MASARYKOVA UNIVERZITA

FAKULTA INFORMATIKY

Generátor efektivního kódu fúzovaných CUDA kernelů

Diplomová práce

Bedřich Lakomý

Brno, 2012

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval(a) samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal(a) nebo z nich čerpal(a), v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Poděkování

Rád bych poděkoval mému vedoucímu práce Jiřímu Filipovičovi za jeho rady, komentáře, podporu a (zřejmě) nekonečnou trpělivost během mé práce na tomto tématu.

Dále bych rád poděkoval autorskému týmu KASu za vytvoření tak zajímavého nástroje, který dal vzniknout i tomuto tématu.

Nakonec bych rád poděkoval své rodině a přátelům za jejich podporu během psaní této práce.

Shrnutí

KAS je source-to-source kompilátor pro tvorbu CUDA C kernelů z elementárních funkcí. Tato práce optimalizuje kód generovaný KASem pomocí následujících úprav KASu: snížení náročnosti přepočtu koordinát vláken v případě volání funkcí s rozdílnou velikostí bloku, vynechávání některých redundantních synchronizací, využití registrů pro mezivýsledky (dosud jen sdílená paměť), zavedení přístupových vzorů pro elementární funkce a úprava modelu výkonu pro zohlednění využití registrů a jejich vlivu na výkon. Je změřen a diskutován vliv těchto optimalizací na výkon.

Klíčová slova

optimalizace, kompilátor, fúze, CUDA, mapování funkcí, kernel, vzor přístupu

Obsah

1 Úvod

Výkon současných GPU je řádově vyšší než výkon současných CPU. Dosahují toho použitím jednodušších, ale ne tak flexibilních výpočetních jader a masivním paralelismem. Efektivní využití GPU je ale obtížné právě díky jeho specifické architektuře. Pro jeho dostatečné využití musí být řešený problém dostatečně paralelizovatelný.

Jedním z přístupů, jak dosáhnout dostatečné úrovně paralelismu, je mapování funkcí. Bohužel, vývoj větších monolitických funkcí na GPU je značně problematický – protože se může v různých místech funkce lišit optimální granularita dat, nebude dosaženo dostatečného výkonu a změny provedené s cílem zvýšit výkon jedné části mohou degradovat výkon jiné. Pro řešení tohoto problému bylo v (1) navrhnuto použít schéma dekompozice – fúze, kdy se problém popíše jako posloupnost elementárních operací. Tyto elementární operace jsou efektivně implementovány a z hlediska výpočetního hlediska jde o operace omezené paměťovou propustností. Tyto operace jsou pak složeny do výsledného celku, kde jsou použity s pro ně optimální granularitou a své výsledky si předávají pomocí rychlé on-chip paměti. Jedním z výstupů výzkumu tohoto problému je KAS (Kernel Assembler), experimentální překladač, který automatizuje proces fúzování posloupnosti operací do větších celků.

Cílem této práce bylo vylepšit KAS tak, aby generoval efektivnější výsledný kód. Identifikované oblasti byly minimalizace aritmetiky nutné k přepočtu koordinát vláken pro jednotlivé fúzované funkce a vynechávání redundantních synchronizací. Dalším cílem bylo prozkoumat možnost výměny mezivýsledků pomocí registrů a v případě jeho implementace úprava modelu výkonu, aby zohlednil dopad této optimalizace na výkon.

Práce je strukturovaná následovně. Po úvodu následuje druhá kapitola, kde jsou představeny méně známé technologie použité při řešení této práce. Třetí kapitola se věnuje popisu KASu před úpravami – jeho celkové architektuře i jeho jednotlivým komponentám. Ve čtvrté kapitole jsou popsány změny provedené v rámci této práce, v páté je pak změřen a vyhodnocen jejich dopad na výkon generovaného kódu. Šestá kapitola pak obsahuje závěr této práce.

2 Použité technologie

2.1. CUDA

CUDA je proprietární standard firmy NVIDIA pro GPGPU – obecné výpočty na grafických akcelerátorech. Ideálními problémy pro řešení pomocí GPGPU jsou problémy vysoce datově paralelní – ideálně tzv. embarassingly parralel problems. Dnešní GPU ale dokáži efektivně počítat i některé problémy s datovými závislostmi.

Výpočetní model CUDA je označován jako SIMT (Single Instruction Multiple Thread) – několik vláken provádí jednu instrukci nad různými daty. CUDA rozděluje kód podle toho, kde má být spuštěn: *host*, který je pro CPU a *device*, který je pro GPU (tzv. kernel). V pojetí CUDA jde o dvě fyzicky oddělené zařízení, každé s vlastním procesorem a pamětí.

CUDA C je rozšíření jazyka C pro práci s CUDA hardware. K dispozici jsou ale i varianty pro jiné jazyky, např. Fortran, C++ nebo Java. Jazyk C je rozšířen o několik kvalifikátorů funkcí a proměnných pro upřesnění jejich umístění/účelu, zabudované proměnné pro kernely, celou knihovnu funkcí pro nastavení prostředí, komunikaci host-device a speciální syntax pro spouštění kernelů.

2.1.1. Kernel

Jedno z hlavních rozšíření jazyka CUDA C. Kernel je funkce, která má být spuštěna na GPU ve více současně běžících (paralelních) instancích - vláknech. Pro označení funkce jako kernelu slouží nové klíčové slovo __global__. Pro volání kernelu byla zavedena tzv. *execution configuration syntax*: <<<…>>>. Každá instance kernel/vlákno má k dispozici své umístění v rámci bloku a mřížky v podobě zabudovaných proměnných threadIdx a blockIdx. Příklad jednoduchého kernelu pro sčítání vektorů viz **Kód 2.1**.

Kód 2.1: Definice a volání CUDA kernelu (převzato z **(2)**)

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

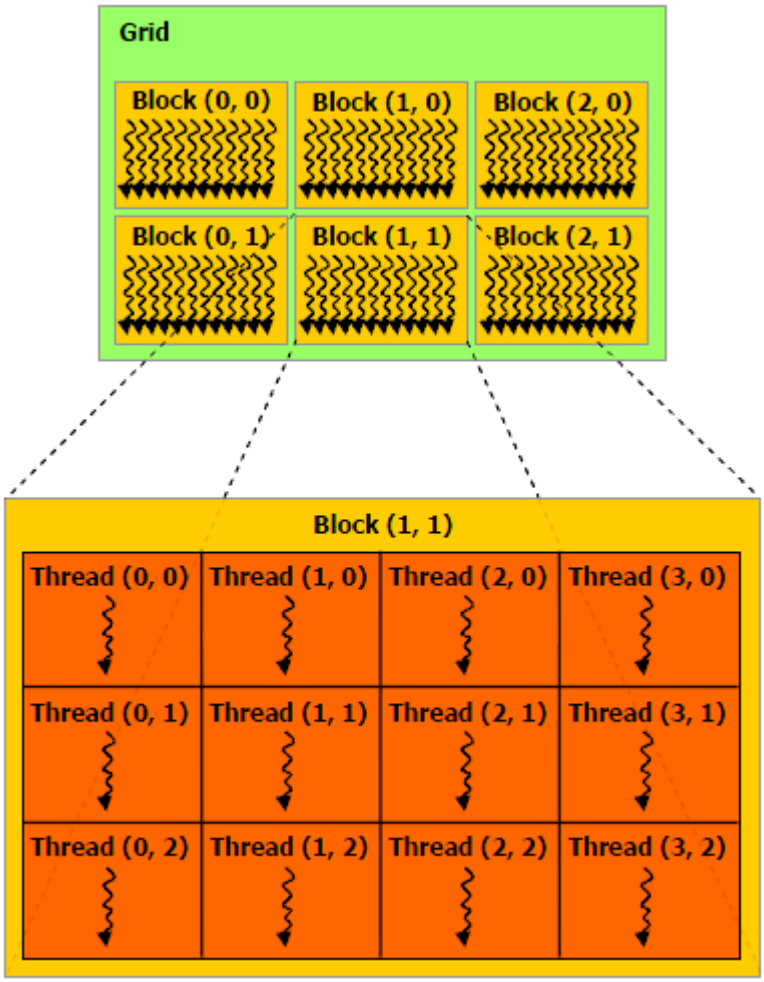
Souvisejícím rozšířením jsou *device* funkce (klíčové slovo __device__), funkce, které lze volat pouze z kernelu, ale nejdou jako kernel spustit z *host* kódu.

2.1.2. Hierarchie vláken

Exekuční model CUDA není plochý – nelze přímo spustit kernel v tisícovce vláken a čekat, že se všechny spustí najednou a bude možná komunikace mezi jedním každým z nich.
Z důvodu škálovatelnosti zavádí CUDA nad vlákny hierarchii o třech stupních: mřížka (grid), blok (block) a vlákno (thread). Mřížka obsahuje několik bloků, a každý blok obsahuje několik vláken.

Tento model kopíruje architekturu CUDA hardware, kde jeden grafický procesor se skládá z mnoha multiprocesorů, každý složený z více jader. Detailní popis architektury je k dipozici v (2).

Obrázek 2.1: Hierarchie vláken v CUDA (převzato z **(2)**)



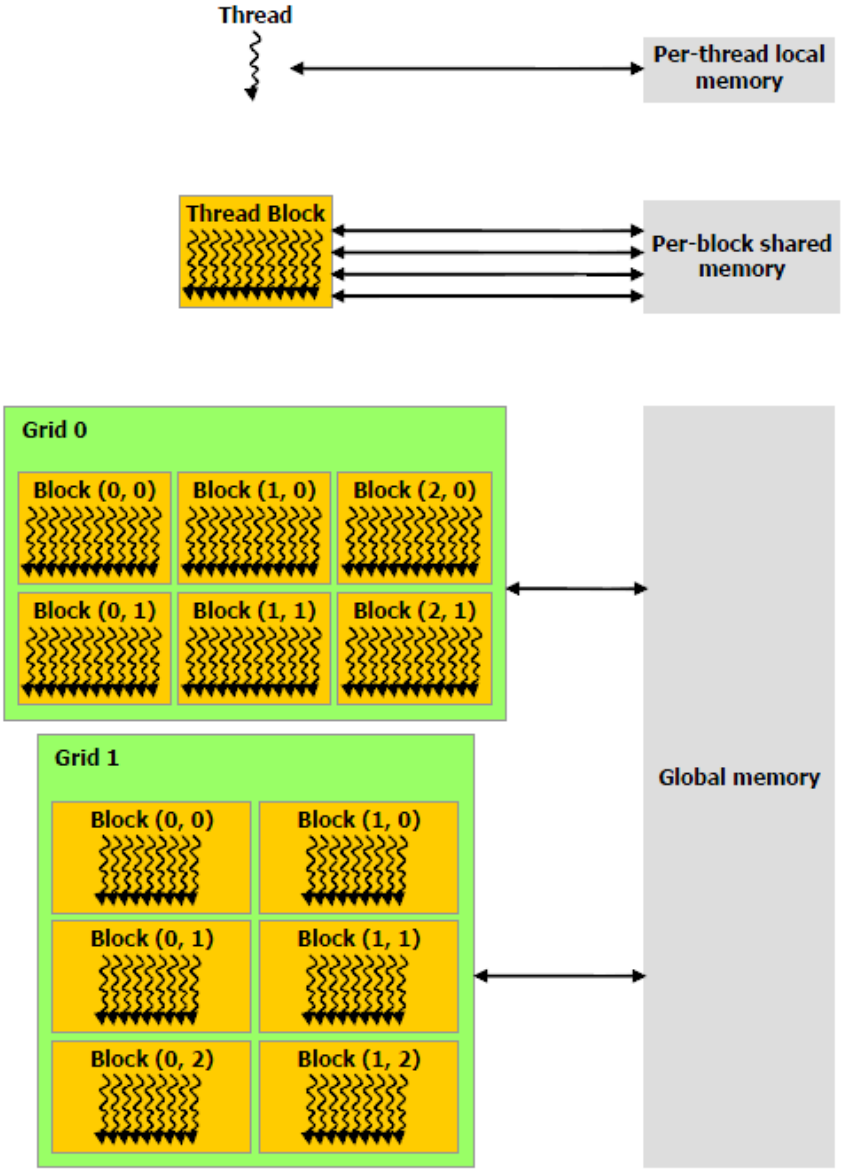
CUDA umožňuje, aby mřížka i blok byly až třídimenzionální. Velikost jednotlivých dimenzí mřížky a bloků jsou povinnými parametry pro spuštění kernelu a je možné k nim v kernelu přistupovat pomocí předdefinovaných proměnných gridDim a blockDim.

Vlákna v jednom bloku spolu mohou komunikovat pomocí výměny dat přes sdílenou paměť a synchronizace, která se provádí pomocí volání funkce __syncthreads() v kernelu.

2.1.3. Hierarchie paměti

Kernely mohou pracovat s několika paměťovými prostory. Každé vlákno má privátní paměťový prostor – registry. Blok vlastní sdílenou paměť, ke které mohou přistupovat všechna vlákna bloku a má stejnou životnost jako blok. Hardwarově jsou umístěny v multiprocesoru. Globální paměť má životnost programu a přístup k ní mají všechny kernely v rámci běhu programu. Fyzicky je umístěna v hlavní paměti karty.

Obrázek 2.2: Hierarchie paměti v CUDA (převzato z **(2)**)



Z hlediska rychlosti (ať už propustnost, nebo latence) jsou nejrychlejší registry, sdílená paměť je pomalejší, ale ne řádově. Globální paměť je v porovnání s předchozími dvěma mnohem (řádově) pomalejší. Používá se tzv. maskování latence, kdy se využívá toho, že na multiprocesoru může být mnohem více vláken, než může současně běžet. V případě čekání na dokončení přístupu do globální paměti se přepne na jinou sadu vláken, která může běžet dál.

Efektivní přístup ke globální paměti vyžaduje tzv. zarovnaný přístup (coalesced access). Velmi zjednodušeně jde o to, aby skupina vláken běžících v SIMT přistupovala k souvislému bloku globální paměti. Pokud toto není dodrženo, drasticky padá propustnost paměti. Novější GPU sice používají pro přístup do globální paměti cache, v případě hodně velkých nebo nepravidelných rozestupů mezi adresami však příliš nepomáhá.

Sdílená paměť je organizována do několika banků, ke kterým může být přistupováno zároveň. Pro efektivní využití sdílené paměti je důležité zabránit bank konfliktům (současný přístup více vláken k jednomu banku), jinak dochází k serializaci přístupu k danému banku sdílené paměti.

Sdílená paměť je také vcelku omezený zdroj, a pokud jí kernel využívá příliš mnoho, může dojít k výraznému omezení paralelismu (méně bloků spuštěných najednou) a tím pádem omezení výkonu. Pokud kernel žádá více sdílené paměti, než je na multiprocesoru fyzicky k dispozici, nelze kernel ani spustit.

V případě nedostatku registrů jsou některá data automaticky přesunuta do lokální paměti – vyhrazené části globální paměti s životností vlákna. Jedná se o tzv. *register spilling*. Výkonově se chová lokální paměť stejně jako globální – fyzicky jde o stejnou paměť.

2.2. muParserX

Jednou z úprav KASu provedenou v rámci této práce je zavedení přístupových vzorů (viz kapitola 4.2). Ty jsou složeny z několika jednoduchých matematických výrazů. Valná část nové funkcionality zavedené v rámci této práce stojí na programovém vyhodnocování těchto výrazů.

Bylo rozhodnuto použít jeden z volně dostupných nekomerčních nástrojů pro vyhodnocování matematických výrazů. Hlavními kritérii pro výběr tohoto nástroje byly jednoduchost přidání do projektu, kvalita dokumentace a jednoduchost použití/kvalita API.

Z tří hlavních kandidátů – muParser (3), muParserX (4), Octave (5) – byl podle výše zmíněných kritérií vybrán muParserX.

Ačkoli má muParserX bohatou funkcionalitu (podpora matic a komplexních čísel, definování vlastních funkcí a operátorů), tato práce využívá pouze její základ: vyhodnocování výrazů obsahujících elementární matematické operace a celočíselné proměnné. Budoucí rozšíření KASu však mohou využít více.

muParserX je distribuován pod BSD licencí. Tato licence byla v souladu se svými podmínkami přiložena ke zdrojovému kódu.

3 Automatická fúze a KAS

V této kapitole je představeno schéma automatické fůze a pro jeho automatizaci vyvinutý kompilátor KAS – Kernel ASsembler.

Prvotní motivace vzešla z akcelerace metody konečných prvků (FEM), kde je na síť elementů/dat aplikován stále stejný výpočet. Ruční optimalizace řešení tohoto problému promocí CUDA bylo představeno v (1). O dva roky později byl publikován (6), který představil semi-automatizaci schématu dekompozice – fúze pomocí pro tento účel vyvinutého source-to-source kompilátoru KAS.

Následující popis je zde především pro poskytnutí kontextu pro kapitolu 4, která se zabývá změnami KASu provedenými v rámci této práce. Tato část práce hluboce čerpá z (7) a (8), které popisují prvotní implementaci KASu.

3.1. Automatická fúze a její optimalizace

Dnešní GPU mají mnohem vyšší propustnost instrukcí než globální paměti. Pro plné využití jejich výpočetní síly je proto nutné provést dostatek výpočetních operací na jednotku dat. Nicméně více výpočtů v jednom kernelu nevede vždy k vyššímu výkonu. Během výpočtu se může měnit počet vláken vhodných pro efektivní paralelní výpočet, což může snížit celkový výkon. Navíc s velikostí kernelu roste i spotřeba lokální paměti mezivýsledky, což vede k omezení paralelismu a může dále snížit výkon.

Pro řešení tohoto problému bylo využito schéma dekompozice – fúze, které navrhuje dekomponovat výpočetní problém na několik jednoduchých samostatných kernelů (elementárních funkcí) a některé z nich poté spojit pro dosažení lepší paměťové lokality. Tyto malé kernely jsou typicky omezené paměťovou propustností, protože mají nízký poměr paměťových a výpočetních operací.

Toto řešení má několik výhod: elementární funkce jsou jednodušší na implementaci a optimalizaci a jsou většinou znovupoužitelné a vývoj komplexnějších kernelů znamená spojení několika těchto jednoduchých, což je jednodušší a méně náchylné k chybám, než vždy vyvíjet nový kernel.

KAS byl vytvořen jako source-to-source kompilátor pro automatické fúzování kernelů. (6)

3.2. Architektura KASu

KAS je optimalizující source-to-source kompilátor, jeho vstupem je popis výpočtu v pro tento účel definovaném jazyce, jeho výstupem zkompilovatelný CUDA kód provádějící daný výpočet. Optimalizace spočívají v hledání co nejvýhodnější podoby fúzovaného kernelu.

Jazyk KASu je popsán v následující kapitole. K jeho rozpoznání a zpracování na jeho přechodnou reprezentaci byl použit volně dostupný nástroj ANTLR (ANother Tool for Language Recognition).

Jako přechodná reprezentace jazyka KASu byla zvolena hierarchie orientovaných acyklických grafů (directed acyclic graph, dále jen DAG), kde uzly představují kroky výpočtu a hrany závislosti mezi nimi. Na vrcholu hierarchie stojí DAG, jehož uzly mohou obsahovat kroky

výpočtu, nebo cyklus. V případě cyklu obsahuje uzel další DAG představující výpočet uvnitř cyklu. Tato struktura je rekurzivní.

Optimalizační fáze pak hledá fúzi elementárních funkcí – jednoduchých kernelů se specifickou strukturou, ze kterých KAS skládá výsledný fúzovaný kernel – s co největším výkonem. Výkon není empiricky měřen, ale je odhadnut na základě modelu výkonu. Elementární funkce jsou popsány v podkapitole 3.4, použitý model výkonu v podkapitole 3.5 a hledání nejlepší fúze pak v podkapitole 3.6.

Posledním krokem je vygenerování CUDA kódu z přechodné reprezentace. Tento proces a jeho výstup je popsán v podkapitole 3.7.

3.3. Jazyk KASu

Pro potřeby KASu byl zaveden jednoduchý jazyk inspirovaný jazykem C. Tento jazyk je používán pro popis výpočtu požadované funkce/kernelu. Struktura musí být následující: definice proměnných, označení vstupních proměnných, příkazy (volání funkcí a cykly) a nakonec označení výstupních proměnných. Volané funkce musí být k dispozici v knihovně elementárních funkcí (více v následující kapitole). Jazyk nepodporuje jakékoliv větvení kódu. Formální popis gramatiky je k dispozici v (7).

Kód 3.1: Příklad popisu požadovaného kernelu (převzato z **(8)**)

```
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;
```

Příklad **Kód 3.1** popisuje výpočet funkce $f: X = \|A \cdot B \cdot v\|_2 \cdot (C \cdot D + C)$, kde A a B jsou matice 3x3, C a D jsou matice 5x5, v je vektor velikosti 3 a $\|x\|_2$ je l^2-norma vektoru. Použité typy musí být definovány v knihovně elementárních funkcí.

3.4. Knihovna elementárních funkcí

KAS ke své práci potřebuje knihovnu elementárních funkcí – soubor implementací elementárních funkcí a dat o jejich výkonu. Knihovna dále obsahuje definici „vyšších“

datových typů (např. matice, tensory), se kterými mohou elementární funkce pracovat. Tyto datové typy mají pevnou velikost – matice 3x3 je odlišný datový typ od matice 5x5.

Elementární funkce je funkce, která se má použít ve výpočtu požadovaného kernelu. Může mít více implementací, které se liší typicky velikostí bloku. Vždy má také referenční CPU implementaci pro testování správnosti výpočtu.

Implementace elementární funkce je zkompilovatelný CUDA kernel, který:

- je rozdělen na load, store a compute části (zapsané jako device funkce)
- obsahuje metadata popisující jeho vlastnosti
- je doprovázen souborem obsahujícím jeho změřený výkon

Výše uvedené vlastnosti slouží jedinému cíli: fúzovatelnosti těchto funkcí do větších celků. Rozdělení na device funkce je nutné, aby šel dobře sestavit výsledný fúzovaný kernel, metadata obsahují informace pro KAS, jak fúzovat a výkonová data poskytují KASu informace o chování funkce pod různým obsazením multiprocesoru.

Jednotlivé funkce tvořící implementaci elementární funkce musí dodržovat následující jmenné konvence:

- kernel: cu_jmenoFunkce_cisloImplementaceFunkce
- load: d_jmenoFunkce_cisloImplementaceFunkce_load_cisloLoadFunkce
- store: d_jmenoFunkce_cisloImplementaceFunkce_save
- compute: d_jmenoFunkce_cisloImplementaceFunkce_compute

Metadata obsahují informace, které KAS používá ve své optimalizační fázi. Jsou zavedeny dva typy metadat: globální pro celý kernel a io pro popis load a store device funkce.

Globální metadata obsahují informace o požadovaných rozměrech mřížky a bloku (GX, GY, GZ pro mřížku/grid, BX, BY, BZ pro blok) a počet datových elementů, které jsou zpracovány sekvenčně (BATCH). Jeden z rozměrů mřížky a bloku může být označen jako volný nastavením jeho hodnoty na size. V případě mřížky je pak nastaven podle velikosti zpracovávaných dat. U bloku jeho nastavení závisí na zvolené fúzi.

Kód 3.2: Funkce pro násobení 3x3 matic, globální metadata

```
/*#
 mmul33_2 metadata
 GX = size
 GY = 1
 GZ = 1
 BX = 3
 BY = size (recommended 16)
 BZ = 1
 BATCH = size (recommended 1)
#*/
__global__ void cu_mmul33_2(matrix3x3 a, matrix3x3 b, matrix3x3 c){
}
```

IO metadata obsahují informace o datovém typu, pro který je io funkce napsaná (Data type), typ paměti (Data location) a ostatní parametry (Other parameters), které se používají pro definici seznamu pozic v bloku a mřížce, na kterých io funkce závisí.

Kód 3.3: Funkce pro násobení 3x3 matic, io metadata pro load funkci

```
/*#
  mmul33_2_load_1 metadata

  Data type =  matrix3x3
  Data location = shared
  Other parameters = tx, ty, bx
#*/
__device__ inline void d_mmul33_2_load_1(matrix3x3 m, float *s_m, int tx, int ty, int bx){
      int gofs = bx*3*3*MMUL33_2_BY + ty*3 + tx;
      int sofs = ty*3 + tx;
      #pragma unroll
      for (int i = 0; i < 3; i++){
            s_m[sofs] = m[gofs];
            gofs += MMUL33_2_BY*3;
            sofs += MMUL33_2_BY*3;
      }
}
```

3.5. Model výkonu

Protože vygenerování všech možných kombinací pro danou fúzi a následné změření jejich výkonu by bylo příliš časově náročné, používá KAS empirický model výkonu pro výběr nejlepších kombinací.

Nejdůležitější faktor, který je brán v úvahu, je systém maskování latence na CUDA hardware – vlákna, která čekají na data z globální paměti, jsou přepnuta za vlákna, mohou pokračovat ve výpočtu. Díky tomuto systému má smysl uvažovat pro nějakou fúzi pouze čas strávený činností, kterou je omezená – tedy buď čas strávený v io rutinách, nebo čas strávený výpočtem. Z toho základní myšlenka modelu výkonu (převzato z (8)):

$$t_{bound} = \max\left(\sum t_{load} + t_{store}, \sum t_{compute}\right)$$

Dalším z omezujících faktorů výkonu fúze je velikost alokované sdílené paměti. Protože CUDA neumožňuje dynamickou alokaci sdílené paměti za běhu kernelu, musí se tato alokovat staticky na maximální velikost, která bude během fúze potřeba. Velikost této alokované paměti je typicky větší než každé jednotlivé funkce, ze které je fúze složena. Jak již bylo zmíněno v kapitole 2.1.3, množství alokované sdílené paměti má přímý dopad na výkon kernelu – čím více alokované paměti, tím méně může na jednom multiprocesoru běžet bloků.

To je součástí výkonových dat jednotlivých implementací elementárních funkcí – pro každou část (load, store a compute funkce) je zaznamenán výkon s další alokovanou sdílenou pamětí v rozmezí od nuly do stanoveného limitu (s rozumně velkým krokem, data jsou později interpolována). Tyto data jsou samozřejmě závislá na hardwaru, na kterém byla získána. Pro získání těchto dat podporuje KAS tzv. *benchmarking*, kdy pro tento účel vygeneruje pro každou implementaci elementární funkce program, která tyto data naměří a uloží.

Po zohlednění výše uvedeného pak vypadá upravený odhad výkonu fúze následovně:

$$t_{fusion} = max\left(\sum_{r_{io} \in R_{io}^{F}} \psi(r_{io}, e, M_{r_{io}}), \sum_{r_c \in R_c^{F}} \psi(r_c, e, M_{r_c})\right)$$

Kde množiny R_{io}^{F} a R_c^{F} jsou io a compute funkce použité ve fúzi F, e je počet elementů zpracovaných jednou instancí fúze, M_r je množina obsahující velikost navíc alokované paměti pro rutinu r vzhledem k fúzi F a ψ je funkce, která pro danou funkci, počet elementů zpracovaných jednou instancí fúze a velikost paměti alokované nad rámec funkce vrátí odpovídající čas potřebný pro vykonání dané funkce. (8)

3.6. Prohledávání stavového prostoru

Stavový prostor možných fúzí pro předepsaný výpočet představují všechny kombinace několika různých rozhodnutí o podobě možné fúze. Tento stavový prostor je generován a poté prohledáván v několika krocích.

Následující podkapitoly stručně popisují jednotlivé kroky, detailní popis včetně algoritmů je k dispozici v (8).

3.6.1. Generování fúzí

V této fázi se vygenerují všechny platné fúze velikosti maximálně k a to tak, že se nejdříve vygenerují všechny podgrafy DAGu o velikosti maximálně k a pak se zachovají pouze ty platné.

Fúze $V_f \subseteq V$ na daném DAGu $G = (V, E)$ je platná, pokud pro každou hranu $(v_1, v_2) \in E$ takovou, že $v_1 \in V_f$ a zároveň $v_2 \notin V_f$, neexistuje cesta z v_2 do žádného vrcholu fúze $v \in V_f$. (8)

3.6.2. Linearizace fúzí

Linearizace fúze – transformace grafu závislostí na posloupnost volání funkcí – je prováděna s cílem minimalizace využití sdílené paměti danou fúzí. Toto je nutné, protože CUDA nepodporuje dynamickou alokaci sdílené paměti. Postupně jsou rekurzivně generovány všechny možné linearizace a hledá se ta s minimálním využitím sdílené paměti. Ačkoliv má algoritmus složitost $O(n!)$ a existuje algoritmus s nižší složitostí, protože je velikost fúzí omezena, nemá toto na výkon KASu podstatný vliv. (8)

3.6.3. Alokace sdílené paměti

Po linearizaci fúze je možné přesně určit, kolik sdílené paměti musí být pro danou fúzi alokováno a vytvořit jednotlivé proměnné. Pro předávání mezivýsledků se ve fúzovaném kernelu používá sdílená paměť. Protože CUDA neumožňuje dynamickou alokaci sdílené paměti a pro dosažení co nejvyššího výkonu je potřeba minimalizovat použití sdílené paměti, používá KAS překrývání proměnných. Ta je implementována jako souvislý blok sdílené paměti, do které pak proměnné přistupují na jim přidělený ofset. Maximální velikost sdílené paměti a ofsety se zjistí spočítáním kolizí proměnných.

3.6.4. Kombinace implementací elementárních funkcí

V tomto kroku se pro každou fúzi vygenerují všechny implementace fúzí – všechny kombinace implementací elementárních funkcí, ze kterých je fúze složena.

Jakmile jsou známy implementace elementárních funkcí, je možné odhadnout potřebnou velikost sdílené paměti a tak i výkon dané implementace fúze (viz kapitola 3.5). V tomto bodě dochází k prořezávání stavového prostoru – fúze s nízkým výkonem jsou zahozeny.

3.6.5. Výběr nejlepších kandidátů

V posledním kroku se vyberou takové fúze, které obsahují všechny z vrcholů původního DAGu právě jednou a vykazují maximální výkon. Jedná se o optimalizační variantu problému pokrytí množin, o které je známo, že je NP-těžká. Nicméně pro prořezaný stavový prostor jde řešit pomocí použití lineárního programování – KAS k tomuto účelu využívá volně dostupnou knihovnu lp_solve.

Algoritmus vždy vybere nejlepší kombinaci fúzí a přesune ji z množiny možných řešení do množiny vybraných řešení. Takto pokračuje, dokud nevybral zadaný počet nejlepších řešení, nebo dokud není množina možných řešení prázdná.

3.7. Generování CUDA kódu

Pro všechny vybrané implementace je vygenerován CUDA kód spolu s malým frameworkem, který umožňuje spustit fúzovaný kernel nad náhodnými daty, změřit jeho výkon a otestovat správnost výpočtu porovnáním dat oproti výpočtu provedenému na CPU.

Velmi zjednodušeně lze generování fúzovaného kernelu popsat jako seskládání elementárních funkcí podle předpisu obsaženého v linearizovaném DAGu fúze.

Jednotlivé části vygenerovaného kódu jsou popsány v následujících podkapitolách.

3.7.1. globalGPU

Soubor globalGPU.cu obsahuje vygenerovaný kernel, který má nasledující strukturu:

- device funkce – zkopírované použité device funkce jednotlivých implementací elementárních funkcí
- (fúzovaný) kernel
    - deklarace proměnných – všechny proměnné využívají sdílenou paměť
    - posloupnost volání device funkcí

Volání device funkce se skládá z dvou až tří kroků:

- přepočet koordinát vláken – v případě, že je velikost bloku volané funkce odlišná od velikosti bloku fúzovaného kernelu, provede se přepočet koordinát vláken pro správné fungování volané funkce.
- volání device funkce
- synchronizace – protože všechny jsou ve sdílené paměti, vynutí se po každém volání device funkce synchronizace všech vláken v bloku (CUDA funkce __syncthreads())

Případ volání device funkce, která má odlišnou velikost bloku od fúzovaného kernelu, je z hlediska výkonu problematický: přepočet koordinát vláken je velmi nákladná operace, která se provádí pro každou takovou funkci. Navíc v tomto případě většinou dochází k tomu, že část vláken je nevyužitých, což vede k dalšímu poklesu výkonu. Detailněji je tento problém rozebrán v (8).

V případě, že fúze obsahuje více kernelů, se tato struktura opakuje. Pokud nebyla nalezena žádná fúze, vygeneruje se kernel pro každý krok požadovaného výpočtu zvlášť. Struktura je stejná jako u fúzovaného kernelu, jen obsahuje pouze jednu compute funkci na kernel.

3.7.2. graphGPU

Soubor graphGPU obsahuje volání kernelů ze souboru globalGPU. Pro zlepšení čitelnosti je členěn na tři části:

- Execution funkce zapouzdřují volání jednotlivých kernelů, obsahují nastavení rozměrů mřížky a bloku a volání kernelu.
- Part funkce obsahují volání execution funkcí a volitelně měření jejich výkonu. Jedna part funkce reprezentuje výpočet jednoho DAGu.
- KasComputation funkce obsahuje alokaci paměti na GPU, kopírování zadání na GPU, volání Part funkcí a kopírování výsledků z GPU. Podává také zprávu o celkovém výkonu výpočtu.

3.7.3. graphCPU

Tento soubor je ekvivalent graphGPU pro kontrolní výpočet na CPU. Obsahuje part a cpuComputation funkce se stejnou sémantikou jako graphGPU, jen pro CPU (každá elementární funkce má i CPU implementaci, viz kapitola 3.4).

3.7.4. gen_main

Soubor gen_main obsahuje main funkci, která spustí výpočet na GPU a CPU a porovná získané výsledky. Pokud se tyto nerovnají, vypíše adresu prvního rozdílného prvku.

4 Provedené změny

KAS byl původně napsán tak, aby v kernelech využíval pouze sdílenou paměť: všechny proměnné pro přenos dat mezi load, compute a save device funkcemi byly umísťovány do sdílené paměti. Ale právě nedostatek sdílené paměti je jedním z hlavních faktorů omezujících paralelismus, a tím i výkon, na platformě CUDA.

Protože původní KAS nemá žádné informace o tom, jak jednotlivé elementární funkce přistupují k proměnným, generuje synchronizace za většinou volání funkcí, aby nedošlo k poškození dat ve sdílené paměti. Synchronizace výrazně zhoršuje schopnost CUDA hardware maskovat latenci přepínáním vláken a každá synchronizace se citelně promítá do výkonu kernelu.

Cílem níže popsaných změn bylo odstranění těchto dvou omezujících faktorů: snížit velikost alokované sdílené paměti na kernel přesunutím některých proměnných do registrů a odstranit redundantní synchronizace. Přístup k proměnným v registrech je navíc rychlejší a může vyžadovat méně instrukcí, jak ukazuje (9).

4.1. Přepočet koordinát vláken

V případě, že se má ve fúzovaném kernelu použít implementace elementární funkce s jinou velikostí bloku, než má kernel, je nutné pro její volání přepočítat koordináty vláken.

Původní verze KASu vždy generovala přepočet pro všechny koordináty (tx, ty, tz), nová verze generuje přepočet, pouze pokud je potřeba (tj. pouze pro dimenze s odlišnou velikostí). Pokud se velikost bloku volané funkce liší ve všech dimenzích, je vygenerovaný kód pro přepočet shodný s kódem vygenerovaným před optimalizací.

Kód 4.1: Přepočet koordinát vlákna, nejhorší případ

$$tx = (threadIdx.z * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x)\%funcDim.x$$

$$ty = (threadIdx.z * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x)/funcDim.x\%funcDim.y$$

$$tz = (threadIdx.z * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x)/funcDim.x * funcDim.y$$

4.2. Metadata

Pro dosažení cílů stanovených v úvodu této kapitoly – předávání mezivýsledků přes registry a odstranění nepotřebných synchronizací – je potřeba, aby každé vlákno kernelu přistupovalo pouze k datům, která se nacházejí v daném vlákně. Protože kernel je tvořen posloupností funkcí, jde problém definovat také tak, že všechny funkce daného kernelu musí přistupovat k datům stejně – se stejným vzorcem přístupu. KAS ve své původní implementaci nemá tuto informaci k dispozici. Protože jde o informaci specifickou pro každou implementaci elementární funkce, bylo rozšíření metadat elementárních funkcí logickým krokem.

Metadata byla obohacena o přístupové vzory (Access Patterns, dále jen AP): element popisující vzorec přístupu vlákna k argumentu device funkce, za předpokladu že je umístěn ve sdílené paměti. Funkcionalita popsaná v kapitolách 4.2 a 4.3 staví na těchto informacích.

Access pattern má tři varianty v závislosti na použití:

- Load Store Pattern pro load a store device funkce. Popisuje vzorec přístupu k argumentu, do kterého se nahrávají data z globální paměti/ze kterého se ukládají data do globální paměti. V metadatech LSPATTERN, v kódu APLS.
- In Pattern pro vstupní argumenty compute device funkce. Popisuje vzorec přístupu k argumentu, ze kterého compute funkce čte data. V metadatech INPATTERN i, v kódu APINi. i je číslo označující pořadí argumentu, číslování začíná od nuly.
- Out Pattern pro výstupní argument compute device funkce. Popisuje vzorec přístupu k argumentu, do kterého compute funkce zapisuje. V metadatech OUTPATTERN, v kódu APOUT.

Všechny varianty pak mají shodné ostatní argumenty, které popisují vlastní vzorec přístupu:

- start je výraz popisující adresu prvního elementu, ke kterému se přistoupí. Výraz může obsahovat koordináty vlákna v bloku, koordináty bloku v gridu a rozměry bloku.
- block je výraz popisující objem dat, která se přečtou v každém cyklu. Může obsahovat rozměry bloku a proměnnou i, která začíná na nule a po každém přečtení bloku se inkrementuje o 1.
- stride je výraz popisující vzdálenost, o kterou se změní adresa čteného elementu v každém cyklu. Může obsahovat to samé, co výraz pro block.
- count je číslo určující počet cyklů/přístupů k proměnné.
- noreg je přepínač pro zákaz umístění proměnné do registru. 0 pro vypnuto (tj. proměnná může být umístěna do registru), 1 pro zapnuto.

Na příkladech **Kód 4.2**, **Kód 4.3** a **Kód 4.4** je vidět kód s použitím AP metadat pro funkci násobení 3x3 matic. Je důležité si všimnout, že nestačí pouze přidat AP do metadat, ale je nutné upravit i vlastní kód funkce: konstanty APLS, APIN a APOUT v tělech funkcí. Toto je nutné pro správné fungování přepočtu koordinát vláken a bude popsáno v následující kapitole. Části s konstantami jsou tučně.

Kód 4.2: Funkce pro násobení 3x3 matic s použitím access pattern metadat, load

```
/*#
  mmul33_2_load_1 metadata

  Data type =  matrix3x3
  Data location = shared
  Other parameters = tx, ty, bx
  LSPATTERN = start "ty*3+tx", block "1", stride "MMUL33_2_BY*3", count 3, noreg 0
#*/
__device__ inline void d_mmul33_2_load_1(matrix3x3 m, float *s_m, int tx, int ty, int bx){
      int gofs = bx*3*3*MMUL33_2_BY + ty*3 + tx;
      int sofs = APLS;
      #pragma unroll
      for (int i = 0; i < 3; i++){
            s_m[sofs] = m[gofs];
            gofs += MMUL33_2_BY*3;
            sofs += MMUL33_2_BY*3;
      }
}
```

Kód 4.3: Funkce pro násobení 3x3 matic s použitím access pattern metadat

```
/*#
  mmul33_2_save metadata

  Data type =  matrix3x3
  Data location = shared
  Other parameters = tx, ty, bx
  LSPATTERN = start "ty*3+tx", block "1", stride "MMUL33_2_BY*3", count 3, noreg 0
#*/
__device__ inline void d_mmul33_2_save(float *s_m, matrix3x3 m, int tx, int ty, int bx){
      int gofs = bx*3*3*MMUL33_2_BY + ty*3 + tx;
      int sofs = APLS;
      #pragma unroll
      for (int i = 0; i < 3; i++){
            m[gofs] = s_m[sofs];
            gofs += MMUL33_2_BY*3;
            sofs += MMUL33_2_BY*3;
      }
}
```

Kód 4.4: Funkce pro násobení 3x3 matic s použitím access pattern metadat, compute

```
__device__ inline void d_mmul33_2_compute(float *s_a, float *s_b, float *s_c, int tx, int ty){
    #pragma unroll
    for (int i = 0; i < 3; i++){
        float tmp = 0.0f;
        #pragma unroll
        for (int k = 0; k < 3; k++)
            tmp += s_a[APIN0+k] * s_b[APIN1+k*3+i];
        s_c[APOUT+i] = tmp;
    }
}

/*#
 mmul33_2 metadata
 GX = size
 GY = 1
 GZ = 1
 BX = 3
 BY = size (recommended 16)
 BZ = 1
 BATCH = size (recommended 1)
 INPATTERN 0 = start "ty*9+tx*3", block "1", stride "1", count 3, noreg 0
 INPATTERN 1 = start "ty*9", block "1", stride "1", count 9, noreg 0
 OUTPATTERN = start "ty*9+tx*3", block "1", stride "1", count 3, noreg 0
#*/
__global__ void cu_mmul33_2(matrix3x3 a, matrix3x3 b, matrix3x3 c){
}
```

4.3. Kompatibilita přístupových vzorů

Dva přístupové vzory (AP) jsou považovány za kompatibilní, pokud n-té vlákno u obou přistupuje na stejné adresy.

Původně byl pro řešení tohoto problému zvažován přístup cestou úprav výrazů. Vzhledem k náročnosti tohoto přístupu a omezené velikosti bloků ale byla použita níže popsaná metoda enumerace adres, ke kterým jednotlivá vlákna pro daný AP přistoupí.

Faktická implementace z důvodu efektivity funguje následovně: během načítání metadat elementárních funkcí z knihovny se pro každý AP spočítají dvě množiny: StartSet a AccessSet. Dva AP jsou pak kompatibilní, pokud se jejich StartSet a AccessSet rovnají (obsahují stejná data). Pro vyhodnocování výrazů byla použita knihovna muParserX (viz kapitola 2.2)

StartSet představuje množinu startovních ofsetů pro jednotlivá vlákna v bloku. Napočítá se pomocí cyklického vyhodnocování start výrazu daného AP. Jednotlivé proměnné se mohou pohybovat ve svých odpovídajících mezích, tj. tx od 0 do velikosti bloku v ose x, ty od 0 do velikosti bloku v ose y apod. Velikosti bloku jsou získány z metadat dané elementární funkce (viz **Kód 4.4**).

AccessSet představuje množinu ofsetů, ke kterým jedno vlákno přistoupí, relativně k start ofsetu daného vlákna. Napočítá se count-krát vyhodnocením block a stride výrazu daného AP. Intuitivně lze chápat jako části datového elementu (např. matice 3x3), ke kterým vlákno přistoupí (např. jeden řádek).

4.4. Využití registrů

Aby dokázal KAS využít paměťový prostor registrů, bylo potřeba zavést nová metadata (viz 4.1) a operace nad nimi (viz 4.2). S pomocí těchto nových nástrojů bylo možné implementovat do KASu podporu pro využití registrů v kernelech. Děje se tak ve dvou hlavních krocích, při tvorbě fůzí a generování CUDA kódu.

Během tvorby fůzí se nad hranami DAGu vytváří objekty reprezentující paměťové proměnné v kernelu. Ty byly rozšířeny o určení paměťového prostoru proměnné: sdílená paměť nebo registry. Při tvorbě objektu se na základě AP a rozměrech bloků jednotlivých funkcí rozhoduje, do jakého paměťového prostoru bude proměnná umístěna.

Aby byla proměnná umístěna do registrů, musí splnit tři hlavní kritéria:

- Počet vláken všech funkcí přistupujících k proměnné musí stejný – v případě, že se fúzují funkce s rozdílnou velikostí bloku a je potřeba přepočítávat koordináty. Pokud by byl počet vláken rozdílný, docházelo by ke ztrátě dat.
- AP všech funkcí přistupujících k proměnné musí být kompatibilní
- AP žádné z funkcí přistupujících k proměnné nesmí zakazovat umístění proměnné do registru

Pokud jsou tyto kritéria splněna, je proměnná umístěna do registrů. Intuitivně lze výše uvedená kritéria chápat jako požadavek, aby všechny funkce přistupovaly ke stejným částem datového elementu uloženého v registrech.

Proměnná umístěná ve sdílené paměti a proměnná umístěná v registrech má odlišnou viditelnost, jsou alokovány na rozdílných úrovních: zatímco proměnná ve sdílené paměti existuje na úrovni bloku a je přístupná všem jeho vláknům, proměnná v registrech existuje pouze na úrovni vlákna. V rámci KASu to znamená, že proměnná ve sdílené paměti obsahuje všechny elementy, které bude daný blok zpracovávat. Proměnná v registrech je naopak jeden element, který bude zpracovávat dané vlákno.

V případě, že funkce zpracovává pouze část elementu, který je uložen v registrech, **nedochází** k plýtvání registry – pro každé vlákno jsou alokovány pouze ty registry, ke kterým bude přistoupeno. Toto je důsledkem několika faktorů.

Prvním z nich je úprava device funkcí během generování CUDA kódu dle umístění proměnné: pokud je proměnná ve sdílené paměti, jsou všechny (typicky ale jen jedna na device funkci) odpovídající AP konstanty přepsány na start výraz AP daného argumentu. V případě, že je proměnná v registrech, se tyto konstanty přepíší na nulu. To zajišťuje správnou adresaci proměnných v obou případech. Tato adresace je navíc v čase kompilace statická.

Druhým z nich je fungování polí v registrech. Zkráceně: CUDA nepodporuje pole v registrech. Pokud je ale možné určit v čase kompilace statické adresování pole, je pro každou položku pole vytvořena zvlášť proměnná a tato umístěna do registrů.

Třetím a posledním faktorem je agresivní optimalizace CUDA kompilátoru během generování kernelu. Díky předchozím krokům máme několik samostatných proměnných v registrech, jejichž počet odpovídajících velikosti zpracovávaného elementu. Pro každou z těchto proměnných pak může kompilátor provést optimalizaci – jmenovitě, pokud je daná proměnná nepoužitá, může být zrušena.

Protože se může stát, že jedna device funkce bude volána vícekrát s argumenty v rozdílných paměťových prostorech a tyto vyžadují rozdílnou adresaci, byly zavedeny variace device funkcí: každý z argumentů s AP může být buď v registrech, nebo ve sdílené paměti a proto je potřeba vygenerovat všechny potřebné varianty dané funkce. Ke jménu device funkce je pak připojen sufix označující danou variaci: pro každý argument s AP buď s pro sdílenou paměť, nebo r pro registry. (Matematicky jde o variaci k-té třídy z dvou prvků (shared, register) s opakováním, kde k je počet argumentů device funkce s AP.)

Kód 4.5 ukazuje compute funkci z **Kód 4.4** ve variacích sss (všechny proměnné ve sdílené paměti, výchozí stav) a rsr (první a třetí proměnná registrech, druhá ve sdílené paměti). Nahrazené části jsou tučně.

Kód 4.5: Variace device funkce

```
__device__ inline void d_mmul33_2_compute_bx3_by96_bz1_sss(float *s_a, float *s_b,
float *s_c, int tx, int ty){
    #pragma unroll
    for (int i = 0; i < 3; i++){
        float tmp = 0.0f;
        #pragma unroll
        for (int k = 0; k < 3; k++)
            tmp += s_a[ty*9+tx*3+k] * s_b[ty*9+k*3+i];
        s_c[ty*9+tx*3+i] = tmp;
    }
}

__device__ inline void d_mmul33_2_compute_bx3_by96_bz1_rsr(float *s_a, float *s_b,
float *s_c, int tx, int ty){
    #pragma unroll
    for (int i = 0; i < 3; i++){
        float tmp = 0.0f;
        #pragma unroll
        for (int k = 0; k < 3; k++)
            tmp += s_a[0+k] * s_b[ty*9+k*3+i];
        s_c[0+i] = tmp;
    }
}
```

4.5. Synchronizace

V původní verzi KASu se synchronizace generovaly po každém volání device funkce. Zavedení access pattern metadat umožnilo vytvoření algoritmu pro vynechání některých zbytečných synchronizací.

Algoritmus má dvě fáze: před každým voláním device funkce se zkontroluje nutnost synchronizace. Pokud je nutná, vygeneruje se synchronizace a všechny ovlivněné proměnné se označí jako synchronizované.

Nutnost synchronizace se zkoumá jako dvě podmínky:

- Přistupuje následující funkce k nesynchronizovaným proměnným, které nemají kompatibilní vzor přístupu?
- Zapisuje následující funkce do oblasti sdílené paměti, do které by stále mohla přistupovat předchozí funkce?

Pokud je alespoň jedna z těchto podmínek splněna, je synchronizace označena jako nutná a proběhne i druhá fáze algoritmu.

Přístup k proměnným v registrech z povahy CUDA hardware není nutné synchronizovat a tak jsou algoritmem ignorovány.

4.6. Predikce výkonu

Algoritmus predikce výkonu byl upraven tak, aby bral v potaz využití paměťového prostoru registrů. Některé fůze mohou díky použití registrů konzumovat méně sdílené paměti a díky tomu v konečném důsledku vykazovat vyšší výkon. Šance, že dojde u fůze ke zmenšení nároků na sdílenou paměť, roste s velikostí fůze – čím více operací, tím větší šance, že půjdou jejich mezivýsledky předat přes registry, které jsou z pohledu algoritmu predikce výkonu „zadarmo“. Obecně jde ale o problém silně závislý na zadání a implementaci elementárních funkcí.

Obsazenost registrů algoritmus zanedbává. Nicméně registrů je k dispozici výrazně víc než sdílené paměti (128 KB vs 48 KB u compute capability 2.x (2)) a CUDA kompilátor použití registrů agresivně optimalizuje. Register spilling tedy není pravděpodobný.

5 Naměřený výkon

Optimalizace popsané v předchozí kapitole byly napsány s cílem zvýšit výkon generovaných kernelů. Přesunutí některých proměnných do registrů by mělo přinést vyšší paralelismus (více bloků na multiprocesoru díky menším nárokům na sdílenou paměť) a rychlejší přístup k těmto proměnným. Odstranění některých synchronizací by mělo umožnit GPU lépe maskovat latenci (překryv přístupu do paměti a výpočtu). Úpravy provedené v predikci výkonu sice nemají přímý vliv na výkon, měly by ale zajistit seřazení fúzovaných kernelů podle výkonu.

Všechny testy a měření byly provedeny na pracovní stanici vybavené Intel Core i7 950 @3.07 GHz, 6 GB RAM, NVIDIA GeForce GTX 480 s ovladačem verze 280.13 a CUDA Toolkit 4.0.

5.1. Testovací zadání

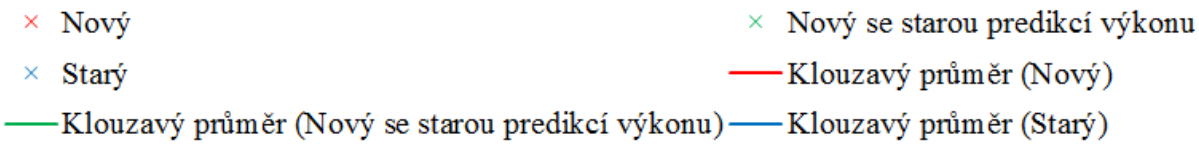
Pro demonstraci dopadu jednotlivých optimalizací byly vybrány tři demonstrativní příklady, které jsou představeny v následujících podkapitolách. V každé je nejprve představeno dané zadání, dále je uveden graf výkonu vygenerovaných kernelů seřazených dle predikovaného výkonu, porovnání výkonu nejrychlejších kernelů a diskuze těchto výsledků.

Pro vygenerování kernelů byly použity tři verze KASu: původní, nová s původní predikcí výkonu a nová. Tyto verze byly vybrány pro zjištění dopadu úprav predikce výkonu.

Každou z verzí KASu bylo vygenerováno prvních 300 implementací. Protože byla použita výkonová data naměřená na GPU s compute capability 2.0, byly implementace zkompilovány pro tuto architekturu použitím přepínače -arch=sm_20 kompilátoru nvcc.

Pro lepší přehlednost následujících grafů je jejich společná legenda uvedena na **Obrázek 5.1**.

Obrázek 5.1: Společná legenda grafů v kapitole 5



5.1.1. BigFusion

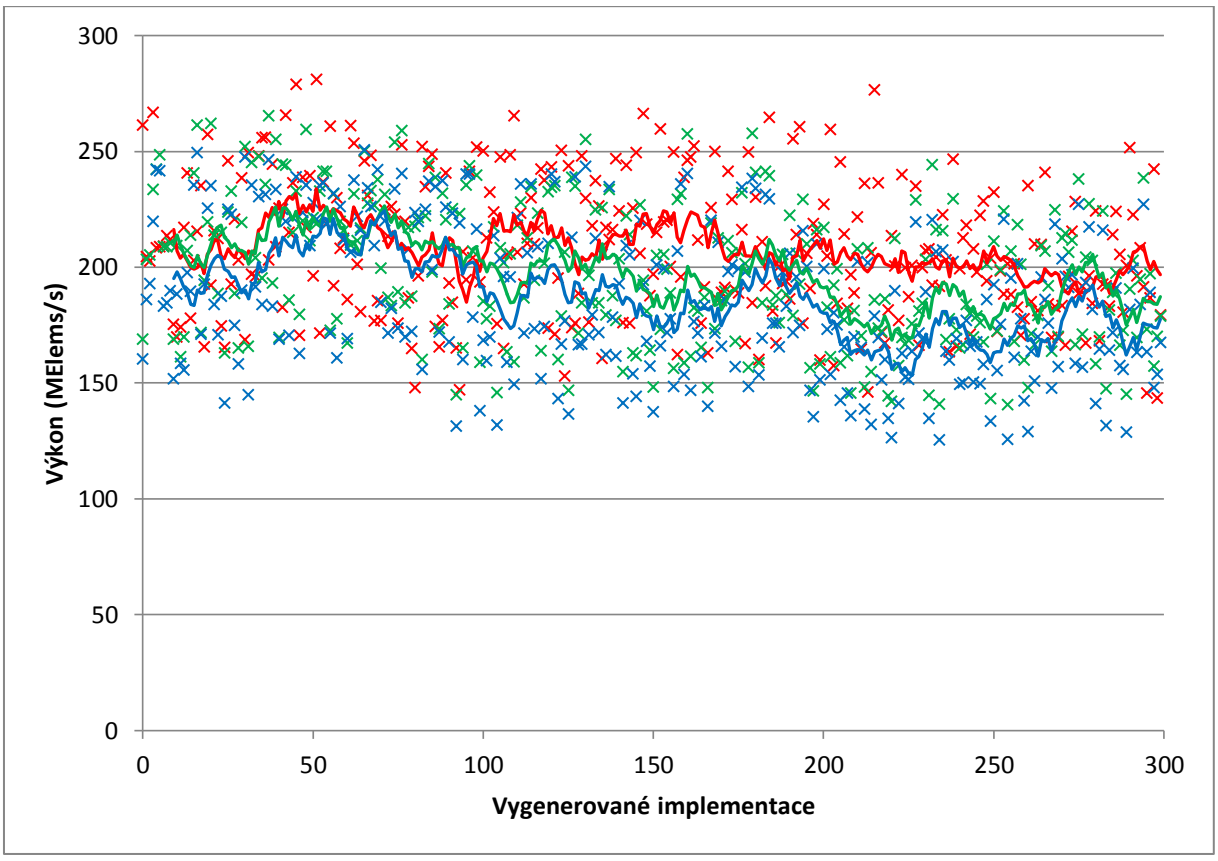
Zadání BigFusion bylo vytvořeno pro otestování schopnosti KASu tvořit velké fúze z odlišných funkcí. Toto zadání provází KAS od jeho vzniku a je zmíněno v (6), (7), (8) a přímo v této práci jako **Kód 3.1**. Pro testování byla použita jeho zjednodušená verze uvedená v **Kód 5.1**.

Kód 5.1: Zadání výpočtu BigFusion (převzato z **(8)**)

```
matrix3x3 A, B, M1;
matrix5x5 D, E, F, M2;
vector3 c, v1;
scalar s1;

input A, B, c, D, E;

M1 = mmul33(A, B);
v1 = mvmul33(M1, c);
s1 = venorm3(v1);
M2 = mmul55(D, E);
F = smmul55(M2, s1);

return F;
```

Graf výkonu vygenerovaných kernelů je vidět v **Graf 5.1**, výkon nejrychlejších implementací a dosažené zrychlení je uvedeno v **Tabulka 5.1**.

Graf 5.1: Vygenerované implementace BigFusion a jejich výkon



Tabulka 5.1: Výkon nejlepších vygenerovaných implementací BigFusion (MElems/s)

Původní	Nový bez predikce	Zrychlení	Nový	Celkové zrychlení
250,57	265,353	1,06x	281,111	1,12x

Jak ukazuje **Tabulka 5.1**, v případě BigFusion bylo dosaženo celkového zrychlení 12% oproti původní verzi. Všechny tři kompilátory shodně zvolily stejnou fúzi funkcí do kernelů: první kernel obsahuje mmul33, mvmul33 a venorm3, druhý kernel mmul55 a smmul55. Rozdíly jsou následující:

- velikost bloku prvního kernelu – zatímco původní a nový bez predikce volí velikost 3:64:1, nový volí větší blok o velikosti 3:96:1
- odstranění redundantních synchronizací – nový a nový bez predikce shodně odstranili tři redundantní synchronizace
- umístění některých proměnných do registrů – nový umístil v prvním kernelu dvě a v druhém kernelu jednu proměnnou do registrů

Z výše uvedeného lze vidět, že zrychlení bez predikce bylo dosaženo pouze díky odstranění redundantních synchronizací a celkového zrychlení bylo dosaženo součinností všech provedených optimalizací. Celkově se ale jednotlivé implementace ani jejich výkon příliš neliší.

Na **Graf 5.1** lze pozorovat velmi mírný sestupný trend výkonu vygenerovaných kernelů, jejich řazení dle výkonu je ale velmi nepravidelné. Nicméně prvních padesát vygenerovaných implementací obsahuje tu nejlepší (v rámci chyby měření) pro všechny tři verze KASu implementace vygenerované novou verzí dosahují stejných nebo vyšších výkonů než implementace starších verzí.

5.1.2. SyncTest

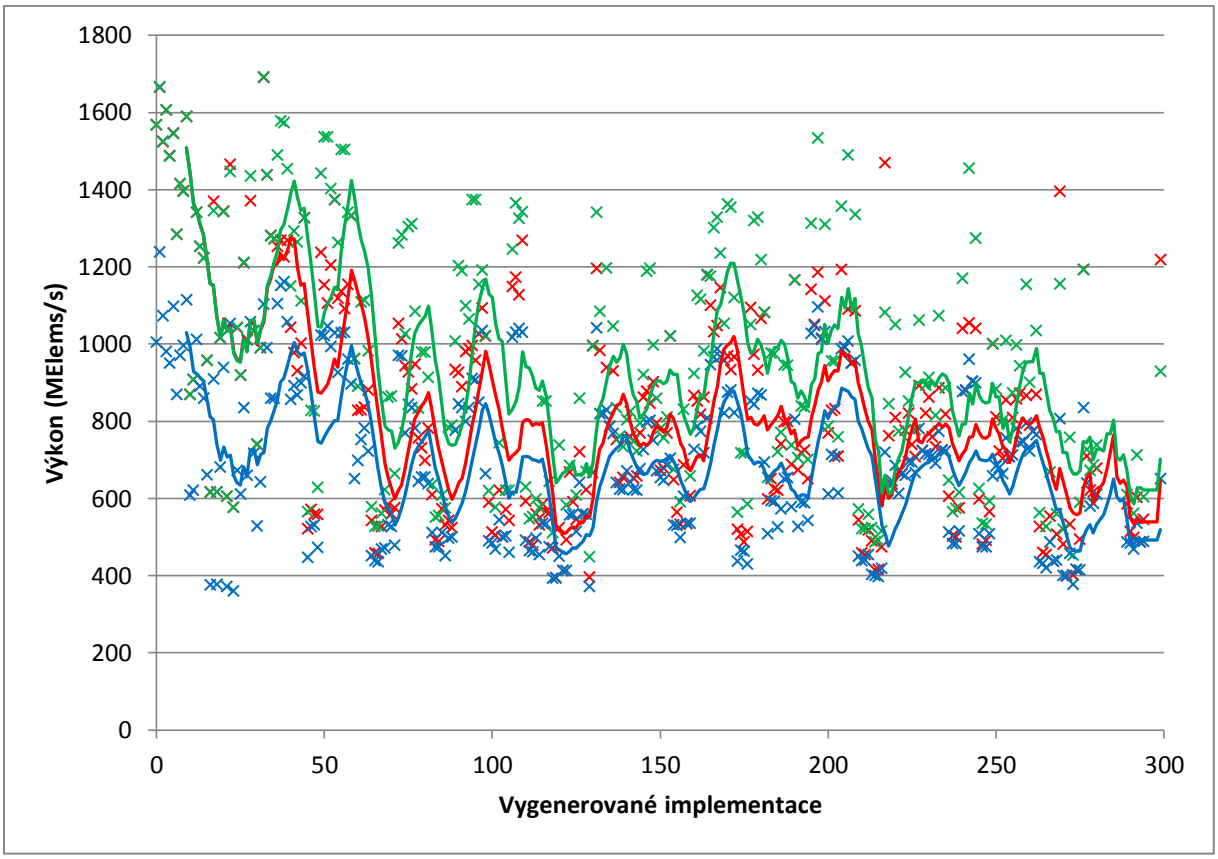
Toto zadání bylo vytvořeno v průběhu této práce pro testování nového algoritmu pro generování synchronizací. V případě použití implementací elementárních funkcí s kompatibilním AP je ve výsledném kernelu potřeba pouze minimální množství synchronizací. Zadání je uvedeno v **Kód 5.2.**

Kód 5.2: Zadání výpočtu SyncTest

```
matrix3x3 A, M_1, M_2, M_3, F;

input A;

M_1 = madd33(A, A);
M_2 = madd33(A, M_1);
M_3 = madd33(A, M_2);
F = madd33(A, M_3);

return F;
```

Graf výkonu vygenerovaných kernelů je vidět v **Graf 5.2**, výkon nejrychlejších implementací a dosažené zrychlení je uvedeno v **Tabulka 5.2**.

Graf 5.2: Vygenerované implementace SyncTest a jejich výkon



Tabulka 5.2: Výkon nejlepších vygenerovaných implementací SyncTest (MElems/s)

Původní	Nový bez predikce	Zrychlení	Nový	Celkové zrychlení
1239,07	1691,75	1,37x	1690,02	1,36x

Výsledky obsažené v **Tabulka 5.2** a **Graf 5.2** poukazují na zajímavou skutečnost: prvních 35 kernelů vygenerovaných novým KASem a novým KASem bez predikce je shodných. Následně začínají výsledky divergovat v neprospěch nového KASu. Po prozkoumání vygenerovaného kódu byla zjištěna příčina: kernely vygenerované novým KASem bez predikce přesunuly více proměnných do registrů. Kód byl jinak identický. Ačkoli jde o nežádoucí chování, není v tomto případě příliš závažné – nejvýkonnější implementace jsou soustředěny do prvních 35 a jsou identické.

Nejvýkonnější implementace se od sebe neliší tvarem fúze (jediný kernel obsahující všechny funkce). Rozdíly jsou následující (nový a nový s původní predikcí jsou identické):

- velikost bloku – původní 64:1:1, nový 3:96:1
- počet synchronizací – snížen z 6 na 3
- přesunutí proměnných do registrů – nový přesunul 3 z 5 proměnných do registrů

Celkově tak byl díky optimalizacím zvýšen výkon o 37%.

Očekávaný sestupný trend výkonu vygenerovaných kernelů je na **Graf 5.2** jasně vidět, včetně prudkého iniciálního poklesu.

5.1.3. Výpočet soustavy rovnic pro StVenant materiál

Na rozdíl od předchozích dvou příkladů je výpočet soustavy rovnic pro StVenant materiál praktickým problémem - je používán v real-time simulátoru hmatové interakce s deformovatelným tělesem. Jeho použití je uvedeno v (10).

Algoritmus 5.1: StVenant element subrutina

1: compute $F = \delta_{xi} + \partial_i u_x$
2: compute $S = \frac{\partial W}{\partial \gamma_{ij}}$
3: compute $inv_1 = \frac{\partial W}{\partial \gamma_{ij}} (\delta_{xi} + \partial_i u_x)$
4: compute $inv_2 = \frac{\partial^2 W}{\partial \gamma_{ij} \partial \gamma_{kl}} \cdot (\delta_{yi} + \partial_i u_y)$
5: // loop over vertices
6: **for** each vertex **do**
7: $A^N \leftarrow inv_1 \cdot \partial_j \phi_N$ // $a_{ij} b_j$
8: $A^N \leftarrow A^N \cdot V$ // $a_i b$
9: $inv_3 \leftarrow inv_2 \cdot \partial_j \phi_N$ // $a_{ijkl} b_i$
10: $inv_3 \leftarrow inv_3 \cdot F$ // $a_{ijk} b_{li}$
11: **for** each vertex **do**
12: $inv_4 \leftarrow inv_3 \cdot \partial_l \phi_M$ // $a_{ijk} b_i$
13: $inv_4 \leftarrow inv_4 \cdot V$ // $a_{ij} b$
14: $inv_5 \leftarrow S \cdot \partial_i \phi_N$ // $a_{ij} b_j$
15: $inv_5 \leftarrow inv_5 \cdot \partial_j \phi_M$ // $|a_i b_i|$
16: $B_{xy}^{NM} \leftarrow I \cdot V$ // $\delta_{ij} \cdot b$
17: $B_{xy}^{NM} \leftarrow B_{xy}^{NM} \cdot inv_5$ // $a_{ij} b$
18: $B_{xy}^{NM} \leftarrow B_{xy}^{NM} + inv_4$ // $a_{ij} + b_{ij}$
19: **end for**
20: **end for**

Algoritmus výpočtu je uveden v **Algoritmus 5.1**, jeho popis je k dispozici v (10). Pro potřeby této práce je podstatný pouze vztah algoritmu k jednotlivým zadáním – zadání **Kód 5.3** počítá řádky 7 – 10, zadání **Kód 5.4** řádky 12 – 18. Je vidět, že jednotlivá zadání korespondují s výpočtem jednotlivých cyklů algoritmu.

Kód 5.3: Zadání výpočtu StVenant

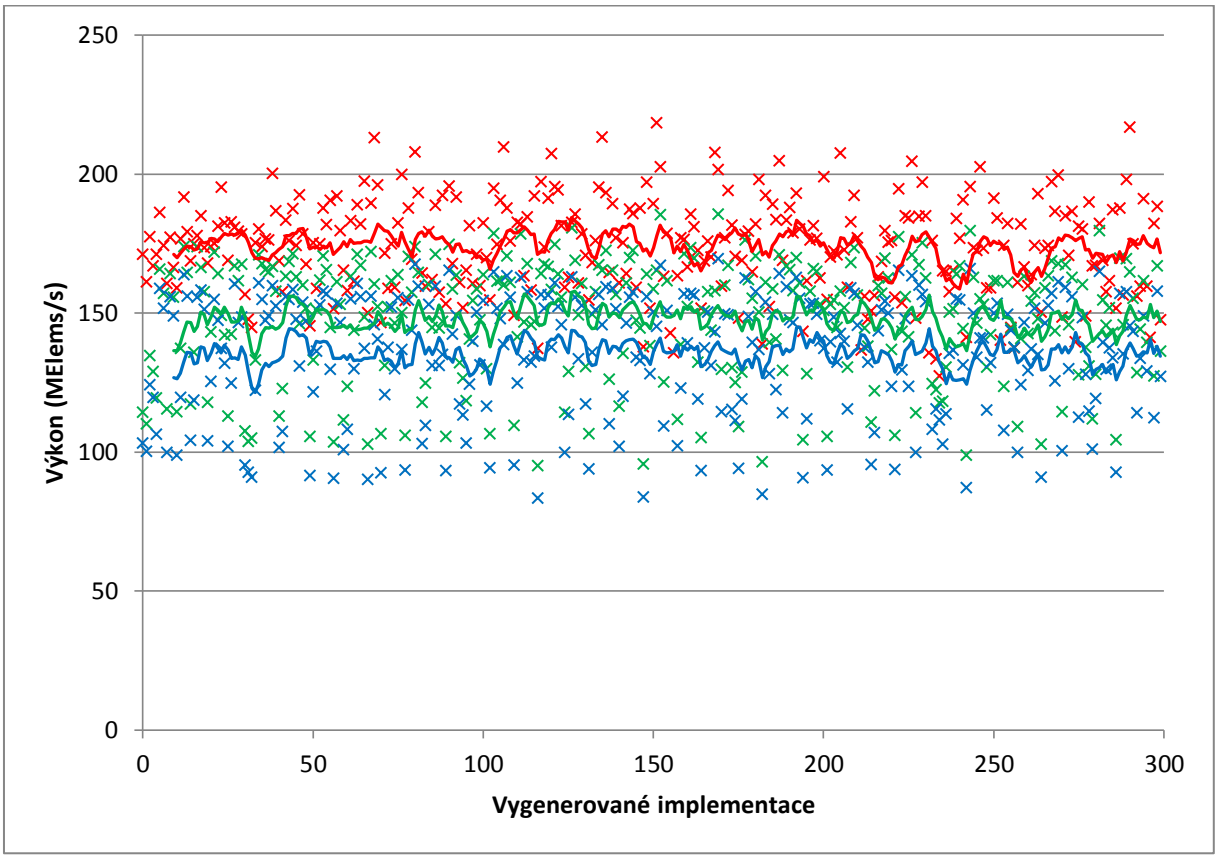
```
tensor3x3x3x3     ten4Help1;
tensor3x3x3       ten3Help1, ten3Help2;
matrix3x3         mat33HelpF, mat33Help1;
vector3           vec3DPhi, vec3NodeStiff;
scalar            vecQuadWeightsSums;

input ten4Help1,mat33HelpF, vec3DPhi, mat33Help1, vecQuadWeightsSums;

vec3NodeStiff = mvmul33(mat33Help1, vec3DPhi);
vec3NodeStiff = svmul3(vec3NodeStiff, vecQuadWeightsSums);
ten3Help1 = ttmul411(ten4Help1, vec3NodeStiff);
ten3Help2 = ttmul3212(ten3Help1, mat33HelpF);

return ten3Help2, vec3NodeStiff;
```

Kód 5.4: Zadání výpočtu StVenantInner

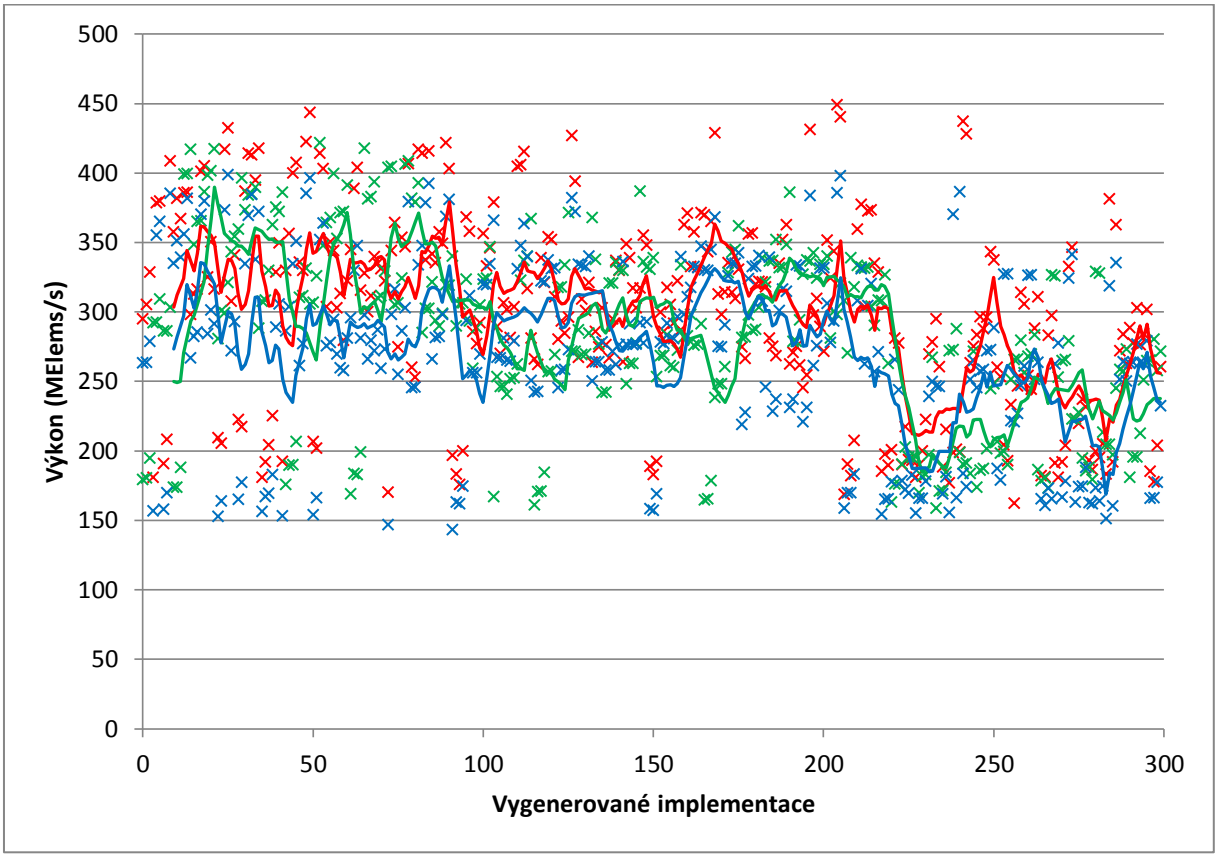
```
tensor3x3x3   ten3Help2;
matrix3x3     mat33Stress, mat33Help2, matTangentHelp;
vector3       vec3DPsi, vec3DPhi, vec3Help;
scalar        scHelp, vecQuadWeightsSums;

input ten3Help2, vec3DPsi, vec3DPhi, mat33Stress, vecQuadWeightsSums;

mat33Help2 = ttmul311(ten3Help2, vec3DPsi);
vec3Help = mvmul33(mat33Stress, vec3DPhi);
scHelp = vvmulabs3(vec3Help, vec3DPsi);
matTangentHelp = smimul33(vecQuadWeightsSums);
matTangentHelp = smmul33(matTangentHelp, scHelp);
mat33Help2 = smmul33(mat33Help2, vecQuadWeightsSums);
matTangentHelp = madd33(matTangentHelp, mat33Help2);

return matTangentHelp;
```

Graf výkonu vygenerovaných kernelů je vidět v **Graf 5.3** a **Graf 5.4**, výkon nejrychlejších implementací a dosažené zrychlení je uvedeno v **Tabulka 5.3** a **Tabulka 5.4**.

Graf 5.3: Vygenerované implementace StVenant a jejich výkon



Graf 5.4: Vygenerované implementace StVenantInner a jejich výkon



Tabulka 5.3: Výkon nejlepších vygenerovaných implementací StVenant (MElems/s)

Původní	Nový bez predikce	Zrychlení	Nový	Celkové zrychlení
169,42	185,509	1,09x	218,346	1,29x

Tabulka 5.4: Výkon nejlepších vygenerovaných implementací StVenantInner (MElems/s)

Původní	Nový bez predikce	Zrychlení	Nový	Celkové zrychlení
398,672	421,841	1,06x	449,109	1,13x

Nejlepší implementace StVenant vygenerovaná novým KASem je o 29% rychlejší než nejlepší implementace původní verze (viz **Tabulka 5.3**). Všechny verze KASu zvolily shodnou fúzi funkcí do kernelů: první kernel tvoří mvmul33 a svmul3, druhou ttmul411 a ttmul3212. Vlastnosti jednotlivých kernelů jsou v **Tabulka 5.5** a **Tabulka 5.6**, za povšimnutí stojí velikost bloku a počet proměnných v registrech u kernelů vygenerovaných novým KASem.

Tabulka 5.5: Vlastnosti nejvýkonnějších implementací StVenant, první kernel

	Původní	Nový bez predikce	Nový
Velikost bloku	128:1:1	128:1:1	128:1:1
Proměnné v registrech	0/5	0/5	2/5
Počet synchronizací	5	4	3

Tabulka 5.6: Vlastnosti nejvýkonnějších implementací StVenant, druhý kernel

	Původní	Nový bez predikce	Nový
Velikost bloku	9:16:1	9:16:1	9:32:1
Proměnné v registrech	0/5	0/5	1/5
Počet synchronizací	6	4	4

Nová predikce výkonu v případě StVenant (viz **Graf 5.3**) konzistentně vybírá fúze s vyšším výkonem než původní verze. Bohužel ale nejsou implementace seřazeny dle svého výkonu a trend jejich výkonu je konstantní. Protože tento problém vykazuje jak původní, tak i nová predikce výkonu, a tak se jedná spíše o nevhodnost použitého modelu predikce výkonu pro tento problém než o chybnou úpravu algoritmu predikce výkonu.

V případě StVenantInner došlo k celkovému zrychlení 13%. Fúze funkcí do kernelů je opět shodná pro všechny verze kompilátoru, všechny funkce byly umístěny do jediného kernelu. Jeho vlastnosti jsou uvedeny v **Tabulka 5.7**. Zrychlení je důsledkem lepšího maskování latence (méně synchronizací) a rychlejšího přístupu k proměnným v registrech. Tyto změny ale nestačily na výraznější zvýšení paralelismu a tak není celkové zrychlení tak vysoké.

Tabulka 5.7: Vlastnosti nejvýkonnějších implementací StVenantInner

	Původní	Nový bez predikce	Nový
Velikost bloku	3:96:1	3:96:1	3:96:1
Proměnné v registrech	0/12	0/12	3/12
Počet synchronizací	13	8	7

Výkon implementací sice vykazuje pozvolný klesající trend, implementace nové verze predikce výkonu ale nevykazují konzistentně vyšší výkon než optimalizované implementace vybrané podle původní verze predikce výkonu (nicméně původní verze bez optimalizací je překonána vždy).

5.2. Vyhodnocení

Dopad odstranění redundantních synchronizací na výkon generovaných kernelů není nijak velký. To je dáno povahou použitých elementárních funkcí a dat – zatížení jednotlivých vláken je téměř uniformní, jediná divergence výkonu kódu nastává v určitých io funkcích a při volání funkcí s odlišnou velikostí bloku. Projevuje se ale pouze tím, že některé vlákna nevykonávají v daný moment žádný kód (tj. pouze if/then, ne if/then/else). Celkově je dopad této optimalizace na výkon v řádu jednotek procent na vynechanou synchronizaci.

Umístění některých proměnných do registrů je složitější k vyhodnocení. Samotné umístění proměnné do registru pouze zrychluje přístupy k dané proměnné. Mnohem důležitější je ale snížení nároků kernelu na sdílenou paměť. Pokud klesne dostatečně na to, aby mohl být na multiprocesor umístěn další blok, dochází k výraznému nárůstu paralelizace a tím i výkonu.

Upravený model výkonu korektně zohledňuje změny zavedené předchozí optimalizací a vykazuje s ní značnou synergii. Díky snížení nároků na sdílenou paměť jsou častěji vybírány fúze s větší velikostí bloku, právě tyto bývají mezi nejvýkonnějšími. Co se týče řazení implementací, vykazuje stejné chování jako jeho původní verze.

Celkový dopad výše popsaných optimalizací na výkon je pouze pozitivní. Jak jde vidět na grafech v této kapitole, vykazují kernely vygenerované novým KASem vždy vyšší výkon než ty původní.

6 Závěr

Cílem této práce bylo vylepšit KAS tak, aby generoval efektivnější výsledný kód. Identifikované oblasti byly minimalizace aritmetiky nutné k přepočtu koordinát vláken pro jednotlivé fúzované funkce a vynechávání redundantních synchronizací. Dalším cílem bylo prozkoumat možnost výměny mezivýsledků pomocí registrů a v případě jeho implementace úprava modelu výkonu, aby zohlednil dopad této optimalizace na výkon.

Těchto cílů bylo dosaženo v plné šíři. Výměna mezivýsledků přes registry se ukázala jako možná a velmi přínosná pro výkon generovaných kernelů. Algoritmus pro vynechávání přebytečných synchronizací dokáže v některých případech identifikovat až polovinu synchronizací jako redundantních a nezahrnout je tak do výsledného kódu, což vede k jeho vyššímu výkonu. Pro potřebu těchto dvou optimalizací byly metadata obohacena o popis přístupových vzorů (což je mnohem lepší řešení, než původně předpokládaná nutnost alternativní verze elementárních funkcí pro práci s registry). Optimalizace aritmetiky přepočtu koordinát vláken omezila výkonový postih za použití elementárních funkcí s odlišnou velikostí bloku. A nakonec upravený model výkonu umožňuje větší využití potenciálu výměny mezivýsledků před registry.

Bohužel, tato práce odhalila i několik nedostatků. Generování implementací trvá o něco déle kvůli novým výpočtům pro porovnávání přístupových vzorů, jejichž implementace není tak efektivní, jak by mohla být. Dalším z problémů je model predikce výkonu, respektive řazení implementací dle výkonu, které už nepodává tak dobře výsledky jako v (6).

Optimalizace KASu provedené v této práci byly použity v (10). I díky nim dokázal KAS vygenerovat implementaci s výkonem o 30% vyšším oproti ručně optimalizované implementaci.

Literatura

1. **Filipovič, Jiří a Igor Peterlík a Jan Fousek.** *GPU Acceleration of Equations Assembly in Finite Elements Method -- Preliminary Results.* Urbana (Illinois) : Symposium on Application Accelerators in High Performance Computing 2009, 2009.

2. **NVIDIA Corporation.** NVIDIA GPU Computing Documentation. *NVIDIA Developer Zone.* [Online] 16. 4 2012. [Citace: 16. 5 2012.] http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

3. **Berg, Ingo.** *muParser - a fast math parser library.* [Online] [Citace: 16. 5 2012.] http://muparser.sourceforge.net.

4. —. *muParserX - A mathematical expression parser with support for arrays, matrices and strings.* [Online] [Citace: 16. 5 2012.] http://code.google.com/p/muparserx/.

5. **Eaton, John W.** *GNU Octave.* [Online] [Citace: 16. 5 2012.] http://www.gnu.org/software/octave/.

6. **Fousek, Jan a Jiří Filipovič a Matúš Madzin.** *Automatic Fusions of CUDA-GPU Kernels for Parallel Map.* London (UK) : Second International workshop on highly-efficient accelerators and reconfigurable technologies (HEART), 2011.

7. **Madzin, Matúš.** *Source-to-source compilation of mapped functions sequences in CUDA.* Brno : Masarykova univerzita, Fakulta informatiky, 2011.

8. **Fousek, Jan.** *Optimization of mapped functions sequences using fusions on GPU.* Brno : Masarykova univerzita, Fakulta informatiky, 2010.

9. **Volkov, V., and Demmel, J. W.** *Benchmarking GPUs to tune dense linear algebra.* Berkeley : 2008 ACM/IEEE Conference on Supercomputing (SC08), 2008.

10. **Filipovič, Jiří a Jan Fousek a Matúš Madzin a Bedřich Lakomý.** *Semi-Automatically Optimized GPU Acceleration of Element Subroutines in Finite Element Method.* Brno : Masaryk Univerzity, 2012. (Zasláno na SAAHPC 2012, v době psaní práce v redakčním řízení).

11. **Aho, Alfred V.** *Compilers: principles, techniques, & tools. 2nd ed.* Boston : Pearson/Addison Wesley, 2007. ISBN 0-321-48681-1.

Přílohy

Přiložené CD

Přiložené CD obsahuje tyto položky:

- tuto práci ve formátu PDF
- zdrojový kód kompilátoru
- soubor obsahující naměřená data použitá v této práci ve formátu XLSX