

# PyCuda

oftewel massief parallel rekenen

Jasper Spaans, Fox-IT <[j@jasper.es](mailto:j@jasper.es)>

PUN meeting 2009-09-24



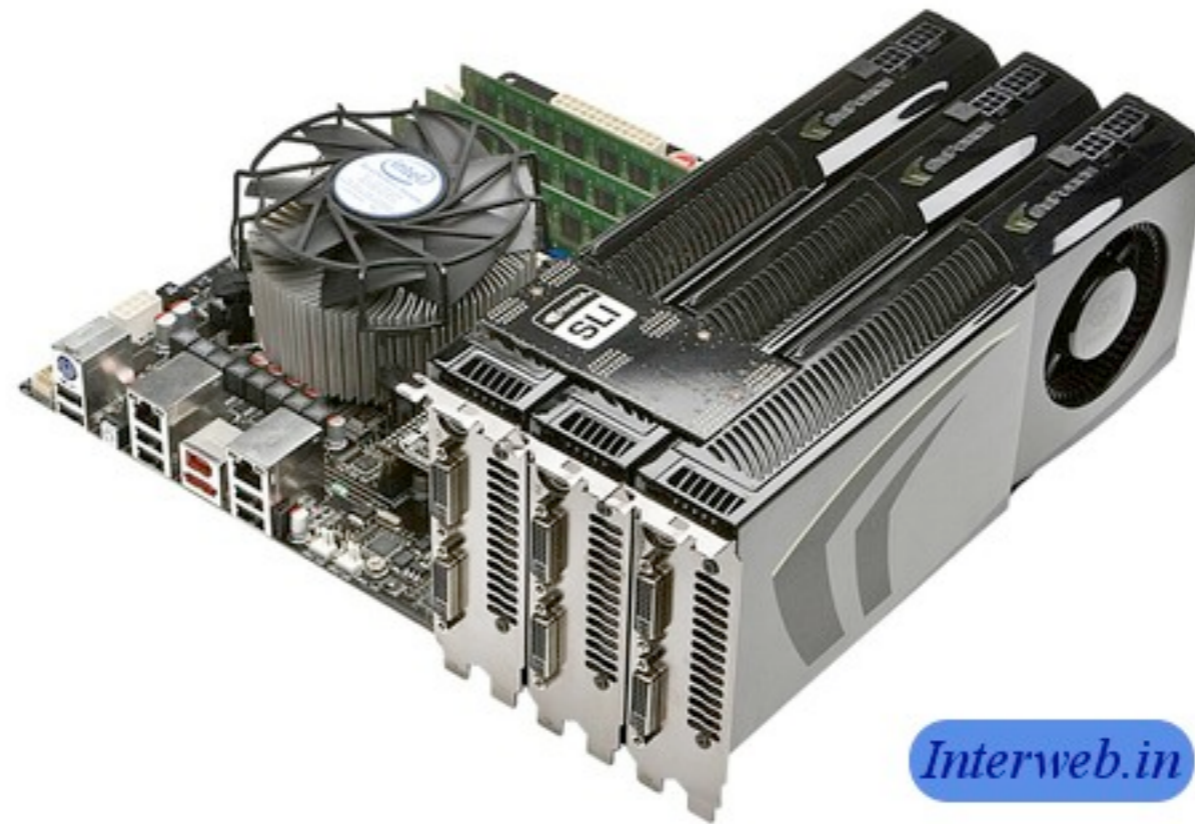
# Cuda

- Techniek om berekingen op je GPU te kunnen uitvoeren (“GPGPU”)



# Cuda

- Techniek om berekingen op je GPU te kunnen uitvoeren (“GPGPU”)



*Interweb.in*

u



# GPGPU vs CPU

CPU / GPU	GFlops <sup>1)</sup>	Prijs <sup>2)</sup>
GeForce GTX 285	1063	€ 300
Radeon HD 4870	1200	€ 140
Intel Core i7-965	102	€ 900

1) Theoretisch maximum, c't 07/2009

2) Prijzen uit pricewatch, 2009-09-20



# Cuda vs OpenCL

Producent	Cuda	Brook	OpenCL
Nvidia	✓	×	✓ / × <sup>1)</sup>
ATI/ AMD	×	✓	✓ <sup>2)</sup>
Intel (Larrabee)	×	×	?

1) Snuipaard heeft support, Linux/Win SDK alleen als beta program voor registered developers (registratie gratis: webform invullen, en dan een weekje wachten)

2) SDK doet nu CPU-only OpenCL, in de toekomst ook op de GPU



# Waarvoor?

- Parallele algoritmes
- Rekenintensief
- “Simpele” problemen
- Bijvoorbeeld:
  - GSM Cracking (A5 / 1), <http://tinyurl.com/lyy3f2> (Cuda)
  - Iterative PCA, <http://arxiv.org/abs/0811.1081>



# Hoe?

- Heel hard rekenen op GPU in Cuda-C++

```
__global__ void vecmul(float *A, float *B, float *C, const int l) {  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < l) {  
        C[i] = A[i] + B[i];  
    }  
}
```



# Hoe?

- Heel hard rekenen op GPU in Cuda-C++

```
__global__ void vecmul(float *A, float *B, float *C, const int l) {  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < l) {  
        C[i] = A[i] + B[i];  
    }  
}
```

- Aanroepen code in C++:

```
M1 = CudaMemAlloc(L * sizeof(float)); // may fail silently  
M2 = CudaMemAlloc(L * sizeof(float)); // may fail silently  
M3 = CudaMemAlloc(L * sizeof(float)); // may fail silently  
CudaMemCpy(M1, A, L * sizeof(float), cudaMemcpyHostToDevice);  
CudaMemCpy(M2, B, L * sizeof(float), cudaMemcpyHostToDevice);  
vecmul<<<256, 0>>> (M1, M2, M3);  
CudaMemCpy(C, M3, L * sizeof(float), cudaMemcpyDeviceToHost);  
CudaFree(M1);  
CudaFree(M2);  
CudaFree(M3);
```



# Hoe?

- Heel hard rekenen op GPU in Cuda-C++

```
__global__ void vecmul(float *A, float *B, float *C, const int l) {  
    const int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < l) {  
        C[i] = A[i] + B[i];  
    }  
}
```

- ~~Aanroepen code in C++:~~

```
M1 = CudaMemAlloc(L * sizeof(float)); // may fail silently  
M2 = CudaMemAlloc(L * sizeof(float)); // may fail silently  
M3 = CudaMemAlloc(L * sizeof(float)); // may fail silently  
CudaMemCpy(M1, A, L * sizeof(float), cudaMemcpyHostToDevice);  
CudaMemCpy(M2, B, L * sizeof(float), cudaMemcpyHostToDevice);  
vecmul<<<256, 0>>(M1, M2, M3);  
CudaMemCpy(C, M3, L * sizeof(float), cudaMemcpyDeviceToHost);  
CudaFree(M1);  
CudaFree(M2);  
CudaFree(M3);
```



# PyCuda fijner!

- Heel hard rekenen op GPU in Cuda-C++

```
kernels = mod = SourceModule("""
__global__ void vecmul(float *A, float *B, float *C, const int l) {
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < l) {
        C[i] = A[i] + B[i];
    }
}
""")
```



# PyCuda fijner!

- Heel hard rekenen op GPU in Cuda-C++

```
kernels = mod = SourceModule("""
__global__ void vecmul(float *A, float *B, float *C, const int l) {
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < l) {
        C[i] = A[i] + B[i];
    }
}
""")
```

- Aanroepen code in PyCuda:

```
vecmul = kernels.get_function("vecmul")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros(a.shape)
vecmul(drv.Out(dest), drv.In(a), drv.In(b), block=(400,1,1))
```



# GPUArray nog fijner!

- Lekker pythonic, geen DRY of stomme boilerplate code voor init/cleanup:

```
# some imports
a = curand((400,))
b = curand((400,))

vecmul = ElementwiseKernel(
    "float *a, float *b, float *c",
    "c[i] = a[i] * b[i]",
    "vecmul")
c_gpu = gpuarray.empty_like(a_gpu)
vecmul(a, b, c_gpu)
```



# Toekomst?

- OpenCL ipv Cuda
- Intel en Sparc: CPUs met enorm veel threads
- (Cuda en OpenCL kernels zijn bijna identiek)
- Als het je boeit: leer parallel denken, oefen met pyCuda/pyOpenCL



# OpenCL

- OpenCL demo!



# URLs

- Slides: <http://jasper.es/talks/pun2009-09/>
- PyCuda: <http://mathematician.de/software/pycuda> (hier leeft ook PyOpenCL)
- PyCuda introduction talk: [http://www.archive.org/details/scipy09\\_advancedTutorial\\_7](http://www.archive.org/details/scipy09_advancedTutorial_7)
- Cuda: [http://www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html)
- OpenCL: <http://www.khronos.org/opencl/>



