Mapované funkce, fúze kernelů

Jiří Filipovič

podzim 2011

Definice map

map($f, L_1, .., L_n$) aplikuje n-ární funkci $f(l_1, .., l_n)$ na seznam(y) vstupních elementů $L_1, .., L_n$, $n \geq 1$.

- f nazýváme *mapovaná funkce*
- jednotlivé instance f jsou nezávislé – triviální paralelizace
- avšak potenciálně problematická implementace f

Naše aplikace

- per-element výpočty v metodě konečných prvků

Paralelní granularita

Paralelní granularita

- samotný map nám pro velké problémy poskytuje dostatečný stupeň paralelismu
- co když jsou ale nároky na on-chip paměťové zdroje funkce f příliš vysoké?
  - paměťové nároky f lze snížit paralelizací
- je-li f paralelní, může ji provádět blok threadů
  - to ale vyžaduje paralelizovatelnost na dostatečný počet vláken

Středně-zrnný paralelismus

Problematické mapované funkce

- nelze efektivně řešit v jednom threadu (příliš mnoho on-chip zdrojů)
- nelze efektivně řešit v bloku (příliš málo paralelizovatelné)

Středně-zrnná implementace

- navrhli jsme zavést úroveň paralelismu mezi thready a bloky
- f je paralelizována na více threadů, ale více instancí f je počítáno v bloku
  - jednoduché pravidlo
  - složitější implementace

Přístup do sdílené paměti

Přístup do sdílené paměti je v případě středně-zrnných implementací složitější

- broadcast v rámci funkce není broadcast v rámci half-warpu (problém odpadá u Fermi)
- konflikty bank mohou vznikat mezi funkcemi
- složité vyčíslení stupně konfliktu

Zamezení konfliktů bank mezi funkcemi

- padding mezi datovými elementy
- prokládané uložení datových elementů

Násobení malých čtvercových matic

1 thread na 1 element výsledné matice, více matic v bloku

- všechny thready zpracovávající řádek C čtou stejnou hodnotu z A
- všechny thready zpracovávající sloupec C čtou stejnou hodnotu z B
- pro matice velikosti nedělitelné 16 požadavek na více broadcastů
  - problém pro c.c. 1.x
- konflikty mezi instancemi funkcí nenastávají

Násobení malých čtvercových matic

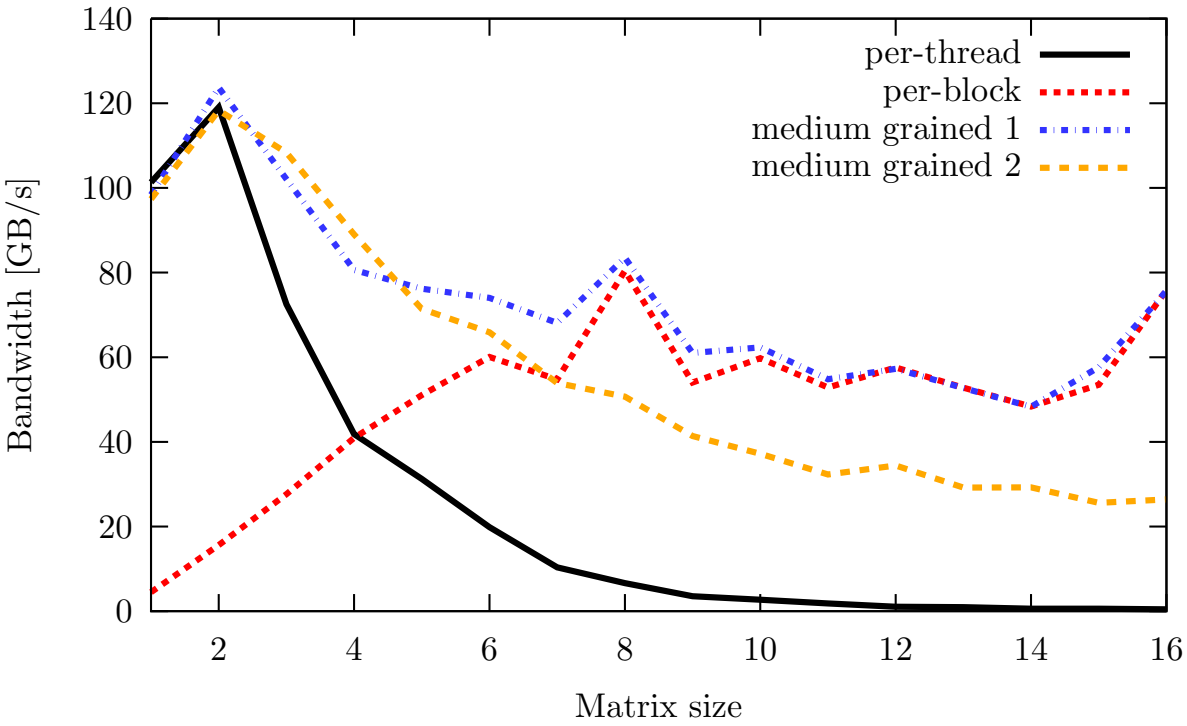
1 thread na řádek výsledné matice, více matic v bloku

- thready zpracovávající paralelně sloupec C čtou sloupec z A
- thready zpracovávající paralelně sloupec C čtou stejnou hodnotu z B
- při paddingu A a C u matic sudé velikosti žádné konflikty bank uvnitř funkcí
- konflikty mezi funkcemi u c.c. 1.x
  - násobné broadcasty z B

1 thread na sloupec výsledné matice, více matic v bloku

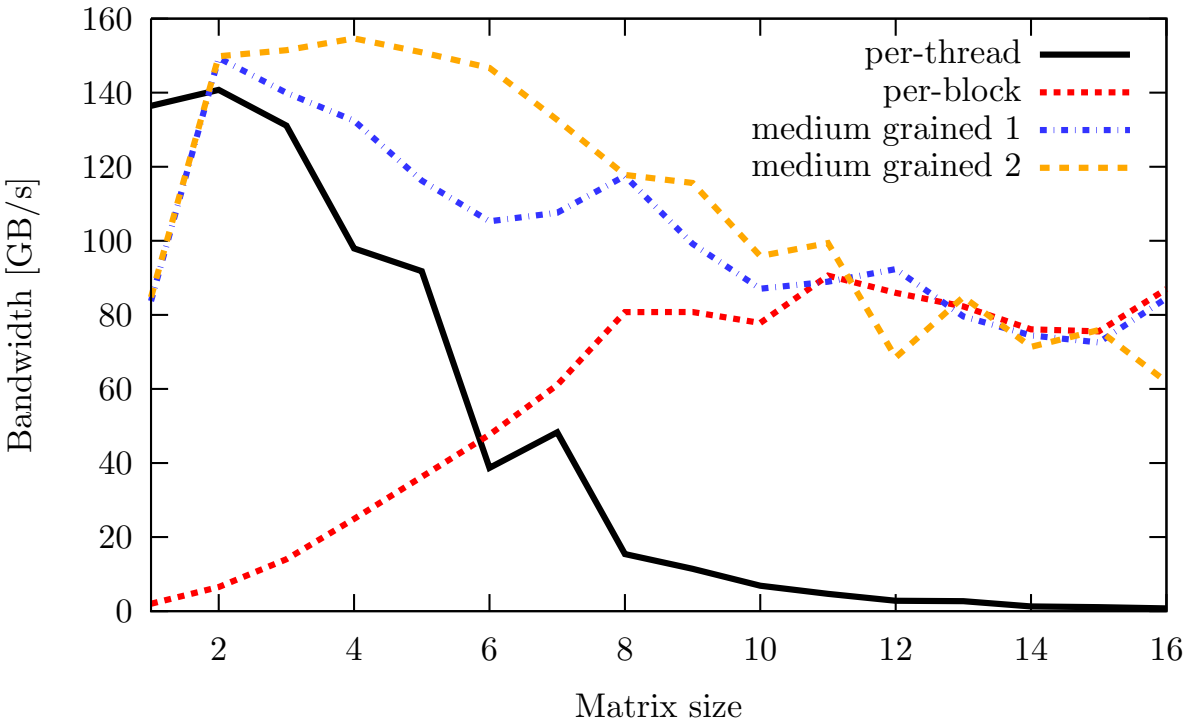
- stejný problém s násobnými broadcasty
- navíc obtížnější vyrovnání-se s konflikty čtení z B

Násobení matic



Obrázek: GeForce 280

Násobení matic



Obrázek: GeForce 480

Někly lze optimalizovat specifické případy

Násobení 4×4 matic

- lze obejít násobné broadcasty
  - při 4 threadech na funkci začneme násobit vektory dávající element (i, j) od pozice j, respektive i počítá-li jeden thread sloupec výsledné matice
  - při 16 threadech na funkci začneme násobit vektory dávající element (i, j) od pozice $(i + j) \mod 4$
- nefunguje obecně

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

Dekompozice-fúze

- 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

Mapované funkce

Naše motivace k optimalizaci paměťové lokality

- per-element výpočty v metodě konečných prvků
- pracují s malými datovými elementy – většina výpočtů omezena propustností paměti (např. násobení 3×3 matic)
- implementace *map* – funkce realizující výpočet je mapována nezávisle na velké množství elementů
- jednoduché fúze – implicitní globální bariéra mezi kernely provádějící *map* lze nahradit lokální bariérou mezi jednotlivými fúzovanými funkcemi

Budeme se zabývat primárně mapováním funkcí.

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× 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 pamětí

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

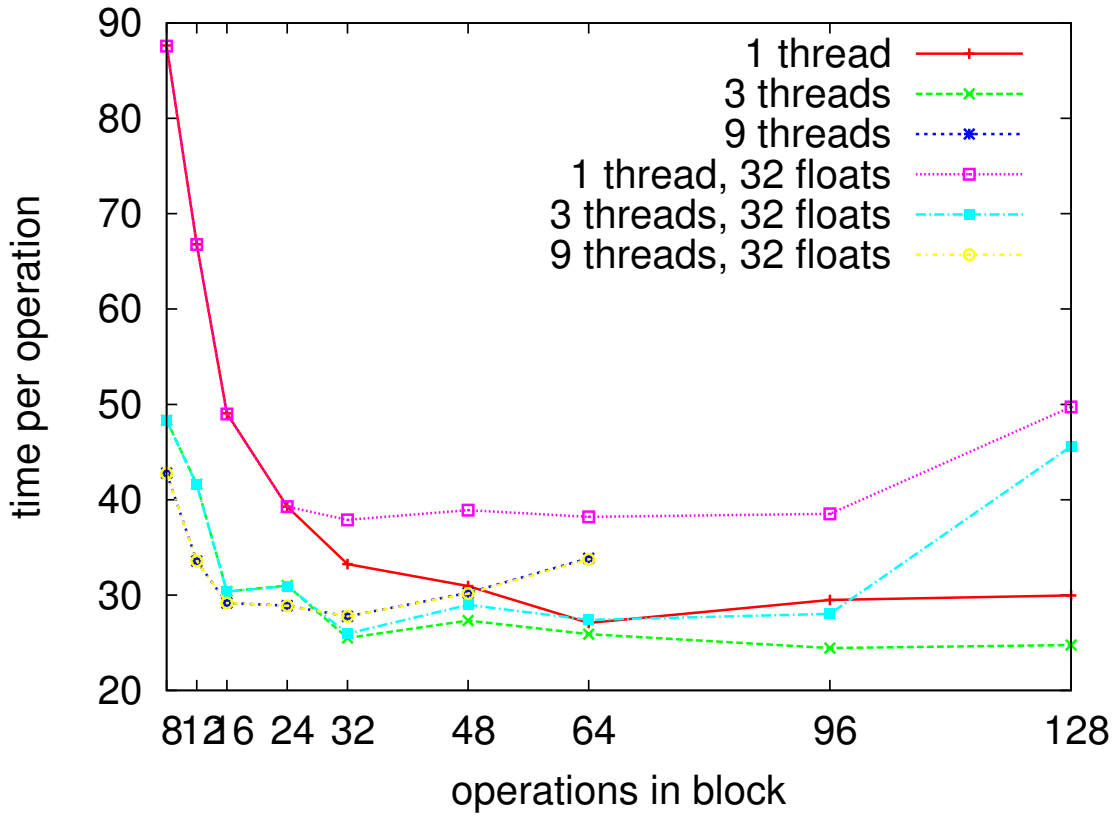
Omezení zrychlení

Rozdílné nároky na paralelismus

- každá instance mapované 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.

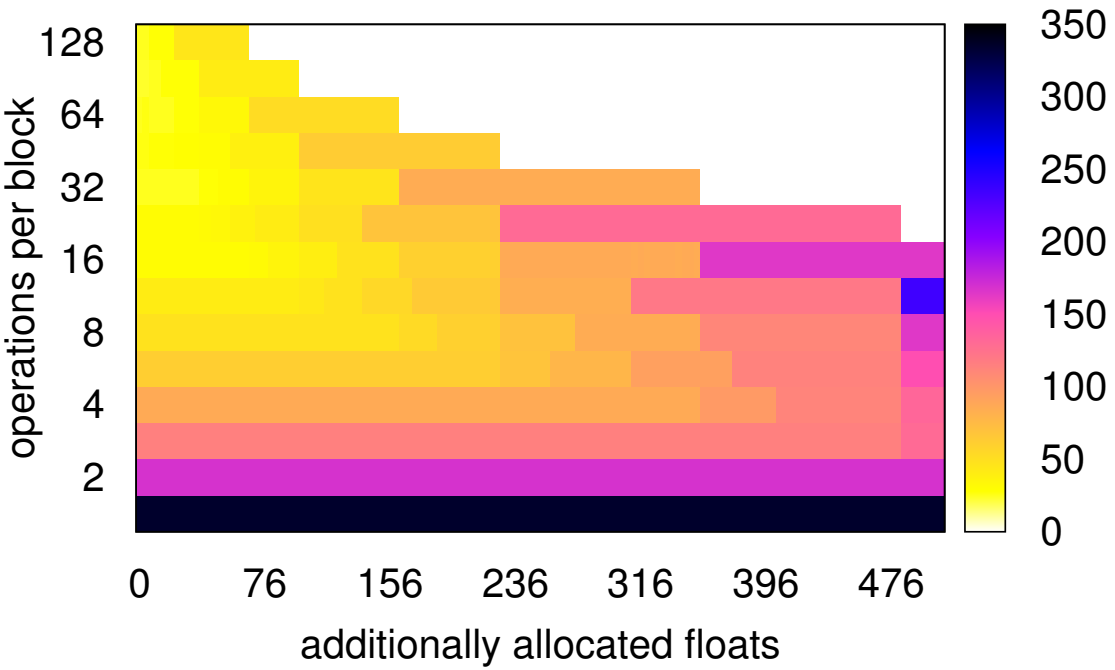


Schéma kompilátoru

Co kompilátor potřebuje?

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

Co musí údělat?

- 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 jejích implementací
  - na základě predikce výkonu vybrat nejvýkonnější implementaci/implementace
- vygenerovat CUDA kód s fúzema

Příklad vysokoúrovňového kódu

Funkci $\mathbf{f} : \mathbf{F} = ||\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{c}||_2 \cdot (\mathbf{D} \cdot \mathbf{E} + \mathbf{D})$, kde A, B jsou 3×3 matice, c vektor velikosti 3 a D, E 5×5 matice zapíšeme v našem jazyce jako:

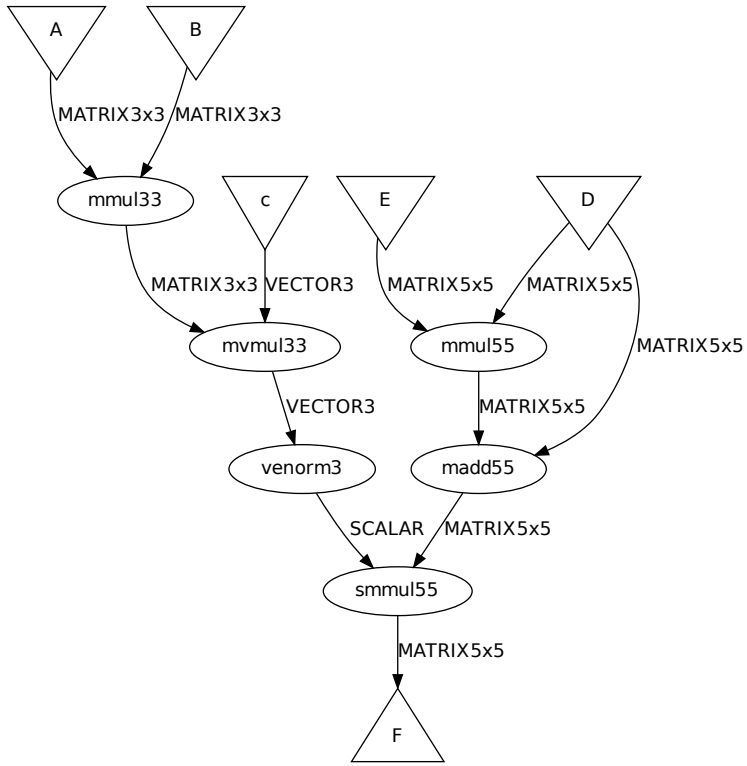
```
MATRIX3x3 A, B, M1;
MATRIX5x5 D, E, F, M2, M3;
VECTOR3 c, v1;
SCALAR s1;

input A, B, c, D, E;

M1 = mmul33(A, B);        // M1 = A · B
v1 = mvmul33(M1, c);      // v1 = M1 · c
s1 = venorm3(v1);         // s1 = ||v1||2
M2 = mmul55(D, E);        // M2 = D · E
M3 = madd55(M2, D);       // M3 = M2 + D
F = smmul55(M3, s1);      // F = M3 · s1

return F;
```

Kód převedený na DAG

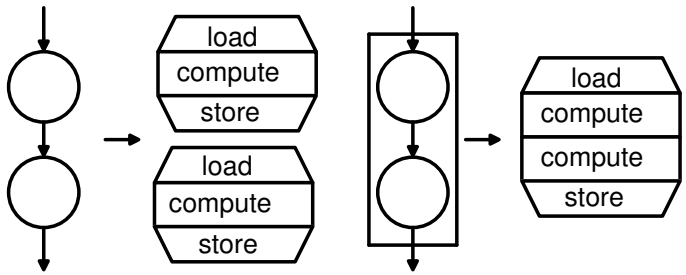


Obrázek: Kód převedený na DAG

Generování kódu I

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

- fúze realizujeme serializací funkcí a odstranění přebytečných load a store rutin
- v současném stavu vývoje pouze přenos dat přes sdílenou paměť



Generování kódu II

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

Predikce výkonu

Hlavní myšlenka

- změříme rychlost rutin v simulovaném prostředí fúze
  - pro měnící-se dodatečnou alokaci sdílené paměti
  - pro rozdílné velikosti bloku

GPU dokáží překrývat paměťové přenosy a výpočty

- pro každou fúzi tedy odhadujeme výkon jako maximum ze sumy časů výpočetních rutin a sumy časů rutin kopírujících data
- pokud dochází k přepočítání souřadnic threadu, přičítáme k času konstantu aproximující čas tohoto přepočítání

V současnosti zanedbáváme

- nedokonalé překrytí výpočtu a paměťových přenosů při redukovaném paralelismu
- režii nepracujících threadů
- celkový mix funkcí ve fúzi (ovlivňuje možnosti GPU přepínat warpy)

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 (spolu s předchozím *implementace fúze*)
- *kombinace implementací fúzí* určuje, které fúze použijeme

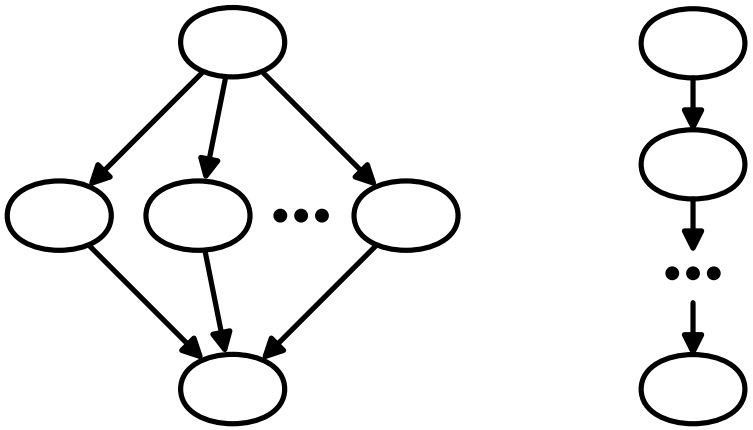
Fúze

Podgrafy DAGu, pro které platí

- neexistuje cesta, která vede ven z fúze a zase se do ní vrací

Fúzí je velké množství

- v nejhorším případě $\mathcal{O}(|2^V|)$, v nejlepším $\mathcal{O}(|n^2|)$



Fúze – prořezávání prostoru

Fúze musí tvořit souvislou komponentu

- jinak nešetříme datové přenosy

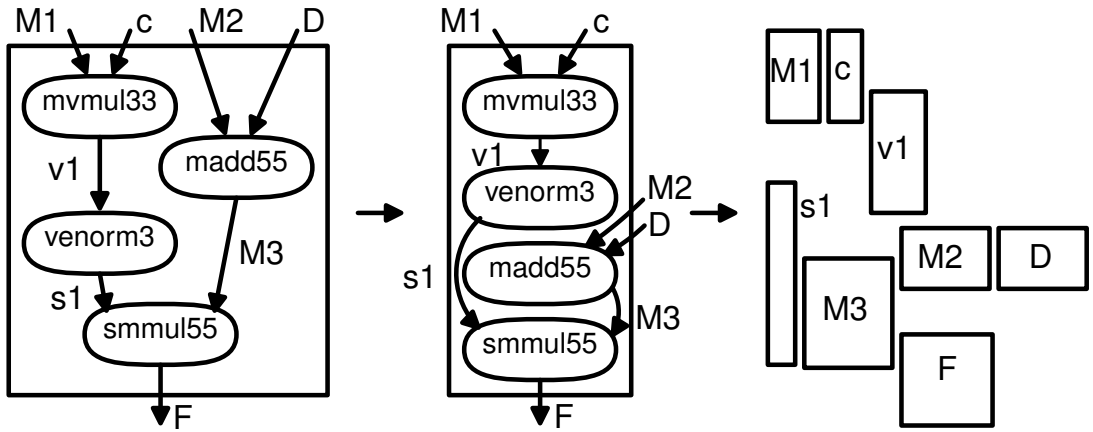
Velikost fúzí lze omezit

- 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 $\sum_{i=0}^{k} \binom{|V|}{i}$
- výrazně zjednoduší prohledávání prostoru implementaci 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 – 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

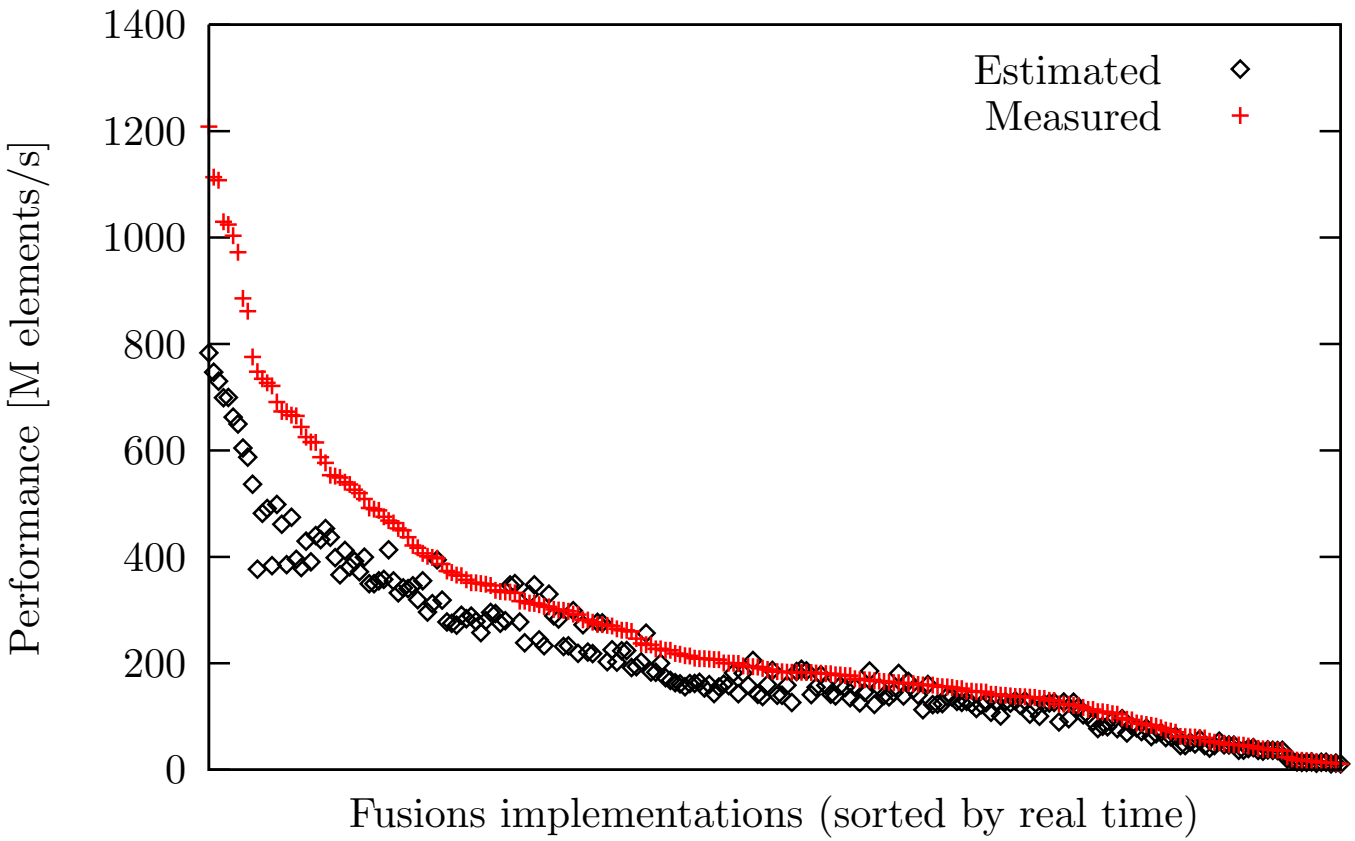
- projdeme všechny přiřazení konkrétních implementací elementárních funkcí: $\prod_{i=0}^{n} \#f_i$, kde $\#f_i$ je počet implementací i-té funkce
- pro každé přiřazení odhadneme výkon pomocí predikce výkonu

Výběr kombinací fúzí

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

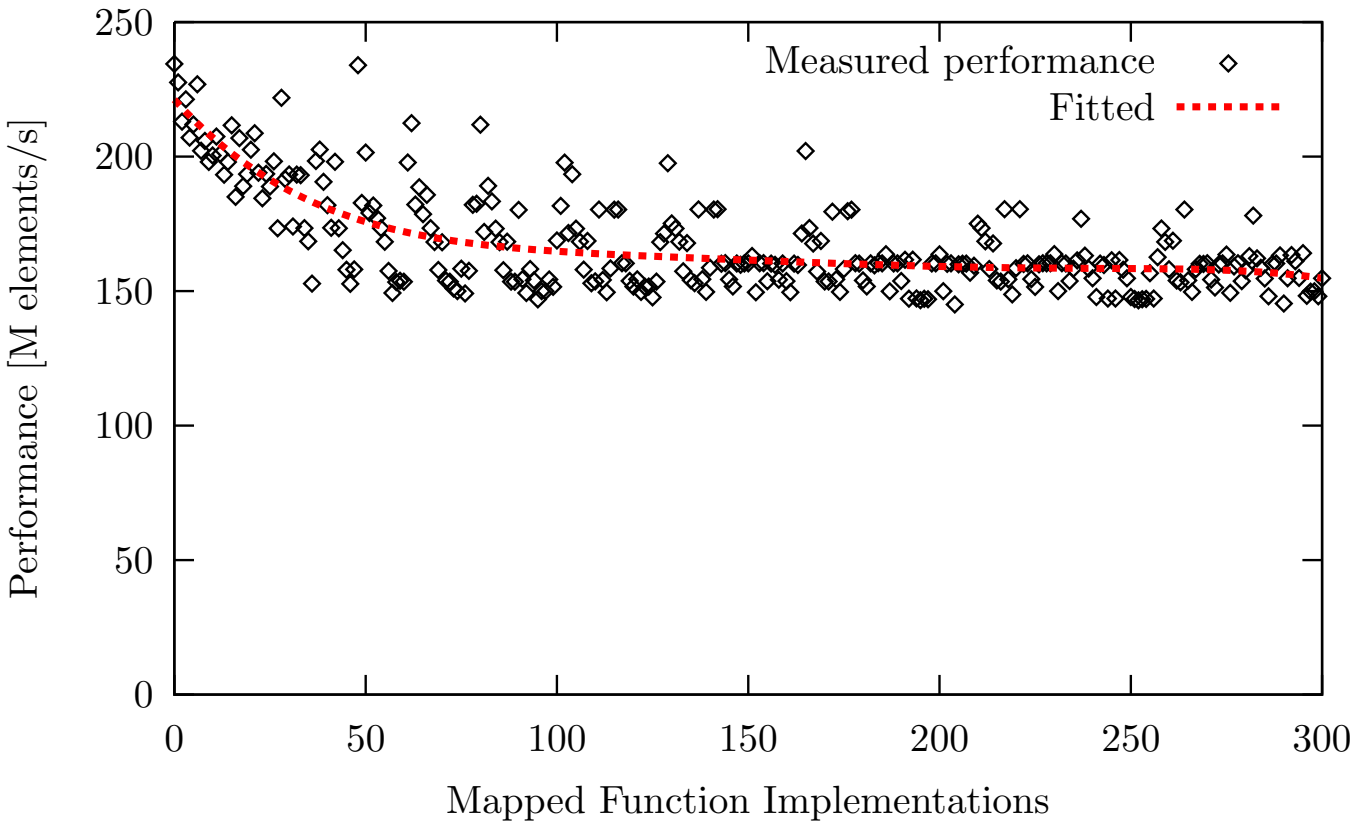
- ze všech kernelů (implementace fúzí a samostatné elementární funkce) vybíráme ty, které dohromady tvoří celý DAG a maximalizujeme odhadnutý výkon
- problém pokrytí množin
- řešeno pomocí lineárního programování

Přesnost predikce



Obrázek: Přesnost predikce výkonu jednotlivých fúzí pro demonstrační příklad, pro GTX480.

Celková úspěšnost výběru implementací I



Obrázek: Reálná rychlost generovaných implementací demonstračního příkladu pro GTX480.

Celková úspěšnost výběru implementací II

rychlost	pořadí výběru
1.	1
2.	49
3.	2
4.	7
5.	29

Tabulka: Pořadí výběru pěti nejrychlejších implementací

Ukázka vybraných fúzí

fúze	granularita
$\{ \|\|A \cdot B \cdot v\|\|_2 \} \cdot \{ (C \cdot D + C) \}$	$\{3\ 3\ 1\}\ \{5\ 5\ 5\}$
$\|\| \{ A \cdot B \cdot v \} \|\|_2 \cdot \{ (C \cdot D + C) \}$	$\{3\ 3\}\ 1\ \{5\ 5\ 5\}$
$\|\|A \cdot B \cdot v\|\|_2 \cdot \{ (C \cdot D + C) \}$	$3\ 1\ 1\ \{5\ 5\ 5\}$
$\{ \|\|A \cdot B \cdot v\|\|_2 \cdot (C \cdot D + C) \}$	$\{3\ 3\ 1\ 5\ 5\ 5\}$
$\|\|A \cdot B \cdot v\|\|_2 \cdot (C \cdot D + C)$	$3\ 1\ 1\ 5\ 5\ 5$

Tabulka: Vybrané fúze seřazené dle rychlosti

Rozšíření kompilátoru

Generování efektivnějšího kódu

- vyjádření vztahu threadů k přistupovaným datům
  - předávání dat pomocí registrů
  - vynechání synchronizace
  - složitější predikce výkonu
- optimalizace mimo bloky (DAG)
  - unrolling for cyklů
- paralelní běh nezávislých kernelů a přenosů paměti

Rozšíření kompilátoru

Efektivnější výběr rychlých implementací

- odhad překryvu paměťových operací a výpočtu
- analytický výpočet doby paměťových přenosů
- redukce systematické chyby u benchmarkingu jednoduchých kernelů

Spousta další implementační práce směrem k produkční verzi :-).

Rozšíření kompilátoru

Samotné mapování má relativně širokou aplikaci

- díky obtížné synchronizaci mezi bloky se jedná o přirozený model pro GPU
- nejsme nutně omezeni na malé datové struktury
  - součet velkých vektorů lze vyjádřit jako mapování mnoha součtů malých vektorů

Velmi zajímavé rozšíření aplikace přinášejí redukce

- vyžadují globální bariéru, výsledek redukce tedy nelze použít ve stejné fúzi
- nejnáročnější první iteraci však můžeme fúzovat s mapovanou funkcí (redukujeme zvlášť výsledek každé instance mapované funkce)
- mapování a redukce nám umožní fúzovat BLAS rutiny