Lauri Sirkka

Samanaikaisuuden ja rinnakkaisuuden keinot ohjelmoinnissa

Tietotekniikan kandidaatintutkielma

26. huhtikuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Lauri Sirkka

Yhteystiedot: lauri.l.sirkka@student.jyu.fi

Ohjaaja: Jonne Itkonen

Työn nimi: Samanaikaisuuden ja rinnakkaisuuden keinot ohjelmoinnissa

Title in English: Concurrency and parallelism in programming

Työ: Kandidaatintutkielma

Opintosuunta: Informaatioteknologia

Sivumäärä: 28+6

Tiivistelmä: Kirjallisuuskatsauksessa tullaan näkemään erilaisia keinoja mahdollistaa ohjelmien samanaikaista ja rinnakkaista suoritusta. Erilaisia keinoja löydettiin kolme kappaletta. Jokaisesta keinosta käydään taustoja ja nähdään esimerkit kuinka niitä voitaisiin käyttää.

Avainsanat: ohjelmointi, samanaikaisuus, rinnakkaisuus, monisäikeistys, näytönohjaimet, asynkronisuus, reaktiivinen ohjelmointi

Abstract: In this literature review we will see different ways to accomplish concurrency and parallelism. There were three different ways found. We will see a review on the backgrounds of each of the different ways and see a short example on how that specific way could be used.

Keywords: programming, concurrency, parallelism, multi-threading, gpu, asynchronous, reactive programming

Kuviot

Sisällys

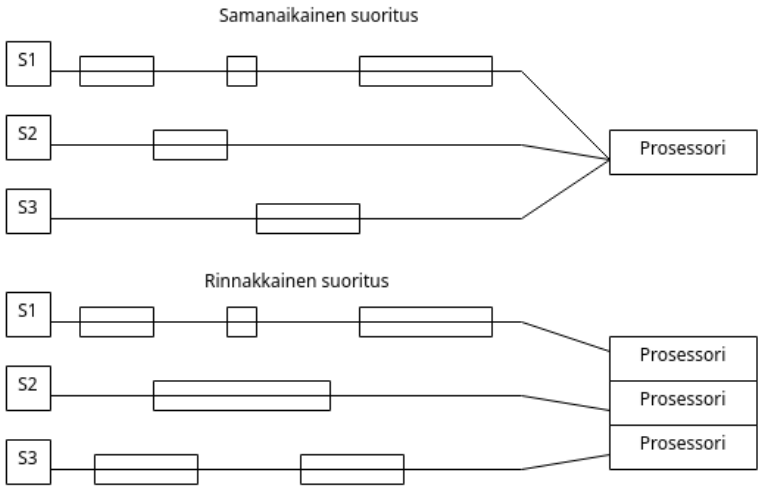
1 Johdanto

Yliopisto-opintojen aikana ohjelmien samanaikainen suorittaminen on jäänyt vähemmälle huomiolle. Aineopintojen projektikurssin aikana törmättiin ongelmiin. Miten käyttöliittymä saataisiin pysymään responsiivisena? Miten olisi mahdollista käyttöliittymää käyttäen yhdistää Android -laite toiseen laitteeseen Bluetooth yhteyden avulla. Kuinka kuunnella yhdistämisen jälkeen laitteen lähettämää dataa? Vastaus ongelmiin oli ohjelman samanaikaistaminen.

Suorituksessa olevaa ojelmaa kutsutaan prosessiksi. Prosessi voi koostua yhdestä tai useammasta *säikeestä*, jotka ovat eri käskyjä seuraavia suorituksen polkuja (Lewis ja Berg 1995, luku 2). Nämä säikeet suorittavat toisista säikeistä riippumattomasti ohjelmaa, mutta jakavat isäntäprosessin muistiavaruuden (Lewis ja Berg 1995, luku 2). Säikeiden suoritus voi tapahtua samanaikaisesti tai rinnakkain.

Enlanninkielessä on kaksi lähes samaa, mutta eri asiaa tarkoittavaa termiä. Suomenkielelle käännettäessä nämä tarkoittavat samaa asiaa, rinnakkaisuutta. Nämä termit ovat *concurrent* ja *parallel*. Ohjelmoinnissa *concurrent* tarkoittaa kahden tai useamman säikeen suorituksen olevan kesken yhtäaikaisesti, *parallel* taas tarkoittaa kahden tai useamman säikeen olevan *samalla ajanhetkellä* suorituksessa yhtä suurella määrällä suorittimia, kuin on säikeitä (Lewis ja Berg 1995, luku 2). *Concurrent* voi myös tarkoittaa monen eri prosessin suorituksen olevan kesken yhtäaikaisesti, mutta tätä tarkoitusta ei tässä asiayhteydessä tulla käyttämään. Erona näillä kahdella termillä onkin tapahtuuko laskenta samalla, vaiko eri ajanhetkellä, jota myös Kuvio 1 havainnollistaa esittäen vaakaviivoina ohjelmien tilaa eri ajanhetkillä, ja palkkeina ohjelmien olevan suorituksessa suorittimella. Näiden kahden termin eron selventämiseksi tullaan näitä kutsumaan suomenkielisillä vastineilla *samanaikaisuus* ja *rinnakkaisuus*.

Tässä kirjallisuuskatsauksessa tullaan näkemään erilaisia keinoja mahdollistaa ohjelmien samanaikaista ja rinnakkaista suoritusta. Tutkimuksen tarkoituksena on muodostaa lukijalle kuva ohjelmien samanaikaistamisen ja rinnakkaistamisen mahdollistavista keinoista, ja milloin mitäkin keinoa voidaan käyttää. Tutkielmassa ei mennä paljoa pintaa syvemmälle toteutuksiin, vaan tarkastellaan erilaisia samanaikaisuuden ja rinnakkaisuuden malleja kokonai-



Kuvio 1. Samanaikaisuus ja rinnakkaisuus, mukaillen Lewis ja Berg (1995, luku 2)

suuksina. Tutkielmassa ei myöskään käsitellä kaikkia samanaikaistuksen ja rinnakkaisuuden eri keinojen mukana tulevia ongelmia. Tutkielman luvuissa käsitellään kolmea eri keinoa toteuttaa samanaikaisuutta ja rinnakkaisuutta sekä nähdään pienoiset esimerkit näiden keinojen käytöstä ohjelmissa. Luku 2 käsittelee monisäikeistystä. Luku 3 käsittelee näytönohjaimia. Luku 4 käsittelee asynkronisuutta, sekä reaktiivista ohjelmointia. Lopuksi luku 5 muodostaa yhteenvedon.

2 Monisäikeistys

Monisäikeistäminen on yksi ohjelmointitapa, jolla voidaan toteuttaa ohjelman samanaikaistamista. Monisäikeistyksessä ohjelmaa suorittavan prosessin suoritus jaetaan pienemmiksi palasiksi, eli säikeiksi (Lewis ja Berg 1995, luku 2). Tämä ohjelman jakaminen riippumattomiksi palasiksi mahdollistaa useamman tehtävän samanaikaisen tai rinnakkaisen suorituksen riippuen siitä, että montako prosessoria laitteistolla on käytettävissä. Hyödyntämällä tietokoneiden moniytimisyyttä onkin mahdollista saada pidempään kestävää laskentaa tekevät ohjelmat suoriutumaan nopeammin tilanteissa, joissa tehtävät on mahdollista jakaa toisistaan riippumattomiin osiin.

Monisäikeistettyjen ohjelmien kanssa täytyy olla tarkkana, sillä niihin on helppo saada hankalasti havaittavia virheitä. Monisäikeistyksestä johtuneiden virheiden havaitsemiseksi on tehty tutkimusta yhtä pitkään, kuin on tutkittu monisäikeistämistäkin. Samanaikaisuuden virheiden havaitsemiseksi on kehitetty erilaisia lähdekoodin seurantaan liittyviä keinoja (Giebas ja Wojszczyk 2021). Algoritmeja on myös hankalaa toteuttaa hyödyntäen monisäikeistystä tehokkaasti (Chen ja Chen 2009).

2.1 Toteutus

Monisäikeistystä on mahdollista toteuttaa monilla eri kielillä hyödyntäen eri kirjastoja tai toteutuksia. Tässä kappaleessa nähdään kahden eri monisäikeistysmenetelmän käyttämistä ohjelmoinnissa. Kappaleessa nähdyt esimerkit ovat lähinnä monisäikeistystä havainnollistavia, eivätkä tarjoa käytännöllistä toimintaa. Esimerkit on toteutettu käyttäen C/C++ kielen Pthreads kirjastoa, ja OpenMP rajapintaa. Kumpikin ohjelma luo viisi säiettä, joista jokainen niin sanotusti *nukkuu* väliltä 1–5 sekuntia arvotun ajan verran tulostaen lopuksi standardi ulostuloon tekstiä.

2.1.1 Pthreads

`Pthreads` kirjasto on C/C++ kielen tarjoama toteutus POSIX säikeistä Unixin kaltaisilla käyttöjärjestelmillä. Tämän kirjaston käyttäminen on kuitenkin vaivalloista ja ohjelmoi-

ja joutuukin kirjoittamaan monta riviä koodia vain jakaakseen työt säikeille (Gonçalves ym. 2016). Kuhn, Petersen ja O'Toole (2000) päätyivät johtopäätökseen, että monisäikeistyksen toteutus on helpompaa käyttäen OpenMP rajapintaa, kuin Pthreads kirjastoa. He muuttivat alun perin `Pthreads` kirjastolla toimineen Genehunter sovelluksen rinnakkaistetun koodin toimimaan käyttäen OpenMP rajapintaa. Tämä kirjaston käytettävyysongelma havaitaan myös seuraavasta esimerkistä, jossa on toteutettu pienehkö kokeilu käyttäen `Pthreads` kirjastoa. Jokaiselle säikeelle täytyy antaa osoitin kyseiselle säikeelle parametrinä menevälle arvolle. Saman parametrien antamisen joutuisi tekemään, mikäli säikeiden tulisi palauttaa jotakin arvoja laskutuloksesta.

Esimerkissä on tehty C++ kielellä käyttäen apuna ja pohjana "Posix Threads" (2022) sivulta löytyvää ohjetta. Ohjelmassa luodaan viisi säiettä riveillä 22-25 käyttäen `Pthreads` kirjastoa, jonka jälkeen pääsäie jää odottamaan työskentelevien säikeiden valmistumista riveille 29-32. Tässä on hyvä huomata, että pääsäie odottaa säikeiden paluuta samassa järjestyksessä, kuin missä ne luotiinkin. Vaikka säie numero 3 valmistuisi ennen säiettä numero 0, ei pääsäie tulostaisi "Thread 3 finished" ennen kuin sitä edeltävät säikeet valmistuvat. Säikeet arpovat rivillä 10 arvon väliltä 1-5, jonka verran ohjelma nukkuu, ja tulostavat säikeelle parametrinä annetun numeron sekä arvotun arvon, jonka jälkeen suoritus palaa pääsäikeelle. Ohjelman kääntäminen vaatii Unixin kaltaisen käyttöjärjestelmän. Kääntäminen onnistuisi esimerkiksi käyttäen g++ kääntäjää seuraavasti: `g++ pthreads.cpp -pthread`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5  #define NUM_THREADS 5
6
7  void *print(void *ptr) {
8    int *num = (int *)ptr;
9    int sleep_time = 1 + rand() % NUM_THREADS;
10   printf("Running thread %d, sleeping for %d \n", *num, sleep_time);
11   sleep(sleep_time);
12   printf("Sleep finished %d\n", *num);
13   return NULL;
14 }
```

```
15
16 int main(void) {
17     pthread_t threads[NUM_THREADS];
18     int thread_args[NUM_THREADS];
19     int return_code;
20     printf("Creating threads\n");
21
22     for (int i = 0; i < NUM_THREADS; i++) {
23         thread_args[i] = i;
24         return_code = pthread_create(&threads[i], NULL, print, &thread_args[i]);
25     }
26
27     printf("Waiting for threads to finish\n");
28
29     for (int i = 0; i < NUM_THREADS; i++) {
30         pthread_join(threads[i], NULL);
31         printf("Thread %d finished\n", i);
32     }
33
34     printf("Threads finished\n");
35     return 0;
36 }
```

2.1.2 OpenMP

OpenMP on korkean tason ohjelmointirajapinta samanaikaisuuden tai rinnakkaisuuden toteuttamiselle. Ohjelmoijan tekemien kommentteja muistuttavien kääntäjän direktiivien määrittämissä lohkoissa haarautuu ajon aikana pääprosessista samanaikaisesti tai rinnakkain suoritettavia säikeitä (OpenMP 2021). OpenMP pyrkiikin luomaan helposti käytettävän keinon toteuttaa rinnakkaisuutta (Dagum ja Menon 1998). Kaikki OpenMP käskyt alkavat direktiivillä `#pragma omp`, jota seuraavat muut käskyt. Käyttäessään OpenMP:tä, ohjelmoijan on mahdollista antaa itse suoritettavien säikeiden määrä käyttäen komentoa `num_threads` tai antaa kääntäjän päättää montaako säiettä käytetään. Gonçalvesin ym. (2016) tekemässä tutkimuksessa havaittiin joillakin opiskelijoilla olleen hankaluuksia oppia käyttämään OpenMP rajapintaa ja tietää mitä kääntäjän käskyjä tulisi käyttää milloinkin, kun taas jotkut kokivat

sen helpommaksi käyttää kuin *Pthreads* kirjaston käytön.

Ohessa sama esimerkki, kuin *pthreads* esimerkissä, mutta nyt käyttäen OpenMP rajapintaa rinnakkaisuuden toteuttamiseksi. Kuten esimerkistä huomataan, tarvitaan vähemmän rivejä koodia `main` funktiossa saman toiminnallisuuden toteuttamiseksi. Ohjelman kääntäminen vaatii OpenMP kirjaston olevan asennettuna koneelle. Kääntäminen onnistuisi käyttäen esimerkiksi g++ kääntäjää seuraavasti: `g++ openmp.cpp -fopenmp`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #define NUM_THREADS 5
5
6  void *print(int *ptr) {
7    int *num = (int *)ptr;
8    int sleep_time = 1 + rand() % NUM_THREADS;
9    printf("Running thread %d, sleeping for %d \n", *num, sleep_time);
10   sleep(sleep_time);
11   printf("Sleep finished %d\n", *num);
12   return NULL;
13 }
14
15 int main(void) {
16   printf("Creating threads\n");
17
18   // Define parallel region using NUM_THREADS of threads
19   #pragma omp parallel num_threads(NUM_THREADS)
20   {
21     // Distribute for loop to given threads
22     #pragma omp for
23     for (int i = 0; i < NUM_THREADS; i++) {
24       print(&i);
25       printf("thread %d finished\n", i);
26     }
27   }
28
29   printf("Threads finished\n");
30   return 0;
```

```
31 }
```

2.2 Käyttökohteita

Androidilla ohjelman käyttöliittymän suoritus tapahtuu pääsäikeessä, jota kutsutaan myös käyttöliittymä säikeeksi (”Android Documentation Processes and threads overview” 2022). Näin ollen pitkään kestävän, eli blokkaavan operaation aikana käyttöliittymä lakkaisi vastaamasta. Tämä ei olisi käyttäjälle kovinkaan mielekästä. Lewis ja Berg (1995, luku 2) toteavat kirjassaan ettei blokkaavan osan ohjelmasta tarvitse pysäyttää koko ohjelman suoritusta. Androidin ohjelmistokehityspaketin dokumentaatiossa kehotetaankin tekemään tausta säie muille kuin välittömästi tapahtuville operaatioille, jotta käyttöliittymä pysyy responsiivisena (”Android Documentation Processes and threads overview” 2022). Monisäikeistystä käytetään myös verkkopalvelinohjelmissa. Esimerkiksi Apachen verkkopalvelin tarjoaa moniprosessointi moduulissaan (MPM) keinon käyttää monia prosesseja ja säikeitä käsittelemään tulevat verkkopyynnöt (”Apache HTTP Server Version 2.4 Multi-Processing Modules(MPMs)” 2021).

3 Näytönohjaimet

Alunperin näytönohjaimet valmistettiin pelejä ja grafiikkaa varten, ja ne onkin suunniteltu rinnakkaisiksi (Chen ja Chen 2009). Nykyiset halvimmatkin näytönohjaimet (GPU) sisältävät tuhansia rinnakaisytimiä (”GeForce RTX 3060 Family” 2022; ”AMD Radeon RX 6700 XT Graphics” 2022). Tätä laskentatehoa ei kuitenkaan ole mahdollista kaikissa ohjelmoinnin tilanteissa hyödyntää. Näytönohjaimet hyödyntävät *yksi käsky, monta säiettä* (SIMT) arkkitehtuuria, eli sama laskutoimitus suoritetaan eri datalla rinnakkain (Nvidia 2022). Tästä johtuen näytönohjaimet soveltuvat parhaiten operaatioille joissa on paljon dataa, joka käsitellään samalla tavalla riippumatta datasta. Näytönohjaimille on myös hankalaa toteuttaa tehokkaita algoritmeja (Chen ja Chen 2009).

Tässä kappaleessa tullaan tutustumaan kahteen eri ohjelmointiympäristöön jotka mahdollistavat laskennan suorittamisen näytönohjaimella. Nämä ympäristöt ovat *CUDA* ja *OpenMP*. Lisäksi nähdään esimerkkit, kuinka kumpaakin näistä voitaisiin käyttää. Molemmissa tutkittavissa ohjelmointiympäristöissä varsinainen näytönohjaimella suoritettava ohjelmakoodi muodustuu *ydinfunktioista* (kernel function). Nämä funktiot suoritetaan rinnakkkain näytönohjaimella vaihtaen vain arvoja, joille laskutoimitus tehdään. Muun ohjelman suoritus tapahtuu prosessorilla tavalliseen tapaan.

3.1 CUDA

Nvidian kehittämä ja vuonna 2006 julkaisema CUDA[1], on ohjelmointirajapinta ja alusta ohjelmien kirjoittamiseksi sekä suorittamiseksi näytönohjaimilla (Nvidia 2022). Sitä ei kuitenkaan ole mahdollista käyttää, kuin Nvidian näytönohjaimilla. CUDA mahdollistaa skaalautuvien massiivisesti rinnakkaistettujen ohjelmien kehittämisen tarjoten minimalistisen abstraktio tason rinnakkaistamiselle (Nvidia 2022).

Nvidian (2022) ohjeen mukaan CUDA ohjelmien kirjoittaminen onnistuu muun muassa C:llä, C++:lla ja Fortranilla. CUDA:n avulla ohjelman laskennallisesti raskaiden osien suorit-

1. Aiemmin käytetty nimeä Compute Unified Device Architecture, nykyisin käytetään vain lyhennettä CUDA

taminen voidaan toteuttaa käyttäen Nvidian näytönohjaimien prosessoria hyväksi. CUDA:n rajapinta koostuu matalan tason ajurirajapinnasta, sekä korkeammasta ajoympäristön rajapinnasta. CUDA ohjelmat koostuvat itse suorittimella suoritettavasta ohjelmasta sekä näytönohjaimella suoritettavista ydinfunktioista (Nvidia 2022). Nämä ydinfunktiot kutsutaan C++ kieltä käyttäen seuraavasti: `Nimi<<<B, N>>>`. Funktiot suoritetaan B*N määrällä säikeitä mahdollistaen skaalautuvuuden tarpeen mukaan.

Seuraavassa esimerkissä nähdään rivillä 5 ydinfunktio, jonka suoritus tapahtuu rinnakkain. Ohjelma on toteutettu käyttäen pohjana, ja apuna ”Cuda Vector Addition” (2022) tutoriaalia. Funktio summaa kahden taulukon alkiot yhteen, ja sijoittaa nämä kolmanteen. Ohjelmoijan täytyy manuaalisesti varata näytönohjaimesta tarvittava muisti sekä siirtää arvot näytönohjaimen muistiin ja takaisin. Nämä operaatiot tapahtuvat riveillä 21-25 ja 32. Ohjelman voisi kääntää seuraavasti: `nvcc cuda.cu`. Ohjelma vaatii CUDA työkalujen olevan asennettuna.

```
1  #include <stdio.h>
2  #include <chrono>
3
4  #define SIZE 100000
5  #define BLOCK_SIZE 1024
6  namespace ch = std::chrono;
7
8  __global__ void add(int *a, int *b, int *result, int size) {
9    // thread id
10   int i = blockIdx.x * blockDim.x + threadIdx.x;
11   if (i < size) {
12     result[i] = a[i] + b[i];
13   }
14 }
15
16 int main() {
17   size_t N = SIZE;
18   int a[N], int b[N], int result[N];
19
20   for (int i = 0; i < N; i++) {
21     a[i] = i;
```

```
22          b[i] = i;
23      }
24
25      int *cuda_a = nullptr;
26      int *cuda_b = nullptr;
27      int *cuda_result = nullptr;
28
29      printf("Copying to gpu\n");
30      cudaMalloc((void **)&cuda_a, sizeof(int) * N);
31      cudaMalloc((void **)&cuda_b, sizeof(int) * N);
32      cudaMalloc((void **)&cuda_result, sizeof(int) * N);
33      cudaMemcpy(cuda_a, a, sizeof(int) * N, cudaMemcpyHostToDevice);
34      cudaMemcpy(cuda_b, b, sizeof(int) * N, cudaMemcpyHostToDevice);
35
36      int gridSize = (int)ceil((float)N/BLOCK_SIZE);
37      printf("Starting parallel\n");
38      auto start = ch::steady_clock::now();
39      add<<<gridSize, BLOCK_SIZE>>>(cuda_a, cuda_b, cuda_result, N);
40
41      cudaDeviceSynchronize();
42      auto stop = ch::steady_clock::now();
43
44      printf("Getting results\n");
45      cudaMemcpy(result, cuda_result, sizeof(int) * N, cudaMemcpyDeviceToHost);
46
47      printf("Finished\n");
48      for (int i = 0; i < N; i++)
49      {
50          printf("%d, %d equal to %d\n", a[i], b[i], result[i]);
51      }
52
53
54      printf("Execution of kernel took %d microseconds\n",
55          ch::duration_cast<ch::microseconds>(stop - start).count()
56      );
57
58      cudaFree(cuda_a);
```

```
59      cudaFree(cuda_b);
60      cudaFree(cuda_result);
61      cudaDeviceReset();
62      return 0;
63  }
```

3.2 OpenCL

OpenCL on Khronos Groupin kehittämä viitekehys ja ajonaikainen järjestelmä, mikä mahdollistaa rinnakkaisohjelmien kirjoittamisen sekä ajamisen. Sekä AMD:n, että Nvidian näytönohjaimilla on mahdollista käyttää OpenCL viitekehystä. OpenCL ei eroa suoritustavastaan juurikaan CUDA:sta. OpenCL ohjelmat koostuvat suorittimella ajetusta ohjelmasta ja ydinfunktioista, jotka suoritetaan rinnakkain (”OpenCL Guide” 2022). Erona CUDA ohjelmaan, OpenCL etsii mitä rinnakkaissuorittimia tai muita suorittamiseen kykeneviä komponentteja laitteesta löytyy ja valitsee näistä löydetyistä laitteista sen, mitä se käyttää (”OpenCL Guide” 2022).

Käytettäessä OpenCL viitekehystä rinnakkaistamiseen, on mahdollista kääntää ydinfunktiot ajon aikaisesti tai etukäteen. Ydinfunktiot sijoitetaan omiin tiedostoihinsa, jotka ajon aikana ladataan ohjelmaan ja suoritetaan. Varsinainen rinnakkain suoritettava ydinfunktio muistuttaa CUDA:n vastaavaa. Seuraavassa esimerkissä nähdään sama ydinfunktio, kuin CUDA esimerkissä. Loput OpenCL:n toistuvasta koodista (boilerplate), joka lataa funktion ja suorittaa sen löytyy liitteestä A.

```
1  __kernel
2  void add(__global int *a, __global int *b, __global int *result) {
3          int i = get_global_id(0);
4          result[i] = a[i] + b[i];
5  }
```

3.3 Käyttökohteita

Näytönohjaimien samanaikaisesta arkkitehtuurista johtuen ne soveltuvat hyvin tehtäviin, jotka voidaan rinnastaa monille säikeille samanaikaisesti. Niitä ei kuitenkaan ole mahdollista käyttää kuin tilanteissa, joissa sama laskutoimitus voidaan suorittaa rinnakkain monilla säikeillä eri arvoilla. Muun muassa kuvankäsittelyyn liittyvän tutkimuksen toteuttaneet Zhang, Chen ja Wang (2010) havaitsivat näytönohjaimelle toteutettujen algoritmien olleen 25 ja 49 kertaa nopeampia, kuin vastaavat suorittimella ajetut. Myös ihonvärin tunnistukseen käytetty pikseleitä tarkasteleva algoritmi oli näytönohjaimilla toteutettuna nopeampi, kuin suorittimen vastaava (Ghorpade ja Thakare 2017). Kumpikin edeltävistä tutkimuksista vertaavat näytönohjaimella pyörivää rinnakkaista algoritmia vastaavaan suorittimella ajettuun sarjassa toimivaan algoritmiin. Hieman pienempiä nopeutuksia havaitsivat myös Catanzaro, Sundaram ja Keutzer (2008) vertaillessaan prosessorille optimisoitua rinnakkaistettua tukivektorikoneen koneoppimisalgoritmia näytönohjaimelle toteutettuun vastaavaan.

Su ym. (2012) havaitsivat testeissään CUDA:n ajurirajapinnan olevan vain 3,8–5,4 prosenttia nopeampi, kuin OpenCL:n ajoympäristön kun taas CUDA:n ajoympäristön rajapinta oli lähes yhtä nopea, kuin OpenCL:n vastaava. Asaduzzaman ym. (2021) taas havaitsivat CUDA:n olevan jopa seitsemän kertaa nopeampi joissain tilanteissa, mutta useimmissa tapauksissa vain kaksinkertainen. Omissa kokeiluissani tilanne oli päinvastainen, mutta tämä voi johtua ihan hyvin ohjelmointivirheistä.

4 Asynkronisuus

Asynkronisuudella tarkoitetaan sitä, että ohjelman suoritus ei etene samassa järjestyksessä, kuin miten se esiintyy lähdekoodissa. Ohjelmoija ei voi olla varma siitä, missä järjestyksessä ohjelman suoritus tapahtuu. Asynkronisessa ohjelmassa usea tehtävä on suorituksessa samanaikaisesti ja tehtävän valmistuessa siitä ilmoitetaan ohjelmalle, joka pääsee käsiksi tapahtuman tulokseen (Haverbeke 2018, luku 11). Asynkronisuutta hyödynnetään varsinkin yhdellä säikeellä suoritettavalla JavaScriptillä toteutetuissa verkkosovelluksissa tiedonsiirrossa (I/O) ja verkkopyynnöissä, jotka muuten voisivat kestää pitkään estäen ohjelman suorituksen (Madsen, Lhoták ja Tip 2017). Nämä verkkosovellukset muodostuvat yleisesti tapahtuman käsittelijöistä, jotka reagoivat asynkronisesti tapahtumankäsittelijään rekisteröityyn tapahtumaan. Verkkosovellus voisi esimerkiksi lähettää HTTP-pyynnön hakea jotain palvelimelta ja ohjelma jäisi kuuntelemaan palvelimelta vastausta tapahtumaan. Samanaikaisesti voisi ohjelma pysyä responsiivisena käyttäjälle ja reagoida palvelimelta saadun vastauksen saapuessa.

Ohessa JavaScriptillä toteutettu esimerkki tapahtumankuuntelijasta. Tapahtumankuuntelija lisätään HTML elementille, jonka id on *select*, rivillä 10. Kuuntelijalle annetaan parametrinä funktio, jota *takaisinkutsutaan* oikean tapahtuman tapahtuessa. Kuuntelija kuuntelee käyttäjän valitsevan HTML pudotusvalikosta `<select>` arvon tallentaen sen globaaliin muuttujaan `selectedValue`. Täysi ohjelma löytyy liitteestä B.

```
1  let selectedValue;
2
3  // Event listener
4  const eventHandler = (event) => {
5              selectedValue = event.target.value;
6  }
7
8  window.onload = () => {
9    const selectDropdown = document.getElementById("select");
10   selectDropdown.addEventListener("change", eventHandler);
11 }
```

4.1 Takaisinkutsut

Jotkin ohjelmointikielet mahdollistavat funktioiden käsittelemisen samalla tavalla kuin muutkin muuttujat. Tällöin funktiot ovat ensimmäisen luokan kansalaisia (Abelson, Sussman ja Sussman 1996, luku 1.3.4). Ensimmäisen luokan funktiot voidaan tallentaa muuttujaan tai tietorakenteeseen, palauttaa tuloksena tai antaa toiselle korkeamman asteen funktiolle parametrinä. JavaScriptissä on myös mahdollista antaa funktion takaisinkutsun parametriksi argumentin paikalle kirjoitettu anonyymi, eli nimetön funktio[1] (”MDN web docs: Functions” 2022). Tällöin funktiota ei voida kutsua muualta ohjelmasta, eikä kirjoitetut funktiot ole uudelleenkäytettäviä. Gallaba, Mesbah ja Beschastnikh (2015) havaitsivat lähdekoodeja tutkiessaan asynkronisten takaisinkutsujen esiintyvän useammin käyttöliittymien lähdekoodissa, kuin palvelimen koodissa.

Oheisessa esimerkissä funktiolle *reduce* annetaan argumenttina nimetön funktio, joka laskee kaksi arvoa yhteen palauttaen tuloksen. Esimerkin suoritus tapahtuu synkronisesti. Täysi ohjelma löytyy liitteestä B.

```
1 const values = [1, 2, 3, 4]
2 const result = values.reduce((previous, current) => {
3   return previous + current
4 })
```

Nimettömien funktioiden ongelmana on niiden kirjoittaminen sisäkkäin aiheuttaen sekavan ohjelmarakenteen, joka tunnetaan yleisesti *takaisinkutsuhelvettinä*. Ongelmalle on myös tehty samanniminen nettisivu, joka pyrkii ohjeistamaan, kuinka välttää tätä ohjelmarakennetta (”Callback hell” 2016). Helpoin keino välttää sekavaa ohjelmarakennetta onkin toteuttaa kaikki funktiot uudelleenkäytettävinä ja nimettyinä funktioina. Gallaba, Mesbah ja Beschastnikh (2015) tutkivat 136 ohjelmaa ja havaitsivat niiden viidessä miljoonassa koodirivissä olevan 43 prosenttia kaikille funktioille annetuista takaisinkutsujen argumenteista sisäkkäisiä.

1. Tunnetaan myös lambda funktiona https://docs.python.org/3/reference/expressions.html#lambda

4.2 Sitoumukset

JavaScriptin ES2015[2] standardin mukana on kieleen tullut eräs toinen tapa asynkronisuudelle, joka tunnetaan nimellä *promise*. Termin suora käännös Suomen kielelle olisi *lupaus*, mutta oikeastaan parempi käännös tuolle termille olisi *sitoumus*. Nimittäin *sitoumukset* ovat tietotyyppejä jotka *sitoutuvat* siihen, että tulevaisuudessa kyseisessä muuttujassa tulee olemaan asynkronisen operaation tulos (”MDN web docs: Promise” 2022). *Sitoumukset* ovat kertakäyttöisiä säiliöitä, joiden arvon voi asettaa vain kerran. Termiä *sitoumus* käyttivät ensimmäisenä Friedman ja Wise (1978).

Ohessa esimerkki *sitoumuksen* toiminnasta. Suoritus jatkuu HTTP-pyynnön vastauksen saapuessa tapahtuman käsittelijään. Oheisessa esimerkissä haetaan puhtaalla JavaScriptillä nettisivulta 10 kappaletta Shopify tuotteiden tietoja asynkronisesti. Avainsana `await` odottaa *sitoumuksen* ratkaisua ennen kuin funktion suoritus jatkuu asynkronisesti (”MDN web docs: await” 2022). Huomiona muun ohjelman toiminta jatkuu normaalisti funktion odottaessa vastausta palvelimelta. Täysi ohjelma löytyy liitteestä B.

```
1 const fetchContentFromWebsite = async () => {
2   const responseContent = await fetch(
3         "https://kbdfans.com/products.json?limit=10"
4   );
5   const resultBody = await responseContent.json();
6   return resultBody.products;
7 }
```

Sitoumuksen tila voi JavaScriptissä olla odottava, hylätty tai ratkaistu, sekä sen voi ratkaista vain kerran (”MDN web docs: Promise” 2022). Sitoumukset eivät kuitenkaan täysin ratkaise sisäkkäisten takaisinkutsujen luomaa ohjelmarakenteen ongelmaa vaan Kambona, Boix ja De Meuter (2013) havaitsivat toisistaan riippuvien lupausten johtavan samankaltaiseen sisäkkäiseen rakenteeseen. Lisäksi sitoumukset voivat johtaa lukuisiin ohjelmointivirheisiin, kuten kadonneeseen sitoumukseen, ennenaikaiseen paluuseen sitoumusta käsittelevästä funktiosta ja sitoumuksen ratkaisemiseen monta kertaa. Näitä ongelmia analysoivat Madsen, Lhoták ja Tip (2017) luoden pohjan sitoumuksien esittämiseksi kuvaajilla virheiden et-

2. Tunnetaan myös nimellä ES6, ECMAScript6

sinnän auttamiseksi. Alimadadi ym. (2018) laajensivat myöhemmin Madsenin, Lhotákin ja Tipin (2017) tutkimuksen pohjalta toteutuksen kattamaan kaikki ES2015 sitoumuksien toiminnot. He myös tutkivat kuinka avoimen lähdekoodin ohjelma `PromiseKeeper` kykenee tuottamaan kuvaajia sitoumuksista. Sitoumuksien piirtäminen virheiden etsinnän keinona ei kuitenkaan näyttäisi olevan kovin suosittua, ja ainoat toteutukset ovat jääneet tutkimuspohjaisiksi.

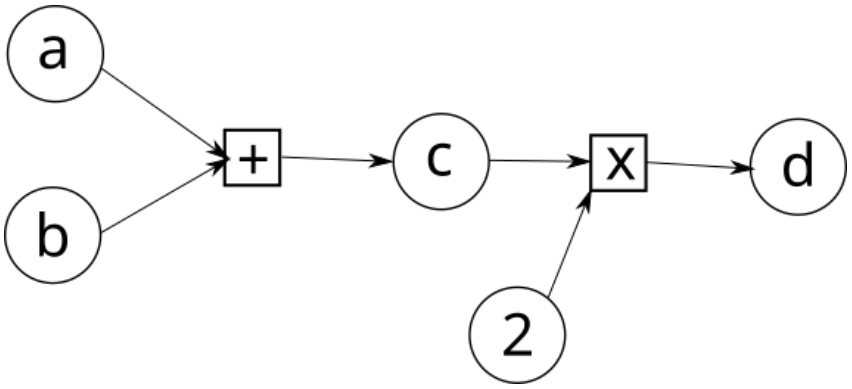
4.3 Reaktiivinen ohjelmointi

Reaktiivinen ohjelmointi on yksi ohjelmointimalli samanaikaisuuden toteuttamiseksi. Salvaneschi, Margara ja Tamburrelli (2015) kuvailevat *reaktiivisten* ohjelmien reagoivan tapahtuviin tapahtumiin, ja suorittavan jonkin laskennan tapahtuman perusteella. Tämän perusteella kaikki tapahtumia käsittelevät ohjelmat olisivatkin *reaktiivisia*. Tämän vuoksi tässä työssä reaktiivisuutta käytetään tarkoittamaan reaktiivisella ohjelmointimallilla toteutettua ohjelmaa. Perinteisesti *reaktiivisten* ohjelmien toteuttamiseen on käyttetty olio-ohjelmoinnin tarkkailijoita (observer) (Maier, Rompf ja Odersky 2010), sekä takaisinkutsuja (Bainomugisha ym. 2013). Reaktiivinen ohjelmointimalli on erotettu tässä työssä omaksi aliluvukseen, sillä se eroaa perinteisestä tapahtumien käsittelystä.

Reaktiivisten ohjelmien toteutukseen käytetään reaktiivisia kieliä tai laajennoksia jo olemassa olevaan kieleen. Reaktiivinen ohjelmointi on keino ilmaista ohjelma reaktioina tapahtumille (Bainomugisha ym. 2013). Reaktiivisessa ohjelmoinnissa ohjelman muuttujiin tapahtuvat muutokset vaikuttavat myös muihin siitä riippuviin muuttujiin. Usein esimerkkinä siitä, miten reaktiiviset ohjelmat toimivat, käytetään taulukkolaskentaohjelmia. Tämä onkin helpoin keino ymmärtää, mitä reaktiivinen ohjelmointi oikein on käytännössä. Samaan tapaan kun taulukon sarakkeessa olevaa arvoa muutetaan, muuttuvat myös siitä riippuvien sarakkeiden arvot. Ohjelmoijan tehdessä sijoitus `c = a + b`, muuttuisi myös muuttujan `c` arvo muuttujien `a` tai `b` arvon muuttuessa. Vastaavasti muuttujan `d` arvo sijoituksen `d = c * 2` jälkeen muuttuisi sen jälkeen, kun muuttujan `c` arvo on ensin muuttunut.

```
1 // Pseudocode
2 a = 1
3 b = 1
```

```
4 c = a + b // c –> 2
5 d = c * 2 // d –> 4
6 a = 3 // c –> 4, d –> 8
```



Kuvio 2. Esimerkki riippuvuusverkosta

Reaktiivinen ohjelmointi jakautuu kahteen eri ohjelmointimalliin, funktionaaliseen reaktiiviseen ohjelmointiin ja reaktiiviseen ohjelmointiin. Reaktiivisen ohjelmoinnin katsotaan olevan lähtöisin funktionaalisesta reaktiivisesta Fran ohjelmointikielestä, jonka kehittivät Elliott ja Hudak (1997). Funktionaalisen reaktiivisen mallin perusominaisuudet ovat jossain muodossa kaikissa reaktiivisissa kielissä (Bainomugisha ym. 2013).

Reaktiivinen ohjelmointi perustuu *tietovoihin* (data-flow), joita voidaan kuvata riippuvuusverkkona (dependency graph). Riippuvuusverkot ovat suunnattuja graafeja. Kuvio 2 havainnollistaa edellisen esimerkin pseudokoodin kulkua ja kuinka muuttujien arvot riippuvat toisistaan. Ohjelmoijan näkökulmasta muuttujien päivittyminen tapahtuu kuin itsestään (Bainomugisha ym. 2013). Muun muassa JavaScriptin päälle rakennetun reaktiivisen Flapjax-kielen toiminta perustuu riippuvuusverkkoihin, joita pitkin se työntää muutoksia (Meyerovich ym. 2009).

4.4 Käyttökohteita

Asynkronisuutta hyödynnetään sekä käyttöliittymän, että palvelimen puolella verkkosovelluksissa tilanteissa, joissa tarvitaan jotain sisältöä muista verkko-osoitteista, tiedonsiirrossa levyltä tai muuten operaatioissa, joiden täytyy odottaa tulostaan. Myös verkkopalveli-

mien toiminnoissa on käytetty asynkronisuutta ja esimerkiksi Nginx-verkkopalvelin käyttää asynkronista tapahtumankäsittelyä palvellessaan asiakkaita mahdollistaen tuhansien pyyntöjen käsittelyn samanaikaisesti yksisäikeisillä *työskentelijöillä* (Brown ja Wilson 2012, luku 14). Fan ja Wang (2015) havaitsivat tutkimuksessaan asynkronisen verkkopalvelimen myös kykenevän parempaan samanaikaiseen palveluun ja pienempään latenssiin johtuen monisäikeisen palvelimen pienemmästä mahdollistesta jonosta. Heidän mukaansa asynkronisesti toimivat palvelimet olisivatkin parempia pilvipohjaisille palveluille.

Reaktiivisen ohjelmoinnin nykyiset toteutukset ovat suurelta osin käyttöliittymiin liittyviä. Reaktiivista ohjelmointimallia onkin esitetty käytettäväksi takaisinkutsujen (Bainomugisha ym. 2013) ja olio-ohjelmoinnin tarkkailijoiden (Maier, Rompf ja Odersky 2010) sijaan tapahtumapohjaisten ohjelmien toteutukseen. Aiheesta on myös aiemmin tehty tutkimusta, ja verrattuna tarkkailijamalliin, reaktiivinen ohjelmointimalli teki ohjelmakoodista helpommin ymmärrettävää (Salvaneschi ym. 2017).

Muutamia esimerkkejä reaktiivisesta ohjelmoinnista ovat käyttöliittymien tekemiseen kehitetty Elm ohjelmointikieli käytten asynkronisuutta ja funktionaalista reaktiivista mallia (Czaplicki ja Chong 2013), sekä Flapjax ohjelmointikieli tai JavaScript laajennos käyttöliittymäpuolen verkkosivujen ohjelmointiin (Meyerovich ym. 2009). On hyvä huomata harhaan johtavasta nimestään huolimatta React.js kirjasto ei ole, eikä sen tekijät halua sen olevan täysin *reaktiivinen* (”React Design Principles” 2022). Reaktiivista ohjelmointia ja riippuvuuskaavioita on myös hyödynnetty Unity-pelimoottorin laajennoksen tekemisessä peliobjektien välisten riippuvuuksien hallinnalle ja objektien päivittämiseksi (Marum, Jones ja Cunningham 2020).

5 Yhteenveto

Kirjallisuuskatsauksessa on tarkasteltu kahden tutkimuskysymyksen johdattamana samanaikaisuuden käyttöä ohjelmoinnissa. Tutkielma antaa yleiskuvan samanaikaisuuden käyttämiseksi ohjelmoinnissa, mutta tiukalla aikataululla toteutetun työn ajanpuutteesta johtuen eri keinojen ongelmiin ei ehditty juurikaan perehtyä. Löydettyjä keinoja löydettiin kolme kappaletta, monisäikeistys, näytönohjaimien käyttäminen sekä asynkronisuus. Jokaisesta keinosta nähtiin selvitys, keinon toteuttamiseen käytettäviä työkaluja esitteltiin sekä nähtiin pieni esimerkki kyseisen keinon käyttötavasta.

Ohjelmoinnissa ei ole samanaikaisuuden ja rinnakkaisuuden toteuttamiseksi yhtä työkalua, jolla voisi toteuttaa kaiken. Kaikille keinoille on omat sovellusalueensa, joissa nämä keinot toimivat parhaiten. Monisäikeistys soveltuu irtonaisten tehtävien samanaikaistamiseen, tai pienien tehtävien rinnakkaistamiseksi. Kuitenkin asynkroniset tapahtumat olisivat samanaikaistusta varten monipuolisempia, kuin monisäikeistyksen keinot. Näytönohjaimet taasen soveltuvat parhaiten rinnakkaistettavien matemaattisten tehtävien ratkaisuun, jossa eri arvoille suoritetaan sama operaatio. Samoin ne soveltuvat hyvin suuren data määrän käsittelyyn massiivisen rinnakkaisuutensa takia. Asynkronisuus soveltuu parhaiten tapahtumien käsittelyyn, sillä ohjelman ei tällöin tarvitse olla jatkuvasti suorittamassa jotain, vaan reagoida tapahtumaan. Samoin sovellus voisi aloittaa jonkin hakuoperaation ja jatkaa suoritusta tuloksen saapuessa samalla kuin suoritettaisiin jotain muuta.

Tutkielmaa toteuttaessa varsinaisia tieteellisiä lähteitä sovelluskohteista ei juurikaan löytynyt. Tästä johtuen sovelluskohteita eri ohjelmointi keinoille voi olla monia muitakin, kuin tässä tutkimuksessa nähtyjä. Myös eri keinoja toteuttaa samanaikaisuutta ja rinnakkaisuutta voi olla, joita ei tässä suppeassa kirjallisuuskatsauksessa käsitelty.

Lähteet

Abelson, Harold, Gerald Jay Sussman ja Julie Sussman. 1996. *Structure and Interpretation of Computer Programs.* Viitattu 19. maaliskuuta 2022. https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html.

Alimadadi, Saba, Di Zhong, Magnus Madsen ja Frank Tip. 2018. ”Finding broken promises in asynchronous JavaScript programs”. *Proceedings of the ACM on Programming Languages* 2 (OOPSLA): 162:1–162:26. Viitattu 22. helmikuuta 2022. https://doi.org/10.1145/3276532.

”AMD Radeon RX 6700 XT Graphics”. 2022. Viitattu 6. maaliskuuta 2022. https://www.amd.com/en/products/graphics/amd-radeon-rx-6700-xt.

”Android Documentation Processes and threads overview”. 2022. Viitattu 20. helmikuuta 2022. https://developer.android.com/guide/components/processes-and-threads.

”Apache HTTP Server Version 2.4 Multi-Processing Modules(MPMs)”. 2021. Viitattu 3. maaliskuuta 2022. https://httpd.apache.org/docs/2.4/mpm.html.

Asaduzzaman, Abu, Alec Trent, S. Osborne, C. Aldershof ja Fadi N. Sibai. 2021. ”Impact of CUDA and OpenCL on Parallel and Distributed Computing”. Teoksessa *2021 8th International Conference on Electrical and Electronics Engineering (ICEEE),* 238–242. Viitattu 29. maaliskuuta 2022. https://doi.org/10.1109/ICEEE52452.2021.9415927.

Bainomugisha, Engineer, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx ja Wolfgang De Meuter. 2013. ”A survey on reactive programming”. *ACM Computing Surveys* 45 (4): 52:1–52:34. Viitattu 14. helmikuuta 2022. https://doi.org/10.1145/2501654.2501666.

Brown, Amy, ja Greg Wilson. 2012. *The Architecture of Open Source Applications, Volume II.* ISBN: 978-1105571817, viitattu 3. maaliskuuta 2022. https://www.aosabook.org/en/index.html.

”Callback hell”. 2016. Viitattu 8. maaliskuuta 2022. http://callbackhell.com.

Catanzaro, Bryan, Narayanan Sundaram ja Kurt Keutzer. 2008. ”Fast support vector machine training and classification on graphics processors”. Teoksessa *Proceedings of the 25th international conference on Machine learning - ICML '08,* 104–111. ACM Press. Viitattu 5. maaliskuuta 2022. https://doi.org/10.1145/1390156.1390170.

Chen, Trista P., ja Yen-Kuang Chen. 2009. ”Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs”. Teoksessa *2009 IEEE International Conference on Acoustics, Speech and Signal Processing,* 613–616. Viitattu 23. maaliskuuta 2022. https://doi.org/10.1109/ICASSP.2009.4959658.

”Cuda Vector Addition”. 2022. Viitattu 27. helmikuuta 2022. https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/.

Czaplicki, Evan, ja Stephen Chong. 2013. ”Asynchronous functional reactive programming for GUIs”. *ACM SIGPLAN Notices* 48 (6): 411–422. Viitattu 14. helmikuuta 2022. https://doi.org/10.1145/2499370.2462161.

Dagum, L., ja R. Menon. 1998. ”OpenMP: an industry standard API for shared-memory programming”. Conference Name: IEEE Computational Science and Engineering, *IEEE Computational Science and Engineering* 5 (1): 46–55. Viitattu 14. helmikuuta 2022. https://doi.org/10.1109/99.660313.

Elliott, Conal, ja Paul Hudak. 1997. ”Functional reactive animation”. Teoksessa *Proceedings of the second ACM SIGPLAN international conference on Functional programming,* 263–273. ICFP '97. New York, NY, USA: Association for Computing Machinery. Viitattu 22. helmikuuta 2022. https://doi.org/10.1145/258948.258973.

Fan, Qi, ja Qingyang Wang. 2015. ”Performance Comparison of Web Servers with Different Architectures: A Case Study Using High Concurrency Workload”. Teoksessa *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb),* 37–42. Viitattu 6. maaliskuuta 2022. https://doi.org/10.1109/HotWeb.2015.11.

Friedman ja Wise. 1978. ”Aspects of Applicative Programming for Parallel Processing”. Conference Name: IEEE Transactions on Computers, *IEEE Transactions on Computers* C-27 (4): 289–296. Viitattu 6. maaliskuuta 2022. https://doi.org/10.1109/TC.1978.1675100.

Gallaba, Keheliya, Ali Mesbah ja Ivan Beschastnikh. 2015. ”Don’t Call Us, We’ll Call You: Characterizing Callbacks in Javascript”. Teoksessa *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM),* 1–10. Viitattu 17. helmikuuta 2022. https://doi.org/10.1109/ESEM.2015.7321196.

”GeForce RTX 3060 Family”. 2022. Viitattu 6. maaliskuuta 2022. https://www.nvidia.com/en-gb/geforce/graphics-cards/30-series/rtx-3060-3060ti/.

Ghorpade, Deepali, ja Anuradha D. Thakare. 2017. ”Human Skin Colour Detection Algorithm Optimization with CUDA”. Teoksessa *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA),* 1–6. Viitattu 16. maaliskuuta 2022. https://doi.org/10.1109/ICCUBEA.2017.8464010.

Giebas, Damian, ja Rafał Wojszczyk. 2021. ”Detection of Concurrency Errors in Multithreaded Applications Based on Static Source Code Analysis”. *IEEE Access* 9:61298–61323. Viitattu 27. maaliskuuta 2022. https://doi.org/10.1109/ACCESS.2021.3073859.

Gonçalves, Rogério, Marcos Amaris, Thiago Okada, Pedro Bruel ja Alfredo Goldman. 2016. ”OpenMP is Not as Easy as It Appears”. Teoksessa *2016 49th Hawaii International Conference on System Sciences (HICSS),* 5742–5751. Viitattu 20. helmikuuta 2022. https://doi.org/10.1109/HICSS.2016.710.

Haverbeke, Marijin. 2018. *Eloquent Javascript, 3rd Edition: A Modern Introduction to Programming.* ISBN: 978-1593279509, viitattu 15. maaliskuuta 2022. https://eloquentjavascript.net/.

Kambona, Kennedy, Elisa Gonzalez Boix ja Wolfgang De Meuter. 2013. ”An evaluation of reactive programming and promises for structuring collaborative web applications”. Teoksessa *Proceedings of the 7th Workshop on Dynamic Languages and Applications - DYLA ’13,* 1–9. ACM Press. Viitattu 9. helmikuuta 2022. https://doi.org/10.1145/2489798.2489802.

Kuhn, Bob, Paul Petersen ja Eamonn O’Toole. 2000. ”OpenMP versus threading in C/C++”. *Concurrency and Computation: Practice and Experience* 12 (12): 1165–1176. Viitattu 4. maaliskuuta 2022. https://doi.org/10.1002/1096-9128(200010)12:12<1165::AID-CPE529>3.0.CO;2-L.

Lewis, Bil, ja Daniel J. Berg. 1995. *Threads primer: a guide to multithreaded programming*. USA: Prentice Hall Press. ISBN: 978-0-13-443698-2.

Madsen, Magnus, Ondřej Lhoták ja Frank Tip. 2017. ”A model for reasoning about JavaScript promises”. *Proceedings of the ACM on Programming Languages* 1 (OOPSLA): 1–24. Viitattu 21. maaliskuuta 2022. https://doi.org/10.1145/3133910.

Maier, Ingo, Tiark Rompf ja Martin Odersky. 2010. ”Deprecating the Observer Pattern”, 18. Viitattu 25. maaliskuuta 2022. http://infoscience.epfl.ch/record/148043.

Marum, João Paulo Oliveira, J. Adam Jones ja H. Conrad Cunningham. 2020. ”Dependency Graph-based Reactivity for Virtual Environments”. Teoksessa *2020 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW),* 246–253. Viitattu 22. helmikuuta 2022. https://doi.org/10.1109/VRW50115.2020.00052.

”MDN web docs: await”. 2022. Viitattu 8. maaliskuuta 2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await.

”MDN web docs: Functions”. 2022. Viitattu 17. maaliskuuta 2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions.

”MDN web docs: Promise”. 2022. Viitattu 17. maaliskuuta 2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

Meyerovich, Leo A., Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield ja Shriram Krishnamurthi. 2009. ”Flapjax: a programming language for Ajax applications”. *ACM SIGPLAN Notices* 44 (10): 1–20. Viitattu 19. helmikuuta 2022. https://doi.org/10.1145/1639949.1640091.

Nvidia. 2022. ”CUDA C++ Programming Guide V. 11.3”. Viitattu 17. helmikuuta 2022. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

”OpenCL Guide”. 2022. Viitattu 6. maaliskuuta 2022. https://github.com/KhronosGroup/OpenCL-Guide.

OpenMP. 2021. ”OpenMP API Specification 5.2”. Viitattu 21. helmikuuta 2022. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

”Posix Threads”. 2022. Viitattu 28. helmikuuta 2022. http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html.

”React Design Principles”. 2022. Viitattu 27. maaliskuuta 2022. https://reactjs.org/docs/design-principles.html#scheduling.

Salvaneschi, Guido, Alessandro Margara ja Giordano Tamburrelli. 2015. ”Reactive Programming: A Walkthrough”. Teoksessa *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* 2:953–954. Viitattu 26. maaliskuuta 2022. https://doi.org/10.1109/ICSE.2015.303.

Salvaneschi, Guido, Sebastian Proksch, Sven Amann, Sarah Nadi ja Mira Mezini. 2017. ”On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study”. *IEEE Transactions on Software Engineering* 43 (12): 1125–1143. https://doi.org/10.1109/TSE.2017.2655524.

Smistad, Erik. 2018. ”Getting started with OpenCL and GPU Computing”. Viitattu 6. maaliskuuta 2022. https://www.eriksmistad.no/getting-started-with-opencl-and-gpu-computing/.

Su, Ching-Lung, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang ja Kuo-Hsuan Wu. 2012. ”Overview and comparison of OpenCL and CUDA technology for GPGPU”. Teoksessa *2012 IEEE Asia Pacific Conference on Circuits and Systems,* 448–451. Viitattu 6. maaliskuuta 2022. https://doi.org/10.1109/APCCAS.2012.6419068.

Zhang, Nan, Yun-shan Chen ja Jian-li Wang. 2010. ”Image parallel processing based on GPU”. Teoksessa *2010 2nd International Conference on Advanced Computer Control,* 3:367–370. Viitattu 21. helmikuuta 2022. https://doi.org/10.1109/ICACC.2010.5486836.

Liitteet

A OpenCL taulukoiden summaus pääohjelma

Ohjelman kääntäminen vaatii OpenCL kirjaston asentamista koneelle. Ohjelma kääntyisi esimerkiksi käyttäen `g++ opencl.cpp -l OpenCL`. Ydinfunktion tulee olla kovakoodatussa tiedostossa 'opencl.cl' samassa tiedostopolussa, kuin itse pääohjelman. Ohjelma on toteutettu käyttäen apuna ja pohjana Smistad (2018) ohjetta.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <CL/cl.h>
4  #include <chrono>
5
6  #define SIZE 10
7  #define MAX_SOURCE_SIZE (0x100000)
8  #define KERNEL_FILE "./opencl.cl"
9
10 namespace ch = std::chrono;
11
12 int main() {
13    const int a[SIZE] = {2, 93, 58, 1, 49, 38, 29, 32, 5, 84};
14    const int b[SIZE] = {7, 3, 48, 83, 29, 30, 59, 8, 72, 50};
15    int result[SIZE];
16
17    printf("Loading kernel file\n");
18    // Load kernel file
19    FILE *fp = fopen(KERNEL_FILE, "r");
20    if (!fp)
21    {
22       printf("Can't load kernel, exiting\n");
23       exit(1);
24    }
25    char *source_str = (char *)malloc(MAX_SOURCE_SIZE);
26    size_t source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
27    fclose(fp);
```

```
28
29      // Platform and device info
30      cl_platform_id platform_id = NULL;
31      cl_device_id device_id = NULL;
32      cl_uint ret_num_devices;
33      cl_uint ret_num_platforms;
34      cl_int err_ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
35      err_ret = clGetDeviceIDs(
36          platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &ret_num_devices
37      );
38
39      // Create an OpenCL context
40      cl_context context = clCreateContext(
41          NULL, 1, &device_id, NULL, NULL, &err_ret
42      );
43
44      // Create a command queue
45      cl_command_queue command_queue = clCreateCommandQueueWithProperties(
46          context, device_id, 0, &err_ret
47      );
48
49      printf("Reserving space and copying to gpu\n");
50      // Create memory buffers on the device
51      cl_mem cl_a = clCreateBuffer(
52          context, CL_MEM_READ_ONLY, SIZE * sizeof(int), NULL, &err_ret
53      );
54      cl_mem cl_b = clCreateBuffer(
55          context, CL_MEM_READ_ONLY, SIZE * sizeof(int), NULL, &err_ret
56      );
57      cl_mem cl_result = clCreateBuffer(
58          context, CL_MEM_WRITE_ONLY, SIZE * sizeof(int), NULL, &err_ret
59      );
60
61      // Copy the arrays to memory
62      err_ret = clEnqueueWriteBuffer(
63          command_queue, cl_a, CL_TRUE, 0, SIZE * sizeof(int), a, 0, NULL, NULL
64      );
```

```
65      err_ret = clEnqueueWriteBuffer(
66          command_queue, cl_b, CL_TRUE, 0, SIZE * sizeof(int), b, 0, NULL, NULL
67      );
68
69      printf("Creating program\n");
70      // Create a program from kernel
71      cl_program program = clCreateProgramWithSource(
72          context, 1, (const char **)&source_str, &source_size, &err_ret
73      );
74
75      // Build the program
76      err_ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
77
78      // Create the OpenCL kernel
79      cl_kernel kernel = clCreateKernel(program, "add", &err_ret);
80
81      // Set the arguments of the kernel
82      err_ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&cl_a);
83      err_ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&cl_b);
84      err_ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&cl_result);
85
86      printf("Starting in paralel\n");
87      // Execute the OpenCL kernel on the list
88      size_t global_item_size = SIZE;
89      size_t local_item_size = 2;
90      auto start = ch::steady_clock::now();
91      err_ret = clEnqueueNDRangeKernel(
92          command_queue, kernel, 1, NULL, &global_item_size,
93          &local_item_size, 0, NULL, NULL
94      );
95
96      // Wait until queue is empty
97      clFinish(command_queue);
98      auto stop = ch::steady_clock::now();
99      printf("Execution took %d microseconds\n",
100         ch::duration_cast<ch::microseconds>(stop - start).count()
101     );
```

```
102
103     printf("Getting results\n");
104     // Read the memory buffer for results on the device to the local variable result
105     err_ret = clEnqueueReadBuffer(
106         command_queue, cl_result, CL_TRUE, 0, SIZE * sizeof(int),
107         result, 0, NULL, NULL
108     );
109
110     // Display the result to the screen
111     for (int i = 0; i < SIZE; i++)
112         printf("%d + %d = %d\n", a[i], b[i], result[i]);
113
114     // Clean up
115     err_ret = clFlush(command_queue);
116     err_ret = clFinish(command_queue);
117     err_ret = clReleaseKernel(kernel);
118     err_ret = clReleaseProgram(program);
119     err_ret = clReleaseMemObject(cl_a);
120     err_ret = clReleaseMemObject(cl_b);
121     err_ret = clReleaseMemObject(cl_result);
122     err_ret = clReleaseCommandQueue(command_queue);
123     err_ret = clReleaseContext(context);
124
125     return 0;
126 }
```

B Javascript esimerkki sivu

HTML tiedosto. Tallentamalla alempaa löytyvän JavaScript koodin tiedostoon `script.js` ja HTML koodin samaan kansioon, voi nettisivun avata aukaisemalla aukaisemalla HTML tiedoston selaimessa.

```
1 <!DOCTYPE html>
2 <head>
3     <title>Kandityön esimerkkejä</title>
4     <script src="script.js"></script>
5 </head>
```

```
<body>
  <select id="select">
    <option selected>Valinta 1</option>
    <option>Jotain sisältöä</option>
    <option>Aku Ankka</option>
  </select>
  <p id="content">Vaihtuva sisältö tulee tähän</p>
  <button id="refresh">Päivitä valittu arvo</button>
  <p id="reduce">callback</p>
  <p>Alla olevat tuotteet ladattu asynkronisesti</p>
</body>
```

JavaScriptin lähdekoodi

```
let selectedValue;

// Event listener
const eventHandler = (event) => {
  selectedValue = event.target.value;
}

const refresh = (event) => {
  const paragraph = document.getElementById("content");
  paragraph.textContent = selectedValue
}

const fetchContentFromWebsite = async () => {
  const responseContent = await fetch(
      "https://kbdfans.com/products.json?limit=10"
  );
  const resultBody = await responseContent.json();
  // Returns json content from found products
  return resultBody.products;
}

const showProducts = async () => {
  let products = await fetchContentFromWebsite();
  const containerOfProducts = document.createElement("div");
```

```
25
26      // Add found products to the page
27      for (let i = 0; i < products.length; i++) {
28          const textElement = document.createElement("p");
29          textElement.textContent = products[i].title;
30          containerOfProducts.appendChild(textElement);
31      };
32      document.body.append(containerOfProducts);
33  }
34
35  const showReduceCallback = () => {
36      let paragraph = document.getElementById("reduce");
37      const values = [1, 2, 3, 4];
38      const result = values.reduce((previous, current) => {
39          return previous + current;
40      });
41      paragraph.textContent = ‘Calculation using anonymous callback
42          with values ${values} equal to ${result}‘;
43  };
44
45  window.onload = () => {
46      const selectDropdown = document.getElementById("select");
47      selectDropdown.addEventListener("change", eventHandler);
48
49      // Default selected value set on start
50      selectedValue = selectDropdown.value;
51      const refreshButton = document.getElementById("refresh");
52      refreshButton.addEventListener("click", refresh);
53
54      showProducts();
55      showReduceCallback();
56  }
```