FAKULTA INFORMAČNÍCH TECHNOLOGIÍ, VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Architektury výpočetních systémů (AVS 2019)
Projekt č. 1: Optimalizace sekvenčního kódu

Gabriel Bordovský (ibordovsky@fit.vutbr.cz)

Termín odevzdání: 11. 11. 2019

1 ÚVOD

Cílem tohoto projektu je vyzkoušet si optimalizaci sekvenčního kódu zejména pomocí vektorizace. Vaším úkolem bude optimalizovat výpočet neuronové sítě pro klasifikaci psaných číslic. Pro analýzu výkonnosti je připravena knihovna PAPI umožňující přístup k zabudovaným hardwarovým *performance counterům* uvnitř procesoru. Veškerý kód lze spustit na školním serveru Merlin, v CVT nebo na superpočítači Anselm. Windows v CVT sice nedisponuje přístupem k HW čítačům, je zde ale integrován Intel kompilátor a Intel Advisor do Visual Studia. **Veškeré požadované měření provádějte na výpočetním uzlu superpočítače Anselm.**

2 SUPERPOČÍTAČ ANSELM

Superpočítač Anselm umístěný na VŠB v Ostravě je složen z celkem 209 uzlů, každý uzel disponuje dvěma procesory Intel Xeon, většina (180) uzlů je založena na procesorech Intel Xeon E5-2665. Tyto procesory mají 8 jader s mikroarchitekturou Sandy Bridge, podporují tedy vektorové instrukce AVX. (Počítače v CVT disponují AVX2.) Pro připojení na superpočítač Anselm je potřeba mít vytvořený účet, s kterým je možné se připojit na tzv. čelní (login) uzel – `anselm.it4i.cz`. Tento uzel **neslouží** ke spouštění náročných úloh, veškeré experimenty je nutné provádět na výpočetních uzlech. Tento projekt není výpočetně náročný, přesto by aktivita jiných uživatelů na login uzlu mohla zkreslit měření výkonosti. Pro účely tohoto projektu je nejjednodušším řešením vytvořit *interaktivní úlohu*. Příkaz `qsub` zadá požadavek na spuštění úlohy do fronty, jakmile bude v systému dostatek volných uzlů, dojde ke spuštění úlohy. Parametr `-A` určuje projekt, v rámci kterého máme alokované výpočetní hodiny (neměnit), `-q` určuje frontu, do které bude úloha zařazena (pokud nebude úloha dlouhou dobu

spuštěna, můžete použít frontu qprod, ale preferujte qexp), parametr -l určuje zdroje, které budou úloze přiděleny (počet uzlů, počet procesorů, čas). Abyste předešli zkreslení výkonových statistik, vždy alokujte celý uzel, tj. 16 jader. Interaktivní úlohu pak získáte parametrem -I. Více o spouštění úloh na superpočítačích IT4I naleznete na stránce https://docs.it4i.cz/anselm/job-submission-and-execution/ . Výsledný příkaz pro získání interaktivní úlohy pak vypadá takto:

```
[ibordovsky@login1.anselm ~]$ qsub -A DD-19-32 -q qexp -l select=1:ncpus=16,walltime=1:00:00 -I
```

Software na superpočítači Anselm je dostupný pomocí tzv. *modulů*. Tyto moduly je potřeba před použitím načíst, jak pro kompilaci na login uzlu, tak po každém spuštění interaktivní úlohy. V tomto projektu budou potřeba moduly intel, PAPI, HDF5, CMake a Python :

```
ml intel PAPI HDF5 CMake Python/3.6.1
```

Příkaz ml zajišťuje práci s balíčky software, moduly. Modul intel zahrnuje C/C++ kompilátor firmy Intel, který je možné vyvolat příkazy icc resp. icpc. Kompilátor firmy Intel je vybrán z důvodu čitelnějších optimalizačních výstupů. Dobrovolně můžete porovnat výkon výsledného řešení kompilovaného jiným kompilátorem (například GNU).
Modul PAPI pak obsahuje knihovnu PAPI, která usnadňuje přístup k hardwarovým performance counterům uvnitř procesoru. Každý procesor (jádro) obsahuje několik (4-8) HW registrů, které jsou schopny počítat předem definované události. Mezi typické události, které nás zajímají, patří počet vykonaných FP/INT/LS instrukcí, IPC, počet přístupů do jednotlivých pamětí cache, propustnost paměti, přesnost predikce skoků atd. Knihovna PAPI obsahuje několik pomocných programů (papi_avail, papi_native_avail, papi_mem_info, ...), pomocí kterých je možné zjistit detaily o podpoře na daném procesoru. Pro zjednodušení práce s country je v projektu použita třída Papi, která obaluje knihovnu PAPI. Její definice se nachází v souboru papi_counter.h. Seznam událostí, které chceme měřit, se předává přes proměnnou PAPI_EVENTS.
Modul HDF5 zajišťuje knihovny a nástroje nezbytné k práci se stejnojmenným souborovým formátem, Hierarchical Data Format (HDF). Ten je využíván k ukládání a organizaci značného množství dat. Modul CMake je použit jako systém pro kompilaci projektu a Python pro skripty kontrolující výsledky.

3 Soubory zadání a překlad

Projekt je složen z několika kroků procházejících různé druhy optimalizace výpočtů. Toto členění na kroky by vám mělo pomoci zamyslet se nad dopadem rozdílných implementací na dané architektuře.
V archívu zadání naleznete:

```
ANN-xlogin.txt
CMakeLists.txt
Data/
Scripts/
PapiHeader/
Step0/
```

Soubor `ANN-xloginNN.txt` obsahuje otázky a před-připravené tabulky pro měření. Složka `Step0` obsahuje kostru implementace, po dokončení kroku 0 složku zkopírujete jako `Step1` a budete pokračovat dalším krokem. Složka `Data/` pak obsahuje tři datové soubory ve formátu HDF5. Soubor `network.h5` z kterého získáte váhy natrénované neuronové sítě, soubor `testData.h5` s daty k testování prvotní implementace a soubor `testRefOutput.h5` který obsahuje referenční výstupy pro dataset `testData.h5`. Větší dataset je pak ke stažení, například pomocí wget, na adrese:
`https://www.fit.vutbr.cz/ibordovsky/avs/datasets/bigDataset.h5`.
Ve složce `Scripts/` naleznete jednoduché skripty v jazyce Python3 pro testování výstupů a pro zobrazení specifického obrázku ze vstupního datasetu.
Nahrajte si zdrojové soubory do svého adresáře na Anselmu. Také si vytvořte nový adresář pro sestavení (build) projektu. Konfigurace CMake pracuje s argumentem `STEPS`, který určuje které kroky vaší implementace se budou překládat. Pokud není zadán, použije se "0"a přeložený bude projekt v adresáři `Step0`. Dalším důležitým argumentem je WITH_PAPI specifikující, zda má být překlad proveden s knihovnou PAPI. Pokud chceme získat hodnoty HW counteru, je PAPI potřeba. Pokud program ladíme, nebo chceme se podívat na optimalizační výpis, překládáme bez něj. Optimalizační reporty jsou bez této knihovny značně přehlednější.
Ve složce pro sestavení můžete zavolat příkaz `cmake` s cestou k zdrojovým kódům a těmito argumenty, například pro překlad adresáře `Step0` a `Step1` do složky `build` uvnitř zadání je možné postupovat následovně:

```
~$ cd Assignment
~/Assignment$ mkdir build && cd build
~/Assignment/build$ cmake .. -DCMAKE_BUILD_TYPE=Release -DWITH_PAPI=0 -DSTEPS='0;1'
~/Assignment/build$ make -j
```

Argument `CMAKE_BUILD_TYPE` je **pro měření nutno nastavit na** `Release`, pro ladění můžete použít i hodnoty `Debug` resp. `RelWithDebInfo`. Po překladu naleznete výstupy v složkách `Step0`, `Step1` atp. Spuštění s testovacím datasetem může pak vypadat následovně:

```
~/Assignment/build$ ./Step0/ANN ../Data/network.h5 ../Data/testData.h5 ./Step0/output.h5
```

Kromě binárního souboru `ANN` naleznete v adresářích pro jednotlivé kroky i optimalizační reporty `{main|neuron}.cpp.optrpt` za předpokladu, že překládáte kompilátorem Intel bez použití knihovny PAPI.
Výstup v terminálu pro neupravený `Step0` by pak měl vypadat následovně:

```
Compiled without PAPI
10000
Image 0: 0: 0.000000  1: 0.000000  2: 0.000000  3: 0.000000  4: 0.000000  5: 0.000000 ...
Image 1: 0: 0.000000  1: 0.000000  2: 0.000000  3: 0.000000  4: 0.000000  5: 0.000000 ...
Image 2: 0: 0.000000  1: 0.000000  2: 0.000000  3: 0.000000  4: 0.000000  5: 0.000000 ...
Image 3: 0: 0.000000  1: 0.000000  2: 0.000000  3: 0.000000  4: 0.000000  5: 0.000000 ...
Image 4: 0: 0.000000  1: 0.000000  2: 0.000000  3: 0.000000  4: 0.000000  5: 0.000000 ...
```

První řádek vždy informuje zda byla nebo nebyla při kompilaci začleněna knihovna PAPI. Druhý řádek pak sděluje počet testovaných obrázků, velikost vstupního datasetu. Následuje

výpis vah spočtených pro prvních 5 obrázků pro rychlé ověření správné funkce sítě. Zatím jsou všechny číslice ohodnoceny nulou, doplnit funkci neuronů bude váš první úkol.
Pokud chce provést nějaké měření, musíme nejdříve přeložit naše soubory s PAPI a zvolit které čítače nás zajímají. Řekněme že chceme znát počet výpadků datové cache L1.
Nejdříve si zkontrolujeme dostupnost daného čítače pomoci příkazu `papi_avail`.

```
~$ papi_avail
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.5.1.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 3300
CPU Min Megahertz        : 1200
...
  PAPI Preset Events
================================================================================
Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
```

Následně při volání programu specifikujeme měřené události prostřednictvím proměnné prostředí PAPI_EVENTS. Pokud chceme zjistit více událostí, oddělujeme je svislici (|), počet registrů je ale omezen a ne všechny události se dají čítat současně.

```
~/Assignment/build$ cmake .. -DCMAKE_BUILD_TYPE=Release -DWITH_PAPI=1 -DSTEPS='0'
~/Assignment/build$ make
~/Assignment/build$ PAPI_EVENTS="PAPI_L1_DCM" \
./Step0/ANN ../Data/network.h5 ../Data/testData.h5 ./Step0/output.h5
Compiled with PAPI
10000
--------------------------------
network :: wall time 0.0287072 s
--------------------------------
THREAD 0   ...  THREAD15     [        TOTAL ]
7369       ...   276         [        20006 ] PAPI_L1_DCM
```

Výpis nyní obsahuje navíc čas výpočtu sítě (`network::wall time`) a stav čítače. Pro účely měření zapisujte hodnoty ze sloupce s nejvyšší hodnotou který není `TOTAL`. (Typicky se bude jednat o `thread0`).

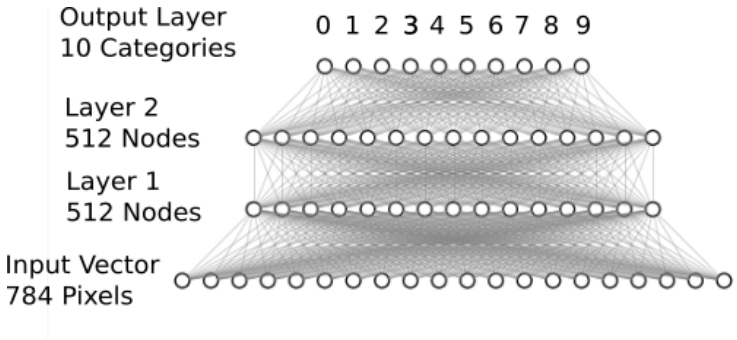
4 Výstup projektu a bodování

Výstupem projektu bude soubor `xlogin00.zip` obsahující všechny zdrojové soubory (StepN) a textový soubor `ANN-xloginNN.txt`. Do tohoto textového souboru zaznamenávejte v každém kroku výkon (čas, FLOPS) daného řešení naměřený pomoci knihovny PAPI pro větší počet obrázků `bigDataset.h5` a zodpovězte požadované otázky. Doporučuji přečíst celý dokument se zadáním a celý soubor pro zaznamenání odpovědí než se pustíte do implementace. V každém souboru (který jste změnili) nezapomeňte uvést svůj login! Hodnotit se bude jak

funkčnost a správnost implementace, tak textový komentář – ten by měl dostatečně popisovat rozdíly mezi jednotlivými kroky a odpovídat na otázky uvedené v zadání. Hodnocení je uvedené u jednotlivých kroků a dohromady tvoří 10 bodů. Projekt odevzdejte v uvedeném termínu do informačního systému.

5 Neuronová síť (10 bodů)

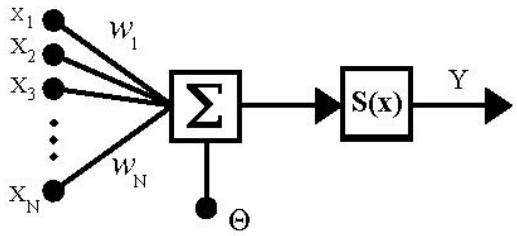
Cílem tohoto projektu bude nejprve implementovat a posléze optimalizovat výpočet neuronové sítě pro klasifikaci rukou psaných číslic. Námi zvolená neuronová síť pochází z článku [1]. Článek popisuje jak tokovou síť sestavit a natrénovat. Pro nás je zajímavé, jak taková síť funguje. Na vstupu obdrží obrázek 28×28 pixelů jako vektor délky 784. Ten je předán první vrstvě která obsahuje 512 neuronů. Výstup této vrstvy je 512 hodnot, které jsou předány druhé vrstvě o stejném počtu neuronů. Poslední, výstupní vrstva obsahuje pouze deset neuronů odpovídající klasifikační množině 0-9.



5.1 Krok 0: základní implementace (2 body)

Ve složce `Step0` naleznete kostru pro implementaci projektu. Seznamte se s obsahem souboru `main.cpp` a `neuron.cpp`. Za zmínku stojí zejména funkce `evaluateLayer` volána tři krát (odpovídá počtu vrstev) pro každý vstupní obrázek ve funkci `main`. Tato funkce přijímá jako své argumenty informace o tom, kolik má vrstva vstupů, kolik má neuronů (výstupů), kde se nachází váhy pro celou vrstvu, prahovou hodnotu, vstupní data a prostor pro zapsání výstupu. Pro každý neuron (výstup) pak volá knihovní funkci `evaluateNeuron`.
Váš první úkol je doplnit tuto funkci v souboru `neuron.cpp`. K tomu potřebujete základní znalost chování umělého neuronu. Jelikož se nejedná o látku tohoto předmětu, bude vysvětlení ve velice zkrácené verzi. Jeden z velmi často používaných modelů neuronu pro použití v neuronových sítích můžete vidět na následujícím obrázku.
Zde uvažujeme že má neuron N vstupů (X_1 až X_N), každý vstup má jinou váhu na fungování neuronu (w_1 až w_n). Dále je zde prahová hodnota Θ, anglicky označována jako bias. (V některých modelech neuronu existuje nultý vstup (X_0) vždy připojen na hodnotu 1, bias je poté jeho váha w_0.) Hodnota neuronu x je poté suma vstupů vynásobenými váhami.

[1] `https://nextjournal.com/gkoehler/digit-recognition-with-keras`



$$x = \left(\sum_{n=1}^{N} w_n \times X_n \right) + \Theta$$

Tato hodnota je pak předána aktivační funkci $S(x)$ která definuje výstup neuronu Y. V našem případě je jako aktivační funkce použitá usměrňovací funkce

$$Y = max(0, x)$$

která vrací hodnotu neuronu pokud je větší nebo rovno 0, a 0 pro záporné hodnoty.
V prvním kroku jsou hodnoty vah `weights` **uspořádaná tak, že identické vstupy všech neuronů vrstvy následují po sobě.** Tzn. první hodnota odpovídá nultému vstupu nultého neuronu, následující hodnota nultému vstupu prvního neuronu. Po vyčerpání všech neuronů následují hodnoty vah pro první vstupy, atd. Jestliže tedy uvažujeme tzv. uložení 2D matice po řádcích (*row major order*) typické pro jazyk C, můžeme tvrdit, že váhy jednotlivých neuronů jsou uloženy po sloupcích.
Během tohoto kroku pracujte pouze s funkcí `evaluateNeuron`. Funkcionalitu řešení můžete ověřit vůči referenčnímu výstupu pomoci scriptu `compareOutputs.py`.

```
Assignment~$ python3 Scripts/compareOutputs.py build/Step0/output.h5 Data/testRefOutput.h5
Maximal error between output and reference is 0.0
OK:Results seems to match the reference
```

Výstup se může lišit s referenčním, v závislosti na pořadí operaci. Zvolená tolerance je 10^{-6}.

5.2 KROK 1: VEKTORIZACE FUNKCÍ (2 BODY)

Jakmile bude implementace funkční, zkopírujte celý adresář `Step0` do nového adresáře `Step1`. Podívejte se na report `main.cpp.optrpt` a na smyčku z funkce `evaluateLayer`. Překladač oznamuje, že něco brání vektorizaci smyčky. Pomocí `#pragma omp simd` je možné vektorizaci vynutit. Opět se podívejte na optimalizační report. Cena smyčky s vektorizaci (*vector cost*) je vyšší než provedení výpočtu bez ní (*scalar cost*). Při měření zjistíte, že doba vykonání opravdu stoupla. Report ovšem navrhuje modifikaci volané funkce. Tuto modifikaci proveďte. (Jedná se o variantu OpenMP pragmy.) Nezapomeňte dodat všechny vhodné dovětky, optimalizační výpis vám opět napoví, jaké zvolit. V tomto kroku pracujte pouze s pragmy OpenMP určenými k vektorizaci (obsahují `simd`).

5.3 Krok 2: přístupy do paměti (3 body)

Zkopírujte celý adresář `Step1` do nového adresáře `Step2`. Jeden z faktorů který se může negativně podepsat na výkonu výpočtu je vzor přístupu do paměti. Je snad základní znalostí, že data jsou organizována v paměti za sebou a přístup po řádcích je vhodnější než po sloupcích. Naše funkce přistupuje k vahám po sloupcích. Transponujte matice vah a upravte volání funkce `evaluateNeuron`. K transpozici máte připravenou funkci, která vytvoří transponovanou kopii matice a uvolní původní paměť. Tuto funkci zavolejte tak, aby byla transpozice započtena v PAPI counterech. Pokud bude matice transponovaná, bude funkce `evaluateNeuron` pracovat s blokem paměti. Zjednodušte rozhraní funkce a její volání ve funkci `evaluateLayer` tak, aby byl předán ukazatel na řádek vah daného neuronu. Nebude již nutné předávat počet ani index neuronu.
Stejně jako v předchozím kroku, bude potřeba modifikovat pragmu nad deklarací funkce přičemž kompilátor bude hledat specifické dovětky. Využijte faktu, že pragmu můžete zapsat nad hlavičku funkce opakovaně, čím donutíte kompilátor vygenerovat několik verzí kódu dané funkce. Ten si pak při linkování vybere tu nejvhodnější.
Úprava těchto pragem ale nemusí mít pozitivní dopad na výkon. Dovětky včetně svého pozorování nezapomeňte popsat do odevzdávaného `ANN-xloginNN.txt` souboru. Odevzdejte tu verzi kroku 2, kterou budete považovat za nejefektivnější vy, nikoli kompilátor.

5.4 Krok 3: správná smyčka a cache (2 body)

Již nyní by měl být patrný značný rozdíl výkonu oproti prvotní implementaci. Následující změny již budou co do výkonu spíše symbolické, ale v složitějších programech mohou být znatelné (a to i výrazně).
Opět začněte kopírováním adresáře `Step2` do nového adresáře `Step3`. Náš kód/přístup byl doposud svým způsobem chybný. Jeden ze základních principů efektivní vektorizace je její aplikaci na nejvnitřnějších smyčkách (zatím co vláknový paralelizmus na nejvnějšnější, ale to je mimo rozsah tohoto projektu). Přesuňte vektorizaci ze smyčky ve funkci `evaluateLayer` do smyčky ve funkci `evaluateNeuron`. Nezapomeňte na správný dovětek ošetřující současnou modifikaci hodnoty ve smyčce. Po otestování funkčnosti opět změřte výsledek.
Kromě toho nemusí být z pohledu cahce optimální vyhodnocování po obrázku, kdy jsou váhy pro výpočet v každé iteraci (pro každý obrázek) střídavě vyměňovány. Upravte alokaci paměti a volání funkce `evaluateLayer` ve funkci `main` tak, aby byly postupně vyhodnoceny všechny obrázky první vrstvou, pak druhou a nakonec třetí. Změřte změnu ve výpadcích L1/L2/L3 cache a výsledky včetně dopadu na rychlost zdokumentujte.

5.5 Krok 4: (ne)volání funkce a zarovnání paměti (1 bod)

Na závěr do posledního kroku `Step4` upravte alokaci ve funkci `allocateMemory` tak, aby byli data zarovnaná na velikost bloku cache. To vynutí také zarovnané načítaní dat do vektorové jednotky, takže se nebude muset vykonávat tzv. *peel loop*. Funkci `evaluateNeuron` ručně inlinujte do těla `evaluateLayer`, čím odstraníte režie volání funkce. Změřte získaný výkon a ohodnoťte jej.