Parallel Computing
The GPU story

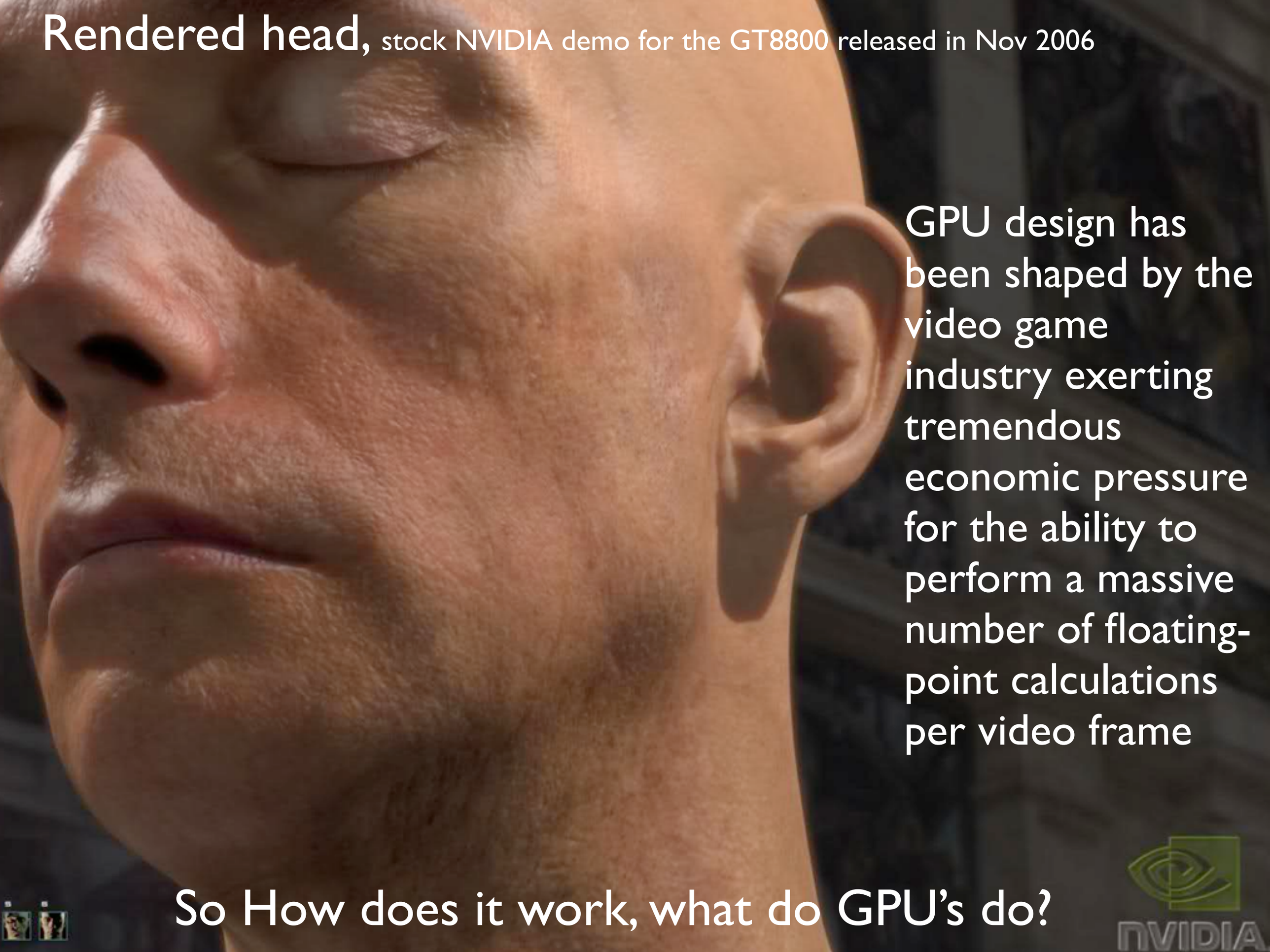
Anthony Morse
anthony.morse@plymouth.ac.uk

http://www.gnd-tech.com/images/imported/2009/04/20.jpg

Why should we care about Graphics Cards for anything other than image processing tasks?

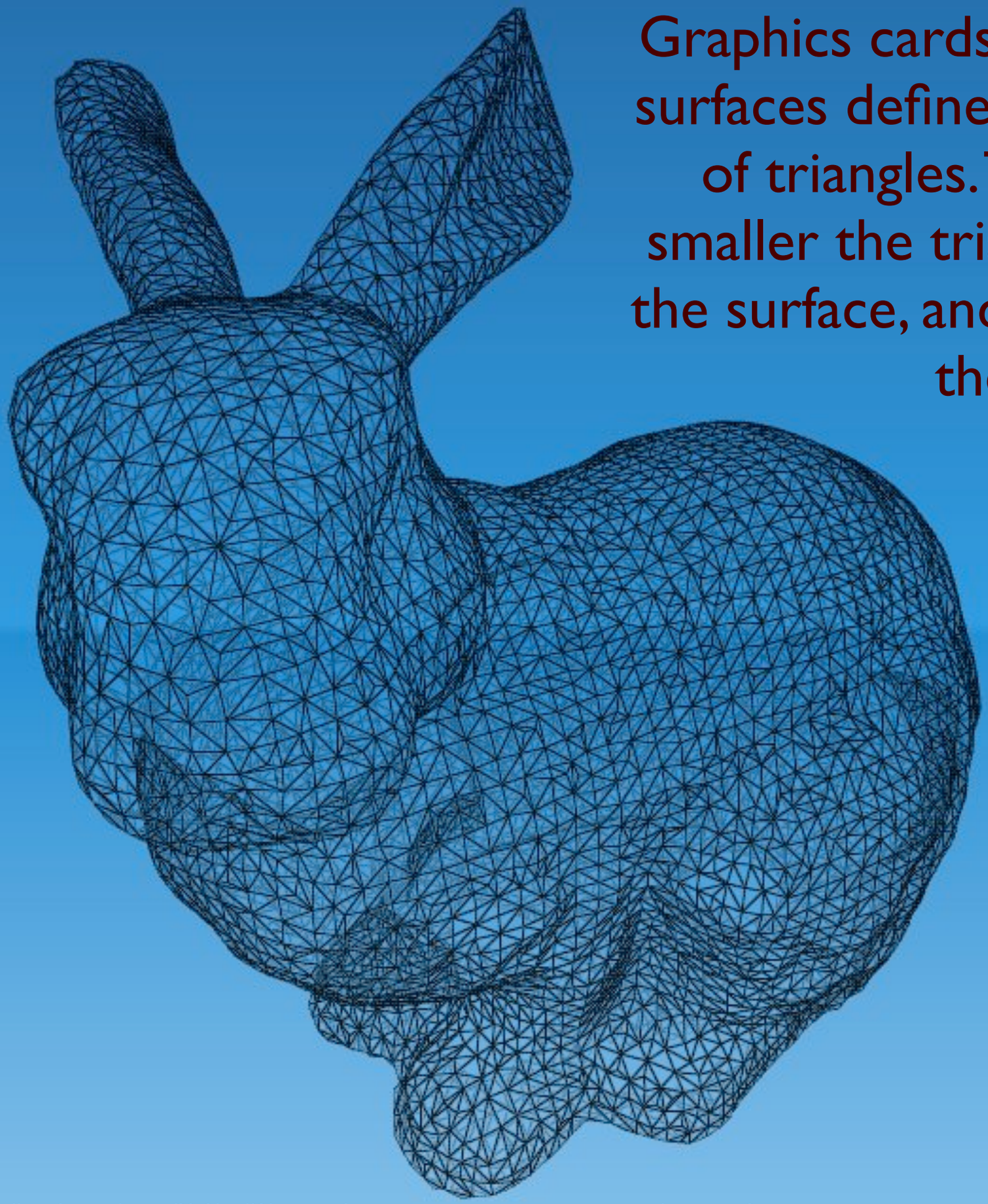
http://www.gnd-tech.com/images/imported/2009/04/20.jpg

Rendered head, stock NVIDIA demo for the GT8800 released in Nov 2006



GPU design has been shaped by the video game industry exerting tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame

So How does it work, what do GPU's do?



Graphics cards are optimized to render surfaces defined by the points (vertices) of triangles. The more vertices, the smaller the triangles, the more detailed the surface, and the higher the quality of the final image.

http://codeflow.org/entries/2012/aug/02/easy-wireframe-display-with-barycentric-coordinates/easy-mesh-2.jpg

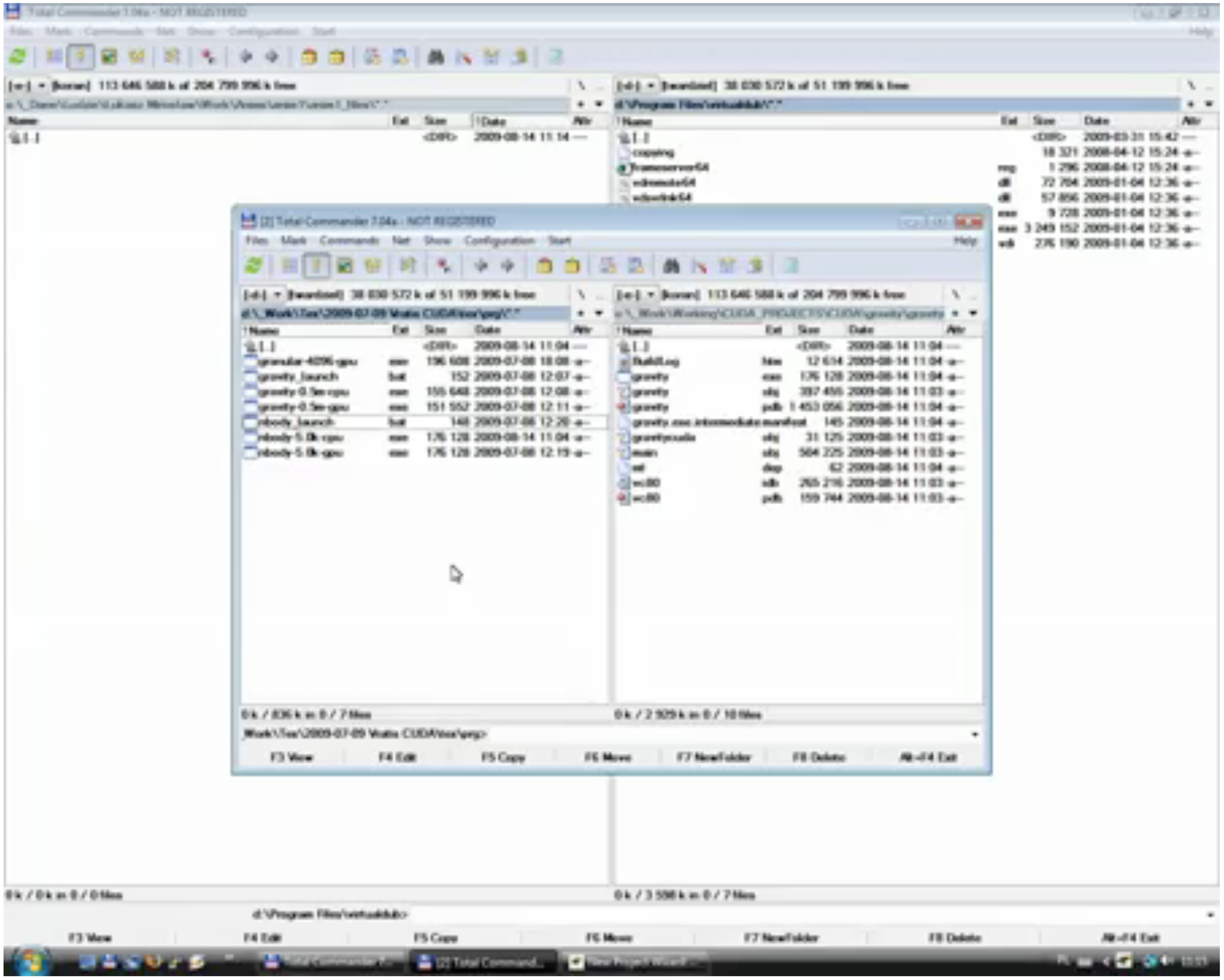
The History of CUDA

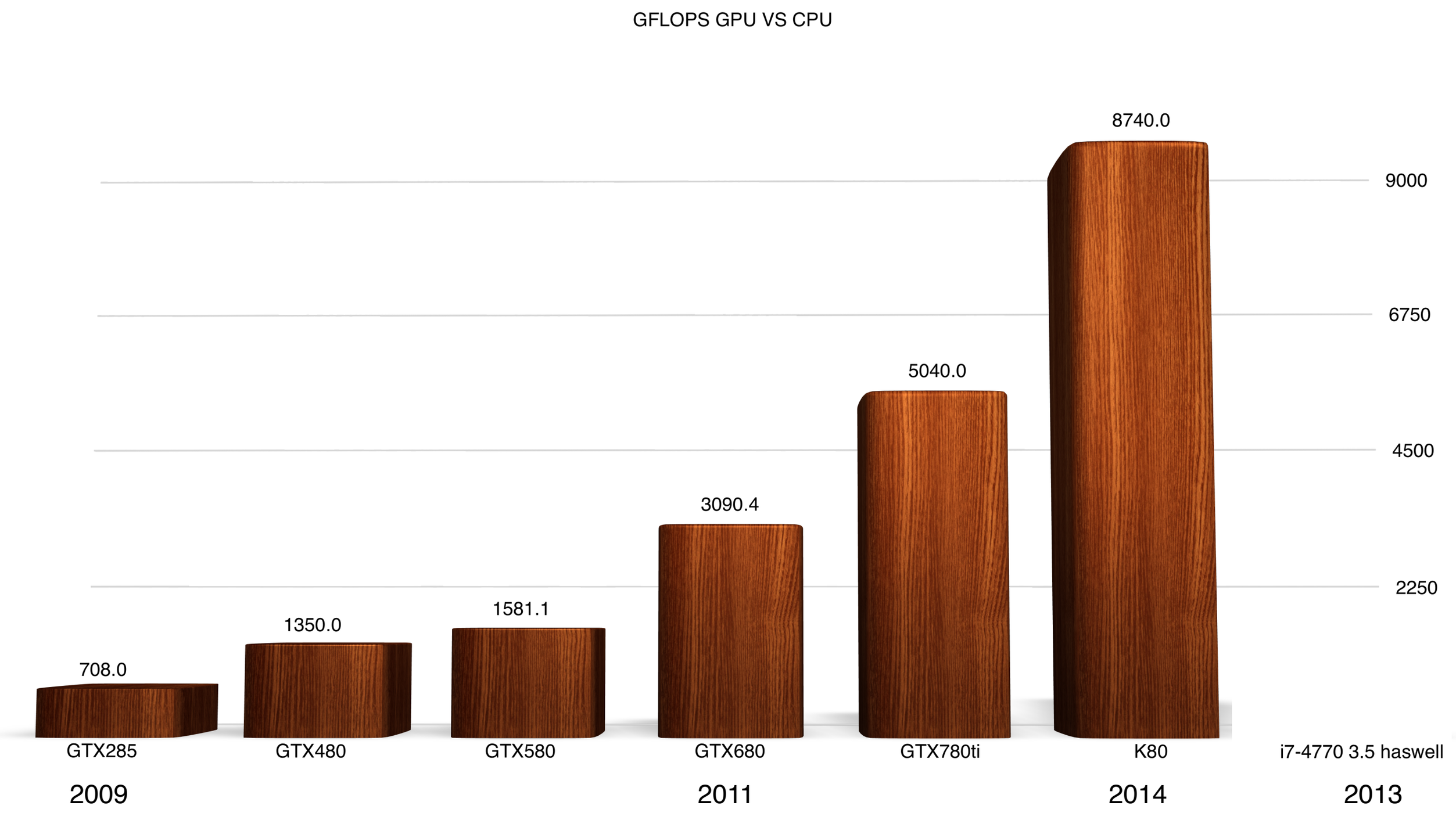
SC08

Conference for High Performance Computing
Austin Texas

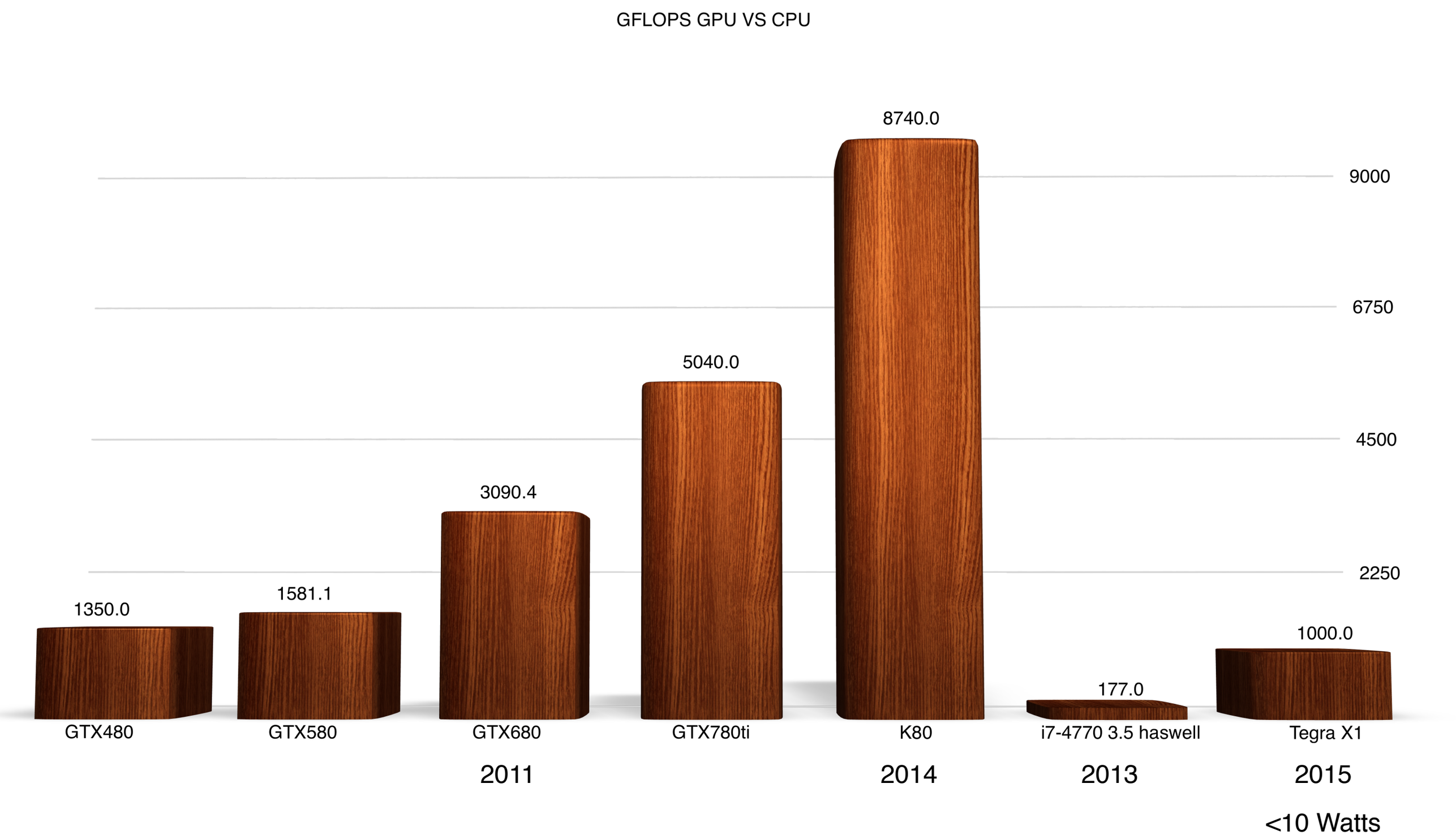
2006 - G80 architecture and CUDA

- General Purpose Programming on the GPU
- C-like language
- Easily get to the power of the GPU... but what power exactly?





FLOPS are not the be all and end all of computing



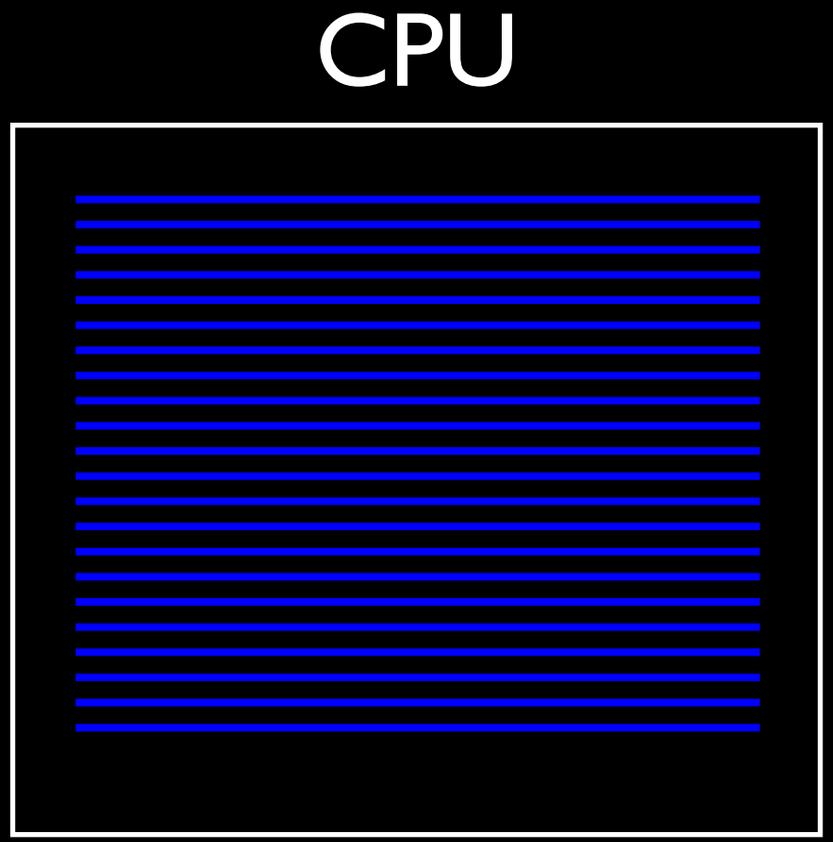
FLOPS are not the be all and end all of computing

GPU Computing is

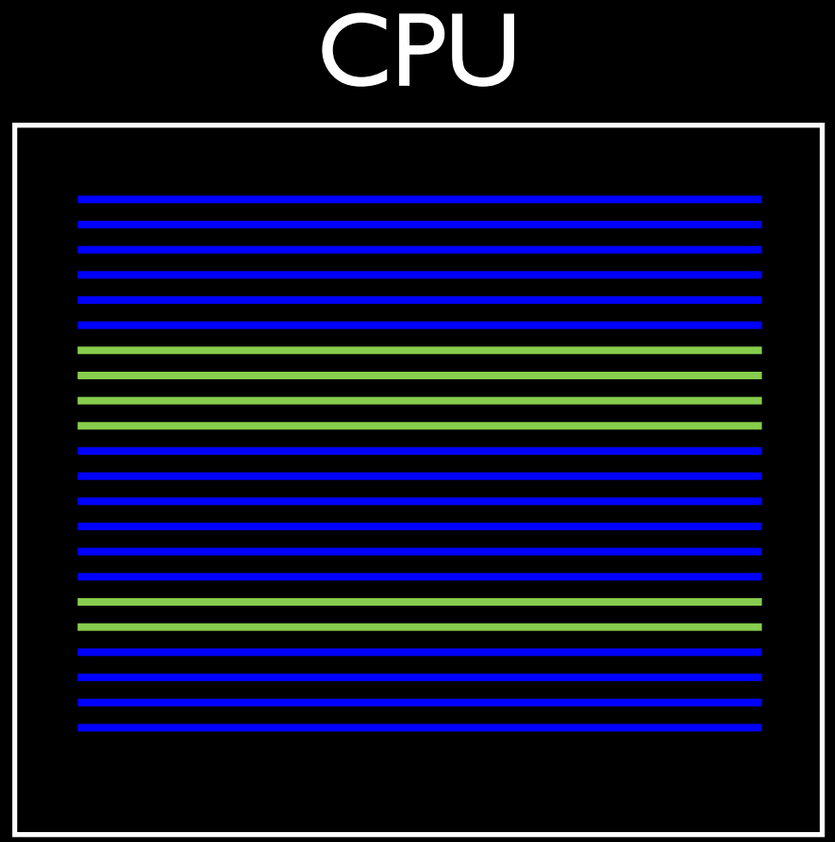
not CPU vs GPU

it is CPU + GPU

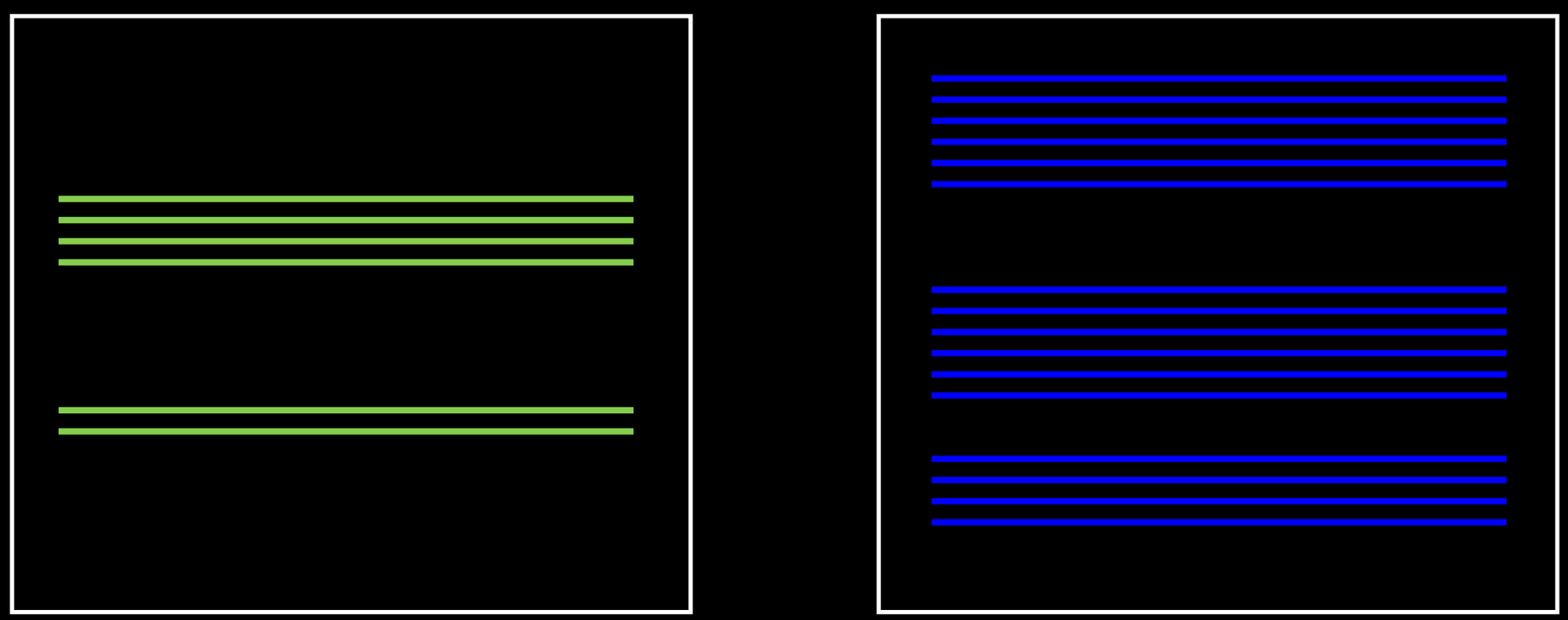
GPU Computing is



GPU Computing is

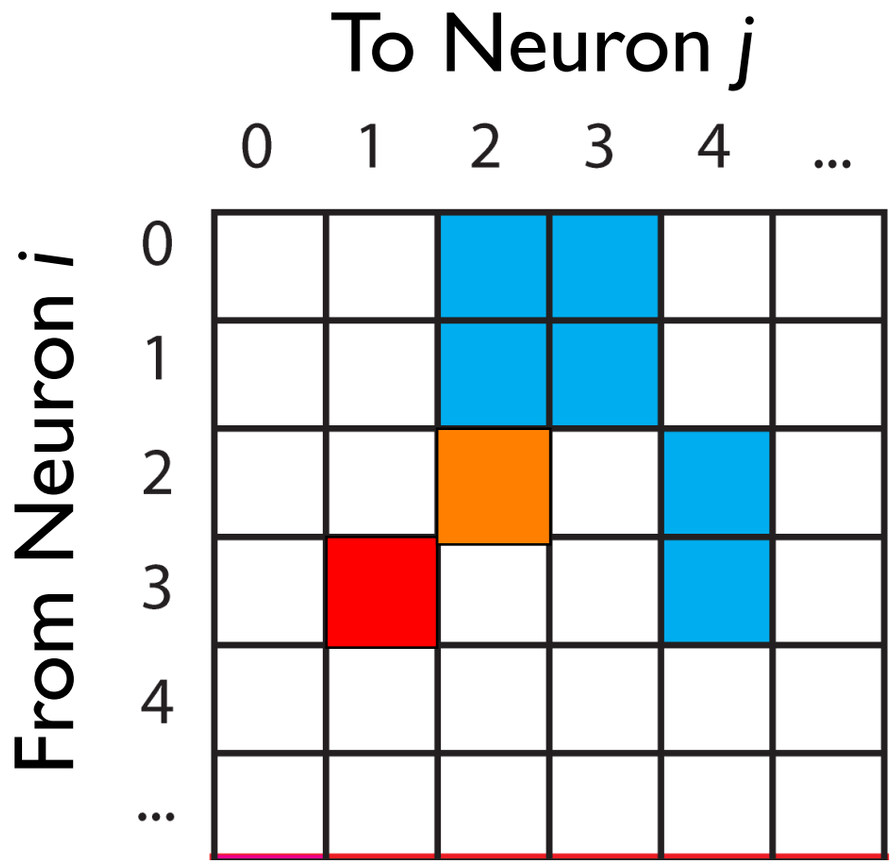


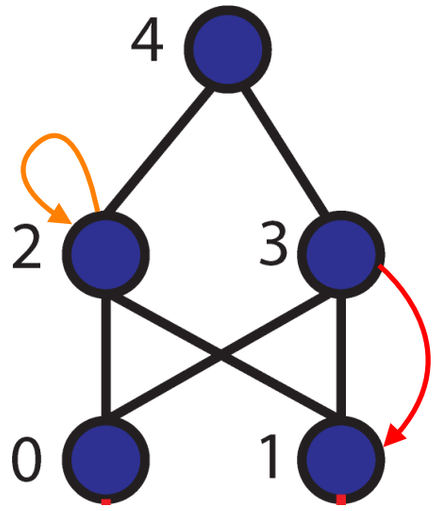
GPU Computing is



Naive CUDA programming example

Simple Neural Network





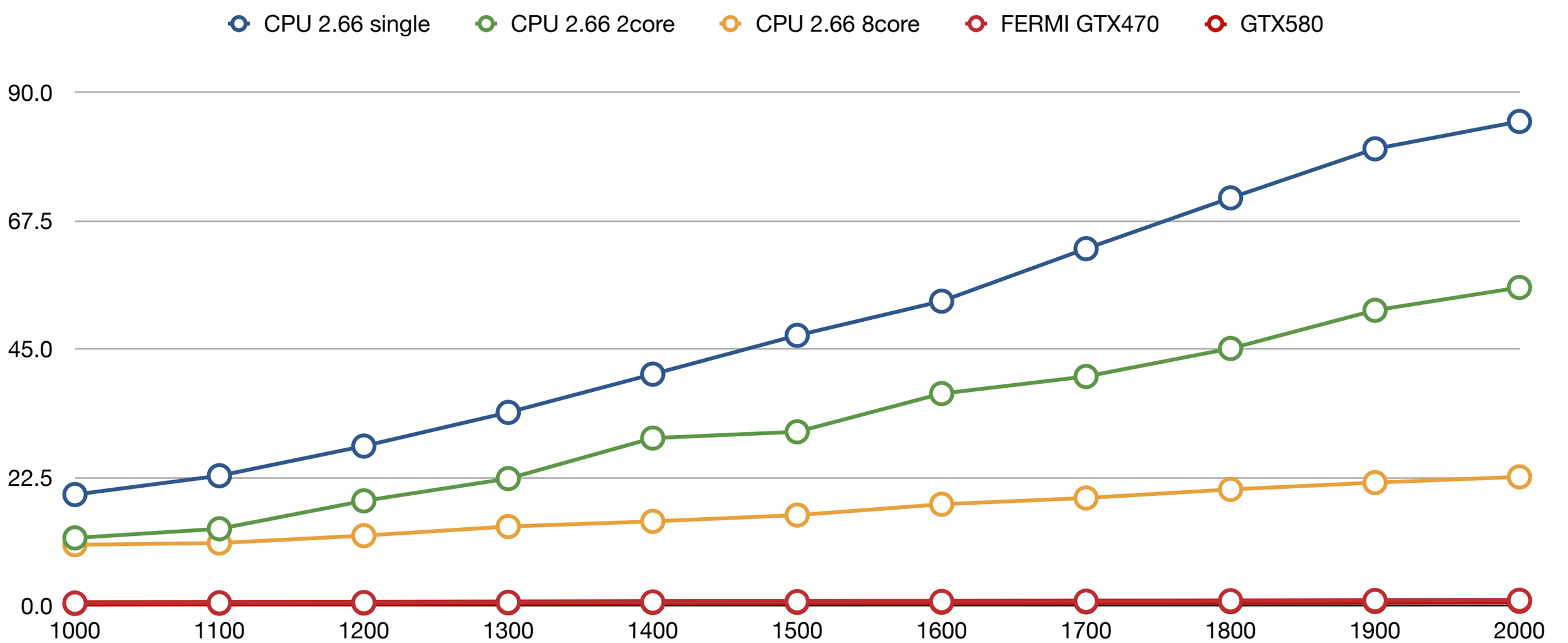
Host Code

```
void NN_OnHost(float *activity, float *weights, int N)
{
  int i,j;
  float new_activity[N];

  for (i=0; i<N; i++) {
  new_activity[i] = 0;
  for(j=0;j<N;j++) {
      new_activity[i] += activity[j] * weights[i + (j * N)];
  }
  }
  for (i=0; i<N; i++) {
  activity[i] = 1/(1+exp(-new_activity[i]));
  }
}
```

Identify data parallel sections of code

CUDA code

```
__global__ void NN_OnDevice(float *activity, float *weights,
                            float *new_activity, int N)
{
   int j, idx = blockIdx.x*blockDim.x + threadIdx.x;
   new_activity[idx] = 0;
   for(j=0;j<N;j++) {
      new_activity[idx] += activity[j] * weights[idx + (j * N)];
   }
   __syncthreads();
   activity[idx] = 1/(1+exp(-new_activity[idx]));
}
```



	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
CPU 2.66 single	19.5	22.8	28.0	33.9	40.6	47.4	53.4	62.6	71.5	80.1	84.9
CPU 2.66 2core	11.9	13.5	18.4	22.3	29.4	30.5	37.2	40.2	45.1	51.8	55.8
CPU 2.66 8core	10.7	11.0	12.3	13.9	14.8	15.9	17.8	18.9	20.4	21.6	22.6
FERMI GTX470	0.48	0.53	0.57	0.62	0.67	0.73	0.76	0.81	0.89	0.93	0.97
GTX580	0.42	0.46	0.50	0.54	0.59	0.62	0.64	0.70	0.73	0.78	0.81

	Intel Zeon 2.66Ghz	using 1 core	85 ms	x1
		using 8 core (OpenMP)	22 ms	x 3.8
	Geforce 9600M	32 cores	5.5 ms	x15.5
	Geforce 8600GT	32 cores	3.6 ms	x23.6
	GTX285	240 cores	1.2 ms	x70.8
	Tesla C1060	240 cores	1.2 ms	x70.8
	Fermi GTX470	440 cores	0.9 ms	x94.4
	GTX580	540 cores	0.8 ms	x105

Room for improvement

- Use a register for intermediate results (new_activity) rather than global memory
- Split the work for each neuron across multiple threads - Like parallel reduction
- Use Texture memory (twice the bandwidth of global memory)

```
Running on GeForce GTX 980

Hebbian Neural Network test on GPU : 2000 neurons

kernel execution time = 0.096224 ms
```

- Thats 882.31 times faster than the original single thread CPU code!

```
Running on GeForce GTX 980

Hebbian Neural Network test on GPU : 10240 neurons

kernel execution time = 0.665504 ms
```

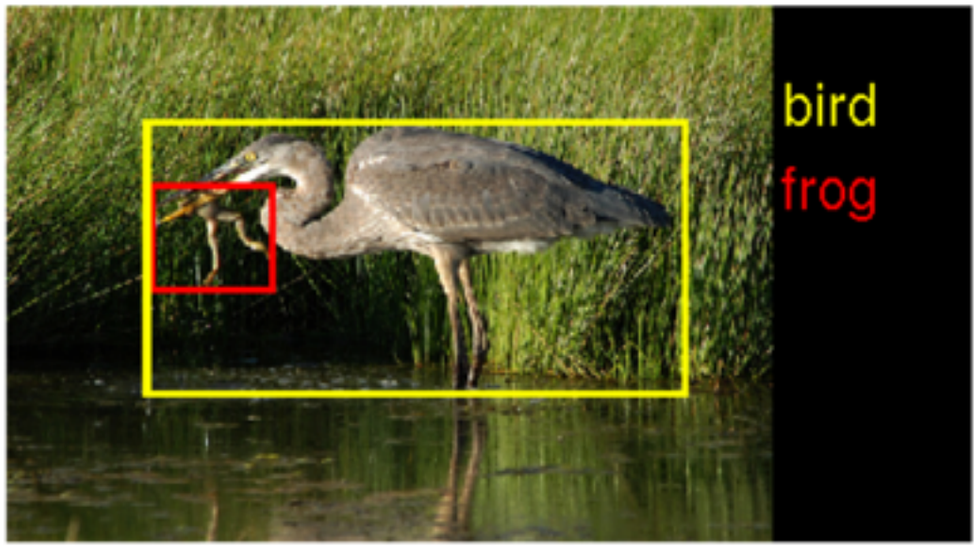
Thats running over 10,000 neurons !!!! and it’s 129 times faster than the single thread CPU code running only 2000 neurons

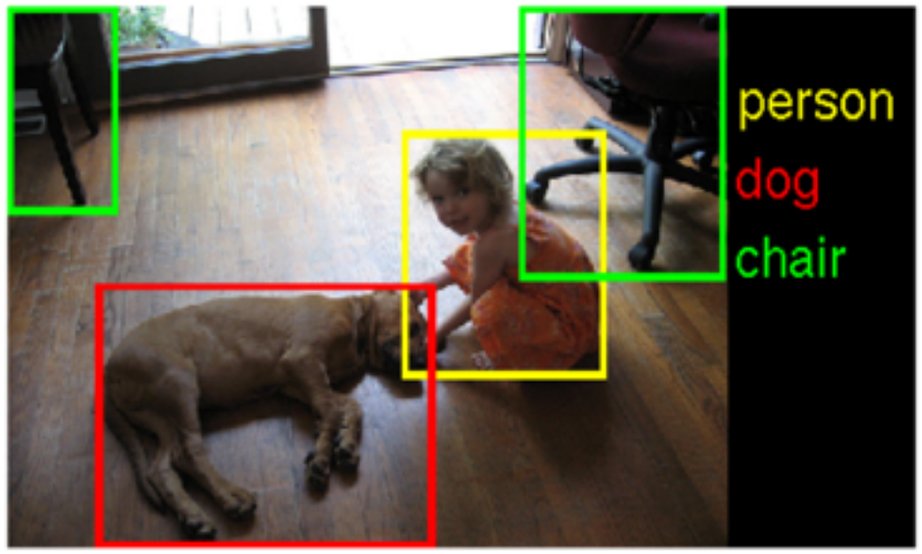
```
Running on Tesla K40c

Hebbian Neural Network test on GPU : 102400 neurons

kernel execution time = 118.299133 ms
```

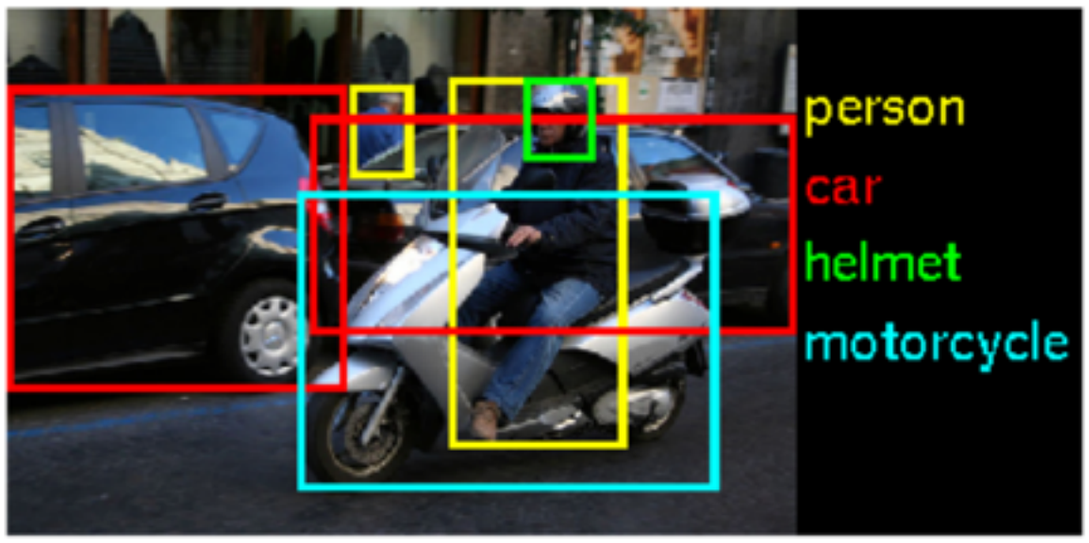
Or we can run over 100,000 neurons !!!! and it’s not much slower (0.73 times) than the single thread CPU code running only 2000 neurons

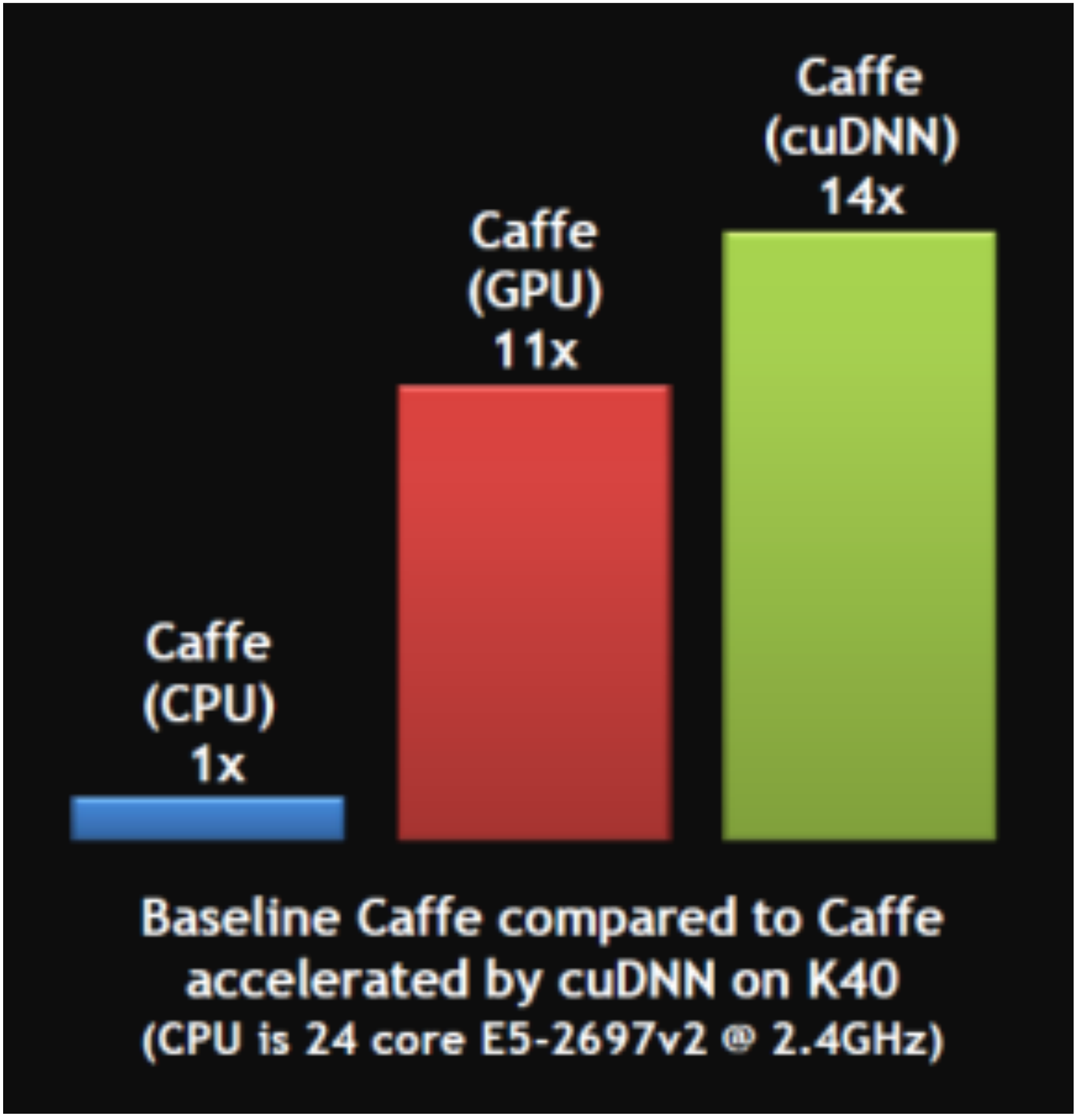
```
__global__ void Spread_OnDevice(float *activity_in, float *activity_out, float *weights)
{
  int j;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;      //idx points to the neuron we are summing for A
  int start = threadIdx.y * (N/blockDim.y);             //the start of the section to be summed by this thread
  int end = start + (N/blockDim.y);                     //the end of the section to be summed by this thread
  extern __shared__ float sdata[];                      // = blockDim.x(portion of N) * blockDim.y(the divsion of sections)

  if(idx < N && end-1 < N)
  {
     float resultA = 0;                          //this will be stored in a register (faster than shared memory!)
     for(j = start; j < end; j++)
     {
        resultA += activity_in[j] * weights[(j*N) + idx];
     }
     //store temporary results
     sdata[(threadIdx.y * blockDim.x) + threadIdx.x] = resultA;

     __syncthreads();

     //Sum over the partial results and update global with sigmoid results
     if(threadIdx.y == 0)
     {
        for(j = 1; j < blockDim.y; j++)              //sum the results of each section (starts from 1 as result is already sdata[0])
        {
           resultA += sdata[(j * blockDim.x) + threadIdx.x];
        }
        activity_out[idx] = 1/(1+__expf(-resultA));
     }
  }
}
```

Example ILSVRC2014 images:








Caffe
(CPU)
1x
Caffe
(GPU)
11x
Caffe
(cuDNN)
14x
Baseline Caffe compared to Caffe
accelerated by cuDNN on K40
(CPU is 24 core E5-2697v2 @ 2.4GHz)

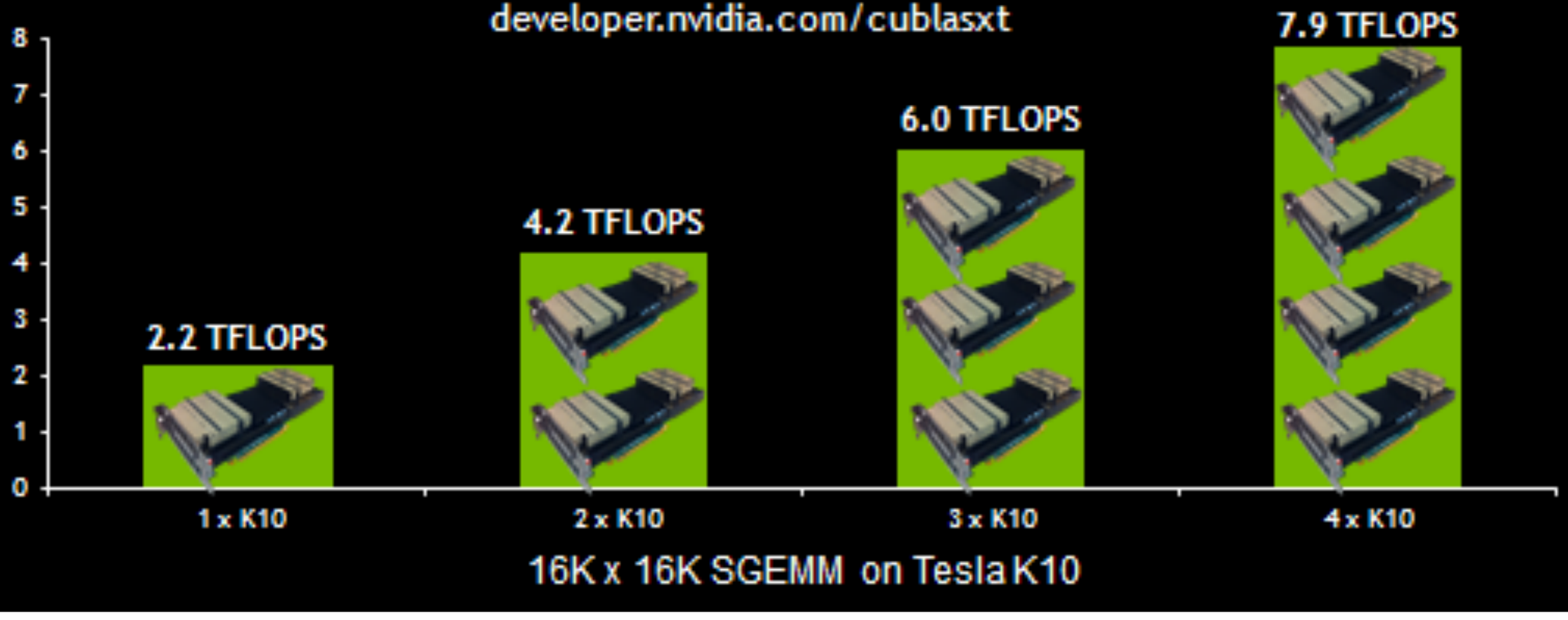
Extended (XT) Library Interfaces

Automatic Scaling to multiple GPUs per node

cuFFT 2D/3D & cuBLAS level 3

Operate directly on large datasets that reside in CPU memory

developer.nvidia.com/cublasxt



cuBLAS ~ 5x faster than BLAS (on one GPU)