**Parallell programmering med CUDA**

|  |  |
| --- | --- |
| **Namn** | Öyvind Larsen |
| **Utbildning** | Software Developer Electrical  & Autonomous Vehicle |
| **Datum** | 2024-01-29 |

## Sammanfattning

I denna rapport beskriver jag parallell programmering med CUDA, Compute Unified Device Architecture, för NVIDIAS grafikkort.

En hastighetsjämförelse görs mellan parallell exekvering på processor och grafikkort.

Jämförelsen visar att parallell exekvering på grafikkort är överlägsen parallell exekvering på processor.

**Innehåll**

1. Projektbeskrivning 4

1.1 Bakgrund 4

1.2 Syfte 4

1.3 Mål 4

1.4 Avgränsningar och fokus 4

1.5 Tidplan/aktiviteter 4

2. Metodbeskrivning 4

3. Resultat 5

4. Avslutning och slutsatser 5

5. Referenslista 6

Bilagor 7

**1.** Projektbeskrivning

## Bakgrund

Jag är nyfiken på NVIDIAs CUDA Toolkit tillsammans med C++, som används av flera biltillverkare bland andra Tesla. (NVIDIA s. 6)

## 1.2 Syfte

Beskriva hur bra CUDA grafikkort är på parallell exekvering av kod.

## 1.3 **Mål**

Göra en hastighetsjämförelse mellan parallell kod på processor (CPU) och på NVIDIAs grafikkort (GPU).

1.4 Avgränsningar och fokus  
Jag kommer inte jämföra andra grafikkort än NVIDIAs. Inte heller görs en teknisk beskrivning av CUDA, då det är väldigt omfattande.

## 1.5 Tidplan/aktiviteter

## Inläsning om parallell programmering och CUDA

* Installera Visual Studio Community Edition och CUDA Toolkit
* Exekvera ett program med CUDA kod i C++

# 2. Metodbeskrivning

I (Ansorge, kapitel 1) finns tre olika program för att jämföra hastigheten att exekvera parallellt.

Första programmet körs i ren C++ på en processor kärna.

Andra programmet körs i ren C++ på fyra processor kärnor.

Tredje programmet, som finns i Bilaga 1*,* körs på grafikkort med CUDA.

# 3. Resultat

Exekvering på en processorkärna ger illusionen av parallell exekvering, då endast ett program kan köras samtidigt.

Exekvering på fyra processorkärnor kan exekvera fyra olika program samtidigt.

Exekvering på grafikkort med CUDA ger mycket högre instruktionsgenomströmning och minnesbandbredd än processorn med samma pris och energiförbrukning. (NVIDIA s. 3)

En bild som visar text, skärmbild, Teckensnitt, nummer

Automatiskt genererad beskrivning  
*Tabell 1: Hastighetsjämförelse*

# 4. Avslutning och slutsatser

Parallell exekvering av kod är överlägset snabbast på grafikkort. För att få motsvarande prestanda på processorer krävs fler processorer som dessutom skulle dra mer ström och kosta mer.

# 5. Referenslista

**Litteratur**

Ansorge, Richard, *Programming in Parallel with CUDA,* Cambridge: Cambridge University Press, 2022. ISBN: 978-1-108-47593-0

NVIDIA, CUDA C++ Programming Guide Release 12.3, pdf-fil från <https://docs.nvidia.com/cuda/cuda-c-programming-guide>

# Bilagor

1. CUDA koden för grafikkortsjämförelsen.

// Programming in Parallel with CUDA - supporting code by Richard Ansorge

// copyright 2021 is licensed under CC BY-NC 4.0 for non-commercial use

// This code may be freely changed but please retain an acknowledgement

// program 1.3 gpusum

// RTX 2070

// C:\Users\Richard\OneDrive\toGit2>bin\gpusum.exe 1000000000 1000

// gpu sum = 2.0000000134, steps 1000000000 terms 1000 time 1881.113 ms

// RTX 3080

// C:\Users\Richard\OneDrive\toGit2>bin\gpusum.exe 1000000000 1000

// gpu sum = 1.9999998123, steps 1000000000 terms 1000 time 726.253 ms

// 1060 gpu sum = 1.9999999727, steps 10000000 terms 1000 time 1883.425 ms

#include "cx.h"

#include "cxtimers.h" // cx timers

\_\_host\_\_ \_\_device\_\_ inline float sinsum(float x, int terms)

{

float x2 = x \* x;

float term = x; // first term of series

float sum = term; // sum of terms so far

for (int n = 1; n < terms; n++) {

term \*= -x2 / (2 \* n \* (2 \* n + 1)); // build factorial

sum += term;

}

return sum;

}

\_\_global\_\_ void gpu\_sin(float\* sums, int steps, int terms, float step\_size)

{

int step = blockIdx.x \* blockDim.x + threadIdx.x; // unique thread ID

if (step < steps) {

float x = step\_size \* step;

sums[step] = sinsum(x, terms); // store sin values in array

}

}

int main(int argc, char\* argv[])

{

int steps = (argc > 1) ? atoi(argv[1]) : 10000000; // get command

int terms = (argc > 2) ? atoi(argv[2]) : 1000; // line arguments

int threads = 256;

int blocks = (steps + threads - 1) / threads; // ensure threads\*blocks ≥ steps

double pi = 3.14159265358979323;

double step\_size = pi / (steps - 1); // NB n-1 steps between n points

thrust::device\_vector<float> dsums(steps); // GPU buffer

float\* dptr = thrust::raw\_pointer\_cast(&dsums[0]); // get pointer

cx::timer tim;

gpu\_sin << <blocks, threads >> > (dptr, steps, terms, (float)step\_size);

double gpu\_sum = thrust::reduce(dsums.begin(), dsums.end());

double gpu\_time = tim.lap\_ms(); // get elapsed time

// Trapezoidal Rule Correction

gpu\_sum -= 0.5 \* (sinsum(0.0f, terms) + sinsum(pi, terms));

gpu\_sum \*= step\_size;

printf("gpu sum = %.10f, steps %d terms %d time %.3f ms\n",

gpu\_sum, steps, terms, gpu\_time);

return 0;

}