# Näytönohjainlaskenta fysiikassa

14.10.2010

Topi Siro

# Sisältö

- GPU vs CPU
- Esimerkki: reduktio
- CUDA-arkkitehtuuri
- Harvat matriisit
- Matriisi * vektori CUDAlla

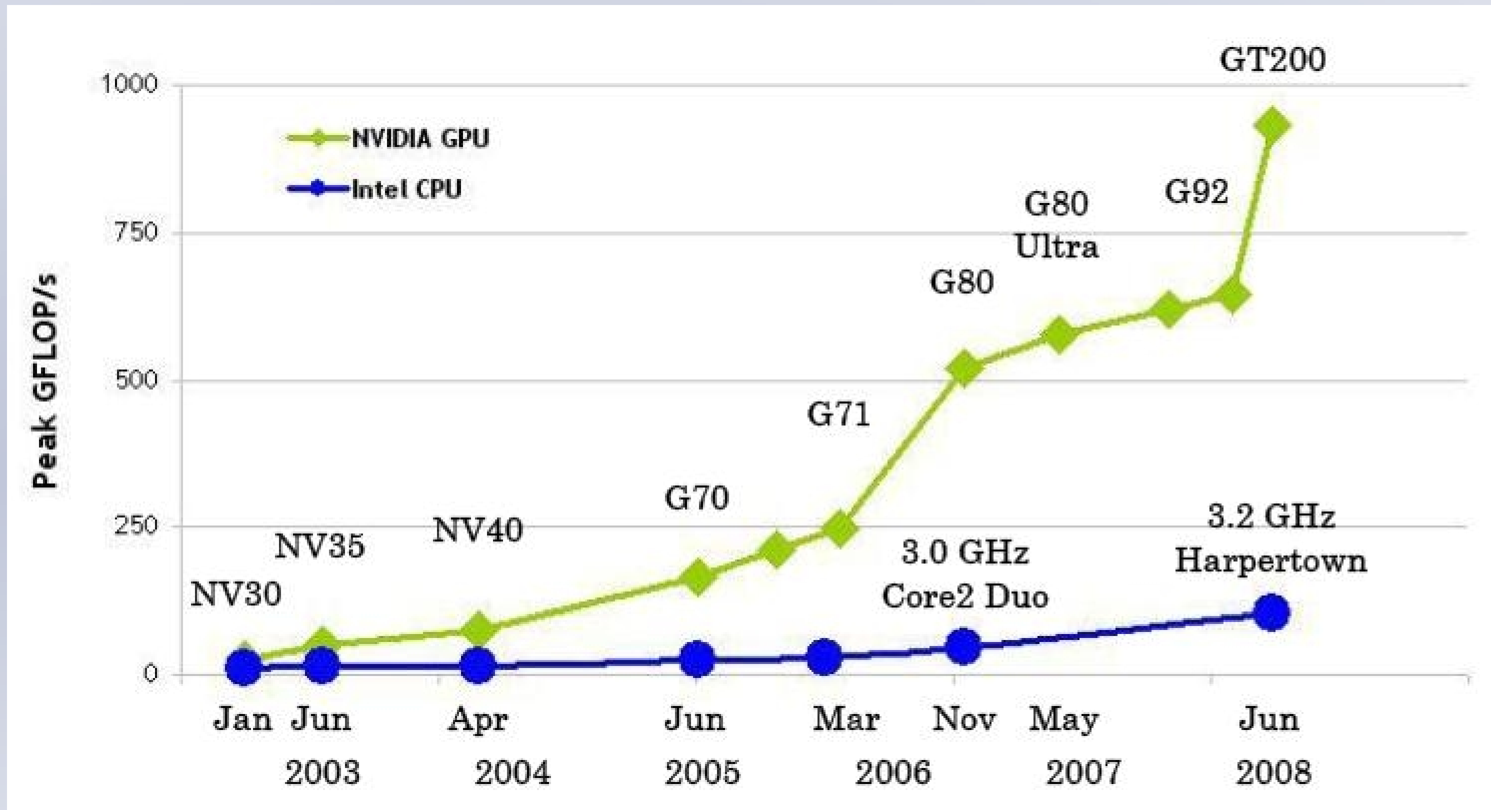
# GPU vs CPU

- Näytönohjainten (GPU) kehitys on viime vuosina ollut nopeaa
- Alunperin tehtävänä grafiikan tehokas piirtäminen
- Nykyään mahdollisuuksia myös suurteholaskennassa (ns. GPGPU)
- CPU: suorittaa yhtä tehtävää nopeasti
- GPU: suorittaa montaa tehtävää hitaammin, mutta rinnakkain

# GPU vs CPU

- GPU parhaimmillaan laskuissa, joissa samaa ohjelmaa ajetaan useita kertoja eri datalle suurella aritmeettisella intensiteetillä (laskuoperaatioiden lkm / muistioperaatioiden lkm)
- Lasku pitää pystyä parallelisoimaan tehokkaasti
  - Rekursiivinen algoritmi sopii CPU:lle:
    - $X_{i+1} = X_{i+1}(X_i)$
  - Esim. matriisi kertaa vektori sopii GPU:lle:
    - $\begin{pmatrix} 1 & 0 & 3 \\ 0 & 6 & 1 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

# GPU vs CPU



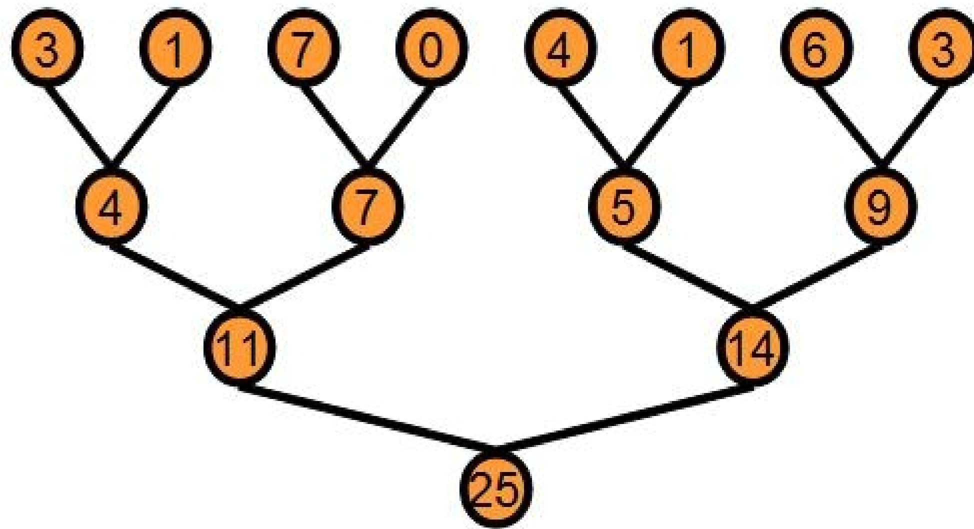
# Esimerkki: reduktio

- Olkoon X taulukko lukuja:
  - X = [ 3, 1, 7, 0, 4, 1, 6, 3 ]
- Tehtävänä on laskea alkioiden summa
- 1. Peräkkäisalgoritmi:

```
int sum = 0;

for (int i=0; i<8; i++)
    {
        sum = sum + X[i];
    }
```

# Esimerkki: reduktio

- 2. Rinnakkaisalgoritmi:




- Jos taulukossa on $2^n$ alkiota, niin algoritmi 1 tarvitsee $2^n$-1 iteraatiota, kun taas algoritmi 2 vain n iteraatiota

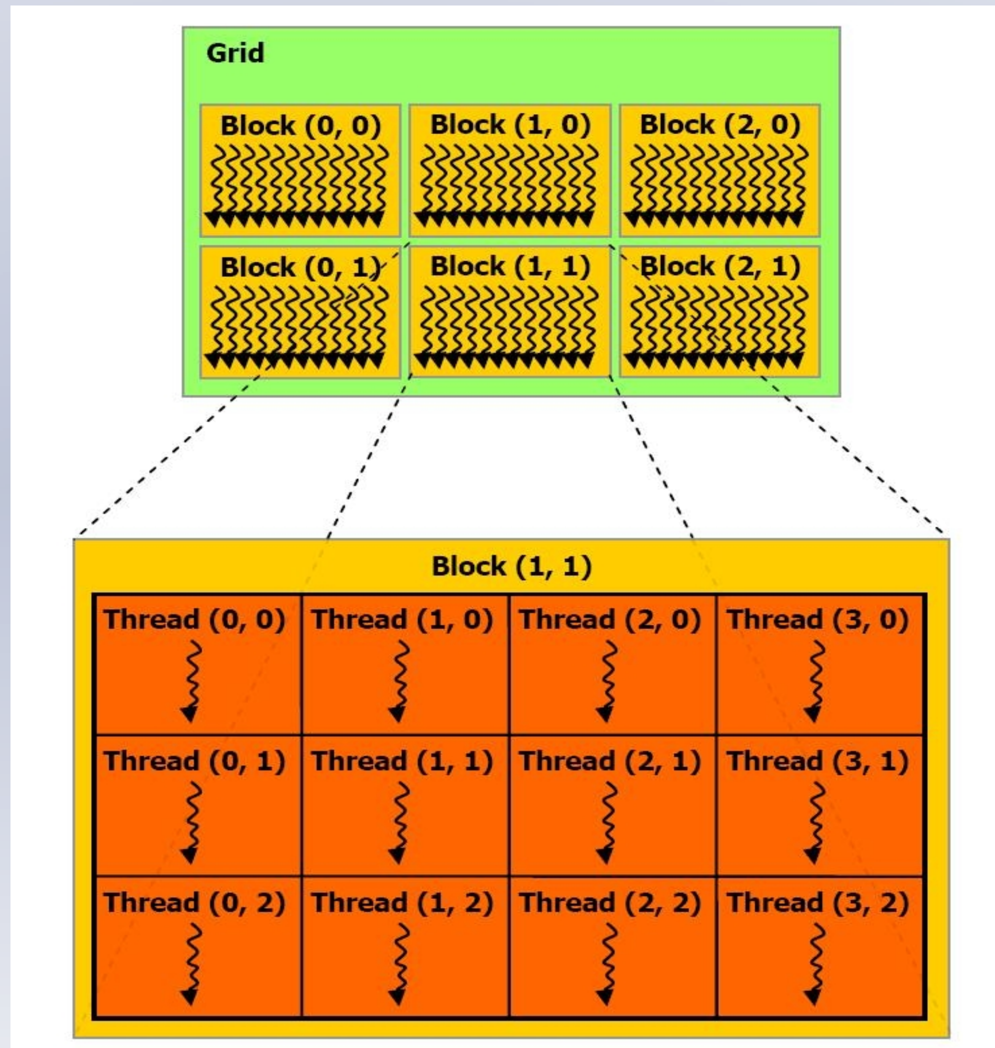
# CUDA

- NVIDIAn kehittämä rinnakkaislaskenta-arkkitehtuuri, joka mahdollistaa yhtiön näytönohjainten valjastamisen laskentakäyttöön
- Ohjelmointikielenä CUDA C, joka on C++:n laajennus
- Olennaisin ero on mahdollisuus määritellä ns. kerneleitä, jotka ovat näytönohjaimella rinnakkain suoritettavia funktioita

# CUDA

- Kerneliä kutsuttaessa määrätään, kuinka monta säiettä kyseistä kerneliä suorittaa
- Kaikki säikeet suorittavat kernelin komennot yhtä aikaa rinnakkain
- Kernelin sisällä säikeiden toimintaa voi ohjailla viittaamalla säikeen järjestysnumeroon, joka on muuttujassa threadIdx
- Säikeet muodostavat blokkeja, ja blokit gridin
- Säikeitä mahtuu näytönohjaimelle yhtä aikaa jopa 30 000

# CUDA



# CUDA

- Esimerkki kernelistä, joka laskee yhteen vektorit A ja B ja tallentaa tuloksen vektoriin C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

# CUDA

- Tärkein asia ohjelman nopean suorituksen kannalta on tehokas muistinkäyttö, koska muistin luku ja kirjoitus on hidasta
- Vierekkäisten säikeiden tulisi tehdä muistipyyntönsä vierekkäsiin kohtiin muistissa, jolloin yksi muistisiirto hyödyttää montaa säiettä kerralla
- Erityisesti siirrot isännän (CPU) ja laitteen (GPU) välillä tulee minimoida

# Harvat matriisit

- Harvat matriisit (suurin osa alkioista nollia) ovat yleisiä fysiikassa, ja keskeinen operaatio on vektorin kertominen matriisilla
- Matriisi*vektori voidaan rinnakkaistaa tehokkaasti, sillä pistetulot matriisin rivien ja vektorin välillä ovat riippumattomia toisistaan
- Mutta miten harva matriisi kannattaisi tallentaa muistiin laskua varten?

# Harvat matriisit

- Otetaan matriisi A
- Neljä yleisintä metodia harvan matriisin esittämiseksi ovat DIA, ELL, CSR ja COO

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

- Menetelmien tehokkuus riippuu matriisin rakenteesta

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

- DIA sopii matriiseille, joissa nollasta poikkeavat alkiot sijaitsevat diagonaaleilla
- ELL puolestaan matriiseille, joilla nollasta poikkeavia on suunnilleen saman verran joka rivillä

DIA format:

$$\texttt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

ELL format:

$$\texttt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

- CSR ja COO ovat yleisempiä menetelmiä, joiden suorituskyky ei juurikaan riipu matriisin rakenteesta

CSR format:

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$
$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$
$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

COO format:

$$\texttt{row} = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}$$
$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$
$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

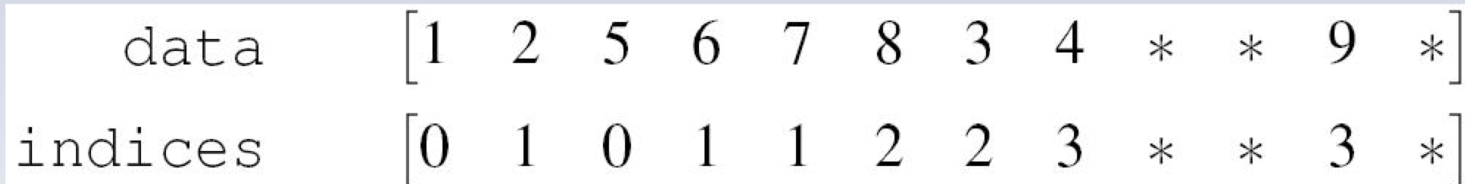
# Harvat matriisit

- Esim. Hubbardin mallille paras säilöntämenetelmä on selkeästi ELL
- Suoritusajat, kun eri menetelmillä säilötyllä 1D-Hubbardin matriisilla kerrottiin vektori (matriisin dimensio n. 2,7 miljoonaa ja nollasta poikkeavia alkioita n. 34 miljoonaa:
  - ELL: 10,7 ms
  - CSR: 45,6 ms
  - COO: 31,0 ms

# Matriisi*vektori Cudalla

- Käytetään matriisin tallentamiseen ELLiä
- Käytännössä matriisi tallennetaan muistiin vektorina:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \longrightarrow \texttt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad \texttt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$



# Matriisi*vektori Cudalla

- HUOM: ELL-matriisit luettiin vektoriksi **sarakkeittain** eikä riveittäin, joka kuulostaisi ehkä luonnollisemmalta
- Tällä tavoin optimoidaan muistinkäyttö, sillä nyt vierekkäisiä rivejä rinnakkain laskevat säikeet hakevat datan vierekkäisistä muistipaikoista
- Nyt voidaan siirtää matriisi ja vektori GPU:lle ja ajaa laskun suorittava kerneli, jossa yksi säie laskee yhden rivin

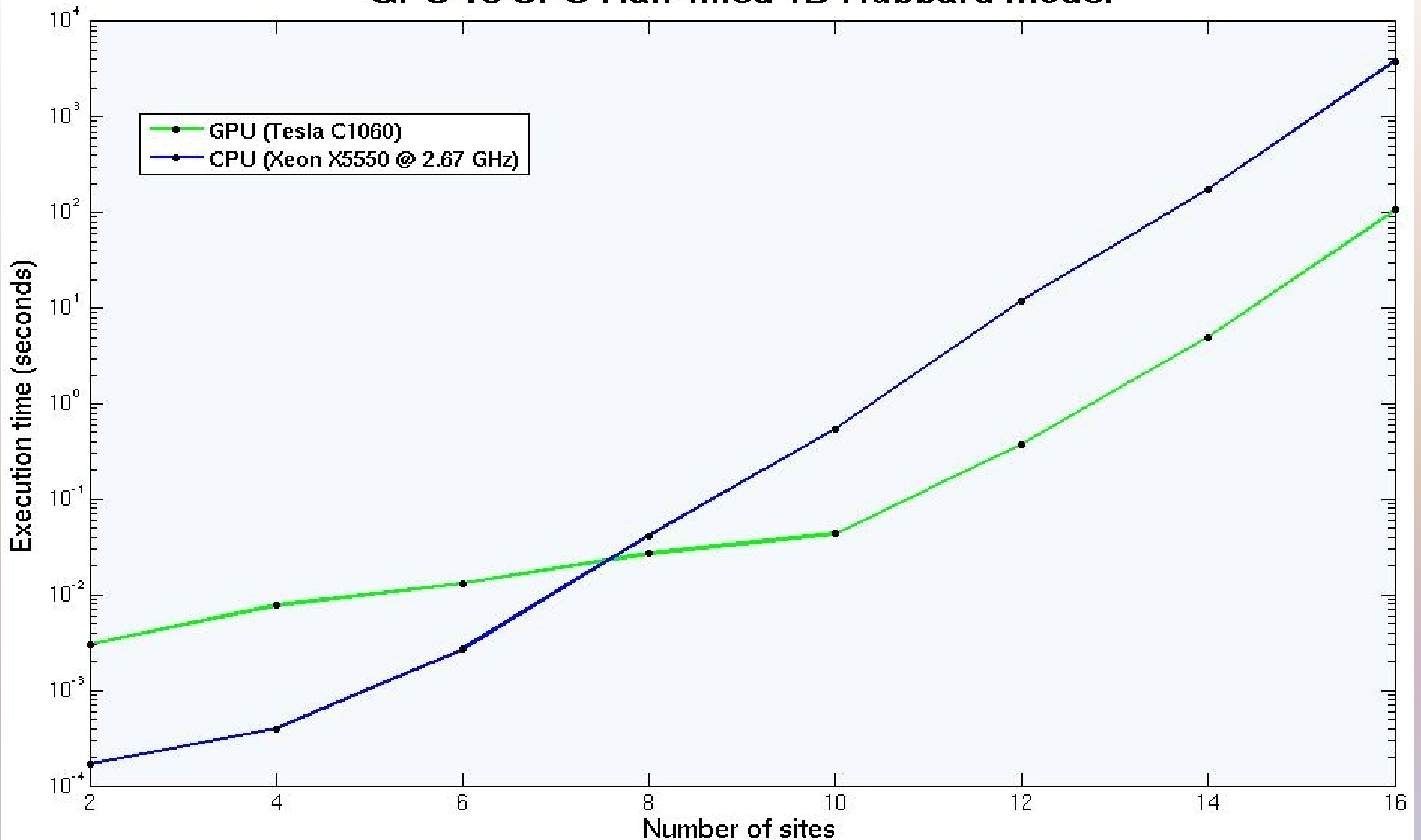
```
__global__ void
spmv_ell_kernel(const int num_rows,
                const int num_cols,
                const int num_cols_per_row,
                const int   * indices,
                const float * data,
                const float * x,
                      float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if(row < num_rows){
        float dot = 0;

        for(int n = 0; n < num_cols_per_row; n++){
            int   col = indices[num_rows * n + row];
            float val = data[num_rows * n + row];

            if(val != 0)
                dot += val * x[col];
        }

        y[row] += dot;
    }
}
```

# Yhteenveto

- Näytönohjaimet ovat nousseet perinteisten prosessorien kilpailijaksi suurteholaskennassa
- Monet fysiikan ongelmat voidaan ratkaista algoritmeilla, jotka ovat hyvin rinnakkaistuvia, mikä on edellytys näytönohjaimen tehokkaalle hyödyntämiselle
- Esimerkiksi vektorin kertominen matriisilla voidaan tehdä erittäin nopeasti näytönohjaimella
- CUDA on NVIDIAn rinnakkaislaskenta-arkkitehtuuri, joka mahdollistaa näytönohjaimella ajettavien funktioiden kirjoittamisen C++:lla

Kiitos!

# Backup




- The GPU surpasses the CPU in speed when the system size becomes large enough to enable large scale parallelization.