FAKULTA INFORMAČNÍCH TECHNOLOGIÍ, VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Architektura výpočetních systémů (AVS 2019)
Počítačové cvičení č. 2: Optimalizace násobení matic

Gabriel Bordovský (ibordovsky@fit.vutbr.cz)
Filip Kuklis (ikuklis@fit.vutbr.cz)
Kristian Kadlubiak (ikadlubiak@fit.vutbr.cz)

Termín: 21. 10. 2019

1 ÚVOD

Cílem tohoto cvičení je vyzkoušet si optimalizovat násobení dvou matic s ohledem na efektivní přístup do jednotlivých pamětí cache (*cache blocking*). Vaším úkolem je změřit výkon demonstračního kódu pro násobení dvou čtvercových matic a následně tento kód optimalizovat. Veškerá práce bude probíhat na lokálnim PC na Linuxu.

2 PAPI

Performance Application Programming Interface (PAPI) začalo jako přenositelné rozhraní pro přístup k hardwarovým čítačům CPU. V současnosti umožňuje i měření výkonosti CUDA kódu, síťových rozhraní a dalších.
PAPI umožňuje specifikovat části kódu, které mají být profilovány. Soubor *lab2.cpp* již obsahuje potřebné příkazy pro čítání jednotlivých částí demonstračního kódu. Kód je potřeba přeložit s PAPI knihovnou.

```
$ make matmul-papi
$ papi_avail
$ papi_native_avail
```

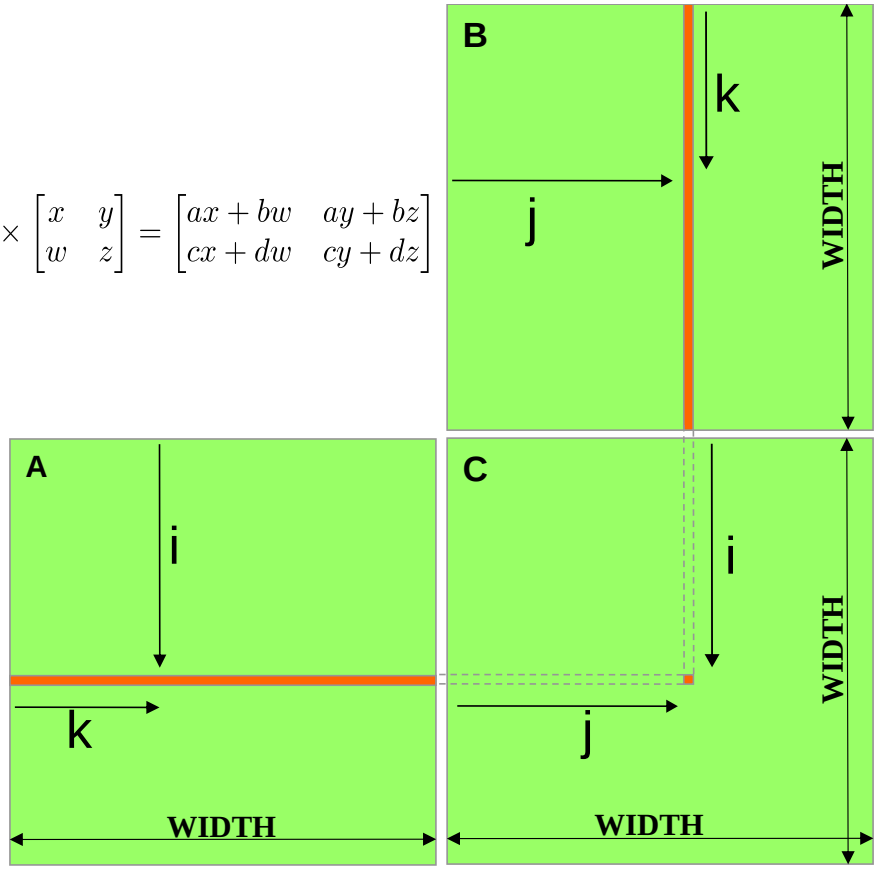
Příkaz, *papi_avail* zobrazuje dostupné eventy. Přiložená knihovna PapiCounters.h umožňuje volit eventy, které budou zjištěny pomocí proměnné prostředí PAPI_EVENTS díky čemuž není potřeba znovu kompilovat pokud je potřeba zjistit stav jiných událostí.

3 Matrix × Matrix

Problém násobení matic je vám jistě známý, a setkáte se s ním ve značném množství vědeckých aplikací. Najivní implementace by mohla vypadat následovně:

```
for (size_t i = 0; i < N; i++){
    for (size_t j = 0; j < N; j++){
        for (size_t k = 0; k < N; k++)
        {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ w & z \end{bmatrix} = \begin{bmatrix} ax+bw & ay+bz \\ cx+dw & cy+dz \end{bmatrix}$$



Tímto způsobem je vždy spočtena jedna buňka výsledné matice C za cenu průchodu celým řádkem i matice A a sloupcem j matice B. Z pohledu vektorizace a práce s cache pamětmi toto není nejvhodnější způsob.

3.1 NAIVNÍ IMPLEMENTACE

Soubor *lab2-mkl.cpp* obsahuje referenční implementaci pomocí Intel Math Kernel Library (MKL[1]). Tato knihovna je vysoce optimalizovaná pro možnosti využití jak paralelního tak vektorového výpočtu. Dosáhnout výkonu této knihovny není cílem tohoto cvičení, může nám ale poskytnout odhad ke kterému se budeme chtít přiblížit.
Pro vygenerování vstupních matic použijeme připravený program *gen*. Ten generuje pseudonáhodné čísla, které zapisuje do textového souboru. Jako argumenty příjmá délku hrany požadované matice a název výstupního souboru.

```
$ make gen
$ ./gen 1000 1k.dat
```

POZOR! Velikost výstupního souboru roste kvadraticky. Matice o hraně 1000 zabírá do 20MB. Matice o hraně 10 000 pak 1.3 GB.
V adresáři se nachází soubor *lab2.cpp* obsahující naivní implementaci násobení matic. Vytvořený program *matmul* přijímá na vstupu čtyři parametry. První udává délku hrany N čtvercových matic. Další dvě jsou názvy vstupních souborů obsahující vstupní matice reprezentované seznamem *float* hodnot oddělených bílými znaky. Poslední parametr je pak soubor který bude přepsán výslednou maticí. Pro měření výkonosti jej můžeme přeložit s modulem *papi*:

```
$ make matmul-papi
$ PAPI_EVENTS="EVENT1|EVENT2" ./matmul-papi 1000 1k.dat 1k.dat 1k.out
```

Úkol 1: změřte procento výpadků v L1 a L2 cache naivní implementace pro matice s délkou hrany 1000.
Vytvořený referenční program *matmul-mkl* má stejné rozhraní jako program *demo*.

```
$ make matmul-mkl
$ PAPI_EVENTS="EVENT1|EVENT2" ./matmul-mkl 1000 1k.dat 1k.dat 1k.out
```

Úkol 2: změřte procento výpadků v L1 a L2 cache MKL implementace pro matice s délkou hrany 1000 a porovnejte je s výsledky naivní implementace.

3.2 OTÁZKY

Jak velký je standartně blok dat v paměti cache? (cache-line)
Kolik paměti zabírá jedna hodnota typu *float*?
Lze procházet matice efektivněji? (Podívejte se na *lab2.optrpt* s *-O2*/*-O3*, a upravte průchod v iteracích.)
Co se změnilo?

[1] https://software.intel.com/en-us/mkl

4 DATA ALIGNMENT A PŘEDNAČÍTÁNÍ KONSTANTNÍCH HODNOT

Data alignment neboli česky zarovnání dat. Znalost, že jsou data zarovnaná, může kompilátor využít ke generování efektivnějšího kódu. Uvažujme blok cache velikosti $4 \times$ *float*. Pokud by na začátku bloku by byl *char*. který zabírá 1/4 datové velikosti *float*, následován čtyřmi *floaty*, poslední by se do bloku nevešel celý, a část by ležela v následujícím bloku. Pokud specifikujeme, že chceme data zarovnaná na velikost *float* bude každý začínat na násobku této velikosti, a nikdy nebude rozděleny mezi dva bloky. Další výhodou zarovnání je skutečnost, že moderní architektury procesorů pracují efektivněji pokud jsou data zarovnána na násobky 64 bajtů.

4.1 MOŽNOSTI ZAROVNÁNÍ

Možnosti jak říct kompilátoru, že data mají být zarovnaná je několik. Existuje snaha o standardizaci (C++11). Pro statickou alokaci lze využít:

```
// GNU Compiler
float A[1000] __attribute__((aligned(64)));

// Intel Compiler
__declspec(align(64)) float A[1000];

//since C/C++ 11
alignas(64) float A[1000];
```

Pro dynamickou lze nahradit funkci *malloc*, připadně i *free*:

```
// POSIX
int posix_memalign(void **memptr, size_t alignment, size_t size);

// Windows
void* _aligned_malloc(size_t size, size_t alignment);

// Intel Compiler
void* _mm_malloc(int size, int align)
void _mm_free(void *p)

//since C/C++ 11
void* aligned_alloc(size_t alignment, size_t size)
```

Kompilátor nedokáže vždy správně identifikovat zda je paměť zarovnaná. Opět existuje několik způsobů jak jej o tom informovat:

```
//GNU Compiler
__builtin_assume_aligned(memptr, size_t alignment);

//Intel Compiler
__assume_aligned(memptr, size_t alignment);

//OpenMP pragma
#pragma omp simd aligned(memptr : aligment)
```

4.2 OTÁZKY

Zarovnejte používaná data a prednačítejte konstantní data v iteracích. Podívejte se do optimalizačního výpisu. Změnilo se něco?

5 CACHE BLOCKING

Další možností pro optimalizaci je rozděleni výpočtů na bloky o velikosti cache. Jedná se o snahu znovu použít již načtená data, předtím než je nahradíme jinými. Zvyušejeme tak aritmetickou intenzitu a limitujeme počet výpadků.
Představme si vektor délky N, který je 12krát větší než L2 cache. Při výpočtu je procházen opakovaně M-krát. Při výpočtu kdy projdeme celý vektor N pro každé M dojde k $M \times 12$ výpadkům. Pokud výpočet otočíme, a každý blok cache projdeme M-krát snížíme počet výpadků na 12.
Tohoto vytváření bloků nelze využít vždy. Je potřeba hlídat si datové závislosti mezi iteracemi. V našem případě aplikovat lze.

5.1 OTÁZKY

Zjistěte velikost jednotlivých cache. (Například pomocí příkazu *lscpu*). Kolik hodnot *float* je možno do jednotlivých cache uložit? Jak by jste rozdělili výpočet matic o délce hrany 3072? Implementujte.