Vektorisointi ohjelmoinnissa ja rinnakkaisuudessa

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tieto-ja viestintätekniikka
Huhtikuu 2025
Benjamin Rauh

TURUN YLIOPISTO
Tietotekniikan laitos

Benjamin Rauh: Vektorisointi ohjelmoinnissa ja rinnakkaisuudessa

TkK-tutkielma, 19 s.
Tieto-ja viestintätekniikka
Huhtikuu 2025

Mikroprosessorien valmistajat ovat siirtyneet tekemään rinnakkaisia laitteistoja. Yleisimpien rinnakkaisten laitteistojen valikoimaan kuuluvat moniydinprosessorit, vektoriprosessorit, sekä näytönohjaimet. Ohjelmisto voi hyödyntää rinnakkaista laitteistoa käyttämällä montaa moniydinprosessoria, ja/tai käyttämällä vektoriprosessorien vektorisaatiota.
Tutkielman tavoitteena on selvittää, mitä rinnakkaisuus on laitteistossa ja ohjelmistossa, mihin vektorisointia yleensä käytetään ja miten kirjoittaa vektoroituvaa koodia. Tämä tutkielma toteutetaan kirjallisuustutkimuksen muodossa.
Tutkielmassa selvitetään mitkä ovat yleisesti käytetyt rinnakkaiset laitteet, vektorisaation käyttökohteet, tavat lisätä vektorisaatiota manuaalisesti ohjelmiston lähdekoodiin ja automaattista vektorisaatiota estävät tekijät.

Asiasanat: vektorisaatio, rinnakkaisuus, kääntäjä, vektoriprosessori, käskykanta, optimointi

Sisällys

1 Johdanto

Mikroprosessorien valmistajat ovat siirtyneet tekemään rinnakkaisia laitteistoja. Yleisimpien rinnakkaisten laitteistojen valikoimaan kuuluu moniydinprosessorit, vektoriprosessorit, sekä näytönohjaimet. Ohjelmisto voi hyödyntää rinnakkaista laitteistoa käyttämällä monta moniydinprosessoria, taikka näytönohjaimen prosessoria samanaikaisesti.[1]

Ohjelmisto voi myös hyödyntää vektoriprosessorin vektorisaatiota. Vektorisaatio mahdollistaa saman operaation suorituksen moneen elementtiin samanaikaisesti [2]. Vektorisaatio on hyödyllisintä tietorinnakkaisuudessa, eli kun käsitellään suurta määrää dataa [2] kuten tietokannoissa tai fysiikan tutkimuksissa. Koodia voidaan vektorisoida manuaalisesti sekä automaattisesti vektorisoivan kääntäjän avulla.

Vektorointiin käytetään vektorioperaatioita niitä sallivista käskykannoista. Prosessorin tai grafiikkasuoritin sekä käyttöjärjestelmän pitää pystyä käyttämään käytettyjä vektorioperaatioita.[3] Keinoja lisätä vektorisointia ovat assemblyn kirjoittaminen, luontaisten funktioiden (engl. *intrinsic function*) sekä vektoriluokkien käyttö.[2]

Kääntäjän automaattinen vektorointi on turvallisempaa, ja vaatii vähemmän vaivaa kuin manuaalinen vektorisointi[3]. Käytetyn kääntäjän tarvitsee olla vektorisoiva kääntäjä. Kääntäjän täytyy myös olla varma, että koodi on vektoroituvissa. Koodia voi kirjoittaa tavalla, joka auttaa kääntäjää ymmärtämään voiko koodia vektorisoida. Kääntäjälle voi myös kertoa lisää tietoa koodista kääntäjädirektiivien avulla.

[4]

Tutkielman tavoitteena on selvittää, miten ohjelmistoa saadaan rakennettua hyödyntämään saatavilla olevaa laskentatehoa paremmin kuin perinteisellä ohjelmointitavalla, jossa ei kiinnitetä huomiota tehoon. Tutkielmassa keskitytään vektorisaatioon tapana tehostaa ohjelmia. Tutkielmassa tarkastellaan, miten vektorisoivaa koodia kirjoitetaan. Tutkielmassa selvitetään myös, mihin käyttöön vektorisaatio on hyödyllistä.

Työssä tarkasteltavat tutkimuskysymykset ovat:

- Mitä rinnakkaisuus on laitteistossa ja ohjelmistossa?
- Mihin vektorisointia yleensä käytetään?
- Miten kirjoittaa vektoroituvaa koodia?

Tämä tutkielma toteutetaan kirjallisuustutkimuksen muodossa. Tutkielma käyttää pääasiassa tieteellisiä artikkeleita IEEE (Institute of Electrical and Electronics Engineers) ja ACM (Association for Computing Machinery) julkaisijoilta.

Hakutuloksia rajattiin haulla (vectorization AND programming) OR (SIMD AND programming) ja käytettiin lähdemateriaalin etsimiseen ACM Digital Library- ja IEEE Xplore -tietokannoista sekä Google Scholar -palvelusta. Hakutuloksia valittiin otsikon osuvuuden, tiivistelmän ja lyhyen läpikatsauksen perusteella. Kirjoituksen aiheeseen sopivat artikkelit pidettiin.

Tutkielmassa luvussa 2 tutustutaan rinnakkaiseen laitteistoon ja sen kehitykseen. Kappaleessa tutustutaan myös siihen, miten ohjelmisto hyödyntää rinnakkaista laitteistoa. Tutkielman luvussa 3 tarkastellaan mihin vektorisaatio on hyödyllistä esimerkkien avulla. Tutkielman luvussa 4 tarkastellaan miten vektorisaatiota lisätään koodiin manuaalisilla keinoilla, sekä katsotaan, miten kirjoitetaan automaattisesti vektoroituvaa koodia. Tutkielman luku 5 on yhteenveto aikaisemmista kappaleista.

2 Rinnakkainen laitteisto

Mikroprosessorien keksimisestä lähtien niiden nopeus on kasvanut vuosi vuodelta, mutta mikroprosessorien nopeutuminen alkoi hidastua 2000-luvun alussa. Tämä tarkoitti, että ohjelmien nopeuttamiseen piti keksiä muita keinoja kuin vain nopeampien mikroprosessorien valmistus. Mikroprosessorien valmistajat siirtyivät valmistamaan moniydinprosessoreita (engl. *multi-core processor*), jotka yhdistivät monia yitimiä, eli laskentaan kykeneviä laitteita yhteen prosessoriin.[1]

Vektoriprosessorit ovat prosessoreita, jotka pystyvät tallentamaan, sekä operoimaan vektoritietorakennetta tuetuilla vektorikomennoilla. Niillä on myös kyky hakea tietoa eri muistipaikoista samanaikaisesti. Skalaariprosessorit sen sijaan operoivat vektorirakenteiden yksittäisiä elementtejä.[1]

Ensimmäinen vektorisointia tukeva tietokone oli Illiac IV vuonna 1972. Se oli laite, jossa 64 prosessoria toimivat rinnakkaisesti. Laitteessa kaikki prosessorit tekivät saman operaation yhdelle elementille samanaikaisesti. [5] Tämän jälkeen vektoriprosessoreita käytettiin usein supertietokoneissa. Näistä ensimmäiset tunnetut olivat CRAY-1 ja STAR 100. Nämä olivat paljon tehokkaampia kuin Illiac IV ja pystyivät käyttämään enemmän vektorioperaatioita.[6] Nykyään yleiset moniydinprosessorit ovat yhdistelmä vektoriprosessorien ja skalaariprosessorien välillä ja niillä saattaa olla osa ytimistä vektorisoivia ja osa skalaarisia.[1]

Toinen nykyään kovasti käytetty rinnakkaisen laskentayksikön tyyppi on näytönohjain (engl. *Graphics processing unit*). Näytönohjaimet ovat kehittyneet erinomai-

siksi käsittelemään laskuja nopeasti ja rinnakkaisesti. Näytönohjaimia usein käytetään ohjelmistokehitykseen, joka tekee renderöintiä, mutta niiden korkean suorituskyvyn takia niitä ollaan alettu käyttämään monenlaisiin tehtäviin. Näiden tehtävien yleiset ominaisuudet ovat suuret laskennalliset vaatimukset, mahdollisuus rinnakkaiselle ohjelmoinnille ja algoritmeille, joissa suoritusteho on tärkeämpi kuin laskennan alkamisen ja loppumisen välinen viive.[7]

Moniytimisiä laitteistoja luokitellaan myös riippuen ytimien kyvystä jakaa ja lukea dataa. Yksi kategoria on jaettu muisti (engl. *shared-memory*). Jaetun muistin arkkitehtuurimallissa kaikki ytimet voivat lukea ja kirjoittaa melkein kaikkialle tietokoneen muistiin. Ytimet voivat koordinoida prosessoinnin järjestystä lukemalla ja päivittämällä muistia. Toinen on hajautettu muisti (engl. *distributed-memory*), jossa jokaisella ytimellä on oma muisti.[1] Laitteisto voi olla myös näistä yhdistetty systeemi. Yhdistetyssä systeemissä on jaettua, sekä hajautettua muistia, mutta yleisesti tällaisella laitteistolla on jaettu muisti.[8] Seuraavassa luvussa keskitytään jaettuun muistiin.

2.1 Rinnakkaisuuden hyödyntäminen ohjelmistossa

Ohjelmisto, jota ei olla tehty rinnakkaista laitteistoa hyödyntäväksi ei saa rinnakkaisesta laitteistosta lisää tehoa. Rinnakkaista laitteistoa varten pitää tehdä niitä hyödyntäviä ohjelmia.[1] Rinnakkaisen laitteiston kasvua ei olla hyödynnetty täysin monessa ohjelmistossa. Kääntäjät voivat itsestään tuottaa rinnakkaiselle arkkitehtuurille tarkoitettua koodia. Ohjelmat joita ei olla kirjoitettu rinnakkaista laskentamallia käyttäen saattavat olla kääntäjille hankala rinnakkaistaa. Ohjelmat pitää siis kirjoittaa mahdollisimman rinnakkaisiksi parasta tulosta varten. Yleinen tapa rinnakkaisia ohjelmia tehdessä on jakaa ohjelman tehtävät (tehtävärinnakkaisuus) tai ohjelman käsittelemä data osiin (datarinnakkaisuus) ja jakaa osat eri laitteiston ytimille tehtäväksi.[8]

Useat ohjelmat voidaan jakaa tehtäviin, jossa jokainen tehtävä ratkaisee osan ongelmasta. Tehtävät voivat kommunikoida toistensa kanssa kesken selvittämisen, tai ne voivat saada tarvittavat tiedot tehtävän alussa, ja yhdistää vasta tulokset. Tehtävät saattavat myös luoda uusia tehtäviä. Tehtävärinnakkaisuuteen kuuluu tehtävien osittaminen, ajoittaminen ja niiden jakaminen prosessoreille. Datarinnakkaisuudessa sen sijaan pyritään tekemään operaatioita monelle elementille samanaikaisesti. [9]

Rinnakkaista ohjelmointia varten on tehty sitä helpottavia työkaluja. Jaetulla muistilla oleviin arkkitehtuureihin suosittuja valintoja ovat OpenMP- ja POSIX Threads-ohjelmointirajapinnat. OpenMP on toteutettu kääntäjädirektiiveillä, jotka luovat säikeitä, tekevät synkronointioperaatioita ja hallitsevat jaettua muistia.[8] POSIX Threads on sen sijaan kirjasto, joka on toteutettu eri funktioilla. OpenMP on helpompi käyttää, mutta antaa vähemmän hallintaa ohjelmoijalle ja vaatii sitä ymmärtävän kääntäjän. POSIX Threads antaa enemmän hallintaa ja toimii kaikilla C-kielen kääntäjillä, mutta vaatii enemmän ohjausta ja on herkkä virheille.[1] Grafiikkasuorittimien rinnakkaistamiseen suosittu valinta on NVIDIAn luoma ohjelmointirajapinta CUDA. CUDA käyttää prosessoria tehtävien koordinointiin ja se jakaa työn grafiikkasuorittimelle rinnakkaistettavaksi.[8]

Ohjelmistossa rinnakkaisuutta usein käytetään eri rinnakkaisuusmallien avulla. Haarautus/yhdistys (engl. *fork/join*) on malli, joka luo dynaamisesti rinnakkaisia tehtäviä tietyissä kohtia ohjelmaa ja sitten kerää ja yhdistää ne toisissa kohdissa ohjelmaa. Ohjelman alussa ja lopussa ainoastaan yksi osa ohjelmaa on käynnissä, mutta ohjelman aikana saattaa syntyä monia eri rinnakkaisia tehtäviä. Se antaa mallille kyvyn sopeutua eri prosessorien saatavuuksiin, välttämällä prosessoreita, joissa on ruuhkaa. Malli vaatii, että tehtävät jaetaan prosessoreille kesken ohjelman käynnissä olemisen. Mallia voi käyttää eri kirjastojen kuten POSIX Threadsin ja OpenMP-n avulla.[9]

Ohjelmalistaus 1 Esimerkki OpenMP:n ja haarautus/yhdistys-mallista.

```
#include <omp.h>
int main(void)
{
   printf("Ennen: säikeiden määrä on %d\n", omp_get_num_threads());
   #pragma omp parallel
   {
      printf("Säie id on %d\n", omp_get_thread_num());
   }
   printf("Jälkeen: säikeiden määrä on %d\n", omp_get_num_threads());
}
```

Ohjelmalistauksessa 1 käytetään OpenMP-ohjelmointirajapintaa haarautus/yhdistys-mallilla. Koodin alussa tuodaan OpenMP-kirjaston määritykset näkyviin ohjelman käytettäväksi `omp.h` nimellä. Ohjelman funktion `main` alussa on vain pääsäie käynnissä. Säikeiden määrä tulostetaan OpenMP funktiolla `omp_get_num_threads()`, joka palauttaa alussa 1. Seuraavaksi ohjelma haarautuu OpenMP- kääntäjädirektiivillä `#pragma omp parallel`. Pääsäie luo direktiiviin kohdassa uusia lapsisäikeitä hoitamaan rinnakkaisesti sen direktiivin jälkeistä tehtävää. Esimerkissä tehtävänä tulostetaan prosessia hoitavan pääsäikeen numero, joka saadaan komennolla `omp_get_thread_num()`. Ohjelma tulostaa 0 ensimmäiselle ja 1 toiselle säikeelle, koska ohjelma luo toisen säikeen hoitamaan direktiivin jälkeisen print-komennon. Direktiivin jälkeen säikeiden suoritus yhdistetään, joka todennetaan taas tulostamalla säikeiden määrä 1.

Yksi ohjelma eri dataa(SPMD)-mallissa luodaan tietty määrä rinnakkaisia tehtäviä ohjelman alussa, ja niiden määrä pysyy samana koko ohjelman ajan. Mallissa ohjelmoija on vastuussa rinnakkaisten tehtävien jakamisesta. Tehtävään kuuluu tehtävien kokojen määrääminen, sekä tehtävien jako eri prosessoreille hoidettavaksi. Se tekee mallista työlään mutta nopean, sillä koneen ei tarvitse dynaamisesti tehdä jakamista. [9]

3 Vektorisoinnin käyttö ohjelmoinnissa

Vektorisointi on hyödyllistä, kun tehdään laskuja suuriin määriin dataa, ja jossa sama operaatio tehdään monelle elementille.[2]

3.1 Esimerkkejä vektorisoinnin käytöstä fysiikassa

Monet fysiikan tutkimukseen liittyvät asiat vaativat suuren määrän datan prosessointia. Monte Carlo -menetelmä on yksi tärkeimmistä laskennallisista tekniikoista analysoimaan partikkelienkuljetusongelmia. Menetelmän suuri haittapuoli on, että se vaatii suuren määrän laskennallisia resursseja. Monte Carlo ei ole suoraan helposti vektorisoitavissa, joten sen perusteella luotiin uusi vektorisoituva tapahtumaperusteinen (engl. *event-based*) algoritmi muokkaamalla huonosti vektorisoituva Monte Carlon käyttämä data vektorisoituvaksi.[10]

Suuren energian fysiikan (engl. *High Energy Physics, HEP*) ohjelmistosimulaatiot vievät myös hyvin suuren määrän laskennallisia resursseja. Näiden simulaatioiden laatu myös paranee ohjelmiston suorituskyvyn parantuessa. Näiden tekijöiden takia HEP-ongelmiin kehitettiin rinnakkaisia menetelmiä. Tavanomainen tapa HEP-simulaatioille oli prosessoida tapahtumaan vaikuttavat tekijät peräkkäin. Rinnakkaistamista varten alettiin lajittelemaan samantyyppiset tapahtumat yhteen ja prosessoitiin nämä vektorisaation avulla samanaikaisesti.[11]

3.2 Esimerkkejä vektorisoinnin käytöstä tietokannoissa

Reaaliaikainen datan analysointi on tärkeä osa liiketoimintatiedon hyödyntämistä. Se vaatii suuren määrän datan tallentamista ja siirtämistä tietokannoissa mahdollisimman nopeasti. Tietokantojen käyttöön liittyy monta askelta, joihin vektorisaatiota hyödynnetään. Olennaiset tietokantaoperaatiot lataa ja tallenna olevat muutettu toimimaan vektori syötteellä. Syöttämällä vektoritietorakenteen, joka on täytetty tietokannan osoitteilla, voidaan osoitteista ladata tai osoitteisiin varastoida rinnakkaisesti. Myös tietokannasta tietyn hajautusavaimen etsiminen on vektorisoitu. Avaimet ovat jaettu halutun kokoisiin ämpäreihin, joiden koko sisältö voidaan verrata etsittyyn avaimeen samalla vektorisoinnilla.[12]

Toinen tärkeä tietokantaoperaatio on skannaaminen. Skannaaminen on tietokannasta tietynlaisen tiedon löytäminen. Yksi esimerkki vektorisointiin soveltuvasta operaatiosta on sellainen skannnaus, joka löytää ensimmäisen tiedon, joka tyydyttää halutut kriteerit. Oikea tieto löydetään käymällä kaikki tietokannan avaimet läpi kunnes oikea löytyy. Skannaus on vektorisoitu vertaamalla kriteerin avainta moneen tietokannassa olevaan avaimeen samanaikaisesti.[13]

Tietokannoissa halutaan myös koota (engl. *aggrate*) tietynlaisia tietoja. Sanotaan, että haluttaisiin koota tietyn tietokannan kaikki tiedot, jotka tyydyttävät tietyn kriteerin. Tietokannassa kokoamista ollaan vektorisoitu ensin käymällä tietoja läpi samanaikaisesti, ja muuttaa kaikki tiedot, jotka eivät täytä kriteereitä nolliksi. Sen jälkeen kaikki jäljelle jääneet tiedot ovat kerätty.[13]

Tietokannoissa halutaan myös yhdistää tauluja jollain kriteereillä. Tämä usein tehdään sisäkkäisillä silmukoilla, joista ulompi silmukka käy ensimmäisen taulun läpi ja toinen silmukka vertaa jokaista ensimmäisen taulun arvoa jokaisen toisen taulun arvoihin. Silmukat on vektorisoitu valitsemalla ensimmäisestä taulusta monia arvoja

joilla täytetään vektoritietorakenne, sitten vektorisaatiolla verrataan kaikkia vektoritietorakenteessa olevia avainarvoja samanaikaisesti johonkin kohdetaulun tiettyyn avaimeen.[13]

3.3 Esimerkkejä vektorisoinnin käytöstä neuroverkoissa

Konenäkötehtäviin neuroverkkojen koulutus vaatii valtavan määrän datan prosessointia. Tärkeät osat neuroverkkojen koulutusta ja testaamista ovat matriisi- ja vektorioperaatiot, jotka vievät paljon prosessointiaikaa. Suuri ongelma neuroverkkojen parannuksessa on ollut korkean laskentatehon laitteistojen heikko saatavuus, mikä on muuttanut alan keskittymisen tehokkuuden saamiseen rinnakkaisilla algoritmeillä. Rinnakkaisuuden, varsinkin vektorisoinnin käyttö on suuresti parantanut neuroverkostojen tehokkuutta.[14]

Neuroverkkoon kuuluu monta kerrosta, jossa tehdään eri operaatioita. Valtaosiin kerroksista on keksitty tapoja hyödyntää vektorisaatiota. Konvoluutio (engl. *convolution*)kerrokseen kuuluu matriiseihin, sekä vektoreihin liittyviä matemaattisia operaatioita. Kerroksessa muunnetaan vektoreita matriiseiksi, joiden käyttöä on helpompi vektorisoida. Kerroksessa kerrotaan matriiseja toisillaan, sitä varten valmiiksi tehdyllä vektorioperaatiolla.[14]

Toiseen kerrokseen kuuluu yhdistäminen (engl. *pooling*). Kerroksessa yhdistetään monia matriiseja samaan matriisiin. Tämä on vektorisoitu yhdistämällä monia matriiseja samanaikaisesti.[14]

4 Vektorisoiminen koodissa

Vektorisointi on osa datarinnakkaisuutta. Vektorisoinnissa keskitytään koodissa silmukoihin. Silmukat ovat koodin osa, jotka tekevät halutut operaatiot halutun määrän kertoja. Vektorisaation avulla voidaan tehdä sama operaatio monelle elementille samaan aikaan, mikä lyhentää silmukan suorittamisen vaativaa aikaa. Vektorisoinnissa elementit laitetaan vektoriin ja operaatio tehdään koko vektorille samanaikaisesti. Elementtien määrä joille operaatio voidaan sehdä samanaikaisesti riippuu vektorin pituudesta, ja tehtyjen operaatioiden määrä on aina sama kuin vektorin pituus. Myös peräkkäin tehdyt operaatiot ilman silmukkaa on vektoroitavissa, mutta tämä on harvoin käytännöllistä.[2].

Vektorisointi tapahtuu eri vektorisointiin luoduilla käskykannoilla (engl. *instruction set*). Eri vektorisaatioon keskittyvät kannat erottuvat niiden käsiteltyjen vektorien maksimikoosta ja niiden kyvystä suorittaa eri operaatioita. Yleensä laitteistolla on käytössä vain yksi käskykanta, joka tulee laitteiston valmistajilta. Uudemmissa laitteissa on myös usein kehittyneempi ja uudempi käskykanta. Näistä vektorisoivista käskykannoista nykyään suosittuja ovat SSE- (*Streaming SIMD Extensions*) ja myöhemmin tulleet AVX-kannat (*Advanced Vector Extensions*).[2]

4.1 Manuaalinen vektorisointi

Vektorioperaatioiden manuaalisessa käytössä pitää olla varuillaan laitteistovaatimuksista. Prosessorin tai grafiikkasuorittimen sekä käyttöjärjestelmän pitää pystyä

käyttämään valittuja vektorioperaatioita.[2] Varsinkin vektorikäskykantojen eri versioita pitää ottaa huomioon. Se tekee manuaalisesti vektorisoidusta ohjelmasta eri laitteiston välillä huonosti toimivan.[15] Ohjelmiin joutuu usein lisäämään monia versioita eri käskykannoille. Ohjelman alussa voi testata, mikä on paras saatavilla oleva käskykanta laitteistolle. Sen perusteella voi valita oikean ohjelman version.[2] Manuaalinen vektorisointi on myös hyvin altista virheille ja vaikea ylläpitää[15].

Yksi tapa lisätä vektorisointia koodiin on kirjoittaa suoraan koodiin vektorisoivaa assembly-kieltä. Kirjoittamalla assembler-kielen koodia muun ohjelmointikielen koodin sekaan saadaan suuri määrä hallintaa, siitä mitä ohjelma tekee. Sen avulla voidaan lisätä vektorisointia melkein mihin vain kohtaan ohjelmaa. Assembly-kielen kirjoittaminen on kuitenkin hyvin monimutkaista ja työlästä kirjoittaa. Vektorisointi sen avulla vaatii taitoa ja aikaa.[3]

Toinen keino on käyttää luontaisia funktioita, jotka ovat valmiiksi tehtyjä funktioita, jotka käsittelevät vektoreita. Niiden käyttö on huomattavasti helpompaa kuin assemblyn kirjoittaminen. Mutta ne eivät takaa parasta ohjelman optimointia.[3] Luontaiset funktiot hoitavat assemblyn ohjelmoijan puolesta. Kuten assemblyn kirjoittamisen vaikeat osuudet niinkuin rekistereiden varaukset ja funktiokutsut. Luontaisia funktioita on tuhansia, joten oikean löytäminen saattaa olla hankalaa.[2]

Ohjelmointikieliin, jotka sisältävät luokkia on kolmas keino lisätä vektorisaatiota koodiin käyttämällä vektoriluokkia. Ne ovat vielä helpompia käyttää kuin luontaiset funktiot. Lisäksi niiden käyttö luo vaihtoehdoista selvintä koodia, mutta toisaalta ne antavat vähiten varmuutta optimoinnista.[3]Vektoriluokka sisältää luontaisia funktioita, joita voi kutsua helposti ja jotka tuottavat selvempää koodia.[2].

Ohjelmalistaus 2 Esimerkki luontaisten funktioiden käytöstä.

```
#include <emmintrin.h> // SSE2 luontaisten funktioiden lisäys


//Lukee taulukon kohdasta p 128 bittiä kok.lukudataa vektoriin
static inline __m128i LataaVektori(void const * p) {
    return _mm_loadu_si128((__m128i const*)p);
}
//Tallentaa vektorin kaikki 128 bittiä taulukkoon kohtaan d
static inline void VarastoiVektori(void * d, __m128i const & x) {
    _mm_storeu_si128((__m128i *)d, x);
}
void LisääVektoriin(short int aa[]) {
    // Luo vektorin ja täyttää sen 16 kpl:lla 16-bittisiä kakkosia
    __m128i two = _mm_set1_epi16(2);

    //Silmukka: käy läpi taulukon vektorin levyisinä (8) siivuina
    //vektorin elementtien määrä
    for (int i = 0; i < 256; i += 8) {
        // Lataa 8 numeroa taulukosta aa kerrallaan vektoriin b
        __m128i b = LataaVektori(aa + i);

        // Lisää kaksi jokaiseen vektorin b elementtiin kerrallaan
        __m128i c = _mm_add_epi16(b, two);

        //Talenna vektorin c elementit takaisin taulukkoon
        VarastoiVektori(aa + i, c);
    }
}
```

Ohjelmalistaus 2 on C++-ohjelmointikielellä kirjoitettu ohjelma, jossa on kolme vektorisaatiota varten tehtyä funktiota. Ensin lisätään kirjasto, jonka nimenä on emmintrin.h. Kirjasto sisältää luontaiset funktiot. Sitten luodaan funktio, jonka palautuksen tyyppi on `m128i`. Tyyppi `m128i` kertoo, että funktio palauttaa 128 bitin kokoisen vektorin, johon mahtuu esimerkiksi kahdeksan 16 bitin täysnumeroa. Funktio ottaa taulukon osoittimen vastaa ja siirtää taulukon sisällön vektoriin SSE2 luontaisella funktiolla. Luontaiset funktioden nimet alkavat kirjaimilla mm. Esimerkin toinen funktio ottaa vastaan taas taulukon osoittimen ja vektorin. Se siirtää vektorin sisällön taulukkoon luontaisella funktiolla, eikä palauta mitään.

Esimerkin kolmas funktio ottaa vastaan taulukon täynnä 16-bittisiä täyslukuja, eikä palauta mitään. Funktiossa luodaan 128-bittinen vektori täynnä kakkosia luontaisella funktiolla. Funktiossa on myös silmukka, joka käy vektorin läpi sen elementtien määrän kokoisella hyppyvälillä. Jokaisen silmukan kierroksella ladataan taulukosta vektoriin hyppyvälin verran elementtejä samanaikaisesti. Sitten silmukassa lisätään luodun vektorin elementteihin kakkosilla täytetyn vektorin elementit samanaikaisesti. Lopuksi silmukkassa varastoidaan muunnettu vektori takaisin taulukkoon.

4.2 Automaattinen vektorisointi

Automaattinen kääntäjän vektorisointi on helpoin ja yksinkertaisin tapa lisätä vektorisointia ohjelmaan.[2] Monet kääntäjät tehostavat silmukat lisäämällä vektorisointia, jos niissä ei ole mitään vektorisaatiota estäviä tekijöitä. Ohjelmoija voi kirjoittaa koodia, joka on kääntäjälle helpompi vektorisoida välttämällä näitä tekijöitä. Ohjelmoija voi myös itse tietäessään koodin pätkän vektoroituvan kertoa kääntäjälle sen kääntäjädirektiiveillä. Kääntäjädirektiivit ovat komentoja, jonka avulla ohjelmoija voi vaikuttaa kääntäjään. Niiden käyttö voi lisää ohjelmoijalle paljon hallintaa siitä, mitkä kohdat koodia kääntäjä vektorisoi.[4] Kääntäjädirektiivit al-

kavat #pragma sanalla ja ne sijoitetaan ennen kohtia koodissa, joihin direktiivillä halutaan vaikuttaa[16].

Taulukko 4.1: Kääntäjädirektiivejä Intelin C++-kääntäjille.

Komento	Vaikutus
ivdep	Sivuttaa mahdolliset datan riippuvuudet
loop count (n)	Selventää silmukan tyypillisen kierrosmäärään.
vector always	kääntäjä aina vektorisoi pystyessä
vector align	Kaikki silmukassa oleva data on riville asetettu
novector	Kääntäjä ei vektorisoi silmukkaa
nontemporal	Dataa ei käytetä uudestaan

Taulukossa 4.1 on yleisiä vektorisoinnin yhteydessä käytettyjä kääntäjädirektiivejä C++-ohjelmointikielen Intelin kääntäjille. Komento ivdep kertoo kääntäjälle, että se voi turvallisesti jättää mahdolliset datan riippuvuudet välittämättä. Kääntäjä ei silti sivuta riippuvuuksia. Komento loop count (n) kertoo kääntäjälle monta kierrosta silmukassa tyypillisesti on. Se voi auttaa kääntäjää laskemaan onko silmukka vektorisoinnin arvoinen. Vector always on käsky kääntäjälle vektorisoida silmukka, jos se on mahdollista. Komentoa käyttäessä kääntäjä ei laske onko silmukka vektorisoinnin arvoinen. Novector komennolla kääntäjä ei vektorisoi silmukkaa. Komento nontemporal kertoo kääntäjälle, että dataa ei tulla käyttämään uudestaan.

Automaattista vektorisointia voi ja kannattaa myös yhdistää manuaalisella vektoroinnin kanssa. Monessa ongelmatilanteessa kääntäjän kanssa voi vektorisoida koodia manuaalisilla keinoilla.[2] Seuraavana käsitellään yleisimpiä vektorisaatiota estäviä tekijöitä.

Yksi estävä tekijä on se, että silmukoiden iteraatiot vaativat muuttuvaa tietoa. Kääntäjälle voi selventää mahdollisten muuttujien käyttäytymisestä kirjoittamalla silmukan käsittelemien elementtien aloitusarvoja.[4]

Ohjelmalistaus 3 Esimerkki muuttuvan elementin selventämisestä pseudokoodilla.

```
program main
    do i=1,ni
        do j=1,nj
         call foo(x,j,mode)
        enddo j
    enddo i
end program main

sivurutiini foo(x,itr,iflag)
    integer x
    integer itr
    integer iflag

    if (iflag == 1) then
       x = itr
    else if (iflag == 2) then
       x = itr*2
    end if
end sivurutiini foo
```

Ohjelmalistauksessa 3 sivurutiinia foo kutsutaan kaksinkertaisen silmukan sisältä. Sivurutiini päivittää muuttujan `x` vain jos `iflag` on 1 tai 2. Jos muuttuja `x` päivittyisi jokaisella silmukan kierroksella, voisi kääntäjä vektorisoida silmukan, koska edellisen silmukan kierroksen `x` arvoa ei tarvitsisi muistaa. Kääntäjä ei tiedä, että esimerkissä `iflag` on aina 1 tai 2, joten se jättää silmukan vektorisoimatta. Kääntäjää voi auttaa lisäämällä sivurutiiniin alkuarvon muuttujalle `x`, esimerkiksi nollan. Siitä kääntäjä voi olla varma, että muuttujan `x` arvoa ei tarvitse seuraavaan silmukan kierrokseen.[4]

Ongelma on kuitenkin yleisimmillään, kun käsitellään listaa silmukassa. Kääntäjä ei aina tiedä listan sisältöä kääntämisajankohdassa, eikä voi varmistaa sopiiko se vektorisointiin. Näissä tilanteissa on ohjelmoijan hyvä kertoa kääntäjälle kääntäjädirektiiveillä.[4]

Ohjelmalistaus 4 Esimerkki kääntäjädirektiivin käytöstä datariippuvuusongelmassa.

```
#pragma ivdep
void kopioi(char *cp_a, char *cp_b, int n) {
    for (int i = 0; i < n; i++) {
        cp_a[i] = cp_b[i];
    }
}
```

Ohjelmalistauksen 4 funktio `kopioi` ottaa kaksi osoitinta ja kokonaisluvun sisääntulona. Funktio kopioi osoittimen `n` elementtiä toiseen osoittimeen samassa silmukassa. Kääntäjä ei voi olla varma, että osoittimien sisällöt eivät ole riippuvia, joten kääntäjä jättää silmukan vektorisoimatta. Komento `#pragma ivdep` ennen funktiota kertoo kääntäjälle, että funktiossa ei ole datariippuvuutta (engl. *data dependency*) ja se on vektoroitavissa.

Toinen ongelma on, jos silmukassa käytetty muuttuja voisi olla määrittelemätön. Tämä saattaa tapahtua esimerkiksi entä-jos tilanteissa, joissa ei selvennetä millaiseksi muuttuja muuttuu. Tällaiset muuttujat estävät koodin inline-optimoinnin, ja sen takia myös vektoroinnin. [4] Ohjelmallistauksessa 3 sivurutiinissa muuttujan `iflag` arvoa katsotaan kahdella `else-if`-komennolla. Se jättää mahdollisuuden sille, että `iflag` voisi olla jokin muukin kuin `else-if`-komennoilla katsotut arvot. Jos ohjelmoija tietää, että `iflag` voi olla vain yksi tai kaksi, niin koodin voisi muuttaa `if-` ja `else`-komentoihin. Tämä poistaisi muut `iflag`-muuttujan mahdollisuudet ja kääntäjä voisi helpommin vektorisoida silmukan.

Kääntäjä ei voi vektorisoida silmukkaa, jos sen sisällä kutsutaan funktiota, joka ei ole inline-optimoitu. Ohjelmoija pystyy tässä tapauksessa manuaalisesti sijoittaa funktion yhdelle riville. Kääntäjä myös saattaa jättää lyhyet silmukat vektorisoimatta. Ohjelmoija voi yhdistää silmukat tilanteessa, jossa on monta pienempää silmukkaa sisäkkäin.[4]

Koodissa olevat hakutaulut voivat myös estää kääntäjän vektorisointia. Yleisesti

kääntäjä ei oikein pysty vektorisoimaan liian monimutkaista koodia. Kääntäjän kykyyn vektorisoida koodia voi vaikuttaa kirjoittamalla koodin yksinkertaisemmalla tavalla.[2]

Erilaisilla kääntäjillä on eri kykyjä vektorisoida ja niiden valinta vaikuttaa ohjelmien vektorisaation tehokkuuteen. Oletettavasti kannattaa valita hyvä kääntäjä vektorisointiin. Kirjoitushetkellä hyvin vektoroiviksi voi laskea mm. C/C++-kääntäjät GNU GCC, Clang (LLMV) tai Intel ICC. Moniin kääntäjiin tulee myös generoidun koodin suorituskykyä nopeuttavia päivityksiä, joten kääntäjiä kannattaa päivittää. Eri vektorienkäskykantojen hyödyllisyydet ovat vaihtelevia eri laitteistossa ja ohjelmistossa. Käyttöön kannattaa valita kääntäjän asetuksilla parhain käskykanta.[2]

Koodin rakenne voi myös nopeuttaa kääntäjän tekemää vektorisointia. Silmukan kierrosten luku on hyvä olla jaollinen sen käsittelemän tietorakenteen koolla. Ohjelmoija voi lisätä silmukkaan tyhjiä kierroksia, jotta näin on. Hakutaulujen käyttöä kannattaa myös välttää vektoritietorakenteen elementtejä käsitellessä.[2]

5 Yhteenveto

Luvussa 2 tarkasteltiin mitä rinnakkaisuus on laitteistossa ja ohjelmistossa. Yleisiä rinnakkaisia laitteistoja ovat moniydin-, vektori sekä grafiikkaprosessorit. Nykyisin monet prosessorit ovat hybridimallisia. Niissä on vektori- ja skalaariytimiä.

Rinnakkaisten laitteiden käyttöönottoa varten on tehty helpottavia ohjelmia. Tunnettuja esimerkkejä ovat OpenMP- ja POSIX Threads- ohjelmointirajapinnat prosessoreita varten ja näytönohjaimia varten NVIDIAn ohjelmointirajapinta CUDA.

Rinnakkaisuudessa käytetään usein rinnakkaisuusmalleja. Rinnakkaistamista jaetaan usein data- ja tehtävärinnakkaisuuteen. Kaksi esimerkkiä malleista olivat yksi ohjelma eri dataa sekä haarautus/yhdistysmalli. Näissä malleissa jaetaan tehtäviä eri tavalla hoidettaviksi.

Luvussa 3 käsiteltiin mihin vektorisointia yleensä käytetään katsomalla vektorisaation käyttökohteita. Vektorisointi on hyödyllistä, kun tehdään laskuja suurella määrällä dataa, ja kun sama operaatio tehdään monelle elementille. Tutkielmassa esiteltiin vektorisaation käyttöä fysiikassa, tietokannoissa, sekä neuroverkoissa. Fysiikan tutkimuksissa käytetään joskus simulaatioita, jotka vaativat paljon datan käsittelyä. Simulaatioissa on muunneltu algoritmeja paremmin vektorisoituiksi. Tietokannoissa on datan käsittelyyn monia operaatioita, joita on vektorisoitu. Neuroverkoissa käsitellään dataa monessa kerroksessa. Eri kerrokset sisältävät matriisien ja vektorien käsittelyä eri matemaattisilla operaatioilla, joita on vektorisoitu.

Luvussa 4 käsiteltiin miten vektoroituvaa koodia kirjoitetaan kertomalla manuaalisesta sekä automaattisesta vektorisoinnista. Manuaalisessa vektoroinnissa lisätään koodin vektorisointia manuaalisesti eri työkalujen avulla. Yksi tapa lisätä vektorisaatiota on kirjoittaa assembler-kieltä suoraan koodiin. Se antaa ohjelmoijalle suuren määrän hallintaa, mutta sitä on monimutkaista ja työlästä tehdä. Toinen keino on käyttää luontaisia funktioita tai vektoriluokkia. Näiden käyttö on assemblerkielen koodin kirjoittamiseen verrattuna helppoa ja nopeaa. Manuaalisessa vektoroinnissa pitää ottaa huomioon käytettyjen vektorikäskykantojen laitteistovaatimuksia.

Vektorisoivat kääntäjät vektorisoivat koodia ohjelmoijan puolesta automaattisesti. Automaattinen vektorisaatio on yksinkertainen tapa saada vektorisaatiota ohjelmiin. Kääntäjän pitää olla varma, että koodin vektorisaatioita ei estä mikään tekijä. Kääntäjää voi auttaa kirjoittamalla koodia, joka on selvästi vektorisoituvaa. Kääntäjälle voi myös kertoa tietoa koodista kääntäjädirektiiveillä.

Lähdeluettelo

[1] P. S. Pacheco, *An Introduction to Parallel Programming*. Boston: Morgan Kaufmann, 2011. DOI: https://doi.org/10.1016/C2009-0-18471-4.

[2] F. Agner, *Optimizing software in C++ An optimization guide for Windows, Linux, and Mac platforms*. maaliskuu 2004-2024, website accessed: 15.2.2024. url: https://agner.org/optimize/optimizing_cpp.pdf.

[3] H. Jeong, S. Kim, W. Lee ja S.-H. Myung, "Performance of SSE and AVX instruction sets", *CoRR*, vol. abs/1211.0820, marraskuu 2012.

[4] "Vectorization-aware loop optimization with user-defined code transformations", *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2017-September, s. 685–692, syyskuu 2017, ISSN: 15525244. DOI: 10.1109/CLUSTER.2017.102.

[5] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh ja D. Slotnick, "The Illiac IV system", *Proceedings of the IEEE*, vol. 60, nro 4, s. 369–388, 1972. DOI: 10.1109/PROC.1972.8647.

[6] R. M. Russell, "The CRAY-1 computer system", *Commun. ACM*, vol. 21, nro 1, s. 63–72, tammikuu 1978, ISSN: 0001-0782. DOI: 10.1145/359327.359336.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone ja J. C. Phillips, "GPU Computing", *Proceedings of the IEEE*, vol. 96, nro 5, s. 879–899, 2008. DOI: 10.1109/JPROC.2008.917757.

[8] J. Diaz, C. Muñoz-Caro ja A. Niño, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era", *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, nro 8, s. 1369–1386, 2012. DOI: 10.1109/TPDS.2011.308.

[9] C. Kessler ja J. Keller, "Models for Parallel Computing : Review and Perspectives", *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen*, vol. 24, s. 13–29, 2007.

[10] W. R. Martin, "Successful vectorization - reactor physics Monte Carlo code", *Computer Physics Communications*, vol. 57, nro 1, s. 68–77, 1989, ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(89)90193-8.

[11] Amadio, Guilherme, Ananya, Apostolakis, John et al., "Electromagnetic physics vectorization in the GeantV transport framework", *EPJ Web Conf.*, vol. 214, s. 02 031, 2019. DOI: 10.1051/epjconf/201921402031.

[12] O. Polychroniou, A. Raghavan ja K. A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases", sarja SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, s. 1493–1508, ISBN: 9781450327589. DOI: 10.1145/2723372.2747645.

[13] J. Zhou ja K. A. Ross, "Implementing database operations using SIMD instructions", sarja SIGMOD '02, Madison, Wisconsin: Association for Computing Machinery, 2002, s. 145–156, ISBN: 1581134975. DOI: 10.1145/564691.564709.

[14] J. Ren ja L. Xu, "On Vectorization of Deep Convolutional Neural Networks for Vision Tasks", *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, nro 1, helmikuu 2015. DOI: 10.1609/aaai.v29i1.9488. url: https://ojs.aaai.org/index.php/AAAI/article/view/9488.

[15] O. Reiche, C. Kobylko, F. Hannig ja J. Teich, ”Auto-vectorization for image processing DSLs”, *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference of Languages, Compilers, and Tools for Embedded Systems*, vol. 52, nro 5, s. 21–30, 2017, ISSN: 0362-1340. DOI: 10.1145/3140582.3081039.

[16] Intel, *A guide to Vectorization with Intel C++ compilers*, 2010. url: https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf.