Obecné výpočty na grafických kartách – použitelnost, vlastnosti, praktické zkušenosti

Martin Kruliš, Jakub Yaghob

KSI MFF UK

Malostranské nám. 25, Praha

{krulis,yaghob}@ksi.mff.cuni.cz

Abstrakt. Nedávný pokrok ve vývoji grafických čipů umožnil provádět na běžných grafických kartách obecné výpočty. V kombinaci s univerzální platformou OpenCL, jejíž první implementace pro GPU byly uvedeny v druhé polovině roku 2009, slibují tyto technologie značné zrychlení paralelizovatelných úloh. Tento článek se zabývá použitelností těchto technologií na různé praktické problémy. Zároveň nabídne výkonnostní srovnání prototypových implementací na soudobém hardware.
Klíčová slova: GPGPU, grafické karty, paralelizace

1 Úvod

Jedním z hlavních cílů informatiky jak na poli teoretickém tak praktickém je snaha optimalizovat řešení nejrůznějších problémů za účelem jejich rychlejšího zpracování. Efektivnější algoritmy zvládnou řešit problémy rychleji, případně vyřešit větší problémy ve stejném čase. Paralelizace úloh byla vždy jedním ze slibných směrů optimalizace řešení. V minulosti byl tento přístup výsadou specializovaných výpočetních stanic a sálových počítačů. Díky nedávné revoluci na poli čipů grafických karet je dnes možné z běžně dostupných komponent postavit domácí počítač, který bude mít výpočetní výkon srovnatelný s o několik let staršími superpočítači.

Tento článek shrnuje výsledky zkoumání použitelnosti grafických karet pro obecné výpočty nejrůznějších typů. U všech příkladů uvádíme také použitý algoritmus a popisujeme nutné úpravy pro jeho nasazení na GPU. Každý příklad také demonstruje určitý problém, který se může při těchto úpravách projevit, zejména pak otázka latence a propustnosti pamětí.

1.1 Hardware a metodika měření

Všechny příklady byly testovány na grafické kartě ATI Radeon HD 5870 s čipem RV870 a 1GB interní paměti DDR5 taktované na 4800MHz s 256 bitovou sběrnicí. Jádro pracuje na frekvenci 850MHz a obsahuje 1600 5-cestných shader jednotek (což

odpovídá 320 výpočetním jádrům), které běží na stejné frekvenci [1]. Testovací sestavu pohání procesor Intel Core i7 860 obsahující 4 fyzická jádra s technologií Hyperthreading (tzn. 8 logických jader) taktovaná na 2.8GHz. Sestava obsahovala 4GB operační paměti DDR3 na frekvenci 1333MHz a 64-bitový operační systém Windows 7.

Každý test byl opakován desetkrát a výsledné naměřené časy jsou aritmetickým průměrem těchto měření. Jednotlivá měření každého testu se od sebe nelišila o více než 5% od výsledného průměru.

1.2 Osnova

Sekce 2 stručně shrnuje nejdůležitější aspekty architektury GPU a její abstrakce, kterou poskytuje framework OpenCL. Sekce 3 představuje nejjednodušší možný problém – jednoduché SIMD výpočty nad jednorozměrným vektorem čísel. Sekce 4 rozebírá velmi známý problém násobení velkých matic, na kterém demonstruje problémy s přístupem do globální paměti GPU. Sekce 5 navazuje Floyd-Warshallovým algoritmem, který je do jisté míry podobný problému násobení matic. Sekce 6 se věnuje otázce NP problémů a jejich řešení pomocí backtrackingu. Sekce 7 shrnuje poznatky z celého článku.

2 Architektura GPU a OpenCL framework

V této sekci se budeme věnovat nejdůležitějším vlastnostem architektury GPU [1, 2] a frameworku OpenCL [3] pro paralelní výpočty. Neklademe přitom důraz na úplnost, spíše se snažíme zdůraznit aspekty, na které je třeba brát zvláštní ohled při paralelizaci algoritmů.

2.1 Architektura GPU

Architektury GPU a CPU se značně liší v celé řadě ohledů, zejména

- v počtu výpočetních jader,
- ve výpočetním výkonu (zejména v plovoucí desetiné čárce) a
- v přístupu k řešení problému latence pamětí.

GPU má mnohem vyšší počet jader než soudobá CPU, avšak tato jádra mají celou řadu omezení. Jádra jsou seskupena do SMP jednotek[1], které aplikují SIMD[2] výpočetní model. Všechna jádra jedné SMP jednotky mají tedy stejný program a vykonávají vždy tutéž instrukci (pouze nad jinými daty). Například ATI Radeon použitý pro testy obsahuje 320 jader, která jsou seskupena do 20 SMP jednotek po 16 jádrech.

Na jednu SMP jednotku bývá typicky naplánováno více vláken, než kolik má jednotka procesorů, přičemž počet těchto vláken je násobkem počtu procesorů. Všechna tato vlákna (jak běžící tak naplánovaná) běží ve střídavém SIMD režimu – tedy musí

[1] Streaming MultiProcessor Units
[2] Single Instruction Multiple Data

mít stejný program, který ale nevykonávají zcela souběžně, nýbrž se po skupinách střídají na dostupných jádrech. V případě, že právě běžící sada vláken je nucena z nějakého důvodu čekat (např. na načtení dat z globální paměti), plánovač tuto sadu odstaví a mezi tím nechá počítat jinou sadu naplánovaných vláken, aby byly procesory co nejvíce vytíženy. Touto technikou se GPU snaží minimalizovat efekt latence pamětí.

Grafická karta obsahuje několik druhů pamětí:

- *Registry* – velmi rychlá paměť, kterou má k dispozici každé jádro. Nejnovější GPU nabízí až 1024 registrů po 32 bitech.
- *Sdílená paměť* – vyrovnávací paměť SMP jednotky, do které mohou přistupovat všechna vlákna. Pro grafické výpočty funguje zpravidla podobně jako L1 cache na CPU. Tato paměť je stejně rychlá jako registry, ale přístup do ní má jistá omezení (viz dále). Současná GPU mají řádově desítky kB paměti na SMP jednotku (typicky 32 nebo 64 kB).
- *Globální paměť* – je společná pro celý čip (všechny SMP jednotky). Přístup do ní je výrazně pomalejší neboť všechny SMP sdílí paměťovou sběrnici a data z ní nejsou většinou ukládána do žádné (nebo jen velmi malé) vyrovnávací paměti. Současné karty mají stovky MB až jednotky GB této paměti.

Kromě těchto pamětí je potřeba ještě zdůraznit, že při kombinování výpočtů na CPU a GPU jsou data, která chceme použít, typicky umístěna v paměti hostujícího PC (v operační paměti CPU). Proto je potřeba nejprve data přenést do globální paměti grafické karty, než je vůbec možné začít samotný výpočet.

2.2 Spouštění vláken

Framework OpenCL nabízí řadu mechanismů, pomocí kterých je schopen detekovat paralelní hardware a spustit na něm požadovaný kód. Kód je v programu reprezentován zpravidla ve zdrojovém tvaru a před spuštěním je zkompilován přímo pro cílovou architekturu. Funkce, které slouží jako vstupní body vláken, se nazývají *kernely*. Kernel může být spuštěn buď klasickým způsobem (jako jedna instance vlákna), nebo paralelně na více dat.

Při paralelním spuštění jednoho kernelu jsou instance vláken organizovány do jedno až tří rozměrného pole. Svou pozici v poli může kernel zjistit pomocí vestavěných funkcí a tato pozice také jednoznačně určuje, jakou část práce má instance vlákna vykonat. Kromě toho se instance vláken sdružují do skupin. Velikost skupiny musí být v každém rozměru soudělná s velikostí pole vláken. Vlákna v jedné skupině jsou fyzicky mapována na jednu SMP jednotku se všemi výhodami (např. sdílená paměť) a omezeními (např. SIMD režim), která z toho vyplývají.

Všechny operace prováděné na zařízení jsou spravovány tzv. frontou požadavků. Do této fronty se vkládají požadavky na spuštění kernelů, přesuny dat z/do grafické karty a také synchronizační značky a bariéry. Technické detaily si dovolíme z důvodu úspory místa vynechat.

2.3 Správa paměti

Nyní se podíváme na paměť ještě jednou, tentokrát z pohledu architektury frameworku, a zmíníme také některá podstatná omezení. Z hlediska OpenCL dělíme paměť na:

- *Privátní* – paměť vlastní jednotlivým instancím kernelů. Tato paměť v podstatě odpovídá registrům jednotlivých jader.
- *Lokální* – paměť sdílená mezi vlákny v jedné skupině. Tato paměť odpovídá sdílené paměti SMP jednotek.
- *Globální* – paměť sdílená všemi instancemi kernelu, což odpovídá globální paměti GPU.
- *Konstantní* – speciální případ globální paměti určené pouze pro čtení. Díky tomu, že je dopředu deklarováno, že se jedná o konstantní paměť, může být přístup k datům v této paměti optimalizován.

Běžící vlákna si nemohou žádnou paměť alokovat. Veškeré alokace musí být deklarovány před spuštěním kernelu a jednotlivé instance mohou manipulovat pouze s pamětí, na kterou dostanou ukazatele v argumentech kernelu. Z toho vyplývá poměrně zásadní omezení, že jednotlivé instance musí vracet výsledky konstantní velikosti nebo je potřeba velikost výsledku nejprve dopředu spočítat.

Při práci s pamětí GPU musíme mít na paměti ještě dvě důležitá omezení [1, 2]. První omezení se týká načítání dat z globální paměti. Globální paměť je přístupná skrze relativně širokou sběrnici, avšak latence požadavků je poměrně výrazná. Z tohoto důvodu se optimalizují přenosy, při kterých sousední vlákna přenáší sousední bloky paměti (tzv. coalesced load). Pokud alespoň 16 sousedních vláken z jedné skupiny začne ve stejném okamžiku číst nebo zapisovat 16 sousedních bloků paměti (o velikosti 32 bitů) a celý tento blok (64B) je správně zarovnaný, hardware zpracuje tento přesun v jediném požadavku.

Druhé omezení se týká přístupu do lokální (sdílené) paměti. Lokální paměť je rozdělena do tzv. *bank* (na nejnovější architekturách od je těchto bank zpravidla 16). Dvě sousední 32-bit. buňky paměti se nachází v následujících dvou bankách (modulo 16). Pokud dvě jádra v jedné instrukci přistoupí do stejné banky, je tento přístup serializován, čímž dojde ke zpomalení výpočtu. Výjimkou je, pokud všechna vlákna čtou stejnou hodnotu z lokální paměti. Takový přístup je detekován v hardware a data jsou k jádrům přenesena pomocí broadcastu.

3 Jednoduché SIMD výpočty

Jak jsme zmínili v předchozí sekci, architektura grafických procesorů je od základu postavena na SIMD[3] paralelizaci. První příklad proto otestuje výkonnost grafické karty při jednoduchých vektorových výpočtech, které se nejlépe zpracovávají právě na SIMD výpočetním modelu.

[3]Single Instruction Multiple Data

V příkladu vyzkoušíme dvě vektorové operace. Obě mají dvě vstupní pole x a y a výstupní pole z. Výpočet probíhá nezávisle nad všemi prvky pole, tedy pro všechna i z rozsahu pole provedeme $z[i] = op(x[i], y[i])$. První operace pouze vynásobí oba prvky. Druhá operace je o trochu složitější a při výpočtu uplatní také druhou odmocninu a goniometrické funkce:

$$op_2(x[i], y[i]) = \frac{y[i]\sqrt{x[i]}}{x[i]} + x[i]\cos(y[i])$$

Tato operace nemá žádný praktický základ. Byla pouze uměle vytvořena k otestování více druhů operací a složitějších matematických funkcí.

3.1 Naměřené výsledky

Pro účely následujících testů byly použity vektory čísel x a y o velikosti $16M$ (t.j. 2^{24}) 32-bitových reálných čísel (floatů). Kernely obou operací byly spuštěny paralelně pro každý prvek pole, přičemž velikost skupiny byla nastavena na 256 (což je maximum na použitém GPU). Tato hodnota byla vybrána na základě empirických pozorování.

V tabulce 1 jsou shrnuty výsledky měření. Jednotlivé sloupce obsahují srovnání sériového algoritmu, paralelní implementace používající knihovnu Threading Building Blocks [6], implementace OpenCL běžící na CPU a na GPU. Sloupec GPU obsahuje časy celého výpočtu včetně přesunu dat na grafickou kartu a zpět, zatímco GPU* obsahuje pouze časy samotného výpočtu.

	sériový	TBB	CPU	GPU	GPU*
op_1	38	22	65	189	15
op_2	505	130	165	196	21

Tabulka 1: Naměřené časy SIMD výpočtů v milisekundách

Z naměřených výsledků plyne, že GPU zvládne výpočty provést poměrně rychle, avšak úzkým hrdlem celé operace je přenos dat z hlavní paměti do paměti grafické karty a zpět. U všech výpočtů prováděných na GPU musíme tedy nutně započítat dobu potřebnou na přenos dat, případně plánovat více operací za sebou, jinak se triviální vektorové operace nevyplatí paralelizovat na GPU, neboť procesor je zvládne provést ve srovnatelném čase.

4 Násobení matic

Druhým příkladem je často používaný matematický problém – násobení matic. Přestože tato operace není příliš častá v oblasti zpracování dat, pomůže nám demonstrovat další aspekty přenosů dat mezi pamětmi. Tentokrát se zaměříme na přenosy dat mezi globální pamětí grafické karty a lokální (a privátní) pamětí jednotlivých jader. Pro jednoduchost se omezíme na násobení dvou čtvercových matic, jejichž strany mají délku mocniny 2. Rovněž pro jednoduchost předpokládáme, že druhá matice je již transponovaná, abychom lépe využili cache při výpočtu na CPU.

Přestože existuji lepší algoritmy [4], v našem příkladu použijeme klasický algoritmus pracující v čase $\mathcal{O}(N^3)$, kde N je délka strany matice.

4.1 Naivní implementace

Kernel naivní implementace obsahuje pouze nejvíce vnořený cyklus. Zbývající dvě úrovně cyklů jsou nahrazeny vytvořením příslušného počtu vláken. Každý prvek výsledné matice je tedy počítán paralelně a v případě, že bychom měli k dispozici dostatek jader (řádově $\mathcal{O}(N^2)$), mohli bychom dosáhnout asymptotické složitosti $\mathcal{O}(N)$.

```
__kernel void mul_matrix (__global const float *m1,
                          __global const float *m2,
                          __global float *mRes)
{
    int n = get_global_size(0);
    int r = get_global_id(0);
    int c = get_global_id(1);
    float sum = 0;
    for (int i = 0; i < n; ++i)
        sum += m1[r*n + i] * m2[c*n + i];
    mRes[r*n + c] = sum;
}
```

Přestože zvolený algoritmus je velmi dobře paralelizovatelný a na první pohled slibuje výrazné zrychlení, experimentální výsledky naznačují opak. Časy uvedené v tabulce 2 byly naměřeny při násobení matic 32-bitových reálných čísel pro N rovno 1024 a 2048. Vlákna uspořádána do dvourozměrného pole, které přesně kopíruje tvar matice, a spojena do skupin velikosti 16×16.

Matice	sériový	TBB	CPU	GPU
1024×1024	3630	542	319	392
2048×2048	29370	4060	2311	3400

Tabulka 2: Časy násobení matic v milisekundách

Výpočet na GPU, na kterém spolupracovalo 320 jader, byl znatelně pomalejší než výpočet na čtyřjádrovém CPU s hyperthreadingem. V tomto případě ale neleží problém v přenosu dat z operační paměti do paměti grafické karty, neboť tento přenos zabere pouze 20ms v případě $N = 1024$, resp. 70ms v případě $N = 2048$.

Otázku neuspokojivého výkonu našeho řešení zodpoví profilovací nástroj. V následující tabulce jsou uvedeny nejdůležitější hodnoty naměřené při násobení dvou matic velikosti 1024×1024.

Fetch (kolikrát četlo každé vlákno z globální paměti)	2048×
ALUFetchRatio (poměr operací ALU vůči čtení)	2.51%
ALUFetchBusy (podíl operace čtení na celkovém času)	94.37%
FetchUnitStalled (kolik času čekaly fetch jednotky na data)	83.15%

Z naměřených údajů je jasné, že většinu času celého výpočtu se pouze přenášela data z hlavní paměti k jádrům, přičemž se téměř výhradně na data čekalo. Při optimalizaci se tedy zaměříme hlavně na přenosy dat v rámci grafické karty.

4.2 Cache-aware implementace

Vylepšená implementace využívá faktu, že vlákna jsou spouštěna v SIMD režimu po skupinách. Každá tato skupina má k dispozici lokální pamět'[4], kterou může využít jako cache pro data z globální paměti. Při vhodné úpravě algoritmu můžeme docílit výrazného zrychlení, nebot'

- přístup do lokální paměti je stejně rychlý jako přístup do privátní paměti vlákna,
- vlákna mohou spolupracovat při načítání dat (tzv. coalesced load) a
- většina dat je sdílena mezi vlákny, takže postačí jedno načtení těchto dat do lokální paměti místo aby si každé vlákno tato data stahovalo zvlášt'.

Vlákna jsou uspořádána do skupin velikosti 16×16. Při zpracování matice provedou vlákna v každém průchodu hlavním cyklem dva kroky. V prvním kroku kooperativně načtou čtvercové výřezy velikosti 16×16 prvků z násobených matic. Tyto výřezy obsahují prvky řádků resp. sloupců násobených matic, takže vždy jeden řádek resp. sloupec je sdílen 16 vlákny ve skupině. V druhém kroku si každé vlákno přičte následujících 16 součinů prvků k částečnému součtu. Za prvním i druhým krokem následuje synchronizační bariéra, která zajistí, že všechna vlákna vykonávají stejný krok. Další detaily jsou patrné z následujícího kódu kernelu.

```
__kernel void mul_matrix_opt(__global const float *m1,
                             __global const float *m2,
                             __global float *mRes,
                             __local float *tmp1,
                             __local float *tmp2)
{
    int size = get_global_size(0);
    int lsize_x = get_local_size(0);
    int lsize_y = get_local_size(1);
    int block_size = lsize_x * lsize_y;
    int gid_x = get_global_id(0);
    int gid_y = get_global_id(1);
    int lid_x = get_local_id(0);
    int lid_y = get_local_id(1);
```

[4] V případě použité grafické karty AMD Radeon HD5870 je to 32kB.

```
    int offset = lid_y*lsize_x + lid_x;

    float sum = 0;
    for (int i = 0; i < size; i += lsize_x) {
        // Load data to local memory
        tmp1[offset] = m1[gid_y*size + i + lid_x];
        for (int j = 0; j < lsize_x / lsize_y; ++j)
            tmp2[offset + j*block_size] =
                m2[(gid_x + lsize_y*j)*size + i + lid_x];

        barrier(CLK_LOCAL_MEM_FENCE);

        // Add data from block to the sum
        for (int k = 0; k < lsize_x; ++k)
            sum += tmp1[lid_y*lsize_x + k] * tmp2[lid_x*lsize_x + k];

        barrier(CLK_LOCAL_MEM_FENCE);
    }
    mRes[gid_y*size + gid_x] = sum;
}
```

V tabulce 3 jsou opět naměřené výsledky, včetně optimalizované verze pro GPU, která je označena GPU$^+$. Toto řešení již vykazuje výrazné zrychlení – 45× resp. 52× proti sériové verzi a přibližně 4× proti OpenCL verzi spuštěné na všech jádrech CPU.

Matice	sériový	TBB	CPU	GPU	GPU$^+$
1024×1024	3630	542	319	392	81
2048×2048	29370	4060	2311	3400	564

Tabulka 3: Časy násobení matic v milisekundách

5 Floyd-Warshallův algoritmus

Podobnou charakteristiku z hlediska časové složitosti a přístupu do paměti jako mělo násobení matic vykazuje také známý algoritmus na hledání nejkratších cest v grafu. Násobení matic však mohlo paralelizovat vnější dva cykly, díky čemuž se vnitřní cyklus vykonával přímo v kernelu. Floyd-Warshallův algoritmus proti tomu dokáže paralelismu využít pouze u vnitřních dvou cyklů, protože po každém průchodu vnějším cyklem musí dojít k synchronizaci, abychom udrželi integritu dat. OpenCL bohužel neobsahuje globální bariéru pro všechny instance jednoho kernelu a ani její implementace by nebyla příliš efektivní. Proto přesuneme vnější cyklus mimo grafickou kartu a v každém jeho kroku spustíme paralelně kernel řešící vnitřní dva cykly.

Tabulka 4 shrnuje naměřené výsledky, přičemž verze pro GPU již používá obdobnou optimalizaci, jako algoritmus násobení matic. Z výsledků je patrné, že cyklická invokace kernelů je výrazně méně efektivní než provedení cyklu uvnitř kernelu. I přesto

Vrcholů	sériový	TBB	CPU	GPU
1024	1262	1006	5200	289
2048	10600	7775	35800	3285

Tabulka 4: Časy Floyd-Warshallova algoritmu v ms

však podává naše řešení více než uspokojivé výsledky a použití GPU se i v tomto případě více než vyplatí.

6 NP problémy a backtracking

Dalším ukázkovým problémem je backtracking, který zde zastupuje exaktní způsob řešení NP problémů. NP problémy není[5] v současné době možné řešit v polynomiálním čase. Masivní paralelismus je proto jednou z cest, jak alespoň o trochu posunout hranice velikostí problémů, které je možné v rozumné době spočítat.

Jako reprezentanta jsme vybrali NP-úplnou úlohu známou pod názvem Součet podmnožiny, jejíž zadání je následující. Je dána množina $\mathcal{M}$ celých čísel a celé číslo s. Ptáme se, zdali existuje vybraná podmnožina $\mathcal{M}' \subset \mathcal{M}$ taková, že součet všech jejích prvků je právě s.

Princip řešení je velice snadný. Vybereme všechny existující podmnožiny $\mathcal{M}$ (kterých je $2^{|\mathcal{M}|}$) a u každé z nich ověříme, zda nemá součet s. Testy budeme provádět na množině velikosti $|\mathcal{M}| = 30$. Každou podmnožinu identifikujeme 30-bitovým číslem, kde bity odpovídají jednotlivým prvkům, přičemž 1 znamená, že je daný prvek v podmnožině přítomen a 0, že v ní přítomen není.

Každý kernel pak dostane prefix délky 24 bitů, které použije jako pevný základ a pro zbývajících 8 bitů vyzkouší všechny možnosti. Tabulka 5 shrnuje naměřené výsledky.

$\lvert\mathcal{M}\rvert$	sériový	TBB	CPU	GPU
30	6625	1865	3820	595

Tabulka 5: Časy hledání podmnožiny s daným součtem v ms

Při testování jsme si dovolili malé zjednodušení – všechna čísla v množině byla sudá, zatím co hledaný součet s byl lichý, abychom donutili algoritmus projít všechny podmnožiny. Tento přístup ponechává ve vzduchu otázku jak zastavit běžící výpočet, když jedno z vláken nalezne řešení. Vzhledem k množství vláken a nemožnosti efektivní komunikace mezi nimi se jako nejlepší způsob jeví spouštění kernelů po vhodně velkých částech tak, aby se příliš nezvýšila režie, ale zároveň aby se jednotlivé části nepočítali příliš dlouho. Po skončení výpočtu každé části se provede kontrola, zda byl již výsledek nalezen, a pokud ano, výpočet skončí.

[5] Autor se zde přiklání k všeobecně rozšířené domněnce, že P≠NP.

7 Závěr

V tomto článku jsme představili relativně novou architekturu na poli paralelních výpočtů a ověřili její použitelnost na řadě různorodých problémů. Výsledky naznačují, že se jedná o slibnou technologii, avšak její použitelnost je do značné míry omezována nutností v některých případech poměrně rozsáhlých úprav algoritmů a programovacích technik. Zatím co na CPU je programátor často zachráněn přítomností velké cache, na GPU je potřeba přístupy do paměti pečlivě plánovat a optimalizovat.

V budoucí práci bychom se rádi zaměřili na využití GPU při zpracování databázových dotazů. Zejména nás zajímají možnosti nasazení na relační a sémantická data (RDF [5], linked-data). V této souvislosti bychom také rádi prozkoumali možnosti zpracování grafových algoritmů, které by mohli být využity při zpracování indexů síťových dat.

Literatura

[1] ATI Stream Computing – OpenCL Programming Guide. `http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf`.

[2] NVIDIA OpenCL Programming Guide. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf`.

[3] OpenCL Framework Manual. `http://www.khronos.org/opencl/`.

[4] R.P. Brent and STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE. *Algorithms for matrix multiplication*. Citeseer, 1970.

[5] Frank Manola and Eric Miller. RDF Primer, W3C Recommendation, February 2004. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[6] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.