Naivní rozdělení programu na dvě části obalené smyčkami, work-items si přepíšou hodnotu proměnné `id` (opět je pro zjednodušení použit kód v C místo LLVM IR):

```
int id;
for (int i = 0; i < 4; i++) {
    id = i;
}

for (int i = 0; i < 4; i++) {
    dst[id] = src[id] * factor;
}
```

Správné řešení s ukládáním a obnovováním kontextu:

```
int context[4];
for (int i = 0; i < 4; i++) {
    int id = i;
    context[i] = id;
}

for (int i = 0; i < 4; i++) {
    int id = context[i];
    dst[id] = src[id] * factor;
}
```

Na úrovni IR dojde k vytvoření každé smyčky vložením několika nových základních bloků. Základní blok `for.init` inicializuje řídící proměnnou smyčky, `for.body` obsahuje kód paralelního regionu, `for.inc` inkrementuje řídicí proměnnou, `for.cond` kontroluje podmínku cyklu (dosažení počtu iterací). V případě splnění podmínky se skočí zpátky na začátek bloku `for.body`, jinak na blok `for.end`, který celý cyklus uzavírá.

K paměťovým instrukcím (`load` a `store`) uvnitř paralelního regionu jsou připojena metadata typu `llvm.loop.parallel`, která značí, že jednotlivé iterace smyčky mohou být vykonávány paralelně a nejsou mezi nimi žádné závislosti. To je v souladu se standardem OpenCL, který říká, že jednotlivé work-items jsou na sobě navzájem nezávislé [14]. Informaci reprezentovanou těmito metadaty může později využít optimalizátor pro aplikování některých optimalizací, které by nebylo možné použít, kdyby mezi iteracimi alespoň teoreticky nějaká závislost existovala. Mezi takové optimalizace patří např. automatická vektorizace, viz dále.

K příkazu skoku na konci bloku `for.cond` jsou připojena i další metadata a to tzv. vektorizační nápovědy (*Vectorization Hints*). Ty obsahují informace pro vektorizátor smyček, který je součástí optimalizátoru. V tomto případě jsou tvořeny metadaty `llvm.loop.vectorize.width` obsahující počet work-items v dané dimenzi (pro vynucení určitého vektorizačního faktoru) a `llvm.loop.vectorize.enable` s booleovskou hodnotou `true` (pro vynucení vektorizace jako takové).

8.2.2 Výsledek

Po dokončení průchodu `WorkItemLoops` je spuštěn vektorizátor smyček. Ten využije informace uložené v metadatech a nejvnitřnější ze smyček vektorizuje s vektorizačním faktorem rovným počtu work-items. To znamená, že smyčku odstraní a všechny skalární instrukce v ní nahradí instrukcemi vektorovými pracujícími současně nad požadovaným počtem prvků. Při tom rovněž využije informaci, že mezi jednotlivými iteracemi nejsou žádné závislosti a v případě např. přístupu do polí přes ukazatele nevloží žádné běhové kontroly, které by jinak kód zesložitily a zpomalily.

Příklad 8.2.3. Specifikace počtu vektorových elementů atributem `reqd_work_group_size`
Kód v jazyce C:

```
__kernel
__attribute__((reqd_work_group_size(4, 1, 1)))
void simple_demo(__global int *src, __global int *dst, int factor)
{
    int i = get_global_id(0);
    dst[i] = src[i] * factor;
}
```

Odpovídající kód v LLVM IR po provedení `WorkItemLoops` a autovektorizace:

```
%0 = insertelement <4 x i32> undef, i32 %factor, 0
%1 = shufflevector <4 x i32> %0, <4 x i32> undef, <4 x i32>
   zeroinitializer
%2 = load <4 x i32>, <4 x i32>* %src, align 4
%3 = mul nsw <4 x i32> %1, %2
store <4 x i32> %3, <4 x i32>* %dst, align 4
```

Kód v jazyce symbolických instrukcí cílové architektury (Codasip uRISC SIMD):

```
VEC_INSERT_IMM V0, R3, 0, V0
SHUFFLEVECTOR_4x32 V0, V0, V0, V1
LOAD_4x32 V2, R4, 0
MUL_4x32 V2, V2, V0
STORE_4x32 V2, R5, 0
```

Později, při překladu IR na assembler v backendu překladače, jsou tyto vektorové instrukce namapovány na odpovídající SIMD instrukce dané architektury (pokud je architektura obsahuje). Tím je realizováno paralelní vykonávání work-items pomocí SIMD instrukcí. Tento princip je znázorněn na příkladu 8.2.3.

Pokud by daná architektura vektorové instrukce nepodporovala (nebo sice podporovala, ale s jinými datovými typy nebo jejich počtem), může dojít k tomu, že backend překladače bude muset kód vytvořený vektorizátorem zpátky skalarizovat (tedy zkonvertovat vektorové instrukce na posloupnost skalárních). S tím může být spojena určitá režie a výsledný kód tak může být pomalejší, než kdyby k žádné vektorizaci původně vůbec nedošlo. Uživatel tak musí při nastavování počtu paralelních work-items vzít vždy v úvahu vlastnosti architektury pro kterou překlad probíhá. Projevuje se zde tak opět problematická přenositelnost výkonu OpenCL aplikací, o které byla řeč už na začátku kapitoly 2.

Určitá úskalí plynou i z použití vektorizátoru smyček použitého pro vygenerování vektorových instrukcí. Ten totiž nedokáže vektorizovat libovolnou smyčku a pokud obsahuje určité příkazy a konstrukce, může se stát nevektorizovatelnou. To se týká hlavně komplikovanějšího větvení toku programu, kde např. všechny podmíněné příkazy musí být možné nahradit instrukcí `select`. Některé příkazy nelze vektorizovat vůbec, např. `switch`. Pokud k takové situaci dojde a vektorizace se nepovede, uživatel je o tom během překladu kernelu informován hláškou „`Failed explicitly specified loop vectorization`“. Je ovšem nutné dodat, že i v takovém případě bude program stále korektně fungovat, pouze dojde k provedení jednotlivých work-items za sebou pomocí původní nevektorizované smyčky.

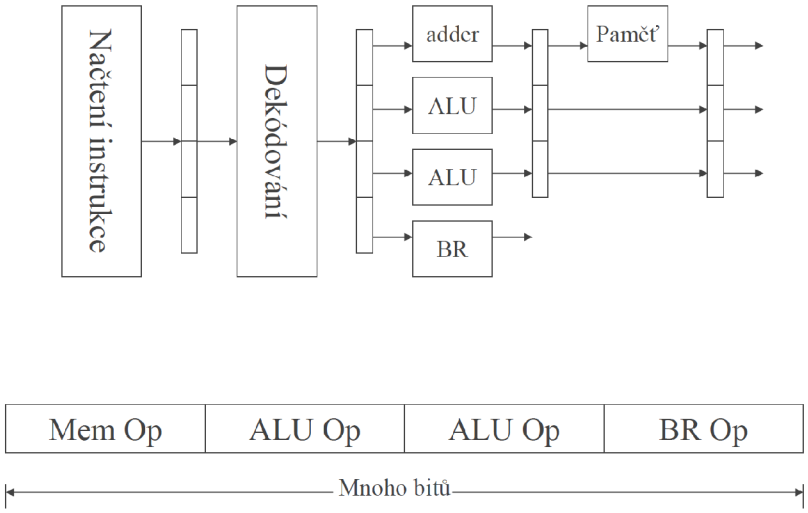
Kapitola 9

Optimalizace pro VLIW architektury

Dalším, poměrně rozšířeným typem architektur, které podporují paralelní provádění instrukcí, jsou architektury typu VLIW. Také ty lze navrhovat pomocí vývojového prostředí Codasip Studio a generovat pro ně všechny potřebné nástroje. Tato kapitola se tak zaměřuje na způsob, jakým lze optimalizovat provádění OpenCL aplikací na těchto architekturách.

9.1 Architektury typu VLIW

Architektury typu VLIW (*Very Long Instruction Word*, velmi dlouhé instrukční slovo) jsou dalším typem paralelních architektur. Narozdíl od architektur se SIMD instrukcemi, založených na datovém paralelismu, však využívají tzv. instrukční paralelismus (*instruction-level paralelism*, ILP). To znamená, že vykonávají současně několik na sobě nezávislých instrukcí využívajících nezávislé výpočetní jednotky. Tyto instrukce jsou u architektur typu VLIW „sbaleny“ do jednoho dlouhého instrukčního slova (odtud název VLIW), které je jako celek načteno, dekódováno a provedeno. Takové instrukční slovo může obsahovat např. čtyři 32-bitové instrukce a mít tak šířku 128 bitů [4].



Obrázek 9.1: Schéma funkce typické VLIW instrukce [26]

Pozice instrukcí v instrukčním slově se označují jako tzv. instrukční sloty. S každým slotem jsou spojeny výpočetní jednotky dostupné instrukci v tomto slotu a je běžné, že některé jednotky jsou dostupné jen z určitých slotů. Každý slot tak obecně podporuje jen instrukce určitého typu a ty tak nemohou být vloženy na libovolnou pozici v instrukčním slově. To je ukázáno na příkladu 9.1, instrukční slovo se skládá ze čtyř instrukcí, první slot je určen pro instrukce pro práci s pamětí, sloty 2 a 3 pro aritmetické a logické operace a poslední slot slouží pro instrukce skoku.

To, které instrukce budou sbaleny do jednoho instrukčního slova a tedy provedeny paralelně, je určeno staticky při překladu a je tedy nutná speciální podpora na straně kompilátoru. Ten musí rovněž instrukce umístit do správných slotů podle jejich funkce. V případě, že nedokáže naplánovat tolik paralelních instrukcí, aby obsadily všechny sloty, musí nevyužité sloty vyplnit prázdnými instrukcemi `NOP` (ty mohou být umístěny do libovolného slotu). To se může stát, když mezi instrukcemi existují závislosti a nelze je přeskládat tak, aby vznikly skupiny nezávislých instrukcí. Problémem jsou i skoky v programu, kdy není při překladu jasné, které instrukce se mají začít provádět po provedení skoku. Často se tak stává, že instrukční slovo obsahuje např. jen jednu nebo dvě skutečné instrukce a zbytek je vyplněn prázdnými instrukcemi. Právě nízká průměrná obsazenost slotů při překladu běžných programů a s tím spojené malé využití paralelních výpočetních jednotek, je jedním z hlavních problémů těchto architektur [4].

Mezi zástupce používaných procesorů typu VLIW patří např. Hexagon od firmy Qualcomm, který podporuje i SIMD instrukce a kombinuje tak instrukční a datový paralelismus. Současné kompilátory (gcc a LLVM/Clang) překlad pro tyto architektury podporují na úrovni implementace v backendu. Na rozdíl od architektur se SIMD instrukcemi, kde bývá podpora implementována v optimalizátoru (autovektorizace) a/nebo i přímo ve frontendu (přímý zápis vektorových operací ve zdrojovém programu), se podpora pro VLIW architekrury v těchto částech překladače nenachází a ty tak o instrukčním paralelismu na cílové architektuře nemusí vůbec vědět.

V rámci Codasip Studia lze tyto procesory efektivně navrhovat a generovat pro ně všechny nástroje včetně překladače. Typickým procesorem typu VLIW používaným v Codasipu je Codix Titanium, čtyřslotová architektura s délkou instrukčního slova 128 bitů.

9.1.1 Podpora pro OpenCL

Co se týče podpory pro překlad a spouštění OpenCL aplikací pomocí `pocl`, nebylo potřeba pro získání základní podpory žádných zvláštních úprav. Protože se podpora pro architektury VLIW řeší v backendu a `pocl` pracuje na úrovni optimalizátoru (nad IR), lze stejně jako pro jakýkoli jiný typ architektury popsaný v Codasip Studiu použít vytvořenou implementaci vrstvy zařízení založenou na zařízení `basic`. Na příkladu 9.1.1 je ukázána část kódu kernelu z příkladu 3.2.1, přeloženého pro Codix Titanium. Je dobře vidět nízká průměrná obsazenost slotů (způsobená latencemi a závislostmi mezi instrukcemi) a tím i nízká úroveň instrukčního paralelismu.

Příklad 9.1.1. Část kódu přeloženého kernelu z příkladu 3.2.1, jednotlivá instrukční slova jsou oddělena pomocí `--`.

```
$tmp3:
    nop
    nop
    st r9, [ sp + 4 ]
    st r8, [ sp + 0 ]
```

```
--
    nop
    nop
    r8 = ld [ r11 + 16 ]
    r9 = ld [ r11 + 28 ]
--
    nop
    nop
    st fp, [ sp + 12 ]
    st r10, [ sp + 8 ]
--
    fp = add r9, r8
    nop
    nop
    nop
--
    nop
    nop
    r8 = ld [ sp + 0 ]
    r10 = ld [ sp + 8 ]
--
    nop
    nop
    r9 = ld [ sp + 4 ]
    nop
--
    nop
    nop
    r8 = ld [ r8 + shl2 fp ]
    nop
--
    r9 = add r9, shl2 fp
    nop
    nop
    nop
--
    r8 = mul r8,  r10
    nop
    nop
    nop
--
    nop
    nop
    st r8, [ r9 + 0 ]
    nop
--
```

9.2 Optimalizace pro OpenCL

Kód kernelu v předchozím příkladu bude vykonáván téměř sekvenčně, jakoby by byl prováděn na obyčejné skalární architektuře. Jednotlivé instance jsou na sobě nezávislé a jejich operace by tak mohlo jít zkombinovat, problémem ovšem je, že kód popisující jednu instanci kernelu (jeden work-item) je umístěn v samostatné funkci. Ta je samostatně volána

z jiné funkce pro každý work-item v globálním indexovém prostoru a na místo tohoto volání funkci nelze inlinovat, protože oba kódy se nachází v odlišných modulech. Při překladu funkce s kernelem tak překladač neví nic o tom, že funkce bude volána opakovaně, ani že jednotlivé příkazy by šlo vykonávat paralelně nad daty různých work-items. Technik, jak tuto situaci zlepšit, určitým způsobem tyto znalosti překladači dodat a tím alespoň potenciálně vylepšit průměrnou úroveň instrukčního paralelismu, existuje několik. V následujících podkapitolách jsou představeny přístupy založené na (částečném) rozbalení smyček, replikaci work-items a softwarovém zřetězení.

9.2.1 Částečné rozbalení

Prvním a zároveň nejjednodušším způsobem, jak teoreticky zrychlit provádění OpenCL aplikací na VLIW architekturách, je částečné a nepřímé rozbalení smyčky, zajišťující provedení kódu kernelu pro každý work-item v globálním indexovém prostoru. V této smyčce dochází k přípravě argumentů kernelu a následnému volání funkce, která obsahuje jeho kód. Počet iterací této smyčky je dán celkovým počtem work-items, což je hodnota zadávaná v hostitelském programu a není tak známa v době překladu. To znamená, že tuto smyčku nelze při překladu rozbalit a to ani částečně.

Příklad 9.2.1. Rozbalený kód z příkladu 8.2.2

```
int id;
int context[4];

id = 0;
context[i] = id;
id = 1;
context[i] = id;
id = 2;
context[i] = id;
id = 3;
context[i] = id;

id = context[0];
dst[id] = src[id] * factor;
id = context[1];
dst[id] = src[id] * factor;
id = context[2];
dst[id] = src[id] * factor;
id = context[3];
dst[id] = src[id] * factor;
```

Při překladu sice neznáme celkový počet work-items, ovšem při použití implementovaného statického překladu kernelů známe velikost lokálních work-groups (zadanou atributem `reqd_work_group_size`). Protože celkový počet work-items musí být dle OpenCL standardu dělitelný beze zbytku velikostí lokálních work-groups, víme minimální možný celkový počet work-items (daný právě touto velikostí). Tím se nám zároveň naskýtá možnost, jak dosáhnout alespoň částečného rozbalení smyčky provádějící jednotlivé instance kernelu. Idea je tato: tuto smyčku rozdělíme tak, že do ní vložíme další vnořenou smyčku s pevným počtem iterací, rovným velikosti lokálních work-groups. Počet iterací hlavní smyčky pak tímto počtem vydělíme (resp. budeme dělit proměnnou obsahující za běhu aplikace celkový počet work-items zadaný uživatelem). Výpočet tak nyní nebude probíhat v jedné smyčce,

ale ve dvou a to při zachování správné sémantiky programu. Tímto jsme sice provádění work-items zdánlivě zesložitili a přidali dodatečnou režii spojenou s další smyčkou (inicializace řídicí proměnné, její testování atd.) ovšem díky jejímu konstatnímu počtu iterací je možné ji při překladu snadno rozbalit, dodatečné režie se zbavit a vytvořit mnohem delší souvislý proud instrukcí.

Postup popsaný výše ovšem narazí na problém: smyčka zajišťující provedení všech work-items je součástí generovaného pomocného kódu pro spouštění staticky přeloženého kernelu a obsahuje jen připravu argumentů kernelu (uložení hodnot do polí) a následné zavolání funkce kernelu. Při jejím částečném rozbalení bychom dostali sekvenci instrukcí, které nelze provádět paralelně (protože nelze např. zavolat funkci několikrát naráz), výsledný kód by se musel stejně provádět sekvenčně a v ničem bychom si nepomohli. Řešením je přenesení dodatečné smyčky až dovnitř samotného kernelu. To nápadně připomíná něco, s čím jsme se již setkali v minulé kapitole při optimalizaci pro architektury se SIMD instrukcemi. A skutečně, zde lze použít naprosto stejný postup: pomocí průchodu `WorkItemLoops` v kódu identifikujeme jednotlivé paralelní regiony, okolo kterých vytvoříme požadované smyčky, čímž zároveň implementujeme bariéry. Rozdíl je ovšem v tom, že následně nevytvoříme žádné vektorizační nápovědy a nenecháme vektorizátor smyčky vektorizovat. Naopak vektorizaci smyček zakážeme a nastavíme tzv. rozbalovací práh (*unroll threshold*) tak, aby při následném spuštění optimalizací došlo k rozbalení těchto smyček.

Použití průchodu `WorkItemLoops` ovšem přináší i jednu nepříjemnost. Jak již bylo popsáno v předchozí kapitole, musí zde být speciálně ošetřeno používání proměnných mezi jednotlivými paralelními regiony. To se realizuje pomocí ukládání a obnovování kontextu. V případě použití nerozbalených smyček nebo jejich vektorizace je to jediné řešení, jak zajistit zachování sémantiky programu. Ovšem v případě rozbalení smyček toto není potřeba. Každá rozbalená iterace získá vlastní sadu proměnných a ty tak lze používat napříč paralelními regiony přímo, bez nutnosti dodatečného kontextu. Při rozbalení kódu vygenerovaného pomocí `WorkItemLoops` se zde tak nachází potenciálně velké množství (dané počtem work-items a počtem proměnných, které je potřeba ukládat do kontextu) zbytečného ukládání hodnot do pomocných polí a jejich následného načítání. Tyto operace jsou pak na úrovni assembleru realizovány jako ukládání a načítání hodnot ze zásobníku (instrukce `load` a `store`), které nelze paralelizovat, prodlužuje výsledný kód a zpomaluje program. Schematická ukázka rozbalení kódu kernelu tímto způsobem je ukázána na příkladu 9.2.1.

V případě velmi jednoduchých kernelů, jako je ten z příkladu 3.2.1, dokáže většinou optimalizátor práci s kontextem automaticky odstranit, ovšem v případě složitějších kernelů je to nad jeho síly. V průchodu `WorkItemLoops` ovšem nelze vytváření kontextu potlačit, rozbalený kód by nefungoval správně. Řešením je vyhnout se mezikroku s generováním smyček a rozbalený kód generovat přímo, o čemž pojednává následující podsekce.

9.2.2 Replikace work-items

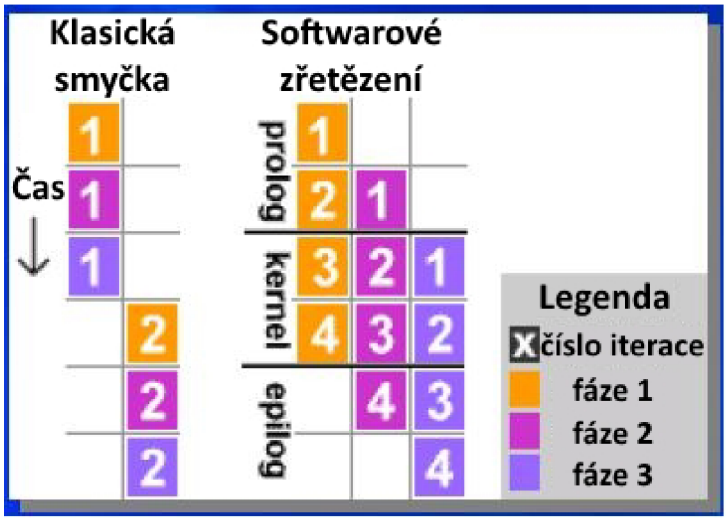
Překladač kernelů tak byl rozšířen o další průchod nazvaný `WorkItemReplication`. Nejprve dojde k identifikaci jednotlivých paralelních regionů a ke zjištění velikosti lokálních work-groups. Poté dojde k vygenerování kopií (replikaci) regionů tolikrát, aby každý work-item získal vlastní kopie všech paralelních regionů (tedy např. pokud máme čtyři paralelní work-items, dojde k vytvoření čtyř kopií každého paralelního regionu). Tyto kopie jsou následně vloženy do programu tak, že nejprve jsou vloženy všechny kopie prvního paralelního regionu, pak všechny kopie druhého atd. až nakonec program obsahuje kopie všech regionů. Tím zároveň dojde k implicitní implementaci bariér.

Všechny instrukce uvnitř paralelního regionu, které využívají výsledky spočtené v některém z regionů předchozích, jsou přemapovány tak, aby používaly hodnoty ze správné kopie tohoto regionu (tedy první kopie regionů používají svoje výsledky, druhé kopie používají svoje výsledky atd.). Tím se obejde nutnost používání pomocného kontextu, protože instrukce jsou spolu spojeny přímo. Do každého regionu je rovněž na začátek vložen tzv. prolog, který nastaví správné hodnoty lokálního a globálního identifikátoru dané instance kernelu. Výsledek průchodu `WorkItemReplication` je schematicky znázorněn na příkladu 9.2.2.

Příklad 9.2.2. Kernel z příkladu 8.2.2 rozbalený pomocí `WorkItemReplication`

```
int id0, id1, id2, id3;

id0 = 0;
id1 = 1;
id2 = 2;
id3 = 3;

dst[id0] = src[id0] * factor;
dst[id1] = src[id1] * factor;
dst[id2] = src[id2] * factor;
dst[id3] = src[id3] * factor;
```

9.2.3 Softwarové zřetězení



Obrázek 9.2: Princip (schéma) softwarového zřetězení [20]

Technika softwarového zřetězení (*Software Pipelining*), známá také jako zřetězení smyček (*Loop Pipelining*), slouží k optimalizaci smyček tím, že zřetězí instrukce z po sobě následujících iterací. Napodobuje hardwarové zřetězení, pracuje ovšem na úrovni celých instrukcí a ne jejich částí (kde dochází k prokládání jednolivých stupňů provádění instrukce). Příkazy uvnitř smyčky jsou většinou na sobě závislé (následující příkaz využivá výsledek předchozích) a tak je nelze provádět paralelně. Softwarové zřetězení smyčku transformuje tak, že ji nahradí novou smyčkou, která obsahuje na sobě nezávislé příkazy z po sobě následujících iterací. Stejně jako v případě HW zřetězení, je i zde prolog (plnění linky), kernel

(neplést s OpenCL kernelem) a epilog (vyprázdnění linky). Obecný princip softwarového zřetězení je ukázán na příkladu 9.2.

Pro lepší využití funkčních jednotek VLIW architektur na OpenCL aplikacích by šlo softwarové zřetězení využít následujícím způsobem: pomocí atributu `reqd_work_group_size` a průchodu `WorkItemLoops` by došlo k vygenerování smyček provádějících jednotlivé work-items stejně jako v případě architektur se SIMD instrukcemi. Tyto smyčky by ovšem následně nebyly vektorizovány, ale zřetězeny právě pomocí softwarového zřetězení. Tím by vznikly nové smyčky obsahující nezávislé a tedy snadno paralelizovatelné operace, které lze na VLIW architekturách seskupit do jednoho instručního slova. Tímto způsobem by se tak vlastně dosáhlo paralelního provádění kódu jednotlivých work-items stejně jako v případě architektur se SIMD instrukcemi a autovektorizace, ovšem v tomto případě by se použily VLIW instrukce a softwarové zřetězení.

Softwarové zřetězení není standardně součástí kompilačního frameworku LLVM, ovšem v rámci Codasipu byla jeho varianta známá jako *Swing Modulo Scheduling* implementována Ondrejem Glasnákem v rámci jeho bakalářské práce [6]. Bohužel v době psaní této diplomové práce nebyla tato implementace ve stavu, kdy by mohla být použita v kombinaci s překladem OpenCL kernelů pro efektivní implementaci paralelního vykonávání work-items. Využití této techniky je zde tak zmíněno pouze pro úplnost a jako možné zajímavé budoucí rozšíření.

Kapitola 10

Testování a výsledky

`Pocl` bylo tedy úspěšně integrováno do překladače automaticky generovaného v rámci Codasip Studia. V této kapitole jsou ukázány výsledky testování na sadě různě složitých aplikací a porovnání výkonu s a bez implementovaných optimalizací pro architektury se SIMD instrukcemi. Pro testování bylo použito procesorové jádro Codasip uRISC SIMD. Codasip uRISC je jednoduchá 32-bitová architektura typu RISC s téměř minimální instrukční sadou nutnou pro vytvoření překladače jazyka C. Tato architektura je často používána pro testování a jako reference. Codasip uRISC SIMD je verze této architektury rozšířená o základní SIMD instrukce (aritmetické, logické, posuvy apod.), pracující nad 128-bitovými vektory ve formě čtyř 32-bitových celých čísel.

10.1 Základní testy

Pro základní otestování funkčnosti byly použity testy dodávané spolu s `pocl`. Ty jsou rozděleny do kategorií pro testování runtime knihovny, knihovny builtin funkcí, testů pro otestování výkonnosti a regresních testů pro ověření, že do implementace nebyly znovu zaneseny dříve opravené chyby. Nejprve tedy byly provedeny testy runtime a builtin knihovny, které dopadly úspěšně, což bylo vzhledem k poměrně malým (runtime knihovna) nebo žádným (builtin knihovna) úpravám v těchto částech očekávatelné. Tímto způsobem bylo rovněž ověřena funkčnost principu překladu kernelů odděleného od překladu hostitelské aplikace. Dále bylo provedeno několik testů zaměřených na změření výkonnosti pro zjištění urychlení dosaženého pomocí implementovaného paralelního provádění instancí kernelu využitím SIMD instrukcí cílové architektury. K tomu bylo vybráno několik jednoduchých aplikací, které byly spouštěny pomocí automaticky vygenerovaného instrukčního simulátoru. Jako hostitelská aplikace sloužil jednoduchý program zajišťující vytvoření testovacích dat, provedení kernelu nad těmito daty a převzetí výsledků. Na konci simulace byl vypsán celkový počet provedených instrukcí a ten byl vynesen do grafu a tabulky uvedených dále. Měřena byla doba trvání provedení volání funkce `clEnqueueNDRangeKernel` případně `clEnqueueTask` zajišťující vykonání funkce kernelu nad všemi daty. Každý test byl vyzkoušen na testované architektuře s různým nastavením velikosti lokálních work-groups (tedy s různým požadavkem na počet paralelně vykonávaných work-items).

Konkrétnimi testovanými aplikacemi byly:

- **cldemo**
  Jednoduchý test provádějící vynásobení prvků vstupního pole zadaným faktorem a

uložení výsledku do výstupního pole. Každá instance kernelu zpracuje jeden prvek pole.

```
__kernel void cldemo(__global int *src, __global int *dst, int
    factor)
{
    int i = get_global_id(0);
    dst[i] = src[i] * factor;
}
```

- **power4**
  Vypočítá čtvrtou mocninu prvků vstupního pole a výsledek uloží to výstupního pole. Výstupní pole je ovšem pouze lokální, což v tomto případě znamená, že hostitelská aplikace nemůže přistupovat k jeho obsahu. Tento test tak slouží hlavně pro změření rychlosti provedení.

```
__kernel void power4(__global int *input, __local int *output)
{
    int i = get_global_id(0);
    int value = input[i];
    output[i] = value * value * value * value;
}
```

- **arithm**
  Vezme čtyři po sobě následující prvky ze vstupních polí (začátek odkud brát je dán globálním identifikátorem instance kernelu) a provede nad prvky základní aritmetické operace sčítání, odčítání, násobení a dělení. Výsledek uloží do čtyř po sobě následujících prvků výstupního pole.

```
__kernel void arithm(__global int *A, __global int *B, __global
    int *C)
{
    int base = 4*get_global_id(0);

    C[base+0] = A[base+0] + B[base+0];
    C[base+1] = A[base+1] - B[base+1];
    C[base+2] = A[base+2] * B[base+2];
    C[base+3] = A[base+3] / B[base+3];
}
```

- **vecadd_double**
  Sečte odpovídající si prvky dvou polí a výsledek uloží do třetího pole. Jak název testu napovídá, byl vytvořen za účelem provedení pomocí vektorových instrukcí. Test používá datový typ `double`, který je v OpenCL podporován jako rozšíření (`pocl` toto rozšíření podporuje), které je nutné zapnout pomocí direktivy pragma.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void vecadd_double(__global double *a, __global double
    *b, __global double *c, const unsigned int n)
{
    int id = get_global_id(0);
    c[id] = a[id] + b[id];
}
```

- **dot_product**
  Vypočítá skalární součin dvou čtyřprvkových vektorů zadaných pomocí vektorového datového typu `int4` a výsledek uloží do výsledného pole typu `int`. Získání hodnot ze vstupních polí a samotný výpočet jsou odděleny bariérou. To má za následek, že první část pracující s vektorovými typy zůstane nevektorizována (LLVM nedokáže vektorizovat kód, ve kterém se již vektory vyskytují) a druhá část s výpočtem, pracující se skalárními typy, vektorizována bude. Kdyby zde bariéra nebyla, k žádné vektorizaci by nemohlo dojít, protože celé tělo kernelu by tvořilo jeden paralelní region.

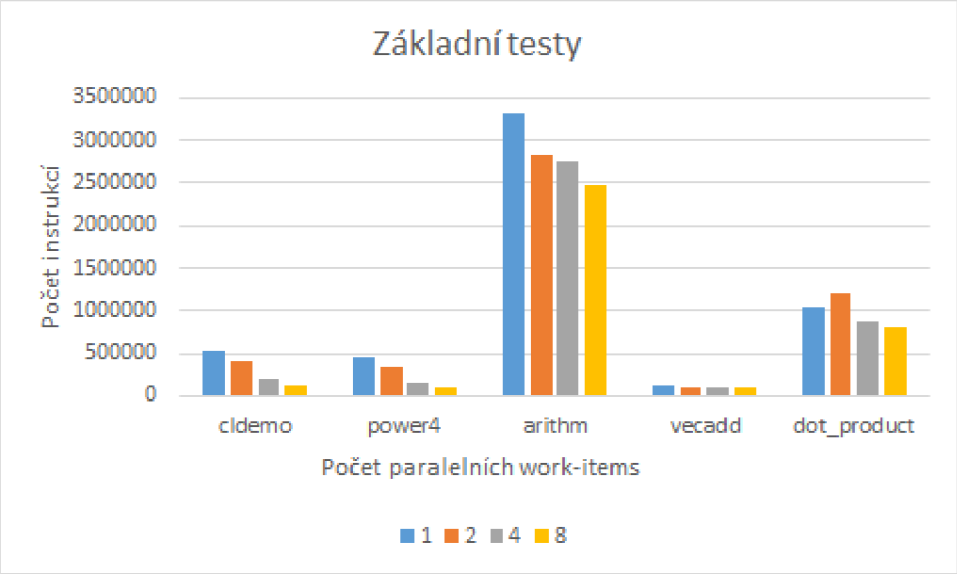
```
__kernel void dot_product (__global const int4 *a, __global
    const int4 *b, __global int *c)
{
  int id = get_global_id(0);

  int ax = a[id].x, ay = a[id].y, az = a[id].z, aw = a[id].w;
  int bx = b[id].x, by = b[id].y, bz = b[id].z, bw = b[id].w;

  barrier(CLK_LOCAL_MEM_FENCE);

  c[id] = (ax * bx) + (ay * by) + (az * bz) + (aw * bw);
}
```

V tabulce 10.1 jsou výsledky testování znázorněny pomocí tabulky a na obrázku 10.1 pomocí grafu, ukazujících počet provedených instrukcí v závislosti na různých nastaveních velikosti work-group v x-ové dimenzi. Konkrétními testovanými velikostmi byly 1, 2, 4 a 8. Toto nastavení nesmí mít vliv na funkčnost aplikace, pouze na rychlost jejího provedení, což se ve všech testovaných případech potvrdilo. Vzhledem k jednoduchosti kernelů musely být spouštěny nad velkým množstvím dat pro získání relevantních výsledků, typicky tedy pole obsahující vstupní data měly 10000 prvků.

test/počet WI	**1**	**2**	**4**	**8**	**zrychlení**
cldemo	514367	404317	194265	125463	4,10x
power4	463785	328733	156181	107379	4,32x
arithm	3315827	2818453	2752136	2486870	1,33x
vecadd_double	113779	103729	100278	98675	1,15x
dot_product	1034415	1214366	886813	808639	1,28x

Tabulka 10.1: Počet provedených instrukcí u jednotlivých základních testů v závislosti na nastavení atributu `reqd_workgroup_group_size` a maximální dosažené zrychlení

Nastavení velikosti work-group na 1 znamená, že žádné instance kernelu nejsou prováděny paralelně, hodnoty naměřené při tomto nastavení tedy byly brány jako referenční a ostatní výsledky byly vůči nim porovnávány. Z výsledků je vidět, že již při nastavení velikosti work-groups na 2 může dojít k urychlení provádění kernelu. Zde sice dojde k vektorizaci na úrovni IR a vytvoření vektorů o dvou prvcích, k použití SIMD instrukcí v backendu překladače ale nedojde, protože cílová architektura podporuje jen vektory o čtyřech prvcích. Dosažené zrychlení tak vyplývá z částečného rozbalení smyčky zajišťující provedení všech work-items. Může zde ovšem vzniknout i určitá režie spojená s vektorizací na úrovni IR a následnou skalarizací v backendu a tak v případě testu `dot_product` došlo k určitému zpomalení.



Obrázek 10.1: Počet provedených instrukcí u jednotlivých základních testů

Při požadavku na paralelní provádění čtyř, respektive osmi work-items již dochází k výraznému zrychelní běhu a to až 4,32x v případě testu `power4` a osmi paralelních work-items. Při těchto nastaveních již překladač SIMD instrukce použije, požadavek na provádění čtyřech paralelních operací nad celými 32-bitovými čísly přesně odpovídá instrukcím cílové architektury. V případě osmi dojde k rozdělení na dvě operace po čtyřech (tedy neprovádí se osm paralelních výpočtů, což na dané architektuře není možné).

Je vidět, že k největšímu urychlení dochází v případě, že se podaří vektorizovat celý kernel, tedy všechny paralelní regiony, ze kterých se skládá. To se týká testů `cldemo` a `power4`. V případě testu `dot_product`, který je tvořen dvěma regiony, lze vektorizovat jen jeden a výsledné urychlení proto není tak výrazné. Nicméně i zde došlo k nárůstu výkonu až o 28% a je možné, že při nastavení většího velikosti work-groups by došlo k urychlení ještě výraznějšímu.

Výjimkou je test `vecadd`, který využívá datový typ `double`, který není na architektuře Codasip uRISC SIMD nativně podporován a to ve skalární ani vektorové podobě. Operace nad tímto typem se emulují pomocí volání funkcí z knihovny `compiler-rt` a požadavek na provádění paralelních výpočtů nemá na výsledný výkon téměř vliv, protože tato knihovna vektorové výpočty nepodporuje. Toto chování bylo nicméně očekávané a test slouží hlavně pro demonstraci funkčnosti i kernelů pracujících s nativně nepodporovanými datovými typy.

10.2 Komplexní testy

Dále byla implementace ověřována na složitějších testech (kernelech a hostitelských aplikacích), které již více odpovídají aplikacím používaným v praxi. Bylo zvoleno pět programů, z nichž některé jsou dodávány spolu s `pocl` a některé jsou ze sady benchmarků dodavaných firmou AMD. Následuje jejich výčet spolu s krátkým popisem, s důvodu složitosti a délky není již uváděn kód daného kernelu.

- **Game of Life**
  Známá hra Life (život), ve které se pomocí evoluce celulárního automatu simuluje

chování živých buněk. V této implementaci uživatel zvolí velikost desky, na které bude hra probíhat a počet iterací algoritmu. Aplikace desku zaplní náhodnými hodnotami 1/0 podle toho, zda se na daném políčku desky nachází buňka. Poté je opakovaně spouštěn algoritmus, který na základě aktuálního stavu desky vypočítá stav nový. Po provedení zadaného počtu iterací je výsledek zobrazen uživateli. Jako OpenCL kernel je zde realizován algoritmus výpočtu nového stavu desky na základě stavu předchozího.

- **Mersenne Twister**
  Nejpoužívanější pseudonáhodný generátor čísel. Délka jeho periody je tzv. Mersennovo prvočíslo, tzn. je ve tvaru $M_p = 2^p - 1$, kde p je prvočíslo. V této implementace se použije Mersenne Twister pro vygenerování pole náhodných hodnot, které se pak použije k aproximaci hodnoty π metodou Monte Carlo. Pole vygenerovaných hodnot se chápe jako dvourozměrná matice (čtverec), která obsahuje čtvrtinovou výseč kruhu. Pomocí Monte Carlo určíme plochu této části kruhu a z ní a vzorce $P = \pi * r^2$ hodnotu π. Výpočet je rozdělen do dvou kernelů a několika pomocných funkcí. První kernel generuje pole náhodných hodnot a druhý určí kolik z těchto hodnot se nachází uvnitř výseče kruhu.

- **Matrix Transpose**
  Transpozice matice neboli záměna řádků a sloupců. Uživatel zadá rozměry matice, která je následně vytvořena a naplněna náhodnými celočíselnými hodnotami. Celá matice je posléze transponována a vypsána uživateli. Transpozici provádí OpenCL kernel, který kromě vstupního a výstupního pole (ty jsou označeny ___global), pracuje ještě s pomocným polem označeným ___local, obsahujícím jednu „poddlaždici“ a jehož obsah je viditelný všem paralelně vykonávaným work-items. Každá instance kernelu tak nejprve do tohoto pole zapíše hodnoty, které transponovala, poté následuje bariéra pro synchronizace a následně všechny paralelní instance toto pole čtou pro získání výsledných hodnot a jejich uložení do výstupního pole.

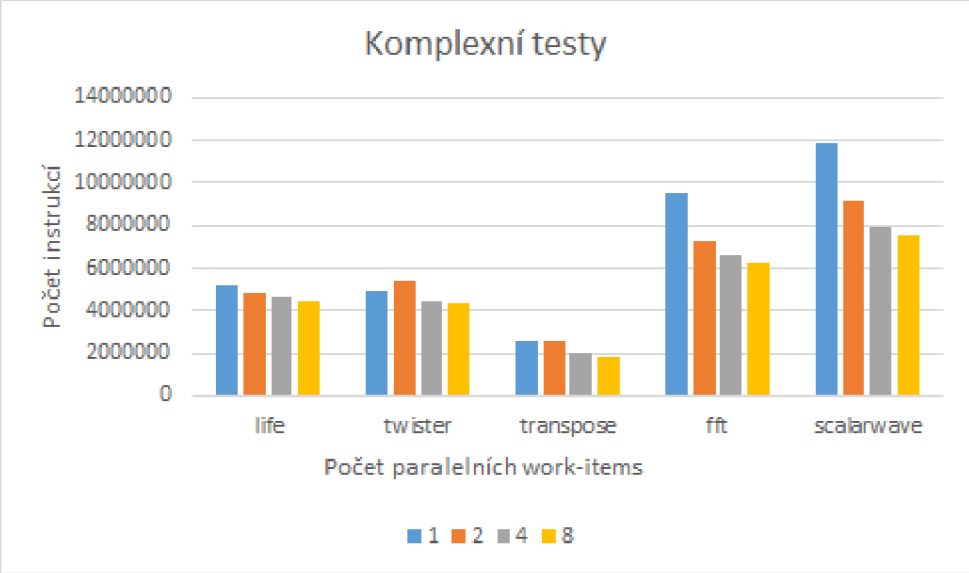
- **Fast Fourier Transform**
  Rychlá Fourierova transformace (FFT) je algoritmus, který počítá diskrétní Fourierovu transformaci (DFT) se složitostí pouze $O(n * log(n))$ oproti standardnímu $O(n^2)$. Realizuje převod signálu z časové do frekvenční oblasti a často se používá např. u různých filtrů, které je snadnější realizovat právě ve frekvenční oblasti. Tento test používá FFT v kombinaci s filtrem typu horní propusť pro zaostření šedotónového obrázku, který je načten z externího souboru. V rámci OpenCL je zde implementováno celkem šest kernelů. Jeden realizuje zmíněný filtr typu horní propusť a další provádějí jednotlivé fáze výpočtu FFT.

- **Scalar Wave Evolution**
  Tento test provádí výpočet evoluce skalární vlnové funkce v čase. Skalární vlnová funkce je zadána parciální diferenciální rovnicí $\frac{\partial^2 u}{\partial t^2} = \nabla^2 u$ a úkolem aplikace je vypočítat evoluci této rovnice v čase při dodržení Dirichletových hraničních podmínek. Počet kroků zadává uživatel (standardně 4) a na konci dojde k vypsání výsledných hodnot v koncovém čase. Kernel slouží pro výpočet jednoho kroku evoluce.

Výsledky testování jsou znázorněny v tabulce 10.2 a grafu 10.2. I tentokrát se potvrdila funkčnost implementace a implementovaných optimalizací. V těchto složitějších testech již nedochází k tak výraznému urychlení, což je způsobeno jejich problematičtější vektorizací

test/počet WI	1	2	4	8	zrychlení
life	5200682	4837168	4618455	4431054	1,17x
twister	4917266	5411494	4423038	4324192	1,14x
transpose	2594698	2520564	2038691	1853356	1,40x
fft	9538137	7307303	6638052	6191885	1,54x
scalarwave	11922671	9154932	7897766	7576646	1,57x

Tabulka 10.2: Počet provedených instrukcí u jednotlivých komplexních testů v závislosti na nastavení atributu `reqd_workgroup_group_size` a maximální dosažené zrychlení



Obrázek 10.2: Počet provedených instrukcí u jednotlivých komplexních testů

(již se nejedná o tzv. „triviálně paralelizovatelné“ problémy), i přesto však došlo ke zrychlení téměř u všech testů, které např. u testu `scalarwave` dosahuje až 57%.

10.3 Testování optimalizací pro VLIW architektury

Stejné testy, na kterých bylo testováno urychlení dosažené optimalizacemi pro architektury se SIMD instrukcemi, byly použity i pro otestování implementovaných optimalizací pro VLIW architektury. V tomto případě bylo ovšem měřeno zlepšení ve využití paralelních výpočetních jednotek, tedy průměrné obsazenosti instrukčních slotů. Slot se považuje za obsazený, pokud obsahuje nějakou „užitečnou“ instrukci, tedy jakoukoli kromě instrukce `NOP`.

Testována byla metoda replikace work-items, protože ta je hlavní optimalizací implementovanou pro tyto architektury. Zkoumán byl vliv na funkci reprezentující výpočet jedné instance kernelu, protože právě pouze tato funkce je ovlivněna průchodem provádějícím replikaci. V případě, že daný test používá více kernelů, byly uvažovány všechny dohromady. Jako testovací architektura byl použit Codix Titanium, o kterém byla řeč už v kapitole 9. Výsledky jsou shrnuty v tabulce 10.3.

Z naměřených výsledků je vidět, že ve všech testovaných případech došlo k určitému zlepšení ve využití paralelních jednotek (až cca 15%), s rostoucí velikostí work-groups se

test/počet WI	1	2	4	8	16	zlepšení
cldemo	26,09%	26,79%	28,95%	29,31%	32,91%	6,82%
power4	21,30%	23,44%	25,00%	24,66%	27,50%	6,20%
arithm	30,26%	35,53%	36,05%	36,99%	37,62%	7,35%
vecadd_double	27,68%	30,00%	34,91%	38,56%	41,21%	13,53%
dot_product	33,09%	44,39%	40,78%	48,30%	33,35%	15,21%
life	40,35%	43,57%	43,32%	43,46%	43,50%	3,22%
twister	37,02%	39,50%	39,56%	40,17%	40,97%	3,95%
transpose	20,45%	20,34%	19,24%	18,93%	21,20%	0,75%
fft	34,36%	37,13%	38,17%	38,88%	39,47%	5,11%
scalarwave	33,81%	35,70%	36,53%	36,64%	36,78%	2,97%

Tabulka 10.3: Průměrná obsazenost slotů při použití replikace work-items a maximální dosažené zlepšení vůči velikosti 1 (odpovídá nepoužití průchodu)

téměř vždy zvyšovala i průměrná obsazenost slotů. Lepších výsledků se obecně dosahovalo u jednodušších testů, což je dáno tím, že v jejich případě dokázal backend překladače lépe zkombinovat operace z jednotlivých work-items, protože ty se v kódu nacházely blíže sebe. Při dalším zvyšování počtu paralelních work-items by jistě docházelo k dalšímu zlepšování, nicméně vzhledem k faktu, že celkový počet work-items musí být počtem těch paralelních beze zbytku dělitelný, nelze toto zvyšování u reálných aplikací provádět do nekonečna.

10.4 Zhodnocení a návrh optimalizací pro zvýšení výkonu

Z výsledků provedených testů vyplývá, že implementované optimalizace pro architektury se SIMD instrukcemi skutečně ve většině případů vedou k vyššímu výkonu dané aplikace. V případě jednoduchých testů, které jsou triviálně paralelizovatelné se snadno dosáhne urychlení v násobcích původního výkonu. V případě složitejších testů, které lze např. vektorizovat jen z části nebo dokonce vůbec dochází i přesto k urychlení v řádu až desítek procent. Mezi nejčastější důvodu, proč vektorizace selže patří větvení programu, které nelze nahradit sekvencí příkazů `select` a používání nevektorizovatelných datových typů (vektorové typy OpenCL). Tyto nedostatky by mohly být vyřešeny v budoucích verzích LLVM.

Replikace work-items, jako optimalizace pro VLIW architektury, rovněž skutečně vede k lepšímu kódu na těchto architekturách, i když v případě složitějších kernelů dojde typicky ke zlepšení jen v řádu jednotek procent.

Zajímavou věci, která se projevila během testování (u testu `dot_product`), je možnost využití bariér pro rozdělení nevektorizovatelného kernelu na více částí, z nichž některé se pak mohou vektorizovatelnými stát. Dalším zajímavým poznatkem je, že implementované optimalizace ve většině případů vedou ke zrychlení i když se kernel nepodaří vektorizovat. To se zdá být způsobeno tím, že smyčka vytvařená okolo kernelu (resp. okolo jeho jednotlivých paralelních regionů) má konstatní a malý počet iterací, takže dojde k jejímu rozbalení a částečnému snížení režie spojené se spouštěním jednotlivých instancí kernelu.

Rovněž je vidět, že na velikost výsledného urychlení má velký vliv nastavení počtu paralelně prováděných work-items. Nejlepších výsledků se dosáhne, když daný počet odpovídá tomu fyzicky podporovanému na cílové architektuře nebo je jeho násobkem. Programátor tak pro dosažení optimálních výsledků musí tyto znalosti o architektuře mít a při programování v OpenCL je zohlednit. V případě přenosu programu mezi různými architekturami

je pro zachování ideálního výkonu potřeba nastavení paralelismu odpovídajícím způsobem upravit.

Mezi další možné budoucí optimalizace implementace patří automatický výběr metody pro realizaci work-group funkce na základě vlastností cílové architektury a podpora pro architektury podporující paměťové regiony OpenCL nejen v softwaru (jak je běžné), ale i přímo v hardwaru. Těmto optimalizacím a zajimavé optimalizační technice zvané automatická paketizace se věnují následující podsekce.

10.4.1 Automatická detekce typu architektury

V současné implementaci uživatel sám zadává způsob jakým mají být jednotlivé work-items paralelizovány pomocí argumentu překladového skriptu (výchozí je `WorkItemLoops`). V předchozí práci autora [30] byl popsán způsob pro dodávání informací o cílové architektuře vektorizátorům v LLVM. Ve zkratce lze uvést, že optimalizátor za běhu načte z externího textového souboru určité informace, např. velikost registrů a seznam nativně podporovaných operací na dané architektuře. Tento soubor musel být původně psán ručně, nicméně později bylo implementováno jeho automatické vytváření generátorem backendu (program `backendgen`) překladače, který je součástí sady nástrojů Codasip Studia. Ukázka obsahu tohoto souboru je na příkladu 10.4.1.

Příklad 10.4.1. Informace o cílové architektuře pro vektorizátory

```
processor codasip_urisc.ia
widest_scalar_registers 32:32
widest_vector_registers 16:128
legal_types i32 v4i32

add Other:-1 i1:-1 i8:-1 i16:-1 i32:1 i64:-1 i128:-1 f16:-1 f32:-1
   f64:-1 f80:-1 f128:-1 ppcf128:-1 v2i1:-1 v4i1:-1 v8i1:-1 v16i1:-1
    v2i8:-1 v4i8:-1 v8i8:-1 v16i8:-1 v32i8:-1 v1i16:-1 v2i16:-1
   v4i16:-1 v8i16:-1 v16i16:-1 v1i32:-1 v2i32:-1 v4i32:1 v8i32:-1
   v16i32:-1 v1i64:-1 v2i64:-1 v4i64:-1 v8i64:-1 v16i64:-1 v2f16:-1
...
```

Jak je vidět, jsou zde již přítomny informace, podle kterých lze určit, že daná architektura podporuje SIMD instrukce – přítomnost vektorových registrů, legálního vektorového typu (těch může být i víc) a rovněž podporované určité vektorové operace. V případě umožnění přístupu skriptu překládajícího kernely k těmto informacím lze automaticky určit, zda by měla být použita metoda `WorkItemLoops` a automatická vektorizace.

V případě VLIW architektur už situace není tak přímočará. Jelikož vektorizátory nepotřebují vědět, jestli je daná architektura VLIW, není v souboru tato informace uvedena. To nelze ani nijak vyvodit z podporovaných datových typů a instrukcí. Pro automatickou volbu metody `WorkItemReplication` (případně jednoduššího rozbalení smyček) by tak bylo nutné informace v souboru rozšířit a přidat např. příznak indikující, zda se jedná o architekturu typu VLIW.

Pokud architektura podporuje VLIW i SIMD instrukce zároveň, je možné zvolit libovolný algoritmus, ovšem lze očekávat, že vyššího výkonu se dosáhne přidáním datového paralelismu (SIMD instrukcí) k implicitně přítomnému instrukčnímu paralelismu (VLIW instrukcím).

10.4.2 Mapování paměťových regionů

Dalším možným rozšířením implementace by mohla být speciální podpora pro architektury, které podporují přímo v hardwaru obdobu OpenCL paměťových regionů. Na takových architekturách tak existuje přímé mapování mezi privátním, lokálním a konstatním regionem a oddělenými fyzickými paměťmi, případně fyzickými adresnými prostory (které mohou být ve výsledku mapovány i do různých částí jediné paměti). Takový typ architektury sice není mezi vestavěnými systémy rozšířený, nicméně je možné jej pomocí Codasip Studia navrhnout a implementovat.

Průchod pro modifikaci adresných prostorů na úrovni IR, popsaný v kapitole 7.4, sice primárně provádí jejich zploštění (tedy odstranění), dokáže ovšem i umisťovat určité datové typy do určitých prostorů, což už poměrně běžné je (např. dvě paměti, jedna obsahující 32-bitová slova a druhá 128-bitová slova). Konkrétní mapování se zadává parametrem příkazové řádky ve formátu `typ:prostor`, kde `typ` je libovolný datový typ LLVM IR a `prostor` je číslo adresného prostoru, do kterého mají být hodnoty daného typu umisťovány. Jednotlivé dvojice jsou odděleny znakem `-`, takže mapování může vypadat např. takto: `i16:1-v4i32:2`. Typy, které nejsou uvedeny, se považují za patřící do prostoru 0.

Toto mapování a celý průchod lze přirozeným způsobem rozšířit tak, aby prováděl i přemapování vstupních adresných prostorů, reprezentujících paměťové regiony v kernelech, na výstupní, reprezentující fyzické/logické paměti cílové architektury. Konkrétní mapování by bylo zadáváno stejným způsobem jako v případě mapování typů s tím, že `typ` by byl nahrazen číslem vstupního adresného prostoru a mapování by tedy mohlo vypadat např. takto: `1:0-2:1-3:3`.

10.4.3 Automatická paketizace

Další velmi zajímavou technikou pro zvýšení výkonu je automatická paketizace kódu, popsaná v článku [13]. Tato technika, zvaná také „vektorizace celých funkcí“ (*Whole Function Vectorization*, WFV), je speciální verzí automatické vektorizace. Narozdíl od té standardně implementované v LLVM, která efektivně pracuje pouze nad smyčkami, dokáže vektorizovat celé funkce a poradí si i s větvením programu, což je v současnosti velká slabina vektorizátoru smyček.

WFV je založena na konverzi řídícího toku programu na datový tok, což znamená, že dojde k odstranění všech podmínek, nepodmíněnému vykonání všech příkazů a následnému výběru správných výsledků operací `select` nebo maskováním. Tímto způsobem je transformován celý obsah funkce, který je následně vektorizován nahrazením všech skalárních příkazů jejich vektorovými variantami (s výjimkou těch z principu nevektorizovatelných jako je volání funkce).

Ve zmíněném článku byl tento algoritmus implementován do LLVM frameworku a dokonce testován na vektorizaci OpenCL kernelů. Mohl by tak tedy teoreticky být začlěnen do `pocl` a umožnit vektorizaci v současnosti nevektorizovatelných kernelů. Tím by mohlo dojít k výraznému nárůstu výkonu na SIMD architekturách zvláště v případě složitějších kernelů s komplikovaným větvením, kdy standardní autovektorizace smyček selhává.

Kapitola 11

Závěr

V rámci této práce byl popsán framework pro vytváření programů určených pro spouštění na heterogenních platformách zvaný OpenCL se zaměřením na části důležité pro tuto práci. Byly popsány a zhodnoceny existující implementace OpenCL a na základě toho bylo vybráno `pocl`, které nabízí kombinaci dobré funkčnosti a ideálních licenčních podmínek. `Pocl` bylo následně integrováno do překladače LLVM, který je součástí Codasip Studia, což je vývojové prostředí sloužící pro návrh procesorů s aplikačně specifickou instrukční sadou.

Překladač kernelů `pocl` a hostitelská runtime knihovna byly upraveny pro potřeby vestavěných systémů a byl implementován plně statický překlad a linkování kernelů. Rovněž byla implementována vlastní verze průchodu `TargetAddressSpaces` realizující mapování paměťových regionů OpenCL na fyzické adresné prostory cílového zařízení.

Pro architektury se SIMD instrukcemi byl implementován speciální optimalizační průchod `WorkItemLoops`, který jednotlivé paralelní regiony uvnitř každého kernelu obalí smyčkami, které jsou následně vektorizovány. Tím je dosaženo paralelního provádění instancí kernelů a zároveň jsou efektivně implementovány všechny bariéry.

Podobný optimalizační průchod, nazvaný `WorkItemReplication`, byl implementován i pro VLIW architektury. V tomto případě dojde k replikaci jednotlivých paralelních regionů, díky čemuž může dojít ke zkombinování instrukcí z různých instancí kernelu a optimálnějšímu využití výpočetních jednotek těchto architektur. Rovněž dojde k implicitní implementaci bariér.

Implementace byla otestována a demonstrována na sadě testovacích programů a bylo zjištěno, že implementované optimalizace pro architektury se SIMD instrukcemi skutečně vedou ke zrychlení spouštěných kernelů a optimalizace pro VLIW architektury k vyššímu využití paralelních výpočetních jednotek. Dosažené zrychlení se u netriviálních aplikací typicky pohybovalo v rozmezi 20% - 40% a zlepšení ve využití jednotek v rozmezí 5% - 10%. Nakonec byly navrhnuty možné budoucí optimalizace, které by mohly usnadnit používání implementace a vést k vyššímu výkonu překládaných aplikací.

Hlavním přínosem této práce je tak vytvoření způsobu, jakým lze spouštět OpenCL aplikace na vestavěných zařízeních a obecně na systémech, které nepodporují překlad kernelů za běhu programu. Dále také návrh a implementace speciálních optimalizací zmíněných výše, které umožňují paralelní provádění kernelů i jinak než pomocí obvyklých procesů a vláken.

Literatura

[1] AMD: APP SDK - A Complete Development Platform [online]. 2015-10-14, [cit. 2015-11-30].
URL http://developer.amd.com/tools-and-sdks/opencl-zone

[2] Brochard, R.: FreeOCL [online]. 2011-10-24, [cit. 2015-12-01].
URL http://www.zuzuf.net/FreeOCL/

[3] Codasip: Codasip Studio User Guide, 2015-09-03, version 1.0.0.

[4] Fisher, A. J.; Faraboschi, P.; Young, C.: *Embedded computing: a VLIW approach to architecture, compilers and tools*. Boston: Morgan Kaufmann Publishers, 2005, ISBN 1558607668.

[5] Gerber, R.: *The software optimization cookbook: high-performance recipes for IA-32 platforms*. Hillsboro: Intel Press, druhé vydání, 2006, ISBN 09-764-8321-1.

[6] Glasnák, O.: *Software pipelining v překladači LLVM*. bakalářská práce, FIT VUT v Brně, Brno, 2014.

[7] IBM: OpenCL Development Kit for Linux on Power [online]. 2015-12-27, [cit. 2016-01-06].
URL
https://www.ibm.com/developerworks/community/alphaworks/tech/opencl

[8] Intel Corporation: Intel SDK for OpenCL Applications [online]. 2015-09-23, [cit. 2015-11-30].
URL https://software.intel.com/en-us/intel-opencl

[9] Intel Corporation: Beignet [online]. 2015-11-29, [cit. 2015-11-30].
URL https://01.org/beignet

[10] ISO Committee: Defintion of ISO C99 [online]. 2007-09-07, [cit. 2016-01-08].
URL http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

[11] Jaaskelainen, P.: pocl - Portable Computing Language [online]. 2015-11-18, [cit. 2015-11-30].
URL http://pocl.sourceforge.net

[12] Jaaskelainen, P.; de La Lama, C.; Schnetter, E.; aj.: Pocl - A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, ročník 43, č. 5, 2015: s. 752–785, ISSN 08857458, doi:10.1007/s10766-014-0320-y.

[13] Karrenberg, R.; Hack, S.: Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, 2011, ISBN 9781612843568, s. 141–150.

[14] Khronos Group: Khronos OpenCL Registry [online]. 2016-02-18, [cit. 2016-04-10].
URL https://www.khronos.org/registry/cl/

[15] Khronos Group: The open standard for parallel programming of heterogeneous systems [online]. 2016-05-09, [cit. 2016-05-16].
URL https://www.khronos.org/opencl/

[16] Khronos Group: Connecting software to silicon [online]. 2016-05-10, [cit. 2016-05-16].
URL https://www.khronos.org/

[17] Lattner, C.: The Architecture of Open Source Applications [online]. 2012-07-07, [cit. 2015-12-20].
URL http://www.aosabook.org/en/llvm.html

[18] Lattner, C.: The LLVM Compiler Infrastructure [online]. 2015-12-20, [cit. 2016-01-06].
URL http://www.llvm.org/

[19] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Březen 2004.

[20] Lemay, D.: Into the Itanium, Part 2 - Software Pipelining and Register Stacking [online]. 2004-12-27, [cit. 2016-03-15].
URL http://www.devhardware.com/c/a/Computer-Processors/Into-the-Itanium-Part-2/3/

[21] LLVM Project: LLVM Language Reference Manual [online]. 2015-12-10, [cit. 2015-12-17].
URL http://llvm.org/docs/LangRef.html

[22] Marlow, S.: The Glasgow Haskell Compiler [online]. 2015-12-08, [cit. 2016-02-13].
URL https://www.haskell.org/ghc/

[23] Muchnick, S. S.: *Advanced compiler design and implementation*. San Francisco: Morgan Kaufmann, 1997, ISBN 15-586-0320-4.

[24] Munshi, A.: *OpenCL programming guide*. Upper Saddle River: Addison-Wesley, 2012, ISBN 978-0-321-74964-2.

[25] NVIDIA Corporation: CUDA Parallel Computing Platform [online]. 2016-03-01, [cit. 2016-05-16].
URL http://www.nvidia.com/object/cuda_home_new.html

[26] Petr Hanáček: VLIW - Very Long Instruction Word. Slajdy k předmětu Paralelní a distribuované algoritmy, FIT VUT v Brně, 2008.

[27] QuantAlea GmbH: LLVM and NVVM [online]. 2012, [cit. 2015-12-19].
URL https://www.quantalea.net/media/_doc/2/7/manual/index.html?LLVMandNVVM.html

[28] Steck, D.: Clover - OpenCL 1.1 Software Implementation [online]. 2011-08-22, [cit. 2015-12-01].
URL http://people.freedesktop.org/steckdenis/clover/

[29] Stellard, T.: libclc [online]. 2013-05-11, [cit. 2015-12-01].
URL http://libclc.llvm.org/

[30] Šnobl, P.: *Podpora SIMD instrukcí v překladači LLVM*. bakalářská práce, FIT VUT v Brně, Brno, 2014.

Přílohy

Seznam příloh

Příloha A

Obsah CD

Na přiloženém CD se nacházejí tyto adresáře a soubory:

- /src/scripts - skripty pro překlad kernelů
- /src/pocl - zdrojové kódy pocl 0.12 (šířené pod licencí MIT)
- /lib - překladač kernelů ve formě dynamické knihovny `LLVMPocl.so`
- /pocl-rt/lib - runtime knihovna `libOpenCL.a`, knihovna bultin funkcí `kernel-lib.bc` a knihovna pomocných funkcí `libpoclu.a` přeložené pro Codix uRISC SIMD
- /pocl-rt/include - hlavičkové soubory nutné pro překlad kernelů
- /examples - testy a příklady použité v této práci
- /doc - technická zpráva ve formátu pdf včetně návodu na použití
- /tex - zdrojové soubory technické zprávy

Příloha B

Návod

Předpokládá se nainstalované a správně nakonfigurované Codasip Studio s platnou licencí a model procesoru Codasip uRISC SIMD.

Postup překladu OpenCL aplikací na architektuře Codasip uRISC SIMD:

1. pomocí Codasip Studia vygenerovat všechny nástroje, povolit překlad knihoven `compiler-rt` a `newlib` a exportovat vytvořené nástroje do zvoleného adresáře,
2. adresář `/pocl-rt` z přiloženého CD zkopírovat do adresáře s exportovaným toolchainem (je nutné zachovat jméno adresáře),
3. knihovnu `LLVMPocl.so` z adresáře `/lib` a skripty z adresáře `/src/scripts` zkopírovat do adresáře `/bin` v exportovaném toolchainu,
4. přeložit kernely aplikace příkazem

```
~/export/urisc/bin/urisc-opencl-kernel-compiler cldemo.cl
```

5. přeložit hostitelskou aplikaci, přilinkovat přeložené kernely a runtime knihovnu a vytvořit spustitelný soubor příkazem

```
~/export/urisc/bin/urisc-clang cldemo.c cldemo.bc -o cldemo.xexe
    -O3 -lsim -lOpenCL
```

6. odsimulovat pomocí

```
~/export/urisc/bin/urisc-isimulator.ia -r cldemo.xexe
```