Opinnäytetyö (AMK)

Tietotekniikka

Sulautetut ohjelmistot

2016

Kalle Väärikkälä

KUVAPROSESSOINNIN OPTIMOINTI KÄYTTÄEN CUDA-ARKKITEHTUURIA

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Sulautetut ohjelmistot

2016 | 27 sivua

Kalle Väärikkälä

KUVAPROSESSOINNIN OPTIMOINTI KÄYTTÄEN CUDA-ARKKITEHTUURIA

Tässä opinnäytetyössä oli tavoitteena tutkia GPU-ohjelmoinnin hyödyllisyyttä ja sen tuomaa tehokkuutta yleishyödylliseen ohjelmointiin Työssä käytiin läpi GPU-ohjelmoinnin teoriaa ja erityisesti CUDA-arkkitehtuuri ja siinä käytettävät ohjelmointitekniikat. Tavoitteena oli myös tutkia, onko kannattavaa ottaa CUDA-arkkitehtuuria käyttöön Ksenos VMS -ohjelmistoon. Ksenos-ohjelmisto on videovalvontaohjelmisto, jolla voi hallita suuriakin määriä kameroita.

Työn teoriaosuutta varten tutkittiin CUDA-arkkitehtuurin tuomia tekniikoita ja niiden käyttöä. Työssä tutkittiin myös GPU:n ja CPU:n välisiä eroja arkkitehtuurissä sekä käyttötarkoituksissa. Tutkimuksen pohjalta kehitettiin testiohjelma mittaamaan ajoaikoja GPU:n ja CPU:n välillä käyttäen CUDAa.

Opinnäytetyön tuloksena saatiin selkeä käsitys CUDAn käytöstä yleishyödyllisessä ohjelmoinnissa. Toteutetuista ohjelmista saatiin ajoaikoja GPU:n ja CPU:n välillä, jotka vaikuttavat Ksenos-yrityksen päätökseen jatkokehittää CUDAa ohjelmistossaan.

ASIASANAT:

GPGPU, CUDA, kuvaprosessointi

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information technology | Embedded systems

2016 | 27 pages

Kalle Väärikkälä

OPTIMIZING IMAGE PROCESSING USING THE CUDA ARCHITECTURE

The aim of this thesis was to study the usefulness of GPU programming and its effectiveness in general purpose computing. The thesis goes through the theory of GPU programming and especially the CUDA architecture and programming techniques which come with it. The aim was also to examine if it was worthwhile for Ksenos to implement CUDA in their program. Ksenos VMS is a video management software that can manage large amount of cameras.

For the theoretical part of the thesis the CUDA architecture was examined and all the techniques that come with it as well as their usage. Also in the theoretical part the differences in architectural and general usage of CPU and GPU was examined.

As a result of the thesis a clear understanding of CUDA was obtained and of the usefulness of it in general purpose programming. From the produced programs comparisons could be made in runtimes between the CPU and GPU.

KEYWORDS:

GPGPU, CUDA, image processing

SISÄLTÖ

KÄYTETYT LYHENTEET JA SANASTO **6**

1 JOHDANTO **7**

2 GPU-OHJELMOINTI **8**

2.1 GPU:n ja CPU:n erot 9

2.2 GPU:n tekniikka 11

3 CUDA **13**

3.1 Kernelit 14

3.2 Säiehierarkia 15

3.3 Muistihierarkia 16

3.4 Isäntä ja laite 17

4 KUVAPROSSOINNIN VERTAILU GPU:N JA CPU:N VÄLILLÄ **19**

4.1 CUDAn edut 19

4.2 Kuvadatan käsittely 20

4.2.1 CPU ja kuvadata 20

4.2.2 GPU ja kuvadata 21

4.3 Kuvien skaalaus 22

4.3.1 CPU-implementaatio 22

4.3.2 CUDA-implementaatio 23

4.3.3 Vertailu 24

5 YHTEENVETO **26**

LÄHTEET **27**

LIITTEET

Liite 1. Kuvan skaalauksen alustus CUDAlla.
Liite 2. Mustavalkokuvan skaalaus CUDAlla.
Liite 3. RGB-kuvan skaalaus CUDAlla.

KUVAT

Kuva 1. CPU:n ja GPU:n eroavaisuus (NVIDIA Corporation 2015). 9
Kuva 2. GPU-ohjelmoinnin toiminta (NVIDIA Corporation 2015). 12
Kuva 3. Säielohkoruudukko (NVIDIA Corporation 2015). 16
Kuva 4. Muistihierarkia (NVIDIA Corporation 2015). 17
Kuva 5. Isäntä ja laite (NVIDIA Corporation 2015). 18

KUVIOT

Kuvio 1. Liukulukuoperaatiot sekunnissa CPU:lle ja GPU:lle (NVIDIA Corporation 2015). 10
Kuvio 2. CPU ja GPU muistin teoreettinen kaistanleveys (NVIDIA Corporation 2015). 11
Kuvio 3. Kuvan skaalauksen vertailu. 25

KOODIT

Koodi 1. Vektorilisäys funktioiden vertailu. 14
Koodi 2. CUDA-Kernelin esimerkki. 15
Koodi 3. Matriisilasku kerneli. 15
Koodi 4. Kuvadatan käsittely CPUlla. 21
Koodi 5. Kuvadatan käsittely CUDAlla. 21
Koodi 6. Kuvan skaalaus CPU:lla. 22
Koodi 7. Kuvan skaalausalustus CUDAlla. 23
Koodi 8. Pikselimuunnos mustavalkokuvaan. 24

KÄYTETYT LYHENTEET JA SANASTO

API	Ohjelmointirajapinta
CUDA	Compute Unified Device Architecture , ohjelmointirajapinta-arkkitehtuuri
DDR3	Double Data Rate 3
DRAM	Dynamic random access memory
DirectX	Ohjelmointirajapinta multimedialle
Direct3D	Ohjelmointirajapinta 3d-grafiikalle
GDDR5	Graphics Double Data Rate 5
GPGPU	General-purpose graphics processing unit
GPU	Graphical Processing Unit
OpenCL	Open Computing Language, viitekehys
OpenCV	Open Source Computer Vision, kirjasto
OpenGL	Ohjelmointirajapinta 3d- ja vektorigrafiikalle
OpenACC	Open Accelerators standardi
VMS	Video Management Software

1 JOHDANTO

Opinnäytetyössä tutustutaan graafisten ytimien käyttöön yleishyödyllisessä laskennassa, jolla siirretään taakkaa pois pelkästään keskusytimen harteilta. Erityisesti käydään läpi CUDA ja sen arkkitehtuuri, joka antaa käyttäjälleen hyvät työkalut aloittaa GPU-ohjelmointi NVIDIAn näytönohjaimilla.

Tavoitteena opinnäytetyössä oli tulla tutuksi GPU-ohjelmoinnin kanssa tutkimalla sen historiaa ja tekniikkaa sekä tutustua CUDA-arkkitehtuuriin. Näiden tietojen pohjalta toteutetaan Ksenos-ohjelmistoon CUDA-arkkitehtuurilla kuvankäsittelyratkaisu, joka toimii rinnan keskusytimen kanssa. Toteutuksen tarkoituksena on poistaa taakkaa keskusytimeltä ja nopeuttaa kuvankäsittelyoperaatioita Ksenos-ohjelmistossa. Näiden pohjalta pohditaan, onko kannattavaa ottaa CUDA-arkkitehtuuri käyttöön ja kehittää sitä ohjelmistossa.

Työssä käydään läpi GPU -ohjelmointia ja sen tuomia etuja verrattuna normaaliin prosessoriohjelmointiin, jota käsitellään luvussa 2. Työ toteutetaan käyttäen CUDA-arkkitehtuuria ja siinä käytettävää c/c++ kieltä. Tämän vuoksi opinnäytetyössä käydään tarkasti läpi CUDA-arkkitehtuuri ja sen tuomat ominaisuudet, joita käsitellään luvussa 3.

Opinnäytetyön toimeksiantajana toimi Ksenos OY. Ksenos on suomalainen, videovalvonnan ohjelmistokehitykseen keskittynyt yritys. Yrityksen päätuote on Ksenos VMS, joka on edistyksellinen videovalvontaohjelmisto. Ohjelmistoa haluttiin optimoida käyttämään hyödykseen myös graafisen ytimen tarjoamaa laskutehoa, jotta keskusytimeen kohdistuvaa taakkaa voitaisiin pienentää ja toiminnallisuuksia ohjelman sisällä nopeuttaa. Ksenos halusi selvittää CUDA-kehitystyöhön panostamisen kannattavuuden ohjelmiston sisällä. (Ksenos Oy 2015.)

2 GPU-OHJELMOINTI

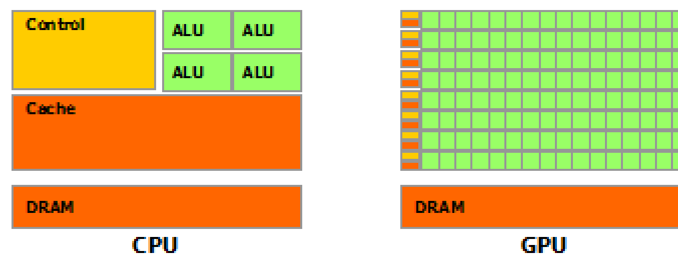
Graafisten suorittimen käyttö piirtoon ja muuhun tietokoneen grafiikkaan on hyvin tunnettu ja käytetty, mutta näiden suorittimien käyttö yleiseen ohjelmointiin on vasta lähiaikoina tullut pinnalle. Rinnakkaisalgoritmit voivat suorittaa tehtävänsä GPGPU-tekniikalla jopa 100 -kertaisella nopeudella CPU:hun verrattuna. Yleiskäyttöinen GPU suorittaa ei- erikoistuneita laskelmia, jotka on tyypillisesti jätetty CPU:lle. (Cruz 2009.)

GPU- ohjelmoinnin tehokkuus perustuu ytimiin, joita on useasti satoja. Vaikka GPU:t toimivat matalammilla taajuuksilla kuin CPU:t, ne käsittelevät ytimiensä ansioista graafista dataa lähes sata kertaa nopeammin. GPGPU tuli käytännölliseksi ja suosituksi 2000- luvun taitteessa, kun ohjelmoitavat varjostimet tulivat arkipäiviäisiksi. Ongelmat esimerkiksi matriisi- ja vektorilaskennassa pystyttiin tämän jälkeen kääntämään suoraan graafisen ytimen tehtäväksi. Ensimmäiset yritykset käyttää GPU:ta yleisen käytön prosessoreina vaativat laskutoimitusten kääntämistä graafisiksi primitiiveiksi tätä tuki kaksi ohjelmointirajapintaa OpenGL ja DirectX. (Wikipedia 2015.)

GPGPU:ta käytetään tehtäviin, jotka olivat aiemmin suuritehoisten prosesoreiden tehtäviä, kuten fysiikan laskelmat, salaus, saulauksien purku, tieteelliset laskutoimitukset ja virtuaalivaluutat kuten Bitcoin. Koska näytönohjaimet rakennetaan massiivisesti rinnakkaisiksi, niiden laskentateho ohittaa helposti kaikkein tehokkaimmatkin prosessorit. Samat shader- eli varjostinytimet, jotka mahdollistavat useiden pikselien suorittamisen samanaikaisesti voivat vastaavasti käsitellä useita datavirtoja samanaikaisesti. Vaikka varjostinydin ei ole läheskään niin monimutkainen kuin CPU, korkean tason GPU:ssa voi olla tuhansia varjostinytimiä; sitä vastoin moniytimisessä CPU:ssa voi olla neljästä kahteentoista ydintä. (Haughn 2015.)

2.1 GPU:n ja CPU:n erot

Kuva 1 selittää prosessorin ja graafisen ytimen eroavaisuuksia. Prosessorin ohjaus, laskentayksiköt ja välimuistit ovat huomattavasti suuremmat ja toisinsanoen tehokkaammat kuin grafiikkapiirillä, mutta DRAM -muistien koot ovat samankokoisia.

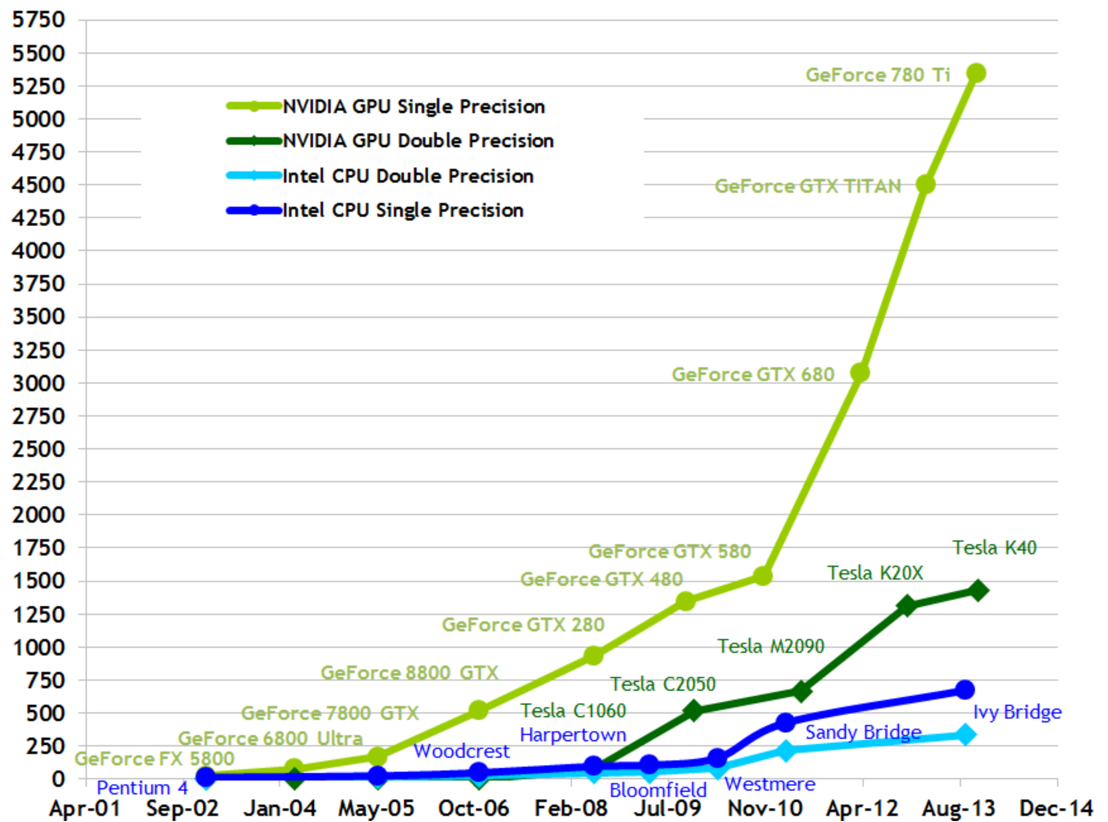


Kuva 1. CPU:n ja GPU:n eroavaisuus (NVIDIA Corporation 2015).

Arkkitehtuurilliset eroavaisuudet ovat johtaneet siihen, että tämän hetken parhaimmistoon luketuvissa näytönohjaimissa on käytössä GDDR5-muistia, kun taas prosessoreilla on käytössä DDR3-muistia. Muistien kaistanleveyksissä erot ovat moninkertaiset, sillä GDDR5-muisteilla pystytään saavuttamaan näytönohjaimilla maksimissaan yli 300 Gt:n/s, kun taas DDR3:lla nopeudet jäävät alle 100 Gt:n/s, kuten nähdään kuviosta 2.

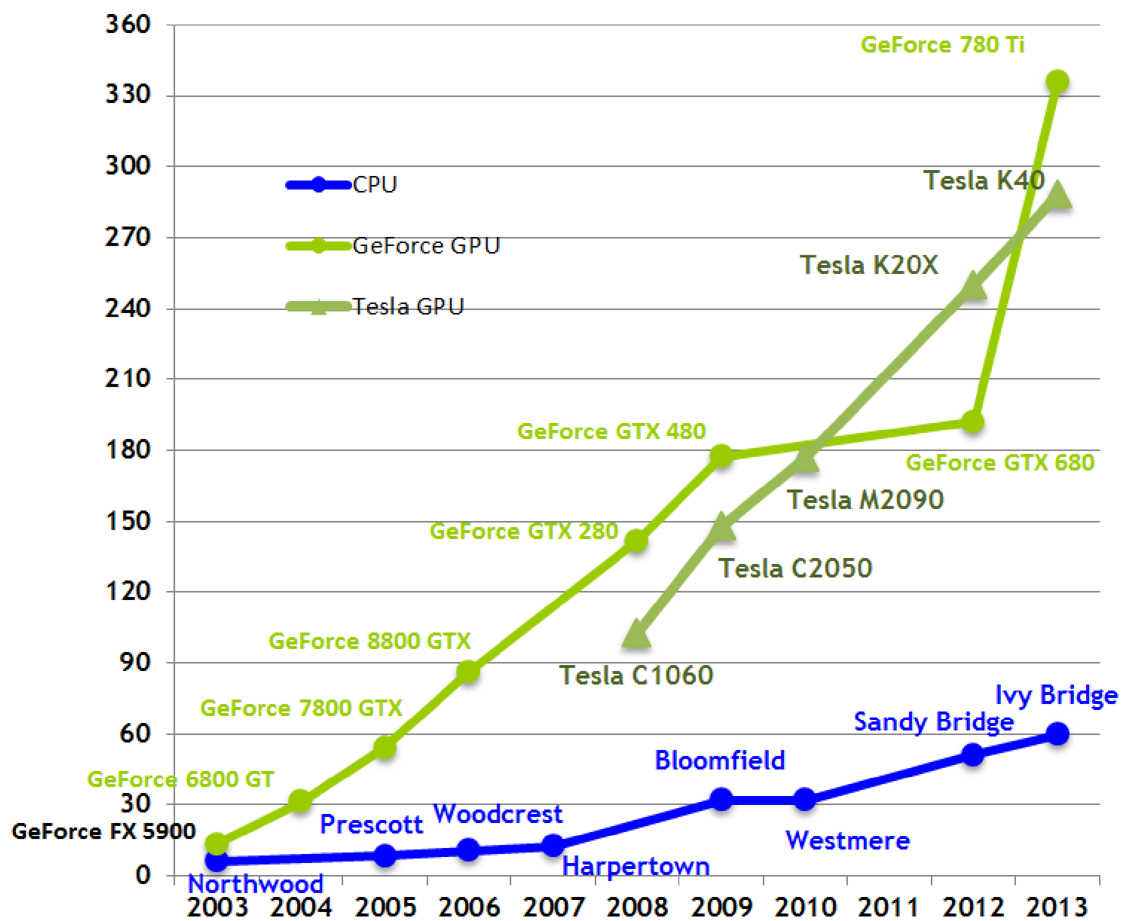
Suuren kysynnän reaaliaikaisen, teräväpiirtoisen 3D-grafiikan ansiosta ohjelmoitava GPU on kehittynyt erittäin rinnakkaiseksi, monisäikeiseksi, moniytimiseksi prosessoriksi valtavalla laskennallisella hevosvoimalla(NVIDIA Corporation 2015.).

Kuviosta 1 nähdään kuinka suuret erot nykyajan prosessoreiden ja näytönohjainten välillä liukuluku operaatioissa. Tämä johtuu yksinkertaisesti sanottuna arkkitehtuurillisista ratkaisuista. Koska näytönohjain on luotu grafiikan käsittelylle ja prosessori ei, tarjoaa näytönohjain tietyissä tilanteissa parempaa suorituskykyä.



Kuvio 1. Liukulukuoperaatiot/s CPU:lla ja GPU:lla (NVIDIA Corporation 2015).

Kuviossa 2 nähdään kaistanleveyden suuret erot prosessorien ja näytönohjainten välillä. Kaistanleveyden erot johtuvat käytössä olevista muisteista, jotka ovat näytönohjaimille GDDR5-muistia ja prosessoreille DDR3-muistia.

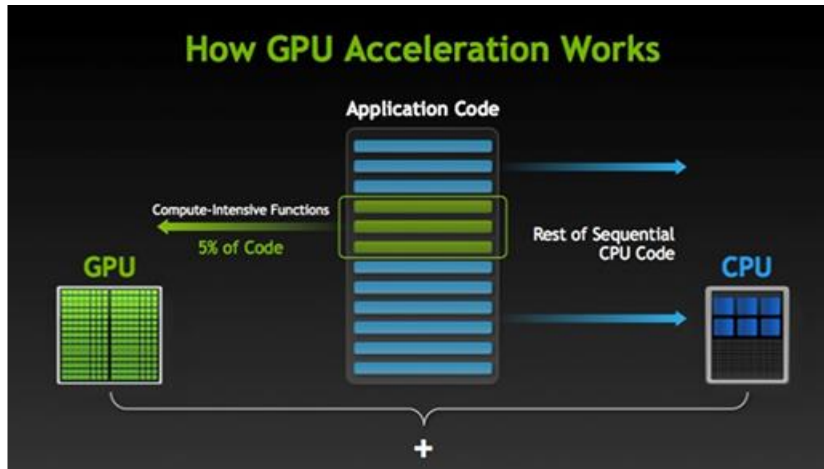


Kuvio 2. CPU:n ja GPU:n muistin teoreettinen kaistanleveys (NVIDIA Corporation 2015).

Syynä CPU:n ja GPU:n väliseen liukulukuvalmiuden eroavaisuuteen on, että GPU on erikoistunut laskentatehoa vaativaan, todella rinnakkaiseen laskentaan juurikin mitä grafiikan renderöinti on ja siksi GPU on suunniteltu siten, että enemmän transistoreja on omistettu tietojenkäsittelyyn.(NVIDIA Corporation 2009.)

2.2 GPU:n tekniikka

GPU on erityisen hyvä käsittelemään ongelmia, jotka voidaan ilmaista rinnakkaisdata-laskelmina, sama ohjelma suoritetaan dataelementeissä rinnakkain. Koska sama ohjelma suoritetaan jokaiselle dataelementille, on pienempi vaatimus hienostuneeseen virtauksen ohjaukseen sekä muistinkäytön latenssi voidaan piilottaa laskemilla ison välimuistin sijasta. (NVIDIA Corporation 2015.)



Kuva 2. GPU-ohjelmoinnin toiminta (NVIDIA Corporation 2015).

Kuvassa 2 havainnolistetaan GPU-ohjelmoinnin toiminnallisuutta, jossa osoitetaan miten CPU:lla ajetaan suurin osa koodista ja GPU:lle annetaan vain eniten rinnakkaislaskentatehoa vaativat funktiot. Kuva kiteyttää hyvin GPGPU:n idean, joka on taakan poistaminen prosessorin harteilta hyvinkin pienillmä muutoksilla.

3 CUDA

Useimmat luulevat CUDAn olevan vain ohjelmointikieli tai ohjelmointirajapinta, mutta CUDA on NVIDIAn rinnakkaislaskennan arkkitehtuurin nimi, joka on käytössä NVIDIAn tuottamilla graafisilla ytimillä. NVIDIAn toolkit tarjoaa kattavat työkalut CUDA-arkkitehtuurin ohjelmointia varten. CUDA toolkit sisältää kääntäjän, virheiden jäljittäjän, profiloijan, kirjastot ja muut tiedot kehittäjille, jotka ovat ottamassa CUDA-arkkitehtuurin käyttöön. (NVIDIA Corporation 2015.)

CUDA alusta on suunniteltu toimimaan ohjelmointikielillä kuten c, c++ ja Fortran. Tämä monipuolisuus helpottaa kehittäjien rinnakkaisohjelmointityötä, toisin kuin CUDAa edeltävät rajapintaratkaisut, kuten Direct3D ja OpenGL, jotka vaativat kehittyneitä taitoja grafiikkaohjelmoinnissa. Cuda kuitenkin tukee viitekehyksiä kuten OpenACC ja OpenCL. (Wikipedia 2016.)

Moniytimiset GPUt ja CPUt, jotka ovat lähes jokaisessa tietokoneessa ovat nyt rinnakkaisia järjestelmiä. Lisäksi niiden rinnakkaisuus skaalautuu Mooren lain kautta. Haasteena on kehittää sovellusohjelmistoja, jotka käyttävät tehokkaasti kaikki saatavillat olevat ytimet. CUDAn rinnakkaisohjelmointimalli on suunniteltu voittamaan tämä haaste säilyttäen alhainen oppimiskäyrä ohjelmoijille, jotka osaavat standardeja ohjelmointikieliä kuten c:tä. (Ebersole 2012.)

CUDAn ydin toimii kolmella keskeisellä seikalla: hierarkia säieryhmien välillä, jaettu muisti ja estesynkronointi. Nämä tuodaan ohjelmoijalle yksinkertaisina ja pieninä kielilaajennuksina. Nämä ydinseikat tarjoavat helppokäyttöisen tiedon rinnakkaisuuden, sekä säie rinnakkaisuuden hyödyntämisen. Ne ohjaavat ohjelmoijaa eristämään ongelmat aliongelmiksi, jotka voidaan ratkaista itsenäisesti rinnakkain omissa lohkoissaan. Kukin aliongelma voidaan paloitella vielä pienempiin osiin, jotka voidaan ratkaista samanaikaisesti rinnakkain lohkon sisällä olevissa säikeissä.(NVIDIA Corporation 2015.)

Koodissa 1 on esimerkki c-kielellä toteutetusta yksinkertaisesta vektorilisäys funktiosta sekä CUDA-arkkitehtuurilla toteutetusta vektorilisäys kernelistä. Koodista huomataan, että CUDAlla toteutettuna silmukoita ei enää tarvita vaan ne on korvattu toimittamalla laskutoimitukset suoraan omille säikeilleen. Laskutoimitukset suoritetaan rinnan ja saavutetaan nopeampia ajoaikoja.

```
1  //CPU vektorilisäys
2  void add(double *a, double *b, double *c, int SIZE){
3          int i;
4          for(i = 0; i < n; ++i) {
5              c[i] = a[i] + b[i];
6      }
7  }
8  //CUDA vektorilisäys kerneli
9  __global__ void add(double *a, double *b, double *c){
10     int i = threadIdx.x; c[i] = a[i] + b[i];
11 }
```

Koodi 1. Vektorilisäys funktioiden vertailu.

3.1 Kernelit

CUDA C laajentaa tavallista C:tä sallimalla ohjelmoijan määritellä funktioita, nimeltään Kernelit niin, että niitä suoritetaan N kertaa rinnakkain, N tarkoittaen CUDA säikeitä. Toisin kuin tavallisessa C ohjelmassa, jossa funktiot suoritetaan vain kerran CUDA suorittaa ne ohjelmoijan haluamalla määrällä.(NVIDIA Corporation 2009)

CUDA kernelit määritellään käyttäen __global__ ilmoituksen julistajaa, ja CUDA säikeiden määrä, jolla funktio suoritetaan määritetään syntaksilla:<<<... >>>. Jokaiselle säikeelle, joka suorittaa funktioita annetaan uniikki säie ID, joka on käytettävissä threadIdx muuttujalla.(NVIDIA Corporation 2009)

Koodissa 2 on esimerkki kernelin luonnista ja kutsumisesta CUDA-arkkitehtuurilla. Koodissa suoritetaan yksinkertainen vektorilisäys A ja B muuttujien välillä sekä talletetaan tulos C muuttujaan. Koodi suoritetaan N määrällä säikeitä, joka riippuu ohjelmoijan haluamasta määrästä.

```
1  // Kernel luonti
2  __global__ void VecAdd(float* A, float* B, float* C) {
3      int i = threadIdx.x; C[i] = A[i] + B[i];
4  }
5  int main() { ... // Kernelin kutsuminen N määrällä säikeitä
6      VecAdd<<<1, N>>>(A, B, C); ...
7  }
8
9
```

Koodi 2. CUDA-Kernelin esimerkki.

3.2 Säiehierarkia

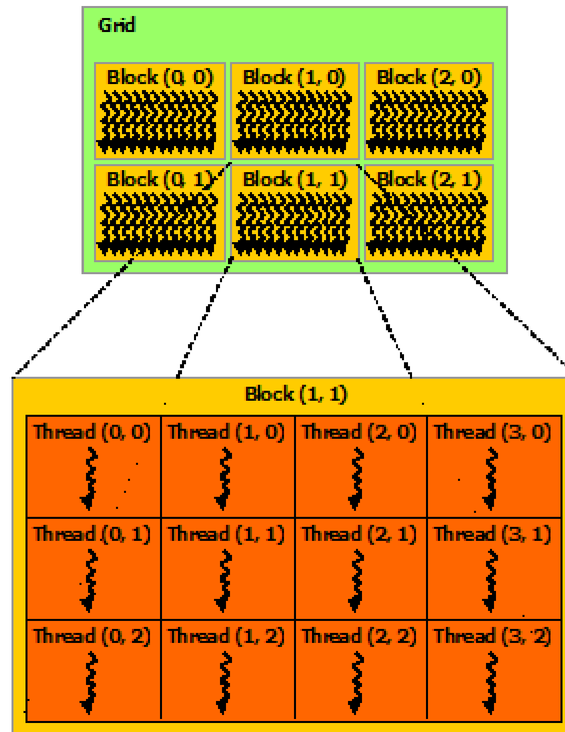
Mukavuussyistä threadIdx on 3-komponentin vektori, jonka avulla säikeet voi tunnistaa yksi-, kaksi- tai kolmiulotteisina säie indekseinä. Tämän avulla voidaan muodostaa säielohkoja, joka antaa luonnollisen tavan kutsua laskentaa kaikille elementeille esimerkiksi vektori- ja matriisilaskuilla. Esimerkkinä tästä koodissa 2 lasketaan matriisit A ja B yhteen N määrällä säikeitä ja talletetaan arvot matriisiin C.(NVIDIA Corporation 2009)

```
1  // Kernelin luonti
2  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]){
3   int i = threadIdx.x;
4   int j = threadIdx.y;
5   C[i][j] = A[i][j] + B[i][j];
6  }
7  int main(){
8   // Kernelin kutsuminen
9   dim3 dimBlock(N, N);
10  MatAdd<<<1, dimBlock>>>(A, B, C);
11 }
```

Koodi 3. Matriisilasku kerneli.

Säikeiden määrä lohkoissa on rajoitettu, sillä kaikki nämä säikeet sijaitsevat samassa prosessoriytimessä, jolla on tietty muistimäärä. Tämänhetkisissä ytimissä säielohko voi sisältää enintään 1024 säiettä.

Lohkot on järjetetty yksi- , kaksi- tai kolmeulotteiseksi ruudukoksi kuten esitetään kuvassa 2. Säielohkojen määrä ruudukossa määräytyy prosessoitavan datan määrän mukaan tai järjestelmässä olevien prosessorien mukaan, joka tarjoaa ohjelmoijalle vapauden päättää itse haluamansa määrän lohkoja ja säikeitä.



Kuva 3. Säielohkoruudukko (NVIDIA Corporation 2015).

Säikeiden määrä lohkossa ja lohkojen määrä voi olla koodissa muotoa int tai dim3 ja se määritellään:<<< ... >>> syntaksin sisällä. Jokainen lohko ruudukon sisällä pystytään tunnistamaan indeksillä, johon pääsee käsiksi käyttämällä blockIdx muuttujaa, kun taas lohkojen ulottuvuuteen pääsee käsiksi käyttämällä blockDim- muuttujaa.(NVIDIA Corporation 2009)

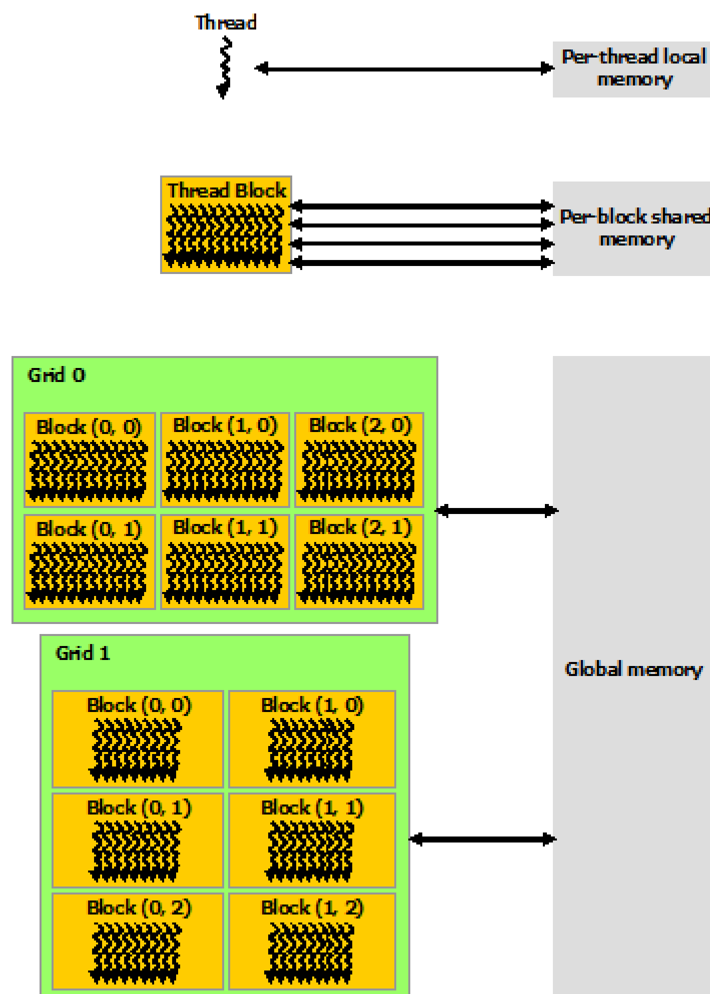
Säikeet lohkon sisällä pystyvät toimimaan keskenään jakamalla dataa jaetun muistin kanssa ja synkronoimalla toteuttamista, jotta voidaan ohjailla muistinkäyttöä. Tarkemmin sanottuna voidaan asettaa synkronointipisteitä ohjelmaan kutsumalla syncthreads() funktiota, joka toimii esteenä kaikille säikeille lohkon sisällä oleville säikeille.(NVIDIA Corporation 2009)

3.3 Muistihierarkia

CUDA -säikeet pystyvät hakemaan dataa useista muistipaikoista toteutuksen aikana kuten näytetään kuvassa x. Jokaisella säikeellä on oma paikallinen musti sekä lohkon

sisällä oleville säikeille on jaettu muisti. Kaikki säikeet pystyvät myös käyttämään globaalia muistia.(NVIDIA Corporation 2009.)

On myös olemassa kaksi kirjoitussuojattua muistipaikkaa johon säikeet pääsevät käsiksi: vakio ja tekstuuri muistipaikat ovat optimoitu erilaisiin muistikäyttö tarkoituksiin kuten vakioiden tallentamiseen. Globaali, vakio sekä tekstuurimuisti pysyvät samoina jokaisen kerneli käynnistyksen aikana ohjelman sisällä. Kuvassa 5 osoitetaan miltä muistihierarkia CUDAn sisällä näyttää.(NVIDIA Corporation 2009.)

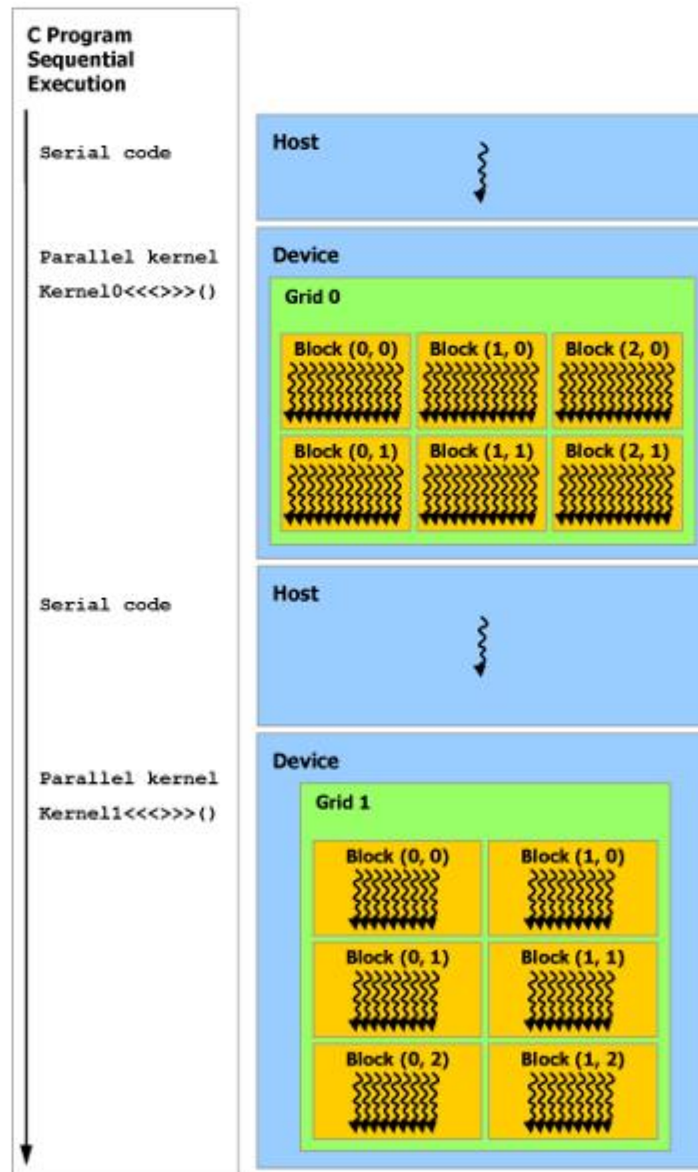


Kuva 4. Muistihierarkia (NVIDIA Corporation 2015).

3.4 Isäntä ja laite

CUDAssa isännällä viitataan CPU:hun ja laitteella GPU:hun. CUDAn ohjelmointimalli olettaa, että säikeet suoritetaan eri fyysisellä laitteella, joka toimii rinnakkaisuorittimena

isännälle, joka suorittaa C ohjelmaa. Tämä oletus toteutuu, kun GPU suorittaa kerneliä ja loppu C ohjelma suoritetaan CPU:lla. CUDAn ohjelmointimalli olettaa myös että isännällä ja laitteella on oma DRAM eli isäntämuisti ja laitemuisti. Isännän ja laitteen suhde on osoitettu kuvassa 6, jossa demonstroidaan ohjelman kulkua.(NVIDIA Corporation 2009.)



Kuva 5. Isäntä ja laite (NVIDIA Corporation 2015).

4 KUVAPROSSOINNIN VERTAILU GPU:N JA CPU:N VÄLILLÄ

Ksenos -videovalvontaohjelmisto on tehokas työkalu suurienkin kameramäärien valvontaan. Useiden kameroiden prosessointi vie kuitenkin paljon prosessoritehoja ja yritys haluaa optimoida ohjelmaansa käyttämään hyödykseen GPU:n tarjoamaa rinnakkaislaskentaa. Optimointi kohdistuu tietyille osa-alueille, jotka vaativat paljon pieniä laskelmia, kuten kuvadatan käsittely ja kuvien skaalaus. Optimoinnin tarkoituksena on poistaa taakkaa CPU:lta sekä nopeuttaa tiettyjä osia ohjelmistosta.

Pienet laskutoimitukset ohjelman sisällä vievät pidemmällä aikajaksolla paljon aikaa CPU:lta, joten ne voidaan CUDAn avulla siirtää GPU:n tehtäväksi. Laskennat voidaan suorittaa rinnan CPU:lla, jolloin siihen kohdistuva taakka laskee huomattavasti pidemmällä aikavälillä.

Kaikki tässä osiossa osoitetut testitulokset on suoritettu kokoonpanolla, joka koostuu 4-ytimisestä prosessorista ja CUDA -laskentakykyversio 3.0 sisältävästä GPU:sta. Tämä kokoonpano on hyvä osoittamaan toiminnallisuuserot, koska nämä ovat tämän hetken yleisimmät komponentit normaalikäytössä. (Athow 2015.)

4.1 CUDAn edut

Kuva- ja videoprosessointi ovat tehty rinnakkaisdata prosessointiin, koska esimerkiksi jokainen pikseli voidaan kartoittaa suoraan omalle säikeelleen tai jopa useampi pikseli voidaan osoittaa säikeelle. CUDA tarjoaa hyvän tavan suorittaa säiesynkronointia sekä runsaasti jaettua muistia, joka helpottaa kuvadatan käsittelyä. (NVIDIA Corporation 2009)

Yksikin teräväpiirtokuva voi sisältää yli 2 miljoonaa pikseliä. Useat kuvien prosessointialgoritmit vaativat kymmeniä liukulukulaskuja, mikä aiheuttaa hidasta ajoaikaa jopa nopeimmillakin CPU yksiköillä. Kuvien prosessointi kartoittuu tosin CUDAlle todella hyvin. Kuvalaatat sopivat suoraan ruudukkoon ja sen sisällä oleviin säielohkoihin. Suuret määrät dataa käsitellään nopeasti suuren muistin kaistanleveyden ansiosta.

CUDAlla käytetään yleensä tasan yksi säie pikseliä kohden ja jokainen säie on vastuussa yhden pikselin prosessoinnista. Koska kuvat ovat kaksiulotteisia on luonnollista, että myös lohkot ovat sellaisia. Lohkoja on tehtävä tarpeeksi monta x- ja y-ulottuvuudessa, jotta katetaan koko kuva, esimerkiksi 1024 x 768 kuvaan tarvitaan ruudukko, joka sisältää 32x48 lohkoja ja näiden sisällä säikeitä 32 x 16. CUDA-arkkitehtuurin säiesynkronointi vastaa siitä, että kaikki käytetyt säikeet suorittavat tehtävänsä oikein.

4.2 Kuvadatan käsittely

Kuvadata käsitellään ohjelmistoissa yleensä pikseleittäin tai pikselialueittain. CPU-implementaatiot datan käsittelystä vaativat paljon aikaa, sillä ne toteutetaan silmukoilla, joilla haetaan oikeat kohdat akseleilta. Oikeiden pikselikohtien haun jälkeen ne prosessoidaan ohjelmoijan haluamalla tavalla kuten värimuutoksilla. (Roberts 2009.)

Kuvadatan käsittely on tärkeä osa Ksenos-ohjelmistoa, sillä videovalvonta perustuu kuvien analysointiin ja datan käsittelyyn. Kuvien ja erityisesti yksittäisten pikselien tiedonhaku suoritetaan CPU:lla yleisesti käyttäen yksinkertaista silmukkaa, joka hakee oikean paikan x- ja y-akseleilta. GPU tekee haut massiiviseen rinnakkaisuuten sopivalla tavalla ja hoitaa pikselit yksittäin omilla säikeillään.

4.2.1 CPU ja kuvadata

Kun CPU:lla käsitellään kuvadataa esimerkiksi pikseleitä on ne prosessoitava yksi kerrallaan. Säikeitä on rajoitettu määrä, joka tällä hetkellä on korkeimmillaan 12. Koodissa 2 CPU:lla suoritetaan kuvan pikseleiden haku sekä funktio, joka muuttaa pikseleiden arvoja. Tämä on yleisin tapa hakea tietty pikselikohta kuvasta, kun käytetään c-kieliä. (Roberts 2009.)

```
for(int y=0; y < height; y++) {
    for(int x = 0; x < width; x++) {
        //haetaan pikselin arvo kohdasta x,y.
        int pos =y * width + x
        image[pos]
        processImage(image[pos]);
    }
}
```

Koodi 4. Kuvadatan käsittely CPUlla.

CPU:lla tehtävät pikselimuunnokset vievät paljon aikaa, koska pikselit haetaan yksi kerrallaan. Isommat kuten teräväpiirto kuvat vievät paljon turhaa ajoaikaa CPU:lla ja sen suuri laskentateho ja erityisesti aika menee hukkaan pikseleiden mikrolaskuihin.

4.2.2 GPU ja kuvadata

CUDAlla voidaan for-silmukat kääntää suoraan laskutoimituksiksi, jotka näkyvät koodissa 3. Pikseleiden haku toteutetaan laskutoimituksilla, jotka määrittävät oikean kohdan lohkojen ja niiden säikeiden sekä kuvan välillä. GPU:lla kuvan pikselit on mahdollista kartoittaa suoraan säikeille ja prosessoida ne rinnan, joten käsittelyaika vähenee huomattavasti.

```
void cudaMap(int width, int height){
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= height || j >= width) return;
}
```

Koodi 5. Kuvadatan käsittely CUDAlla.

CUDA antaa myös muistihierarkiansa ansiosta mahdollisuuden jakaa dataa kerneleiden välillä. Tämä mahdollistaa sen, että jos pikselit ovat riippuvaisia toisien pikselien arvoista, tieto saadaan kulkemaan ilman vaikeuksia.

4.3 Kuvien skaalaus

Ksenos-ohjelmisto käyttää kuvien skaalausta hyödykseen, joka tehdään tässä tapauksessa laskemalla 4 lähimmän pikselin keskiarvo ja ydistämällä ne yhteen pikseliin. Kuvien skaalauksessa tehdään useita pieniä keskiarvolaskuja, joten luonnollinen valinta tähän on CUDA. Kuvien skaalausalgoritmejä on olemassa useita yksinkertaisesta pikselin ohituksesta kehittyneempiin kerneleihin. Nopeat ratkaisut kuten pikselien ohitus kuitenkin aiheuttavat usein kuvissa aliasoitumista, mikä vääristää kuvaa. Suositut kuvan prosessointikirjastot kuten OpenCV tekevät sen yksinkertaisena geometrisenä muuntamisena, jossa alueet kuvasta pienennetään itsenäisesti.

4.3.1 CPU-implementaatio

CPU-implementaatio skaalaukseta on helppo tehdä käyttäen OpenCV kirjastoa kuten nähdään koodista 4. OpenCV:llä toteutettu ohjelma toimii vakaasti ja on yksi ajoaikaa katsoen nopeimmista skaalausratkaisuista. Toteutuksessa käytetään kirjaston tarjoamaa resize funktiota, joka ottaa vastaan parametreinä sisääntulokuvan, ulostulokuvan ja skaalausarvot. CV_INTER_AREA muuttuja viittaa tyyliin, jolla kuva muunnettaan. Tässä tapauksessa käytetään lähimpien 4 pikselin keskiarvon laskemista yhteen pikseliin.

```
1  void processUsingOpenCvCpu(std::string input_file, std::string output_file) {
2      Mat input = imread(input_file, CV_LOAD_IMAGE_COLOR);
3      if(input.empty())
4      {
5          std::cout<<"Image Not Found: "<< input_file << std::endl;
6          return;
7      }
8      //ulostulo
9      Mat output;
10     GpuTimer timer;
11     timer.Start();
12     // 4x alasskaalaus x ja y akselilla
13     resize(input, output, Size(), .25, 0.25, CV_INTER_AREA);
14
15     timer.Stop();
16     printf("Cpu koodi ajettiin ajassa: %f msecs.\n", timer.Elapsed());
17
18     imwrite(output_file, output);
19 }
```

Koodi 6. Kuvan skaalaus CPU:lla.

4.3.2 CUDA-implementaatio

GPU-implementaatio voidaan tehdä käyttäen OpenCV kirjastoa, mutta ajastustestit osoittavat kuvan prosessoinnin olevan nopeampaa, kun se tehdään CUDAa hyödyntäen. CUDA-implementaatiossa on tehtävä muistinvaraukset isännälle ja laitteelle sekä määrittää ruudukon ja lohkojen koko kuvaa vastaavaksi, jotta kuva saadaan oikeana CUDA kerneliin ja ulos. Koodissa 5 on alustus kuvanskaalausohjelmalle, jossa tehdään muistinvaraukset, ruudukko- ja lohkomääritykset, kernelin ajo ja mitataan kulunut aika.

```
void downscaleCuda(const cv::Mat& input, cv::Mat& output) {
    //Lasketaan input ja output kuvasta
    const int inputBytes = input.step * input.rows;
    const int outputBytes = output.step * output.rows;
    //laitemuuttujat
    unsigned char *d_input, *d_output;

    //Laitemuistivaraus
    checkCudaErrors(cudaMalloc<unsigned char>(&d_input,inputBytes),"CUDA Malloc Failed");
    checkCudaErrors(cudaMalloc<unsigned char>(&d_output,outputBytes),"CUDA Malloc Failed");

    GpuTimer timer;
    timer.Start();

    //Lohkoalustus
    const dim3 block(16,16);

    //Koko kuvan kattava ruudukko
    const dim3 grid((output.cols + block.x - 1)/block.x, (output.rows + block.y - 1)/block.y);

    //Kernelin laukaisu
    resizeCudaKernel<<<grid,block>>>(d_input,d_output,output.cols,output.rows,input.step,output.step, input.channels());

    timer.Stop();
    printf("Cuda koodi ajettiin ajassa: %f msecs.\n", timer.Elapsed());

    //Kopioidaan kuvadata laitteelta isännälle
    checkCudaErrors(cudaMemcpy(output.ptr(),d_output,outputBytes,cudaMemcpyDeviceToHost));

    //Vapautetaan laitteen muisti
    checkCudaErrors(cudaFree(d_input));
    checkCudaErrors(cudaFree(d_output));
}
```

Koodi 7. Kuvan skaalausalustus CUDAlla.

Itse CUDA kernelin sisällä määritetään ja lasketaan kuinka suuri pikselialue halutaan muuntaa yhdeksi pikseliksi sekä luodaan tämä alue sisääntulevalle kuvalle. Pikselialueen määrityksen jälkeen haetaan oikeat pikselit x- ja y-akseleilla ja yhdistetään ne yhteen ulostulopikseliin. Koodissa 6 nähdään implementaatio mustavalkoiseen kuvaan.

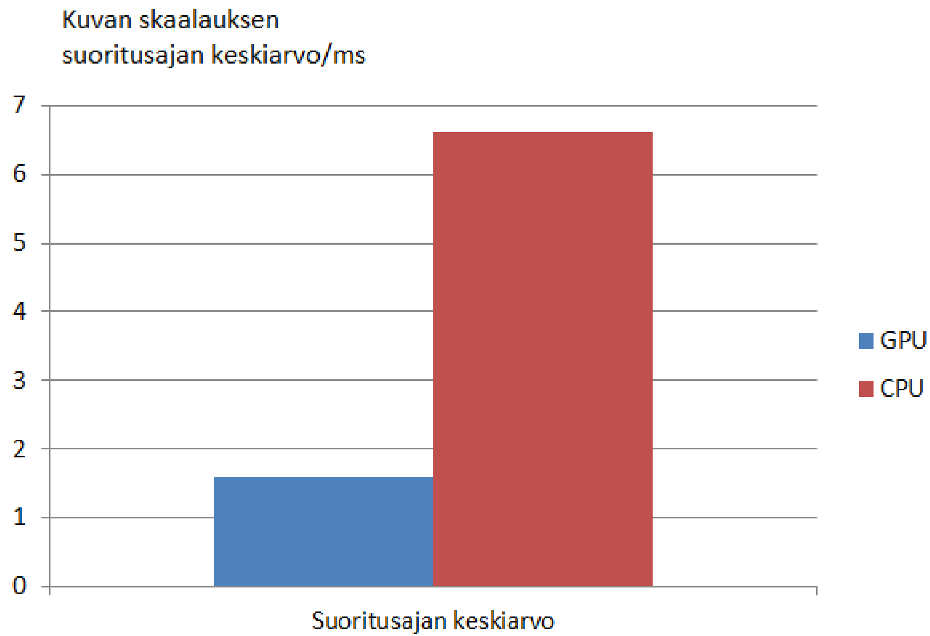
```
if(inputChannels==1){ // Mustavalko
    float channelSum = 0;
    //y akseli
    for(int intputYIndex=intputYIndexStart; intputYIndex<intputYIndexEnd; ++intputYIndex){
        //x akseli
        for(int intputXIndex=intputXIndexStart; intputXIndex<intputXIndexEnd; ++intputXIndex){
            int input_tid = intputYIndex * inputWidthStep + intputXIndex;
            channelSum += input[input_tid];
        }
    }
    //Kirjoitetaan ulostulo kuvaan
    output[output_tid] = static_cast<unsigned char>(channelSum / pixelGroupArea);
}
```

Koodi 8. Pikselimuunnos mustavalkokuvaan.

CUDA-ratkaisu antaa käyttäjälle enemmän muokata ohjelmaa kuten muistin jakaminen kernelien välillä, jolloin dataa ei tarvitse jatkuvasti kopioida CPU:n ja GPU:n välillä. RGB-kuvan skaalaus on myös helppo jatkaa toteutuksesta, koska siihen tarvitaan 2 ylimääräistä kanava muuttujaa.

4.3.3 Vertailu

Erot CPU- ja GPU-ratkaisujen ajoajassa ovat suuria, kuten näkee kuviosta 3. CPU-ratkaisu toteutettuna OpenCVtä käyttäen suoritti yhden kuvan skaalauksen keskivertona 20 otetusta ajoajasta n.6,5 ms:ssa. GPU ratkaisu käyttäen CUDA-arkkitehtuuria suoriutui samasta tehtävästä n.1,6 millisekunnissa. Ero on suuri ottaen huomioon käytetyn OpenCV-kirjaston, joka on tämänhetkisistä vaihtoehdoista nopein. 5 ms ero ei välttämättä yhdessä kuvan skaalauksessa tunnu suurelta ajalta, mutta esimerkiksi Ksenos ohjelmistossa skaalauksia voidaan tehdä tuhansia lyhyen aikavälin sisällä, jolloin millisekuntit muuttuvat minuuteiksi turhaa ajoaikaa.



Kuvio 3. Kuvan skaalauksen vertailu.

Ksenos-ohjelmistossa käytetään paljon kuvadataan liittyviä laskuja, jotka ovat kuin luotu CUDA-arkkitehtuurille. Ksenos halusikin tutkia onko kannattavaa panostaa CUDA kehitystyöhön ohjelmiston sisällä. Vertailun pohjalta ainakin kuvien skaalaus suoritetaan CUDA-arkkitehtuurilla nopeammin. Vaikka jokin ominaisuus suoriutuisi vain marginaalisesti paremmin tai jopa heikommin CUDAa käyttäen on se silti pois CPU:n taakasta joka jo itsessään on hyvinkin positiivista.

5 YHTEENVETO

Opinnäytetyön tavoitteena oli toteuttaa tutkimus GPU-ohjelmoinnista ja CUDA-arkkitehtuurista, jota voitaisiin lisätä Ksenos VMS -ohjelmistoon. Toteutuksen tuli sisältää testiajo tuloksia tietyistä ohjelmiston osa-alueista, joita voitaisiin käyttää päätökseen jatkaa CUDA-kehitystä ohjelmiston sisällä. CUDA-arkkitehtuurista saatiin hyvä tietämys tutkimuksen pohjalta, jota hyödynnettiin tehtäessä omia implementaatioita.

Opinnäytetyön tavoitteet saavutettiin ja Ksenos harkitsee vahvasti CUDA-arkkitehtuurin käyttöönottamista ohjelmistossaan. Tutkimus arkkitehtuurista antoi vahvan pohjan aloittaa GPU:n hyödyntämisen ohjelmistossa.

Valmis tutkimus arkkitehtuurista sekä kuvien skaalaus implementaatio antoivat hyvän käsityksen yritykselle miten CUDAa voitaisiin heidän ohjelmistossa hyödyntää. Lisäksi vankka kokemus c-kielistä ja sen hyödyllisyyden osoittaminen esimerkki koodeissa tukee CUDA-kehityksen käyttöönoton helppoutta.

CUDA-arkkitehtuuri antaa ohjelmoijille loistavat työkalut GPU:n hyödyntämistä varten. Esimerkiksi kuvadatan käsittelyssä on tärkeää saada oikeat säiemäärät käyttöön vastaamaan kuvan suuruutta. Säiemäärät voidaan itse määrittää vakioiksi tai laskea ohjelman sisällä vastaamaan operaation suuruutta.

Ongelmana CUDA-arkkitehtuurin käyttöönotossa on tämänhetkinen kehitysympäristötuki. CUDA ei vielä tue Visual Studio 2015 –kehitysympäristöä, johon Ksenos on viime aikoina siirtynyt. Myös säikeiden käyttö ja niiden hierarkia on vaikeasti omaksuttava jos aikaisempaa kokemusta säikeiden käytöstä ei ole.

Toteutettua kuvien skaalaus ratkaisua on tarkoitus vielä verrata Ksenosin omaan implementaatioon, mutta kehittäjien mukaan se on luultavasti vielä hieman hitaampi kuin OpenCV-ratkaisu.

LÄHTEET

Desire, A. 2015. Best CPU: 10 top processors from AMD and Intel. Viitattu 20.3.2016 http://www.techradar.com/news/computing-components/processors/best-cpu-the-8-top-processors-today-1046063

Felipe A. Cruz 2009. Tutorial on GPU computing. Viitattu 27.3.2016 http://lorenabarba.com/gpuatbu/Program_files/Cruz_gpuComputing09.pdf

GeForce 2015. CUDA faq. Viitattu 10.3.2016 http://www.geforce.com/hardware/technology/cuda/faq.

Intel Corporation 2015. OpenCV documentation. Viitattu 12.4.2016 http://docs.opencv.org/master/index.html#gsc.tab=0

Ksenos OY 2015. Osaamme videovalvonnan. Viitattu 1.4.2016. http://ksenos.fi/fi/tietoa-meista.

Matthew Haughn 2015. GPGPU (general purpose graphics processing unit). Viitattu 20.3.2016 http://whatis.techtarget.com/definition/GPGPU-general-purpose-graphics-processing-unit.

Mark Ebersole 2012. What is CUDA?. Viitattu 24.3.2016 //blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.

NVIDIA Corporation 2015. CUDA Toolkit Documentation. Viitattu 15.3.2016 http://docs.nvidia.com/cuda/index.html#axzz40B1pPyee.

NVIDIA Corporation 2015. What is gpu accelerated computing?. Viitattu 16.3.2016 http://www.nvidia.com/object/what-is-gpu-computing.html.

NVIDIA Corporation 2009. NVIDIA CUDA Programming Guide. Viitattu 10.2.2016 http://www.cs.colostate.edu/~cs675/cudaDocs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

Stephen Roberts 2008. Digital signal & image processing. Viitattu 12.3.2016 http://www.robots.ox.ac.uk/~sjrob/Teaching/B4_SP/b4_sp.pdf.

Wikipedia 2016. CUDA. Viitattu 25.3.2016 https://en.wikipedia.org/wiki/CUDA.

Wikipedia 2015. General-purpose computing on graphics processing units. Viitattu 15.2.2016 https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units.

Kuvan skaalauksen alustus: CUDA

```
void processUsingCuda(std::string input_file, std::string output_file) {
    //luetaan kuva
    cv::Mat input = cv::imread(input_file,CV_LOAD_IMAGE_UNCHANGED);
    if(input.empty())
    {
        std::cout<<"Image Not Found: "<< input_file << std::endl;
        return;
    }

    //luodaan ulostulokuva sekä varataan uudet koot
    Size newSize( input.size().width / 4, input.size().height / 4 );
    Mat output (newSize, input.type());

    downscaleCuda(input, output);

    imwrite(output_file, output);
}
```

Mustavalkokuvan skaalaus: CUDA

```
__global__ void resizeCudaKernel( unsigned char* input,
        unsigned char* output,
        const int outputWidth,
        const int outputHeight,
        const int inputWidthStep,
        const int outputWidthStep,
        const float pixelGroupSizeX,
        const float pixelGroupSizeY,
        const int inputChannels)
{
    //Tämänhetkinen kohta
    const int outputXIndex = blockIdx.x * blockDim.x + threadIdx.x;
    const int outputYIndex = blockIdx.y * blockDim.y + threadIdx.y;

    if((outputXIndex<outputWidth) && (outputYIndex<outputHeight))
    {
        // Aloituspikseli
        int output_tid  = outputYIndex * outputWidthStep + (outputXIndex * inputChannels);

        // Pikselialueen lasku
        const float pixelGroupArea = pixelGroupSizeX * pixelGroupSizeY;

        // Pikselialue sisääntulossa
        const int intputXIndexStart = int(outputXIndex * pixelGroupSizeX);
        const int intputXIndexEnd = int(intputXIndexStart + pixelGroupSizeX);
        const float intputYIndexStart = int(outputYIndex * pixelGroupSizeY);
        const float intputYIndexEnd = int(intputYIndexStart + pixelGroupSizeY);
        // Mustavalkokuvan muunto
        if(inputChannels==1) {
            float channelSum = 0;
            for(int intputYIndex=intputYIndexStart; intputYIndex<intputYIndexEnd; ++intputYIndex) {
                for(int intputXIndex=intputXIndexStart; intputXIndex<intputXIndexEnd; ++intputXIndex) {
                    int input_tid = intputYIndex * inputWidthStep + intputXIndex;
                    channelSum += input[input_tid];
                }
            }
            output[output_tid] = static_cast<unsigned char>(channelSum / pixelGroupArea);
        }

    }
}
```

RGB-kuvan skaalaus: CUDA

```
1  __global__ void resizeCudaKernel( unsigned char* input,unsigned char* output,const int outputWidth,const int outputHeight,
2          const int inputWidthStep,
3          const int outputWidthStep,
4          const float pixelGroupSizeX,
5          const float pixelGroupSizeY,
6          const int inputChannels)
7  {
8      //Tämänhetkinen kohta
9      const int outputXIndex = blockIdx.x * blockDim.x + threadIdx.x;
10     const int outputYIndex = blockIdx.y * blockDim.y + threadIdx.y;
11
12     if((outputXIndex<outputWidth) && (outputYIndex<outputHeight))
13     {
14         // Aloituspikseli
15         int output_tid  = outputYIndex * outputWidthStep + (outputXIndex * inputChannels);
16
17         // Pikselialueen lasku
18         const float pixelGroupArea = pixelGroupSizeX * pixelGroupSizeY;
19
20         // Pikselialue sisääntulossa
21         const int intputXIndexStart = int(outputXIndex * pixelGroupSizeX);
22         const int intputXIndexEnd = int(intputXIndexStart + pixelGroupSizeX);
23         const float intputYIndexStart = int(outputYIndex * pixelGroupSizeY);
24         const float intputYIndexEnd = int(intputYIndexStart + pixelGroupSizeY);
25         // RGB-kuvan skaalaus
26         if(inputChannels==3) {
27             float channel1stSum = 0;
28             float channel2stSum = 0;
29             float channel3stSum = 0;
30             for(int intputYIndex=intputYIndexStart; intputYIndex<intputYIndexEnd; ++intputYIndex) {
31                 for(int intputXIndex=intputXIndexStart; intputXIndex<intputXIndexEnd; ++intputXIndex) {
32                     int input_tid = intputYIndex * inputWidthStep + intputXIndex * inputChannels;
33                     channel1stSum += input[input_tid];
34                     channel2stSum += input[input_tid+1];
35                     channel3stSum += input[input_tid+2];
36                 }
37             }
38             output[output_tid] = static_cast<unsigned char>(channel1stSum / pixelGroupArea);
39             output[output_tid+1] = static_cast<unsigned char>(channel2stSum / pixelGroupArea);
40             output[output_tid+2] = static_cast<unsigned char>(channel3stSum / pixelGroupArea);
41         }
42     }
43 }
```