Násobení polynomů

✍ Robert David
☎
✉ davidrob@fit.cvut.cz

Popis algoritmů

Program se skládá ze 3 algoritmů. Referenční naivní algoritmus (násobení čísel jako na základní škole).
Pokročilejší Karatsubův algoritmus pro násobení velkých čísel. A v hardware často implementovaný FFT algoritmus.
Všechny algoritmy jsou implementované v C. V rámci jednoduchosti jsou testovací data pouze velikosti 2^n, protože zvolená implementace FFT algoritmu zvládá pouze taková data.

Složitosti

Naivní	n^2
Karatsuba	$n^{log(3)}$
FFT	$n * log(n)$

Karatsuba

Tento algoritmus je implementován standardně (např [1]).
Výpočet využívá pomocné pole *D*.

$$D[i] = A[i] * B[i]$$

Výsledný součin následně v poli *C* je vypočítán následovně:

$$C[i] = \sum((A[p] + A[q]) * (B[p] + B[q]) - D[p] - D[q]))$$

kde $p + q = i$ a $p > q >= 0$.

FFT

Pro implementaci byl použit nerekurzivní FFT algoritmus (zdroj [2]).
Tento algoritmus vychází z poznaktu jak je rekurzivně vykonáván výpočet. Místo rekurze pak vhodně permutuje vstupní data a posléze nad nimi provádí výpočet. Na rozdíl od karatsubova algoritmu je tento poněkud složitější.
Nejprve jsou předpočítány hodnoty kořenů jednotkové kružnice v komplexním prostoru. Počet kořenů je jako stupeň výstupního polynomu. Po permutaci vstupních dat následuje vlastní FFT algoritmus. Ten je zde implementován tak, že prochází bloky dat postupně, od bloku velikosti 1 do velikosti výstupního pole. Následující blok je vždy 2x větší. V každém bloku se provede přepočet hodnot v bloku s koeficientem kořenu jednotkové kružnice.
Po FFT jsou přepočítaná vstupní data roznásobena.

$$C[i] = A[i] * B[i]$$

posléze se provede pouze inverzní FFT (FFT s obrácenými kořeny) a data se permutují zpět.

OpenMP

Všechny algoritmy jsou postavené na for cyklech, tedy je použita pouze direktiva FOR. Jediné zásadnější ladění bylo zvolit správné for cykly a nastavit správně proměnné. Většina for cyklů neobsahuje sdílené proměnné, nebo jsou sdílené *chytře*. Nedochází tak zapisování do jednoho místa najednou. Tím pádem jsou pole značená jako FIRSTPRIVATE (ukazatel na pole je private a ukazuje na stejné prvky).

Testování

OpenMP verze je testována na multiprocesorovém stroji:

Označení	Sun Enterprise T5440
CPU	4x SPARC T2+, 1600MHz
HW vlákna	256
Paměť	256GB
Kompiler	gcc 4.5
Flagy	-march=native -O3 -pipe -fopenmp

CUDA verze je testována na GPU clusteru alpha.

GPU	Nvidia GF GTX 480
Kompiler	gcc 4.5
Flagy	-march=native -O3 -pipe -fopenmp

Implementace CUDA

Pro každý algoritmus byla zvolena rozdílná metoda implementace. Karatsubův algoritmus byl implementován pomoci existujícího kódu, postupným přepisem na CUDA. FFT byl přepsán v podstatě kompletně ale pomoci knihovny cuFFT.

Karatsuba

Zde byla využita původní implementace, postupnými kroky převedena na optimální výpočet. Původní funkce byly z převážné části zachovány, byly odstraněny hlavní for cykly a proběhly drobné přesuny. Výsledkem je tak vice funkci které mohou některé běžet nezávisle na přenosu vstupních dat na grafickou kartu.
Po testování počtu vláken grafické karty nakonec padla volba na 1024, tedy největší možný. Počet bloku už závisí na velikosti vstupních dat.
Nakonec pomoci profilace v *nvvp* byl kód upraven, a dosáhlo se ještě dalšího urychlení (cca 200%). Většinou šlo o optimalizaci (minimalizaci) čtecích a

zapisovacích operaci, popřípadě jejich vhodné poskládání.
Dále jsem testoval využití texturovací paměti pro data pro čtení (všechny vstupní), výsledek byl ale spíše horší než lepší. Nakonec jsem se tedy vrátil k původní implementaci. Na GPU třídy 1.x by to asi melo smysl, na 2.x už ne.

FFT

Tato část byla v podstatě přepsána na zelené louce. Vzhledem k jednoduchosti použití cuFFT to byl jednodušší proces. O implementaci ani není potřeba moc mluvit, protože téměř vše obstará knihovna. Stačí jen vstupní data vhodně přeformátovat a podstrčit knihovně. S transformovanými daty posléze provést na GPU roznásobeni a poté výsledek zpět pomoci inverzní fft převést.
Je vidět ze inženýři z Nvidia si dali opravdu záležet a samotné výpočty fft jsou velmi rychlé. Problém ale nastává při inicializaci GPU. Proto pro zvolená testovací data bylo dosaženo menšího zrychlení. Zde by se rychlost naplno projevila až pro data v řádu 10^6

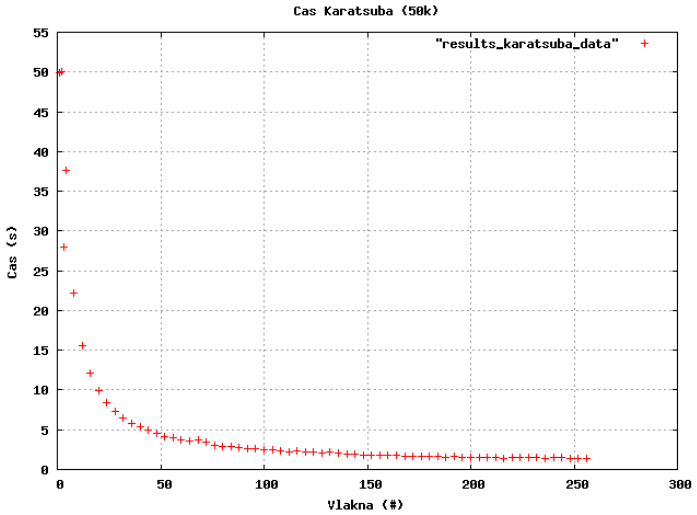
Závěr

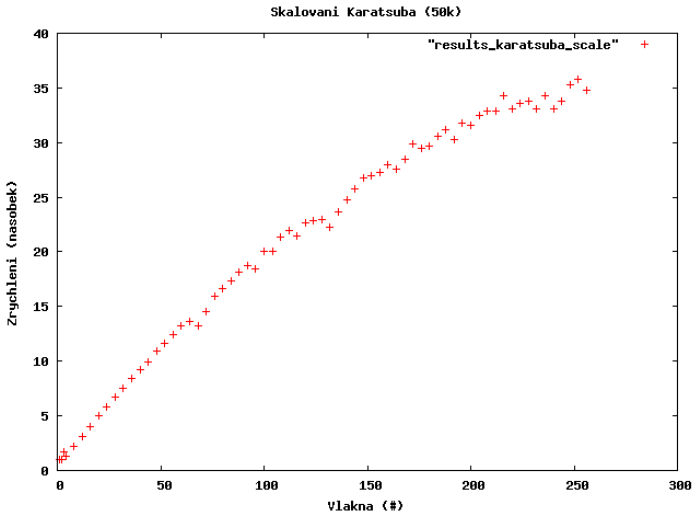
Jak je vidět Karatsubův algoritmus se velmi dobře paralelizuje a to jak pouze na cpu tak na GPU. Z grafu je patrné ze Karatsubův algoritmus na procesoru SPARC zrychluje lineárně v poměru cca 1/8. To částečně vychází z toho, že v tomto cpu nejsou všechna vlákna plně samostatná a jedná se v podstatě o obdobu HyperThreadingu na cpu Intel.
FFT vzhledem ke své sekvenční rychlosti je na paralelizaci daleko vice náročná. Je to vidět i z grafu zrychleni pro fft, kde ke zrychlení dochází pouze do cca 40 vláken. Pote už jde spíše o zátěž. To nesvědčí o tom, že by to nešlo lépe zparalelizovat, ale pouze, že jednoduché použití OpenMP direktiv není všespásné a musely by se použít vice nízkoúrovnové optimalizace a paralelizace.

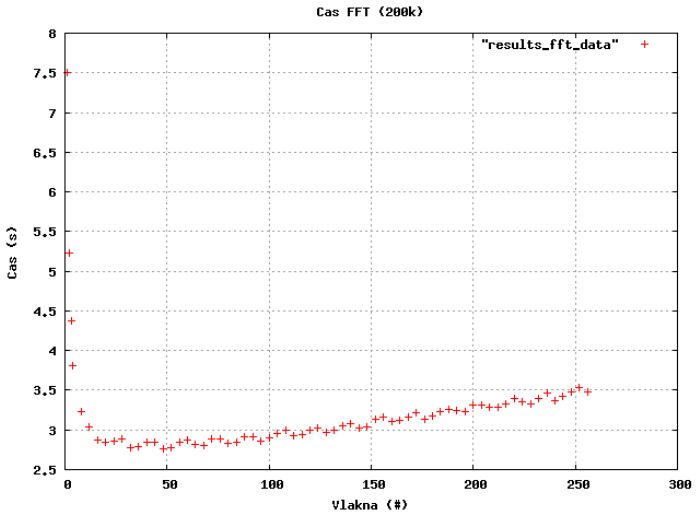
Zdroje

[1] *http://en.wikipedia.org/wiki/Karatsuba_algorithm*
[2] *http://www.math.ethz.ch/education/bachelor/seminars/fs2008/nas/woerner.pdf*

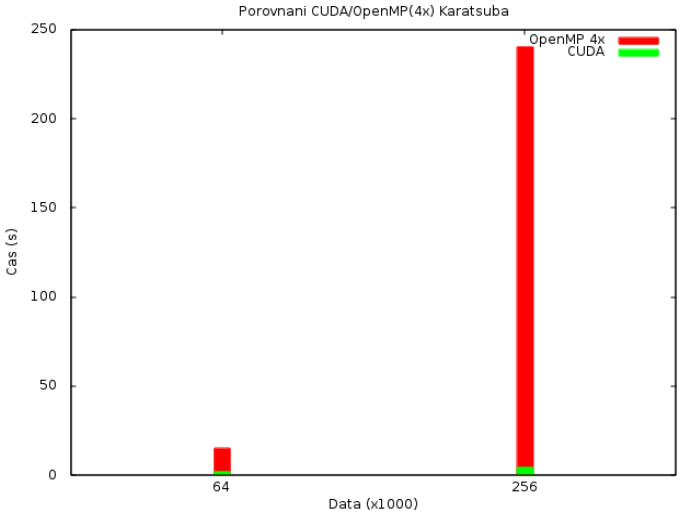
Výsledky OpenMP

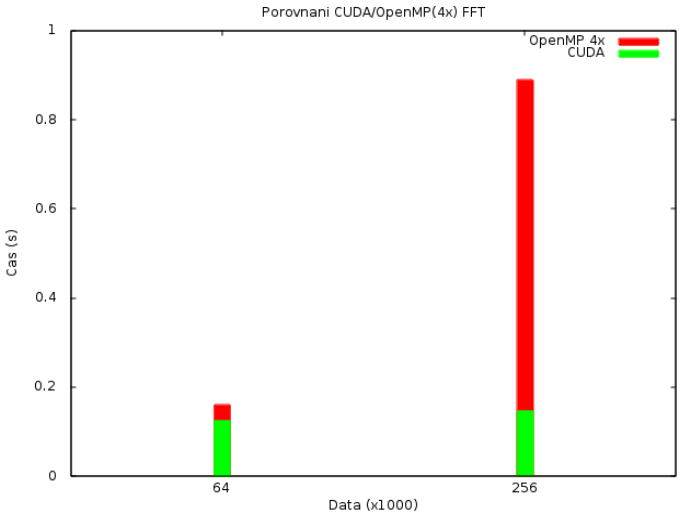


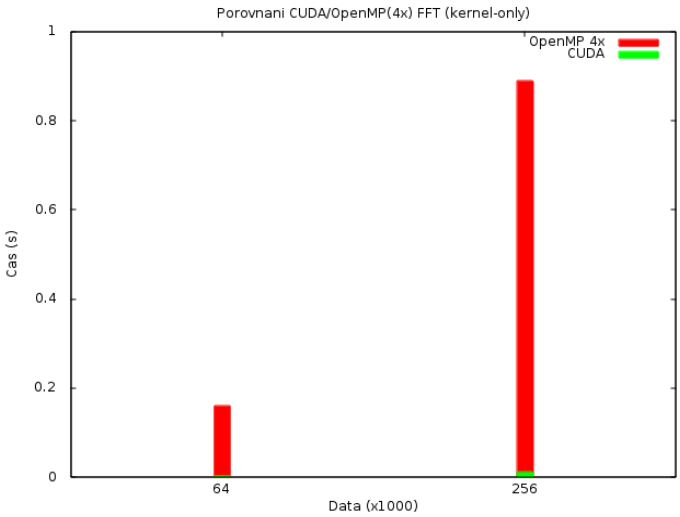




Výsledky CUDA







Výsledky OpenMP