Tuomas Pitkänen

Monte Carlo -hiukkassimulaation toteuttaminen ja kiihdyttäminen Pythonin Numba-kirjastolla

Tietotekniikan pro gradu -tutkielma

13. kesäkuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Tuomas Pitkänen

Yhteystiedot: tuomas.t.pitkanen@student.jyu.fi

Ohjaajat: Ilkka Pölönen (informaatioteknologian tiedekunta) ja Mikko Laitinen (matemaattis-luonnontieteellinen tiedekunta, fysiikan laitos)

Työn nimi: Monte Carlo -hiukkassimulaation toteuttaminen ja kiihdyttäminen Pythonin Numba-kirjastolla

Title in English: Implementing and accelerating a Monte Carlo particle simulation with Python using the Numba library

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 89+0

Tiivistelmä: Tutkielmassa vertaillaan alkuperäisen C-kielisen MCERD-simulaatiosovelluksen ja siitä tutkielmaa varten kehitettyjen Python-kielisten versioiden suorituskykyä. MCERD simuloi Elastic Recoil Detector (ERD) ja Rutherford Backscattering (RBS) -mittauksia Monte Carlo -menetelmällä. Uusia toteutuksia on kolme: puhtaalla Pythonilla tehty ohjelma sekä tästä Numballa kiihdytetyt, yksi- ja monisäikeiset versiot. Lisäksi tarkastellaan Nvidian CUDA-näytönohjainalustaa hyödyntävän Numba-version toteutettavuutta. Havaittiin, että Numballa saavutettiin lähes C:n tasoinen suorituskyky yhdellä säikeellä, ja monella säikeellä ylitettiin alkuperäinen, joskaan ei lineaarisesti skaalautuen. Havaittiin myös, että pelkkää Pythonia käyttävä versio on hyvin hidas ja näytönohjainversion toteuttaminen olisi vaikeaa.

Avainsanat: MCERD, Monte Carlo -menetelmä, ERD, RBS, simulaatio, Numba, rinnakkaistaminen, GPGPU

Abstract: This thesis compares the original MCERD simulation program made in C to new version of it made in Python for the thesis. MCERD simulates Elastic Recoil Detector (ERD) and Rutherford Backscattering (RBS) measurements using the Monte Carlo method. There are three new versions: one made in pure Python and two accelerated with Numba for

single-threaded and multi-threaded usage. The viability of a GPU version for Nvidia's CUDA platform using Numba is also considered. It was observed that Numba almost matched the performance of C when single-threaded, and exceeded it when multi-threaded, albeit not scaling linearly. It was also observed that the pure Python version is very slow, and that the GPU version would be difficult to implement.

Keywords: MCERD, Monte Carlo method, ERD, RBS, simulation, Numba, parallelization, GPGPU

Termiluettelo

C	on käännettävä, nopea ohjelmointikieli, joka on suhteellisen matalatasoinen.
CPython	on Pythonin viitetoteutus (engl. *reference implementation*).
CUDA	(engl. *Compute Unified Device Architecture*) on Nvidian GPGPU-toteutus.
Decorator	on Pythonissa funktio tai muu kutsuttava objekti (engl. *callable object*), joka muuttaa toisen kutsuttavan objektin toimintaa.
Energiaspektri	kuvaa tarkasteltavan hiukkasen määrää energian funktiona.
ERD(A)	(engl. *Elastic Recoil Detection (Analysis)*) on menetelmä, jossa ohuen näytteen rakennetta tutkitaan ampumalla siihen kiihdytettyjä hiukkasia ja tarkkailemalla näytteestä rekyloivia hiukkasia.
get_espe	(engl. *get energy spectrum*) on ohjelma, joka muuttaa MCERDin generoiman datan energiaspektriksi.
GPGPU	(engl. *General-purpose computing on graphics processing units*) tarkoittaa näytönohjaimen käyttämistä yleiseen laskentaan 3D-grafiikan sijaan.
GSTO	on ohjelma, joka tuottaa jarruuntumisenergioita alkuaine- tai isotooppiparien vuorovaikutukselle. GSTO on nykyään osa JIBAL-kirjastoa.
Hyperthreading	on Intelin toteutus SMT-teknologiasta. Se kaksinkertaistaa loogisten ytimien määrän.
JIBAL	on hiukkasdatakirjasto, joka koostaa useita eri lähteitä.
JIT-kääntäjä	(engl. *Just-In-Time compiler*) eli ajonaikainen kääntäjä kääntää ohjelmakoodia suorituksen aikana, jotta ohjelman suorituskyky paranisi.
Looginen ydin	on suoritettavien ohjelmien kannalta erillinen prosessoriydin.
MCERD	on ERD- ja RBS-simulaatio-ohjelma, joka käyttää Monte Carlo -menetelmää.

Monte Carlo -menetelmä	on keino tuottaa tilastollista dataa toistamalla laskua satunnaisilla syötearvoilla.
Numba	on JIT-kääntäjä ja kirjasto Pythonille, sekä rajapinta GPGPU:lle.
Potku	on ERD- ja RBS-analyysiohjelma, joka käyttää mm. MCERDiä.
Python	on tulkattava ohjelmointikieli, joka on melko korkeatasoinen.
Ohutkalvo	(engl. *thin film*) on substraatille valmistettava pinnoite. Kalvon tyypillinen paksuus on nanometriluokkaa.
RBS	(engl. *Rutherford Backscattering Spectrometry*) on menetelmä, jossa ohuen näytteen rakennetta tutkitaan ampumalla siihen kiihdytettyjä hiukkasia ja tarkkailemalla niiden takaisinsirontaa (engl. *backscattering*).
Rekylointi	(engl. *recoiling*) tarkoittaa ionien irtoamista näytteestä siihen törmänneen atomin liike-energialla.
ROCm	(engl. *Radeon Open Compute*) on AMD:n GPGPU-toteutus.
SMT	(engl. *simultaneous multithreading*) on tekniikka, jolla fyysisiä prosessoriytimiä voidaan monistaa useaksi loogiseksi ytimeksi lisäämällä samalle ytimelle monta toiminnanohjauskokonaisuutta.
Staattinen kääntäjä	kääntää ohjelmakoodin ennen sen suorittamista.
Substraatti	on ohutkalvon pohjamateriaali.
Syvyysprofiili	on tutkittavan näytteen alkuaine- tai isotooppijakauma eri syvyyksissä.
Tyyppiannotaatio	(engl. *type annotation*) on koneluettava merkintä lähdekoodissa, joka kertoo käytettävän arvon tietotyypin.
Ydinfunktio	(engl. *kernel function*) on GPGPU:ssa näytönohjaimella suoritettava laskuoperaatiosarja.
Yleiskustannus	(engl. *overhead*) tarkoittaa ohjelman tapauksessa niitä käytettyjä resursseja, jotka eivät suoraan edistä ohjelman tavoitetta, mutta jotka tarvitaan ohjelman suorittamiseen.

Kuviot

Taulukot

Sisältö

1 Johdanto

Jyväskylän yliopiston kiihdytinlaboratoriossa tutkitaan ohutkalvotekniikkaa hiukkaskiihdytinmittauksilla. Ohutkalvoja hyödynnetään monissa eri sovellutuksissa: niitä käytetään optiikassa muokkaamaan linssien heijastavuutta ja muita ominaisuuksia, puolijohteissa pienentämässä komponenttien kokoja sekä optoelektroniikassa osana valonlähteitä ja sensoreita (Alca Technology 2021). Käytetyt kalvot ovat paksuudeltaan alle nanometristä noin sataan mikrometriin. Kalvojen ohuus tekee niistä haastavia tuottaa ja tarkastella, sillä pienikin epäpuhtaus tai poikkeama paksuudessa voi olla kalvon kokoon suhteutettuna suuri ja tehdä kalvosta sopimattoman käyttötarkoitukseensa.

Ohutkalvojen tarkastelun kustannustehokkaina apuvälineinä ovat simulaatiot. Mittausasetusten selvittäminen hiukkaskiihdyttimellä kokeilemalla olisi hidasta ja kallista, joten ohutkalvon oletetun koostumuksen ja simulaatioiden perustella voidaan selvittää sopivat asetukset. Jotta simulointi olisi nopeaa, simulaatio-ohjelman on oltava hyvin optimoitu ja käytettävissä on oltava riittävästi laskentatehoa. Tässä gradussa tutkitaan simulointiin käytettävien valittujen ohjelmointitekniikoiden soveltuvuutta ja suorituskykyä hiukkassimulaatioihin.

Gradun tutkimuksen kohteena olevalla MCERD-ohjelmalla (engl. *Monte Carlo Elastic Recoil Detection*) simuloidaan hiukkaskiihdytinmittauksia Monte Carlo -menetelmällä, jossa simulaatiodataa tuotetaan toistamalla laskua eri lähtöarvoin. Monte Carlo -menetelmä on hyödyllinen erityisesti silloin, kun simuloitavaa ilmiötä on vaikeaa tai mahdotonta esittää suoraan analyyttisessä muodossa, jolloin sitä täytyy tarkastella muilla tavoilla, esimerkiksi simuloimalla.

Monte Carlo -menetelmä rinnakkaistuu yleensä hyvin, koska toistettavat laskut ovat yleensä toisistaan riippumattomia. Tyypillisessä nykyprosessorissa on kymmenkunta yleiskäyttöistä suoritusydintä, joilla voidaan laskea rinnakkain. Suurilla syötemäärillä laskettavaa on laskuydinten määrään nähden paljon, jolloin kaikkien syötteiden laskemiseen kuluu huomattavasti aikaa. Sen sijaan näytönohjaimissa on tuhansia tietyntyyppisiin laskuihin erikoistuneita suoritusytimiä, joilla voidaan käsitellä laajojakin aineistoja nopeasti. Näytönohjaimen käyttöä yleiseen laskentaan kutsutaan termillä GPGPU (engl. *General-purpose computing on*

graphics processing units). Luonteensa puolesta Monte Carlo -menetelmä sopisi hyvin näytönohjaimella laskettavaksi. Haasteina GPGPU:ssa ovat hitaiden datansiirto-operaatioiden minimointi sekä keskussuorittimella suoritettavasta koodista eroava rakenne.

Tutkimuskysymykset ovat:

1. Voidaanko korkeatasoisella ja hitaanakin tunnetulla Python-ohjelmointikielellä saavuttaa C-kieleen verrannollinen suorituskyky MCERD-ohjelmassa?
2. Miten valittu Python-nopeutusteknologia soveltuu MCERD-ohjelman toteuttamiseen verrattuna alkuperäiseen C-kieleen?
3. Kuinka paljon rinnakkaistaminen nopeuttaa MCERD-ohjelman suorittamista?

Tutkielman rakenne on seuraava: luvussa 2 tarkastellaan alkuperäistä MCERD-ohjelmaa ja sen fysikaalista perustaa. Luvussa 3 käsitellään C- ja Python-kieltä, GPGPU-tekniikkaa sekä teknologioita Python-koodin suorituksen nopeuttamiseen. Luku 4 käsittelee uusien MCERD-versioiden toteuttamista, jonka jälkeen luvussa 5 esitellään saatuja tuloksia. Luvussa 6 analysoidaan saatuja tuloksia, tarkastellaan kohdattuja haasteita ja esitetään aiheita jatkotutkimukselle. Lopuksi luvussa 7 on yhteenveto.

2 MCERD

MCERD on Helsingin yliopiston kiihdytinlaboratoriossa kehitetty Elastic Recoil Detection (ERD) -hiukkasimulaatiosovellus (Arstila, Sajavaara ja Keinonen 2001). Myöhemmin sitä on kehitetty Jyväskylän yliopiston kiihdytinlaboratoriossa (University of Jyväskylä 2020a).

MCERD simuloi mittausdataa käyttäjän määrittelemällä näyte- ja ilmaisinkonfiguraatiolla fyysisten mittausten tueksi. Simulaatiot ovat alkuaine- tai isotooppikohtaisia, joten ohjelmaa ajetaan yleensä useita kertoja näytettä kohden. Tuotetusta mittausdatasta voidaan muodostaa energiaspektrejä mukana tulevalla get_espe-ohjelmalla.

Tässä luvussa selvitetään MCERD-ohjelman ERD-menetelmän fysikaaliset perusteet, jonka jälkeen tarkastellaan ohjelman rakennetta.

2.1 Fysikaalinen perusta

Elastic Recoil Detection on menetelmä, jossa näytteen koostumus selvitetään irroittamalla siitä hiukkasia hiukkaskiihdyttimellä törmäyttämällä (Bik ja Habraken 1993). Irtohiukkasista eli rekyyli-ioneista mitataan ilmaisimilla (engl. *detector*) suureita, joiden perusteella hiukkasen alkuaine (tai isotooppi) ja lähtösijainti näytteessä tunnistetaan. Menetelmää sovelletaan ohuiden näytekalvojen koostumuksen selvittämiseen.

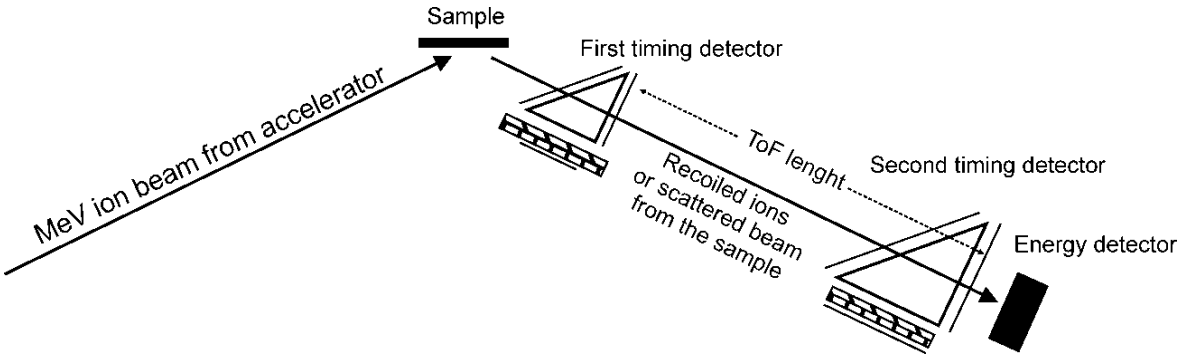
ERD perustuu hiukkasten elastiseen vuorovaikutukseen: kun hiukkaskiihdyttimen hiukkassuihkussa eli beamissa olevat ionit (engl. *incident ion*) törmäävät näytteeseen, ne aiheuttavat liike-energian säilyttävää kimpoilua näytteen atomien välille (Assmann ym. 1994). Liike-energian jakautuminen saa aikaan hiukkasten rekyloitumista ja siroamista näytteestä.

Näytteessä edetessään beamin ionit jarruuntuvat näytteen atomien elektronikenttien kanssa tapahtuvan vuorovaikutuksen seurauksena (Bik ja Habraken 1993). Alkuperäinen beamin ioni voi mm. pysähtyä näytteeseen tai läpäistä sen. Jos käytetty ioni on kevyt verrattuna näytteen ioneihin, se voi myös palata takaisinsirontana (engl. *backscatter*).

Vuorovaikutusten lopputulosten esiintyvyydet ja hiukkasten lentoradat vaihtelevat tapaus-

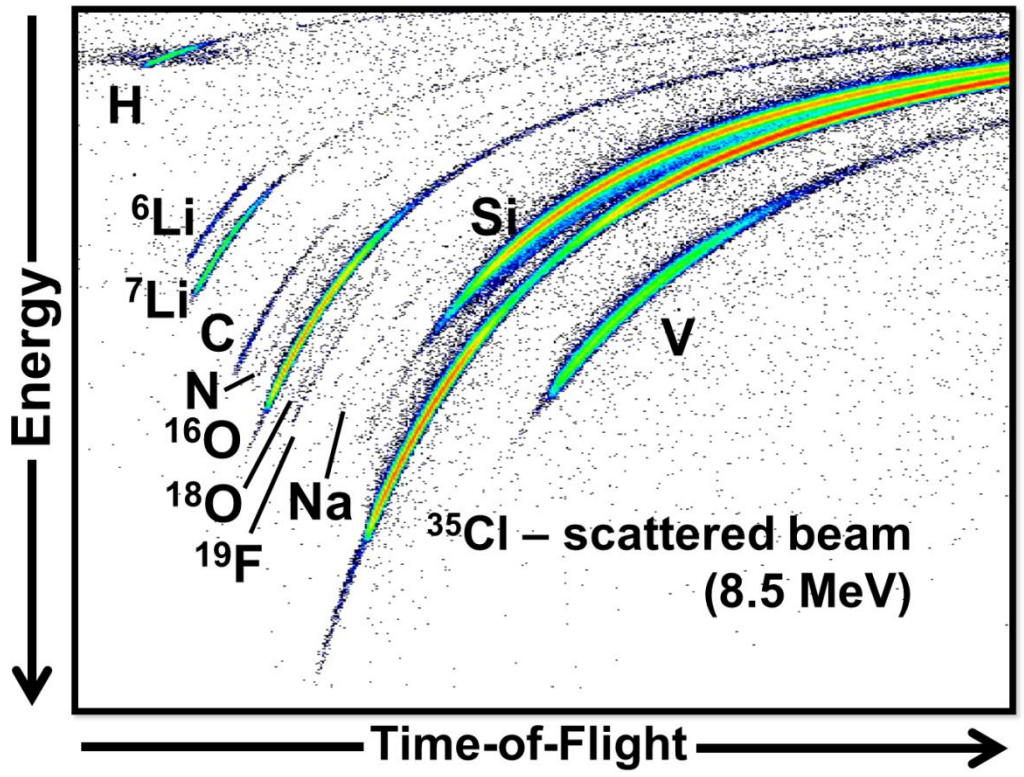
kohtaisesti, sillä niihin vaikuttavat mm. törmäävien hiukkasten massa, liike-energia, keskinäinen kulma sekä osumakohta. Koska muuttujia on useita, jokaiselle mittaukselle pitää selvittää oma konfiguraationsa, jotta mahdollisimman moni irronnut hiukkanen havaitaan. Konfiguraatiota voidaan säätää vaihtamalla käytetyn beamin ionityyppiä ja energiaa, sekä muuttamalla mittauslaitteiston kulmia hiukkaskiihdyttimen, näytteen ja ilmaisimien osalta.

Eräs ERD-menetelmän muoto on TOF-ERD (engl. *Time-of-Flight Elastic Recoil Detection*), jossa tarkastellaan hiukkasten lentoaikaa (Hellborg, Whitlow ja Zhang 2010, s. 177–178). Kuviossa 1 on havainnekuva mittauslaitteistosta, joka koostuu hiukkaskiihdyttimestä (engl. *accelerator*), näytteestä (engl. *sample*), kahdesta aikailmaisimesta (engl. *timing detector*) ja yhdestä energiailmaisimesta (engl. *energy detector*). Aikailmaiset mittaavat hiukkasen lentoaikaa tietyllä matkalla ja energiailmaisin liike-energiaa, joista saadaan aikaan lentoaika–energiahistogrammi 2. Mitatuista tiedoista voidaan myös päätellä rekyloituneen atomin tyyppi ja syvyys näytteessä, jolloin saadaan muodostettua näytteen syvyysprofiili.



Kuvio 1. TOF-ERD-laitteisto (Laitinen 2013, s. 2).

ERD-menetelmään liittyy läheisesti *Rutherford Backscattering Spectrometry* (RBS) -menetelmä, jossa tarkastellaan beamin ionien takaisinsirontaa (Bik ja Habraken 1993). Siinä beamin ioni ja energia ovat ERD-menetelmässä käytettyjä kevyempiä, sillä suuremman liike-energian ionit eivät yleensä siroaisi takaisin, vaan uppoaisivat näytteeseen. Menetelmissä käytetään samankaltaisia ilmaisimia, vaikkakin eri siroamissuuntien vuoksi ne ovat eri puolilla näytettä.



Kuvio 2. Lentoaika–energiahistogrammi (Laitinen 2013, s. 3).

2.2 Ohjelman rakenne

Arstilan, Sajavaaran ja Keinosen (2001) mukaan ERD-menetelmässä esiintyvää moninkertaista sirontaa (engl. *plural scattering*) eli ionin kimpoilua näytteessä ei voida tarkastella analyyttisesti suuren kulman sironnan (engl. *large angle scattering*) tapauksessa. Analyyttisessä tarkastelussa ionin polku näytteessä approksimoidaan suoraksi linjaksi, mikä johtaa vääristyneeseen dataan, jos polun suunta muuttuu merkittävästi kesken etenemisen. Tarkempaa dataa saadaan tuotettua Monte Carlo -menetelmällä simuloimalla, jolloin yksittäisten ionien liikerataa näytteessä voidaan seurata törmäyksestä toiseen, eikä etenemistä tarvitse tällöin approksimoida. Tutkielmassa käsiteltävä MCERD-ohjelma käyttää tätä menetelmää hiukkasmittausdatan tuottamiseen. Monte Carlo -menetelmälle tyypillisesti simulaation variointi tapahtuu satunnaislukugeneraattoria käyttämällä, MCERDin tapauksessa ionien osumakohtien, siroamiskulmien ja isotooppien valintaan.

Syötteenään MCERD ottaa tekstimuotoisia asetustiedostoja, jotka määrittävät mm. mittauslaitteiston sijainnin ja rakenteen, näytteen koostumuksen kerroksittain, sekä isotooppien massaluvun ja suhteellisen esiintyvyyden (University of Jyväskylä 2020a, read_input.c, read_target.c, read_detector.c). Tulosteena syntyy dataa, joka koostuu mm. ionien siroamisenergiasta ja massasta, syvyydestä näytteessä sekä ilmaisindatasta.

Ohjelman suoritus alkaa tietueiden alustamisella, konfiguraatiotiedostojen jäsentämisellä ja hakutaulujen generoinnilla. Tietueille asetetaan vakio- ja lähtöarvoja konfiguraatiotiedostojen mukaan. Yhteenveto olennaisimmista tietueista löytyy taulukosta 1. Hakutaulujen käyttö taas nopeuttaa simulointia, sillä niihin voidaan laskea raskaita laskutoimituksia kerralla ennakkoon ja sitten simulaation aikana interpoloida tarkempi arvo, jolloin laskuja miljoonia kertoja toistettaessa säästyy aikaa.

Simulaatio-osuus jakautuu kahteen päävaiheeseen: esisimulaatioon ja pääsimulaatioon. Vaiheet ovat samanlaiset, mikä mahdollistaa niiden suorittamisen samassa silmukassa (University of Jyväskylä 2020a, main.c). Suoritettu ohjelmakoodi valitaan ehtolauseilla, joissa tarkastellaan mm. simulaation vaihetta, hiukkasen tilaa ja sironnan tietoja. Vaiheiden näkyvin ero on sironneiden ionien käsittelyssä: esisimulaatiossa tallennetaan siroamistietoja myöhempää kulmansäätöä varten, kun taas pääsimulaatiossa ajetaan varsinainen ilmaisinkoodi.

Nimi	Kuvaus
Global	simulaation asetukset ja yleinen tila
Master (osa Global:ia)	tiedosto-osoittimet ja ohjelman argumentit
Ion	etenevä hiukkanen, simulaatiossa 2–3 kappaletta taulukossa
Potential	hakutaulu esilasketuille sähköpotentiaaleille
Scattering	hakutaulu esilasketuille sirontakulmille ja -energioille
SNext	nykyisen sironnan parametrit
Target	näyte
Detector	ilmaisimet

Taulukko 1. MCERD-ohjelman olennaisimmat ylätason tietueet (University of Jyväskylä 2020a, general.h).

Esisimulaation tehtävä on selvittää rekyylien avaruuskulmille sellainen väli, joka kattaa suurimman osan ilmaisimeen päätyvistä hiukkasista (Arstila, Sajavaara ja Keinonen 2001). Näin vältetään ilmaisimiin osumattomien hiukkasten tarpeetonta simulointia ja säästetään aikaa ilman että tulokset vääristyvät merkittävästi.

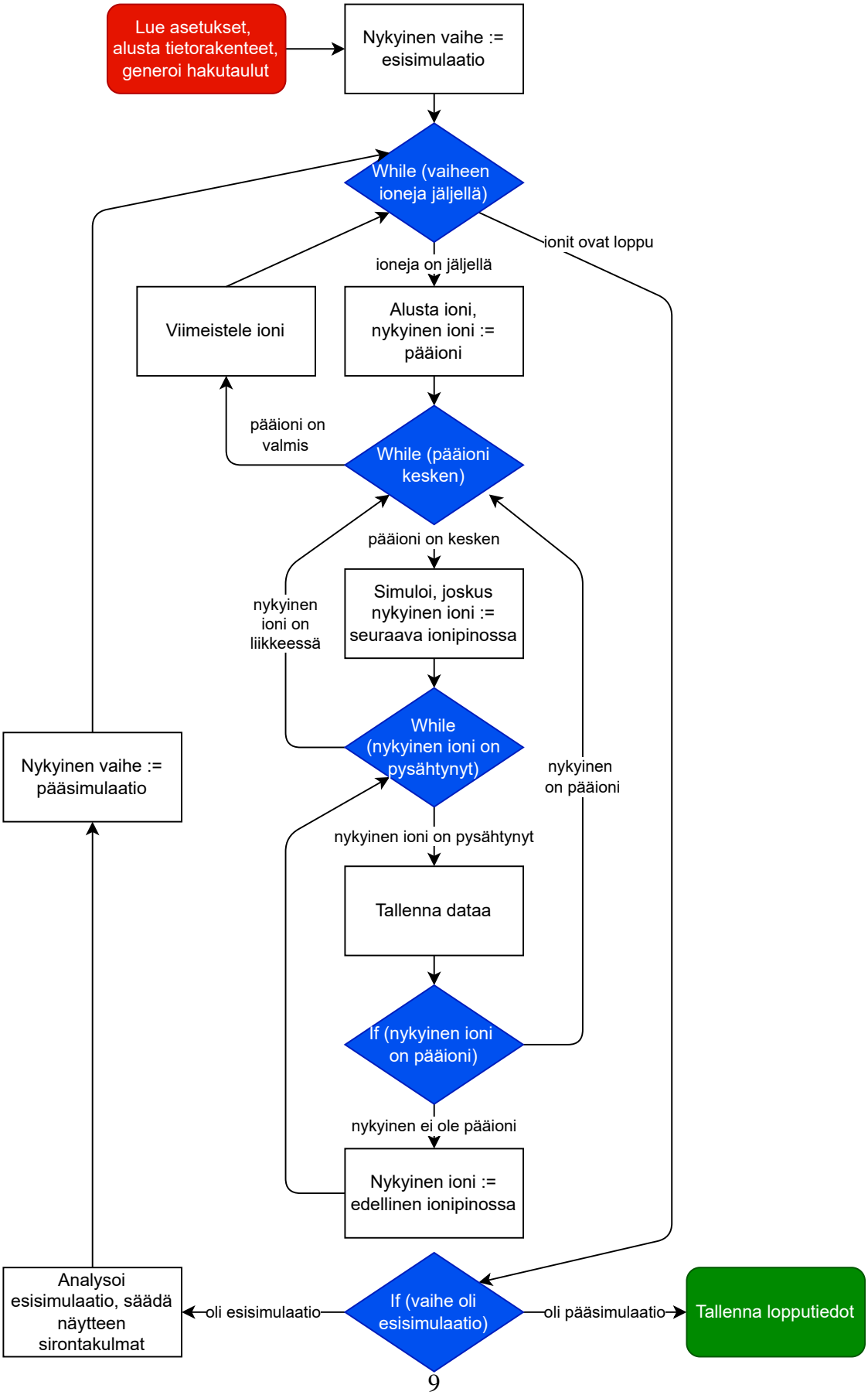
Pääsimulaatio puolestaan tuottaa varsinaisen simulaatiodatan. Simulaatio etenee hiukkanen kerrallaan, törmäyksestä toiseen. Näytteessä edetessään hiukkasen energia pienenee, kunnes hiukkanen pysähtyy tai päätyy näytteestä ulos. Mahdollisia hiukkasen tiloja on 11 erilaista (University of Jyväskylä 2020a, general.h), kuten ”ioni etenee”, ”ioni on läpäissyt näytteen”, ”ioni on sironnut takaisin näytteestä” (RBS) tai ”näytteestä on irronnut rekyyli-ioni” (ERD). Täysi listaus hiukkasten tiloista löytyy taulukosta 2.

Kuviossa 3 on yleiskatsaus MCERDin rakenteesta. Alku on merkitty punaisella ja loppu vihreällä pyöreäreunaisella suorakulmiolla. Ehto- ja toistolauseet ovat merkitty sinisinä ruutuina, tavalliset operaatiot suorakulmioina. Kuviossa esiintyvä ionipino tarkoittaa ionitaulukkoa, jossa on näytteessä etenevä pääioni sekä yhdestä kahteen apuionia, riippuen siitä onko kyseessä ERD- vai RBS-simulaatio. ERD-simulaation tapauksessa apuioni on näytteestä siroava ioni. Ionipino mahdollistaa simulaatiokoodin uudelleenkäytön vaihtelemalla ”nykyistä”, käsiteltävää ionia. Kuvion ”Simuloi, joskus nykyinen ioni := seuraava ionipi-

Tilan nimi	Kuvaus
NOT_FINISHED	Ioni etenee
FIN_STOP	Ioni on pysähtynyt
FIN_TRANS	Beamin ioni on läpäissyt näytteen
FIN_BS	Beamin ioni on sironnut takaisin näytteestä
FIN_RECOIL	Näytteestä on rekyloitunut ioni
FIN_NO_RECOIL	Näytteen ionia ei voinut rekyloida liian suuren kulman vuoksi
FIN_MISS_DET	Rekyyli-ioni ei osunut johonkin ilmaisimen aukkoon
FIN_OUT_DET	Rekyyli-ioni tuli ulos ilmaisimien välistä
FIN_DET	Rekyyli-ioni tuli ulos viimeisestä ilmaisinkerroksesta
FIN_MAXDEPTH	Beamin ioni saavutti rekyloinnin maksimisyvyyden
FIN_RECTRANS	Rekyyli-ioni läpäisi näytteen

Taulukko 2. Hiukkasen mahdolliset tilat.

nossa” -kohta sisältää ohjelman laskennallisesti raskaimmat osat, ehtolauseiden määrittämää laskemista ja nykyisen ionin vaihtamista.



Kuvio 3. Yleiskatsaus MCERDin rakenteesta.

3 Toteutusteknologiat

Ohjelmia voidaan tuottaa monella tavalla. Tässä luvussa tarkastellaan C- ja Python-ohjelmointikieliä, näytönohjaimen käyttämistä yleislaskennassa sekä keinoja tehostaa Python-koodin suorittamista.

3.1 Tarkasteltavat kielet ja teknologiat

Yleiskäyttöisiä ohjelmointikieliä, kuten C:tä ja Pythonia, yhdistää se, että niillä voidaan ratkaista samanlaisia tehtäviä. Niillä voidaan esimerkiksi muokata tekstiä, luoda graafisia käyttöliittymiä sekä kommunikoida internetissä, olettaen että niihin on kehitetty tähän soveltuvat rajapinnat. Tästä syystä tavallisten ohjelmointikielten ero ei ole niinkään siinä, mitä niillä pystytään tekemään, vaan siinä, miten ohjelma lopulta toteutetaan. Merkittäviä kielten eroja ovat esimerkiksi käytettävissä olevat ohjelmointiparadigmat, suoritusnopeus, kehittämisnopeus sekä kielen ominaisuus- ja tietoturvapäivitykset. Sopiva kieli vaihtelee tehtävittäin, esimerkiksi käyttöjärjestelmän kehityskielen on syytä mahdollistaa nopeat ja muistia tehokkaasti käyttävät ohjelmat, jotta käyttöjärjestelmä itse ei vie tarpeettoman suurta osuutta sen päällä pyörivien ohjelmien resursseista. Korkeamman tason sovelluksissa puolestaan tärkeämpiä ominaisuuksia voivat olla kehittämisen helppous ja kehityskustannusten minimointi, tuotettavan koodin suorituskyvyn ollessa toisarvoista.

C ja Python ovat monissa asioissa toisensa vastakohtia: siinä missä C on suhteellisen matalan tason kielenä nopea suorittaa mutta kehittäjälle työläs käyttää, Python keskittyy abstrahoimaan yleisiä toiminnallisuuksia helposti käytettäviksi mutta joskus hitaasti suoritettaviksi kokonaisuuksiksi. Keskussuorittimella ajettavien kielten lisäksi on olemassa näytönohjaimilla suoritettavia teknologioita, jotka vievät matalan tason ja nopeuden käsitteet äärimmilleen, sillä niillä voidaan valjastaa tuhansia suoritusytimiä sisältävät laitteet rinnakkaislaskentaan. Toisaalta GPGPU-teknologioille on ominaista kehittämisen vaikeus, sillä näytönohjaimet ovat toiminnaltaan keskussuorittimia erikoistuneempia mm. käskykannoiltaan ja muistinkäytöltään. Tämä näkyy näytönohjaimella suoritettavaksi soveltuvien tehtävien luonteessa: niissä käsitellään isoja määriä numeerista dataa, ja alkioille tehtävät operaatiot toistuvat

samanlaisina. Esimerkki GPGPU-teknologiasta on Nvidian CUDA-alusta.

3.1.1 C

C on yleiskäyttöinen ohjelmointikieli, joka kehitettiin vuosina 1969–1973 UNIX-käyttöjärjestelmän pääkieleksi (engl. *system programming language*) (Ritchie, Labs ja Hill 1993). Ritchie kertoo luoneensa C-kielen Ken Thompsonin B-kielen pohjalta, joka taas perustuu Martin Richards'n BCPL-kieleen. C-kielen lähtökohtana on olla korkeatasoisempi kuin assembly-kieli, mutta silti melkein yhtä tehokas. Käytännössä tämä tarkoittaa sitä, että yksi C:n operaatio vastaa muutamaa konekielistä operaatiota. Lisäksi tästä seuraa se, että tyypitys on staattista mutta heikkoa, jolloin yleensä epäyhteensopivilla datatyypeillä saatetaan tehdä laskutoimituksia vahingossa tai tarkoituksella. C-kieltä on levinnyt laajaan käyttöön sen tehokkuuden ansiosta, ja sillä on tehty esimerkiksi Pythonin viitetoteutus, CPython (Python Software Foundation 2022b).

C:n muistinhallinta on manuaalista (ISO/IEC 2007, s. 313–314), joten käyttäjän pitää itse huolehtia muistin tyhjentämisestä, varaamisesta ja muistiosoitteiden osoittimien käytöstä.

3.1.2 Python

Python on ohjelmointikieli, jonka kehitys alkoi 1990-luvun vaihteessa. Kielen kehittäjä, Guido van Rossum (2009), kuvailee suunnittelun lähtökohtia seuraavasti: ohjelmakoodin on oltava ulkoasultaan eleganttia, yksinkertaista ja luettavaa. Erikoismerkkejä on käytettävä harkitusti, ja niiden pitäisi toimia samalla tavalla kuin lukiomatematiikassa tai kirjoitetussa englannissa. Lisäksi käyttäjää ei pidä häiritä asioilla, jotka konekin voi hoitaa.

Python on dynaamisesti tyypitetty, joten samaan muuttujaan voidaan sijoittaa erityyppisiä arvoja, esimerkiksi ensin merkkijono ja sitten liukuluku (Hjelle 2022). Tyypitys on lisäksi vahva, joten implisiittisiä tyyppimuunnoksia esiintyy vähän, esimerkiksi merkkijonoa ja kokonaislukua ei voi laskea yhteen suoraan, vaan jompi kumpi arvoista pitää käsin muuttaa yhteensopivaksi datatyypiksi. Tämä ehkäisee vahinkoja. Huomattava poikkeus tähän on kokonaislukujen muuntaminen liukuluvuiksi tarvittaessa. Lisäksi totuusarvot *False* ja *True* käyttäytyvät kuten kokonaisluvut 0 ja 1.

Muistinhallinta on pääsääntöisesti automaattista: objektille allokoidaan muisti luonnin yhteydessä, ja muisti vapautetaan roskienkeruulla (engl. *garbage collection*) jossain vaiheessa ohjelman toimintaa (VanTol 2022). Roskienkeruu ei kuitenkaan poista kaikkea vastuuta resurssien käsittelystä: esimerkiksi auki jäänyt tiedosto voi jäädä ohjelman kannalta lukkoon, jos sitä yritetään käsitellä uudelleen, ennen kuin roskienkeruu tai ohjelmakoodi itse on sulkenut sen.

Pythonin standardikirjasto sisältää kattavasti valmiita toteutuksia yleisiin tehtäviin ”batteries included” -periaatteella. Standardikirjastoon kuuluvat esimerkiksi **re**-moduuli tekstioperaatioihin säännöllisillä lausekkeilla, **zipfile** ZIP-arkistojen käsittelyyn ja **sqlite3** SQLite 3 -tietokannan käyttöön (Foundation 2022). Kolmannen osapuolen moduuleille puolestaan on virallinen pakettivarasto (engl. *software repository*) Python Package Index (PyPI) (Python Software Foundation 2021).

3.1.3 GPGPU

GPGPU on menetelmä, jossa näytönohjaimien sadoilla tai tuhansilla laskentaytimillä prosessoidaan yleistä dataa esimerkiksi näytönohjaimille tyypillisen 3D-grafiikan sijaan. Tämä mahdollistaa suuren rinnakkaistamisen asteen edullisesti keskussuorittimiin verrattuna (Nvidia 2022, luku 1.1.), olettaen että ohjelmasta on tehty GPGPU-käyttöön sopiva versio. Näytönohjaimet eivät voi suorittaa tavallista prosessorikoodia, vaan niitä pitää ohjata erityisen GPGPU-ohjelmointirajapinnan avulla (HEAVY.AI 2022). GPGPU-laskennalle on tyypillistä matala abstraktiotaso ja numeeriseen laskentaan erikoistuminen (Toss ja Gautier 2012).

3.2 Python-ohjelmien nopeuttaminen

Pythonin dynaamisuuden varjopuolena on sen suoritusnopeus: esimerkiksi tieteellinen laskenta voi olla paljon hitaampaa puhtaalla Pythonilla kuin esimerkiksi käännetyllä C-kielellä. Pythonin hitaus johtuu monen tekijän vaikutuksesta, joista yksi on sen CPython-viitetoteutuksen GIL (engl. *Global Interpreter Lock*) -toteutus, joka lukitsee Python-tulkin tilan kun siirrytään suorittamaan Python-koodia (Ramalho 2015). GIL heikentää yhden ja monen säikeen laskennan suorituskykyä. Toinen Pythonia hidastava tekijä on sen suunnittelufilosofia:

kaikki arvot ovat olioita (Lutz 2007, luku 4), mikä tekee niiden alustamisesta ja muokkaamisesta hitaampaa.

Tässä aliluvussa käsitellään Pythonin hitauden kiertämiseen luotuja teknologioita.

3.2.1 Nopeutusteknologiat

Käsiteltävät nopeutusteknologiat voidaan jakaa karkeasti kolmeen ryhmään: valmiita laskuoperaatioita tarjoaviin kirjastoihin, kääntäjiin sekä GPGPU-kirjastoihin. Laskentakirjastot käyttävät varsinaisiin laskuihinsa tyypillisesti jotakin Pythonia nopeampaa kielellä, esimerkiksi C:tä tai Fortrania. Tämän lisäksi niissä on adapteriosa, jonka avulla dataa siirretään Pythonin ja muun laskentakoodin välillä.

CPython-viitetoteutuksesta poikkeavat kääntäjät tai tulkit taas suorittavat Python-lähdekoodia jollakin toisella teknologialla. Ne voivat korvata CPythonin kokonaan, kuten PyPy, tai osittain, kuten Numba. Kääntäjät voidaan ryhmitellä myös käännöksen ajankohdan mukaan, joko staattisesti ennen ohjelman suorittamista kääntäviin, kuten Cython, tai ensisijaisesti ajon aika kääntäviin, kuten Numba ja PyPy. Ajonaikaisesti kääntävissä toteutuksissa voi olla myös mahdollisuus esikääntää koodia myöhempää suorittamista varten.

GPGPU-kirjastot puolestaan tarjoavat Python-koodiin keinon hallita näytönohjaimen laskentaa, dataa ja konfiguraatiota. Ne toimivat rajapintana johonkin valmiiseen GPGPU-toteutukseen, kuten CUDAan tai OpenCL:ään.

3.2.2 Teknologiavaihtoehdot

Taulukossa 3.2.2 on yleiskatsaus nopeutusteknologiavaihtoehdoista.

NumPy on matriisilaskukirjasto, jonka keskeinen ominaisuus on tarjota tehokas toteutus *ndarray*-nimiselle, n-ulotteiselle taulukkotietotyypille (Harris ym. 2020). Kirjasto toimii perustana monelle muulle tieteellisen laskennan kirjastolle, mukaan lukien monelle tässä gradussa käsiteltäville nopeutusteknologialle, kuten PyCUDAlle, PyOpenCL:lle, Numballe ja CuPylle.

Nimi	Kehittäjä	Tyyppi
CUDA Python	Nvidia (2021)	CUDA-rajapinta*
CuPy	Preferred Networks (2020)	CUDA- ja ROCm-rajapinta
Cython	Behnel ym. (2021)	staattinen kääntäjä
Numba	Anaconda (2018)	JIT-kääntäjä, CUDA- & ROCm-rajapinta**
NumPy	Harris ym. (2020)	laskuoperaatiokirjasto
PyCUDA	Klöckner (2021a)	CUDA-rajapinta
PyOpenCL	Klöckner (2021b)	OpenCL-rajapinta
PyPy	The PyPy Team (2021)	JIT-kääntäjä
SciPy	Virtanen ym. (2020)	laskuoperaatiokirjasto

Taulukko 3. Python-kielisiä teknologiavaihtoehtoja. (* Hyvin varhainen versio. ** ROCm-tuen kehitys keskeytetty.)

SciPy on laskukirjasto, joka rakentuu NumPyn päälle (Virtanen ym. 2020). Se tarjoaa laskuoperaatioita mm. signaalinkäsittelyyn, optimointiin ja interpolointiin. Tutkielman kannalta NumPy ja SciPy ovat työkaluja datan säilöntään ja käsittelyyn, eivätkä täysipainoisia nopeutusteknologioita.

Cython on kääntäjä, joka muuntaa omalla syntaksillaan tyyppiannotoitua Python-lähdekoodia optimoiduksi C:ksi (Behnel ym. 2011). Sen kehittäjien mukaan käyttökohteita ovat ohjelmakoodin pullonkaulojen eliminointi muokkaamalla ne Pythonista Cython-kielisiksi, sekä valmiiden C-, C++- tai Fortran-kirjastojen integrointi Python-koodista kutsuttaviksi.

PyCUDA ja PyOpenCL ovat kääntäjiä, jotka toimivat rajapintoina näytönohjaimille (Klöckner ym. 2012). Suoritettavat ydinfunktiot määritellään merkkijonoina C:nkaltaisella kielellä, joista PyCUDA tai PyOpenCL generoi suorittavalle näytönohjaimelle optimoitua ohjelmakoodia. PyCUDA tuottaa Nvidian näytönohjaimilla toimivaa CUDA-koodia, kun taas PyOpenCL tukee muitakin alustoja, esimerkiksi AMD:n näytönohjaimia. Käsiteltävä data on NumPy-yhteensopivissa taulukoissa tai perustietotyyppeinä (Klöckner ym. 2012).

CuPy on NumPy:n syntaksiin perustuva matriisikirjasto CUDA-alustalle (Okuta ym. 2017). Lisäksi AMD:n ROCm-alustalle on kokeellinen tuki Linux-pohjaisilla käyttöjärjestelmil-

lä (Preferred Networks 2021b). Okutan ym. (2017) mukaan NumPy-yhteensopivuus mahdollistaa matriisilaskujen siirtämisen suorittimelta näytönohjaimelle pelkällä kirjastonvaihdoksella, tyypillisesti ilman muita muutoksia. CuPy:n yhteensopivuustaulukon (Preferred Networks 2021a) mukaan se toteuttaa merkittävän osan NumPy:n ja SciPy:n perusoperaatioista, matriiseista, lineaarialgebrasta, Fourier-muunnoksista sekä satunnaisluvuista. Lisäksi käyttäjä voi itse kirjoittaa CUDA-ydinfunktioita C-kielellä.

CUDA Python on Nvidian virallinen CUDA-rajapinta Pythonille (Nvidia 2021). Sitä ei ollut saatavilla tutkielman aloitusvaiheessa, joten se rajataan tarkastelun ulkopuolelle.

Numba on JIT-kääntäjä sekä GPGPU-rajapinta CUDA-alustalle (Lam, Pitrou ja Seibert 2015). Lisäksi siinä oli mukana tuki ROCm-alustalle, mutta tämä todettiin lopulta kannattamattomaksi, joten alustan integrointi keskeytettiin virallisesti versiossa 0.54.0 (Anaconda 2021). Kirjaston käyttö perustuu *@jit*-dekoraattoriin: sillä annotoidut funktiot käännetään optimoiduksi LLVM-infrastruktuurin tavukoodiksi. LLVM on puolestaan monikäyttöinen kääntäjä, jolla voidaan suorittaa natiivisti mm. C/C++-koodia, ja kolmannen osapuolen komponenteilla esimerkiksi Ruby- ja Rust-kieliä (LLVM Developer Group 2022). Numban tuetut käyttökohteet painottuvat numeeriseen laskentaan, ja niissä hyödynnetään erityisesti NumPy-kirjaston matriisityyppiä ja operaatioita.

PyPy on CPython-viitetoteutuksen kokonaan korvaava JIT-kääntäjä, joka on tehty sille varta vasten kehitetyllä RPython-kielellä (The PyPy Team 2022), esimerkiksi CPythonin C:n sijaan. RPython (*Restricted Python*) on nimensä mukaisesti Pythonin alijoukko, josta on rajattu suoritusta hidastavat osuudet pois. PyPyn tavoite on olla Pythonia nopeampi, mutta silti mahdollisimman yhteensopiva tavallisen Python-koodin kanssa.

Käsittelen Python-kielisen MCERD-toteutuksen nopeutusteknologian valintaa luvussa 4.1.

4 Testiohjelma

Tutkielmaa varten MCERD-ohjelma on kirjoitettu uudelleen Python-kielellä. Kehitys alkoi 2.11.2020 julkaistusta MCERD-versiosta, jossa on noin 4900 koodiriviä, 600 kommenttiriviä ja 1100 tyhjää riviä cloc-sovelluksella (Danial 2020) mitattuna. Lisäksi MCERD käyttää muita kirjastoja, erityisesti useita lähteitä koostavaa hiukkasdatakirjastoa JIBAL (Julin 2020) sekä satunnaislukugeneraattoria PCG (O'Neill 2018).

Tarkastelen tässä luvussa testiohjelman toteutusprosessia suunnittelusta toteuttamiseen. Analysoin toteuttamisen aikana kohtaamiani haasteita myöhemmin luvussa 6.3. Toteuttamisen lisäksi esittelen testilaitteiston sekä suunnitelmat suorituskykymittaukseen ja simulaatiotulosten oikeellisuuden verifiointiin.

4.1 Teknologian valinta ja suunnittelu

Toteutuskieleksi valitsin Pythonin, koska se on laajasti käytössä luonnontieteissä (Millman ja Aivazis 2011), sekä ohjelmointikielenä MCERDiä hyödyntävässä Potku-sovelluksessa (University of Jyväskylä 2020b), jota olen ollut jatkokehittämässä. Vaikka myös C-kielellä voisi rinnakkaistaa ohjelman ja käyttää GPGPU-rajapintoja, päätin vaihtaa toteutuskielen korkeamman tason ohjelmointikieleen toteutuksen ja testauksen nopeuttamiseksi sekä mahdollisen jatkokehittämisen helpottamiseksi.

Käytettäväksi nopeutusteknologiaksi valikoitui Numba kahdesta pääsyystä. Ensimmäinen on Python-koodin optimoinnin suoraviivaisuus, sillä Numban kehittäjien mukaan (Anaconda 2018) optimoitava koodi vaatii parhaimmillaan vain *@jit*-decoratorin lisäämisen aliohjelmaan. Näin siirtymä puhtaasta Python-koodista Numba-kiihdytettyyn versioon pitäisi olla yksinkertainen. Jotkin muut vaihtoehdot, kuten Cython (Behnel ym. 2011), vaativat koodin kirjoittamista niiden erityissyntaksilla, joka ei ole yhtä lähellä puhdasta Pythonia. Tällöin rinnakkaisversioiden, kuten puhtaan Pythonin ja kiihdytetyn version, tekeminen olisi työläämpää, eikä niiden välillä siirtyminen onnistuisi yhtä helposti. Toinen syy valintaan on Numban yhdistetty tuki sekä CPU- että GPU-laskennalle. Tällöin prosessorille kiihdytetystä toteutuksesta voisi olla helppoa siirtyä asteittain hyödyntämään näytönohjainta, ilman suur-

ten koodimuutosten tarvetta. Ihannetilanteessa näin saisi yhdistetyn ja helposti ylläpidettävän koodikannan näytönohjaimelle ja prosessorille, jossa vain GPGPU-osuudet vaatisivat oman erityiskoodinsa. Siirtymä näytönohjaimelle ei lopulta toteutunut, mistä kerron lisää osiossa 6.3.

Koska Numba on vielä kehitysvaiheessa, siihen liittyy todennäköisesti enemmän riskitekijöitä kuin virallisesti käyttövalmiisiin, ”1.x”-versioisiin kirjastoihin. Teknologiaa valitessani arvioin Numban suurimmiksi riskeiksi puuttuvan tuen koodin vaatimilta ominaisuuksilta, vähäisen dokumentaation sekä yleisen bugisuuden. Tuettujen CPU- (Anaconda 2022f, 2022e) ja GPU-ominaisuuksien (Anaconda 2022g) perusteella Numba vaikutti riittävän valmiilta kohtuullisen tasokkaaseen MCERD-toteutukseen, sillä tuki löytyy niin peruslaskutoimituksille kuin taulukkorakenteille. Käsittelen riskien toteutumista osiossa 6.3.

Valittuani käytettävät teknologiat, suunnittelin testiohjelman toteuttamisen tarkemmin. Päätin tehdä ohjelmasta iteratiivisesti kehittyneempiä versioita, jotta eri toteutusten vertailu lopuksi on helppoa. Ensimmäinen versio on suoraviivainen muunnos C:n syntaksista Pythoniin, ilman nopeutuksia. Sen pitäisi antaa samoja tuloksia kuin alkuperäisen ohjelman, kunhan sen satunnaislukugeneraattori vastaa alkuperäistä PCG-toteutusta. Tämän version tarkoitus on toimia suorituskykyvertailun lähtökohtana alkuperäisen ohjelman rinnalla, sekä olla helposti debugattavissa. Seuraavana vuorossa on Numba-kiihdytetty versio, joka käyttää Python-muunnoksen koodia mahdollisimman vähillä muutoksilla. Numba-kiihdytetty versio on vertailukelpoinen alkuperäisen MCERDin kanssa, sillä kummatkin käyttävät vain yhtä prosessorisäiettä. Kahden viimeisen version piti haarautuva: toinen rinnakkaistuisi prosessorille, toinen näytönohjaimelle. Tällöin ne edustaisivat eri alustojen laskentakapasiteetin täyttä hyödyntämistä. Ainoastaan rinnakkaistuminen prosessille toteutui käytännössä ja näytönohjaintoteutus jäi teoreettiselle tasolle.

4.2 Kehitysympäristö

Numban yhteensopivuus voi riippua käytetystä alustasta, joten toistettavuuden vuoksi esittelen käyttämäni kehitysympäristön. Valitsin kehityslaitteeksi Intel i7-4930K-pohjaisen tietokoneen, sillä se on vanhimpana laitteistani todennäköisesti rajoittava tekijä tuettujen omi-

naisuuksiensa osalta. Esittelen käytetyt testilaitteet tarkemmin luvussa 4.5. En kohdannut ohjelmia kehittäessä mitään ominaisuutta, joka olisi ollut tuettuna vain uudemmilla prosessoreilla.

Kehitystietokoneen käyttöjärjestelmänä on Microsoft Windows 10 Home, ja ohjelmointikielten versiot ovat CPython 3.8.10 sekä Microsoft Visual C/C++ v142. Päivitin Numbaa sitä mukaa kun uusia päivityksiä tuli saataville, kaikkiaan 0.52.0:sta 0.55.1:een. Käytetyt ohjelmat ovat ensisijaisesti 64-bittisiä. Suoritin C-kielisen MCERD-ohjelman ajamisen, debuggaamisen ja profiloinnin Microsoft Visual Studio 2019 -ohjelmalla. Python-kehitykseen käytin JetBrains PyCharm -ohjelman versioita 2020.2 ja 2021.2. Versiohallintaan minulla oli Git-työkalu, jonka tietovarasto on GitHub-verkkopalvelussa.

4.3 Toteutusprosessi

Varmistin valitsemani teknologian toimivuuden pienillä testiohjelmilla ennen varsinaista toteutusta. Kokeilin MCERDissä tarvittavia laskutoimituksia, kuten potenssiin korottamista, trigonometrisiä funktioita ja neliöjuuren laskemista sekä prosessorilla että näytönohjaimella. Testasin myös piin likiarvon laskemista Monte Carlo -simulaatiolla, joka on menetelmänä samanlainen kuin MCERDissä, mutta toteutukseltaan yksinkertaisempi. Testiohjelmien suorituskyky vastasi odotettua, ja sain samalla pystytettyä toimivan kehitysympäristön.

Aloitin toteuttamisen määrittelemällä ohjelmassa käytettävät vakiot ja tietueet, joista sekä arvojoukot että numeeriset vakiot olivat alun perin määritettyinä samalla tapaa *#define*-makroina (University of Jyväskylä 2020a, general.h). Paransin tilannetta ryhmittelemällä arvojoukot enumeraatioiksi omassa toteutuksessani. Toteutin perusversion tietueet Pythonin dataclass-luokilla (Python Software Foundation 2022a), sillä katsoin niiden vastaavan C:n struct-rakennetta parhaiten. Numbassa puolestaan ei ollut tukea tavallisille Python-luokille, joten päädyin käyttämään sen omaa Jitclass-toteutusta (Anaconda 2022c) JIT-versioissa. Jitclass-luokat toimivat vastaavalla tavalla kuin tavalliset luokat, mutta niiden attribuutit ja arvotyypit ovat staattiset, eivätkä dynaamiset kuten Pythonissa yleensä. Jitclass-luokkien dynaamisuus on todennäköisesti rajallinen siksi, että ne olisivat tehokkaita toteuttaa Numban tavukoodissa.

Ohjelmoinnin määrän minimoimiseksi toteutin vain ne koodipolut, joita simulaation ajaminen valitsemallani konfiguraatiolla vaatii. Tämä menettely karsi toteutettavia rivejä noin 500 ja vähensi mahdollisia asetuskombinaatioita merkittävästi. Halusin pitää erilaisten asetusten määrät minimissä, sillä kaikkien koodipolkujen toimivuuden varmistaminen olisi työlästä, varsinkin ilman automaattisia testejä. Aluksi myös tein parannuksia koodin luettavuuteen ja rakenteeseen mm. nimeämällä muuttujia uudelleen, täydentämällä aliohjelmien dokumentaatiota ja muotoilemalla rakenteita vastaamaan enemmän Pythonin ohjelmointityyliä, mutta lopulta totesin tämän liian työlääksi hyötyyn nähden ja päätin pitäytyä alkuperäiselle ohjelmalle uskollisemmassa muotoilussa.

Etenin suoritusjärjestyksessä, jotta pystyin koko ajan testaamaan ohjelmaani ajamalla sitä ja vertaamalla alkuperäiseen. Em. vakioiden ja tietueiden toteuttamiseen jälkeen jatkoin konfiguraatiotiedostojen jäsentämiseen. Samalla tein JIBAL-kirjastosta (Julin 2020) yksinkertaistetun version, joka tarjoaa vain simulaatiossa käytettävän datan isotooppien esiintyvyyksistä ja massasta. JIBALista olisi tarvinnut toteuttaa myös noin 1300-rivinen GSTO-osa, joka laskee simulaatiossa käytettävät jarruuntumisenergia-arvot eri alkuainepareille (Arstila ym. 2014). Päätin jättää tämän osuuden tekemättä, sillä sen lyhyt suoritusaika ei muuta tutkimuksen tuloksia merkittävästi, joten sen uudelleenkirjoittamiseen kuluva aika menisi pro gradun kannalta hukkaan. Koska GSTO on MCERDistä irrallinen, sen alkuperäistä versiota voi käyttää sellaisenaan. Haittapuolena käyttäjän pitää muistaa ajaa GSTO jokaiselle eri simulaatiokonfiguraatiolle uudelleen, koska koodissa ei ole automatiikkaa tämän suhteen.

Eri versiot etenivät sykleittäin, pääpiirteittäin suunnitelmien mukaan. Tein ensimmäistä versiota puhtaalla Pythonilla hakutaulujen generointiin asti, jonka jälkeen toteutin saman Numballa. Näin sain kattavan yleiskuvan projektista jo ennen ohjelmoinnin puoliväliä. Tuloksena oli suorituskyvyltään kohtuullisen hyvä versio, mutta ennen sitä kohtasin useita Numban rajoitteita, joista kerron luvussa 6.3. Tässä vaiheessa sain alustavan vastauksen tutkimuskysymykseen C:n muuttamisesta Pythoniin verrannollisella suorituskyvyllä, joka on: muunnos on todennäköisesti mahdollinen suorituskyvyn kannalta, mutta se voi vaatia koodin laatua alentavia ”workaroundeja”.

Tämän jälkeen sain tehtyä puhtaan Python -version simulaatio-osuudet valmiiksi ilman suurempia ongelmia. Suorituskyky oli huonompi, kuten oli odotettavissa. Numba-versiossa sen

sijaan ilmeni uusia ongelmia mm. syvästi sisäkkäisten oliorakenteiden ja olioiden luonnin suhteen. Myös rinnakkaistetuun versioon liittyi omat ongelmansa, mutta sain tehtyä sen loppuun. Osittain näytönohjaimella toimiva MCERD sen sijaan osoittautui toteutuskelvottomaksi, sillä siirtyminen eri suoritinlaitteilta oli liian hidasta. Tarkastelen näitäkin haasteita osiossa 6.3.

Toteutin Numba-versiot mahdollisimman yksinkertaisesti: kopioimalla ja muokkaamalla koodia aliohjelma kerrallaan. Tämä onnistui varsin hyvin, kunhan käytettävät ominaisuudet olivat tuettuina. Merkittäviä puuttuvia ominaisuuksia olivat mm. sisäkkäiset luokat, tiedostoon kirjoittaminen ja säiekohtaiset muuttujat rinnakkaistetussa koodissa.

Syntyneet moduulit jakautuivat kahteen ryhmään: puhtaalla Pythonilla tehtyihin perusversioihin sekä Numballa kiihdytettyihin versioihin, joiden nimeämisessä käytetään *_jit*-päätettä. Rinnakkaistettu versio toimii samoilla moduuleilla kuin yhden säikeen versiokin, joten tämän suhteen riitti tehdä kummallekin toteutuneelle Numba-variaatiolle vain oma pääohjelmansa, sekä muutama apufunktio rinnakkaistettulle koodille laskukuorman jakamiseen ja yhdistämiseen. Muutaman CUDA:lle muokatun aliohjelman perusteella GPGPU-toteutus olisi vaatinut erilliset *_cuda*-versiot moduuleista, sillä monet Numban CPU:lla toimivat ominaisuudet (Anaconda 2022g, 2022f) eivät ole tuettuina CUDA-alustalla (Anaconda 2022e).

Tein testiohjelman toteuttamisen avuksi muunnostyökaluja, joilla C-koodia saisi muunnettua Pythoniksi. Ensiksi yritin tehdä täysipainoista muunnosohjelmaa, joka olisi valmistuttuaan muuntanut säännöllisten lausekkeiden avulla C-koodin automaattisesti vastaaman syntaksiltaan Python-koodia. Sain sen toimimaan *#define*-makroilla määritellyille ja kommentteja sisältäville vakioille, mutta totesin ohjelman olevan liian kankea kehittää ja käyttää, joten hylkäsin sen. Myöhemmin tein Notepad++-tekstieditoriin suoraviivaisemman esikäsittelyskriptin, joka hoitaa vain helposti automatisoitavat tekstimuunnokset: operaattorien ja muuttujien uudelleennimeämisen (esim. puolipisteen poisto rivin lopusta ja nuolioperaattorin muunto pisteeksi), sekä moduulien nimien lisäämisen Python-funktioihin, kuten siniin (math.sin). Monimutkaisempia muunnoksia oli lopulta niin vähän, että niiden tekeminen käsin oli nopeampaa kuin automaation kautta. Tämä lähestymistapa osoittautui toimivaksi, sillä käytin samaa editoria muuntamisen apuna muutenkin. Näin sain paketoitua sen ”korvaa kaikki” -operaatiot yhteen skriptiin, eikä muunnettavia funktioita tarvinnut kierrättää väliaikaistie-

doston ja toisen ohjelman kautta.

4.4 Suorituskyvyn mittaaminen

Ohjelmien toteuttamisen jälkeen seuraa tutkielman kannalta tärkeä vaihe, suorituskyvyn mittaaminen. Mittaan eri toteutusten suorituskykyä kahdella simulaatiokonfiguraatiolla käyttäen kolmea testilaitetta. Valitsin vertailun kohteiksi konfiguraatiot, jotka vastaavat tyypillisiä simulaatioita.

Suorituskykymittausten pääsuureena on käyttäjän kannalta olennaisin tekijä: kuinka kauan simulaation suorittamisessa kestää eri tehoisilla prosessoreilla. Suoritusaika määrää sen, kuinka kauan tuloksia joutuu odottamaan. Mitä lyhyempi tämä aika on, sitä vähemmän se häiritsee työskentelyä.

Myös muistinkäyttöön on syytä kiinnittää huomiota, koska sillä voi olla merkittävä vaikutus suoritusaikaan. Jos käynnissä olevien ohjelmien vaatima muistimäärä ylittää saatavilla olevan kapasiteetin, suorituskyky kärsii kun käyttöjärjestelmä joutuu vapauttamaan muistia karsimalla vanhoja ohjelmia tai siirtämällä osan muistista levylle. Tällä välin suoritettavat ohjelmat saattavat pysähtyä odottamaan pääsyä keskusmuistiin. Tarkastelun kohteena on muistin käyttö erityisesti simulaatiovaiheessa, koska tällöin muistin riittävyydellä on suurin merkitys vaiheen pitkän kokonaiskeston ja jatkuvan muistikuorman vuoksi. Muut vaiheet vievät simulointiin verrattuna paljon vähemmän aikaa, joten niiden pitkittymisellä ei pitäisi olla yhtä suurta vaikutusta.

Mittausten aikaisten taustaohjelmien määrä minimoidaan, jotta niillä olisi mahdollisimman pieni vääristävä vaikutus tuloksiin.

Numban JIT-optimoinnin vaikutusten mittaamiseksi jokainen Numba-ajo suoritetaan kolmesti peräkkäin. Ennen jokaisen konfiguraation ensimmäistä ajokertaa Pythonin väliaikaistiedostokansio *__pycache__* (Warsaw 2009) tyhjennetään moduulien käännetyistä tavukoodeista, jotta tilanne vastaisi tyhjästä aloittamista. Toinen kierros vastaa suunnilleen tilannetta, jossa jotain parametria olisi muutettu ensimmäisen ajokerran jälkeen ja sitten ajettu uudelleen. Kolmannen kierroksen päätarkoitus on varmentaa, ettei ohjelman suoritusaika enää

muutu toisesta ajokerrasta. Lisäksi kolmas kierros tasaa satunnaisen vaihtelun vaikutusta siltä varalta, että testilaitteen taustaohjelmat aiheuttaisivat yllättävää kuormitusta.

Simulaatiokonfiguraatioiden ainoat muuttujat ovat tarkasteltava alkuaine sekä sille GSTO-ohjelmalla generoidut jarruuntumisarvot. Simuloitavana ovat kaksi alkuainetta, atomimassaltaan ja laskennan kannalta keskiraskas happi (O) sekä raskaampi titaani (Ti). Minimoin muutettavien asetusten määrän, jotta alkuaineiden tulokset olisivat mahdollisimman hyvin vertailtavissa, ilman että mittauksiin kuluu tutkielman viitetyömäärään nähden liikaa aikaa.

Muuten käytetty konfiguraatio noudattaa pitkälti Potku-ohjelman oletusasetuksia (Arstila ym. 2014). Kohde koostuu sadan nanometrin paksuisesta titaaninitridi (TiN) -kerroksesta, jonka alla on 300 nm:n piisubstraatti (Si). Beami on kupari-63-isotooppi 13 MeV:in energialla, mikä etenee korkeahkon painonsa ja täten liikemääränsä vuoksi syvälle näytteeseen. Tämän seurauksena laskettavia törmäystapahtumia on melko paljon. Ilmaisin on tyypiltään TOF eli lentoaikailmaisin. Simuloitavia ioneja on vakiona 100 000 esisimulaatiossa ja 1 000 000 pääsimulaatiossa. Valmiit testiohjelmat ja niiden konfiguraatiot ovat saatavilla GitHub-repositoriossani [1]. Lisäksi mittauksessa käytetyt skriptit löytyvät osoitteesta [2].

4.5 Testilaitteisto

Mittauksissa käytettävät testilaitteet kattavat laajan skaalan tietokoneita kahdella, kuudella ja kahdellatoista fyysisellä ytimellään. Ydinmäärän lisäksi laitteiden ydinkohtainen suorituskyky vaihtelee: kaksi vanhinta laitetta ovat yksittäisiltä ytimiltään suunnilleen yhtä nopeat, kun taas uusin on niitä huomattavasti nopeampi. Testilaitteet ovat esiteltynä taulukossa 4.

Laskennan kannalta laitteiden muut osat ovat varsin samanlaiset, sillä kaikilla on käytössä Windows 10 -käyttöjärjestelmä ja SSD-massamuisti. Prosessorin lisäksi eroavaisuuksia löytyy keskusmuistista, sillä uusimmalla testilaitteella on muita nopeammin toimivaa DDR4-muistia, mikä vaikuttaa erityisesti muistikaistan määrään ja täten mahdollisesti siirtonopeuteen prosessorille. Kannettavan tietokoneen osalta muistikaistan määrää heikentää sen kaksi muistikanavaa, sillä muissa laitteissa kanavia on neljä. Näytönohjaimella ei puolestaan pitäi-

1. `https://github.com/tpitkanen/numba_mcerd`
2. `https://github.com/tpitkanen/numba_mcerd_benchmark`

Prosessoriytimet	8 × Intel i7-12700K @ 5.0 GHz (P-ytimet)
Prosessoriytimet	4 × Intel i7-12700K @ 3.9 GHz (E-ytimet)
Näytönohjain	Nvidia RTX 3070 @ 1920 Mhz, 8 Gt GDDR6 VRAM
Keskusmuisti	32 Gt DDR4 @ 3600 MT/s
Massamuisti	2 Tt Samsung 970 EVO Plus NVMe SSD
Käyttöjärjestelmä	Windows 10 Education, 64-bit, versio 20H2 (19042.1526)
Prosessoriytimet	6 × Intel i7-4930K @ 4.2 GHz
Näytönohjain	Nvidia GTX 980 @ 1279 Mhz, 4 Gt GDDR5 VRAM
Keskusmuisti	16 Gt DDR3 @ 1600 MT/s
Massamuisti	1 Tt Samsung 860 EVO SATA SSD
Käyttöjärjestelmä	Windows 10 Home, 64-bit, versio 20H2 (19042.1526)
Prosessoriytimet	2 × Intel i5-6200U @ 2.7 GHz
Näytönohjain	Intel HD Graphics 520 @ ? MHz
Keskusmuisti	8 Gt DDR3L @ 1600 MT/s
Massamuisti	0.5 Tt SATA SSD
Käyttöjärjestelmä	Windows 10 Home, 64-bit, versio 20H2 (19042.1526)

Taulukko 4. Testilaitteistojen kokoonpanot.

si olla vaikutusta prosessorilaskentaan, mutta GPGPU-laskentaan sillä olisi.

Laitteista kaikki paitsi i7-12700K:n E-ytimet tukevat Intelin Hyperthreading-ominaisuutta, joka tuplaa loogisten suoritinydinten lukumäärän sisällyttämällä jokaista fyysistä kohden kaksi rekisterikokonaisuutta, mikä mahdollistaa suuremman laskentakapasiteetin rekisterejä vuorottelemalla. Toinen virtuaaliytimistä voi siirtyä käyttämään prosessoria, kun toinen odottaa käskyjä.

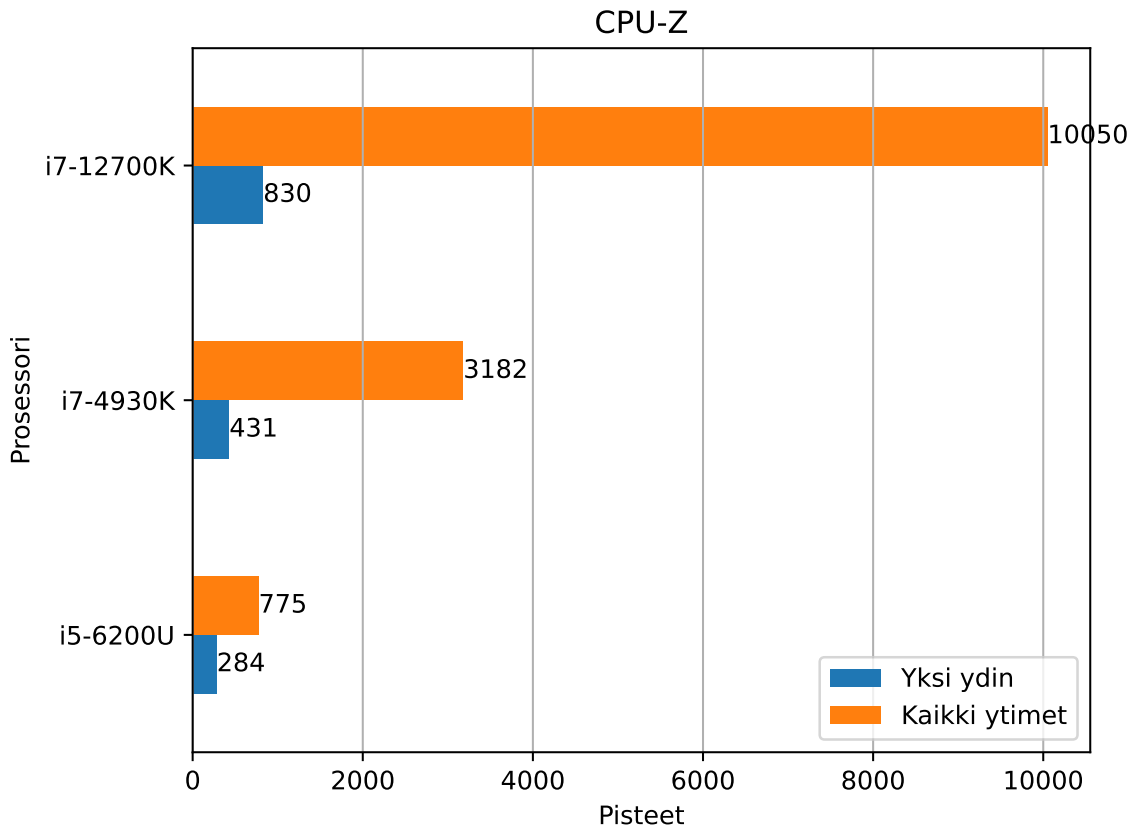
Testilaitteista hitain on kannettava tietokone, jossa on prosessorina Intel i5-6200U. Prosessori on julkaistu vuoden 2015 loppupuolelta, eikä se ollut alun perinkään kovin tehokas vain kahden fyysisen ytimen ja matalan maksimitehonkulutuksen vuoksi. Toisin kuin muita testilaitteita, ajan tätä vakioasetuksilla. Laite edustaa vanhentunutta ja kohtuullisen hidasta kannettavaa tietokonetta tai taulutietokonetta.

Keskitason suorituskykyä edustaa Intel i7-4930K:lla prosessorilla varustettu pöytäkone. Prosessori on julkaistu vuoden 2013 loppupuolella, joten se on yli kahdeksanvuotiaana varsin vanha. Aikakaudelleen poikkeuksellisesti siinä on kuusi fyysistä ydintä tavallisen neljän sijaan. Olen ylikellottanut sen 4.2 Ghz:iin vakion 3.6 GHz:n sijaan. Laite vastaa vanhentunutta peli- tai työasemakonetta.

Testilaitteista nopein on varustettu Intel i7-12700K:lla. Prosessori on vuoden 2021 loppupuolelta, ja se hyvin nopea yhden ja monen säikeen suorituskyvyssä. Kyseinen prosessori noudattaa hybridiarkkitehtuuria, jossa tavalliset P(erformance)-ytimet ovat nopeita ja suurikokoisia, kun taas E(fficient)-ytimet ovat hitaampia eivätkä tue Hyperthreading-ominaisuutta, mutta niitä mahtuu neljä yhden P-ytimen kokoiseen tilaan, ja ne ovat energiatehokkaampia. Olen ylikellottanut sen kahdeksan P-ytimen kellotaajuuden 4.7 Ghz:stä 5.0 Ghz:iin ja E-ytimet 3.6 GHz:stä 3.9 Ghz:iin.

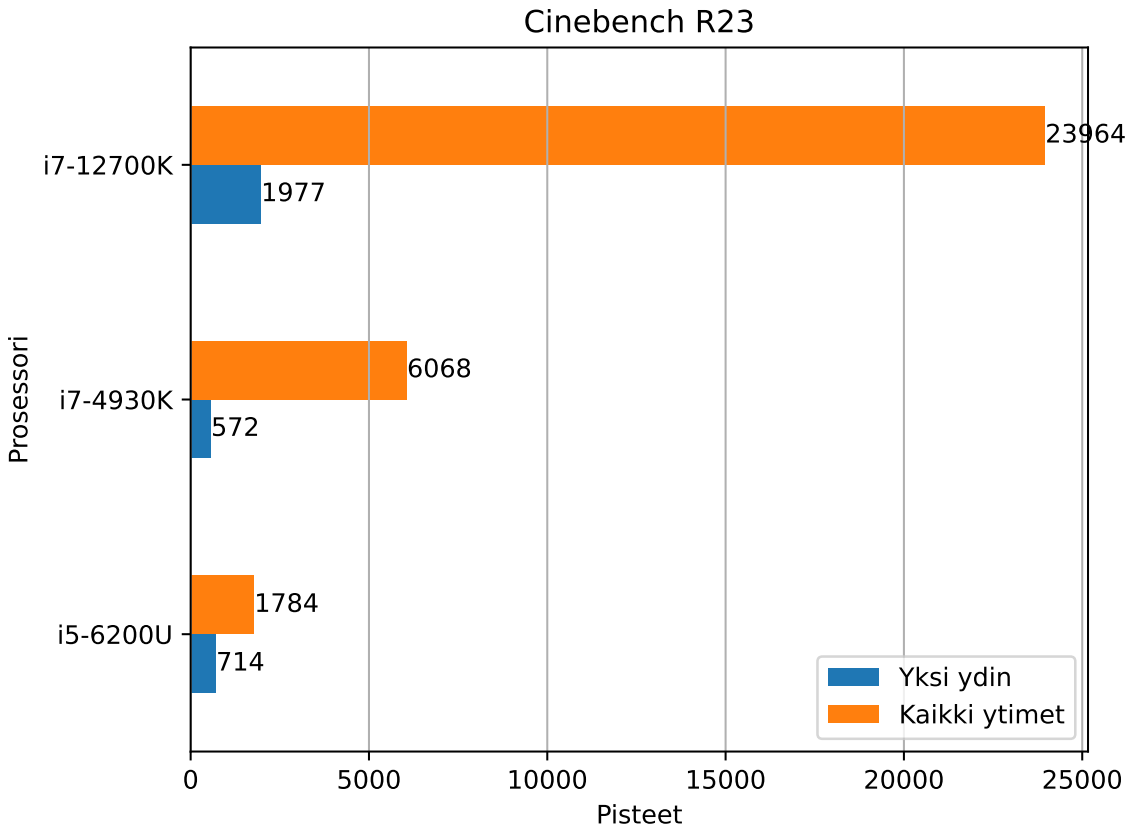
Laitteiden suorituskyky on mitattu vertailun vuoksi CPU-Z- (CPUID 2022) ja CineBench-testiohjelmilla (Maxon 2022), joiden tulokset löytyvät taulukoista 4 ja 5. Mittaustulokset tukevat tulkintaa siitä, että i7-12700K on moninkertaisesti muita laitteita nopeampi yhden ytimen sekä erityisesti kaikkien ytimien suorituskyvyn osalta. i7-4930K on puolestaan monta kertaa i5-6200U-prosessoria nopeampi kaikkien ytimien laskennassa, sillä siinä on kolminkertainen määrä laskentaytimiä. i7-4930K:n yhden säikeen tulos on i5-6200U:n tulosta

heikompi CineBench R23 -testissä, mikä johtuu todennäköisesti siitä, että testi hyödyntää AVX2-käskykantaa versiosta R20 alkaen (Ung 2022). i7-4930K tukee vain lyhyempää ja hitaampaa AVX-käskykantaa (Intel 2022c), kun muut prosessorit tukevat myös pidempää AVX2-kantaa (Intel 2022a, 2022b).



Kuvio 4. CPU-Z-suorituskykytestin tulokset.

Pöytäkoneiden tehorajoitukset ja jäähdytys ovat riittävällä tasolla, jotta ne eivät rajoita suorituskykyä pidemmissäkään testeissä, vaan prosessorien kellotaajuus pysyy koko ajan maksimissa. Kannettavan tietokoneen vähävirtaisen U-sarjan prosessorin tehoraja taas on niin matala, että prosessorin kellotaajuus vaihtelee jonkin verran laskukuorman mukaan, erityisesti monisäikeisessä laskennassa. Tämä on mobiiliprosessoreille tyypillistä, joten ominaisuuden voi nähdä muille kannettaville tietokoneille yleistämisen kannalta myönteisenä piirteenä.



Kuvio 5. CineBench R23 -suorituskykytestin tulokset.

4.6 Oikeellisuuden verifiointi

Suorituskyvyn mittaamisen lisäksi on olennaista myös varmistua siitä, että uudet ohjelmat tuottavat samanlaisia tuloksia kuin alkuperäinenkin. Verifioin eri toteutusten tuottaman datan oikeellisuuden vertaamalla keskeisimpiä simulaatioiden suureita alkuperäiseen toteutukseen. Koska C-ohjelmassa on käytössä erilainen satunnaislukugeneraattori kuin muissa versioissa, tulokset eivät ole täysin identtisiä, joten on syytä tarkastella lukuja tilastollisesti. Keskeisimmät suureet ovat ionien lopputilojen lukumäärät, ERD-tapahtumadata sekä sisä- ja ulkosilmukan pääionikohtaiset kierroslukumäärät.

MCERD ei alun perin tuottanut dataa silmukoiden kierrosmääristä, vaan lisäsin laskurin erillistä mittauskertaa varten. Näin kierroslukujen mittaaminen ja tallentaminen ei vaikuta mitattuihin suoritusaikoihin.

5 Tulokset

Mittasin alkuperäisen MCERD-ohjelman sekä toteuttamieni Python-muunnosten suorituskykyä eri alkuaineilla, laitteilla ja säiemäärillä. Tarkasteltavat toteutusmuodot ovat yksisäikeinen puhdas Python, yksisäikeinen Numba ja monisäikeinen Numba. Vertailukohtana toimii alkuperäinen C-ohjelma, joka on yksisäikeinen. Mitatut suureet ovat suoritusaika ja keskusmuistinkäyttö simulaatiovaiheiden aikana. Lisäksi tarkastelun kohteena on tuotettu mittausdata. Analysoin saatuja tuloksia luvussa 6.

Havaitsin mittausten jälkeen, että toteuttamissani testiohjelmissa on ohjelmointivirhe, joka aiheuttaa alkuperäistä versiota noin 22.8–24.6 prosenttia enemmän simulointia. Simulaatiotulokset vaikuttavat muuten vastaavan alkuperäistä ohjelmaa, joten uusien toteutusten suoritusajat ovat todennäköisesti tämän verran pidempiä kuin niiden pitäisi olla. Käsittelen asiaa tarkemmin luvussa 5.4 ja 6.2.

5.1 Suoritusaika

Mitatut suoritusajat hapelle ovat taulukossa 5 sekä titaanille taulukossa 6. Vertailu suoritusaikojen suhteellisista kestoista löytyy taulukosta 7. Tekstissä esitettävät ajat ovat muotoa s, m:s tai h:m:s.

Alkuperäisen C-ohjelman suoritusajat vaihtelevat välillä 10:25–26:03 hapelle ja 23:53–58:08 titaanille. Toinen ajokerta ei juurikaan vaikuta suoritusaikaan, paitsi Intel i5-6200U-prosessorilla hapen suoritusaika on noin kuusi prosenttia lyhyempi. Esi- ja pääsimulaatioiden osuus suoritusajasta on vähintään 99.6 prosenttia, jolloin muihin vaiheisiin kuluu korkeintaan 0.4 prosenttia ajasta. C-versio on yksisäikeisistä toteutuksista selkeästi nopein kaikilla testatuilla laitteilla ja alkuaineilla.

Kiihdyttämättömän Python-toteutuksen ainoa täysimittainen suoritusaika on Intel i7-12700K-prosessorilla 905:18 (15:05:18) hapelle. Simulaatioiden osuus on taulukon arvoa tarkemmin ilmoitettuna 99.983 prosenttia. Muut vaiheet vievät 0.017 prosenttia eli 9.5 sekuntia. Puhdas Python on muihin toteutuksiin verrattuna hyvin hidas, suoritusajaltaan esimerkiksi C-

Suoritin	Toteutus	Ajokerta	Alustus	Esisim.	Analyysi	Pääsim.	Loppu	Summa (m:s)
i7-12700K	Python	I	0.0	6.4	0.0	93.6	0.0	905:18
i7-12700K	C	I	0.3	7.4	0.0	92.3	0.0	10:26
i7-12700K	C	II	0.3	7.4	0.0	92.3	0.0	10:25
i7-12700K	JIT	I	0.5	7.1	0.7	91.4	0.4	12:51
i7-12700K	JIT	II	0.2	6.0	0.5	93.0	0.3	12:33
i7-12700K	JIT	III	0.2	5.9	0.5	93.1	0.3	12:49
i7-12700K	JIT (20-T)	I	1.3	9.7	1.6	86.6	0.8	4:58
i7-12700K	JIT (20-T)	II	0.7	6.8	1.4	90.2	0.9	4:32
i7-12700K	JIT (20-T)	III	0.7	6.9	1.5	90.0	0.9	4:29
i7-4930K	C	I	0.3	7.5	0.0	92.1	0.0	19:46
i7-4930K	C	II	0.3	7.5	0.0	92.2	0.0	19:38
i7-4930K	JIT	I	0.7	8.5	0.9	89.4	0.5	29:33
i7-4930K	JIT	II	0.2	5.9	0.9	92.5	0.5	28:15
i7-4930K	JIT	III	0.2	5.9	0.9	92.5	0.5	28:30
i7-4930K	JIT (12-T)	I	2.7	13.4	3.9	77.5	2.3	6:44
i7-4930K	JIT (12-T)	II	1.4	7.1	4.0	85.1	2.5	6:00
i7-4930K	JIT (12-T)	III	1.3	7.1	3.9	85.2	2.4	6:07
i5-6200U	C	I	0.3	7.6	0.0	92.1	0.0	26:03
i5-6200U	C	II	0.3	7.5	0.0	92.2	0.0	24:32
i5-6200U	JIT	I	1.1	7.7	0.7	90.1	0.4	36:04
i5-6200U	JIT	II	0.2	5.9	0.6	92.9	0.3	34:18
i5-6200U	JIT	III	0.2	6.0	0.7	92.8	0.3	33:58
i5-6200U	JIT (4-T)	I	1.4	10.6	1.6	85.7	0.8	16:12
i5-6200U	JIT (4-T)	II	0.5	6.7	1.5	90.5	0.8	15:23
i5-6200U	JIT (4-T)	III	0.6	6.9	1.6	90.2	0.8	15:05

Taulukko 5. Suoritusaikojen jakauma hapelle. Vaiheet ovat prosentteina kokonaisajasta. (n-T = n-säikeinen.)

Suoritin	Toteutus	Ajokerta	Alustus	Esisim.	Analyysi	Pääsim.	Loppu	Summa (m:s)
i7-12700K	C	I	0.1	6.6	0.0	93.3	0.0	24:19
i7-12700K	C	II	0.1	6.6	0.0	93.3	0.0	23:53
i7-12700K	JIT	I	0.2	5.8	0.3	93.6	0.1	30:01
i7-12700K	JIT	II	0.1	5.5	0.2	94.1	0.1	29:24
i7-12700K	JIT	III	0.1	5.3	0.2	94.3	0.1	29:44
i7-12700K	JIT (20-T)	I	0.7	7.4	0.8	90.9	0.4	10:07
i7-12700K	JIT (20-T)	II	0.3	5.7	0.7	93.0	0.4	10:11
i7-12700K	JIT (20-T)	III	0.3	5.8	0.7	92.8	0.4	9:49
i7-4930K	C	I	0.2	6.7	0.0	93.1	0.0	43:55
i7-4930K	C	II	0.1	6.6	0.0	93.2	0.0	44:10
i7-4930K	JIT	I	0.2	6.0	0.4	93.2	0.2	68:28
i7-4930K	JIT	II	0.1	5.4	0.4	94.0	0.2	65:51
i7-4930K	JIT	III	0.1	5.2	0.4	94.1	0.2	65:57
i7-4930K	JIT (12-T)	I	1.2	8.9	1.9	87.0	1.0	14:13
i7-4930K	JIT (12-T)	II	0.5	5.8	1.8	90.8	1.0	13:16
i7-4930K	JIT (12-T)	III	1.3	5.8	1.8	90.1	1.0	13:10
i5-6200U	C	I	0.1	6.8	0.0	93.0	0.0	58:08
i5-6200U	C	II	0.1	6.7	0.0	93.2	0.0	57:52
i5-6200U	JIT	I	0.2	6.1	0.3	93.2	0.1	82:46
i5-6200U	JIT	II	0.1	5.3	0.3	94.2	0.1	80:14
i5-6200U	JIT	III	0.1	5.3	0.3	94.2	0.1	80:36
i5-6200U	JIT (4-T)	I	0.6	7.3	0.7	91.1	0.3	37:06
i5-6200U	JIT (4-T)	II	0.2	5.7	0.6	93.2	0.3	36:05
i5-6200U	JIT (4-T)	III	0.4	5.7	0.6	93.0	0.3	35:21

Taulukko 6. Suoritusaikojen jakauma titaanille. Vaiheet ovat prosentteina kokonaisajasta. (n-T = n-säikeinen.)

Suoritin	Toteutus	Ajokerta	Kesto (O)	Kesto (Ti)
i7-12700K	Python	I	86.77	-
i7-12700K	C	I	**1.00**	**1.00**
i7-12700K	C	II	1.00	0.98
i7-12700K	JIT	I	1.23	1.23
i7-12700K	JIT	II	1.20	1.21
i7-12700K	JIT	III	1.23	1.22
i7-12700K	JIT (20-T)	I	0.48	0.42
i7-12700K	JIT (20-T)	II	0.43	0.42
i7-12700K	JIT (20-T)	III	0.43	0.40
i7-4930K	C	I	**1.00**	**1.00**
i7-4930K	C	II	0.99	1.01
i7-4930K	JIT	I	1.49	1.56
i7-4930K	JIT	II	1.43	1.50
i7-4930K	JIT	III	1.44	1.50
i7-4930K	JIT (12-T)	I	0.34	0.32
i7-4930K	JIT (12-T)	II	0.30	0.30
i7-4930K	JIT (12-T)	III	0.31	0.30
i5-6200U	C	I	**1.00**	**1.00**
i5-6200U	C	II	0.94	1.00
i5-6200U	JIT	I	1.38	1.42
i5-6200U	JIT	II	1.32	1.38
i5-6200U	JIT	III	1.30	1.39
i5-6200U	JIT (4-T)	I	0.62	0.64
i5-6200U	JIT (4-T)	II	0.59	0.62
i5-6200U	JIT (4-T)	III	0.58	0.61

Taulukko 7. Suoritusaikojen suhteelliset kestot hapelle ja titaanille. Vertailuarvo lihavoitu. (n-T = n-säikeinen.)

ohjelman nähden noin 87-kertainen.

Yksisäikeisen Numba-version suoritusajat vaihtelevat välillä 12:33–36:04 hapelle ja 29:24–82:46 titaanille. Toinen ajokerta on kaikilla mitatuilla yhdistelmillä joitakin prosentteja ensimmäistä nopeampi. Myös kolmas ajokerta on ensimmäistä nopeampi, mutta hapen ja i5-6200U-prosessorin yhdistelmää lukuun ottamatta toista kertaa hitaampi. Simulaatioiden osuus on vähintään 97.8 prosenttia, jolloin muiden vaiheiden kestoksi jää korkeintaan 2.2 prosenttia. Yksisäikeisen Numba-version ajaminen kestää alkuperäiseen C-ohjelmaan verrattuna noin 1.20–1.56-kertaisen ajan. Numba-optimointi lyhentää puhtaan Pythonin suoritusajan noin 1.4 prosenttiin eli kiihdyttää suorituskyvyn noin 71-kertaiseksi i7-12700K-prosessorilla.

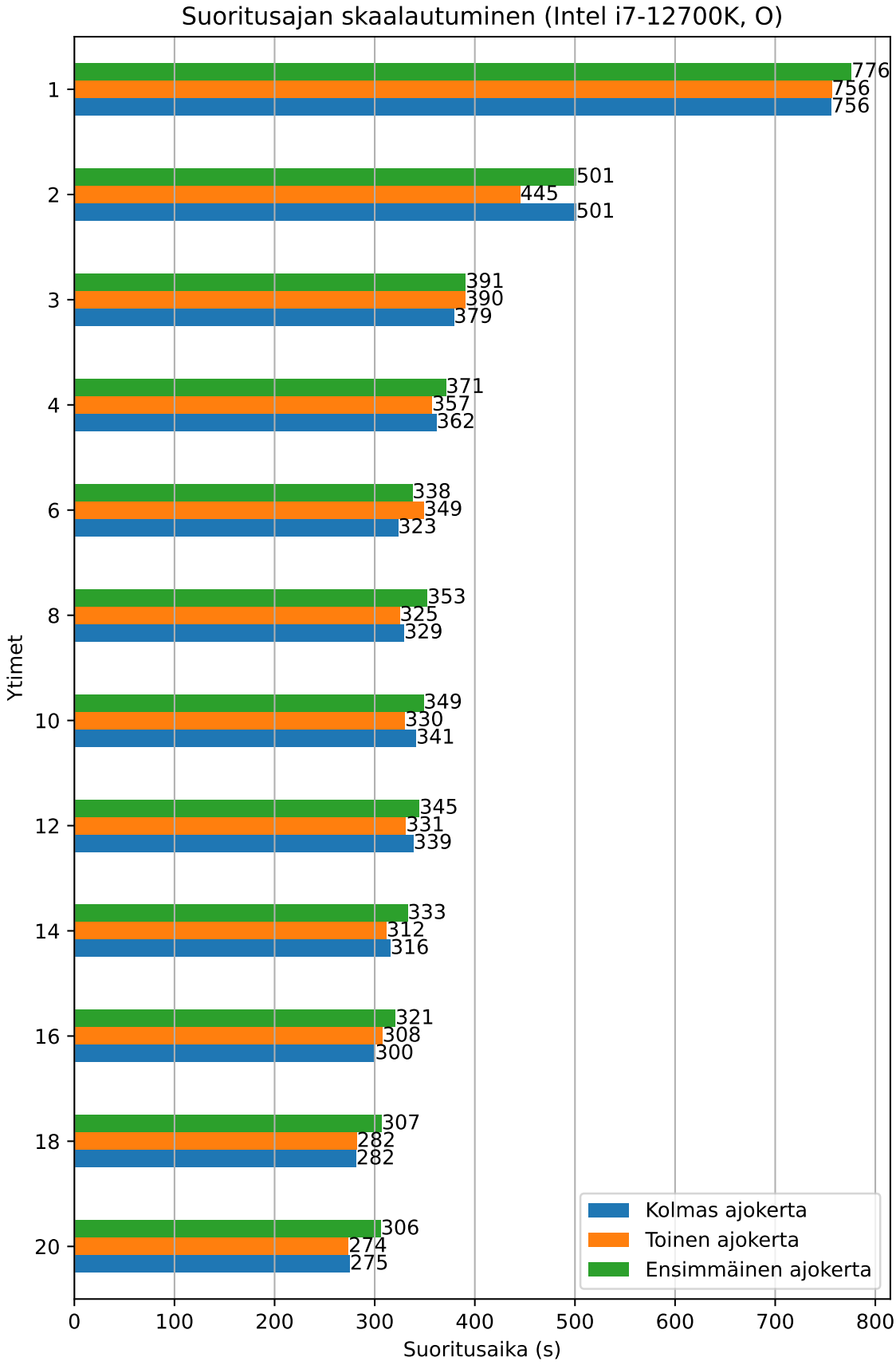
Rinnakkaistetun Numba-ohjelman suoritusajat vaihtelevat välillä 4:29–16:12 hapelle ja 9:49–37:06 titaanille. Toinen ajokerta on ensimmäistä merkittävästi nopeampi. Kolmas ajokerta on toista nopeampi kaikilla paitsi hapen ja Intel i7-4930K-prosessorin yhdistelmällä. Simulaatioiden osuus on vähintään 90.9 prosenttia ensimmäisellä suorituskerralla ja 92.3 prosenttia myöhemmillä kerroilla. Tällöin muiden vaiheiden osuus on korkeintaan 9.1 prosenttia ensimmäisellä ja 7.7 prosenttia myöhemmillä suorituskerroilla. Rinnakkaistetun Numba-version suoritusaika on C-ohjelmaan verrattuna huomattavasti lyhyempi, noin 0.30–0.64-kertainen. Rinnakkaistettuna Numba kiihdyttää puhtaan Pythonin suorituskyvyn noin 180–200-kertaiseksi i7-12700K-prosessorilla.

5.2 Suoritusajan skaalautuminen

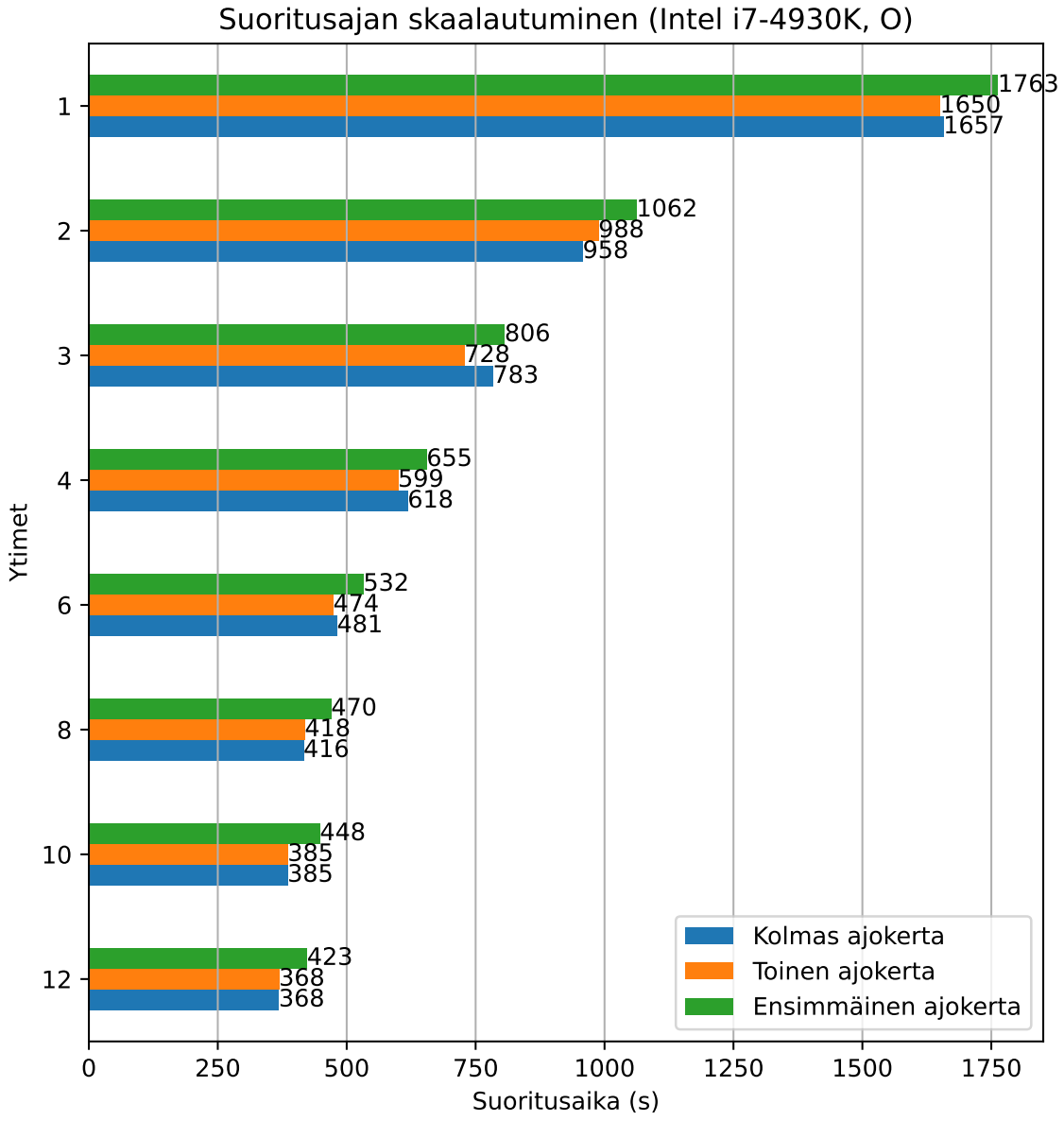
Tulokset eri säiemäärien suoritusaikojen skaalautumisesta hapelle löytyvät seuraavista kuvioista: i7-12700K: 6, i7-4930K: 7 ja i5-6200U: 8. Ilmoitetut ajat ovat teknisistä syistä sekunteina. Lisäksi taulukkoon 8 on koottuna hapen suoritusajan suhteellinen skaalautuminen.

Mittaustulokset eroavat hieman aiempana ilmoitetuista arvoista yhden ja maksimisäikeiden osalta, sillä ne ovat mitattu uudelleen käsin konfiguroiduilla säiemäärillä. Toisin sanoen käytössä on rinnakkaistettu toteutus yhteen säikeeseen rajoitettuna rinnakkaistomattoman version sijaan. Maksimisäikeinen ajo on puolestaan määritelty käsin käyttämään kaikkia prosessorien loogisia ytimiä, eikä automaattisesti tunnistettua lukumäärää. Normaalisti automaat-

titunnistus ottaa käyttöön kaikki loogiset ytimet, joten käsin määritellyn maksimisäiemäärän tehtävä on varmentaa automatiikan toimivuus ja mahdollinen vaikutus suoritusaikaan. Automatiikan toimivuuden varmentaminen on olennaista erityisesti i7-12700K:n hybridiarkkitehtuurin kohdalla.



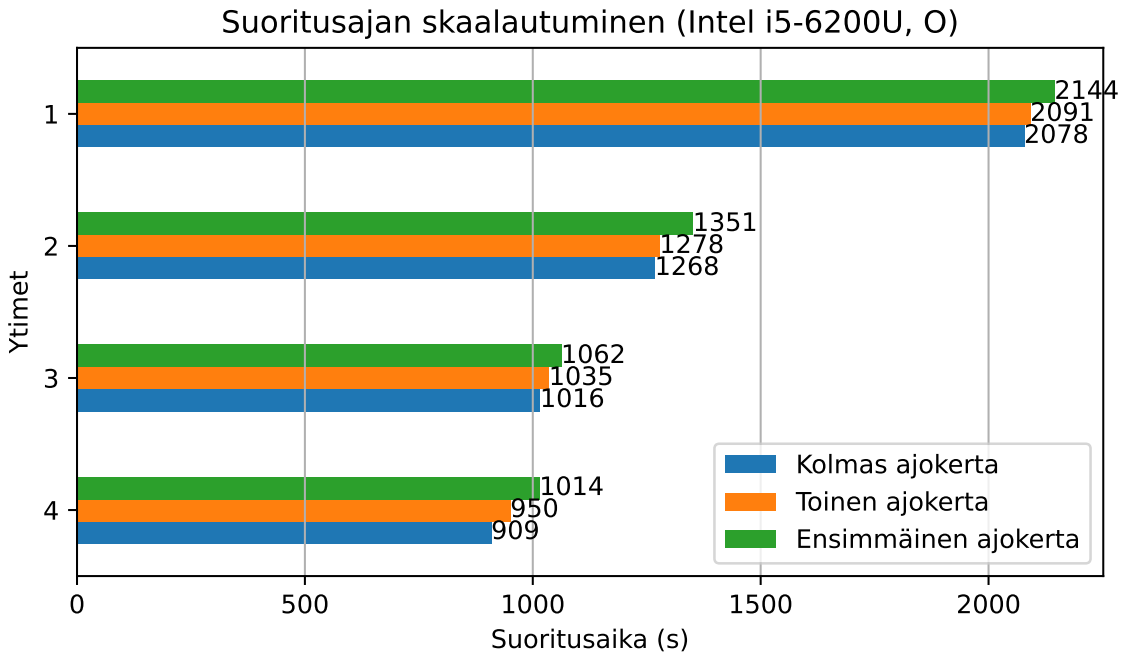
Kuvio 6. Rinnakkaistetun Numba-toteutuksen suoritusajan skaalautuminen hapelle eri ydinmäärillä i7-12700K-prosessorilla.



Kuvio 7. Rinnakkaistetun Numba-toteutuksen suoritusajan skaalautuminen hapelle eri ydinmäärillä i7-4930K-prosessorilla.

Suoritin	Säikeitä	I ajokerta	II ajokerta	III ajokerta
i7-12700K	1	**1.00**	0.97	0.97
i7-12700K	2	0.65	0.57	0.65
i7-12700K	3	0.50	0.50	0.49
i7-12700K	4	0.48	0.46	0.47
i7-12700K	6	0.43	0.45	0.42
i7-12700K	8	0.45	0.42	0.42
i7-12700K	10	0.45	0.43	0.44
i7-12700K	12	0.44	0.43	0.44
i7-12700K	14	0.42	0.40	0.41
i7-12700K	16	0.41	0.40	0.39
i7-12700K	18	0.40	0.36	0.36
i7-12700K	20	0.39	0.35	0.35
i7-4930K	1	**1.00**	0.94	1.00
i7-4930K	2	0.60	0.56	0.54
i7-4930K	3	0.46	0.41	0.44
i7-4930K	4	0.37	0.34	0.35
i7-4930K	6	0.30	0.27	0.27
i7-4930K	8	0.27	0.24	0.24
i7-4930K	10	0.25	0.22	0.22
i7-4930K	12	0.24	0.21	0.21
i5-6200U	1	**1.00**	0.98	0.97
i5-6200U	2	0.63	0.60	0.59
i5-6200U	3	0.50	0.48	0.47
i5-6200U	4	0.47	0.44	0.42

Taulukko 8. Rinnakkaistetun Numba-toteutuksen suoritusaikojen suhteellinen skaalautuminen hapelle. Vertailuarvo lihavoitu.



Kuvio 8. Rinnakkaistetun Numba-toteutuksen suoritusajan skaalautuminen hapelle eri ydinmäärillä i5-6500U-prosessorilla.

i7-12700K:n kohdalla merkittävää nopeutusta tarjoavat kahden ja kolmen säikeen konfiguraatiot, jonka jälkeen nopeutuminen hiipuu. Suoritusaika laskee lopulta kaikilla säikeillä 0.39–0.35-kertaiseksi yksisäikeiseen toteutukseen verrattuna.

i7-4930K hyötyy säikeistä kaikilla määrillä, joskaan Hyperthreading-ominaisuudella saadut lisäsäikeet eivät nopeuta suoritusta yhtä paljon kuin fyysiset säikeet yksinään. Kahdentoista säikeen suoritusaika on 0.24–0.21-kertainen yhteen säikeeseen nähden.

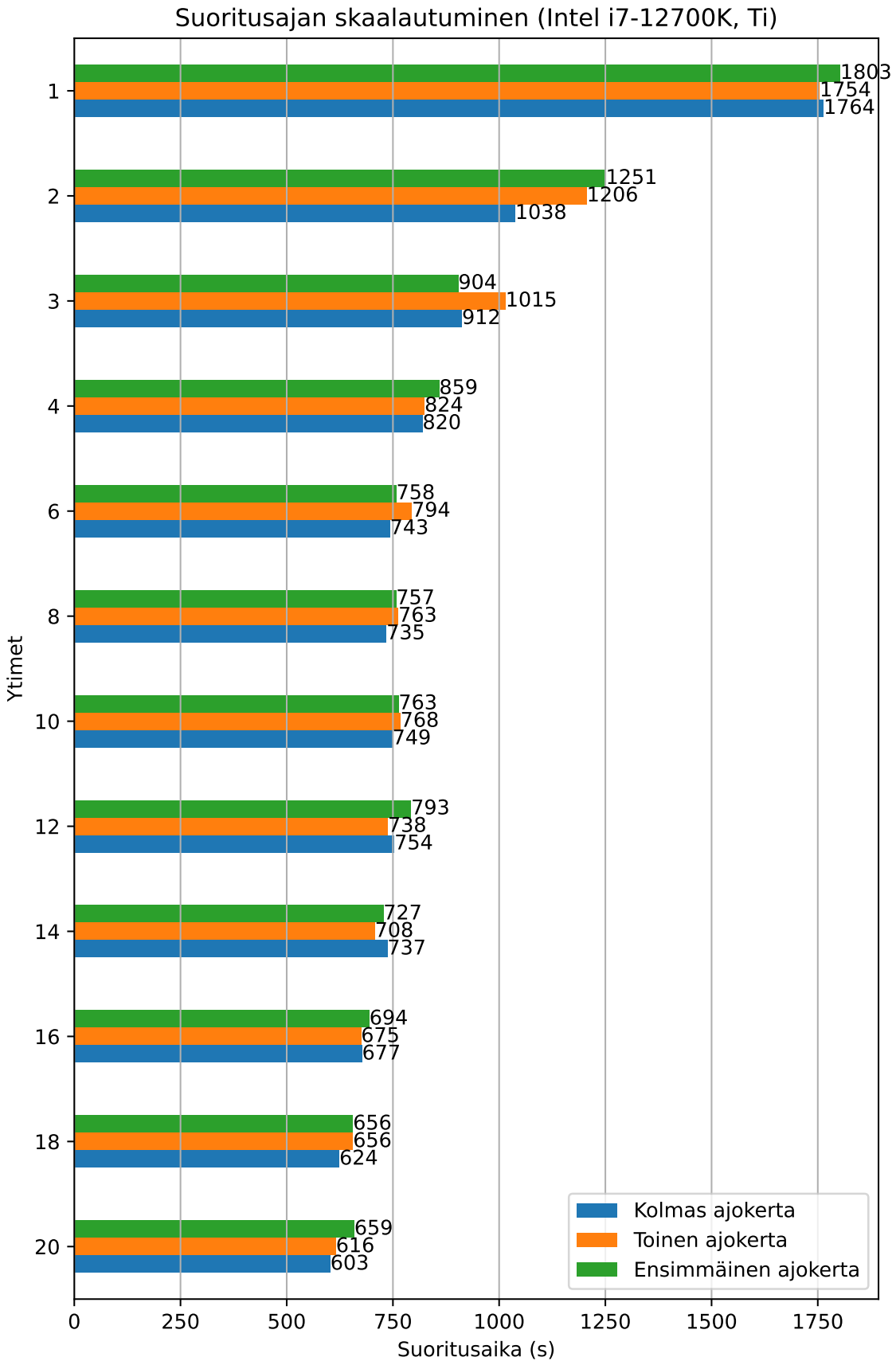
i7-4930K:n mittauksessa ilmeni poikkeama kahden säikeen konfiguraatiossa, jolla jokainen ajokerta hidasti ohjelmaa merkittävästi. Nämä poikkeavat tulokset löytyvät taulukosta 9. Uudelleen mitattuna vastaavaa poikkeamaa ei ilmennyt, joten kuviossa 7 ja taulukossa 8 käytetään uusia mittaustuloksia.

Suoritin	Säikeitä	Ajokerta	Suoritusaika (s)
i7-4930K	2	I	987
i7-4930K	2	II	1072
i7-4930K	2	III	1153

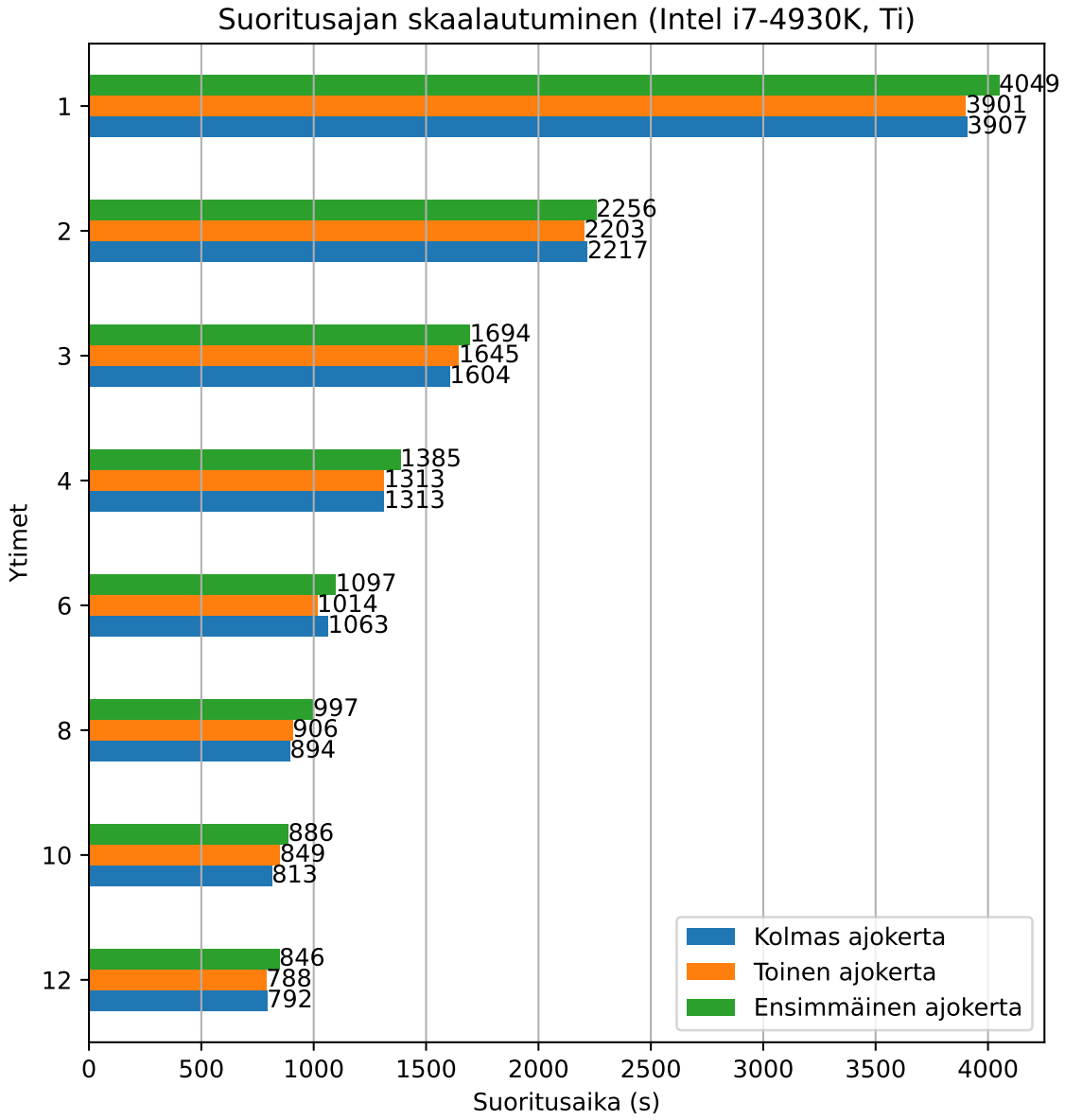
Taulukko 9. Poikkeukselliset mittaustulokset hapen skaalautumiselle, jossa jokainen ajokerta hidastuu edellisestä.

i5-6200U nopeutuu merkittävästi kolmella ensimmäisellä säikeellä, joista viimeinen on Hyperthreading-säie. Neljäs säie puolestaan lyhentää suoritusaikaa suhteellisesti vähemmän, mutta silti huomattavasti. Suoritusaika tippuu kaikilla neljällä säikeellä 0.47–0.42-kertaiseksi yksisäikeisestä konfiguraatiosta.

Tulokset suoritusaikojen skaalautumisesta titaanille löytyvät kuvioista 9, 10 ja 11. Taulukkoon 10) on koottuna titaanin suoritusajan suhteellinen skaalautuminen.



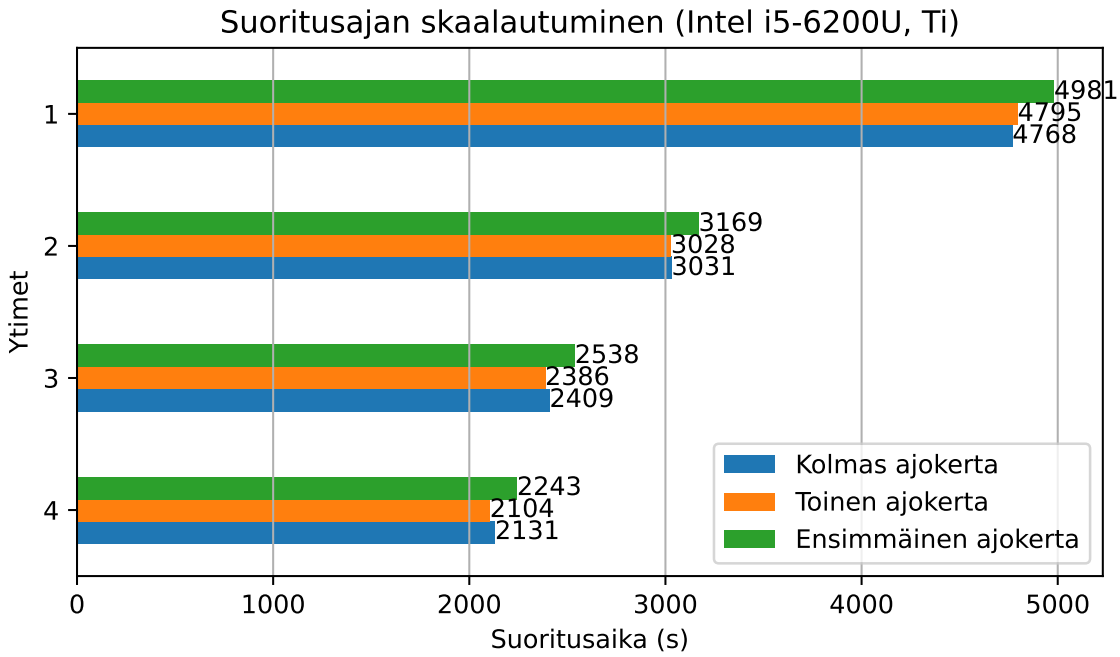
Kuvio 9. Rinnakkaistetun Numba-toteutuksen suoritusajan skaalautuminen titaanille eri ydinmäärillä i7-12700K-prosessorilla.



Kuvio 10. Rinnakkaistetun Numba-toteutuksen suoritusajan skaalautuminen titaanille eri ydinmäärillä i7-4930K-prosessorilla.

Suoritin	Säikeitä	I ajokerta	II ajokerta	III ajokerta
i7-12700K	1	**1.00**	0.97	0.98
i7-12700K	2	0.69	0.67	0.58
i7-12700K	3	0.50	0.56	0.51
i7-12700K	4	0.48	0.46	0.45
i7-12700K	6	0.42	0.44	0.41
i7-12700K	8	0.42	0.42	0.41
i7-12700K	10	0.42	0.43	0.42
i7-12700K	12	0.44	0.41	0.42
i7-12700K	14	0.40	0.39	0.41
i7-12700K	16	0.39	0.37	0.38
i7-12700K	18	0.36	0.36	0.35
i7-12700K	20	0.37	0.34	0.33
i7-4930K	1	**1.00**	0.96	0.96
i7-4930K	2	0.56	0.54	0.55
i7-4930K	3	0.42	0.41	0.40
i7-4930K	4	0.34	0.32	0.32
i7-4930K	6	0.27	0.25	0.26
i7-4930K	8	0.25	0.22	0.22
i7-4930K	10	0.22	0.21	0.20
i7-4930K	12	0.21	0.19	0.20
i5-6200U	1	**1.00**	0.96	0.96
i5-6200U	2	0.64	0.61	0.61
i5-6200U	3	0.51	0.48	0.48
i5-6200U	4	0.45	0.42	0.43

Taulukko 10. Rinnakkaistetun Numba-toteutuksen suoritusaikojen suhteellinen skaalautuminen titaanille. Vertailuarvo lihavoitu.



Kuvio 11. Rinnakkaistetun Numba-toteutuksen suoritusajan skaalautuminen titaanille eri ydinmäärillä i5-6500U-prosessorilla.

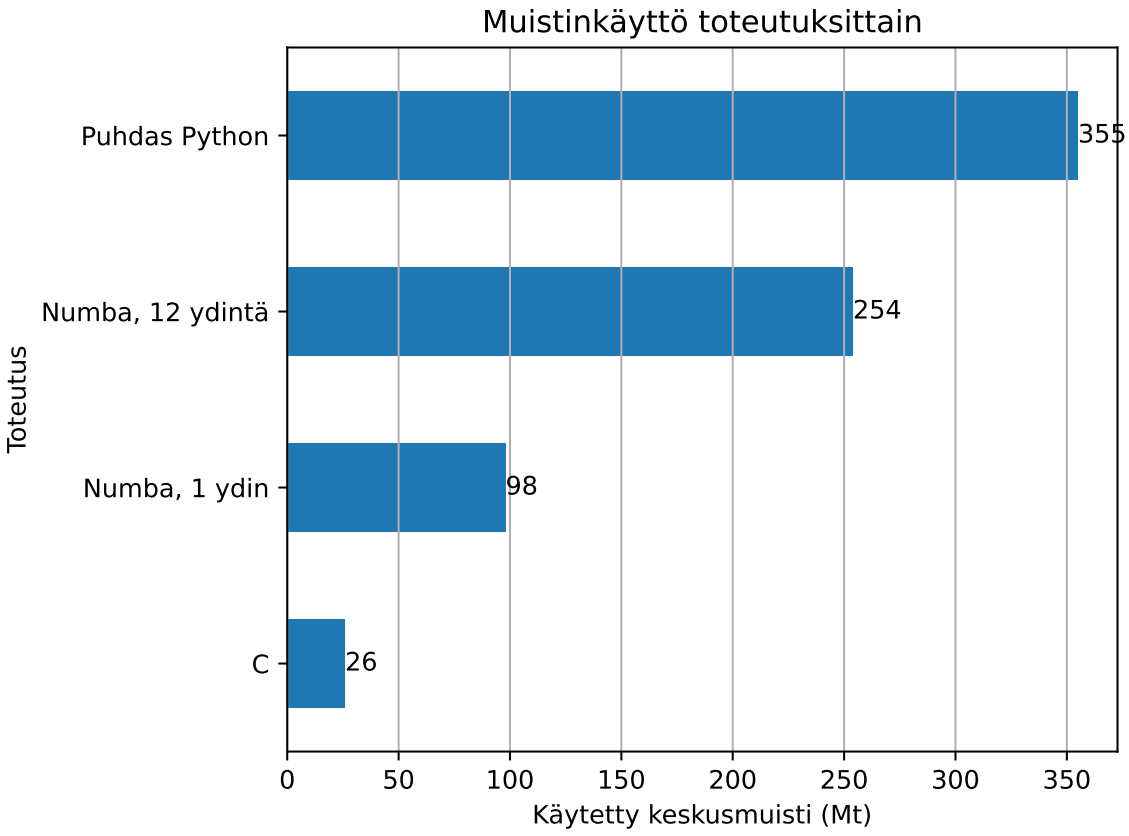
Titaani skaalautuu pääpiirteittäin samalla tapaa kuin happi. i7-12700K:n ja i5-6200U:n osalta alkuaineiden erot näkyvät säiemäärien ääripäissä: titaani skaalautuu hieman happea paremmin pienillä säiemäärillä, kun taas suurilla säiemäärillä tilanne on vastakkainen. i7-4930K puolestaan skaalautuu titaanilla happea hieman heikommin kaikilla säiemäärillä.

5.3 Muistinkäyttö

Kuviossa 12 on listattuna toteutusten muistinkäyttö titaania simuloitaessa. Luin käytetyn muistin määrän Windowsin Tehtävienhallinnasta, ohjelman prosessijoukon kohdalta. Ohjelmien todellista muistinkäyttöä saattavat kasvattaa apuprosessit, jotka ovat ryhmiteltynä Tehtävienhallinnassa muualle. Jätän mahdolliset apuprosessit huomioimatta, koska pääprosessit antavat riittävän kokonaiskuvan.

Alkuperäinen C-ohjelma kuluttaa muistia 26 Mt:a. Yksisäikeinen Numba taas käyttää 98 Mt:a muistia, joka on noin 3.8-kertainen määrä C:hen verrattuna. 12-säikeiseksi rinnakkaistettu Numba vaatii 254 Mt:a muistia, joka on noin 9.7-kertainen määrä C-toteutuksen muistia tai 2.6 kertaa yksisäikeisen Numban muistinkäyttö. Puhdas Python vie muistia 355 Mt:a eli 13.6:n C-version tai 3.6:n yksisäikeisen Numba-version verran.

Kokonaismuistinkäytön lisäksi selvitin NumPy-objektien teoreettiset minimikoot, jotka löytyvät taulukosta 11. Luvut ovat peräisin NumPyn *nbytes*-attribuutista, joka kattaa taulukon sisällön mutta ei muita attribuutteja (NumPy Developers 2022). Yksisäikeisellä Numba-ohjelmalla muistia tarvitaan vähintään 82.8 megatavua, jos muistia ei pakata. Tällöin yleiskustannukseksi jää noin 15 megatavua (16 prosenttia), joka kattaa varsinaisen Pythonin, käytetyt kirjastot sekä funktiot. 12-säikeisessä versiossa yleiskustannuksen määrä kasvaa noin 182 megatavuun (65 prosenttia), kun jaettavat kirjoituspuskurit on yliallokoitu vakiona 1.2-kertaisiksi (100 megatavuun). Yliallokoin kirjoituspuskurit (Erd_buf ja Range_buf) varmuuden vuoksi, jotta ne eivät täyttyisi ennenaikaisesti ohjelman satunnaisuuden takia.



Kuvio 12. Muistinkäyttö titaania simuloitaessa.

Objekti	Koko (Mt)	Muutos rinnakkaistaessa
Presimus	45.8	pilkotaan
Erd_buf	18.5	pilkotaan
Range_buf	16.8	pilkotaan
Scattering	1.6	pysyy vakiona
Target	0.2	pysyy vakiona
Detector	0.0	pysyy vakiona
Ions	0.0	monistetaan
Master	0.0	monistetaan
Global	0.0	monistetaan
SNext	0.0	monistetaan
Potential	(0.0)	poistetaan ennen simulaatiota
Summa	82.8	-

Taulukko 11. Numba-toteutuksen NumPy-objektien laskennalliset minimikoot.

5.4 Simulaatiotulosten jakaumat

Luvun mittausdata on tuotettu i7-4930K-prosessorilla happea simuloiden. Muut testilaitteet tuottavat samankaltaista dataa, joten niiden tuloksia ei käsitellä ajan säästämiseksi. Tarkasteltavat toteutukset taas ovat pääsääntöisesti C ja yksisäikeinen Numba. Muita toteutuksia ei käsitellä tarkemmin, koska niiden .erd-tiedostojen tulokset ylikirjoittuivat vahingossa. Kehitysvaiheen kokeilun ja ionien lopputilojen jakaumien perusteella ylikirjoittuneet tulokset olisivat olleet samanlaisia kuin luvussa käsitellyt.

Taulukossa 12 on vertailu simuloitujen ionien lopputiloista, ja taulukossa 13 on näiden arvojen suhteellinen jakauma alkuperäiseen C-versioon nähden. Toisin kuin muissa kohdissa, myös rinnakkaistetun version tulokset ovat saatavilla, sekä i7-12700K-prosessorilla ajetun puhtaan Pythonin tulokset.

Pääioneja on yhtä paljon kaikissa toteutuksissa, kun taas rekyyli-ioneja on Python-toteutuksissa 25.7–27.7 prosenttia enemmän kuin alkuperäisessä. Yhteensä simuloitavia ioneja on tällöin 22.8–24.6 prosenttia enemmän kuin alkuperäisessä. Ionien suhteellinen jakauma on sen sijaan kaikilla toteutuksilla samanlainen. Yksisäikeiset toteutukset tuottavat deterministisesti samoja tuloksia eri suorituskerroilla, kun taas rinnakkaistetun version tulokset vaihtelevat säikeiden suoritusjärjestyksen erojen vuoksi.

Taulukoissa 14 ja 15 on yhteenveto simuloidusta ERD-tapahtumadatasta. Tarkastellut arvot ovat ionin energia (mega-elektronivoltteina), rekyyli-ionin kemiallinen järjestysluku ja atomimassa, ionin syvyys näytteessä (nanometreinä), tilastollinen paino, viettämä aika ilmaisimessa (nanosekunteina) sekä ilmaisimen osumakohdan x- ja y-koordinaatit (millimetreinä). Luvut ovat samankaltaisia kummallakin toteutuksella tilastollista painokerrointa lukuun ottamatta.

Taulukoissa 16 ja 17 on yhteenveto ulko- ja sisäsilmukoiden lukumäärästä. Simuloitavien ionien määrä on teknisistä syistä 110 000 tyypillisen 1 100 000:n sijaan. Esisimulaatiossa kierrosten lukumäärä on suunnilleen sama kummallekin toteutukselle. Sen sijaan pääsimulaation aikana Numba-toteutuksessa on noin 23.9 prosenttia enemmän kierroksia ulkosilmukassa ja 24.7 prosenttia sisäsilmukassa C-toteutukseen verrattuna.

Ioni	Lopputila	C	Numba	P. Python	Numba (12-T)
Pääioni	NOT_FINISHED	0	0	0	0
Pääioni	FIN_STOP	1218	1241	1241	1 285
Pääioni	FIN_TRANS	931 523	931 471	931 461	931 394
Pääioni	FIN_BS	515	543	552	568
Pääioni	FIN_RECOIL	0	0	0	0
Pääioni	FIN_NO_RECOIL	77	78	79	86
Pääioni	FIN_MISS_DET	0	0	0	0
Pääioni	FIN_OUT_DET	0	0	0	0
Pääioni	FIN_DET	0	0	0	0
Pääioni	FIN_MAXDEPTH	166 667	166 667	166 667	166 667
Pääioni	FIN_RECTRANS	0	0	0	0
Pääioni	Yhteensä	1 100 000	1 100 000	1 100 000	1 100 000
Rekyyli	NOT_FINISHED	0	0	0	0
Rekyyli	FIN_STOP	2 379 860	3 083 312	2 983 406	3 008 226
Rekyyli	FIN_TRANS	0	0	0	0
Rekyyli	FIN_BS	0	0	0	0
Rekyyli	FIN_RECOIL	0	0	0	0
Rekyyli	FIN_NO_RECOIL	0	0	0	0
Rekyyli	FIN_MISS_DET	6 160 307	7 827 124	7 750 873	7 757 027
Rekyyli	FIN_OUT_DET	108 807	137 698	136 716	136 641
Rekyyli	FIN_DET	43 379	55 059	54 491	55 254
Rekyyli	FIN_MAXDEPTH	0	0	0	0
Rekyyli	FIN_RECTRANS	0	0	0	0
Rekyyli	Yhteensä	8 692 353	11 103 193	10 925 486	10 957 148

Taulukko 12. Hiukkasten lopputilat hapelle.

Ioni	Lopputila	C	Numba	P. Python	Numba (12-T)
Pääioni	NOT_FINISHED				
Pääioni	FIN_STOP	**1.000**	1.019	1.019	1.055
Pääioni	FIN_TRANS	**1.000**	1.000	1.000	1.000
Pääioni	FIN_BS	**1.000**	1.054	1.072	1.103
Pääioni	FIN_RECOIL				
Pääioni	FIN_NO_RECOIL	**1.000**	1.013	1.026	1.117
Pääioni	FIN_MISS_DET				
Pääioni	FIN_OUT_DET				
Pääioni	FIN_DET				
Pääioni	FIN_MAXDEPTH	**1.000**	1.000	1.000	1.000
Pääioni	FIN_RECTRANS				
Pääioni	Yhteensä	**1.000**	1.000	1.000	1.000
Rekyyli	NOT_FINISHED				
Rekyyli	FIN_STOP	**1.000**	1.296	1.254	1.264
Rekyyli	FIN_TRANS				
Rekyyli	FIN_BS				
Rekyyli	FIN_RECOIL				
Rekyyli	FIN_NO_RECOIL				
Rekyyli	FIN_MISS_DET	**1.000**	1.271	1.258	1.259
Rekyyli	FIN_OUT_DET	**1.000**	1.266	1.257	1.256
Rekyyli	FIN_DET	**1.000**	1.269	1.256	1.274
Rekyyli	FIN_MAXDEPTH				
Rekyyli	FIN_RECTRANS				
Rekyyli	Yhteensä	**1.000**	1.277	1.257	1.261

Taulukko 13. Hiukkasten suhteelliset lopputilat hapelle. Vertailuarvo lihavoitu.

Suure	Rivejä	Keskiarvo	Keskihajonta	Min	25 %	50 %	75 %	Max
energy	43 379	2.81	1.25	0.99	1.73	2.57	3.89	5.18
recoil_Z	43 379	8.00	0.00	8.00	8.00	8.00	8.00	8.00
recoil_mass	43 379	16.00	0.10	15.99	15.99	15.99	15.99	18.00
sample_depth	43 379	156.0	121.3	0.0	33.9	147.8	260.3	400.0
stat_weight	43 379	25 669	13 341	1 877	20 195	27 035	34 441	131 450
detector_time	43 379	108.9	23.2	77.2	86.9	105.1	126.2	170.4
detector_hit_x	43 379	0.00	3.48	-6.99	-2.77	-0.02	2.79	6.98
detector_hit_y	43 379	0.04	8.72	-17.48	-7.02	0.00	7.13	17.49

Taulukko 14. Yhteenveto .erd-tiedoston tuloksista C-toteutuksella hapelle. Otsikoiden prosenttiluvut ovat kvartiileja.

Suure	Rivejä	Keskiarvo	Keskihajonta	Min	25 %	50 %	75 %	Max
energy	55 059	2.81	1.24	0.99	1.73	2.58	3.87	5.24
recoil_Z	55 059	8.00	0.00	8.00	8.00	8.00	8.00	8.00
recoil_mass	55 059	15.99	0.10	15.99	15.99	15.99	15.99	18.00
sample_depth	55 059	156.3	121.4	0.0	35.2	147.6	261.1	400.0
stat_weight	55 059	20 270	10 500	1 476	15 984	21 289	27 220	125 376
detector_time	55 059	109.0	23.3	75.7	87.1	105.0	126.2	165.3
detector_hit_x	55 059	-0.00	3.47	-6.99	-2.74	-0.03	2.76	7.00
detector_hit_y	55 059	-0.02	8.72	-17.48	-7.05	-0.01	7.04	17.48

Taulukko 15. Yhteenveto .erd-tiedoston tuloksista Numba-toteutuksella hapelle. Otsikoiden prosenttiluvut ovat kvartiileja.

Toteutus	Vaihe	Ioneja	Keskiarvo	Keskihajonta	Min	25 %	50 %	75 %	Max
C	Esisim.	10 000	1 685	572	100	1 296	1 633	2 012	8 378
Numba	Esisim.	10 000	1 690	578	20	1 304	1 638	2 014	8 897
C	Pääsim.	100 000	2 080	1 228	4	1 433	2 197	2 884	12 306
Numba	Pääsim.	100 000	2 577	1 474	4	1 881	2 754	3 527	15 382

Taulukko 16. Ulomman simulaatiosilmukan toistomäärät hapelle. Otsikoiden prosenttiluvut ovat kvartiileja.

Toteutus	Vaihe	Ioneja	Keskiarvo	Keskihajonta	Min	25 %	50 %	75 %	Max
C	Esisim.	10 000	1.196	0.944	1	1	1	1	30
Numba	Esisim.	10 000	1.196	0.935	1	1	1	1	27
C	Pääsim.	100 000	9.798	4.394	1	7	10	13	69
Numba	Pääsim.	100 000	12.213	5.332	1	9	13	16	74

Taulukko 17. Sisemmän simulaatiosilmukan toistomäärät hapelle. Otsikoiden prosenttiluvut ovat kvartiileja.

6 Pohdinta

Tässä luvussa analysoin saatuja mittaustuloksia, tarkastelen tuotettuja testiohjelmia, käsittelen kohdattuja ongelmia ja esitän mahdollisia aiheita jatkotutkimukselle. Samalla vastaan tutkimuskysymyksiin.

6.1 Suorituskyvyn analyysi

Tässä aliluvussa analysoin toteutuneiden testiohjelmien suorituskykyä.

6.1.1 Odotusten toteutuminen

Kuten odotin, alkuperäinen C-ohjelma oli yksisäikeisistä toteutuksista selkeästi nopein. Perustin odotukseni siihen, että C:ssä ei ole niin paljoa ”ylimääräisiä” abstraktiotasoja kuin Pythonissa, jolloin suorituskyky lähestyy teoreettista maksimia C:n vähäisen yleisrasitteen tason ansiosta.

Vastaavasti puhdas Python oli selkeästi hitain, kuten oli odotettavissa. Toteutuksen kohtuuttoman pitkän suoritusajan vuoksi en mitannut sitä hitaammilla laitteilla tai raskaammalla titaanilla, sillä kumpikin vaihtoehto olisi yksinään kestänyt noin 2.0–3.0-kertaisen ajan muiden tulosten perusteella arvioituna. Silloin suoritusaika olisi jommalla kummalla vähintään 30 tuntia ja kummatkin yhdessä noin 60–135 tuntia (2.5–5.6 päivää).

Odotin yksisäikeisen Numba-toteutuksen pääsevän suoritusajaltaan noin 25 prosentin päähän alkuperäisestä. Päädyin tähän lukuun tulkitsemalla C-version ”täydelliseksi” suoritusajan alirajaksi, jonka päälle kasautuvat Pythonin ja Numban yleisrasitteet. Simuloinnin osuus on suuri, jolloin kiihdyttämättömien osuuksien merkitys kokonaisaikaan jää melko pieneksi. Testilaitteista i7-12700K saavutti tavoitteen kummallakin alkuaineella ja i5-6200U oli melko lähellä hapen osalta. Sen sijaan titaanilla i5-6200U jäi kauemmaksi tavoitteesta, ja i7-4930K oli edeltäviä hitaampi kummallakin alkuaineella.

Simulaatiosta mitattujen kierrosmäärien perusteella liiallista simulointia aiheuttava ohjelmointivirhe pidentää suoritettujen simulaatiosilmukoiden määrää noin 24 prosenttia. Koska

saadut simulaatiotulokset eivät rekyyli-ionien määrää lukuun ottamatta muuttuneet alkuperäiseen verrattuna, voidaan bugin olettaa kasvattavan suoritetun simuloinnin määrää tämän verran. Korjattu suoritusaika voi siis olla noin $1 - (0.24/1.24) \approx 0.806$-kertainen esi- ja pääsimulaation osalta, jotka puolestaan kattavat vähintään 97.9 prosenttia yksisäikeisten toteutusten kokonaissuoritusajasta. Jos kyseistä virhettä ei olisi, Numba-toteutus saattaisi jopa alittaa alkuperäisen C-toteutuksen suoritusajan i7-12700K-prosessorilla. Myös puhtaan Pythonin toteutus olisi nopeampi, mutta edelleen hyvin hidas. Arvioin bugin vaikutusta suoritusaikaan tarkemmin luvussa 6.1.5.

Odotin rinnakkaistetun Numba-version skaalautuvan lähes lineaarisesti säikeiden määrän mukaan, sillä yksittäisten ionien simulaatiot ovat toisistaan riippumattomia, eivätkä muut vaiheet kestä kauaa. i7-4930K ja i5-6200U pääsivät lähelle tätä tavoitetta Hyperthreading-ominaisuuden avulla, kun taas i7-12700K jäi pitkälle tavoitteesta.

Esimerkiksi i7-4930K:n kuuden säikeen ihanneaika saadaan arvioimalla rinnakkaistamattoman osuuden kestoksi noin 0.03 yksisäikeisen Numban kokonaisajasta, jolloin rinnakkaistuva loppuosa olisi noin $(1.00 - 0.03)/6 \approx 0.16$. Lisäksi rinnakkaistamisen ylläpito mm. säikeiden luonnin ja yhdistämisen osalta vie jonkin verran aikaa, ehkä noin 0.01 kertaa alkuperäisestä (18–41 s). Pelkillä fyysisillä ytimillä saavutettiin 0.25–0.30 kertaa alkuperäinen aika, mutta Hyperthreading-ytimillä kerroin oli lopulta 0.19–0.24, joka on lähellä tavoitteena ollutta 0.20:n kerrointa. Ihannetilanteessa jo fyysisillä ytimillä olisi saavutettu tavoitenopeus, jolloin Hyperthreading olisi nopeuttanut suoritusta entisestään tuntemattoman määrän. Koska Hyperthreading-ominaisuuden hyöty riippuu laskentakuormasta (Zhang ym. 2016), sen tarkkaa vaikutusta on vaikea arvioida ennalta.

Koska rinnakkaistettu toteutus tuottaa samankaltaisia tuloksia kuin yhden säikeen versio, voidaan korjaamatta jääneen ohjelmointivirheen vaikutus simulaatiosilmukoiden lukumäärään arvioida vastaavaksi. Rinnakkaistetussa versiossa on suhteellisesti enemmän yleisrasitetta, joten bugin korjaamisesta aiheutuva suoritusajan parannus jäisi hieman vähemmän merkittäväksi, mutta kokonaissuoritusajan kannalta silti suureksi. Tällöin yksisäikeiseen Numba-toteutukseen verrattuna suhteellinen nopeutuskerroin heikkenisi hieman entisestään, kun taas alkuperäiseen toteutukseen nähden suoritusaika laskisi eli suorituskyky paranisi huomattavasti.

Odotin titaanin kiihtyvän ja rinnakkaistuvan happea paremmin, koska sen simulointi kestää happea paljon pidempään, jolloin optimoitavan laskennan osuutta on enemmän. Rinnakkaistumisen osalta arvioni piti paikkansa kannettavaa i5-6200U-prosessoria lukuun ottamatta. Yhden säikeen Numba-toteutus taas kiihtyi hapella yhtä hyvin tai paremmin kuin titaanilla kaikilla testilaitteilla.

6.1.2 Yleisrasite ja nopeutuminen

Vastoin odotuksiani C-toteutuksen toisten kierrosten suoritusajat olivat jopa joitakin prosentteja nopeammat kuin ensimmäisten kierrosten, vaikka C-toteutuksen ei pitäisi olla ajonaikaisesti JIT-optimoitava. Ilmeisesti myös staattisesti käännetyille ohjelmille voi tapahtua jonkinlaista optimointia. En saanut selville miten ja mihin mahdolliset optimoinnin välitulokset tallentuvat, joten en kyennyt säätelemään optimointia poistamalla välituloksia, kuten Python-toteutuksessa. Seurauksena C-ohjelman tuloksissa on pientä epävarmuutta siitä, mitkä ajot ovat optimoidut.

C-toteutuksen havaittu nopeutuminen voi johtua myös satunnaisvaihtelusta, sillä jotkin suorituskerrat ovat hitaampia toisella kuin ensimmäisellä kerralla. Nopeutuminen havaittiin kaksiytimisellä prosessorilla, jonka pieni ydinmäärä voi myös saturoitua taustaprosesseista helposti. Ensimmäisen suorituskerran aikana on siis saattanut käynnistyä yhtä aikaa jokin ajastettu toiminto, kuten virustorjunnan päivitys. Vaikka yritin eliminoida tällaiset häiriötekijät, osa on saattanut jäädä estämättä. Mittaukset olisi voitu suorittaa Windowsin vikasietotilassa, jolloin ohjelmia olisi ollut päällä vielä vähemmän. Toisaalta vikasietotila voi aiheuttaa yhteensopivuusongelmia karsittujen ominaisuuksiensa vuoksi, eikä se vastaa kunnolla ohjelmien tavallista suoritusympäristöä, jossa taustaprosesseja on päällä.

Yksisäikeinen Numba-toteutus puolestaan nopeutui ensimmäisen kierroksen jälkeen suhteellisesti vähemmän kuin odotin, vain joitakin prosentteja. Pääsyy vähäiseen suhteelliseen nopeutumiseen on todennäköisesti se, että vanhemmilla laitteilla suoritusajat alkavat noin puolesta tunnista, jolloin minuutin nopeutuminen vastaa vain noin kolmea prosenttia. Matala uusintakierrosten nopeutumisen aste viittaa kolmeen mahdolliseen johtopäätökseen: Numba-optimoinnin yleisrasitteen määrä on alhainen mutta vaikutus hyvä, optimointi on ainakin

suunnilleen vakioaikaista, tai optimointi on nopeaa koska se ei toteudu kovin tarkkaan.

Matalan yleisrasitteen tasoa tukee se, että osa ensimmäisen ajokerran suoritusajasta kuluu puhtaalla Pythonilla kirjoitettujen osien kääntämiseen, joten Numban osuus on vain osa yleisrasitteesta. Täten pidän päätelmää todennäköisenä.

Vakioaikaisen optimointi- ja kääntämisajan puolesta puhuu se, että pitkäkestoisimmat titaanisimulaatiot nopeutuivat suhteellisesti vähemmän kuin happisimulaatiot. Toisaalta titaanisimulaatiotkaan eivät lähestyneet C:n nopeutta, joten osa Numban yleisrasitteesta kuuluu ilmeisesti suoritusaikaan, mikä selittäisi miksi Numba on kymmenien prosenttien päässä C:stä. Ilman profilointitukea tätä on vaikea varmentaa, mutta ainakin mitatut osat yleisrasitteesta ovat vakioaikaiset hiukkasten massan osalta. Sen sijaan suuremmat ionimäärät johtavat suurempiin objekteihin, jotka ovat todennäköisesti myös hitaampia käsitellä.

Numban mahdollinen suorituksenaikainen yleisrasite viittaa myös kolmanteen vaihtoehtoon. Tällöin optimoinnissa on ainakin teoriassa parannettavaa nopeuseron verran. Puhtaan Python-toteutuksen suoritusajan huomioiden Numba kuitenkin kiihdyttää suoritusta varsin hyvin, joten pitäisin nopeuserosta C:hen huolimatta optimoinnin tasoa varsin hyvänä, jolloin kolmas johtopäätös ei ole voimassa.

Simulaatiomäärää nostavan bugin ei pitäisi vaikuttaa Python-toteutusten ajokertojen väliseen nopeutumiseen merkittävästi, sillä se kasvattaa jokaisen ajokerran silmukkamäärää yhtä paljon. Tämä tukee osaltaan ensimmäistä ja toista päätelmää Numba-optimoinnin osalta, sillä oikealla simulaatiomäärällä suoritusaika olisi nykyistä pienempi. Myös tämä on ristiriidassa kolmannen päätelmän kanssa, sillä suoritusaika pääsee lopulta lähelle alkuperäistä toteutusta, joka on myös hyvin optimoitu.

Puhtaalla Pythonilla toteutettuna MCERD taas on niin hidas, että funktioiden kääntämiseen ja olioiden alustamiseen kuluva aika peittyy täysin simulaatio-osuuden pitkään kestoon. Tällöin toteutuksen yleisrasite löytyy lähes yksinomaan varsinaisen laskentakoodin suorituksesta. Koska Numba-toteutusten uusinta-ajokertojen nopeutuminen vaikuttaa tulevan erityisesti funktioiden kääntämisestä, on perusteltua sanoa, että puhtaalla Pythonilla uusintakierrosten vaikutus nopeuteen olisi todennäköisesti minimaalista.

Myös rinnakkaistetussa Numba-toteutuksessa MCERDistä on samat yleisrasitteet kuin yksisäikeisessäkin versiossa. Niiden lisäksi siinä on säikeiden ylläpitoon liittyvää yleisrasitetta, joka todennäköisesti näkyy odotettua heikompana skaalautumisena suuremmilla säiemäärillä.

6.1.3 Skaalautuminen

Rinnakkaistettu versio skaalautui hyvin vanhemmilla testilaitteilla, mutta uusimmalla laitteella se toimi odotettua paljon hitaammin. Suoritusaika laski 12 fyysisellä ja 20 loogisella ytimellä vain noin kolmannekseen yhden säikeen versiosta, vaikka sen voisi odottaa tippuvan helposti vähintään kuudesosaan jo 8 P-ytimellä. Vanhemmilla prosessoreilla suoritusaika putosi lähelle fyysisten ytimien määrää vastaavaa murto-osaa, joten on poikkeuksellista että uusin prosessori skaalautui näin heikosti. Siinä missä i7-4930K:n yhden säikeen suoritusaika on noin 2.25-kertainen i7-12700K-prosessoriin nähden, rinnakkaistettuna suoritus kestää enää noin 1.33-kertaisesti maksimiydinmäärällä. Tämä on erityisen heikko tulos suhteutettuna siihen, että i7-4930K:ssa on vain 6 fyysistä ydintä, kun i7-12700K:ssa niitä on 12.

Juurisyy i7-12700K:n huonoon rinnakkaistumiseen jää tässä tutkielmassa selvittämättä, sillä mikään löytämistäni syistä ei selittänyt ilmiötä luotettavasti. Aiempana esitetyistä, synteettisistä suorituskykymittauksista nähtiin, että myös uuden prosessorin pitäisi rinnakkaistaa laskukuormaa kuten vanhempienkin prosessorien, joten yleisesti huono rinnakkaistuminen ei todennäköisesti ole syynä. Itse ohjelmassa taas ei voi olla yleisluonteista ongelmaa rinnakkaistumisen kanssa, sillä kaksi muuta prosessoria toimivat odotetusti.

Kolmas mahdollinen syy on i7-12700K-hybridiprosessorin käyttäminen Windows 10 -käyttöjärjestelmällä, sillä tämän on raportoitu aiheuttavan joidenkin ohjelmien kohdalla suorituskykyongelmia ("W1zzard" 2021), koska Windows 10:n säikeidenhallinta ei ole yhtä hyvin optimoitu hybridiprosessoreille kuin Windows 11:ssä. Laskentakuorma saattaa nimittäin kategorisoitua väärin, jolloin suuren prioriteetin säikeitä päätyy suoritettavaksi E-ytimillä vaikka P-ytimiä olisi vapaina, mikä huonontaa suorituskykyä. Tämä ei kuitenkaan selitä heikko skaalautumista, sillä esimerkiksi maksimisäiemäärällä kaikkia ytimiä käytetään yhtä aikaa.

Totesin tämän myös kytkemällä E-ytimet pois päältä, eikä tämä juurikaan vaikuttanut ainakaan maksimisäikeiden suorituskykyyn. Lisäksi Tehtävienhallinnan mukaan E-ytimiä ei käytetty simulointiin ennen kuin säikeiden määrä ylitti kahdeksan eli fyysisten P-ytimien määrän, joten Numba-versio kategorisoitui oikein.

Myös epästabiili muisti- tai suoritinylikellotus voi aiheuttaa suorituskyvyn heikkenemistä. Näin ei kuitenkaan ollut tässä tapauksessa, sillä kun muisti ja suoritin olivat vakioasetuksilla, suorituskyky ei ollut ylikellotettua parempi.

6.1.4 Mittausasetelman luotettavuus

Mittausasetelmat eivät täysin kuvasta ohjelmien todellista käyttöä. Ensimmäinen ajokerta on epätavallisen pitkä, koska siinä nollataan kaikki esikäännökset. Oikeasti näin tapahtuisi vain kerran sovelluksen käyttöönoton jälkeen, jolloin käytettävät kirjastot prosessoidaan kokonaan. Tämän jälkeen simulaatiokonfiguraation vaihtaminen vaatii vain joidenkin Numba-optimoitujen funktioiden kääntämistä uudelleen, koska käytettyjen objektien koko ja täten rakenne muuttuu. Tavallisten moduulien sekä osan JIT-funktioista pitäisi sen sijaan toimia ilman uudelleen kääntämistä, parametreista riippumatta, koska niiden tavukoodi ei riipu tietynkokoisista objekteista.

Toinen kierros taas on hieman virheellinen kumpaankin suuntaan, sillä toisaalta osa JIT-kiihdytetyistä funktioista ei tallennu levylle ensimmäisen kierroksen jälkeen Numban ajonaikaisten ilmoitusten perusteella. Toisaalta joidenkin parametrien muuttaminen vaikuttaa objektien kokoon, jolloin Numba kääntäisi näitä rakenteisia taulukoita käyttävät aliohjelmat uudelleen. Kolmas ajokerta ei puolestaan olisi tavallisessa käytössä lainkaan mielekäs, sillä samojen laskujen toistaminen ei yleisesti tuota uusia tuloksia rinnakkaistetun version pientä vaihtelua lukuun ottamatta. Viimeisen ajokerran tarkoitus onkin tuottaa lisädataa ohjelman suorituskyvyn luonteesta gradussa analysoitavaksi.

Oikeassa käytössä pitää myös huomioida vaaditun laskentatehon ja suoritusajan suhde, sillä yleensä halutaan simuloida useita alkuaineita tai isotooppeja kerralla, jolloin MCERD pitää ajaa monta kertaa. Koska rinnakkaistetun version suoritusaika ei skaalaudu lineaarisesti säiemäärään nähden, on todennäköisesti tehokkainta järjestellä simulaatiot siten, että kaik-

kia ajetaan tasaisesti yhtä aikaa. Tällöin koko prosessorin laskentakapasiteetti on täydessä käytössä mahdollisimman pitkään, ja rinnakkaistumisen yleisrasitteen määrä on minimoitu. Jakoa vaikeuttaa se, että painavampien alkuaineiden simulointi kestää pidempään kuin kevyempien. Lisäksi hybridiarkkitehtuuriprosessorien kaikki ytimet eivät ole yhtä nopeita. Näistä syistä esimerkiksi raskaampien aineiden simulaatiot saattavat jäädä päälle vain osalle ytimistä, kun kevyempien aineiden simulaatiot ovat jo ehtineet loppua.

6.1.5 Ohjelmointivirheen vaikutus suoritusaikaan

Kuten on jo aiemmin mainittu, Python-toteutuksiin jäänyt ohjelmointivirhe lisää simulaatiosilmukoiden määrää yhteensä noin 24 prosenttia, mikä pitkittää esi- ja pääsimulaation laskentaosuuksia. Koska en ehtinyt selvittämään bugin syytä, arvioin sen vaikutusta suoritusaikaan teoreettisesti.

Arvioin bugittoman suoritusajan kaavalla 6.1

$$korjattu\_aika = kokonaisaika - (1 - korjauskerroin) * (\frac{2}{3} * e.sim.\_kesto + p.sim.\_kesto) \tag{6.1}$$

, jossa $\frac{2}{3}$ on esisimulaation simuloinnin arvioitu osuus, $e.sim.\_kesto$ on esisimulaation kesto ja $p.sim.\_kesto$ on pääsimulaation kesto. Mittaustulosten perusteella ensimmäisellä ajokerralla noin kolmannes esisimulaatiosta kuluu simulaatiosilmukan kääntämiseen sekä muuhun oletettavasti vakioaikaiseen yleisrasitteeseen. Muilla ajokerroilla tämän yleisrasitteen osuus on pienempi, mutta se jätetään nyt huomioimatta yksinkertaisuuden vuoksi. Jo aiemmin laskettu korjauskerroin on $1 - (0.24/1.24) \approx 0.806$, joka perustuu siihen että laskettavaa on 1.24-kertaisesti niin paljon kuin pitäisi.

Korjatut suoritusajat löytyvät absoluuttisina taulukosta 18 ja suhteellisina taulukosta 19. Arvojen perusteella yhden säikeen Numba-toteutus on täysin vertailukelpoinen alkuperäisen kanssa i7-12700K-prosessorilla, hieman hitaampi i5-6200U:lla ja hieman enemmän hitaampi i7-4930K:lla. Silti jopa suhteellisesti hitainkin tapaus alittaa alkuperäisen odotukseni, joka oli siis 1.25-kertainen C:hen verrattuna. Rinnakkaistetut toteutukset ovat puolestaan C:tä entistä nopeampia, mutta niiden yleisrasitteen määrä korostuu yhteen säikeeseen nähden. Puhdas Python taas on edelleen hyvin hidas.

Arvioidut suoritusajat saattavat olla hieman optimistisia, sillä vaikuttaa epätodennäköiseltä että Python-toteutus voi olla alkuperäistä C-ohjelmaa kaksi prosenttia lyhyempi. Arvioista puuttuu GSTO, mutta sen suoritusaika on vain 3.2 sekuntia i7-4930K:lla, joten se ei käytännössä muuta tuloksia. Mitatuista silmukka- ja ERD-tapahtumamääristä päätellen oikea suoritusaika on kuitenkin lähellä arvioitua.

Suoritin	Toteutus	Ajokerta	Kesto (O)	Kesto (Ti)
i7-12700K	Python	I	733:50	-
i7-12700K	C	I	10:26	24:19
i7-12700K	C	II	10:25	23:53
i7-12700K	JIT	I	10:27	24:21
i7-12700K	JIT	II	10:12	23:51
i7-12700K	JIT	III	10:25	24:06
i7-12700K	JIT (20-T)	I	4:04	8:15
i7-12700K	JIT (20-T)	II	3:42	8:16
i7-12700K	JIT (20-T)	III	3:40	7:59
i7-4930K	C	I	19:46	43:55
i7-4930K	C	II	19:38	44:10
i7-4930K	JIT	I	24:07	55:35
i7-4930K	JIT	II	22:59	53:25
i7-4930K	JIT	III	23:11	53:30
i7-4930K	JIT (12-T)	I	5:37	11:40
i7-4930K	JIT (12-T)	II	4:57	10:50
i7-4930K	JIT (12-T)	III	5:04	10:47
i5-6200U	C	I	26:03	58:08
i5-6200U	C	II	24:32	57:52
i5-6200U	JIT	I	29:25	67:11
i5-6200U	JIT	II	27:52	65:04
i5-6200U	JIT	III	27:36	65:21
i5-6200U	JIT (4-T)	I	13:18	30:13
i5-6200U	JIT (4-T)	II	12:33	29:19
i5-6200U	JIT (4-T)	III	12:19	28:44

Taulukko 18. Suoritusaikojen todellisiksi arvioidut kestot hapelle ja titaanille. (n-T = n-säikeinen.)

Suoritin	Toteutus	Ajokerta	Kesto (O)	Kesto (Ti)
i7-12700K	Python	I	70.31	-
i7-12700K	C	I	**1.00**	**1.00**
i7-12700K	C	II	1.00	0.98
i7-12700K	JIT	I	1.00	1.00
i7-12700K	JIT	II	0.98	0.98
i7-12700K	JIT	III	1.00	0.99
i7-12700K	JIT (20-T)	I	0.39	0.34
i7-12700K	JIT (20-T)	II	0.36	0.34
i7-12700K	JIT (20-T)	III	0.35	0.33
i7-4930K	C	I	**1.00**	**1.00**
i7-4930K	C	II	0.99	1.01
i7-4930K	JIT	I	1.22	1.27
i7-4930K	JIT	II	1.16	1.22
i7-4930K	JIT	III	1.17	1.22
i7-4930K	JIT (12-T)	I	0.28	0.27
i7-4930K	JIT (12-T)	II	0.25	0.25
i7-4930K	JIT (12-T)	III	0.26	0.25
i5-6200U	C	I	**1.00**	**1.00**
i5-6200U	C	II	0.94	1.00
i5-6200U	JIT	I	1.13	1.16
i5-6200U	JIT	II	1.07	1.12
i5-6200U	JIT	III	1.06	1.12
i5-6200U	JIT (4-T)	I	0.51	0.52
i5-6200U	JIT (4-T)	II	0.48	0.50
i5-6200U	JIT (4-T)	III	0.47	0.49

Taulukko 19. Suoritusaikojen todellisiksi arvioidut, suhteelliset kestot hapelle ja titaanille. Vertailuarvo lihavoitu. (n-T = n-säikeinen.)

Teoreettisesti bugikorjattujen tulosten perusteella vastaus tutkimuskysymykseen 1, ”Voidaanko korkeatasoisella ja hitaanakin tunnetulla Python-ohjelmointikielellä saavuttaa C-kieleen verrannollinen suorituskyky MCERD-ohjelmassa?”, on kyllä. Arvioitu vaihteluväli tälle on 0.98–1.27 kertaa alkuperäisen suoritusaika, joten Numballa toteutettu MCERD voi olla jopa vähän alkuperäistä C-versiota nopeampi. Vaikka todellinen suoritusaika ei saavuttaisi arvioitua arvoa, se on korkeintaan 1.20–1.56-kertainen korjaamattomien lukujen perusteella, mikä on silti vertailukelpoinen. Syynä suureen arvoväliin on toteutukseen jäänyt bugi, joka lisää simuloinnin määrää noin 24 prosenttia.

Vastaus tutkimuskysymykseen 3, ”Kuinka paljon rinnakkaistaminen nopeuttaa MCERD-ohjelman suorittamista?”, on: alkuperäiseen verrattuna noin 0.33–0.39-kertaiseksi Intel i7-12700K:lla (12 ydintä, 20 säiettä), 0.25–0.28-kertaiseksi Intel i7-4930K:lla (6 ydintä, 12 säiettä) ja 0.47–0.52-kertaiseksi Intel i5-6200U:lla (2 ydintä, 4 säiettä). Korjaamattomat arvot antavat ylärajaksi 0.40–0.48-kertaisen, 0.30–0.34-kertaisen ja 0.58–0.64-kertaisen arvovälin em. prosessoreille. Rinnakkaistuminen on tehokkaimmillaan pienillä säiemäärillä, suuremmilla määrillä skaalautuminen heikkenee. Tästä huolimatta Hyperthreading-ominaisuudesta on selvää hyötyä Numba-MCERDin rinnakkaistamisessa, vaikka suurilla säiemäärillä rinnakkaistamisen hyötysuhde laskee.

6.1.6 Muistinkäyttö

Alkuperäinen C-ohjelma on muistinkäytöltään selvästi tehokkain. Sen muistinkäyttö on objektien alustuksen jälkeen noin 3.6 megatavua, josta se kasvaa tasaisesti 26 megatavuun esisimulaation aikana. Syy on uusien Presimu-olioiden alustaminen pitkin esisimulaatiota, sillä heti esisimulaation analysoinnin jälkeen Presimu-oliot poistetaan, ja muistinkäyttö tippuu samalla takaisin 3.6 megatavuun, jossa se pysyy koko pääsimulaation. Tulosten tallentamiseen ei kulu paljoa muistia, sillä ne kirjataan suoraan tiedostoihin.

Puhtaalla Pythonilla toteutettu versio on muistinkäytöltään puolestaan tehottomin. Se kasvaa noin 355 megatavuun heti objektien alustamisessa, josta lukema ei juurikaan muutu, sillä mitään merkittäviä objekteja ei enää luoda tai poisteta. Syy suureen muistinkäyttöön löytyy Pythonin toteutuksesta, jossa jokainen arvo on olio (Lutz 2007, luku 4), mikä kuluttaa

enemmän muistia kuin perustietotyyppien käyttö.

Numballa optimoitu, yksisäikeinen versio on muistinkäytöltään toiseksi tehokkain. Alustuksessa se on vähän suurempi kuin perus-Python-versio, sillä se käyttää samaa koodia, minkä lisäksi muistissa ovat Numba- ja NumPy-kirjastot. Muutaman sekunnin kestävän alustuksen jälkeen Python-objektit korvataan huomattavasti pienemmillä NumPy-taulukoilla, jolloin ohjelman koko laskee 93 megatavuun. Lukema kasvaa simulaation aikana joitakin megatavuja. Koska muistinkäyttö on suurta vain lyhyen alustuksen aikana, sillä ei pitäisi olla huomattavaa merkitystä suoritusajan kannalta, vaikka muisti loppuisi hetkellisesti kesken.

Monella ytimellä suoritettava Numba-versio käyttäytyy ennen simulaatiota samalla tapaa kuin yksisäikeinen versio. Tämän jälkeen muistinkäyttö kasvaa 12-säikeisenä 254 megatavuun, koska jokainen laskentaprosessi vaatii oman muistinsa. Kyseessä on ainoa monisäikeinen toteutus, joten se on lähtökohtaisesti muita suurempi. Myös C-versio käyttäisi muistia enemmän rinnakkaistettuna, mutta luultavasti huomattavasti tehokkaammin sen yleisesti pienemmän koon ja täten yleiskustannuksen vuoksi.

Yksisäikeisen Numban ja C-version erot selittyvät kolmella tekijällä. Jos Numbastakin voisi kirjoittaa suoraan tiedostoon, puskureita ei enää tarvittaisi, jolloin muistikäyttö vähenisi puskureiden verran, 98.0 megatavusta 62.7 megatavuun. Presimus-taulukko voisi taas olla kuudenneksen pienempi, jos siinä käytettäisiin 32-bittistä kokonaislukusaraketta 64-bittisen sijaan kuten C-versiossa, jolloin ohjelman koko laskisi 55.0 megatavuun. Viimeinen ero on muistinvaraamisessa: C:n *malloc* varaa muistia mutta ei alusta objekteja kerralla, jolloin kuluu vasta Windowsin virtuaalimuistia (Microsoft 2021). Sitten kun C-ohjelma aidosti käyttää tätä muistia, se varataan fyysisestä muistista, kun taas Python-versiossa objektien alustaminen ja muistin täyttyminen tapahtuu heti. Tällä on erityisesti vaikutusta Presimus-taulukkoon, joka on yliallokoitu kaksinkertaisena jo alkuperäisversiossa. Jos Pythonin muistinvaraus toimisi kuten C:ssä, Presimus-taulukon koko suunnilleen puolittuisi ja muistia kuluisi lopulta noin 36.0 megatavua. Näillä muutoksilla Numba ja C olisivat jo hyvin lähellä toisiaan.

Lisäksi Python-toteutusten pääsimulaation muistinkäyttöä voisi karsia poistamalla Presimuoliot niiden analysoinnin jälkeen kuten C-ohjelmassa. Huomasin tämän optimointimahdolli-

suuden vasta tehtyäni mittaukset, joten se ei enää ehdi tuloksiin mukaan. Muutos vähentäisi pääsimulaation aikaista muistintarvetta Presimus-taulukon verran eli 45.8 megatavua.

Simuloinnin määrää lisäävä ohjelmointivirhe vaikuttaa Python-toteutusten muistinkäyttöön vain tulospuskurien koon kautta. Koska bugi tuottaa enemmän .erd-tuloksia, niiden tallentamiseen tarvitaan pidemmät puskurit. Sen sijaan suurimpaan objektiin eli esisimulaatiotaulukkoon bugi ei vaikuta, sillä pääionien määrä pysyy samana. Myös muiden simulaatiossa käytettävien objektien määrät pysyvät vakioina, sillä niitä tarvitaan vain yksi säiettä kohden, tai sama objekti jaetaan säikeiden kesken.

Versioiden väliset erot ovat suhteellisesti tarkasteltuna suuret, mutta absoluuttisesti pienet. Kun edullisissakin nykytietokoneissa on vähintään 8 gigatavua keskusmuistia, yhden tai kahden sadan megatavun ero on varsin merkityksetön käytettyyn prosessoriaikaan nähden. Vaikka käsitellyt objektit olisivat kertaluokkaa suuremmat, keskusmuisti ei loppuisi kesken. C-toteutusta suuremmilla objekteilla voi olla merkitystä prosessorin välimuistin ja keskusmuistin siirtonopeuksiin ja sitä kautta suoritusnopeuteen, mutta niiden vaikutusten mittaaminen olisi liian työlästä gradun puitteissa. Tuloksista voidaan päätellä, että ainakaan keskusmuistin määrä ei ole rajoittava tekijä MCERDin suorituskyvylle.

Keskusmuistinkäytön sijaan prosessorin välimuistinkäyttö olisi kiinnostavampi tutkimuksen kohde, sillä se voi realistisesti loppua kesken pienen kokonsa vuoksi. Esimerkiksi uusimman prosessorin, Intel i7-12700K:n, tehokkaammilla P-ytimillä on ydinkohtaista välimuistia vain 80 kt L1-tasolla (TechPowerUp 2022) ja 1.25 Mt L2-tasolla, joiden lisäksi kaikkien ytimien yhteistä L3-välimuistia on 25 Mt (Intel 2022b). Jos yhden toteutuksen tietyn vaiheen kaikki operaatiot mahtuvat välimuistiin, mutta toisen ei, dataa pitää siirrellä enemmän keskusmuistin ja prosessorien välimuistien välillä. Tuloksena voi olla huomattava suorituskykyero, sillä L3-tason välimuistilla ja erityisesti keskusmuistilla on paljon suurempi latenssi kuin L1- ja L2-tason välimuisteilla. Tällöin prosessori joutuu odottamaan suoritusohjeita.

6.2 Toteutuneet testiohjelmat

Toteutuneet testiohjelmat ovat puhtaalla Pythonilla kiihdytetty toteutus, sekä yhden ja monen säikeen Numba-versio. Näytönohjaimella kiihdytetty versio ei toteutunut. Ohjelmat ovat

kutsurajapinnoiltaan ja saaduilta tuloksiltaan samanlaiset, jos simulaatiokierrosbugi jätetään huomioimatta. Myös ohjelmien toteutukset vastaavat pitkälti toisiaan niin suoritusjärjestyksen, aliohjelmien kuin tietueiden suhteen. Lisänä Numba-versioissa on koodia Numban rajoitteiden kiertämiseen, joita käsittelen tarkemmin luvussa 6.3.

Simuloidut tulokset poikkeavat hieman toisistaan, sillä C käyttää erilaista satunnaislukugeneraattoria kuin Python-toteutukset.

Toteutin testiohjelmista ne osuudet, jotka vaadittiin mittausten ajamiseen. Tämä tarkoittaa sitä, että konfiguraatioiden numeroarvoja voi muuttaa, mutta esimerkiksi ERD-simulaatiosta RBS-simulaation vaihtaminen ei ole tuettuna. Loppujen ominaisuuksien toteuttaminen olisi suoraviivaista, mutta en toteuttanut niitä, koska en gradun puitteissa testaa niiden suorituspolkujen oikeellisuutta.

Testiohjelmien suurimmat puutteet ovat niihin jäänyt ohjelmointivirhe, joka kasvattaa simulointimääriä, sekä suoran GSTO-tuen puuttuminen. Kuten todettua, bugi pitkittää simulaation kestoa ja vääristää rekyyli-ionien määrää suhteessa pääioneihin. Muuten sillä ei pitäisi olla vaikutusta tuloksiin.

GSTO-tuen puuttuminen puolestaan velvoittaa käyttäjää itse generoimaan jarruuntumisenergiat jokaiselle konfiguraatiolle. Tämä on kohtuullisen vaarallista sen suhteen, että arvojen generointi voi unohtua helposti, jolloin ohjelma vääriä tuloksia. GSTO-ohjelmassa on komentorivikäyttöliittymä, joten sen kutsuminen Python-koodista olisi helppoa lisätä, mutta aiheuttaisi riippuvuuden erilliseen ohjelmaan, mikä vaatisi C-käännösympäristön asentamisen. Toinen vaihtoehto olisi toteuttaa Python-kielinen versio GSTO:sta, mikä olisi melko työlästä sen koon vuoksi.

6.3 Kohdatut haasteet

Kokeellisen teknologian käyttöön liittyy usein odottamattomia haasteita, eikä Numba ole poikkeus. Kuten suunnitteluvaiheessa arvioin, suurimpia toteutuneita riskejä olivat Numban rajalliset ominaisuudet, vähäinen dokumentaatio ja yleinen hiomattomuus. Kohtasin lisäksi Numban bugin, joka oli sen yllättävyyden ja Numban tavallisia kieliä heikomman debugat-

tavuuden vuoksi haastava ratkaista.

MCERDin kannalta Numban puutteellisin osa-alue ovat luokat. Natiivit Python-luokat eivät toimi Numbassa, vaan niille on korvikkeena kokeellinen Jitclass-toteutus (Anaconda 2022c). Valitettavasti Jitclass-luokat vaativat hidasta uudelleenkääntämistä niissä funktioissa, jotka ottavat vastaan tai palauttavat Jitclass-tyyppisiä muuttujia tavallisen Pythonin puolelta. Yhden Numban pääkehittäjän, Stuart Archibaldin (2021a), mukaan tämä johtuu luokkien ajokerroittain muuttuvista muistiosoitteista, jotka estävät käännettyjen funktioiden tallentamisen seuraavia ajokertoja varten. Lisäksi havaitsin syvästi sisäkkäisten, noin 3–4-kerroksisten, Jitclassien aiheuttavan epävakautta ohjelmaan, mikä näkyi satunnaisina kaatumisina. Näistä syistä jouduin korvaamaan Jitclassit NumPyn rakenteisilla taulukoilla, joiden alkioille pystyy määrittelemään C:n struct-tietorakenteen kaltaisen tietotyypin. Tämäkään ei toiminut suoraan, sillä Numba ei tue rakenteisten taulukkojen käyttöä sisäkkäin. Ratkaisuna tähän löysin toisen Numban pääkehittäjän, Graham Markallin (2021), kehittämän kokeellisen muutoksen Numban lähdekoodiin, joka lisää tuen sisäkkäisille, rakenteisille taulukoille. Otin tämän käyttöön nk. ”monkey patch” -muutoksella, jossa korvasin Numban lähdekoodia ajonaikaisesti omallani. Näin pystyin muokkaamaan Numban lähdekoodia ilman, että tarvitsisi paketoida täyttä Numban lähdekoodia ohjelmani rinnalle, vaan Numban voi edelleen asentaa PyPI-pakettivarastosta.

Tämä mahdollisti MCERDin objektien esittämisen varsin tehokkaasti, pl. havaitsemani tarve kääntää osa koodista uudelleen silloin kun objektien koko poikkeaa edellisestä suoituskerrasta. Ohjelman kääntäminen ja uudelleenkääntäminen vaikuttivat huomattavasti nopeammilta rakenteisilla taulukoilla kuin Jitclasseilla.

Toinen Numban puute MCERDin kannalta on tiedosto-operaatioiden täysi puute (Anaconda 2022f). Tämä on ongelmallista tietueiden alustamisen kannalta, sillä simulaatiokonfiguraatiot luetaan tiedostoista. Pystyin kiertämään puuttuvat tiedosto-operaatiot käyttämällä samaa koodia kuin puhtaalla Pythonilla. Uudelleenkäytetty koodi tuottaa Python-objekteja, joten ne eivät toimi Numbassa suoraan. Ratkaisin tämän puolestaan tekemällä muunnosfunktioita (tiedostoon *object_convert_jit.py*), jotka muuntavat sisäkkäiset objektit rekursiivisesti NumPyn taulukoiksi. Muunnos vaati kaikkien ei-triviaalien tietotyyppien määrittelemistä käsin, mikä onnistui lopulta melko helposti. Olisin voinut myös muokata alustuskoodin tuottamaan

suoraan NumPy-objekteja, mutta silloin en olisi voinut käyttää suoraan samaa koodia kuin perusversiossa, lisäksi vaihtelevan kokoisten objektien allokointi oikeankokoisina olisi ollut haastavaa.

Simulaation aikana MCERD puolestaan kirjoittaa simulaatiotapahtumia tiedostoihin, mikä vaatisi Numba-versiossa poistumista JIT-ympäristöstä jokaista kirjoittamista varten. Kirjoitustapahtumia on melko paljon, tulostiedostoina useiden megatavujen verran, joten puhtaan Pythonin puolella käyminen kirjoittamista varten ei olisi hyväksyttävää suoritusajan ja koodin suoraviivaisuuden osalta. Ratkaisuksi keksin kirjata simulaatiotapahtumat erillisiin NumPy-taulukoihin, joita kutsun C-tyylisesti puskureiksi. Arvot tallennetaan liukulukuina, jotta ne ovat mahdollisimman muistitehokkaita ja yhteensopivia Numban kanssa. Alkuperäisessä MCERDissä pitäisi kirjoittaa myös kokonaislukuja sekä erityisesti merkkejä, jotka eivät näyttäneet toimivan Numbassa. Siispä muunsin kaikki arvot liukuluvuiksi, merkkien tapauksessa niiden ASCII-arvon perusteella. Simulaation loputtua puskurien arvot muunnetaan takaisin niiden oikeaan tyyppiin ja kirjoitetaan muotoiltuna tiedostoihin.

Rakenteisiin NumPy-taulukoihin liittyy myös ohjelmani kannalta keskeinen bugi Numban roskienkeruussa: JIT-funktiossa luotu taulukko siivotaan alkioineen muistista kun sitä ei enää tarvita, vaikka alkioihin viitattaisiin edelleen (Archibald 2021b). Seurauksena esimerkiksi uuden objektin tuottava funktio palauttaakin satunnaisia arvoja. Koska yksittäisiä alkioita ei pystynyt alustamaan Numbassa sellaisenaan, jouduin ”kuorimaan” pelkän alkion taulukosta luonnin jälkeen. Loin monet simulaatiossa käytetyistä objekteista näin tavallisen Python-koodin puolella, mutta osa väliaikaisobjekteista luodaan vasta simulaatiokoodissa. Seurauksena ohjelma kaatui tai jäi jumiin satunnaisella simulaatiokierroksella, vaikka kaikkien syötteiden olisi pitänyt pysyä samana ja tällöin ohjelman toimia deterministisesti. Syyn löytäminen kesti kauan, sillä Numba-vika ei luonnollisestikaan ilmennyt JIT-kiihdytys kytkettynä pois päältä, enkä saanut Numba-yhteensopivaa debuggeria toimimaan. Roskienkeruubugi esiintyy ilman Numban taulukkokoodin muokkaamistakin, joten vika ei ole itseaiheutettu.

Sain aikaan myös omia bugeja MCERDin toimintalogiikkaan kopiointi- ja kirjoitusvirheillä, sekä Numban kaatumaan virheellisellä käytöllä, joka johtui ainakin osittain Numban puutteellisesta dokumentaatiosta. Tyypillinen toiminnallinen bugi ilmeni siten, että käytin väärän kentän arvoa tai laskuoperaattoria. Numban kaatuilut taas juontuivat siitä, että välillä oikean

käyttötavan selvittäminen vaati eri variaatioiden kokeilua kunnes jokin tapa ei kaatanut ohjelmaa. Dokumentaatio saattoi sisältää ohjeet peruskäyttöön, mutta edistyneempi käyttö jäi monesti arvailun varaan. Esimerkiksi Jitclassit ja tyypitetyt Numba-listat aiheuttivat ongelmia. Käytön opettelua eivät auttaneet kymmeniä tai satoja rivejä pitkät virheilmoitukset, jotka sisälsivät loppukäyttäjän kannalta liikaa tietoa Numban sisäisestä tilasta, jolloin virheen juurisyyn paikallistaminen kävi työlääksi.

Tutkielman loppuvaiheessa huomasin merkittävän ohjelmointivirheen syntyneiden rekyylien määrässä. Kuten aiemmin sanottu, en ehtinyt korjaamaan sitä.

Debuggeri olisi helpottanut ongelmien selvittämistä, mutta en saanut Numban tukemaa GDB-työkalua (Anaconda 2022i) toimimaan Windows-ympäristössäni. Sain GDB:n käynnistämään Numba-toteutuksen, mutta koodin suoritus ei pysähtynyt siellä asettamiini pysäytyspisteisiin. Kiersin GDB-ongelmaa käyttämällä Numban ”@objmode”-ominaisuutta, jolla tavallista Python-koodia voi kutsua JIT-funktiosta (Anaconda 2022b). Näin sain seurattua ohjelman tilaa PyCharm-ohjelmointympäristöni debuggerilla, mutta askeltaminen vaati kankeasti objmode-koodin lisäämistä jokaisen tarkasteltavan rivin yhteyteen. Lisäksi PyCharmin debuggeri ei osannut käsitellä rakenteisia taulukoita yhtä nopeasti kuin tavallisia NumPy-taulukoita, joten yhteen askeleeseen ja tarkasteltavien objektien latautumiseen saattoi kulua kymmenenkin sekuntia, jos kyseiset objektit edes latautuvat ennen jonkinlaista PyCharmin aikakatkaisua.

Ongelmien selvittämistä vaikeutti myös se, että Python- ja C-toteutukset käyttävät erilaisia satunnaislukugeneraattoreita, jolloin puhtaan Pythonin tai Numban tilaa ei voinut verrata suoraan C-ohjelmaan. Kiersin tätä generoimalla C:n satunnaislukugeneraattorilla lukuja taulukkoon, josta sitten luin niitä aidon satunnaislukugeneraattorin sijaan. Tämä mahdollista jonkinlaisen vertailtavuuden, mutta generoituja satunnaislukuja kuluu simulaation aika hyvin paljon, joten ne loppuivat suurestakin tiedostosta nopeasti kesken.

MCERDin rinnakkaistaminen vaati puolestaan säiekohtaisia muuttujia ionien, siroamistietojen ja globaalin tilan ylläpitämiseen. Numban automaattisesti rinnakkaistava ”prange”-silmukka ei tarjoa tällaista ominaisuutta (Anaconda 2022a), joten jouduin keksimään jonkin muun ratkaisun. Numban kehittäjille suunnattua dokumentaatiota selattuani löysin yksi-

tyisen ”get_thread_id”-funktion, joka on tarkoitettu suoritussäikeen indeksin selvittämiseen testikäyttöä varten (Anaconda 2022d). Loin säiekohtaisille muuttujille omat taulukkonsa, joista säiettä vastaavalla indeksillä sai noudettua vapaan muuttujan. Säikeiden osittainen hallinta varoitustekstein varustetulla apufunktiolla ei ole ohjelman pitkäikäisyyden ja kannettavuuden kannalta optimaalista, sillä Numban versiopäivitykset tai käyttöjärjestelmien erot voivat aiheuttaa bugeja, mutta koin tämän kompromissin hyväksyttäväksi. Tämän funktion käyttö voi myös olla syynä sille, että rinnakkaistettu toteutus ei skaalaudu niin hyvin kuin sen pitäisi. Koska ominaisuus on tarkoitettu debuggaamiseen, voi olla että sen toteutus ei rinnakkaistu hyvin ja täten muodostuu pullonkaulauksi.

Toinen vaihtoehto koodin rinnakkaistamisen olisi Pythonin omien kirjastojen käyttö, mutta Numban dokumentaatiossa varoitetaan mahdollisesta yhteensopimattomuudesta (Anaconda 2022h). Totesin tämän myös käytännössä, sillä *concurrent.futures*-kirjaston *ProcessPoolExecutor*-ominaisuutta kokeillessani päädyin vain tilanteeseen, jossa osa objekteista jäi ”lukkoon” JIT-koodissa, eikä niiden arvoja ei pystynyt muuttamaan.

Eräs todennäköisesti merkittävä puute MCERDin nopeuttamisen kannalta oli CUDA- ja JIT-koodin yhteensopimattomuus: CUDA-koodista voidaan kutsua yhteensopivia JIT-funktioita, mutta CUDA-koodin kutsuminen JIT-koodin puolelta suoraan osoittautui mahdottomaksi. Tavoitteenani oli siirtää simulaatiokoodin hitaimmat osat näytönohjaimelle, jolloin vain osa funktioista olisi tarvinnut uudelleenkirjoittaa CUDA-yhteensopiviksi, ja loput olisi voinut suorittaa prosessorilla. Tällöin koodia hidastavia suoritusympäristön vaihdoksia olisi kertynyt vain yksi suuntaansa. Todellisuudessa siirtymiä kertyisi kaksinkertainen määrä, sillä JIT-koodista pitäisi ensin palata tavalliseen Pythoniin, josta suoritus etenisi CUDA-puolelle. Optimoitua prosessorikoodia tehdessäni arvioin, että MCERD olisi mahdollistaa järjestää uudelleen sellaiseksi, että jokaisen simuloitavan ionin yhden kierroksen voisi suorittaa yhtenä joukkona näytönohjaimella. Simulaation konfiguraatiosta riippuen kierroksia kertyy noin tuhat, mikä olisi vielä siedettävä määrä siirtymiä, sillä MCERDin tyypillinen suoritusaika kestää vähintään minuutteja. Kokonaan Numban sisällä pysyttäessä olisi ainakin teoreettisesti mahdollista vaihtaa suoritusta laitteelta toiselle tehokkaasti, mutta kun JIT- ja CUDA-koodin väliset siirtymät tapahtuisivat tavallisen Pythonin kautta, suoritusaika tuskin lyhenisi puhtaaseen Numba-toteutukseen verrattuna merkittävästi.

Tämän ja aiempien alilukujen perusteella vastaus tutkimuskysymykseen 2, ”Miten valittu Python-nopeutusteknologia soveltuu MCERD-ohjelman toteuttamiseen verrattuna alkuperäiseen C-kieleen?”, on: Numba sopi suorituskykynsä puolesta MCERDin toteuttamiseen hyvin, mutta itse kehitystyö oli varsin kankeaa. Numba tuntui työläältä käyttää, koska kaikkia MCERDin tarvitsemia ominaisuuksia ei ollut vielä toteutettuna, ja saatavilla olevien ominaisuuksien dokumentaatio on paikoin puutteellista. Kun bugeja esiintyi JIT-koodissa, niitä oli hankala selvittää rajallisten debuggausmahdollisuuksien ja vaikeatulkintaisten virheviestien vuoksi. Arvioisin, että suurin osa toteuttamiseen käytetystä ajasta kului Numba-yhteensopivien toteutusratkaisujen etsimiseen sekä pääsääntöisesti omien bugien ratkaisemiseen, sillä kumpikaan ei ollut Numba-koodilla helppoa. Muilta osin koodin muuttaminen C-kielestä Pythoniin ja Numba-kiihdytetyksi sujui helposti.

6.4 Jatkotutkimuksen kohteet

MCERD sisältää runsaasti kustomointivaihtoehtoja ERD-/RBS-valinnasta, kohteesta, beamista ja simuloitavasta alkuaineesta alkaen, joten tarkasteltavia yhdistelmiä on paljon. Lisäksi testilaitteita ja tutkittavia toteutuksia oli monta. Suuren kombinaatiomäärän vuoksi ehdin tutkielmassani tarkastelemaan vain murto-osaa kiinnostavista yhdistelmistä. Uudet konfiguraatiot vaatisivat myös testiohjelmien jatkokehitystä, sillä tällä hetkellä toteutettuina ovat vain osa mahdollisista asetuksista.

Testattuja konfiguraatioita jäi puuttumaan erityisesti RBS:n ja kevyen alkuaineen osalta. Myös beamin energian ja ionin muuttaminen sekä erilainen kohde olisivat olennaisia testattavia tulosten kattavuuden ja täten yleiskuvan kannalta. Vertailtavuuden maksimoinnin kannalta asetuksia kannattaisi muuttaa yksi kerrallaan, jotta merkityksellisimmät asetukset löytyisivät selkeästi. Tällöin erilaisia yhdistelmiä tulisi hyvin paljon, joten ajan optimoimiseksi olisi perusteltua jättää ainakin kolmas suorituskerta ja säiemäärien skaalautumistestit pois.

Mittauksia kertyi jokaista toteutuksen, ajokerran ja laitteen yhdistelmää kohden vain yksi kappale. Tulosten luotettavuuden parantamiseksi mittauksia kannattaisi toistaa esimerkiksi kolmesti, jolloin niiden keskiarvo olisi vähemmän riippuvainen satunnaisesta heittelystä.

Esimerkiksi kannettavan tietokoneen C-version poikkeava nopeutuminen toisella kierroksella saattoi johtua satunnaisesta vaihtelusta. Ilman mittausten toistamista on vaikeaa sanoa, oliko poikkeaman syynä muutakin kuin tavallista vaihtelua.

Tutkielma on keskittynyt yhden alkuaineen simulointiin kerrallaan, mutta oikeassa käytössä tarkasteltavia alkuaineita on useita, jotka halutaan simuloida kerralla. Tämän varjolla myös optimaalisen simulaatiojärjestyksen selvittäminen olisi tärkeää: miten suoritusnopeus muuttuu, kun alkuaineet simuloidaan yhtä aikaa, osajoukko kerrallaan tai täysin sekventiaalisesti? Numban ajonaikainen JIT-optimointi ja sen levylle tallentaminen tuovat aiheeseen omat haasteensa, sillä tällä hetkellä ohjelma käännetään uudelleen erilaisille konfiguraatioille. Jos useita simulaatioita ajetaan yhtä aikaa eri konfiguraatoilla, mitä käännetyille tiedostoille tapahtuu? Jääkö vain esim. ensimmäinen tulos voimaan, vai syntyykö kilpatilanne (engl. *race condition*), jossa eri prosessit yrittävät ylikirjoittaa toistensa väliaikaistiedostoja? Väliaikaistiedostot tallentuvat samanlaisilla nimillä, joten jos Numba ei jotenkin lukitse niitä suorituksen ajaksi, kilpatilanne on todennäköinen. Jos kyseessä on kilpatilanne, yhtäaikaiset ohjelmat eivät välttämättä toimi oikein, jos ollenkaan.

Väliaikaistiedostojen uudelleenkääntämistä voisi tarkastella myös konfiguraatioiden kannalta: mitkä muutokset vaativat uudelleenkäämistä, mitkä eivät? Todennäköisesti vain objektien ja rakenteisten taulukoiden kokoa muuttavan asetukset aiheuttavat tarpeen kääntää niitä käyttävät funktiot uudelleen. Näitä asetuksia ovat esimerkiksi ERD-/RBS-tila ja simuloitavien ionien määrä.

Mittaukset keskittyivät Intel-prosessoreihin ja Windows-käyttöjärjestelmään, mutta muitakin merkittäviä, Numban tukemia alustoja on. Kiinnostavia laitteita olisivat erityisesti AMD:n perinteiset prosessorit sekä Applen ARM-pohjaiset prosessorit. Tarkasteltavia käyttöjärjestelmiä taas voisivat olla macOS- ja Linux-alustat. Erilaiset laitteet ja alustat saattavat käyttäytyä suorituskyvyn ja yhteensopivuuden suhteen eri tavoin.

Tutkielmaan jäi eräs tärkeä kysymys avoimeksi: miksi Intel i7-12700K ei hyödy MCERDissä säikeiden lisäämisestä juurikaan verrattuna muihin prosessoreihin? Kuten mainittua, vastausta kysymykseen on hankala selvittää Numban profilointituen puutteen vuoksi. Ongelmaa voisi lähestyä muokkaamalla lähdekoodia: muokkaamalla tai poistamalla funktioi-

ta käytöstä voi olla mahdollista löytää ongelmakohta. Haasteena tässä on se, että MCERD perustuu simulaatiosilmukkaan, jonka osat riippuvat toisistaan. Funktion poistaminen saattaa johtaa ikuiseen silmukkaan, jos esimerkiksi siirtymävaiheiden tarkistus puuttuu. Toinen mahdollinen lähestymistapa olisi muokata simulaatiosilmukka sellaiseksi, että jokainen sen funktioista suoritetaan kerran ionia kohden. Tällöin voisi olla mahdollista poistaa funktioita käytöstä yksitellen, ilman että ohjelma jää jumiin.

Python-kieliset MCERD-toteutukset tarvitsisivat jatkokehityksessä ainakin simulaatiosilmukkabugin korjauksen, tuen muille konfiguraatioille ja paremman GSTO-integraation. Bugin korjaamisen pitäisi olla melko suoraviivaista, kun debuggaa puhdasta Python -toteutusta käyttäen C:llä esigeneroituja satunnaislukuja. Bugi löytyy siitä kohtaa, missä Pythonin ja C:n arvot poikkeavat toisistaan. Kuten luvussa 6.2 on todettu, muiden konfiguraatioiden tukeminen on yksinkertaista, ja GSTO-integraation voi lisätä joko sen komentorivikäyttöliittymän kautta, tai toteuttamalla uudelleen.

Muita jatkokehityksen kohteita ovat suorituskykyoptimoinnit ja mahdollinen GPGPU-versio. Mahdollisiin optimointeihin kuuluu objektien alustaminen suoraan NumPy-muodossa, jolloin niitä ei tarvitsisi muuttaa toiseen muotoon kesken ohjelman. Tämä vähentäisi alustusvaiheen muistinkäyttöä huomattavasti, lyhentäisi suoritusaikaa hieman sekä karsisi ylläpidettävän koodin määrää. Toisenlainen optimointikeino olisi objektien suurempi yliallokointi, jotta niiden kokoa ei tarvitsisi muuttaa tyypillisillä simulaatioarvoilla ollenkaan. Tällöin koodia ei pitäisi tarvita kääntää uudelleen konfiguraation muuttuessa. GPGPU-versio vaatisi puolestaan simulaation järjestyksen muuttamista tai sitten täyttä toteuttamista näytönohjaimella, joten tehtävää on varsin paljon.

7 Yhteenveto

Tutkielmassa käytiin läpi ERD- ja RBS-menetelmiä simuloivan MCERD-ohjelman perusteita, tarkasteltiin erilaisia ohjelmointiteknologioita suorituskyvyn kannalta ja erityisesti tarkasteltiin Python-ohjelmakoodin nopeuttamista ja rinnakkaistamista.

Kokeellinen osio koostui C-kielisen MCERD-ohjelman suunnittelusta ja toteuttamisesta Python-kielisenä. Tämän jälkeen alkuperäisen ja uusien toteutusten suorituskykyä mitattiin suoritusajan, rinnakkaistumisen ja muistinkäytön suhteen kolmella testilaitteella. Tulokset myös analysoitiin. Lisäksi ohjelmien tuottamien simulaatiotulosten keskinäinen yhtenäisyys varmennettiin. Tuotetut ohjelmat ovat avointa lähdekoodia ja saatavilla osoitteesta [1]. Alkuperäinen ohjelma on saatavilla osoitteesta [2].

Havaittuja tutkimustuloksia olivat, että Numballa on mahdollista kiihdyttää MCERD-simulaatiolle Python-koodista suunnilleen yhtä nopea kuin C-koodi, minkä lisäksi rinnakkaistuminen onnistuu, mutta ei niin lineaarisesti ytimien mukaan skaalautuen kuin voisi odottaa. Numba-teknologian käytettävyys kehittämisen kannalta on puolestaan suorituskykyä heikommalla tasolla, mikä johtaa tarpeettomaan yritykseen-ja-erehdykseen, koodin laatua heikentäviin kiertoratkaisuihin sekä heikkoon debuggattavuuteen ja profiloitavuuteen.

Lisäksi havaittiin, että uusiin toteutuksiin jäi ohjelmointivirhe, joka aiheuttaa ylimääräisiä simulaatiosilmukoita ja täten tarpeettoman pitkiä suoritusaikoja. Ohjelmointivirhe ei kuitenkaan muuta tuloksia rekyloituneiden ionien liian suuren määrän lisäksi. Havaittiin myös, että MCERD-ohjelman siirtäminen Nvidia-näytönohjaimella laskettavaksi ei onnistu kohtuullisella työpanoksella, sillä se pitäisi toteuttaa simulaation osalta kokonaan CUDA-koodina, ja ohjelman suoritusjärjestystä pitäisi muuttaa merkittävästi.

Testiohjelmiin jäänyttä jatkokehitettävää ovat mm. simulaatiosilmukkabugin korjaaminen, muiden suorituspolkujen implementointi ja GSTO-tuen lisääminen. Jatkotutkimusta kannattaisi tehdä Intel i7-12700K-prosessorin heikon rinnakkaistuvuuden selvittämiseksi. Lisäksi erilaisten suoritusalustojen ja konfiguraatioiden suorituskykyä voisi tutkia.

1. `https://github.com/tpitkanen/numba_mcerd`
2. `https://github.com/JYU-IBA/mcerd`

Lähteet

"W1zzard". 2021. “Windows 10 & Intel Core i9-12900K Alder Lake Performance Review - E-Cores Causing Any Trouble? - Conclusion | TechPowerUp”. 13. marraskuuta. Viitattu 12. kesäkuuta 2022. https://www.techpowerup.com/review/alder-lake-windows-10-performance/8.html.

Alca Technology. 2021. “Thin Films in everyday life”. Viitattu 7. marraskuuta 2021. https://www.alcatechnology.com/en/blog/thin-films-in-everyday-life/.

Anaconda. 2018. “Numba: A High Performance Python Compiler”. Viitattu 18. lokakuuta 2020. https://numba.pydata.org/.

———. 2021. “Deprecation Notices — Numba 0.54.1+0.g39aef3deb.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 8. marraskuuta 2021. https://numba.readthedocs.io/en/0.54.1/reference/deprecation.html.

———. 2022a. “Automatic parallelization with @jit — Numba 0.55.2+0.g2298ad618.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 26. toukokuuta 2022. https://numba.readthedocs.io/en/stable/user/parallel.html.

———. 2022b. “Callback into the Python Interpreter from within JIT’ed code — Numba 0.55.2+0.g2298ad618.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 26. toukokuuta 2022. https://numba.readthedocs.io/en/stable/user/withobjmode.html.

———. 2022c. “Compiling Python classes with @jitclass — Numba 0.55.1+0.g76720bf88.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 18. maaliskuuta 2022. https://numba.readthedocs.io/en/stable/user/jitclass.html.

———. 2022d. “Notes on Numba’s threading implementation — Numba 0.55.2+0.g2298ad618.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 26. toukokuuta 2022. https://numba.readthedocs.io/en/stable/developer/threading_implementation.html.

Anaconda. 2022e. “Supported NumPy features — Numba 0.55.1+0.g76720bf88.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 22. maaliskuuta 2022. https://numba.readthedocs.io/en/stable/reference/numpysupported.html.

———. 2022f. “Supported Python features — Numba 0.55.1+0.g76720bf88.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 22. maaliskuuta 2022. https://numba.readthedocs.io/en/stable/reference/pysupported.html.

———. 2022g. “Supported Python features in CUDA Python — Numba 0.55.1+0.g76720bf88.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 22. maaliskuuta 2022. https://numba.readthedocs.io/en/stable/cuda/cudapysupported.html.

———. 2022h. “The Threading Layers — Numba 0.55.2+0.g2298ad618.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 26. toukokuuta 2022. https://numba.readthedocs.io/en/stable/user/threading-layer.html.

———. 2022i. “Troubleshooting and tips — Numba 0.55.2+0.g2298ad618.dirty-py3.7-linux-x86_64.egg documentation”. Viitattu 26. toukokuuta 2022. https://numba.readthedocs.io/en/stable/user/troubleshoot.html.

Archibald, Stuart. 2021a. “cache=True for jitclasses · Issue #4830 · numba/numba · GitHub”. 11. elokuuta. Viitattu 26. toukokuuta 2022. https://github.com/numba/numba/issues/4830#issuecomment-896819248.

———. 2021b. “Works as plain python, core dump as @njit · Issue #5853 · numba/numba · GitHub”. 5. elokuuta. Viitattu 26. toukokuuta 2022. https://github.com/numba/numba/issues/5853#issuecomment-893275330.

Arstila, K., J. Julin, M.I. Laitinen, J. Aalto, T. Konu, S. Kärkkäinen, S. Rahkonen ym. 2014. “Potku – New analysis software for heavy ion elastic recoil detection analysis”. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 331 (heinäkuu): 34–41. ISSN: 0168583X. doi:10.1016/j.nimb.2014.02.016.

Arstila, K, T Sajavaara ja J Keinonen. 2001. “Monte Carlo simulation of multiple and plural scattering in elastic recoil detection”. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 174, numero 1 (maaliskuu): 163–172. ISSN: 0168583X. doi:10.1016/S0168-583X(00)00435-3.

Assmann, W., H. Huber, Ch. Steinhausen, M. Dobler, H. Glückler ja A. Weidinger. 1994. “Elastic recoil detection analysis with heavy ions”. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 89, numero 1 (toukokuu): 131–139. ISSN: 0168583X, viitattu 7. huhtikuuta 2021. doi:10.1016/0168-583X(94)95159-4. https://linkinghub.elsevier.com/retrieve/pii/0168583X94951594.

Behnel, Stefan, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn ja Kurt Smith. 2011. “Cython: The Best of Both Worlds”. *Computing in Science & Engineering* 13, numero 2 (maaliskuu): 31–39. ISSN: 1521-9615, viitattu 21. lokakuuta 2021. doi:10.1109/MCSE.2010.118. http://ieeexplore.ieee.org/document/5582062/.

Behnel, Stefan, Robert Bradshaw, Lisandro Dalcín, Mark Florisson, Vitja Makarov ja Dag Sverre Seljebotn. 2021. “Cython: C-Extensions for Python”. Viitattu 21. lokakuuta 2021. https://cython.org/.

Bik, W M Arnold, ja F H P M Habraken. 1993. “Elastic recoil detection”. *Reports on Progress in Physics* 56, numero 7 (1. heinäkuuta): 859–902. ISSN: 0034-4885, 1361-6633, viitattu 18. toukokuuta 2021. doi:10.1088/0034-4885/56/7/002. https://iopscience.iop.org/article/10.1088/0034-4885/56/7/002.

CPUID. 2022. “CPU-Z | Softwares | CPUID”. Viitattu 5. kesäkuuta 2022. https://www.cpuid.com/softwares/cpu-z.html.

Danial, Al. 2020. “GitHub - AlDanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages.” Viitattu 7. huhtikuuta 2021. https://github.com/AlDanial/cloc.

Foundation, Python Software. 2022. “10. Brief Tour of the Standard Library — Python 3.10.5 documentation”. Viitattu 13. kesäkuuta 2022. https://docs.python.org/3/tutorial/stdlib.html.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser ym. 2020. “Array programming with NumPy”. *Nature* 585, numero 7825 (17. syyskuuta): 357–362. ISSN: 0028-0836, 1476-4687, viitattu 2. lokakuuta 2021. doi:10.1038/s41586-020-2649-2. https://www.nature.com/articles/s41586-020-2649-2.

HEAVY.AI. 2022. “What is GPGPU? Definition and FAQs | HEAVY.AI”. Viitattu 13. kesäkuuta 2022. https://www.heavy.ai/technical-glossary/gpgpu.

Hellborg, Ragnar, Harry J. Whitlow ja Yanwen Zhang, toimittaneet. 2010. *Ion Beams in Nanoscience and Technology*. Particle Acceleration and Detection. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-00622-7 978-3-642-00623-4, viitattu 13. toukokuuta 2021. doi:10.1007/978-3-642-00623-4. http://link.springer.com/10.1007/978-3-642-00623-4.

Hjelle, Geir. 2022. “Python Type Checking (Guide) – Real Python”. Viitattu 13. kesäkuuta 2022. https://realpython.com/python-type-checking/.

Intel. 2022a. “Intel Core i56200U Processor 3M Cache up to 2.80 GHz Product Specifications”. Viitattu 5. kesäkuuta 2022. https://ark.intel.com/content/www/us/en/ark/products/88193/intel-core-i56200u-processor-3m-cache-up-to-2-80-ghz.html.

———. 2022b. “Intel Core i712700K Processor 25M Cache up to 5.00 GHz Product Specifications”. Viitattu 4. toukokuuta 2022. https://ark.intel.com/content/www/us/en/ark/products/134594/intel-core-i712700k-processor-25m-cache-up-to-5-00-ghz.html.

———. 2022c. “Intel® Core™ i7-4930K Processor”. Viitattu 5. kesäkuuta 2022. https://www.intel.com/content/www/us/en/products/sku/77780/intel-core-i74930k-processor-12m-cache-up-to-3-90-ghz/specifications.html.

ISO/IEC. 2007. *ISO/IEC 9899:TC3,* 7. syyskuuta.

Julin, Jaakko. 2020. “GitHub - JYU-IBA/jibal: Jyväskylä Ion Beam Analysis Library (JIBAL)”. Viitattu 7. huhtikuuta 2021. https://github.com/JYU-IBA/jibal.

Klöckner, Andreas. 2021a. “Home - pycuda 2021.1 documentation”. Viitattu 23. elokuuta 2021. https://documen.tician.de/pycuda/.

———. 2021b. “pyopencl 2021.2.6 documentation”. Viitattu 23. elokuuta 2021. https://documen.tician.de/pyopencl/.

Klöckner, Andreas, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov ja Ahmed Fasih. 2012. “PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation”. *Parallel Computing* 38, numero 3 (maaliskuu): 157–174. ISSN: 01678191, viitattu 22. lokakuuta 2021. doi:10.1016/j.parco.2011.09.001. https://linkinghub.elsevier.com/retrieve/pii/S0167819111001281.

Laitinen, Mikko. 2013. “Improvement of time-of-flight spectrometer for elastic recoil detection analysis”. Tohtorinväitöskirja, Jyväskylän yliopisto.

Lam, Siu Kwan, Antoine Pitrou ja Stanley Seibert. 2015. “Numba: a LLVM-based Python JIT compiler”. Teoksessa *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM ’15,* 1–6. Austin, Texas: ACM Press. ISBN: 978-1-4503-4005-2, viitattu 23. syyskuuta 2021. doi:10.1145/2833157.2833162. http://dl.acm.org/citation.cfm?doid=2833157.2833162.

LLVM Developer Group. 2022. “The LLVM Compiler Infrastructure Project”. Viitattu 13. kesäkuuta 2022. https://llvm.org/.

Lutz, Mark. 2007. *Learning Python, 3rd Edition.* 1. marraskuuta. Viitattu 22. toukokuuta 2022. https://www.oreilly.com/library/view/learning-python-3rd/9780596513986/ch04.html.

Markall, Graham. 2021. “Structured dtypes with multi-dim fields: numba either throws or crashes · Issue #3158 · numba/numba · GitHub”. 7. joulukuuta. Viitattu 4. syyskuuta 2021. https://github.com/numba/numba/issues/3158#issuecomment-987826739.

Maxon. 2022. “Evaluate your computer’s hardware capabilities | Cinebench from Maxon”. Viitattu 5. kesäkuuta 2022. https://www.maxon.net/en/cinebench.

Microsoft. 2021. “Allocating Virtual Memory - Win32 apps | Microsoft Docs”. 1. heinäkuuta. Viitattu 22. toukokuuta 2022. https://docs.microsoft.com/en-us/windows/win32/memory/allocating-virtual-memory.

Millman, K. Jarrod, ja Michael Aivazis. 2011. “Python for Scientists and Engineers”. *Computing in Science & Engineering* 13, numero 2 (maaliskuu): 9–12. ISSN: 1521-9615, viitattu 18. elokuuta 2021. doi:10.1109/MCSE.2011.36. http://ieeexplore.ieee.org/document/5725235/.

NumPy Developers. 2022. “numpy.ndarray.nbytes — NumPy v1.22 Manual”. Viitattu 22. toukokuuta 2022. https://numpy.org/doc/stable/reference/generated/numpy.ndarray.nbytes.html.

Nvidia. 2021. “CUDA Python | NVIDIA Developer”. Viitattu 24. elokuuta 2021. https://developer.nvidia.com/cuda-python.

———. 2022. “Programming Guide :: CUDA Toolkit Documentation”. Toukokuu. Viitattu 13. kesäkuuta 2022. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

O’Neill, Melissa. 2018. “PCG, A Family of Better Random Number Generators | PCG, A Better Random Number Generator”. Viitattu 7. huhtikuuta 2021. https://www.pcg-random.org/index.html.

Okuta, Ryosuke, Yuya Unno, Daisuke Nishino, Shohei Hido ja Crissman Loomis. 2017. “CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations”. Teoksessa *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS),* 7.

Preferred Networks. 2020. “CuPy: A NumPy-compatible array library accelerated by CUDA”. Viitattu 18. lokakuuta 2020. https://cupy.dev/.

Preferred Networks. 2021a. “Comparison Table — CuPy 9.5.0 documentation”. Viitattu 6. marraskuuta 2021. https://docs.cupy.dev/en/stable/reference/comparison.html.

———. 2021b. “Installation — CuPy 9.5.0 documentation”. Viitattu 6. marraskuuta 2021. https://docs.cupy.dev/en/stable/install.html.

Python Software Foundation. 2021. “PyPI · The Python Package Index”. Viitattu 8. lokakuuta 2021. https://pypi.org/.

———. 2022a. “dataclasses — Data Classes — Python 3.8.13 documentation”. Viitattu 18. maaliskuuta 2022. https://docs.python.org/3.8/library/dataclasses.html.

———. 2022b. “GitHub - python/cpython: The Python programming language”. Viitattu 13. kesäkuuta 2022. https://github.com/python/cpython.

Ramalho, Luciano. 2015. *Fluent Python: clear, concise, and effective programming*. Second edition. O’Reilly Media. ISBN: 978-1-4920-5635-5.

Ritchie, Dennis M, Bell Labs ja Murray Hill. 1993. “The Development of the C Language”. *ACM Sigplan Notices* 28:201–208.

Rossum, Guido van. 2009. “The History of Python: Early Language Design and Development”. 3. helmikuuta. Viitattu 1. lokakuuta 2021. https://python-history.blogspot.com/2009/02/early-language-design-and-development.html.

TechPowerUp. 2022. “Intel Core i7-12700K Specs | TechPowerUp CPU Database”. Viitattu 4. toukokuuta 2022. https://www.techpowerup.com/cpu-specs/core-i7-12700k.c2507.

The PyPy Team. 2021. “PyPy”. Viitattu 7. huhtikuuta 2021. https://www.pypy.org/.

———. 2022. “PyPy - Features | PyPy”. Viitattu 13. kesäkuuta 2022. https://www.pypy.org/features.html.

Toss, Julio, ja Thierry Gautier. 2012. “A New Programming Paradigm for GPGPU”. Teoksessa *Euro-Par 2012 Parallel Processing,* toimittanut Christos Kaklamanis, Theodore Papatheodorou ja Paul G. Spirakis, 7484:895–907. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-32819-0 978-3-642-32820-6, viitattu 13. kesäkuuta 2022. doi:10.1007/978-3-642-32820-6_88.

Ung, Gordon. 2022. “Meet the new, tougher Cinebench R20 benchmark: We test it on Xeon and Threadripper”. Viitattu 5. kesäkuuta 2022. https://www.pcworld.com/article/403401/meet-the-new-tougher-cinebench-r20-benchmark-we-test-it-on-xeon-and-threadripper.html.

University of Jyväskylä. 2020a. “MCERD - Monte Carlo ERD”. Viitattu 17. lokakuuta 2020. https://github.com/JYU-IBA/MCERD.

———. 2020b. “Potku, analysis software for ToF-ERDA with integrated MCERD — Department of Physics”. Viitattu 17. lokakuuta 2020. https://www.jyu.fi/science/en/physics/research/infrastructures/accelerator-laboratory/pelletron/potku.

VanTol, Alexander. 2022. “Memory Management in Python – Real Python”. Viitattu 13. kesäkuuta 2022. https://realpython.com/python-memory-management/.

Warsaw, Barry. 2009. *PEP 3147 – PYC Repository Directories,* 16. joulukuuta. Viitattu 29. toukokuuta 2022. https://peps.python.org/pep-3147/.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski ym. 2020. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. *Nature Methods* 17, numero 3 (2. maaliskuuta): 261–272. ISSN: 1548-7091, 1548-7105, viitattu 24. elokuuta 2021. doi:10.1038/s41592-019-0686-2. http://www.nature.com/articles/s41592-019-0686-2.

Zhang, Xiao, Ani Li, Boyang Zhang, Wenjie Liu, Xiaonan Zhao ja Zhanhuai Li. 2016. “The Cost Performance of Hyper-Threading Technology in the Cloud Computing Systems”. Teoksessa *Advances in Swarm Intelligence,* toimittanut Ying Tan, Yuhui Shi ja Li Li, 9713:581–589. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing. ISBN: 978-3-319-41008-1 978-3-319-41009-8, viitattu 5. kesäkuuta 2022. doi:10.1007/978-3-319-41009-8_63. http://link.springer.com/10.1007/978-3-319-41009-8_63.