Antti Niemi

Vektorilaskennan historia ja nykytila x86-64-arkkitehtuurin SIMD-ominaisuuksina

Tietotekniikan kandidaatintutkielma

18. toukokuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Antti Niemi

Yhteystiedot: antherni@student.jyu.fi

Ohjaaja: Tuomo Rossi

Työn nimi: Vektorilaskennan historia ja nykytila x86-64-arkkitehtuurin SIMD-ominaisuuksina

Title in English: Vector computing's history and present state as x86-64-architecture's SIMD capabilities

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 25+0

Tiivistelmä: Tutkimuksessa esitellään vektorilaskennan historia sekä nykytila perehtyen eritoten x86-64-arkkitehtuurin SIMD-ominaisuuksiin. Tutkimuksessa myös perehdytään vektorilaskennan teoriaan SIMD-arkkitehtuureissa, sekä esitellään x86-arkkitehtuureista löytyviä vektorilaskentaan liittyviä käskykantalaajennuksia.

Avainsanat: prosessori, x86, x86-64, SIMD, käskykantalaajennus, vektorilaskenta

Abstract: This study introduces vector computing's history and current state by focusing on x86-64 architecture's SIMD capabilities. A survey is also performed of vector computing's theory based on SIMD architecture and vector computing capabilities found from the x86 architecture's instruction set extensions.

Keywords: processor, x86, x86-64, SIMD, instruction set extension, vector computing

Sisällys

1 Johdanto

Tutkimuksen tarkoituksena on perehtyä vektorilaskennan historiaan sekä nykytilaan modernissa x86-64-arkkitehtuurissa, jossa vektorilaskentaa hyödynnetään SIMD-ominaisuuksina (engl. *Single Instruction, Multiple Data*). Modernit x86-64-arkkitehtuurin vektorikäskykantalaajennukset (Luku 4) ovat hyvin merkittävässä roolissa ohjelmakoodin rinnakkaistamisessa ja näin ollen sen nopeuttamisessa, sillä lähes jokainen moderni yleiskäyttöön tehty prosessori sisältää x86-64-arkkitehtuurin vektorikäskykantalaajennuksia. Tämän takia kyseisiin x86-64-arkkitehtuurin SIMD-ominaisuuksiin on tärkeää kiinnittää huomiota. Tutkimuksen pääsisältö on x86-64-arkkitehtuurin SIMD-ominaisuuksien läpikäynti erinäisten käskykantalaajennuksien kautta, joissa on tuotu esille niiden syntyperään vaikuttavia seikkoja sekä niiden tuomia vektorilaskennan ominaisuuksia.

Tutkielman luvussa 2 käydään lävitse vektorilaskennan historiaa sekä kehitystä. Luvussa 3 esitellään vektorilaskennan teoriaa eri SIMD-arkkitehtuuria edustavissa laitteissa. Luvussa 4 esitellään x86-64-arkkitehtuurille kehitettyjä SIMD-ominaisuuksia, joilla voidaan suorittaa vektorilaskentaa. Luvussa 5 esitellään vielä tapoja, kuinka kyseisiä SIMD-ominaisuuksia voidaan hyödyntää x86-64-arkkitehtuureille kääntyvissä ohjelmissa.

2 Vektorilaskennan historia ja nykytila

Tässä luvussa on tarkoitus kartoittaa lyhyesti vektorilaskennan historiaa sekä kehityksen kulkua eri aikakausille sijoittuvien merkittävien laitteistojen kautta. Vektorilaskennan juuret ovat 1960-luvulla kehitellyissä supertietokoneissa: Solomon ja ILLIAC IV. Näissä supertietokoneissa useat prosessointielementit (engl. *processing elements*) laskivat yhdessä rinnakkain.

2.1 Solomon-arkkitehtuuri

Solomon-arkkitehtuuri oli Daniel Slotnickin 1960-luvulla kehittelemä tietokonearkkitehtuuri, jossa suuri määrä prosessointielementtejä laski rinnakkain (Gregory ja McReynolds 1963). Solomon-arkkitehtuuri sisälsi yhden keskusyksikön ja 1024 prosessointielementtiä, jotka pystyivät suorittamaan yksinkertaisia aritmeettis-loogisia operaatioita samalla käskyvirralla usealle eri tietoalkiolle. Täten Solomon-arkkitehtuuri oli ensimmäinen Flynnin taksonomian mukainen SIMD-arkkitehtuuria edustava supertietokone (Flynn 1967). Solomon-arkkitehtuurin prosessointielementit sisälsivät kaksi 4096 bitin kokoista kehystä (engl. *frame*), joista käskyjen operandit luettiin. Operaatioiden suorituksen aikana kehyksissä sijaitsevat operandit välitettiin prosessointielementin aritmeettis-loogiselle yksikölle *sarjassa*, mikä näin ollen asetti operandien maksimikoon yhteen bittiin. Jokaisen prosessointielementin suoritusta ohjattiin yhteisen keskusyksikön kautta (Gregory ja McReynolds 1963).

Solomon-projekti jätettiin toteuttamatta täydessä mittakaavassaan rahoituksen puutteen takia. Sen pohjalta testaukseen tarkoitetut pienoisversiot olivat suurimmillaan sadan prosessointielementin kokoisia (Slotnick 1982). Vaikka Solomon projekti ei menestynyt ja sen kehitys lopetettiin kokonaan vuoteen 1964 mennessä, muodosti se arkkitehtuurisen pohjan *Illiac IV* supertietokoneelle, joka oli aikansa tunnetuimpia ja tehokkaimpia supertietokoneita (Slotnick 1982).

2.2 ILLIAC IV

Illiac IV oli Illinoisin yliopiston hanke, jota rahoitti ARPA (engl. *Advanced Research Projects Agency*). Sen suunnittelu alkoi vuonna 1965 Daniel Slotnickin johdolla ja sen arkkitehtuurisuunnitelma valmistui vuodeksi 1966 (Slotnick 1982). Illiac IV:n alkuperäisten suunnitelmien nojalla olisi sisältänyt neljä keskusyksikköä ja 256 prosessointielementtiä. Prosessointielementit olivat jaettu 64 ryhmiin, joista jokaista ryhmää ohjasi yksi keskusyksikkö. Eräs suurimpia ILLIAC IV:n muutoksia Solomon-arkkitehtuuriin verrattuna oli aritmeettis-loogisen yksikön operandien maksimikoon kasvaminen yhdestä bitistä 64 bittiin. Nämä 64 bitin operandit voitiin myös jaotella vielä pienempiin palasiin, niin että ne pystyivät sisältämään kaksi 32 bitin operandia tai kahdeksan 8 bitin operandia. Näin ollen prosessointielementit pystyivät suorittamaan myös *pakattuja* operaatioita (Barnes ym. 1968).

Alkuperäisestä suunnitelmasta jouduttiin luopumaan kokoonpanossa tapahtuvien viivytysten seurauksena ja laskemaan ILLIAC IV:n prosessointielementtien määrä 256:sta 64:n sekä keskusyksiköiden määrä neljästä yhteen (Slotnick 1982). Vuonna 1971 ILLIAC IV päätettiin siirtää Illinoisista Kaliforniaan NASA:n (engl. *National Aeronautics and Space Administration*) tutkimuskeskukseen yliopiston kampuksella tapahtuvien Vietnamin sotaan liittyvien mielenosoitusten seurauksesta. Vuodeksi 1972 se saatiin siirrettyä NASA:n tutkimuskeskukseen, jossa sen todettiin sisältävän suuren määrän laitteistotason ongelmia, jotka estivät sen toiminnan. NASA:n tekemien lukuisten korjausten jälkeen se saatiin toimintakuntoon vuodeksi 1975, johon mennessä se oli ylittänyt sille varatun 10 miljoonan dollarin budjetin yli 20 miljoonalla dollarilla. Suuren budjetin ylityksen takia ILLIAC IV päätettiin kytkeä ARPANET:iin (engl. *Advanced Research Projects Agency Network*), jolloin useammat organisaatiot pystyivät käyttämään sitä. Kaikesta huolimatta se oli aikansa nopein supertietokone aina vuoteen 1981 asti (Slotnick 1982).

2.3 Vektoriprosessori

Ensimmäiset vektoriprosessoreilla varustetut supertietokoneet olivat vuonna 1972 markkinoille saapuneet CDC:n (Control Data Corporation) kehittämä STAR-100 sekä Texas Instrumentsin ASC (Advanced Scientific Computer). Näistä kumpikaan ei ollut suorituskyvyltään

erityisen hyvä, paitsi tietyn tyyppisissä ongelmissa (ks. Hennessy ja Patterson 2017, Liite M.6.). Suorituskyvyn heikkouteen tärkeimpiä syitä olivat se, että kummatkin näistä suorittivat operaatioita vain ”muistista muistiin”, jossa jokaisen operaation kohdalla jouduttiin tekemään hidas tiedon haku muistista. Tämän etuna kuitenkin oli se, että vektoreiden alkiomäärä pystyi olemaan vaihteleva, eikä näin ollen ollut sidottu rekisterien määrään eikä rekisterin muistin leveyteen. Suorituskykyyn myös vaikutti se, että vaihto vektori- ja skalaarimoodin välillä oli verrattain hidasta (ks. Hennessy ja Patterson 2017, Liite M.6.).

Vuonna 1976 julkaistu Seymour Crayn suunnittelema CRAY-1 vektoriprosessoreihin perustuva supertietokone paikkasi sekä ASC:ssä että STAR-100:ssa olevia heikkouksia. Eräs tärkein muutos oli rekisterien luonti prosessoriin, jossa tietoalkioita tilapäisesti käsiteltiin ennen muistiin tallentamista. CRAY-1 sisälsi kahdeksan vektorirekisteriä, joista jokainen pystyi sisältämään 64 elementtiä. Elementtien maksimikoko oli 64 bittiä. CRAY-1 sisälsi myös muita rekisterejä kuten kahdeksan skalaari- sekä osoiterekisteriä (Russell 1978). Rekisterit nopeuttivat huomattavasti operaatioiden suorittamista STAR-100:n muistista muistiin -operaatioihin verrattuna, sekä mahdollisti *ketjuttamisen*. Ketjuttamisessa edellisen operaation välivaiheen tuottama tulos tallennetaan rekisteriin, josta seuraava operaatio voi käyttää sitä suoraan ilman, että tietoalkiota tarvitsisi hakea muistista erikseen (Russell 1978). CRAY-1 sisälsi 12 suoritusyksikköä (engl. *functional unit*), jotka olivat jaettu neljään ryhmään: Osoite-, skalaari-, vektori- sekä liukulukuryhmiin. Suoritusyksiköistä jokainen pystyi operoimaan samanaikaisesti sekä rinnakkain, mikä edesauttoi käskyjen liukuhihnoitusta 3.1 (Russell 1978).

CRAY-1 myös panosti huomattavasti enemmän skalaarioperaatioihin verrattuna STAR-100:aan ja tarjosi myös siten aikansa nopeimman skalaariprosessorin, mikä oli yksi tärkeimpiä syitä CRAY-1:n suosioon. Yhdessä nämä kaikki tekivät CRAY-1:stä aikansa nopeimman ja suosituimman supertietokoneen, mikä aloitti vektoriprosessorien valtakauden supertietokoneiden kärjessä yhdessä muiden valmistajien (NEC, Fujitsu) kanssa. Vektoriprosessorien valtakausi kesti aina 2000-luvun alkupuoliskolle asti (ks. Hennessy ja Patterson 2017, Liite M.6.).

2.4 Grafiikkasuoritin

Nykyään vektorilaskennan eräs tärkein prosessorityyppi on grafiikkasuoritin (engl. *Graphics processing unit*, lyh. GPU), jonka voidaan ajatella koostuvan massiivisesta määrästä rinnakkain olevia vektoriprosessoreja. Grafiikkasuorittimet tulivat suosioon 1990-luvulla, kun erilaisten graafisten ohjelmien suosio ja tarve lisääntyi. Aluksi grafiikkasuorittimia ei voitu kuitenkaan ohjelmoida, mutta kehittäjien yhä enemmän vaatiessa erityistarpeisiinsa suunniteltuja funktioita suorittimeen joutuivat valmistajat tekemään tähän muutoksen (ks. Hennessy ja Patterson 2017, Liite M.6.). Vuonna 2001 NVIDIA reagoi tähän tarpeeseen julkaistessaan GeForce3-grafiikkasuorittimen. Lindholm, Kilgard ja Moreton 2001 artikkelissaan esittelivät GeForce3:n sisäisen arkkitehtuurin sekä käskykannan, mikä mahdollisti siten kehittäjien omien ohjelmien kääntämisen sekä suorittamisen GPU:ssa. Tämä oli verrattain kankeaa aluksi, koska ohjelmat piti kääntää käyttämään GPU:n grafiikkaprimitiivejä, joita GPU:den tarjoamat ohjelmointirajapinnat (engl. *Application programming interface*) OpenGL ja DirectX tarjosivat. GPU-ohjelmointia onnistuttiin tästäkin huolimatta tekemään onnistuneesti ja eräissä ongelmatyypeissä grafiikkasuorittimet olivat huomattavasti tavallisia prosessoreita tehokkaampia (Kruger ja Westermann 2003).

Suurin ja tärkein muutos ohjelmien suorittamiseen grafiikkasuorittimissa tuli NVIDIA:n kehittämän Tesla-arkkitehtuurin myötä vuonna 2007. Tesla-arkkitehtuuri yhtenäisti ennen erilliset verteksi- sekä pikseliprosessorit sekä julkaisi CUDA-alustan (engl. *Compute Unified Device Architecture*) sekä -ohjelmistorajapinnan. Tähän sisältyi myös CUDA C -kääntäjä, jolla käyttäjät pystyivät kääntämään omia C++- tai C-kielellä toteutettuja ohjelmia suoritettavaksi Tesla-arkkitehtuurin toteuttavissa grafiikkasuorittimissa (Lindholm ym. 2008). Näin ollen grafiikkasuorittimista alettiin puhumaan GPGPU:na (engl. *General-purpose computing on graphics processing units*) ja hyvin monia vektorilaskentaan liittyviä ongelmia alettiin kiihdyttämään grafiikkasuorittimilla. Grafiikkasuorittimet ovatkin tästä asti olleet suosituimpia vektorilaskentaan käytettäviä suoritintyyppejä, eritoten supertietokoneissa (Elster 2021, ks. Taulukko 1.).

3 Vektorilaskennan teoriaa SIMD-arkkitehtuureissa

Vektorilaskennan arkkitehtuurit voidaan jakaa pääpiirteittään kolmen eri laitteiston välillä: Vektoriprosessorien, grafiikkasuorittimien ja modernien prosessorien, joiden vektorilaskenta ominaisuudet perustuvat vektorikäskykantalaajennuksiin (Salapura 2011). Näistä kaikki edustavat SIMD-arkkitehtuuria (Single Instruction, Multiple Data), mutta grafiikkasuorittimet yleensä vielä jaotellaan erillisen SIMT-arkkitehtuurin alle (engl. *Single Instruction, Multiple Threads*) (Lindholm ym. 2008). Pitäydymme vektorilaskennan teoriaa läpikäydessä vektoriprosessoreihin sekä prosessorien vektorikäskykantalaajennuksiin.

3.1 Datatason rinnakkaisuus ja liukuhihnoitus

Vektoriprosessorit sekä vektorikäskykantalaajennukset edustavat datatason rinnakkaisuutta. Vektorin datatason rinnakkaisuudessa sama operaatio suoritetaan usealle eri elementille, joilla ei ole riippuvuuksia keskenään. Tämän takia vektorikäskyjä suorittavat yksiköt voivat olla hyvinkin yksinkertaisia, mikä suoraan verrannollisesti vaikuttaa kehitettävän laitteiston kuluihin (ks. Asanovic 1998, Luku 2.1). Olemassa olevia laitteistoja, kuten yleiskäyttöön tarkoitettuja prosessoreja, on myös hyvin edullista sekä yksinkertaista muokata käyttämään vektorin datatason rinnakkaisuutta. Esimerkiksi MMX-käskykantalaajennuksen kohdalla 64 bitin summain voitiin jakaa niin, että siitä saatiin muodostettua neljä summainta, joista jokainen operoi 16 bitin data-alkioilla. Näin ollen voitiin hyödyntää jo olemassa olevaa laitteistoa minimaalisilla muutoksilla (Amiri ja Shahbahrami 2020).

Eräs tärkeä konsepti, joka koskee prosessoreja, mutta eritoten vektoriprosessoreja on liukuhihnoitus (engl. *pipelining*). Liukuhihnoituksessa prosessorin eri suoritusyksiköt voivat suorittaa eri käskyjä samanaikaisesti yhdessä muiden suoritusyksiköiden kanssa, mikä tehostaa prosessorien suoritusta merkittävästi (ks. Hennessy ja Patterson 2017, Liite C.1.). Esimerkiksi summainyksikkö voi laskea samalla ajanhetkellä kuin rekisterien latausyksikkö lataa muistista rekisteriin. Vektoriprosessorien käskyissä - joissa on määritelty operaatio sekä vektorinpituus - riittää, että vektoriprosessori purkaa (engl. *decode*) käskyn vain kerran saadakseen selville tulevien vektorialkioiden muistialueen. Tämän jälkeen vektoriprosessori

voi liukuhihnoittaa operaatioita: Vektorialkioiden muistista rekisteriin lataamisen sekä niille suoritettavan operaation. Myös normaaleihin skalaarioperaatioihin verrattuna vektorioperaatiot eivät sisällä mahdollisia datan riippuvuussuhteita alkioiden välillä, mikä entisestään helpottaa liukuhihnoitusta.

Tällä liukuhihnoituksella on merkittävä rooli myös vektoriprosessorin operaatioiden suoritusajalle: Ensimmäisen operaation kohdalla latauksessa menee maksimaalinen aika, joka muistista ladattaessa rekisteriin kuluu. Mutta jo toisen operaation kohdalla data voi olla ladattuna jo vektorirekisterin muistiin suoraan, koska sen lataaminen alkoi samalla ajanhetkellä kuin ensimmäiselle vektorirekisterille suoritettiin operaatiota. Usein vektoriprosessorien kohdalla puhutaankin syvistä liukuhihnoituksista, sillä vektoriprosessorien kohdalla yhdellä käskyllä voidaan määrittää operaatioita tuhansille eri vektorialkioille (ks. “Computer Architecture, Lecture 28: SIMD and GPUs” 2010, sivu 22). Nämä syvät liukuhihnoitukset ovatkin eräs merkittävä ero prosessorien skalaarioperaatioihin verrattuna, joissa jokaisen operaation kohdalla joudutaan tekemään kallis operaation dekoodaus sekä mahdollisesti muistista rekisteriin -latausoperaatio. Prosessorit liukuhihnoittavat myös skalaarikäskyjä, mutta ne eivät voi tehdä liukuhihnoitusta yhtä syvästi kuin vektorikäskyissä, jotka pakkaavat mahdollisesti tuhansia toisistaan riippumattomasti suoritettavia operaatioita ja muistista rekisteriin lataamisia yhteen käskyyn (ks. Hennessy ja Patterson 2017, Luku 4.2.).

3.2 Vektorikäskyt ja -rekisterit

Vektorikäskyt yleisessä muodossa koostuvat operandeista, vektorin pituudesta sekä operaatiosta, joka suoritetaan jokaiselle vektorin elementille (ks. Asanovic 1998, Luku 2.2). Operandit edustavat usein vektoreita, ja niiden lukumäärä voi vaihdella. Vektorin pituus ilmaisee, kuinka monelle operandin sisältämälle vektorielementille kyseinen operaatio suoritetaan. Esimerkiksi käsky, joka sisältää kaksi lähdeoperandia (vektorit *A, B*), operaation(*op*) ja tallentaa tuloksen kohdeoperandiin (vektori *C*) saa muodon:

$$C_i = A_i \text{ op } B_i. \tag{3.1}$$

Yllä olevassa kaavassa i edustaa indeksiä, joka voi saada arvoja nollan ja vektorinpituuden väliltä. C, A ja B ovat vektoreita, jotka edustavat operandeja, jossa tulos tallennetaan C-

vektoriin. Suoritettavaa operaatiota - esimerkiksi summausta - vektorien elementeille edustaa kaavassa op (ks. Asanovic 1998, Luku 2.2). Näin ollen yhdellä vektorikäskyllä voidaan määritellä suoritettava operaatio N:lle eri elementille. Tällä on huomattava ero verraten skalaarikäskyihin, joissa N:lle eri elementille täytyy määritellä N käskyä. Täten vektorikäskyt säästävät merkittävästi virtaa sekä lisäävät prosessorin tehokkuutta, koska prosessorin täytyy tulkata käsky vain yhden kerran suorittaakseen useita operaatiota (Salapura 2011). Vektorikäskyt voivat sisältää myös skalaariosan:

$$C_i = S \text{ op } A_i \text{ tai } C_i = A_i \text{ op } S.$$

Yllä olevassa kaavassa kaikki ovat muuten samoin kuin edellisessä (3.1), mutta S edustaa skalaarioperandia, joka ei muutu indeksin perusteella.

Vektorikäskyjen operandit voivat sijaita joko muistissa (kuten STAR-100:ssa) tai rekistereissä (kuten CRAY-1:ssä). Näiden huomattavin ero on se, että muistista muistiin -arkkitehtuurissa jokaisen operaation tulos joudutaan tallentamaan muistiin. Täten mikäli tallennettu tieto on vain välitulos, jollekin toiselle vektorikäskylle tai -operaatiolle, joudutaan se hakemaan muistista uudestaan, mikä on monta kertaluokkaa hitaampaa kuin tiedon hakeminen rekisteristä prosessointiyksikölle (ks. Asanovic 1998, Luku 2.2.1).

Vektoriarkkitehtuurit, jotka sisältävät vektorirekisterejä sisältävät aina erilliset lataus- sekä tallennuskäskyt, joilla ohjelmoija voi määritellä milloinka rekisterissä oleva arvo tulee siirtää muistiin tai ladata muistista rekisteriin (ks. Asanovic 1998, Luku 2.2.1). Tämän lisäksi rekisteriarkkitehtuurit sisältävät usein vektorin tai vektorien välillä tarvittavia elementtien siirtokäskyjä, joilla vektorien elementtien järjestystä voidaan muokata suoraan rekistereissä (ks. Asanovic 1998, Luku 2.2.2).

Vektorirekisterien koko on myös aina sidottu tiettyyn bittimäärään, eikä sitä voi jälkikäteen kasvattaa kuten keskusmuistin määrää. Näin ollen vektorirekisteriä käyttävillä arkkitehtuureilla on aina maksimi vektoripituus, joka määrittää vektorirekisterissä sijaitsevan vektorin elementtien maksimimäärän. Tällä on erityisen tärkeä rooli x86-arkkitehtuurin SIMD-ominaisuuksissa, jossa uusimmat AVX-käskykantalaajennukset sisältävät suurimmillaan 512 bitin kokoisia vektorirekisterejä (Amiri ja Shahbahrami 2020). Näin ollen esimerkiksi AVX:n 512 bitin vektorirekisteri voi sisältää enintään 64 elementtiä, jotka ovat 8 bitin kokoisia. Eräs

tapa ajatella vektorirekisterejä onkin, että sen jokainen vektorielementti muodostaa virtuaalisen prosessorin (ks. Asanovic 1998, Luku 2.2.3).

Vektorirekistereillä on myös muita implikaatioita eritoten x86-arkkitehtuurissa, jossa datan ryhmittymisellä muistiin vektorielementtien koon monikertana on erittäin kriittinen rooli: Mikäli data ei ole ryhmitelty muistiin oikein aiheuttaa se useita muistihakuja eri rekistereihin, jotka pitää vielä tämän jälkeen uudelleenjärjestää rekisterien välillä ennen varsinaisen operaation suorittamista (Eichenberger, Wu ja O'brien 2004).

Vektorikäskykanta kuitenkin tuo merkittäviä hyötyjä skalaarikäskyihin verrattuna hyvin pienillä laitteistomuutoksilla olemassa oleviin arkkitehtuureihin (ks. Asanovic 1998, Luku 2). Vektorikäskykannoilla "voidaan luoda koodia, joka on kompaktia, ilmaisuvoimaista sekä skaalautuvaa" (ks. Asanovic 1998, Luku 2.2.4). Kompaktia, koska yksi käsky sisältää monta operaatiota. Ilmaisuvoimaista, koska yhdestä operaatiosta laitteisto voi päätellä monia tärkeitä asioita, kuten että jokainen operaatio on homogeeninen sekä riippumaton. Skaalautuvaa, koska tehokkuutta voidaan suoraan säädellä rinnakkaisilla laskentayksiköillä (ks. Asanovic 1998, Luku 2.2.4).

4 x86-64-arkkitehtuurin SIMD-ominaisuudet

Tässä luvussa on tarkoitus kuvata eritoten Intelin luomia x86-64-arkkitehtuurin käskykanta-laajennuksia, joilla voidaan suorittaa vektorilaskennan SIMD-operaatioita. Tärkeää on huomata, että x86-64-arkkitehtuuri edustaa eri prosessorien toteuttamaa käskykanta-arkkitehtuuria, eikä kyseessä ole mikroprosessoriarkkitehtuuri. Käskykanta-arkkitehtuuri tarjoaa ohjelmoijalle rajapinnan käskyistä ja toimii myös sopimuksena, jonka erinäiset mikroprosessorit lupaavat pitää, mikäli ne toteuttavat kyseisen käskykanta-arkkitehtuurin. Esimerkiksi käskykanta-arkkitehtuurin tarjoama summauskäsky kahdella eri operandilla *lupaa* laskea operandien summan ja tallentaa sen rekisteriin, mutta käskykanta-arkkitehtuuri *ei ota kantaa*, kuinka mikroprosessori tämän laitteistotasolla tekee. Tässä työssä x86- tai x86-64-arkkitehtuuri tarkoittaa aina käskykanta-arkkitehtuuria.

4.1 MMX

MMX-käskykantalaajennus (MultiMedia eXtension) julkaistiin Intelin Pentium-prosessorille vuonna 1997. Se luotiin nopeuttamaan eritoten multimedia- sekä kommunikaatiosovelluksia, joissa datan sekä operaatioiden yksinkertaisuus mahdollistaa helpon rinnakkaisuuden hyödyntämisen (Thakkur ja Huff 1999). MMX:n rekisterin leveys oli 64 bittiä, jotka fyysisesti sijaitsivat prosessorin x87-käskykannan liukulukurekistereissä. Rekisterien käyttöä eri käskykantojen välillä kutsutaan rekisterin uudelleen nimeämiseksi (engl. *register renaming*). Tämä aiheutti sen, että jokainen x87-käskykannan liukulukuoperaatio saattoi myös muuttaa MMX-rekisterien sisältöä - sekä päinvastoin - ja täten näitä operaatioita ei pystytty suorittamaan samanaikaisesti. Myös näiden rekisterien vaihtaminen x87-käskykantatilaan tai MMX-käskykantatilaan oli verrattain hidas operaatio, mikä heikensi entisestään näiden eri käskykantojen operaatioiden suorittamista ajallisesti lähekkäin ohjelmassa (Hassaballah, Omran ja Mahdy 2008). Yksi tärkeimmistä syistä x87-liukulukurekisterien uudelleen nimeämisessä MMX-rekistereiksi oli, että näin taattiin taaksepäinen yhteensopivuus kaikille IA-32:n (engl. *Intel Architecture*) applikaatioille sekä käyttöjärjestelmille (Peleg ja Weiser 1996).

Rekisterien määrä MMX:ssä oli kahdeksan (nimellisesti MM0 - MM7) sekä tietotyyppinä

toimi vain kokonaisluku. Rekistereistä jokainen pystyi sisältämään kahdeksan 8 bitin kokonaislukua, neljä 16 bitin kokonaislukua, kaksi 32 bitin kokonaislukua tai yhden 64 bitin kokonaisluvun. Rekisterin sisältäessä useita alemman tarkkuuden tietoalkioita alettiin tätä kutsumaan *pakatuksi* datatyypiksi (Peleg ja Weiser 1996).

MMX myös mahdollisti saturaatioaritmetiikan (engl. *Saturation arithmetic*) modulaarisen aritmetiikan rinnalla. Saturaatioaritmetiikassa operaatioilla on maksimi- sekä miniarvonsa, jotka ylittäessä tai alittaessa arvo jää maksimi- tai minimiarvoonsa, eikä pyörähdä ylitse toiseen ääripäähän kuten modulaarisessa aritmetiikassa. Tällä on hyvin tärkeä rooli esimerkiksi tietyissä 3D-grafiikan sekä signaalinkäsittelyn algoritmeissa (Peleg ja Weiser 1996).

MMX-käskykantalaajennus koostui 57 käskystä pakatulle datalle: Summaus- sekä vähennyslaskuoperaatioista, kertolaskuoperaatioista, siirto-operaatioista (engl. *shift*), vertailuoperaatioista, loogisista operaatioista, datan uudelleenjärjestämis- sekä purkuoperaatioista ja lataussekä tallennusoperaatioista (Peleg ja Weiser 1996, ks. Taulukko 1). Datan manipulointioperaatioihin kuuluivat sekoitus- (engl. *shuffle*) sekä purkuoperaatiot. Sekoitusoperaatiolla voitiin kahden rekisterin välillä valita bittimaskin avulla kohderekisteriin tulevat data-alkiot pakatusta datasta sekä kääntää alkioiden järjestys (“Cray XC30 Day 2 - Programming AVX Intrinsics” 2013). Purkuoperaatiolla voitiin valita data-alkiot puolittamalla lähderekisterit ja valitsemalla joko merkitsevimmät data-alkiot tai vähiten merkitsevimmät data-alkiot päätymään kohderekisteriin (ks. Kusswurm 2014, Luku 5).

MMX-käskykannan operaatioiden tuottama nopeutus oli suurimmillaan kaksinkertainen skalaarioperaatioihin verrattuna (Peleg ja Weiser 1996). Myös eräitä tärkeitä operaatioita kuten minimi ja maksimi puuttui MMX:n käskykannasta (Amiri ja Shahbahrami 2020). MMX-käskykantalaajennus olikin monelta osin puutteellinen sekä ongelmallinen, mutta se avasi tien x86-arkkitehtuurin SIMD-ominaisuuksille sekä niiden tutkimukselle (Amiri ja Shahbahrami 2020).

4.2 SSE

SSE-käskykantalaajennus (engl. *Streaming SIMD extension*) julkaistiin vuonna 1999 Intelin Pentium III -prosessorille (Thakkur ja Huff 1999). Se oli suora päivitys edellisen sukupol-

ven MMX-käskykantalaajennukselle. SSE:tä tukevat prosessorit pystyivät siten käyttämään myös MMX-käskykannan operaatioita (Hassaballah, Omran ja Mahdy 2008).

Suurin muutos SSE:n MMX:n välillä oli, että se ei enää käyttänyt x87-liukulukurekisterejä, vaan sille luotuja kahdeksaa (nimellisesti XMM0 - XMM7) 128 bitin kokoista rekisteriä. Se myös sisälsi 32 bitin MXCSR-rekisterin, joka sisälsi erilaisia kontrolli- sekä tilatietoja liukulukuoperaatioille (ks. Kusswurm 2014, Luku 7). XMM-rekisterien ansiosta SSE-käskykannan operaatioiden suorittaminen oli mahdollista yhtä aikaa joko x87-käskykannan tai MMX-käskykannan operaatioiden kanssa. Se myös laajensi datatyypin pelkästä pakatusta kokonaisluvusta pakattuihin liukulukuihin tarjoten näille kaksi eri moodia: IEEE-standardin yksittäisen tarkkuuden moodin sekä FTZ-moodin (engl. *flush-to-zero*) (Raman, Pentkovski ja Keshava 2000). FTZ-moodi oli välttämätön joissain graafisissa algoritmeissa sekä tarjosi nopeutusta reaaliaikaisissa sovelluksissa pienellä tarkkuuden häviöllä. IEEE-standardin mukainen liukuluku oli tärkeä yhteensopivuuden kannalta tulevaisuutta ajatellen sekä tarkkuutta vaativissa sovelluksissa (Thakkur ja Huff 1999). Merkittävä heikkous SSE:ssä oli, että sille tarkoitettuja 128 bitin rekisterejä ei voinut käyttää kokonaislukuoperaatioissa, vaan näiden pakatut versiot täytyi edelleen sijaita MMX-rekistereissä (Amiri ja Shahbahrami 2020, ks. Taulukko 3).

Eräs tärkeä seuraus datatyypin laajenemisesta SSE:ssä liukulukuihin oli, että se täten mahdollisti tehokkaamman käskykannan myös yksittäisille liukulukuoperaatioille verrattuna x87-käskykantaan. Tämä johtui siitä, että x87-käskykantalaajennus käytti rekistereissään pinotietorakennetta, ja näin ollen sen rekistereihin ei voitu suoraan viitata toisin kuin SSE:ssä (ks. Kusswurm 2014, Luku 7).

SSE:n käskykanta sisälsi 70 operaatiota, joista suurin osa oli tehty pakattujen liukulukujen käsittelyyn. Operaatiot sisälsivät tyypilliset aritmeettis-loogiset operaatiot liukuluvuille sekä MMX:stä tutut keräys- ja purkuoperaatiot. SSE tarjosi myös uusia käskyjä kuten esihakukäskyn (engl. *Prefetch*), jolla ohjelmoija pystyi määrittelemään usein käytetyn datan vähemmän käytetystä. Näin ollen laitteisto pystyi mahdollisesti säätelemään välimuistiin tulevaa dataa ja estämään välimuistin mahdollisen saastumisen väliaikaisella datalla (Raman, Pentkovski ja Keshava 2000). Myös muunto-operaatiot pakatuista liukuluvuista pakattuihin kokonaislukuihin - ja toisinpäin - löytyivät SSE:stä. Graafisten sovellusten laskuissa käy-

tetty neliöjuuren käänteisluku lisättiin SSE:ssä suoraan laitteistotasolle 12 bitin mantissan tarkkuudella. SSE lisäsi myös käskykantaansa uusia eritoten mediasovelluksissa käytettäviä käskyjä, kuten absoluuttisen erotuksen summan (engl. *sum of absolute differences*), jolla on merkittävä rooli esimerkiksi videokoodekeissa (Thakkur ja Huff 1999).

4.3 SSE2, SSE3, SSSE3, SS4

Seuraavien vuosien saatossa Intel jatkoi SSE-käskykantalaajennuksen kehittämistä. SSE2 julkaistiin jo vuoden päästä SSE:n julkaisemisesta vuonna 2000. Sen tärkeimpiä uudistuksia olivat MMX:ssä esitellyt pakattuja kokonaislukuja koskevat operaatiot käyttämään SSE:n 128 bitin XMM-rekisterejä. Se myös mahdollisti kahden pakatun kaksoistarkkuuden liukuluvun (engl. *double-precision floating point*) käytön, sekä laajensi käskykantaa 70:stä 144:ään, joista suurin osa oli MMX-käskykantalaajennuksen käännöksiä käyttämään SSE:n tarjoamia XMM-rekisterejä (Amiri ja Shahbahrami 2020).

Myöhemmin tulleet SSE3 (vuonna 2004), SSSE3 (Supplimental Steaming SIMD) sekä SSE4 (vuonna 2007) laajensivat entisestään käskykantaa. Useimmat käskyistä olivat hyvinkin sovelluskohtaisia, sekä parannuksia jo olemassa oleviin käskyihin. Eräs tärkeä uudistus SSE3:ssa oli pakattujen liukulukujen horisontaaliset vähennys- sekä summausoperaatiot, joissa vierekkäiset vektorialkiot joko summattiin tai vähennettiin keskenään (ks. Kusswurm 2014, Luku 7).

SSE4 oli myös jaettu SSE4.1:een sekä SSE4.2:een. SSE4.1 tarjosi datan manipulointikomennon (engl. *blend*), jolla kohderekisteriin voitiin valita kahdesta eri rekisteristä sisältö bittimaskin avulla tulemaan joko ensimmäisestä tai toisesta lähderekisteristä vaihtamatta alkioiden järjestystä (“Cray XC30 Day 2 - Programming AVX Intrinsics” 2013). Lisäksi SSE4.1 tarjosi eritoten graafisissa sovelluksissa tärkeän pistetulon pakatuille liukuluvuille. SSE4.2:n tärkeimpiä uudistuksia oli sen kehittelemät pakattujen merkkijonojen operaatiot (ks. Kusswurm 2014, Luku 7). Rekisterien leveys ei SSE:n kohdalla kuitenkaan noussut 128 bitistä ylöspäin, ja siten SIMD:ssä saavutettava teoreettinen maksiminopeutus ($\frac{128}{k}$, missä k on datatyypin koko biteissä) ei lisääntynyt (Amiri ja Shahbahrami 2020).

4.4 AVX

AVX-käskykantalaajennus (engl. *Advanced Vector Extensions*) julkaistiin vuonna 2011 Intelin Sandy Bridge -mikroarkkitehtuurille. Sen tärkeimpiä uudistuksia SSE:hen verrattuna oli rekisterien määrän lisääntyminen kuuteentoista (nimellisesti YMM0-YMM15) sekä niiden leveyden kasvaminen 256 bittiin. YMM-rekisterejä pystyttiin hyödyntämään käskykannan liukulukuoperaatioissa (Amiri ja Shahbahrami 2020). AVX tuki myös pakattujen kokonaislukujen SIMD-operaatioita, mutta vain 128 bitin leveydellä (ks. Kusswurm 2018, Luku 4, Taulukko 4-1). AVX:n käskykannan koodaus myös muuttui ja tätä kutsuttiin VEX:ksi (engl. *Vector extension*). VEX:n tärkein uudistus oli, että se käytti käskyissä kahden operandin sijasta kolmea, mikä näin ollen mahdollisti tuhoutumattomat lähdeoperandit (Amiri ja Shahbahrami 2020). Lisäksi AVX mahdollisti muunnoksen yksittäisen tarkkuuden liukulukujen (32 bittiä) ja puolen tarkkuuden liukulukujen (16 bittiä) välillä. Puolen tarkkuuden liukuluvuille ei voitu suorittaa pakattuja operaatioita, mutta ne mahdollistivat datan pakkaamisen muistissa tiheämpään (ks. Kusswurm 2014, Luku 12). Näiden ohella eräitä AVX:n tuomia käskykannan parannuksia olivat pakattujen liukulukujen lähetys- (engl. *broadcast*) sekä permutaatio-operaatiot. Lähetysoperaatiolla voitiin muistista kopioida tietty data-alkio pakatun liukuluvun jokaiseksi data-alkioksi rekisteriin. Permutaatio-operaatiolla sen sijaan voitiin uudelleenjärjestää *yhden* rekisterin sisältö bittimaskin avulla (“Cray XC30 Day 2 - Programming AVX Intrinsics” 2013).

4.5 AVX2

Vuonna 2013 julkaistun AVX2-käskykantalaajennuksen merkittävimpiä uudistuksia oli pakattuja kokonaislukuja koskevat operaatiot käyttämään 256 bitin YMM-rekisterejä. Se myös paranteli AVX:n lähetys- sekä permutaatio-operaatioita (Amiri ja Shahbahrami 2020). Päivitetyillä AVX2:n permutaatio-operaatioilla voitiin järjestää *kahdesta* eri rekisteristä tuleva data kohderekisteriin bittimaskin avulla, ja siten tehdä hyvinkin monimutkaisia datan uudelleen järjestelyjä rekisterien välillä (“Cray XC30 Day 2 - Programming AVX Intrinsics” 2013). AVX2:n yhteydessä julkaistiin myös FMA-käskykantalaajennus (engl. *Fused-multiply-add*), joka mahdollisti eritoten matriisilaskuissa hyvinkin tärkeän yhdistetyn kerto- sekä summauslaskun pakatuille liukuluvuille. FMA:lla voitiin suorittaa yhdellä käskyllä kerto- sekä vähennys-

tai summauslasku pakatun datan jokaiselle alkiolle. Näin ollen esimerkiksi

$$a = (b * c) + d$$

voidaan suorittaa yhdellä käskyllä kahden sijaan. FMA:ssa on myös huomattavaa se, että pyöristys tapahtuu ainoastaan yhden kerran kahden sijaan, mikä edelleen lisää FMA-operaation tehokkuutta (ks. Kusswurm 2014, Luku 12). Näiden lisäksi hyvin tärkeä uudistus käskykantaan oli keräysoperaatio (engl. *gather*), jolla AVX:n rekisteriin voitiin hakea muistista epälineaarisesti data-alkioita. Tämän mahdollisti VSIB-muistiviittaustekniikka (engl. *vector scale-index-base*). Täten data-alkioiden ei tarvinnut sijaita muistissa enää lineaarisesti, jotta ne voitiin ladata rekisteriin, vaan ne voitiin ladata täysin eri muistialueilta (ks. Kusswurm 2014, Luku 12).

4.6 AVX-512

AVX-512 (AVX3) on viimeisin vuonna 2016 x86-64-arkkitehtuurille julkaistu käskykantalaajennus. Sen huomattavimpia uudistuksia edellisen sukupolven AVX-käskykantalaajennuksiin on rekisterien määrän kasvaminen 16:sta 32:een (nimellisesti ZMM0-ZMM31), sekä niiden leveyden kasvaminen 512 bittiin (Amiri ja Shahbahrami 2020). AVX-512 sisältää myös kahdeksan (nimellisesti K0-K7) 64 bitin levyistä operaatiomaskirekisteriä (engl. *opmask register*), jotka tarjoavat erilaisia mahdollisuuksia pakattujen operaatioiden ehdolliselle suorittamiselle sekä operaatioiden tuottamien tulosten tallentamiselle rekisteriin (ks. Kusswurm 2018, Luku 12). Esimerkiksi pakattu summausoperaatio voidaan ehdollistaa operaatiomaskirekisterillä tallentamaan kohderekisteriin vain niiden vektorialkoiden summien tulokset, joissa operaatiomaskirekisterin indeksissä sijaitsee arvo 1. Näin ollen operaatiomaskirekistereillä voi hyvinkin monipuolisesti vaikuttaa olemassa olevien operaatioiden suorituksen kulkuun.

AVX-512 sisältää yli kolmesataa operaatiota, jotka on hajautettu eri osalaajennuksien välille (Amiri ja Shahbahrami 2020). AVX-512:n osalaajennuksia on tällä hetkellä seitsemäntoista, joista eräiden kuvaukset löytyvät (ks. Kusswurm 2018, Luku 12, Taulukko 12-1) sekä (Amiri ja Shahbahrami 2020, ks. Taulukko 4). Jokainen AVX-512:sta tukeva prosessori sisältää vähintään AVX-512F-käskykantalaajennuksen (Foundation), joka koostuu yleisesti tarvittavis-

ta pakatun datan manipulointi- ja muistioperaatioista sekä aritmeettis-loogisista operaatioista (ks. Kusswurm 2018, Luku 12, Taulukko 12-4). AVX-512F tarjoaa myös uuden EVEX-koodauksen (engl. *enhanced vector extension*). EVEX mahdollisti muun muassa rekisterien koon kasvamisen 512 bittiin sekä neljän operandin käytön, mikä näin ollen mahdollistaa operaatiomaskin käyttämisen syntaksissa (Amiri ja Shahbahrami 2020). AVX-512F tarjoaa myös uuden hajotusoperaation (engl. *Scatter*), jolla voidaan tallentaa rekisteristä epälineaarisesti muistiin ("Intel® Advanced Vector Extensions 2015/2016 Support in GNU Compiler Collection" 2014, ks. sivu 30). Hajotusoperaatiossa, kuten keräysoperaatiossakin, operandeina toimii kaksi vektorirekisteriä sekä mahdollinen maskirekisteri. Näistä vektorirekistereistä ensimmäinen sisältää tallennettavat arvot ja toinen muistiosoitteet, jonne tallennus tehdään. Maskioperandilla voidaan vielä erikseen hallinnoida, että tuleeko tietyssä indeksissä sijaitseva vektorielementti tallentaa muistiin vaiko ei. Yleisesti ottaen keräys- ja hajotusoperaatiot muodostavat tärkeän kokonaisuuden, joka mahdollistaa monien vaikeasti vektoroituvien ongelmien vektoroimisen (ks. Asanovic 1998, Luku 4.8).

Toinen yleisesti löytyvä käskykantalaajennus AVX-512:sta tukevista prosessoreista on AVX-512CD (Conflict Detection). AVX-512CD tarjoaa mahdollisuuksia vektoroida yleensä vaikeasti vektoroituvia sisäkkäisten rakenteiden päivityksiä ("An Introduction to Knights Landing" 2015, ks. dia 28). Myös eräs kiinnostava AVX-512:n osakäskykantalaajennus on AVX-512VNNI (engl. *Vector Neural Network Instructions*), joka on osin luotu korvaamaan ja nopeuttamaan FMA-käskykantalaajennusta tarjoten pienemmän tarkkuuden yhdistetyn kerto- sekä summauslaskun operaatioita, joilla voidaan nopeuttaa esimerkiksi neuroverkkojen kouluttamista (Carneiro, Serpa ja Navaux 2021).

5 x86-64-arkkitehtuurin SIMD-ominaisuuksien hyödyntäminen ohjelmissa

Ohjelmoijalla on erilaisia tapoja hyödyntää x86-64-arkkitehtuurin vektorikäskykantalaajennuksia ohjelmissa: Ohjelmoimalla Assemblyä, kääntäjien tarjoamien kääreiden (engl. *wrapper*) ja kääntäjälaajennusten avulla tai kääntäjän automaattista vektorointia käyttämällä (ks. Amiri ja Shahbahrami 2020, Taulukko 6). Näistä selkein tapa on Assemblyn kirjoittaminen, joko suoraan käyttäen Assemblereja tai esimerkiksi C/C++-kielestä löytyvien tapojen kautta. Assemblyn suoraan kirjoittaminen ei usein ole suositeltavin tapa, sillä kääntäjien kääreet tarjoavat ylemmän tason rajapinnan sekä hoitavat mekaanisia asioita, kuten muuttujien siirtämisen vektorirekistereihin (Salapura 2011).

Kääntäjien kääreitä käytettäessä ohjelmoija määrittelee operoitavalle käskylle muuttujan tai muuttujat, joille vektorioperaatio tullaan suorittamaan. Tämän etuna on se, että kääreet yleensä hoitavat muuttujan lataamisen vektorirekistereihin sekä muita manuaalisia tehtäviä, mikä helpottaa ohjelmoijan tehtävää (Amiri ja Shahbahrami 2020). Puhtaan Assemblyn kanssa voidaan joutua myös käyttämään mahdollisesti muita kuin vektorikäskykantalaajennuksen tarjoamia operaatioita, jolloin kirjoitettu koodi ei välttämättä ole yhteensopivaa jollain toisella laitteistolla. C/C++-kääntäjien kääreistä löytyy kattava listaus Intelin-verkkosivulta (“Intelin listaus ja opas C/C++-kääreisiin” 2023).

Kääreiden lisäksi myös erinäiset kääntäjälaajennukset tarjoavat mahdollisuuksia hyödyntää x86-64-arkkitehtuurin SIMD-ominaisuuksia. Näistä eräs on OpenMP (“OpenMP-kotisivu” 2023). OpenMP hyödyntää kääntäjälle annettavia direktiivejä, jotka kääntäjän esiprosessointivaihe havaitsee ja vektoroi direktiivillä merkityn ohjelmalohkon (“Zakhar Matveev, OpenMP” 2018). OpenMP:n käyttö on siis yksinkertaisempaa kuin C/C++-kääntäjien tarjoamat kääreet. Ongelmaksi voi kuitenkin nousta vaikeasti vektoroituvat ohjelmalohkot, jolloin joudutaan turvautumaan Assemblyyn tai kääreisiin.

Näiden lisäksi eräs hyvin tärkeä tapa on kääntäjien automaattiset vektoroinnit (engl. *Compiler automatic vectorization*, lyh. CAV). CAV:ssa kääntäjä etsii ohjelmakoodista osia, jotka se kykenee vektoroimaan ja muuntaa kyseisen ohjelmakoodin kohdan vektoroiduksi (Salapura

2011). Tällaisia vektoroituja osia ohjelmista yleisesti ovat ohjelmointisilmukat, joissa ei ole suuria riippuvuuksia silmukan suorituskertojan välillä. Matriisien kertolasku toisella matriisilla on klassinen esimerkki hyvin vektoroituvasta ongelmasta (Amiri ja Shahbahrami 2020). CAV:n tärkeyttä myös lisää se, että usein vektorien käskykantalaajennusten hyödyntäminen vaatii syvällistä ymmärtämistä tietokoneen arkkitehtuurista sekä käytettävistä vektorikäskykantalaajennuksista. Täten kaikista tavoista CAV on ohjelmoijan näkökulmasta helpoin tapa hyödyntää x86-64-arkkitehtuurin tarjoamia SIMD-ominaisuuksia (Amiri ja Shahbahrami 2020).

Kääntäjät eivät kuitenkaan vielä jokaista ongelmaa osaa suoraan vektoroida, jolloin vektoroinnin takaamiseksi usein joudutaan turvautumaan Assemblyyn tai kääreisiin (Amiri ja Shahbahrami 2020). Toisaalta taas joissain ongelmissa kääntäjien automaattiset vektoroinnit olivat ohjelmoijaa parempia vektoroimaan (ks. Amiri ja Shahbahrami 2020, Kuva 28, 29.). Ongelmia kääntäjien automaattisessa vektoroinnissa usein tuottavat ehtolauseet sekä huonosti muistiin ryhmitellyt vektorit (Feng, He ja Tao 2021). CAV:lla tuotetut ohjelmat ovat siirrettävimpiä eri laitteistojen ja arkkitehtuurien välillä, mikäli kyseiselle laitteistolle löytyy kyseisen kielen kääntäjä (Salapura 2011). Huomattavaa kääntäjien automaattisessa vektoroinnissa on vielä se, että kyseessä ei ole uusi ilmiö, vaan aihetta on tutkittu hyvinkin paljon vektoriprosessorien myötä, jotka ovat tarjonneet kääntäjiä eri kielille - kuten C ja Fortran - alusta asti (Russell 1978).

6 Yhteenveto

Tutkimuksessa on esitelty lyhyesti vektorilaskennan historiaa 1960-luvulta aina nykypäivään asti. Vektorilaskennan teoriaa on myös esitelty eri vektorilaskennan arkkitehtuureissa, joissa tutkimuksessa keskeisimpänä ovat erinäiset x86-64-arkkitehtuurin vektorikäskykantalaajennukset. Näiden käskykantalaajennusten syntyhistoriaa ja kehityskaarta on kuvattu yhdessä niiden tuomien tärkeimpien vektorilaskennan ominaisuuksien kanssa. Lopuksi vielä esiteltiin mahdollisia tapoja hyödyntää vektorikäskykantalaajennusten ominaisuuksia ohjelmissa tuoden esille kunkin tavan heikkouksia sekä hyötyjä.

Vektorilaskennan SIMD-ominaisuudet x86-64-arkkitehtuurissa ovat, ja tulevat olemaan, tulevaisuudessa erittäin keskeisessä roolissa eritoten erinäisten palvelimilla ajettavien ohjelmien nopeuttamisessa. Hyvin monia ohjelmia suoritetaan nykyään palvelimilla, joiden prosessorit pohjautuvat x86-64-arkkitehtuuriin. Näin ollen lähes kaikkien ohjelmien kohdalla, jotka sisältävät vektoroituvia ohjelmalohkoja, on mahdollista hyödyntää moderneja vektorikäskykantalaajennuksia. Täten erittäin tärkeä tulevaisuuden tutkimuskohde onkin kääntäjien automaattiset vektoroinnit, jotta mahdollisimman moni ohjelma voisi hyödyntää automaattisesti x86-64-arkkitehtuurin tarjoamia SIMD-ominaisuuksia. Näin ollen kääntäjien automaattiset vektoroinnit olisikin otollinen jatkotutkimuksen kohde.

Lähteet

Amiri, Hossein, ja Asadollah Shahbahrami. 2020. “SIMD programming using Intel vector extensions”. *Journal of Parallel and Distributed Computing* 135:83–100. ISSN: 0743-7315. https://doi.org/10.1016/j.jpdc.2019.09.012.

“An Introduction to Knights Landing”. 2015. Viitattu 4. tammikuuta 2023. https://www.alcf.anl.gov/files/Sewall_ANL-ESPKnightsLanding.pdf.

Asanovic, Krste. 1998. *Vector microprocessors.* University of California, Berkeley.

Barnes, G.H., R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick ja R.A. Stokes. 1968. “The ILLIAC IV Computer”. *IEEE Transactions on Computers* C-17 (8): 746–757. ISSN: 1557-9956. https://doi.org/10.1109/TC.1968.229158.

Carneiro, AndrÃ© Ramos, Matheus S. Serpa ja Philippe O. A. Navaux. 2021. “Lightweight Deep Learning Applications on AVX-512”. Teoksessa *2021 IEEE Symposium on Computers and Communications (ISCC),* 1–6. ISSN: 2642-7389. https://doi.org/10.1109/ISCC53001.2021.9631464.

“Computer Architecture, Lecture 28: SIMD and GPUs”. 2010. Viitattu 10. huhtikuuta 2023. https://course.ece.cmu.edu/~ece740/f10/lib/exe/fetch.php?media=740-fall10-lecture28-simd-gpus.pdf.

“Cray XC30 Day 2 - Programming AVX Intrinsics”. 2013. Viitattu 1. huhtikuuta 2023. https://www.youtube.com/watch?v=7oCK10XW_4I.

Eichenberger, Alexandre E, Peng Wu ja Kevin O’brien. 2004. “Vectorization for SIMD architectures with alignment constraints”. *Acm sigplan notices* 39 (6): 82–93. https://doi.org/10.1145/996841.996853.

Elster, Anne C. 2021. “The European Factor: From ARM to Atos”. Conference Name: Computing in Science & Engineering, *Computing in Science & Engineering* 23, numero 1 (tammikuu): 102–105. ISSN: 1558-366X. https://doi.org/10.1109/MCSE.2020.3044070.

Feng, Jing Ge, Ye Ping He ja Qiu Ming Tao. 2021. “Evaluation of compilers’ capability of automatic vectorization based on source code analysis”. *Scientific Programming* 2021:1–15. https://doi.org/10.1155/2021/3264624.

Flynn, Michael J. 1967. “Very High-Speed Computing Systems”. *Proceedings of the IEEE* 54. https://doi.org/10.1109/PROC.1966.5273.

Gregory, J., ja R. McReynolds. 1963. “The SOLOMON Computer”. *IEEE Transactions on Electronic Computers* EC-12 (6): 774–781. ISSN: 0367-7508. https://doi.org/10.1109/PGEC.1963.263560.

Hassaballah, M., S. Omran ja Y. B. Mahdy. 2008. “A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications”. *The Computer Journal* 51 (6): 630–649. ISSN: 0010-4620, 1460-2067. https://doi.org/10.1093/comjnl/bxm099.

Hennessy, John L, ja David A Patterson. 2017. “Computer architecture: a quantitative approach 6th Edition”. Elsevier. ISBN: 978-0128119051.

“Intel® Advanced Vector Extensions 2015/2016 Support in GNU Compiler Collection”. 2014. Viitattu 4. tammikuuta 2023. https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf.

“Intelin listaus ja opas C/C++-kääreisiin”. 2023. Viitattu 21. huhtikuuta 2023. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Kruger, Jens, ja Rudiger Westermann. 2003. “Linear algebra operators for GPU implementation of numerical algorithms”, https://doi.org/10.1145/882262.882363.

Kusswurm, Daniel. 2014. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX.* Apress. ISBN: 978-1484200650.

———. 2018. *Modern X86 Assembly Language Programming: Covers X86 64-bit, AVX, AVX2, and AVX-512.* Apress. ISBN: 978-1484240625.

Lindholm, Erik, Mark J. Kilgard ja Henry Moreton. 2001. “A user-programmable vertex engine”. Teoksessa *Proceedings of the 28th annual conference on Computer graphics and interactive techniques,* 149–158. ACM. ISBN: 978-1-58113-374-5. https://doi.org/10.1145/383259.383274.

Lindholm, Erik, John Nickolls, Stuart Oberman ja John Montrym. 2008. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. *IEEE Micro* 28 (2): 39–55. ISSN: 0272-1732. https://doi.org/10.1109/MM.2008.31.

“OpenMP-kotisivu”. 2023. Viitattu 21. huhtikuuta 2023. https://www.openmp.org/.

Peleg, Alex, ja Uri Weiser. 1996. “MMX technology extension to the Intel architecture”. *IEEE micro* 16 (4): 42–50. https://doi.org/10.1109/40.526924.

Raman, S.K., V. Pentkovski ja J. Keshava. 2000. “Implementing streaming SIMD extensions on the Pentium III processor”. Conference Name: IEEE Micro, *IEEE Micro* 20 (4): 47–57. ISSN: 1937-4143. https://doi.org/10.1109/40.865866.

Russell, Richard M. 1978. “The CRAY-1 computer system”. *Communications of the ACM* 21 (1): 63–72. https://doi.org/10.1145/359327.359336.

Salapura, Valentina. 2011. “Vector Extensions, Instruction-Set Architecture (ISA)”. Teoksessa *Encyclopedia of Parallel Computing,* toimittanut David Padua, 2129–2135. Springer US. ISBN: 978-0-387-09766-4. https://doi.org/10.1007/978-0-387-09766-4_259.

Slotnick, D.L. 1982. “The Conception and Development of Parallel Processors: A Personal Memoir”. Conference Name: Annals of the History of Computing, *Annals of the History of Computing* 4, numero 1 (tammikuu): 20–30. ISSN: 0164-1239. https://doi.org/10.1109/MAHC.1982.10003.

Thakkur, S., ja T. Huff. 1999. “Internet Streaming SIMD Extensions”. Conference Name: Computer, *Computer* 32 (12): 26–34. ISSN: 1558-0814. https://doi.org/10.1109/2.809248.

“Zakhar Matveev, OpenMP”. 2018. Viitattu 21. huhtikuuta 2023. https://www.openmp.org/wp-content/uploads/SC18-BoothTalks-Matveev.pdf.