HARDWARE ACCELERATION USING FUNCTIONAL LANGUAGES

Andrea Hodaňová
Master's Degree Programme (3), FIT BUT
E-mail: xhodan06@stud.fit.vutbr.cz

Supervised by: Otto Fučík
E-mail: fucik@fit.vutbr.cz

Abstract: The aim of the project is to research how the functional paradigm can be used for the acceleration of data-parallel tasks in FPGAs. A library for data-parallel computations on GPUs called Accelerate is extended to be used for FPGAs. Accelerate uses a domain-specific language embedded into Haskell for the frontend. The original library translates higher-order functions into skeletons of CUDA code. In this project, we use the same frontend, but instead of translating the Accelerate EDSL into CUDA, we produce VHDL. In this paper, we summarize the topic and present a simple example of a data-parallel circuit generation.

Keywords: functional programming, Haskell, high-level synthesis, FPGA, VHDL, Accelerate, GPU, CUDA, Repa, data-parallel computing

1 ÚVOD

Funcionální jazyky mají celou řadu vlastností zajímavých z hlediska návrhu hardwaru. Tento projekt zkoumá jejich potenciální využití pro vysokoúrovňový popis hardwaru, konkrétně v oblasti datově paralelních výpočtů. Tradiční HDL jazyky jako VHDL nebo Verilog sice poskytují obecně několik úrovní abstrakce, nejvyšší – behaviorální – úroveň je však špatně syntetizovatelná, proto většina návrhů standardně probíhá na úrovni RTL. Se zvyšující se mírou integrace prvků na čipu vzrůstá potřeba opravdu vysokoúrovňového abstraktního popisu. Zachycuje se algoritmus [1]. Stejný kód je pak možné jak přeložit do instrukcí procesoru, tak syntetizovat do hardwaru. Komerční aplikace se zaměřují zejména na jazyk C a jeho deriváty. Funcionální jazyky se sice s jazykem C nemohou měřit v rozšířenosti a popularitě mezi programátory, nabízí však jiné výhody, jako výrazně snazší verifikovatelnost, intuitivní zachycení paralelismu nebo kratší kód na vyšší úrovni abstrakce.

2 VÝHODNÉ VLASTNOSTI FUNKCIONÁLNÍCH JAZYKŮ

Základním stavebním prvkem funkcionálních programů jsou funkce v matematickém smyslu slova (levá strana funkce se rovná pravé straně). Jejich výstup závisí pouze na vstupu, nikoli na čase a na pořadí vykonávání, podobně jako je tomu u kombinačních obvodů a napojování signálů. Tato vlastnost umožňuje zachycovat paralelismus vlastní prováděným operacím.

Funkcionální jazyky běžně umožňují funkcím přijímat jiné funkce jako parametry. Konkrétně jazyk Haskell, na který se specializuje tato práce, nabízí ve standardní knihovně několik funkcí vyššího řádu [2]. Funkce `map` přijímá unární funkci a seznam prvků a vrací jiný seznam, jehož obsah vznikl aplikací předané funkce na jednotlivé prvky seznamu. Funkcí `map` tak lze reprezentovat například násobení vektoru koeficientem a jiné datově paralelní (SIMD) operace. Pro aplikaci funkcí s více parametry existují obdobné funkce vyššího řádu, např. `zip`, `zipWith`, `zipWith3` apod.). Další rodinou funkcí vyššího řádu jsou redukce `fold` a `scan`, které stromovitě redukují seznam do jednoho výsledku (respektive seznamu mezivýsledků).

Intuitivním příkladem skládání funkcí vyššího řádu je funkce realizující skalární součin dvou vektorů (`dotproduct`), v [3] pod názvem `dotp`. Vypadá následovně:

```
dotproduct :: [Float] -> [Float] -> Float
dotproduct xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

První řádek definuje datový typ funkce. Operace `dotproduct` přijímá jako parametry dva seznamy čísel `Float` a jako výsledek vrací jedno číslo `Float`. Dva vektory `xs` a `ys` jsou napřed postupně po prvcích vynásobeny funkcí `zipWith` s operací (*). Prvky výsledného vektoru jsou zleva jeden po druhém naakumulovány k počáteční hodnotě 0 funkcí `foldl` (levý `fold`) s operací (+). Tento zápis je výrazně kratší a čitelnější než zápis v C/C++ i než odpovídající návrh komponenty ve VHDL.

Syntéza hardwaru z podobně kompaktního kódu přináší značnou úsporu v objemu kódu i zmenšení prostoru pro chyby. Funkčnost programu je zároveň možné před syntézou vyzkoušet překladem z funkcionálního jazyka do spustitelného kódu nebo spuštěním programu v interpretu.

V poslední době vzniklo několik jazyků pro funkcionální návrh hardwaru. Řada z nich je přímo vestavěná do Haskellu, který se pro svou čistě funkcionální povahu, silný typový systém a dostupnost stabilních programátorských nástrojů jeví jako vhodný prostředek. Tyto vestavěné jazyky se však často orientují spíše na popis na úrovni propojování komponent a jejich hlavním zaměřením je verifikace obvodů, kde nahrazují obtížněji verifikovatelné VHDL nebo Verilog. Tato práce se však zaměřuje na vysokoúrovňovou syntézu a hledá pro datově paralelní výpočty alternativu k jazykům C nebo C++, které dnes používají komerční nástroje typu Catapult C.

3 KNIHOVNA ACCELERATE A JEJÍ BACKEND PRO PROSTŘEDÍ CUDA

Tento projekt navazuje na práci týmu z University of New South Wales v Austrálii [3], kde vznikl doménově specifický jazyk Accelerate (vestavěný do Haskellu) a k němu knihovna s backendem pro práci s prostředím NVIDIA CUDA. Knihovna se orientuje na kolektivní operace s pravidelnými vícedimenzionálními poli dat. Je navržená tak, aby bylo možné použít stejný jazyk a frontend s několika různými backendy. Momentálně je implementován backend pro prostředí CUDA a experimentální verze backendů pro OpenCL a pro knihovnu Repa [4] (pro rychlé paralelní operace na vícejádrových procesorech), vše je dostupné v repozitáři hackage[1].

Projekt představený v tomto článku přidává další backend, a sice výstup do VHDL pro čipy FPGA.

Backend pro CUDA negeneruje statický CUDA kód, ale funguje jako dynamické prostředí pro spouštění akcelerace na GPU. Accelerate nabízí několik základních kolektivních operací (`map`, `zipWith`, `fold`, `scanl`, `scanr`, `permute`, `backpermute`), které překládá za použití předpřipravených šablon kernelů CUDA.

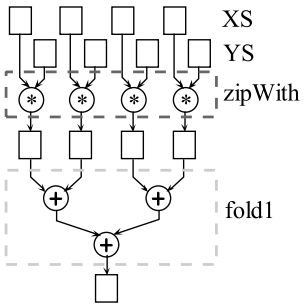
Dále nabízí datové typy pro rozlišení běžných dat uložených v paměti procesoru a dat, která budou nahrána do paměti grafické karty. Datové typy pro akcelerovaná pole využívají polí typu *delayed*, která jsou definována nikoli jako úsek konkrétních dat v paměti, ale prostřednictvím své indexovací funkce. Jednotlivé operace nad takovým polem je pak možné chápat jako kompozici fukncí, které provádí transformace nad těmito daty, aniž by musely mezi jednotlivými transformacemi vznikat pole mezivýsledků. V konktextu návrhu hardwaru odpovídá tento typ fúze operací řetězení kombinačních jednotek. Zatímco však vlákna GPU provádí takovou posloupnost funkcí sekvenčně (paralelně nad větším množstvím dat), nabízí se v hardwaru možnost zřetězeného zpracování (*pipeline*), a tedy dalšího zrychlení.

[1] http://hackage.haskell.org/package/accelerate, http://hackage.haskell.org/package/repa

4 NAVRŽENÉ ROZŠÍŘENÍ KNIHOVNY ACCELERATE

Narozdíl od backendu pro grafické karty, za něhož řeší rozdělení operací mezi vlákna GPU přímo prostředí CUDA, se backend pro FPGA musí zabývat také plánováním, alokací a přiřazením, podobně jako ostatní nástroje pro vysokoúrovňovou syntézu, např. Catapult C.

```
dotproduct :: Acc (Vector Float)
           -> Acc (Vector Float)
           -> Acc (Scalar Float)
dotproduct xs ys = fold1 (+) (zipWith (*) xs ys)
```



Obrázek 1: Kód v jazyce Accelerate a odpovídající schéma obvodu pro `dotproduct`. V příkladu je pro redukci použita funkce `fold1`, která vyžaduje asociativní operaci a nepotřebuje počáteční prvek.

Datový typ `Acc` značí akcelerované pole, které vznikne z běžného typu `Vector` funkcí `use`. Ta je v kontextu převodu do GPU chápána jako pokyn pro přenesení pole dat z hostitelského systému do paměti grafické karty. V případě generování kódu pro hardware označuje pole dat, které bude v FPGA uloženo v paměti. Frontend knihovny je podrobně popsán v [3].

5 ZÁVĚR

Implementace převodu abstraktního kódu v jazyce Haskell do syntetizovatelného VHDL jako rozšíření již existující knihovny s backendem pro GPU má řadu výhod: Stejný kód ve vestavěném jazyce Accelerate je možné vyzkoušet nejen v FPGA, ale i v grafické kartě nebo na vícejádrovém procesoru. Jednotlivé výsledky je možné mezi sebou porovnávat z hlediska výkonu, rychlosti, spotřeby apod. Frontend knihovny Accelerate specifikuje několik základních kolektivních operací, z nichž je možné poskládat složitější transformace nad vícerozměrnými bloky dat, například různé DSP operace (konvoluce), transformace obrazu (detekce hran) nebo řešení celulárních automatů. Díky existenci ustáleného frontendu se tento projekt může plně zaměřit na výstup ve VHDL, jeho čitelnost a modularitu, a také latenci a vhodné použití zdrojů na čipu.

REFERENCE

[1] COUSSY, Philippe a Adam MORAWIEC. High-level synthesis: from algorithm to digital circuit. Springer, 2008, 297 s. ISBN 978-1-4020-8587-1.

[2] O'SULLIVAN, Bryan. *Real world Haskell*. 1. vyd. Sebastopol, CA, USA: O'Reilly, 2008, 670 s. ISBN 978-0-596-51498-3.

[3] CHAKRAVARTY, Manuel M. T., Gabriele KELLER, Sean LEE, Trevor L. MCDONELL a Vinod GROVER. Accelerating Haskell array codes with multicore GPUs. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. New York, USA: ACM, 2011, s. 3-14.

[4] KELLER, Gabriele, Manuel M.T. CHAKRAVARTY, Roman LESHCHINSKIY, Simon PEYTON JONES a Ben LIPPMEIER. Regular, shape-polymorphic, parallel arrays in Haskell. *ACM SIGPLAN Notices*. 2010-09-27, roč. 45, č. 9. ISSN 03621340.