

Optimalizace CUDA kernelů pomocí fúzí

aneb kterak zrychlit (izolovaně) nezrychlitelné

Jiří Filipovič, Jan Fousek, Matúš Madzin

5. prosince 2013

Kernely omezené propustností paměti

GPU vykoná desítky operací na přenos jednoho slova z/do globální paměti

- pokud náš kernel provádí aritmetických operací méně, je rychlost jeho provádění omezena rychlostí paměti
- data se zpravidla nevyskytují ve vyrovnávací paměti od předchozího běhu kernelu (ty jsou příliš malé)

Důvod vzniku kernelů omezených rychlostí paměti

- řešíme paměťově omezený problém (např. $\mathbf{a} + \mathbf{b}$)
 - z principu nelze ovlivnit
- píšeme rozumně znovupoužitelný kód (např. $\mathbf{a} + \mathbf{b} + \mathbf{c}$ voláním kernelů pro součet dvou vektorů)
 - omezení snížíme použitím kernelu sčítajícího tři vektory, mezivýsledky zůstanou uloženy ve sdílené paměti nebo v registrech

Fúze kernelů

Máme-li sekvenci volání paměťově omezených kernelů, které si vzájemně předávají data

- mohou být zpravidla nahrazeny komplexnějšími kernely, které vykazují lepší paměťovou lokalitu (některá data se předávají pomocí rychlejších on-chip pamětí)

„Ruční“ vývoj komplexních kernelů je dosti nákladný

- mnoho kombinací, omezená znovupoužitelnost
- od určitého okamžiku nemusí být provádění více výpočtů v jednom kernelu výhodné, nalezení optima složité

Jak z toho ven?

- implementujeme jednoduché, znovupoužitelné kernely
 - každý kernel volá rutiny pro načtení vstupu, výpočet a uložení výsledku
- v závislosti na předávání dat tyto kernely spojujeme ve větší celky
 - lze provádět automaticky

Horní hranice zrychlení

U paměťově omezených kernelů zrychlení zhruba odpovídá procentu ušetřených přenosů paměti.

Např. $\mathbf{a} + \mathbf{b} + \mathbf{c}$

- 2 kernely – čtení 4 vektorů, uložení 2
- fúze – čtení 3 vektorů, uložení 1
- fúze odstraní 1/3 přenosů, tzn. $1.5\times$ zrychlení
- zrychlení může být v praxi větší (overhead spouštění kernelu, maskování latence)

Omezení zrychlení

Jakmile přestane být kernel paměťově omezený

- další redukce paměťových přenosů nezvyšuje rychlost výpočtu (není-li problém latence paměti)
- rychlost výpočtu však může být vyšší (overhead spouštění kernelu, maskování latence, optimalizace sekvenčního kódu)

Konzumace on-chip paměti

- ve fúzi může být vyšší (mezidata používané dalšími fúzovanými funkcemi)
- vyšší nároky na on-chip paměti omezují dosažitelný stupeň paralelismu, může snižovat výkon

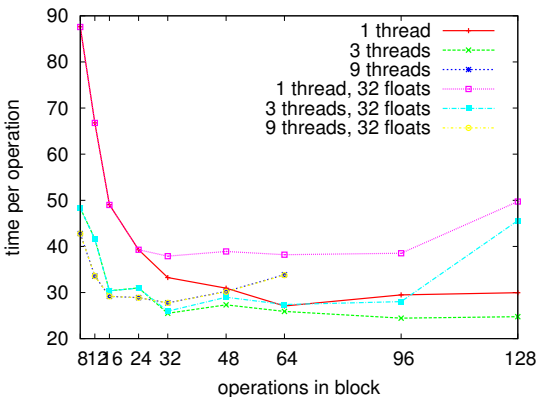
Omezení zrychlení

Rozdílné nároky na paralelismus

- každá instance funkce může běžet ve více vláknech (dosažení vhodného poměru počtu vláken ke spotřebované on-chip paměti)
- pro každou fúzovanou funkci ale může být efektivní jiný počet vláken
- při fúzi funkcí běžících v rozdílném počtu vláken tak musí být přepočítávány koordináty vláken a některá vlákna jsou část výpočtu nevyužita
- vynutíme-li naopak stejné počty vláken pro všechny fúzované funkce, obecně nepoužíváme nejefektivnější dostupné implementace

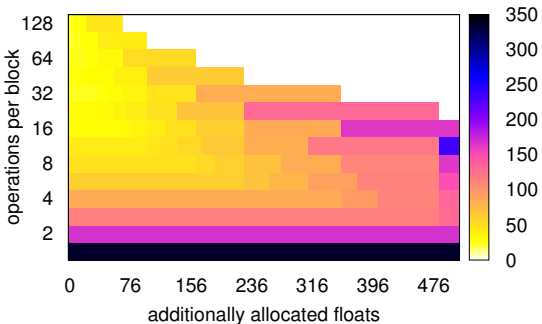
Výkon rozdílně paralelních implementací

Sčítání 3×3 matic pomocí 1, 3 a 9 threadů, a s 32 floaty alokovanými navíc ke každé funkci.



Vztah výkonu a zvýšené alokace paměti

Sčítání 3×3 matic pomocí 3 threadů.



Automatizace fúzí

Obecné kernely se fúzují špatně

- libovolné mapování vláken a bloků na data
 - abychom mohli nechat data v on-chip paměti, musí být totožné
 - jak je analyzovat? či upravit?
- globální bariéra mezi běhy kernelů
 - nelze implementovat uprostřed běhu kernelu
 - lze ji nahradit lokální? či vypustit?

Můžeme fúzovat kernely provádějící konkrétní funkci vyššího řádu.

Map

Definice map

- Necht' $map(f, L) = [f(e_1), \dots, f(e_n)]$.
- Kde $L = [e_1, e_2, \dots, e_n]$ je seznam n elementů e_1, \dots, e_n .
- Funkce f
 - může být provedena více vlákný
 - musí být proveditelná v jednom bloku

Důsledky pro fúzovatelnost

- spustíme stejný počet instancí fúzovaných funkcí na blok
 - pak můžeme nahradit globální bariéru lokální (i -tá instance zpracovává i -tý element)
- pokud zároveň předáváme data přes sdílenou paměť
 - není problém s mapováním vláken na data

Reduce

Definice reduce

- $reduce(\oplus, L) = e_1 \oplus e_2 \oplus \dots \oplus e_n$, kde $L = [e_1, e_2, \dots, e_n]$
- \oplus je asociativní

Pro získání výsledku musíme zpracovat celý L

- neobejdeme se bez globální bariéry

Důsledky pro fúzovatelnost

- Díky asociativitě \oplus můžeme ale fúzovat parciální redukce
 - redukuje vše, co máme lokálně dispozici
 - redukce je dokončena až po doběhnutí redukujícího kernelu, její výsledek tedy nemůžeme použít v kernelu, který ji provádí

Mapování na data triviální

- vstup redukce musí být ve sdílené paměti či registrech, pak lze redukovat vše v bloku

Vyjádření BLAS funkcí jako map a reduce

BLAS (Basic Linear Algebra Subprograms)

- standard pro knihovny implementující základní funkce z lineární algebry na maticích a vektorech
- hojně využívaný (nejen) ve vědeckém software
- obecně velmi dobře optimalizované implementace
- jakékoliv zrychlení zajímavé

Výkon některých funkcí omezený rychlostí paměti

- BLAS-1 (vektor-vektor), lze vyjádřit pomocí map a reduce
- BLAS-2 (matice-matice), část lze vyjádřit pomocí vnořených map a reduce

Sekvence volání BLAS-1 a BLAS-2 lze zrychlit fúzema

- není možno v izolovaných BLAS funkcích
- obecné zrychlení BLAS, navíc obecnou metodou

Příklad vyjádření BLAS-2 funkce

Násobení matice vektorem $y = Ax$ vyjádříme jako

$$y = \text{map}(\text{reduce}(+, \text{map}(\cdot, A_i, x)), A)$$

kde $A = [A_1, \dots, A_m]$, $A_i = [a_{i,1}, \dots, a_{i,n}]$ a $x = [x_1, \dots, x_n]$.

Pozn. proč ne $y = \text{map}(\text{dotprod}(A_i, x), A)$?

Schéma kompilátoru

Co kompilátor potřebuje?

- knihovnu elementárních funkcí (v CUDA)
- script definující sekvenci jejich volání

Co musí udělat?

- analýzu kódu
 - správně rozpoznat, jaké má k dispozici funkce a jak je použít
 - přečíst vysokoúrovňový kód a na jeho základě vybudovat DAGy volání a předávání parametrů
- optimalizace
 - prohledat a prořezat prostor fúzí nad každým DAGem
 - pro každou fúzi prohledat a prořezat prostor jejich implementací
 - na základě predikce výkonu zkombinovat implementace fúzí a nefúzovaných kernelů pro maximalizaci výkonu
- vygenerovat CUDA kód s fúzema

GEMVER

$$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$$

$$x \leftarrow \beta B^T y + z$$

$$w \leftarrow \alpha Bx$$

GEMVER

```
TILE32x32 A, B, C;
subvector32 u1, u2, v1, v2, w, x, y, z;
globalscalar alpha, beta;

input A, u1, u2, v1, v2, y, z, alpha, beta;

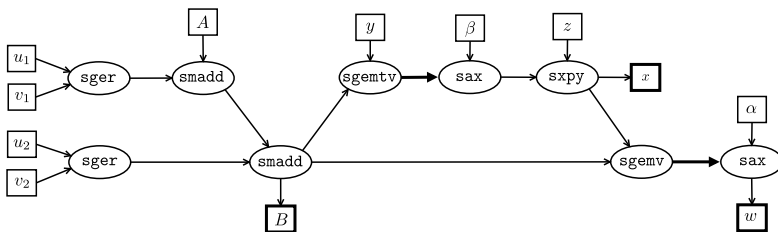
C = sger(u1, v1);
B = smadd(A, C);
C = sger(u2, v2);
B = smadd(B, C);

x = sgemtv(B, y);
x = sax(beta, x);
x = sxpy(x, z);

w = sgemv(B, x);
w = sax(alpha, w);

return B, x, w;
```

GEMVER



Prostor optimalizací

Optimalizace mají mnoho stupňů volnosti

- fúzovatelné podgrafy DAGu tvoří *fúze*
- linearizace fúzí určuje pořadí volání funkcí ve fúzi a tím i spotřebu paměti
- implementace elementárních funkcí ve fúzi a paralelismus (spolu s předchozím) *implementace fúze*
- *kombinace implementací fúzí* určuje, které fúze použijeme

Fúze – prořezávání prostoru

Fúze musí tvořit souvislou komponentu

- jinak nešetříme datové přenosy

Velikost fúzí lze omezit

- čím větší fúze, tím méně ušetříme paměťových přenosů přidáním další funkce
- s velikostí fúze roste šance, že nepůjde implementovat efektivně
- omezení velikosti na k funkcí snižuje složitost nejhoršího případu na $\sum_{i=0}^k \binom{|V|}{i}$, popř. $\mathcal{O}(k|V|)$
- výrazně zjednoduší prohledávání prostoru implementací fúzí

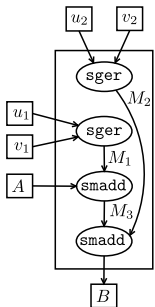
Linearizace fúze

Každá fúze obsahuje podgraf DAGu, pro implementaci je třeba určit pořadí spouštění funkcí

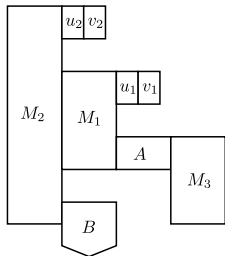
- to je důležité, jelikož ovlivňuje množství alokované on-chip paměti
- máme až $\mathcal{O}(|V|!)$ linearizací, pro každou z nich existuje exponenciálně mnoho možností jak alokovat paměť

Linearizace fúze

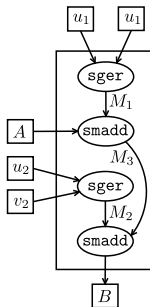
Linearization



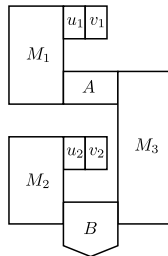
Allocation



Linearization



Allocation



Linearizace fúze – prořezávání prostoru

Vybereme linearizaci s nejnižším dolním odhadem alokované paměti

- jedná se tedy o aproximaci, nepostihneme fragmentaci paměti
- pro vybranou linearizaci spočítáme precizně alokaci paměti pomocí branch-and-bound algoritmu v $\mathcal{O}(m^n)$ kde m je celková velikost alokované paměti a n počet funkcí

V praxi

- linearizací bývá výrazně méně, než určuje horní mez
- před branch-and-bound algoritmem najdeme počáteční řešení polynomiálním greedy algoritmem, ten často najde optimum
- i ve zlomyslném případě je řešení dosažitelné díky omezení velikosti fúzí

Výběr implementací elementárních funkcí

Dále je třeba vybrat konkrétní implementace elementárních funkcí

- projdeme všechny přiřazení konkrétních implementací elementárních funkcí: $\prod_{i=0}^n \#f_i$, kde $\#f_i$ je počet implementací i -té funkce
- pro každé přiřazení odhadneme výkon pomocí predikce výkonu

Výběr kombinací fúzí

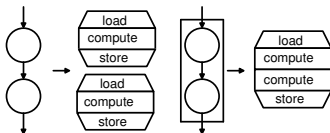
Máme-li seznam implementací fúzí s odhadem jejich výkonu, je ještě zapotřebí určit, které budou použity

- ze všech kernelů (implementace fúzí a samostatné elementární funkce) vybíráme ty, které dohromady tvoří celý DAG a maximalizujeme odhadnutý výkon
- problém pokrytí množin
- řešeno pomocí lineárního programování

Generování kódu

Elementární funkce mají předepsanou strukturu „load-compute-store”

- fúze realizujeme serializací funkcí a odstranění přebytečných load a store rutin
- a také dogenerováním dalšího kódu, jako je alokace proměnných, cykly, výpočet indexů vláken či omezení paralelismu



Generování kódu

Každá funkce pracuje s různými vstupy a výstupy v různé paralelní granularitě

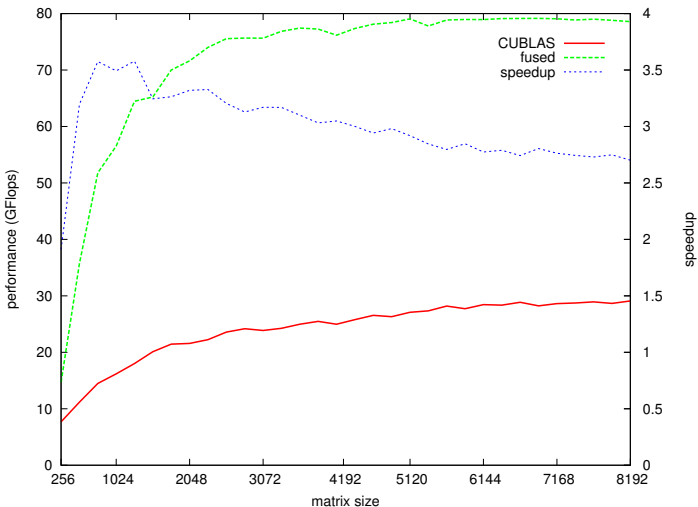
- knihovna obsahuje vedle kódu elementárních funkcí také metadata
- umožňují silnou typovou kontrolu
- generátor kódu dokáže spojovat funkce s rozdílnými nároky na paralelismus přepočítáním koordinát threadu a omezením paralelismu pro některé fúzované funkce

Zrychlení oproti CUBLAS

Název	Operace	Zrychlení	Tag
AXPYDOT	$z \leftarrow w - \alpha v$ $r \leftarrow z^T u$	1.97×	FS
GEMVER	$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$ $x \leftarrow \beta B^T y + z$ $w \leftarrow \alpha Bx$	2.63×	FS
GESUMMV	$y \leftarrow \alpha Ax + \beta Bx$	1×	(F)
VADD	$x \leftarrow w + y + z$	2.35×	FS
WAXPBY	$w \leftarrow \alpha x + \beta y$	2.54×	FS

Tabulka: Tagy: F=zlepšitelné fúzí, S=zlepšitelné specializací kernelu, E=používá efektivnější $A^T x$.

Škálování GEMVER



Prostor implementací

Název implementace	Celkem implementací	Nejlepší nalezena	Výkon první	Výkon nejhorší
AXPYDOT	19	2.	77.1 %	35 %
GEMVER	2443	72.	96.8 %	30 %
GESUMMV	164	17.	99.8 %	90 %
VADD	34	17.	94.7 %	52 %
WAXPBY	104	7.	94.6 %	32 %