Fakulta informačních technologií, Vysoké učení technické v Brně

Architektury výpočetních systémů (AVS 2020)
Projekt č. 1: Optimalizace sekvenčního kódu

Jakub Budiský (ibudisky@fit.vutbr.cz)

Termín odevzdání: 13. 11. 2020

1 Úvod

Cílem tohoto projektu je najít pomocí profileru úzká hrdla dané aplikace a následně se pokusit pomocí drobných úprav a pokynů pro kompilátor docílit zrychlení pomocí automatické vektorizace. Profilovacím nástrojem bude Intel VTune, profilovanou aplikací referenční implementace AV1 kodéru.
Projekt lze přeložit a spustit prakticky kdekoliv (vyžaduje CMake a kompilátor), budeme se ale spoléhat na optimalizační reporty poskytované Intel kompilátorem. Je žádoucí, aby cílový procesor disponoval podporou vektorového rozšíření AVX2. Referenčním strojem je výpočetní klastr Salomon, na jehož výpočetním uzlu provádějte všechna měření do tohoto projektu. K ladění vašeho řešení můžete využít například i počítače v CVT nebo vlastní počítač, jestliže si nainstalujete nástroje od Intelu (jsou pro studenty zdarma).

2 Superpočítač Salomon

Superpočítač Salomon umístěný na VŠB v Ostravě je složen z celkem 1009 uzlů, každý uzel disponuje dvěma procesory Intel Xeon E5-2680v3. Tyto procesory mají 12 jader s mikroarchitekturou Haswell, podporují tedy vektorové instrukce AVX2. Pro připojení na superpočítač Salomon je potřeba mít vytvořený účet, se kterým je možné se připojit na jeden z čtveřice tzv. čelních (login) uzlů – `login1.salomon.it4i.cz` až `login4.salomon.it4i.cz`.
Login uzly **neslouží** ke spouštění náročných úloh, všechny experimenty je nutné provádět na výpočetních uzlech. Tento projekt sice není výpočetně náročný, přesto by aktivita jiných

uživatelů na login uzlu mohla zkreslit měření výkonosti. Je však možné využít těchto uzlů k prohlížení získaných profilovacích dat a ke kompilaci.

2.1 Přístup na výpočetní klastr a moduly

Přístup k Salomonu je možný výlučně prostřednictvím SSH s použitím Vašeho privátního klíče. V rámci předmětu jste po podepsání souhlasu s podmínkami užívání infrastruktury (letos podepíše na zkoušce) obdrželi přihlašovací údaje, které jste následně ve cvičení použili k získání tohoto klíče. Jestliže klíčem nedisponujete, je možné ho získat na stránce SCS training [1] (po přihlášení). **Klíč pečlivě uschovejte, budete jej potřebovat při každém přihlašování na klastr.**
V projektu budete využívat grafický nástroj VTune, který na dálku funguje nejlépe prostřednictvím vzdálené plochy. Na své pracovní stanici si nainstalujte VNC klienta s podporou tunelování přes SSH a postupujte dle návodu v dokumentaci IT4I [2].
Dle tohoto návodu výše může váš postup na klastru pro vytvoření interaktivní úlohy vypadat například následovně:

```
[ibudisky@local ~]$ ssh salomon.it4i.cz -l dd-20-28-263 -i ~/.ssh/id_rsa-dd-20-28-263
      _____       _
     / ____|     | |
    | (___   __ _| | ___  _ __ ___   ___  _ __
     \___ \ / _` | |/ _ \| '_ ` _ \ / _ \| '_ \
     ____) | (_| | | (_) | | | | | | (_) | | | |
    |_____/ \__,_|_|\___/|_| |_| |_|\___/|_| |_|

                        ...running on CentOS 7.x


[dd-20-28-263@login1.salomon ~]$ vncpasswd
Password:
Verify:
Would you like to enter a view-only password (y/n)? n
A view-only password is not used
[dd-20-28-263@login1.salomon ~]$ ps aux | grep Xvnc | \
    sed -rn 's/(\s) .*Xvnc (\:[0-9]+) .*/\1 \2/p'
...
[dd-20-28-263@login1.salomon ~]$ vncserver :3 -geometry 1600x900 -depth 32
xauth:  file /home/training/dd-20-28-263/.Xauthority does not exist

New 'login1:3 (dd-20-28-263)' desktop is login1:3

Creating default startup script /home/training/dd-20-28-263/.vnc/xstartup
Creating default config /home/training/dd-20-28-263/.vnc/config
Starting applications specified in /home/training/dd-20-28-263/.vnc/xstartup
Log file is /home/training/dd-20-28-263/.vnc/login1:3.log

[dd-20-28-263@login1.salomon ~]$ logout
Connection to salomon.it4i.cz closed.
```

Následně se je možné připojit pomocí VNC klienta. Ten se v příkladu výše musí připojit na `login1.salomon.it4i.cz` (viz hostname v konzoli, na tom konkrétním login uzlu byl puštěn VNC server) na port 5900 + číslo displeje, v příkladě výše 3, t.j. 5903. Připojení musí být

[1] `https://scs.it4i.cz/training`
[2] `https://docs.it4i.cz/general/accessing-the-clusters/graphical-user-interface/vnc/`

prostřednictvím SSH tunelu, protože port 5903 není přístupný mimo síť it4i. Jestliže váš VNC klient nepodporuje tunelování přes SSH, vytvořte si jej dle zmíněného návodu (v GNU/Linuxu pomocí příkazu ssh, ve Windowsech např. prostřednictvím PuTTY). Pak VNC server připojíte k lokálnímu počítači na protunelovaný port.
Po připojení ke vzdálené ploše můžete otevřít emulátor terminálu (menu Aplikace → Systémové nástroje → Terminál) a vytvořit interaktivní úlohu následujícím způsobem:

```
[dd-20-28-263@login1.salomon ~]$ xhost +
access control disabled, clients can connect from any host
[dd-20-28-263@login1.salomon ~]$ qsub -A DD-20-28 -q qexp     \
    -l select=1:ncpus=24,walltime=1:00:00,vtune=2019_update4  \
    -I -v DISPLAY=$(uname -n):$(echo $DISPLAY | cut -d ':' -f 2)
qsub: waiting for job 10185302.isrv5 to start
qsub: job 10185302.isrv5 ready

[dd-20-28-263@r30u18n807.salomon ~]$
```

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 budete na spuštění úlohy čekat příliš dlouho, můžete zkusit frontu `qprod`, ale preferujte `qexp`), parametr `-l` určuje zdroje, které budou úloze přiděleny (počet uzlů, počet procesorů, čas) a další možnosti (v našem případě načtení modulu pro profiler). Interaktivní úlohu pak získáte parametrem `-I`. Pomocí `-v` definujeme proměnnou prostředí `DISPLAY` pro připojení grafických aplikací. Více o spouštění úloh na superpočítačích IT4I naleznete v dokumentaci[3].
Nyní jste již v terminálu připojeni k výpočetnímu uzlu (viz nový hostname v terminálu) s tím, že by jste měli být schopní spustit grafickou aplikaci:

```
[dd-20-28-263@r30u18n807.salomon ~]$ ml VTune/2019_update4
[dd-20-28-263@r30u18n807.salomon ~]$ amplxe-gui
```

Software na superpočítači je dostupný pomocí tzv. *modulů*. Práci s nimi zajišťuje příkaz `ml` (*module load*). Tento příkaz bez parametrů vypíše aktuálně načtené moduly, a všechny moduly specifikované jako parametry se pokusí načíst. Příkaz `ml purge` je všechny odstraní.
Moduly je potřeba načíst po každém přihlášení nebo získání výpočetního uzlu (jsou implementovány proměnnými prostředí). V tomto projektu budou pro překlad (a spouštění) potřeba moduly `intel/2020a` (kompilátor), `CMake/3.16.4-GCCcore-9.3.0` (překladový systém) a pro profilování modul `VTune/2019_update4` (jak bylo již demonstrováno výše). **Jestliže jej potřebujete, modul profileru načtěte jako první!** Závisí na starších knihovnách, které mohou být přepsány novějšími po načtení kompilátoru. Pro profilování kvůli knihovnám musíte mít načtený i modul `intel/2020a`, jinak se vaše přeložena aplikace nespustí.

[3] `https://docs.it4i.cz/anselm/job-submission-and-execution/`

3 Zdrojové soubory, překlad a testování

Cílem této sekce je připravit zdrojové kódy, představit způsob překladu a také soubory pro testování a profilování.
Dle postupu níže nejdříve načtěte moduly, stáhněte si zdrojový repozitář a přepněte se na revizi (tag `v2.0.0`) kterou použijeme pro tento projekt. Vytvořte složku pro kompilaci a spusťte konfiguraci pomocí `cmake`.

```
[dd-20-28-263@login1.salomon ~]$ ml intel/2020a CMake/3.16.4-GCCcore-9.3.0
[dd-20-28-263@login1.salomon ~]$ git clone https://aomedia.googlesource.com/aom
...
[dd-20-28-263@login1.salomon ~]$ cd aom
[dd-20-28-263@login1.salomon aom]$ git checkout v2.0.0
[dd-20-28-263@login1.salomon aom]$ mkdir ../aom-build && cd ../aom-build
[dd-20-28-263@login1.salomon aom-build] cmake ../aom                  \
    -DCMAKE_C_COMPILER=icc -DCMAKE_CXX_COMPILER=icpc                  \
    -DCMAKE_BUILD_TYPE=RelWithDebInfo                                 \
    -DCMAKE_C_FLAGS='-march=native' -DCMAKE_CXX_FLAGS='-march=native' \
    -DCONFIG_INTERNAL_STATS=1                                         \
    -DENABLE_EXAMPLES=1                                               \
    -DAOM_TARGET_CPU=generic                                          \
    -DCONFIG_AV1_DECODER=0 -DCONFIG_MULTITHREAD=0 -DENABLE_DOCS=0
...
```

Použité přepínače konfigurace zabezpečují následující (dle řádků):

1. Volba kompilátoru pro C a C++ (Intel).
2. Základní flagy kompilátoru pro výkonnostní překlad. Ladící symboly jsou potřebné pro profilování.
3. Flag cílové architektury kompilátoru `-march=native`. Kompilátor použije všechny dostupné instrukční sady pro aktuální CPU (včetně AVX2, je-li k dispozici).
4. Interní statistiky obsahující výpis PSNR (*Peak Signal to Noise Ratio*). Ten využijeme pro kontrolu funkčnosti kódu.
5. Příklady zahrnují spustitelný soubor (`aomenc` = *AOM Encoder*) který budeme profilovat. Bez něj se přeloží pouze knihovny.
6. Cílová architektura knihovny je nastavena na „všeobecnou", čím vynutíme nespecifické rutiny v jazyce C.
7. Zbytek voleb je pro urychlení překladu (vypnutí nadbytečných funkcí).

Následuje samotný překlad, který vytvoří kodér `aomenc`:

```
[dd-20-28-263@login1.salomon aom-build]$ make -j
...
[dd-20-28-263@login1.salomon aom-build]$ file aomenc
aomenc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs),
BuildID[sha1]=538e26c32e50c8fd880f1419a00e2c0a5b3441ea,
for GNU/Linux 2.6.32, not stripped
```

Stáhněte a rozbalte si krátká videa, na kterých budete řešení profilovat:

```
[dd-20-28-263@login1.salomon aom-build]$ mkdir ../aom-test && cd ../aom-test
[dd-20-28-263@login1.salomon aom-test]$ wget http://www.fit.vutbr.cz/~ibudisky/avs-videos.tar.gz
...
[dd-20-28-263@login1.salomon aom-test]$ tar -xf avs-videos.tar.gz
[dd-20-28-263@login1.salomon aom-test]$ ls
avs-videos.tar.gz  park_joy_90p_8_444.y4m  park_joy_90p_8_444.y4m.fpf
rush_hour_cut_444.y4m  rush_hour_cut_444.y4m.fpf
```

Nyní máte k dispozici dvě videa a odpovídající `*.fpf` soubory, které byly vygenerované prvním průchodem enkodéru. Nás bude zajímat druhý průchod, který je výpočetně náročnější. Video `park_joy_90p_8_444.y4m` je kratší a vhodné k běhům profileru (čas běhu cca 90 sekund), druhé je delší a vhodné pro závěrečné srovnání (doba běhu cca 30 minut). Výsledek profilování se sice pro oba vstupy mírně liší, ale podstatné informace naleznete v zadání. Spuštění přeloženého enkodéru pro daná videa *na výpočetním uzlu* vypadá následovně:

```
[dd-20-28-263@r30u18n807.salomon aom-test]$ ../aom-build/aomenc park_joy_90p_8_444.y4m   \
    -o park_joy_90p_8_444.webm -p 2 --pass=2 --fpf=park_joy_90p_8_444.y4m.fpf --good     \
    --cpu-used=0 --target-bitrate=200 --auto-alt-ref=1 -v --minsection-pct=0             \
    --maxsection-pct=800 --lag-in-frames=25 --kf-min-dist=0 --kf-max-dist=99999          \
    --static-thresh=0 --min-q=0 --max-q=63 --drop-frame=0 --bias-pct=50                  \
    --minsection-pct=0 --maxsection-pct=800 --psnr --arnr-maxframes=7 --arnr-strength=3
Warning: automatically updating to profile 1 to match input format.
Pass 2/2 frame   10/10      5588B    4470b/f  223500b/s   73322 ms (0.14 fps)
Stream 0 PSNR (Overall/Avg/Y/U/V) 36.971 37.066 34.040 38.868 42.044  223520 bps   73322 ms

[dd-20-28-263@r30u18n807.salomon aom-test]$ ../aom-build/aomenc rush_hour_cut_444.y4m    \
    -o rush_hour_cut_444.webm -p 2 --pass=2 --fpf=rush_hour_cut_444.y4m.fpf --good       \
    --cpu-used=0 --target-bitrate=200 --auto-alt-ref=1 -v --minsection-pct=0             \
    --maxsection-pct=800 --lag-in-frames=25 --kf-min-dist=0 --kf-max-dist=99999          \
    --static-thresh=0 --min-q=0 --max-q=63 --drop-frame=0 --bias-pct=50                  \
    --minsection-pct=0 --maxsection-pct=800 --psnr --arnr-maxframes=7 --arnr-strength=3
Warning: automatically updating to profile 1 to match input format.
Pass 2/2 frame   50/50     58031B    9284b/f  232100b/s 2019413 ms (0.02 fps)
Stream 0 PSNR (Overall/Avg/Y/U/V) 42.529 42.625 39.382 46.039 46.366  232124 bps 2019413 ms
```

Po provedení veškerých úprav v tomto projektu zkontrolujte, zda se hodnoty PSNR příliš nezměnily (nezhoršily, vyšší čísla jsou lepší, ale ve všeobecnosti by se neměly výrazně / vůbec měnit). Jejich hodnoty (a více metrik než na výstupu) naleznete v souboru `opsnr.stt`, který je doplněn po každém spuštění enkodéru.

4 Výstup projektu a bodování

Výstupem projektu bude archiv `xlogin00.zip` obsahující **složky pojmenované dle jednotlivých kroků step1 – step4** přičemž jejich obsahem bude:

- **Ve všech krocích** soubor `repo.patch` obsahující změny v zdrojácích pro splnění aktuálního kroku, získaný pomocí příkazu `git diff > repo.patch` ve složce se zdrojovými soubory (`aom`).

- **Ve všech krocích** budou ve složce také optimalizační reporty `aom_convolve.c.optrpt` a `convolve.c.optrpt`, které po prvním kroku naleznete v kompilační složce[4][5] (a každý krok vygeneruje nové)
- **V kroku 1** screenshot z profileru VTune dle zadání.
- **V krocích 1, 2 a 4** textový soubor nebo soubor ve formátu PDF s odpovědmi na otázky ze zadání.

Hodnocení je uvedené u jednotlivých kroků a dohromady tvoří 15 bodů. Archiv odevzdejte v uvedeném termínu do informačního systému.

5 Optimalizace konvolucí AV1 kodéru pomocí kompilátoru

Při řešení projektu berte prosím na vědomí:

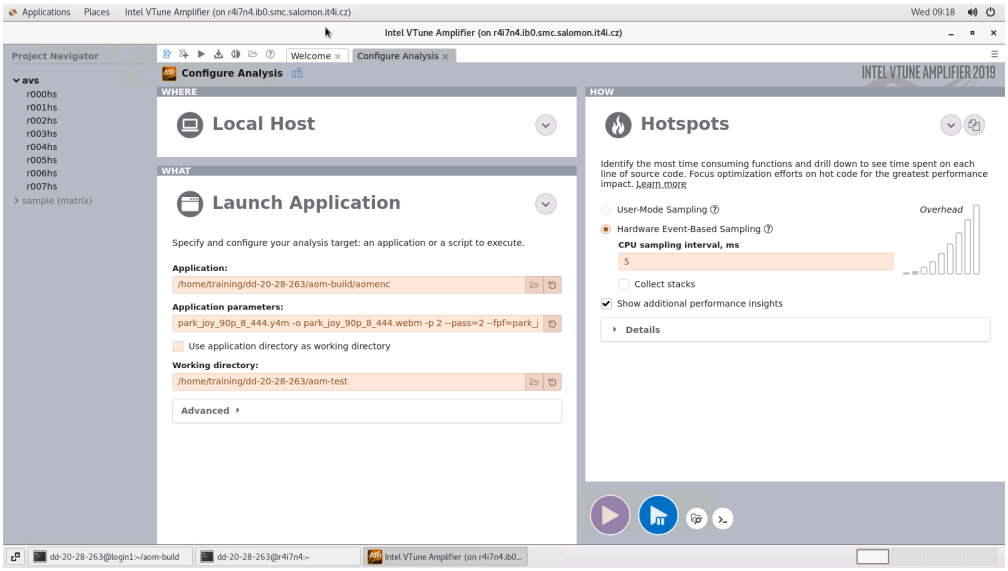
- Děláme black-box optimalizace a o funkčnosti kodéru téměř nic nevíme.
- Soustředíme se na malou část kodéru v části zdrojového kódu pokrytém vybranými testovacími videi.
- Referenční implementace, kterou používáme, není zrovna opěvována pro její skvělý výkon.
- Celkové časy jednotlivých běhů se mohou mezi různými výpočetními uzly lišit. Při posuzování zrychlení se řiďte proto raději časem a podílem (procenty z běhu) vámi optimalizovaných funkcí než celkovým časem. Pro měření celkového zrychlení na závěr projektu je lepší spustit kodér na jednom uzlu (klidně ve stejný čas).

Očekávejte urychlení částí kódu, které budete optimalizovat. Celý proces kódování se vám v rámci projektu nepodaří výrazně urychlit. Koncepty se však neliší od velikosti projektu (z dobrého návrhu profituje celý projekt).

5.1 Krok 1: Profilování a optimalizační výpisy (4 body)

Načtěte moduly a spusťte na výpočetním uzlu VTune (viz výše). Přivítá vás ukázkový projekt násobení matic. V nabídce z tabu *Welcome* pod tlačítkem *Configure Analysis…* klikněte na *New Project…*. Zadejte název a potvrďte.
Následně v novém projektu vyberte *Configure Analysis…*. Otevře se nový tab pro novou analýzu. V levé části nastavte analýzu – zadejte celou cestu k programu `aomenc` (pole *Application*), parametry stejně jako v přikladu spuštění výše (zvolte parametry pro video *park joy*). Nezapomeňte specifikovat pracovní adresář (*Working directory*) na složku obsahující zdrojová videa nebo upravit parametry tak, aby se odkazovali na správné soubory.

[4] `aom-build/CMakeFiles/aom_dsp_common.dir/aom_dsp/aom_convolve.c.optrpt`
[5] `aom-build/CMakeFiles/aom_av1_common.dir/av1/common/convolve.c.optrpt`



V případě, že výběr pomocí dialogu způsobuje pád aplikace je nutné jej napsat ručně. Tento problém pravděpodobně způsobuje nekompatibilita knihoven pro verzi `2019_update` (a novší `VTune/2020` nefunguje na Salomonu správně).
V pravém okně zkontrolujte správný typ měření *Hotspots*, a vyberte *Hardware Event-Based Sampling*. Nastavte *CPU sampling interval* na 1 – 5 ms. Ve spodní části okna klikněte na *Start* (modré kolečko s trojúhelníkem) a nechte proběhnout měření (cca. 90 sekund).
Po ukončení měření si na stránce *Summary* ověřte čas, a projděte si ostatní výsledky. Ve vhodném zobrazení (tabu) s hotspoty udělejte screenshot funkcí běžících nejdelší čas a tento přidejte do adresáře s výsledky.
Všimněte si, že značný počet těchto funkcí se nachází výlučně ve dvou souborech, `convolve.c` a `aom_convolve.c`. Upravte CMake soubory tak, aby k těmto souborům přibyly optimalizační výstupy kompilátoru Intel. Použijte flagy kompilátoru `-qopt-report=4` a `-qopt-report-phase=vec,loop`[6]. Projekt znovu přeložte a prohlédněte si vygenerované výstupy. Jejich umístění naleznete v poznámce u kapitoly 4.
Dotazy a úkoly:

- Jaké je maximální dosažitelné zrychlení celého běhu kodéru, jestli se soustředíme na optimalizaci nejzásadnějších konvolučních funkcí (v *Bottom-up* zobrazení by se mělo jednat o 7 vrchních záznamů)? Výpočet lze v realizovat dle Amdahlova zákona.
- Jaké je dle vás realisticky dosažitelné zrychlení (nebude přihlíženo na kvalitu vašeho odhadu)?
- Ze získaných optimalizačních reportů vypište nejméně 3 příčiny, které brání kompilátoru kód vektorizovat. Nemusíte se omezovat na konkrétní funkce.

Nezapomeňte odevzdat všechny soubory dle pokynů zadání.

[6] Může vám pomoct například CMake funkce `set_source_files_properties`

5.2 Krok 2: (Ne)existence závislosti mezi poli (4 body)

V optimalizačním výpisu jednoho ze souborů z předcházejícího kroku (`aom_convolve.c`) si kompilátor stěžuje na potenciální závislost mezi filtrem a cílovým polem. Neumí totiž odvodit, zda se tato pole nepřekrývají, tzv. pointer aliasing
My můžeme předpokládat, že tento případ nenastane. Je potřeba do zdrojového souboru na vhodné místa přidat pragmy OpenMP SIMD. K tomu, aby jej kompilátor vůbec respektoval, je potřeba přidat flag `-qopenmp-simd`. Upravte CMake tak, že rozšiřte překlad souborů s konvolucí z předcházejícího kroku o tento flag.
Při pohledu „shora" nás zajímají hlavně dvě funkce pro konvoluci (jedna provádí vertikální a druhá horizontální). Do obou z těchto funkcí jsou inlinovány další funkce. Při použití vektorizace je obecně dobrou praxí vektorizovat nejvnitřnější smyčku. Nemusí to být ale nejrychlejší řešení při použití složitějších přístupových vzorů k datům. Je potřeba si uvědomit, které funkce optimalizujeme, jak jsou smyčky zanořeny a jakou práci uvnitř vykonávají. VTune umí rozložit funkci na všechny volané, nacházející se ve stejném souboru. Nemělo by být obtížné je najít.
Smyčky s nízkým počtem iterací je výhodnější rozbalit než vektorizovat (a kompilátor to za vás udělá). Nalezněte nejvhodnější smyčky pro vektorizaci a přidejte OpenMP SIMD pragmu (včetně případných dovětků) do správných funkcí v souboru `aom_convolve.c`. Optimalizačními reporty vám napoví. Ověřte správné umístění výsledkem nového profilování (ve VTune po úpravě klikněte na tab *Welcome* a spusťte novou analýzu pomocí *Configure Analysis*…)
Nezapomeňte ověřit zda výsledný kodér pořád generuje správné výsledky.
Dotazy a úkoly:

- Proč kompilátoru (ve všeobecnosti) vadí překrytí filtru a cílového pole?
- Co komplikuje vektorizaci horizontální konvoluce na úrovni smyčky ve směru x? Co zhoršuje vektorizaci na úrovni smyčky ve směru y?
- Který směr konvoluce (horizontální nebo vertikální) se vám povedlo urychlit (více)? V čem je zásadní rozdíl mezi těmito kernely?

5.3 Krok 3: Lepší krok 2 (2 body)

Nyní se pokusíme o zlepšení zejména horizontální konvoluce z předešlého kroku. K tomu budeme potřebovat kód mírně upravit.
Díky orákulu (nebo studování optimálnější implementace pro architekturu x86) víme, že parametry `x_step_q4` a `y_step_q4` „velmi často" nabývají hodnoty 16. S přihlédnutím na konstanty `SUBPEL_BITS` a `SUBPEL_MASK` dělící indexy `x_q4` resp. `y_q4` k indexování polí je možné konstatovat, že inkrementace indexu `*_q4` o 16 zvyšuje vrchní část indexu (`*_q4 >> SUBPEL_BITS`) o 1, a spodní 4 bity (`*_q4 & SUBPEL_MASK`) nijak nemění.
Rozšiřte funkce z kroku 2, aby podmíněně (`*_step_q4 == (1 << SUBPEL_BITS)`) využily výše popsaného vztahu, a pro tento případ upravte adresování ve funkcích tak, aby bylo lineární a nepoužívalo bitové operace pro „dělení" indexu `x_q4` a `y_q4`.

Díky této úpravě je možné optimálněji umístit pragmy OpenMP pro horizontální konvoluci a dosáhnout lepšího výkonu výsledného kódu.

5.4 Krok 4: `av1_dist_wtd_convolve_2d_c` (5 bodů)

V posledním kroku se podíváme na nejpomalejší konvoluci `av1_dist_wtd_convolve_2d_c` v souboru `aom/av1/common/convolve.c`. Z optimalizačního reportu vyplývá, že smyčky již byly vektorizovány. Nezůstává nám nic jiného, než se rozhlédnout po možných transformacích.
2D konvoluce implementované v tomto souboru jsou separabilní (a oddělené), to znamená že je nejdříve provedena horizontální a pak vertikální. Vybraná funkce je komplikovanější, protože je její chování a ukládání výsledku ovlivněno předanými parametry.
Jestliže umístíte OpenMP SIMD pragmu na některou ze smyček ve směru `x` nebo `y`, kompilátor vám začne napovídat, že jiné pořadí smyček se jeví jako optimálnější. Splňte přání kompilátoru a prohoďte pořadí smyček tak, jak to po Vás žádá.
Několik poznámek k implementaci:

- Požadovaná transformace je netriviální. Pro horizontální konvoluci můžete k akumulaci sumy využít existující `im_block`, přičemž původní konvoluční cyklus musíte rozdělit na „inicializační", „konvoluční" fázi a fázi „uložení výsledků". Přibudou vám proto minimálně dva cykly.

- Vertikální konvoluci transformujte tak, že nejprve vyjmete podmínky z vnitra cyklu ven (podmíníte různé varianty téhož cyklu, který následně zjednodušíte). Pak ho transformujete stejným způsobem jako horizontální konvoluci, jako akumulační buffer si vyrobte „nový" `im_block` **používající datový typ** `int32_t`. „Verzovaný" (podmíněný) nakonec zůstane jenom cyklus implementující fázi „uložení výsledků".

Nezapomeňte zkontrolovat funkčnost výsledného kodéru. Profilovaním ověřte, zda jste zachovali funkčnost kodéru.
Otázky a úkoly pro tenhle krok:

1. Povedlo se vektorizovat všechny vzniklé cykly ve funkci? Proč?

2. Otestujte dosažené zrychlení na videu *rush hour*, aby jste ověřili „přenositelnost" optimalizací na jiné video. Dosáhli jste stejných výsledků jako v případě videa *park joy*?

3. Jaké jsou výhody psaní „vektorizovatelného" a jinak paralelizovatelného kódu a využívaní kompilátoru oproti psaní v jazyce symbolických instrukcí nebo využívání intrinsic funkcí? (Pro inspiraci se podívejte např. do `aom/av1/common/x86/`).