



TELJESÍTMÉNY NÖVELÉS SIMD-UTASÍTÁSKÉSZLET SEGÍTSÉGÉVEL

MILEFF PÉTER

Miskolci Egyetem, Informatikai Intézet

mileff@iit.uni-miskolc.hu

Absztrakt. A számítógépes alkalmazásfejlesztés hosszú múltra tekint vissza. A hardver folyamatos fejlődésével mind a szoftverek, mind pedig a alkalmazások készítését lehetővé tevő programozási nyelvek és keretrendszerek is együtt változtak. Ahogy haladunk az időben előre, úgy váltak elérhetővé az egyre nagyobb kapacitású számítógépek. Ez hatással volt a fejlesztési folyamatra is, hiszen megjelentek a magasabb szintű programozási nyelvek, egyre magasabb szintű API-k váltak elérhetővé, lassan eltűntek a pointerek, memórafoglalások és -felszabadítások is a programozás területéről. A magas szintű nyelvekre való váltással azonban egyre inkább eltűnt a programok optimalizálása vagy annak lehetősége, mivel bizonyos hatékonyságot növelő lehetőségek a magasabb szintű nyelvekben nem, vagy csak korlátozottan elérhetőek. Jelen cikk ezért a hétköznapi programozás körében „elfeledett”, a modern CPU-k által támogatott SSE/AVX SIMD párhuzamosítási lehetőségek adta előnyökkel foglalkozik mélyebben. Bemutatásra kerül, hogy a CPU-ban olyan tartalékok vannak, amelyekkel bizonyos teljesítménykorlátok messze kitolhatók.

Kulcsszavak: SIMD, teljesítmény optimalizáció, SSE, AVX

1. Bevezetés

Egy adott hardverarchitektúra teljesítményének maximalizálása a futó szoftverek precíz optimalizálásával mindig központi területet foglalt el az informatikatudomány történelmében. Természetesen kezdetben, az első személyi számítógépek megjelenésekor ezek a kérdések sokkal égetőbbek voltak, mint manapság, hiszen azok a számítógépek meglehetősen gyenge számítási kapacitással rendelkeztek. A számítógépes játékipar azonban már a kezdetektől lehetőségnek tekintette ezeket és elindult a ma már gigaméreteket öltő játékvilág. Mivel a hardverek nem voltak erősek, GPU sem létezett még, nagyon nagy odafigyelést igényelt egy játék vagy olyan egyéb szoftver elkészítése, amely már nagyobb számításokat szeretett volna végezni. Már rendelkezésre állt a C nyelv, amely akkor még magas szintűnek számított. A programok

nagy része ebben készült a sebességkritikusság miatt. Azonban önmagában ez nem volt elég. A limitált hardver miatt a programozók gépközeli nyelveket használtak a kritikus részek elkészítésére. A számítógépes játékok tipikus mintapéldája volt ennek a vonalnak. A képi megjelenítés motorja, a vektoriális és egyéb kritikus számítások általában Assemblyben készültek, majd később SIMD-utasításkészlettel. Több nyelv erre nagyon jó lehetőséget adott. Képesek voltunk akár néhány sor Assembly kódot egy *for ciklus* belsejébe szűrni a megfelelő sebességért cserébe. Meglehetősen nagyszerű eredmények születtek a precíz és mélyreható optimalizálás, valamint a hardver megfelelő szintű kihasználtsága miatt.

A technológia fejlődésével a hardver egyre erősebb lett. Megjelentek és terjedni kezdtek a magasabb szintű nyelvek: Java, C#, Javascript, Scala, SWIFT stb. Ma már teljesen természetes, ha ezekkel a technológiákkal készítünk alkalmazásokat. A hardver fejlődését tehát követte a szoftver fejlődése is. A magasabb szintű nyelvek növelték az ipari produktivitást. Egységnyi idő alatt gyorsabban készülhettek el a programok, hiszen a programozó már nem feltétlenül a pointerekre fókuszált, vagy a memória felszabadítására, hanem kizárólag az üzleti logikára. Az ipar természetesen nagyon elégedett ezekkel a technológiákkal, hiszen a szoftver gyakran így gyorsabban készül el. Ennek a fejlődési vonalnak a nagy hátránya azonban az, hogy a szoftverek sokszor nem megfelelően optimalizáltak. Sokkal erősebb hardver (gyorsabb CPU, több memória) szükséges hozzá, mint amit valójában igényelne. Mindez azért, mert a magasabb szintű nyelvekkel készített szoftverek egyre több függőséggel rendelkeznek, egyre több a réteg egymáson, valamint sokszor hiányzik az alacsonyabb programozhatóság lehetősége. A hardver erőssége miatt legtöbbször a programozóknak ma már nincs meg az optimalizálási „kényszere”, valamint mivel a magas nyelvekben már nem nyúlhatunk le olyan szintre, amivel megfelelő optimalizációt elvégezhetnénk.

Az úgynevezett SIMD-utasításkészletek már a korai CPU-ban elérhetőek voltak (MMX, SSE), az utóbbi években pedig tovább fejlődött (NEON, AVX, AVX512) a technológia. Kezdetben a játékfejlesztők szent Grálja volt, hiszen az akkori gépek teljesítménye (és a GPU hiánya) önmagában nem volt elégséges egy komolyabb háromdimenziós játék futtatására [14]. Azonban egy sikeres MMX/SSE optimalizációval nagymértékű teljesítménynövekedést lehetett elérni. Ez tette lehetővé az olyan játékok, mint a Doom, Quake, Unreal család, az akkori hardvereken való futás lehetőségét. A mai játékok inkább GPU intenzív programoknak tekinthetőek, a CPU oldalon nem igazán használnak kézi SIMD-alapú optimalizációt. A programozás világában jellemzően magasabb szintű nyelveket (pl. C#) használnak, mert ezekkel a fejlesztés produktivitása jelentősen nőhet. Egyéb esetleg kritikus számítás igénylő programok esetében pedig a több szál alkalmazása az elsődleges megoldási alternatíva.

A SIMD-megoldások direkt használata az átlagos programozó mindennapi munkájából kiszorult. Ennek oka, hogy a SIMD alkalmazása magasabb szintű tudást igényel. A mai fordítók egyre fejlettebbek, képesek a kódot úgy fordítani, hogy azok használják a SIMD bővített utasításkészletet. Bár ez nagyszerű dolog, a legtöbb esetben nem veszi el a versenyt a direkt SIMD kódokkal.

Jelen cikkben a SIMD-alapú párhuzamosítási megoldásokat vizsgáljuk, legfőképpen az SSE és AVX kiterjesztéseket. Egy saját készítésű benchmark csomagon keresztül mutatjuk be a SIMD által nyújtott potenciális sebesség növekedést.

2. Irodalmi áttekintés

A processzor SIMD-típusú utasításkészlet kiterjesztése hosszú múltra tekint vissza. A fejlődés ma is folyamatos, bár nem olyan látványos és médiaközpontú, mint a GPU-k esetben. Az alábbiakban a legfontosabb mérföldköveket ismertetjük.

MMX™ technológia - Intel® Pentium Pentium® with MMX™ és Pentium® II processzorok (1996): 64 bites MMX- (Multimedia Extensions) regiszterek bevezetése az egész típusú SIMD-műveletekhez [13]. Támogatja a SIMD-műveleteket rendezett *byte*, *word* és *double-word* egész számokon. Hasznos multimédiás és kommunikációs szoftverekhez.

3DNOW – AMD (1998): olyan SIMD-utasításokkal egészíti ki az alap x86 utasításkészletet, lehetővé téve a lebegőpontos vektorműveletek vektoros feldolgozását *vector regiszterek* segítségével, ami nagymértékben képes javítani a grafikaintenzív alkalmazások teljesítményét. A 3DNow-t megvalósító első mikroprocesszor az AMD K6-2 volt, amelyet 1998-ban mutattak be. Amennyiben az alkalmazás megfelelően integrálta a technológiát, körülbelül 2-4-szeresére növelte a sebességet [12].

SSE – Intel® Pentium Pentium® III processor (1999): Eredeti neve „Katmai New Instructions” (KNI), mert a Katmai volt az első Pentium III kódjelzése. 128 bites kiterjesztett memóriakezelő (XMM-) regisztereket vezetett be a SIMD egész számokhoz és FP-SP operandusokhoz. Nagy előnye, hogy egyszerre hajtja végre az FP-t és a SIMD-t. Továbbá bevezetésre kerültek olyan utasítások, amelyek lehetővé tették az adatok előzetes betöltését (*data prefetch*). Az SSE-utasításkészlet hasznos 3D geometriához, 3D rendereléshez és videó kódolásához/dekódolásához.

SSE2 – Intel® Pentium Pentium® 4 and Intel 4 and Intel® Xeon processors™ (2000): Extra 64 bites SIMD egész szám támogatás. Ugyanazok az XMM-regiszterek a SIMD egész számokhoz és a lebegőpontos kettős pontosságúhoz (FP-DP). 144 új adattámogatási utasítással rendelkezik (nincs új regiszter). Támogatja a gyorsítótárazhatóságot és a memóriarendezési műveleteket. Az utasításkészlet a 3D-s játékokra, a számítógéppel segített tervezési alkalmazásokra és a videó kódolására/dekódolására összpontosított. Bár az SSE2 akár négy adaton egyszerre tudott műveletet végezni, a teljesítmény nagyjából megegyezett az MMX-szel, amely mindössze két elemben működött. A teljesítményvesztés ilyenkor általában azért következik be, mert az adatok nem megfelelően rendezettek (16 byte-ra) a memóriában.

SSE3 – Intel® Pentium® 4 Processor (2004): A nem megfelelően igazított adatok miatti teljesítményproblémák megoldása érdekében az SSE3 új utasításokat tartalmaz

az ilyen típusú adatok betöltésre, minimalizálva az időzítési „büntetéseket” (*time penalties*). Felgyorsítja a Streaming SIMD Extensions technológia, a Streaming SIMD Extensions 2 technológia és az X87 technológia, valamint az X87-FP matematikai képességek teljesítményét. Hasznos egyes 3D-műveletek (Quaternions) összetett aritmetikai és videokodek algoritmusában.

SSSE3 – Intel® Core® 2 Processor (2006): Az alkalmazás teljesítményének javítása (új utasítások, mint például a szorzás és összeadás és a vektorizálás). Lehetővé teszi a specifikus alkalmazások igényeihez való igazításra.

SSE4 (2007): Az SSE4 54 új utasítást tartalmaz (47 az SSE4.1-ben és 7 az SSE4.2-ben), amelyek a karakterlánc-feldolgozásra szolgálnak. Az új utasítások segítségével a CRC-32 hibaérzékelő folyamat tovább gyorsítható volt. Az SSE4 végleges verziója az SSE4.2.

Az AVX-kiegészítés

Az Intel úgy döntött, hogy a számításokat még szélesebb regiszterekbe helyezi át, és bevezette az *Advanced Vector Extensions* (AVX, más néven *Sandy Bridge New Extensions*) nevű új megoldást. Az AVX az Intel és az AMD mikroprocesszoraihoz készült x86 utasításkészlet architektúra kiterjesztése, amelyet az Intel 2008 márciusában indítványozott. Az első olyan processzor, amely támogatta az AVX-et, az Intel által kiadott Sandy Bridge család volt 2011 első negyedében, majd később pedig az AMD Bulldozer processzorai 2011 harmadik negyedében [7]. Az AVX új funkciókat, új utasításokat és új kódolási sémát kínál. A technológia a következőket tartalmazza:

- 16 darab 256 bites regiszter, YMM0-YMM15
- Három operandus kód írásának képessége assembler alapokon
- A VEX kódolási sémát javasolja, amely megnöveli a műveleti kódok terét
- A VEX előtag hozzáadásával támogatja a régi SSEx-utasításokat is

Az AVX második, 2012-ben kiadott verziója számos egész művelet 256 bites regiszterekre való kiterjesztését tartalmazza. Az AVX2 támogatja a „gather/scatter” műveleteket a regiszterek betöltésére/tárolására a nem szomszédos memóriahelekről/helyekre. Nagy zűrzavart okozott az, hogy az Intel és az AMD is megváltoztatta az SSE5-tel kapcsolatos terveit, az utasításkészletek jövőbeli irányait. Ugyanis a Bulldozer, az AMD legújabb mikroarchitektúrája az XOP- és FMA4-utasításkészleteit valósítja meg, és kompatibilis az AVX-szel is. A Piledriver és a Haswell pedig az AMD és az Intel következő mikroarchitektúrái voltak, amelyek célja további szorzási és összeadási utasítások támogatása mind a lebegőpontos, mind az egész műveletekhez.

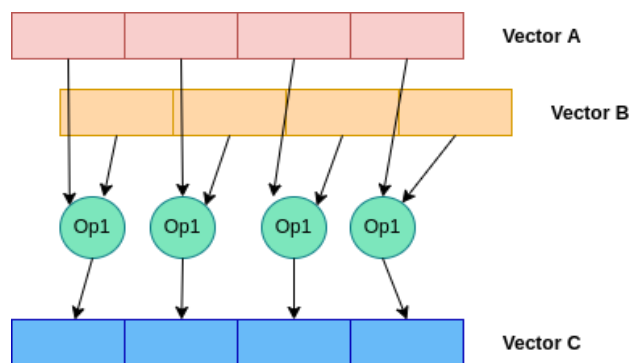
2011-ben az ARM is SIMD-kiterjesztéseket vezetett be az ARM-Cortex architektúráiba a NEON-technológiájukkal. A NEON SIMD-egység 128 bit széles és 16

darab 128 bites regisztert tartalmaz, amelyek 32 és 64 bites regiszterként használhatók. Ezeket a regisztereket azonos adattípusú elemek vektorainak tekinthetjük, melyek az előjeles/előjel nélküli 8, 16, 32, 64 bites és egyetlen precíziós lebegőpontos adattípusok lehetnek.

3. SIMD áttekintése

Az *Single Instruction, Multiple Data* (SIMD) a párhuzamos számítógépek egyik osztálya a Flynn-féle taxonómiában [3]. Több feldolgozó elemmel rendelkező számítógépeket ír le, amelyek ugyanazt a műveletet hajtják végre több adaton egyszerre. Ezzel szemben a hagyományos szekvenciális megközelítést, amely minden egyes adat feldolgozásához egy utasítást használ, skaláris műveleteknek nevezzük. Röviden, a SIMD lehetővé teszi több adatérték egyetlen utasítással való feldolgozását. Az ilyen gépek tehát kihasználják az adatszintű párhuzamosságot, de nem a párhuzamosítást: vannak egyidejű (párhuzamos) számítások, de egy adott pillanatban csak egyetlen folyamat (utasítás) hajtódik végre. A SIMD különösen előnyös olyan feladatoknál, ahol egy nagyobb adathalmazon kell valamilyen számítást végrehajtani. Tipikusan ilyen feladatok találhatók meg a játékokban (vektorok, mátrixok számításai) vagy a multimédiás szoftverekben (képszintézis, képtranszformáció).

Általában egy SIMD-egység bemenetként két vektort kap. Feladata az, hogy ugyanazt a műveletet mindkét vektor adatain végrehajtsa, majd végeredményként egy vektort ad ki [1]. Az 1. ábra egy egyszerű példát mutat be, amelyben egy SIMD-egység négy műveletet hajt végre párhuzamosan:



1. ábra. SIMD-művelet végrehajtás

3.1. SSE-regiszterek

Az SSE-hez 8 darab (64 bites módban 16) XMM-regiszter (XMM0-7(15)) tartozik, melyek 128 bites regiszterek. Egy tipikus 128 bites SIMD-regiszter a következőket tartalmazhatja:

- 16 darab 8 bites egész érték (int8x16 and uint8x16)

- 8 darab 16 bites egész érték (int16x8 and uint16x8)
- 4 darab 32 bites egész érték (int32x4 and uint32x4)
- 4 darab egyszeres pontosságú lebegőpontos érték (float32x4)
- 2 darab dupla pontosságú lebegőpontos érték (float64x2)

SSE adat típusok (16 XMM regiszterek)

__m128	Float	Float	Float	Float	4x 32-bit float										
__m128d	Double		Double		2x 64-bit double										
__m128i	B	B	B	B	B	B	B	B	B	B	B	B	B	16x 8-bit byte	
__m128i	short	short	short	short	short	short	short	short	short	short	short	short	short	8x 16-bit short	
__m128i	int	int	int	int										4x 32bit integer	
__m128i	long long		long long											2x 64bit long	
__m128i	doublequadword														1x 128-bit quad

2. ábra. SSE (128 bites) regiszterek [4]

3.2. Az AVX-kiterjesztés

Az AVX tizenhat YMM-regisztert használ a SIMD-operációk elvégzésére. Minden YMM-regiszter képes egyidejű műveleteket (matematikai) tárolni és végrehajtani: nyolc 32 bites egyszeres pontosságú lebegőpontos számon vagy négy 64 bites duplapontos lebegőpontos számon. A SIMD-regiszterek szélessége 128 bitről 256 bitre nőtt, és átnevezték XMM0–XMM7-ről YMM0–YMM7-re (x86–64 módban, XMM0–XMM15-ről YMM0–YMM15-re) [7].

AVX adat típusok (16 YMM regiszter)

__mm256	Float	Float	Float	Float	Float	Float	Float	Float	Float	8x 32-bit float
__mm256d	Double		Double		Double		Double		4x 64-bit double	
__mm256i	256 bites Integer regiszter. Hasonlóan viselkedik, mint a __m128i. AVX2-ben hasznos									

3. ábra. AVX (256 bites) regiszterek [4]

A régebbi SSE-utasítások továbbra is használhatók. Ezt a VEX-előtag teszi lehetővé, amely az YMM-regiszterek alsó 128 bitét használja a művelet elvégzésére. Az AVX bevezette a VEX kódolási sémának nevezett háromoperandusú SIMD-utasításformátumot, ahol a célregiszter különbözik a két forrás perandustól. Például a hagyományos $a = a + b$ kétoperandusú formát használó SSE-utasítás most már használhat egy háromoperandusú $c = a + b$ formát, megőrizve mindkét forrásoperandust. Eredetileg az AVX háromoperandusos formátuma a SIMD-operandusokkal (YMM)

rendelkező utasításokra korlátozódott, és nem tartalmazott általános célú regiszterekkel (pl. EAX) rendelkező utasításokat. Az említett VEX-kódolást az AVX-512-vel bevezetett k0-k7 maszkregiszterekre vonatkozó utasításokhoz is használják.

A VEX kódolási séma új kódelőtagkészletet vezet be, amely kiterjeszti az opkód teret és lehetővé teszi, hogy az utasítások kettőnél több operandust tartalmazzanak, és lehetővé teszi, hogy a SIMD-vektorregiszterek 128 bitnél hosszabbak legyenek. A VEX-előtag a régi SSE utasításokon is használható, így három operandusból álló formát adva nekik, és hatékonyabbá téve az interakciót az AVX-utasításokkal anélkül, hogy VZEROUPPER- és VZEROALL-utasításokra lenne szükség. Az AVX-utasítások a 128 bites és a 256 bites SIMD-t is támogatják. A 128 bites verziók hasznosak lehetnek a régi kód fejlesztésére anélkül, hogy ki kellene bővíteni a vektorizálást, és elkerülhető az SSE-ről AVX-re való átállás „büntetése”. Emellett gyorsabbak is az AVX egyes korai AMD-megvalósításoknál. Ezt a módot néha AVX-128-nak is nevezik.

4. SIMD használata a gyakorlatban

Az SSE/AVX-képes CPU-k tartalmaznak olyan assembler utasításokat, amelyek lehetővé teszik az XMM- és YMM-regiszterekkel való dolgot. Amennyiben szeretnénk kihasználni ezen lehetőségeket, akkor alkalmazásfejlesztés szempontjából természetesen az elsődleges kapcsolódási pontok a SIMD-képességek kihasználására az alacsony szintű nyelvek és a fordítóprogramok, hiszen általuk készülnek el a megfelelő binárisok. A legtöbb fordító (pl.: GCC, Clang, Intel, Microsoft Visual C++, stb) rendelkezik bizonyos szintű automatikus vektorizálással / SIMD támogatással, amelyek az utóbbi években sokat fejlődtek. A fejlesztő eszközök pedig általában lehetőséget adnak arra, hogy a generált bináris futtatása során beletekinthessünk az adott programrész assembler kódjába, így ellenőrizve a generált kód minőségét.

Természetesen arra is van lehetőség, amennyiben valaki egyedi SIMD kódrészekkel szeretné gyorsítani programjának bizonyos részeit. Az egyik út ilyenkor a közvetlen assembly utasítások alkalmazása, amely ma már túl alacsony szintnek számít a programozók nagy részének. Példaként említhetjük, hogy míg korábban a játék motorok sebességkritikus részei/függvényei assemblyben készültek, ma már nem mennek ilyen szintre le. Ennek természetesen egyik oka, hogy a GPU vette át az egyik legkritikusabb feladatot, a renderelést. A másik út egy könnyebb hozzáférést tesz lehetővé a SIMD-utasításkészlet és -funkciók használatához. A legtöbb fordítóprogram rendelkezik egy úgynevezett „*intrinsic*” függvénykönyvtárral, így a programozóknak nem kell közvetlenül assembler kódot írni, használni. Az *intrinsic* függvénykönyvtárak azért jöttek létre, hogy az assembler utasításokat függvényekkel fedje el és a SIMD-funkciók használata ne különbözzön egy megfelelő paraméterekkel rendelkező függvények meghívásától.

Az *intrinsics* függvények használata nem különbözik bármely más C/C++ alapú függvénykönyvtár használatától. A programozó `include`-álja a használni kívánt

intrinsic típusának megfelelő header fájlt (pl. `#include <xmmintrin.h>`), majd meghívja a kívánt függvényt.

Ahhoz, hogy az SSE/AVX-utasításkészleteket használni tudjuk, rendelkezniünk kell a megfelelő architektúrával. Az SSE/AVX lefordított binárisok csak olyan gépeken képesek futni, amely rendelkezik a megfelelő instrukciócsaláddal. A különböző CPU-k támogatása érdekében a platformspecifikus binárisokat külön, platformspecifikusan kell lefordítani.

Az SSE/AVX intrinsic függvényei a következő elnevezési konvenciót használják (Intel, 2011):

`_<vector_size>_<intrin_op>_<suffix>`

- `<vector_size>`: a regiszter méretét jelző rész. `mm` a 128 bites vectorok számára (SSE), `mm256` a 256 bites vectorok számára (AVX és AVX2), és `mm512` az AVX512 számára.
- `<intrin_op>`: az alkalmazott intrinsic operáció neve. Pl. `add`, `sub`, `mul`, stb.
- `<suffix>`: az adattípusra utaló rész, melyek: `ps` a `float`, `pd` a `double`, és az `ep<int_type>` az integert jelöli. `epi32` az előjeles 32 bites integer, `epu16` az előjel nélküli 16 bites integer, stb.

1. táblázat

SSE/AVX-adattípusok

Utasításkészlet	Float	Double	Integer
SSE	<code>__m128</code>	<code>__m128d</code>	<code>__m128i</code>
AVX	<code>__m256</code>	<code>__m256d</code>	<code>__m256i</code>

Fontos megjegyzés, hogy az XMM és az YMM átfedik egymást. Az XMM-regisztereket a rendszer a megfelelő YMM-regiszter alsó feleként kezeli. Ez bizonyos teljesítményproblémákat okozhat az SSE- és AVX-kódok keverésekor. A programozás szintjén az `__m128i` és `__m256i` unióként kerültek megvalósításra, ezért az adattípusra megfelelően kell hivatkozni. A GCC-fordító lehetővé teszi az adatok tömbként történő elérését, így megkönnyítve a használatukat.

Az *intrinsic* függvények használata függ az alkalmazott fordító architektúráról. Ezért az elérhető függvényekről mindig a fordító dokumentációjából tájékozódhatunk.

4.1. A rendszer képességeinek felismerése

```
#!/bin/bash
#CPU flag detection
echo "****Getting CPU flag capabilities and number of cores"
cat /proc/cpuinfo | egrep "(flags|model name|vendor)" | sort | uniq -c
#Compiler capabilities. -march=native is required!
echo "****Getting GCC capabilities"
gcc -march=native -dM -E - < /dev/null | egrep "SSE|AVX" | sort
```


A fenti sorokkal listázható a CPU-nk tulajdonságai/képességei (*flag*), amely általában egy hosszú lista. Amennyiben gépünk AVX-kompatibilis, úgy a listában szerepelnie kell az AVX flagnek. Ha az AVX2 is megtalálható, akkor a CPU támogatja ezen utasításkészletet is. Az AVX 8x32bit float vectors tárolására képes. Az AVX2 256 bites egész (integer) vektorokkal bővült (például 8x32bit integer). Ennek ellenére úgy tűnik, hogy a 256 bites egész vektorokat ugyanúgy hajtják végre, mint két 128 bites vektort, így a teljesítmény nem javul jelentősen az SSE 128 bites egész vektorokhoz képest. A szkript további részében a GCC-képességek kerülnek lekérdezésre, amelyekből a **#define __AVX__ 1** pragmat kell keresni. Ez azt jelzi, hogy az AVX-ágak engedélyezve lesznek.

Ha a GCC-t a megfelelő *march* flag nélkül futtatjuk, akkor nem kapjuk meg az **__AVX__** képességet, ezért a **-march=native** vagy **-mavx** jelzők használata szükséges. Az alapértelmezett GCC paraméterek általánosak, a megfelelő flag nélkül nem engedélyezi az AVX-et, még akkor sem, ha a CPU AVX-képes.

4.2. Fordító autovektorizáció

A mai GCC- (pl. 13 verzió) fordító már nagyon fejlett. Az **-O3** vagy **-ftree-vectorize** optimalizálási kapcsolókkal a fordító ciklus-vektorizációkat keres (ne felejtjük el megadni a **-mavx** is). A fordítás eredménye során bár a forráskód változatlan marad, de a fordító által összeállított kód teljesen más is lehet.

A GCC nem naplóz semmit az automatikus vektorizálásról, hacsak nincs engedélyezve néhány beállítás. Ha az autovektorizálás eredményeinek részleteire van szükségünk, használhatjuk az alábbi fordítókapcsolókat:

- **-fopt-info-vec** vagy **-fopt-info-vec-optimized**: A fordító loggolni fogja (sor-számmal együtt), hogy mely ciklusok kerültek vektorizálásra.
- **-fopt-info-vec-missed**: Részletes információ azokról a ciklusokról, amelyek nem kerültek vektorizálásra és számos további részletes információ.
- **-fopt-info-vec-note**: Részletes információ minden ciklusról és optimalizálásról, amely megtörtént.
- **-fopt-info-vec-all**: Minden korábbi opció együttesen.

4.3. For ciklus automatikus vektorizálásának kritériuma

Nem minden ciklus vektorizálható. A vektorizálás sikeres végrehajtásához az alábbi követelményeknek kell megfelelnie egy ciklusnak:

- A ciklusszám nem változhat, ha a ciklus elindul. Ez azt jelenti, hogy a ciklus vége lehet egy dinamikus változó, amely tetszés szerint növeli vagy csökkenti az értékét, de ha a ciklus elindul, állandónak kell maradnia.
- A *break* és *continue* használatának korlátai vannak. Néha a fordító elég ügyes ahhoz, hogy elvégezze a vektorizációt, de lesznek olyan esetek, amikor a ciklus nem lesz vektorizálva.
- A cikluson belüli külső függvények meghívásának vannak bizonyos korlátai.

- Nem lehetnek adatfüggőségek a ciklus más indexeivel. Például, ha a ciklus: `for (int i=1; i<N; ++i) x[i]=x[i-1]*2;` amelyet az i változóval járunk be, és az $x[i]$ adat az előző $x[i-1]$ értéktől függ. Mivel az AVX-regiszterek 8 darab *float*ként vannak betöltve, a fordító nem tudja vektorműveletekre konvertálni a számításokat.
- A feltételes mondatok (*if/else*) akkor használhatók, ha nem változtatják meg a vezérlési folyamatot, és csak az A vagy B értékek feltételes betöltésére szolgálnak egy C változóba.

Az autovektorizálás előnye és célja, hogy a fejlesztőnek ne kelljen semmit változtatnia a kódon, és az autovektorizáció segítségével a magasabb szintű képességek kihasználhatók legyenek. Ez sok esetben működik is, azonban (különösen a nagy teljesítményű számítástechnikai alkalmazásokban) a ciklusokat és így a vektorizálást finomhangolni kell, vagy esetleg kézi SSE / AVX-vektorizálással kell biztosítani a maximális átviteli sebességet.

4.4. Hiányzó funkciók az SSE/AVX intrinsicsből

Integerosztás hiánya

Valamilyen oknál fogva az SSE és az AVX nem tartalmaz egész szám osztási operátort. Ennek leküzdésére az alábbi trükköket alkalmazzák:

- Az osztás megvalósítása hagyományos lineáris kódban. A vektorizált kód „megszakítása”, amelyben a normál változó(k)ba hozzuk vissza az adatot, majd elvégezzük az osztást, végül az eredményt visszatöltjük újra vektor(ok)ba. A megoldás sajnos nem gyors.
- Az integervektor floatra konvertálása, az osztály elvégzése, majd ismét integerre konvertálása
- A fordítási időben ismert osztók esetében van néhány olyan szám, amelyekkel az állandóval való osztást szorzássá alakíthatjuk.
- Kettővel való osztás biteltolási művelettel. A 2 egész számmal való osztás ugyanaz, mint a jobbra eltolás. Ez csak akkor lehetséges, ha az összes vektort két szám azonos hatványával osztjuk. Előjeles számok esetében mindig ügyeljünk az előjelbitre.

Trigonometriai függvények hiánya

A vektor *intrinsic* függvényei között nincsenek trigonometrikus függvények. A lehetséges megoldások ilyenkor ezek lineáris kóddal történő kiszámítása (vektorértékenként egyesével), vagy közelítő függvények létrehozása. A Taylor sorozat és a Remez közelítések jó eredményeket adnak, a gyakorlatban előszeretettel alkalmazzák őket.

4.4.1 Teljesítmény “büntetések”

Adatigazítás

A régebbi CPU-architektúrák nem képesek a vektorizációt használni, ha csak az adat mögötti memória nincs megfelelően a vektor méretéhez igazítva (SSE esetén 16 byte-ra). Bizonyos CPU-k képesek nem igazított adatokkal is dolgozni. Ez a működés azonban teljesítménycsökkenéssel, „büntetéssel” jár. A legújabb processzorokban a büntetés mértéke már elhanyagolhatónak tűnik, de a biztonság kedvéért jó ötlet lehet az adatokat továbbra is igazítani, ha nem jár túl sok munkával.

A GCC-ben az adatok igazítása ezekkel a változó attribútumokkal végezhető el: `__attribute__((aligned(16))) __attribute__((aligned(32)))`

SSE és AVX közötti átmenet

Egy további nagy probléma a régebbi SSE-könyvtárak és az új AVX-architektúra keverése. Mivel az XMM és az YMM az alsó 128 biten osztozik, az AVX és az SSE közötti átmenet meghatározatlan értékekhez vezethet a felső 128 biten. Ennek megoldásához a fordítónak el kell mentenie a felső 128 bitet, törölnie kell, végre kell hajtania az SSE-műveletet, majd vissza kell állítania a régi értéket.

Ez észrevehető többletköltséget jelent az AVX-műveleteknél, amely csökkenti a teljesítményt. Ez a probléma nem jelenti azt, hogy ne használhatnánk az `__m128`-at és az `__m256`-ot egyszerre teljesítménycsökkenés nélkül. Az AVX új utasításkészlettel rendelkezik az `__m128`-hoz, VEX-előtagokkal. Ezeknek az új VEX-utasításoknak nem okoz problémát az `__m256` utasításokkal való kombinálása. A teljesítménybüntetés akkor jelentkezik, ha a nem VEX `__m128` utasításokat `__m256` utasításokkal kombinálják. Ez akkor fordul elő, ha régi SSE-könyvtárakat használunk, amelyet új AVX-kompatibilis programokhoz linkelünk.

A büntetések elkerülése érdekében a fordító automatikusan hozzáadhat extra hívásokat a kódhoz. A `-mvzeroupper` fordító paraméter lehetővé teszi, hogy fordításidőben `VZEROUPPER` (törli a felső 128 bitet) vagy a `VZEROALL` (törli az összes YMM-regisztert) hívások bekerüljenek a kódba. ettől függetlenül a programozó ezt manuálisan is megteheti. Ha nem használunk külső SSE-könyvtárakat, és biztosak vagyunk benne, hogy minden kódunk VEX-kompatibilis, és engedélyezett AVX-bővítményekkel lett lefordítva, akkor utasíthatjuk a fordítót, hogy kerülje a `VZEROUPPER`-hívások hozzáadását a következő parancsokkal: `-mno-vzeroupper`.

Adatbetöltés, -kiürítés és -keverés

Az adatok oda-vissza mozgatása az AVX-regiszterekből költséges lehet. Bizonyos esetekben, ha vannak lineáris struktúrákban tárolt adatok, ezeknek az adatoknak az AVX-vektorokba való mozgatása, bizonyos műveletek végrehajtása és az adatok visszahozása a regiszterekből költségesebb, mint egyszerű lineáris számítások elvégzése.

Tehát a programozóknak figyelembe kell venni és számolni kell az adatbetöltés és -kivétel költségeivel. Bizonyos esetekben ez szűk keresztmetszetté is válhat.

5. Teszteredmények

Az SIMD-alapú implementáció nagymértékben növelheti egy program számítási hatékonyságát. Természetesen nem lehet univerzális kijelentéseket tenni, hiszen a „nyereség” függ az adott program jellegétől (CPU-intenzív vagy nem), valamint a SIMD-megvalósítás mikéntjétől is. Nem minden számításnak készíthető el ugyanolyan sikerességgel a SIMD-átirata.

Éppen ezért a SIMD-megoldások hatékonyságának tesztelésére olyan alapvető matematikai feladatokat választottunk, amelyek nagy valószínűséggel a legtöbb CPU intenzív alkalmazásban megjelennek. A hatékonyság mérésére négy darab tesztesetet készítettünk elő, melyek három különböző módszerrel lettek elkészítve: hagyományos implementáció (naiv), SSE- és AVX-változatok.

A megvalósított tesztesetek az alábbiak:

- Skaláris szorzás
- Vektor hosszának kiszámítása
- Vektor skalárral való szorzása
- Vektor egy másik vektorral való szorzása

A programok elkészítéséhez a C++ nyelvet és a GCC 11.1.0 fordítót használtuk, a méréseket pedig egy Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz CPU-val végeztük el. Minden tesztesetet 400 000 darab 4 elemű vektoron [pl. Vector4(4,3,5,1)] hajtottuk végre. A táblázatban szereplő végső eredmények öt darab futtatás eredményének átlagaként kerültek megállapításra. A mérések Linux operációs rendszer alatt készültek, melyek eredményei nanoszekundumban értendők.

2. táblázat

Futtatási eredmények

	Naiv	SSE	AVX	Teljesítmény-növekedés – SSE	Teljesítmény-növekedés – AVX
Skaláris szorzat	3467776	2033760	1423108	41,35%	30%
Vektor hosszának számítása	2500097	2128189	1742673	14,87%	18,11%
Vektor szorzása skalárral	4777252	3873024	267869	18,92%	6,92%
Vektor szorzása vektorral	4539375	2920104	1359712	35,67%	53,43%

Az eredmények egyértelműen alátámasztják a SIMD-változatok előnyeit. A teljesítménynövekedés SSE és AVX esetén is számottevő, melyek megfeleltek a várakozásoknak. Hozzá kell tenni azonban, hogy a SIMD-átalakítás, főként az AVX esetében nem minden esetben triviális. Egy sikeres SIMD-átíráshoz megfelelő adatelőkészítés is szükséges. Főleg az AVX esetében, hiszen ott már meglehetősen nagy regiszterekkel dolgozhatunk. Ha azokat nem töltjük fel megfelelően, akkor a sebességnövekedés elmaradhat. Míg az SSE esetében a négyelemű vektor minden eleme „természetesen” belefér a `__m128` regiszterbe, addig az AVX esetében ez már nem mondható el a `__m256` regiszterről. Így az AVX-transzformáció nemcsak formális kód-módosítást igényelt, hanem a ciklusok átstrukturálását is.

Végül nem szabad megfeledkezni arról a tényről sem, hogy a témában elérhető anyagok mennyisége lényegesen kevesebb azok számára, akik ezzel az optimalizálási szinttel szeretnének foglalkozni.

6. Összefoglalás

A valós életben a programozók csak egy kis százaléka találkozik a CPU-kiterjesztéssel. Egy átlagos alkalmazás készítése gyakran nem igényli az ilyen szintű optimalizációt. Mindezek mellett a mai trendeknek megfelelő magasabb szintű nyelveken való alkalmazásfejlesztés nem teszi lehetővé a modern CPU-k kiterjesztett utasításkészletének közvetlen használatát. Természetesen a nyelvek fordítói ezt megpróbálják automatizáltan elvégezni, de emberi intelligencia nélkül sok esetben ez nem végezhető el maradéktalanul. A teljesítmény elmaradhat a lehetséges maximumtól.

Az SSE/AVX/Neon-alapú kiterjesztésekkel nagymértékben javíthatók a számítási teljesítmények. Gondoljunk csak a korai számítógépes játékokra, amelyek elképesztő teljesítményt voltak képesek elérni (csak MMX és SSE-vel) az akkori lassú CPU-kon is. Ennek azonban ára van. A programozási feladat megoldása nehezebb lesz, az SSE/AVX/Neon kód módosítások magasabb programozói tudást és alacsonyabb szintű nyelvet igényelnek, amelyek csökkenthetik a cég produktivitását. Ez pedig üzleti szempontból nem illik a mai pörgő és haszonorientált világba. Ma már elfogadható kompromisszum a vállalatoknak, hogy az adott programhoz erősebb hardvert javasolnak, mintsem jelentős időt töltsenek a szoftverük optimalizálásával.

Irodalomjegyzék

- [1] Cardoso, J. M. P., F. Coutinho, J. G., Diniz, P. C. (2017). *Embedded Computing for High Performance*, Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, pp. 17–56.
- [2] Intel® 64 and IA-32 Architectures Software Developer Manuals. Published on October 12, 2016, updated May 18, 2018. Available: <https://software.intel.com/en-us/articles/intel-sdm> [Megtekintés: 2022. január 6.].
- [3] Padua, D. (2011). *Encyclopedia of Parallel Computing*, Springer-Verlag New York Inc.

-
- [4] *SSE & AVX Vectorization* (2022). <https://www.codingame.com/playgrounds/283/sse-avx-vectorization/sse-and-avx-usage>
 - [5] Marinescu, D. C. (2018). *Cloud Computing, Theory and Practice, Theory and Practice* (Second Edition), Morgan Kaufmann.
 - [6] Intel Corporation (2011). Intel® Advanced Vector Extensions Programming Reference, <http://www.intel.com>.
 - [7] Intel Corporation (2014). *Optimizing Performance with Intel® Advanced Vector Extensions*. WHITE PAPER, Intel® Advanced Vector Extensions Processor Performance.
 - [8] Jeong, H., Kim, S., Lee, W., Myung, S.-H. (2012). Performance of SSE and AVX Instruction Sets. *30th International Symposium on Lattice Field Theory (Lattice 2012)*.
 - [9] Agner. *Software optimization resources*. C++ and assembly. Windows, Linux, BSD, Mac OS X. URL <http://www.agner.org/optimize/>.
 - [10] Amiri, H., Shahbahrami, A. (2020). SIMD programming using Intel vector extensions. *Journal of Parallel and Distributed Computing*, Volume 135, pp. 83–100.
 - [11] Amiri, H., Shahbahrami, A., Pohl, A., Juurlink, B. (2018). Performance evaluation of implicit and explicit SIMDization, *Microprocessors and Microsystems*. Volume 63, pp 158–168.
 - [12] AMD (2000). 3DNow! Technology Manual. Tech. rep.
 - [13] Peleg A., Weiser U. (1996). MMX technology extension to the intel architecture, *IEEE Micro*, 16 (4), pp. 42–50.
 - [14] Mileff, P., Dudra, J. (2014). Advanced 2D Rasterization on Modern CPUs. *Applied Information Science, Engineering and Technology: Selected Topics from the Field of Production Information Engineering and IT for Manufacturing: Theory and Practice*. Topics in Intelligent Engineering and Informatics, Vol. 7, Chapter 5, Springer International publishing, pp. 63–79.