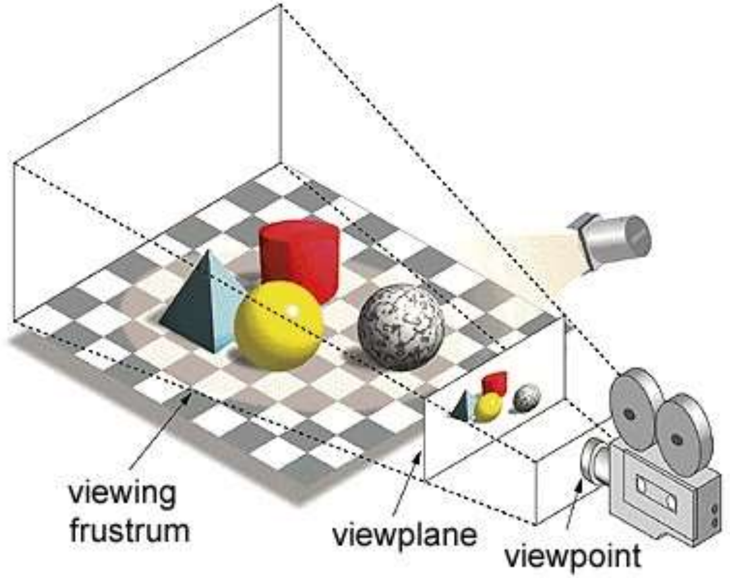
Σπυριδωνίδης Δημήτρης

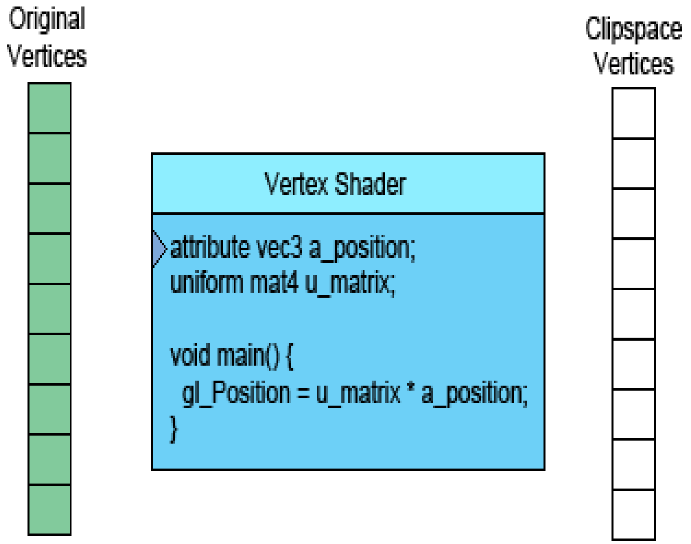
GPU ACCELERATION IN MACHINE LEARNING

From a graphics processing unit to a unparallel fast machine learning processor

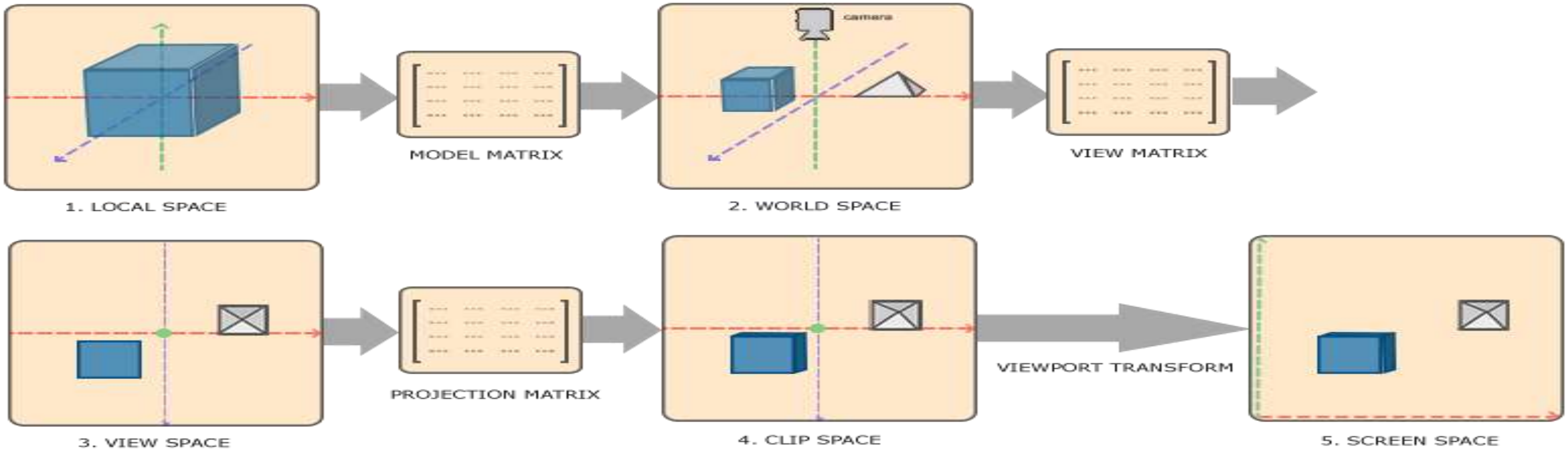
Computer Graphics

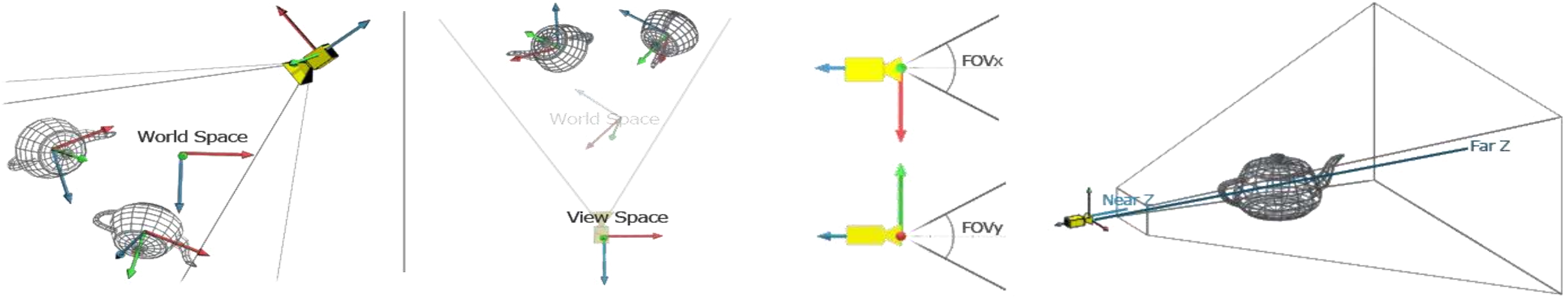
- In this section we will discuss the math behind computer graphics and what it takes to have a 3D model projected in a 2d screen
- How such computations can be accelerated on specialized hardware such as a GPU and the architecture difference between the GPU and the CPU



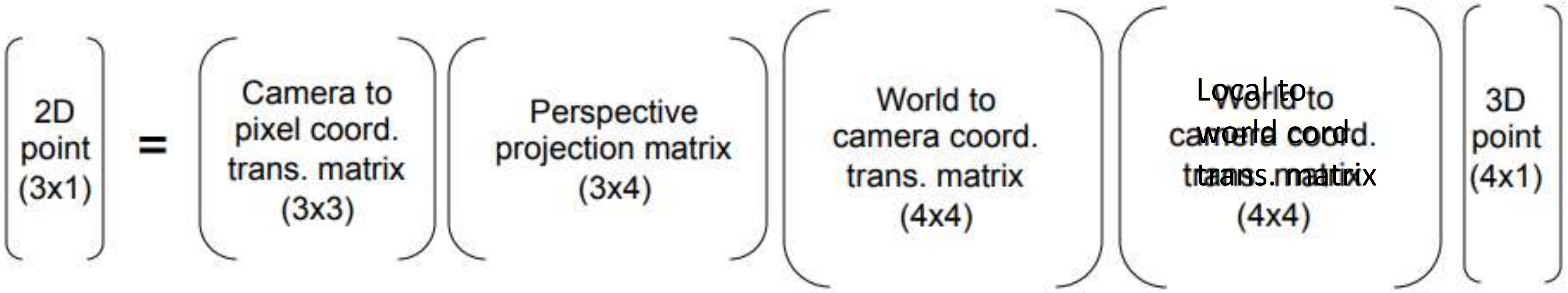


Each model in a scene is made from 3D point "vertices" , those vertices make triangles than make the surface of the object. To be able to see the 3D model on our 2D screen each vertex of the object has to go under a transformation. First the it has to be moved from it's local space to the world space, then from the world space to the view space and finally to be projected on a 2d plane This process has to be done for every vertex of our model and for every model in the scene.


camera
MODEL MATRIX
VIEW MATRIX
1. LOCAL SPACE
2. WORLD SPACE
PROJECTION MATRIX
VIEWPORT TRANSFORM
3. VIEW SPACE
4. CLIP SPACE
5. SCREEN SPACE

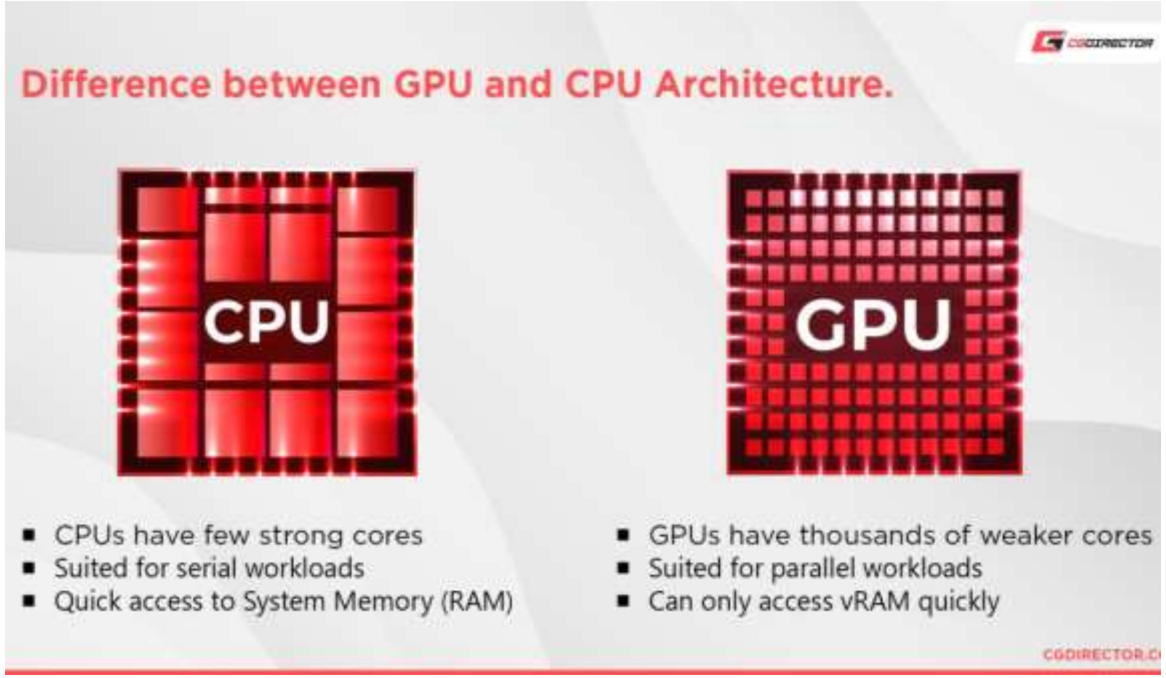

World Space
World Space
View Space
FOVx
FOVy
Near Z
Far Z

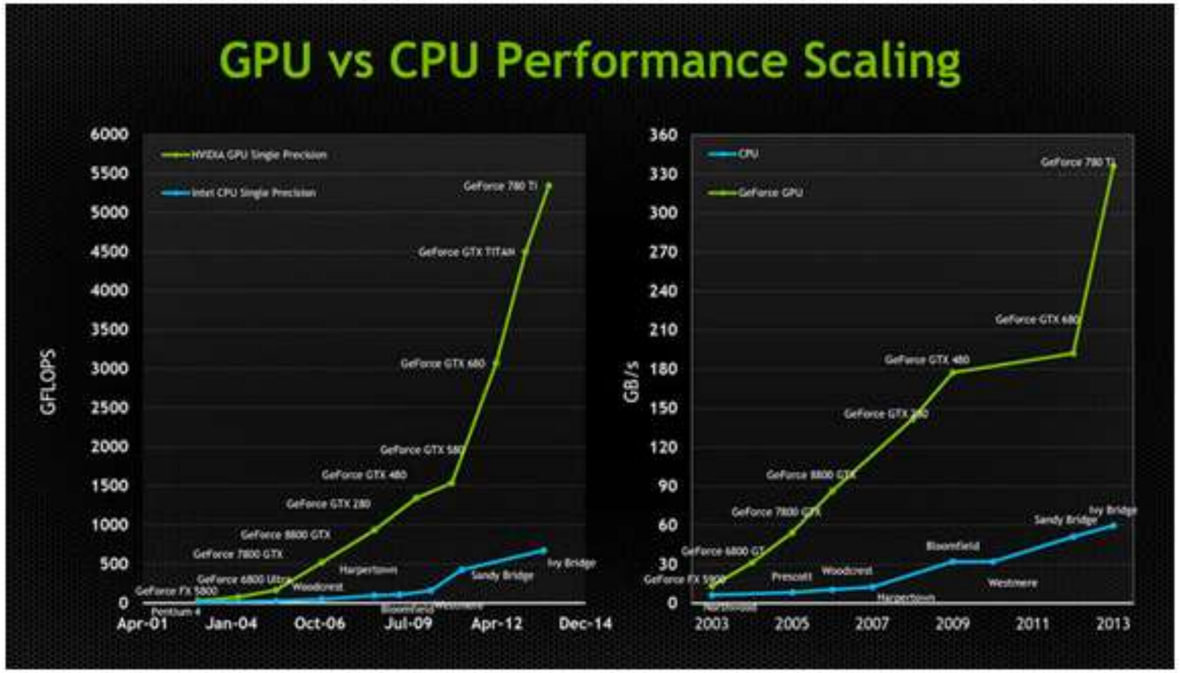
Each transformation is done by multiplying our 3D cord(vertex) by a matrix. The get the final screen coordinates we have to multiplying our original 3D cord by 3 matrices one for each transformation.



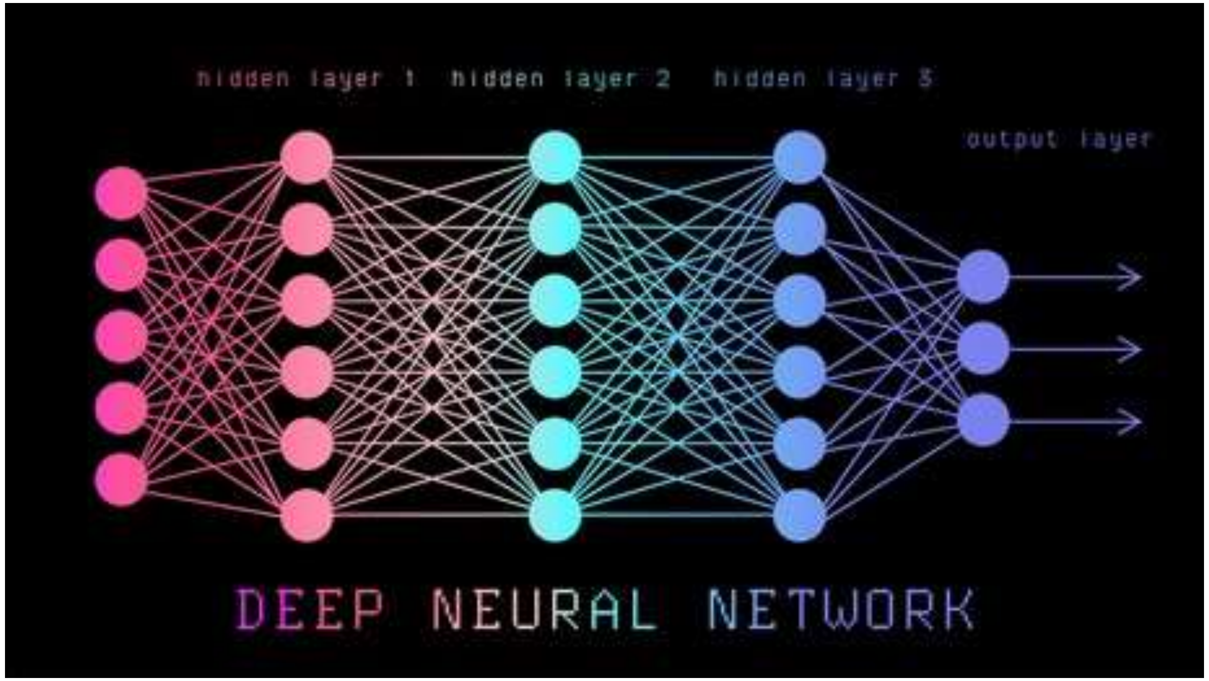
As video games evolved over the year the number of polygons increased so we can have a more detailed world. Modern video game characters have over 100 of thousands of polygons so to render a hole scene our processor has to render millions of triangles every frame

As the popularity of video games rose so did the demand for more detailed and more beautiful scenes. Because every triangle can be rendered independently from the others and a CPU as a serial based processor could not keep up with the rising computational demand of video games , new processor architecture was developed that was specialized in doing a lot simple linear algebra computations in parallel named GPU.





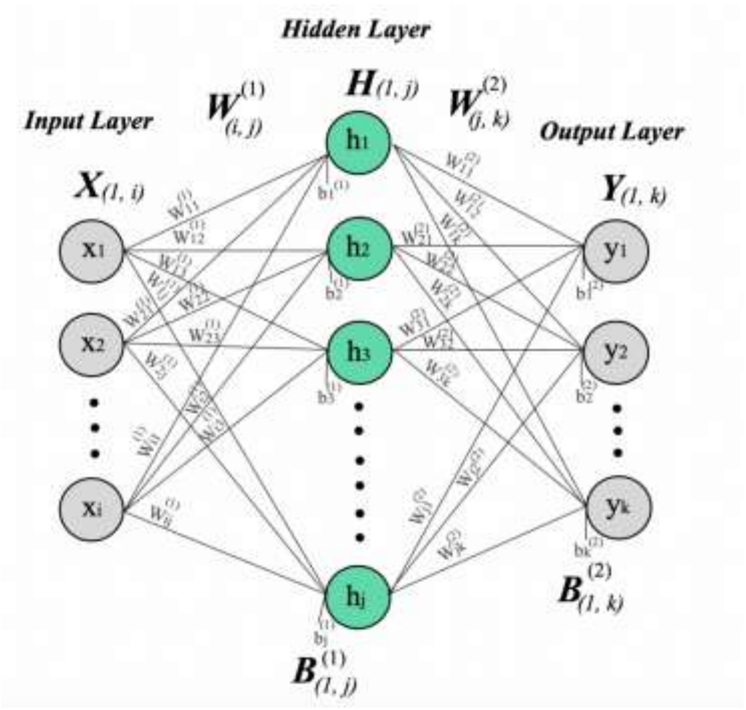
Neural Networks and the GPU

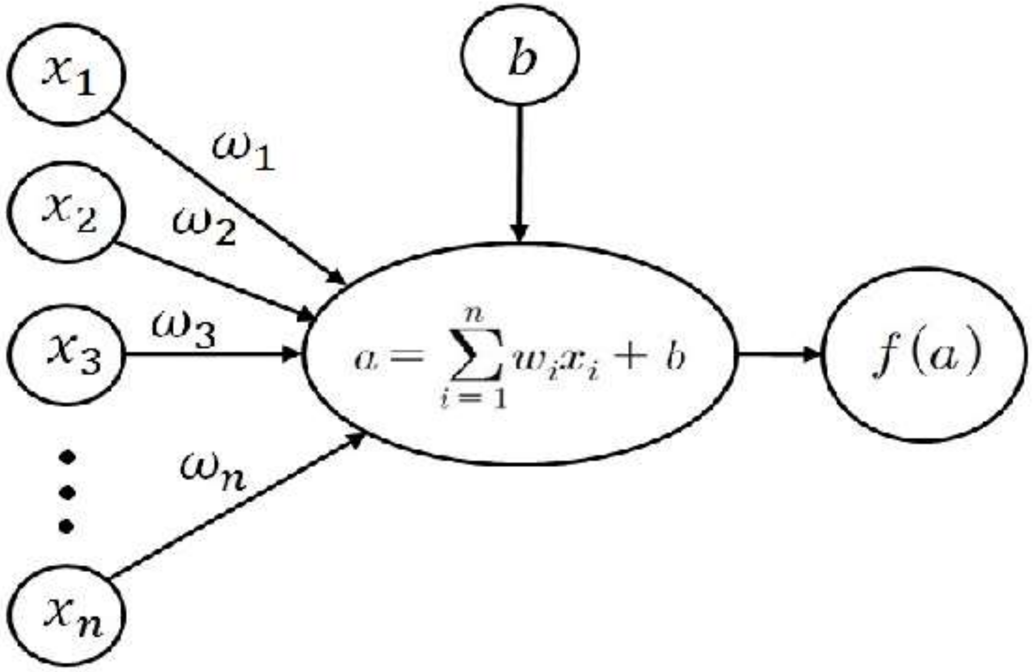


As advancement's in machine learning were made and neural networks became more and more popular it was obvious that the GPU is the perfect processor to run the computations that are needed to train a neural network that are highly parallelizable liner algebra operations as did the video games for witch it was originally developed.
There are several tools to write code in the GPU as openACC, openCL and CUDA.
Today we are going to focus on how accelerate a neural network training using CUDA. But first we are going to see the math behind neural networks and how to run one on the cpu.

The math behind neural networks

All the operations behind training a neural networks are matrix multiplications for this example we are going to use a neural network that has 784 neurons as input, 1000 neurons as a hidden layer an 10 output neurons. First we are going to analyse and demonstrate the math behind our neural network





$O^L = \sigma(W^L \cdot O^{L-1} + b^{L-1}) = \sigma(Z^L)$ $\qquad \sigma = \text{sigmoid}$

784 input

1000 neurons hidden layer

10 neurons output

$Z1_{1000\times1} = WL1_{1000\times784} \cdot Input_{784\times1} + b1_{1000\times1}$

$OL1_{1000\times1} = \sigma(Z1)$

forwardpass

$Z2_{10\times1} = WL2_{10\times1000} \cdot OL1_{1000\times1} + b2_{10\times1}$

$OL2_{10\times1} = \sigma(Z2)$

$Cost = \sum_{i=0}^{10} (OL2i - target\,i)^2$

$W_L = W_L - \alpha \cdot \underbrace{\frac{\partial C}{\partial W_L}}_{\Delta W_L}$

$\Delta W_L = \frac{\partial C}{\partial O_L} \odot \frac{\partial \sigma(Z_L)}{\partial Z_L} \cdot \frac{\partial Z_L}{\partial W_L}$

backward pass

$\Delta W_L = \left\{ \frac{\partial C}{\partial O_L} \odot \frac{\partial \sigma(Z_L)}{\partial Z_L} \right\} \cdot O_{L-1}^T$

$\Delta W_L = \delta_L \cdot O_{L-1}^T$

last layer

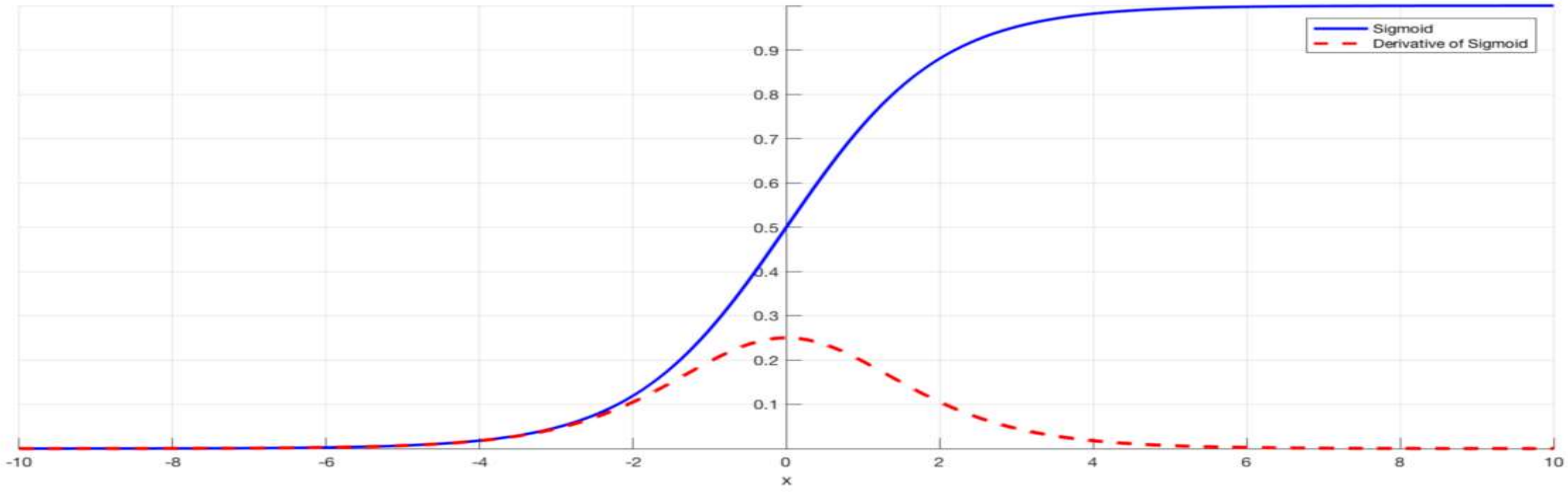
$\delta_L = \frac{\partial C}{\partial O_L} \odot \frac{\partial \sigma}{\partial Z_L}$

$\delta_L = \left((W^{L+1})^T \cdot \delta^{L+1} \right) \odot \frac{\partial \sigma}{\partial z^l}$

$\Delta WL2_{10\times1000} = \underbrace{\left\{ \frac{\partial C}{\partial OL2_{10\times1}} \odot \frac{\partial \sigma(Z2)}{\partial Z2_{10\times1}} \right\}}_{\delta_2} \cdot OL1^T_{1\times1000}$

$\Delta WL1_{1000\times784} = \left(WL2^T_{1000\times10} \cdot \delta 2_{10\times1} \right)_{1000\times1} \odot \frac{\partial \sigma(Z1)}{\partial Z1}_{1000\times1} \cdot Input^T_{1\times784}$

For this example we are going to use the mnist dataset and train the network for 60000 epochs and see how much time it takes, one important note is because the changes that we do in weights depend on the derivative of the sigmoid which is 0 when the sigmoid is close to 0 or 1 so the weights don't change. To overcome this problem we add a small value to the derivative of the sigmoid.



The calculation for our network are done by this 3 functions

```
void activateNN(float* Vector){
  #pragma omp parallel for
    for (int i = 0; i < L1; i++) {
    OL1[i] = 0;
    for (int y = 0; y < N; y++)
      OL1[i] += WL1[i][y] * Vector[y];
    OL1[i] += WL1[i][N];
    OL1[i] = activation_Sigmoid(OL1[i]);
  }

  #pragma omp parallel for
    for (int i = 0; i < L2; i++) {
    OL2[i] = 0;
    for (int y = 0; y < L1; y++)
      OL2[i] += WL2[i][y] * OL1[y];
    OL2[i] += WL2[i][L1];
    OL2[i] = activation_Sigmoid(OL2[i]);
  }

}
```

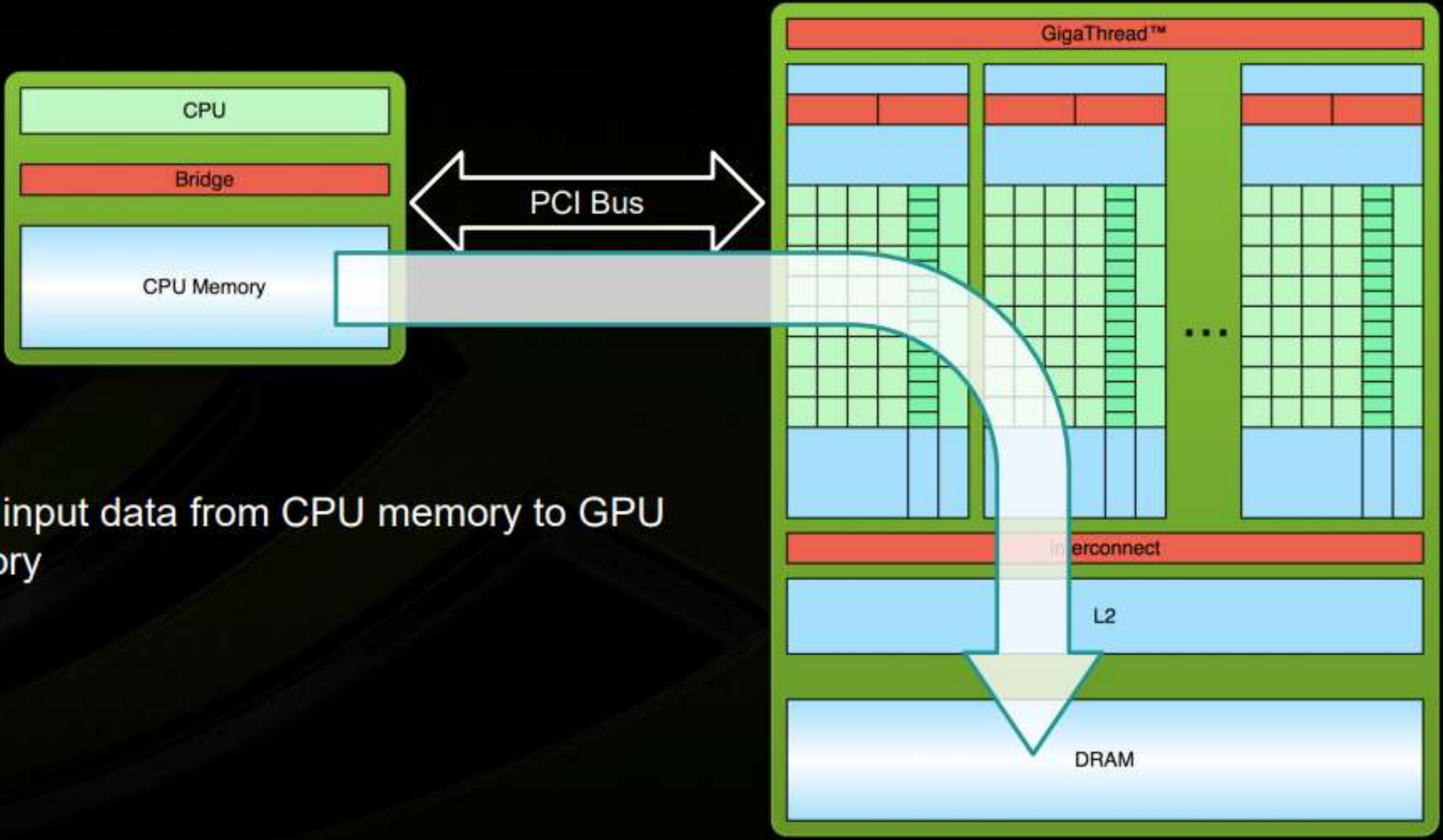
```
void trainNN(float* input, float* target)
{
  #pragma omp parallel for
  for (int i = 0; i <L2 ; i++) {
    for (int j = 0; j < L1; j++) {
      WL2[i][j] -= a * EL2[i] * OL1[j];
    }
    WL2[i][L1] -= a * EL2[i];
  }
  #pragma omp parallel for
  for (int i = 0; i < L1; i++) {
    for (int j = 0; j < N; j++) {
      WL1[i][j] -= a * EL1[i] * input[j];
    }
    WL1[i][N] -= a * EL1[i];
  }

}
```

```
void calc_Error(float *target) {
 #no reason for parallelization we lose time
 for (int i = 0; i < L2; i++) {
    EL2[i] = (OL2[i] - target[i]) * (derivative_Sigmoid(OL2[i])+a);
  }
  #pragma omp parallel for
  for (int i = 0; i < L1; i++) {
    EL1[i] = 0;
    for (int i2 = 0; i2 < L2; i2++) {
      EL1[i] += EL2[i2] * WL2[i2][i] * (derivative_Sigmoid(OL1[i])+a);
    }
  }
}
```

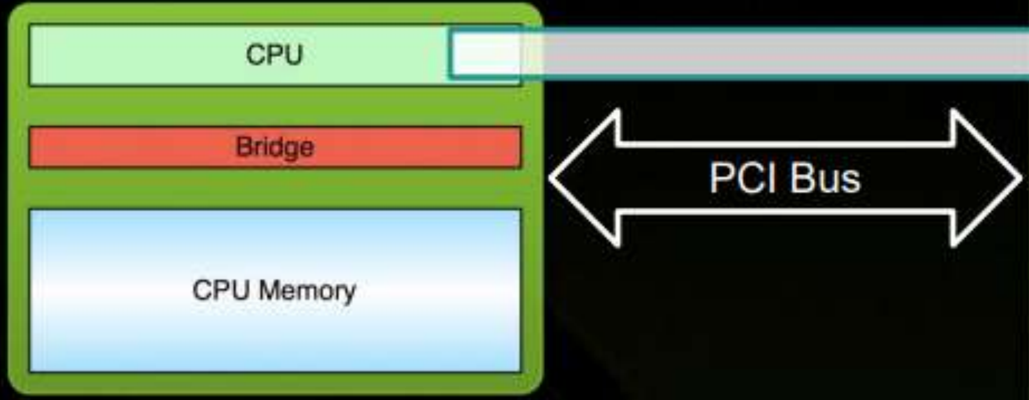
We run our code on google colab. Time for 60000 epochs no optimization 6 min 22 sec. We run the code a second time now taking advantage of vectorization and using both threads of the CPU. Time for 60000 epochs -O3 optimization plus using 2 threads 53sec 7.2x faster. Now that we have made our neural network as fast as possible on the CPU it’s time to try to run it on the GPU. First we are going to make a fast intro to CUDA

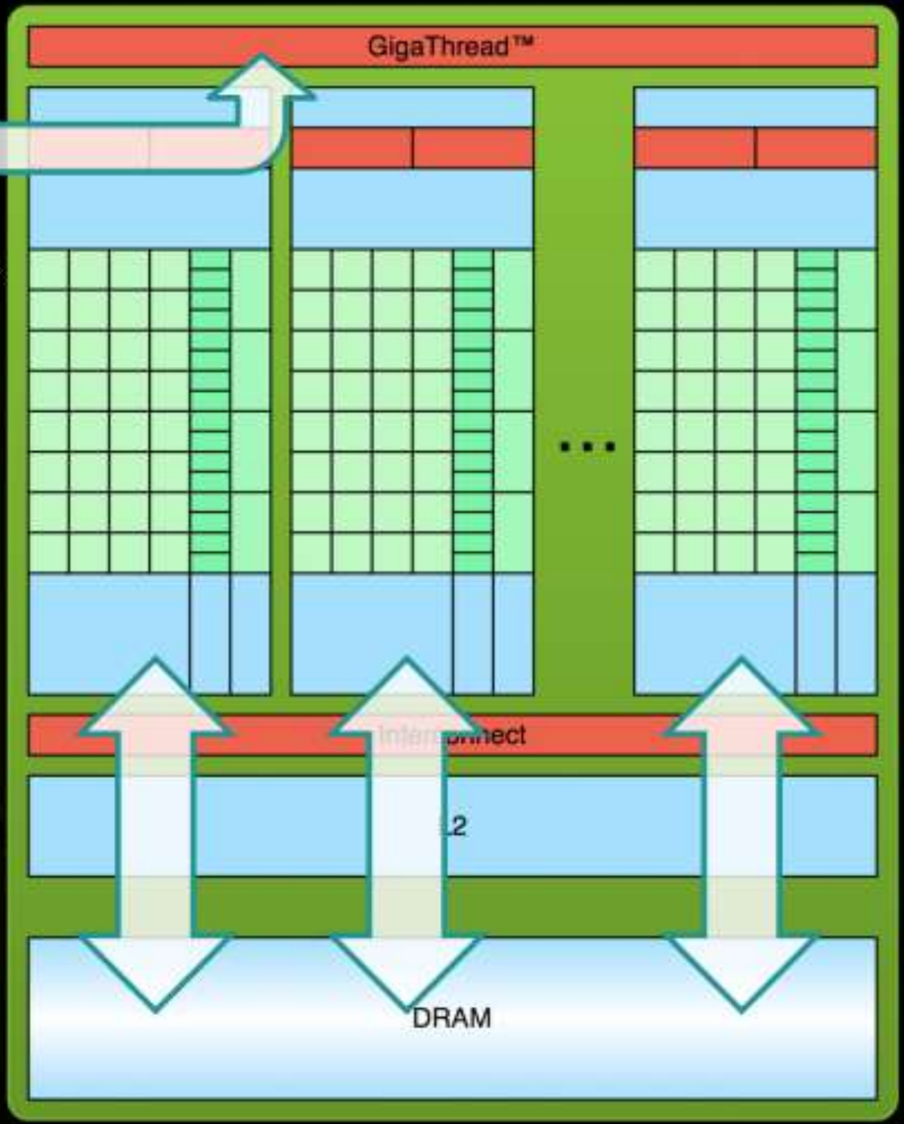
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

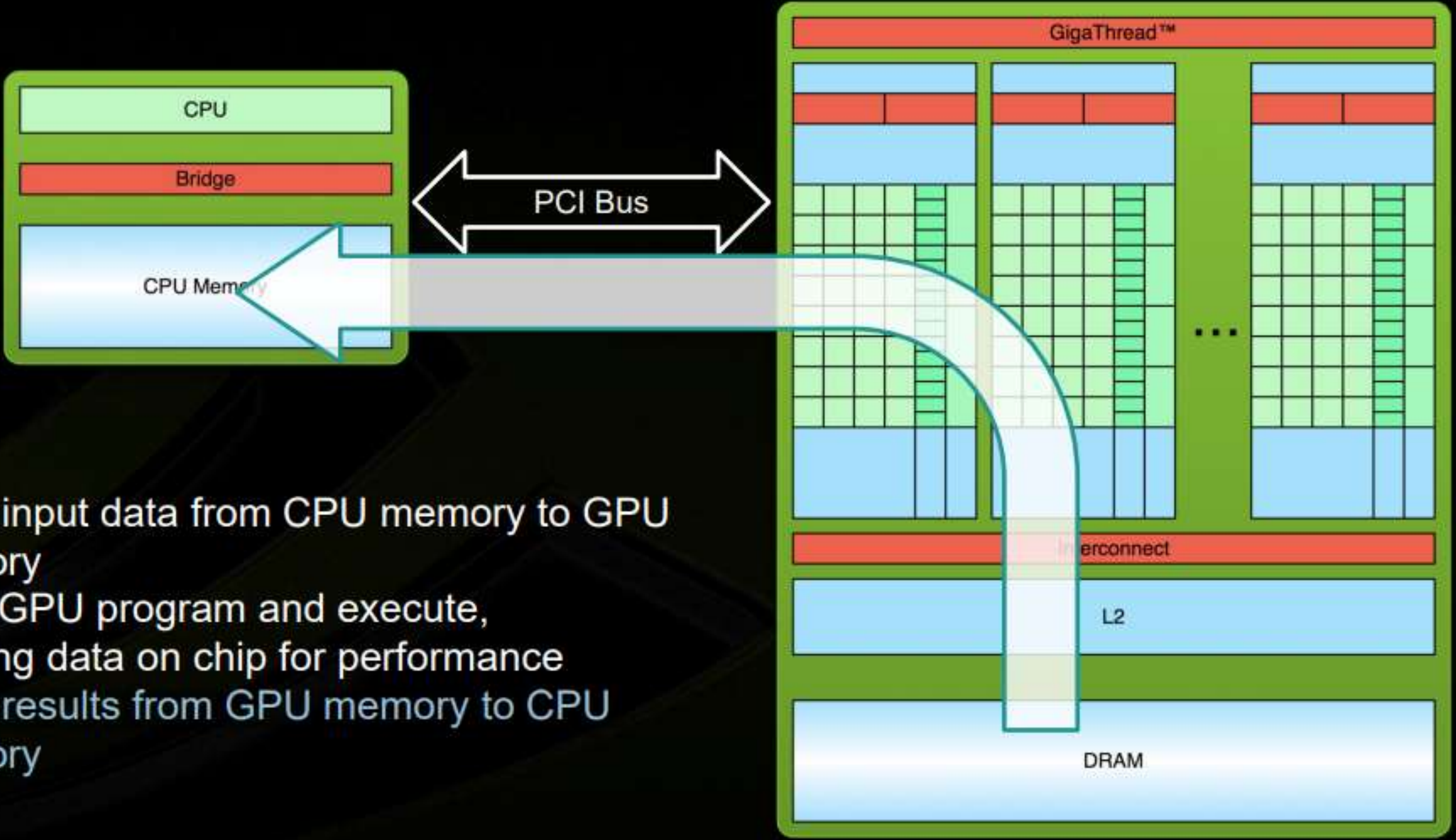
Simple Processing Flow





1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance

Simple Processing Flow

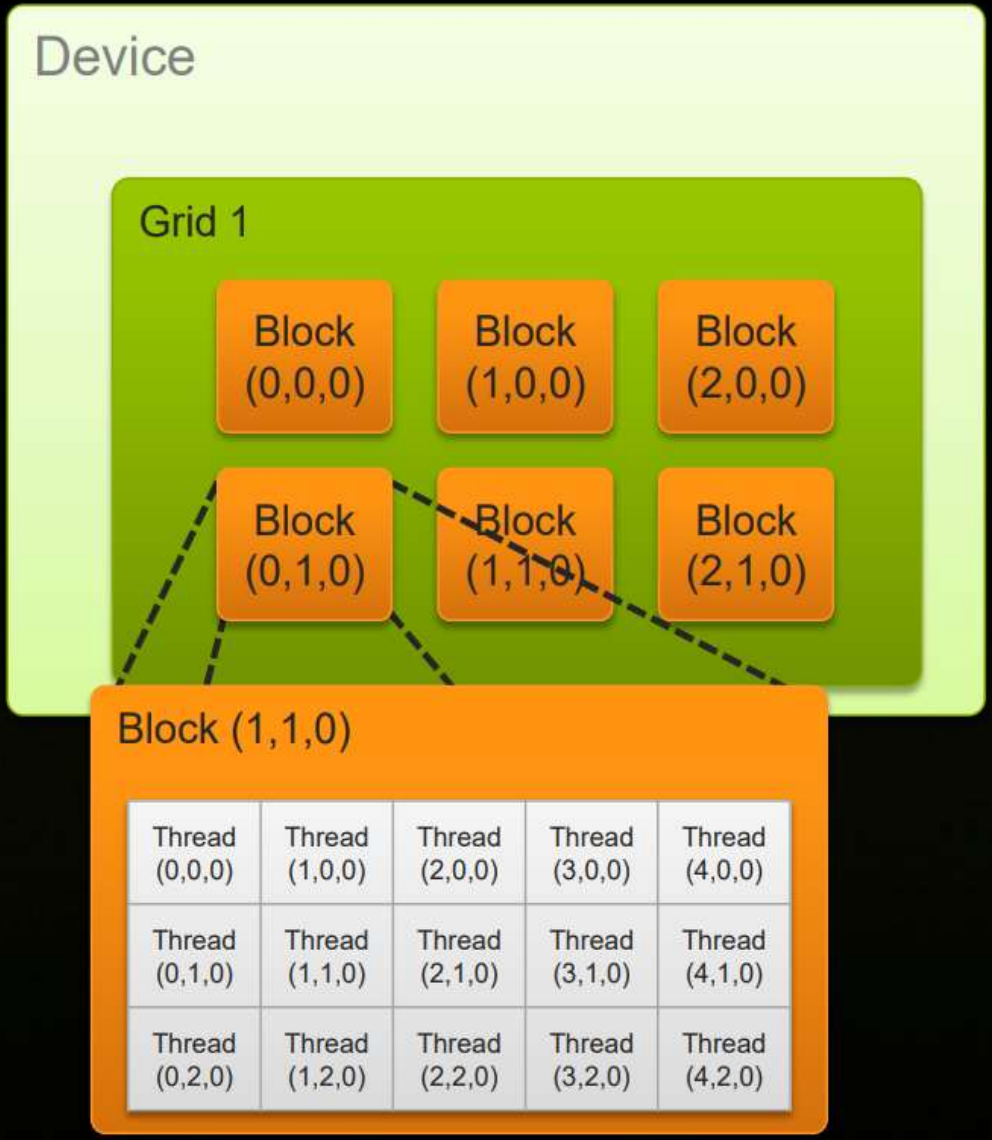


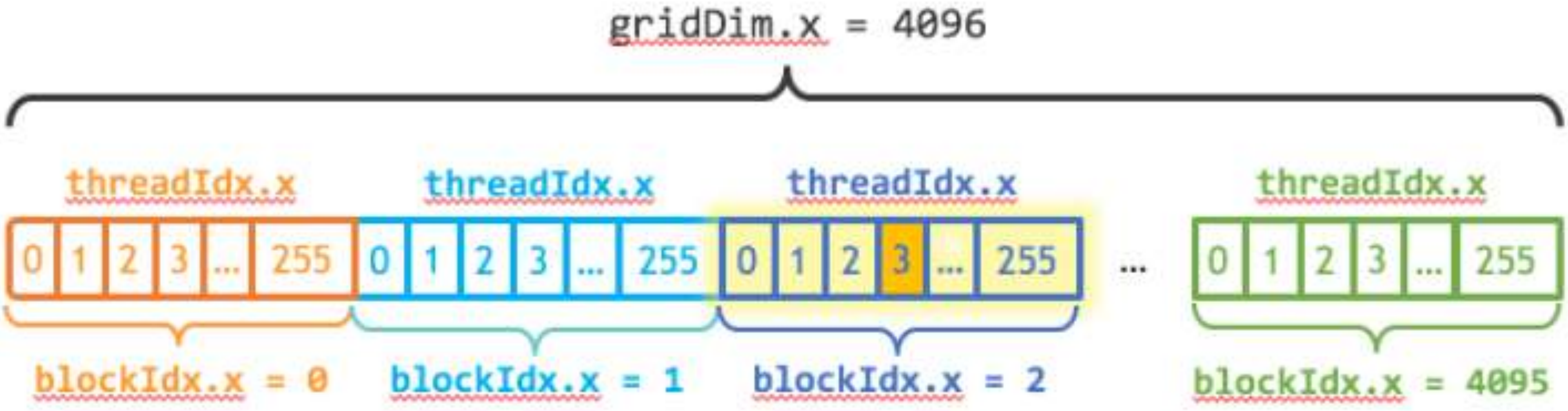
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (`x`)

- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`






```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +    (3)       = 515
```

CUDA loads one dimensional array of pointers to the GPU so we have to change our 2D matrices to 1D arrays

Πολλαπλασιασμός μήτρας με διάνυσμα

```
/* Για κάθε γραμμή τής A */
for (i = 0; i < m; i++) {
   /* Υπολογισμός εσωτερικού γινομένου i-οστής γραμμής με το x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Σειριακός ψευδοκώδικας

Σειριακός πολλαπλασιασμός μήτρας με διάνυσμα

```
void Mat_vect_mult(
      double  A[]  /* είσοδος */,
      double  x[]  /* είσοδος */,
      double  y[]  /* έξοδος  */,
      int     m    /* είσοδος */,
      int     n    /* είσοδος */) {
   int i, j;

   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
}  /* Mat_vect_mult */
```

Now we are going to spread each loop of our functions between threads of the GPU.

Below the main 3 functions in CUDA.

```
__global__ void activateNN(float* Input, int index, float* WL1, float* WL2,
float* OL1, float* OL2){

  int thr_index = blockDim.x * blockIdx.x +threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = thr_index; i < d_L1; i+=stride) {
    OL1[i] = 0;
    for (int y = 0; y < d_N; y++)
      OL1[i] += WL1[i*d_N+y] * Input[index*d_N+y];
    OL1[i] += WL1[i*d_N+d_N];
    OL1[i] = activation_Sigmoid(OL1[i]);
  }
  __syncthreads();

  for (int i = thr_index; i < d_L2; i+= stride) {
    OL2[i] = 0;
    for (int y = 0; y < d_L1; y++)
      OL2[i] += WL2[i*d_L1+y] * OL1[y];
    OL2[i] += WL2[i*d_L1+d_L1];
    OL2[i] = activation_Sigmoid(OL2[i]);
  }
}
```

```
__global__ void calc_Error(float *target, int index, float* WL2, float* OL1,
float* OL2, float* EL1, float* EL2) {

  int thr_index = blockDim.x * blockIdx.x +threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = thr_index; i < d_L2; i+=stride) {
    EL2[i] = (OL2[i] - target[index*d_L2+i]) * (derivative_Sigmoid(OL2[i])+a);
  }
  __syncthreads();

  for (int i = thr_index; i < d_L1; i+=stride) {
    EL1[i] = 0;
    for (int i2 = 0; i2 < d_L2; i2++) {
      EL1[i] += EL2[i2] * WL2[i2*d_L2+i] * (derivative_Sigmoid(OL1[i])+a);
    }
  }
}
```

```
__global__ void trainNN(float* input, float* target, int index, float* WL1, float* WL2
, float* OL1, float* OL2, float* EL1, float* EL2)
{
    int thr_index = blockDim.x * blockIdx.x +threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = thr_index; i <d_L2 ; i+=stride) {
      for (int j = 0; j < d_L1; j++) {
        WL2[i*d_L1+j] -= a * EL2[i] * OL1[j];
      }
      WL2[i*d_L1+d_L1] -= a * EL2[i];
    }

    __syncthreads();

    for (int i = thr_index; i < d_L1; i+=stride) {
        for (int j = 0; j < d_N; j++) {
          WL1[i*d_N+j] -= a * EL1[i] * input[index*d_N+j];
        }
        WL1[i*d_N+d_N] -= a * EL1[i];
  }

}
```

We run the algorithm again using the GPU of google Collab, the GPU that was used is tesla K80. Training took 11 min, slower than the CPU ? . The truth is that we are not utilizing fully the power of GPU with this simple algorithm because we have a lot of threads that go unused. In forward pass we have each thread calculate the output of one neuron which is a sum. The first optimization that we can do is to use even more threads to do reduction on the sum. The second is when we update the weights we use use one thread to update the weights of each neuron , so we simply going to assign one thread per weight.

```
  __device__ void warpReduce(volatile float* sdata, int tid) {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid + 8];
      sdata[tid] += sdata[tid + 4];
      sdata[tid] += sdata[tid + 2];
      sdata[tid] += sdata[tid + 1];
  }


__global__ void activateNN_L1(float* Input, int index, float* WL1, float* WL2, float* OL1, float* OL2)
{
    __shared__ float sInput[d_N+240];
    unsigned int block_index = blockIdx.x;
    unsigned int thr_index = threadIdx.x;

    if(block_index<d_L1 && thr_index<d_N)
        sInput[thr_index] = WL1[block_index*d_N+thr_index] * Input[index*d_N+thr_index];
    else if (block_index<d_L1 && thr_index>=d_N && thr_index<d_N+240)
        sInput[thr_index] = 0;

    __syncthreads();

    if(block_index<d_L1) {

      if (thr_index < 512) sInput[thr_index]+= sInput[thr_index+512];
      __syncthreads();
      if (thr_index < 256) sInput[thr_index]+= sInput[thr_index+256];
      __syncthreads();
      if (thr_index < 128) sInput[thr_index]+= sInput[thr_index+128];
      __syncthreads();
      if (thr_index < 64) sInput[thr_index]+= sInput[thr_index+64];
      __syncthreads();
      if (thr_index < 32) warpReduce(sInput, thr_index);

      if(thr_index == 0){
          OL1[block_index] = sInput[0] + WL1[block_index*d_N+d_N];
          OL1[block_index] = activation_Sigmoid(OL1[block_index]);
      }

    }

}
```

```
__global__ void activateNN_L2(float* Input, int index, float* WL1, float* WL2, float* OL1, float* OL2){
    __shared__ float sInput[d_L1+24];
    unsigned int block_index = blockIdx.x;
    unsigned int thr_index = threadIdx.x;

    if(block_index<d_L2 && thr_index<d_L1)
        sInput[thr_index] = WL2[block_index * d_L1 + thr_index] * OL1[thr_index];
    else if (block_index<d_L2 && thr_index>=d_L1 && thr_index < d_L1+24)
        sInput[thr_index] = 0;

    __syncthreads();

    if(block_index<d_L2) {

        if (thr_index < 512) sInput[thr_index]+= sInput[thr_index+512];
        __syncthreads();
        if (thr_index < 256) sInput[thr_index]+= sInput[thr_index+256];
        __syncthreads();
        if (thr_index < 128) sInput[thr_index]+= sInput[thr_index+128];
        __syncthreads();
        if (thr_index < 64) sInput[thr_index]+= sInput[thr_index+64];
        __syncthreads();
        if(thr_index<32) warpReduce(sInput, thr_index);

        if(thr_index == 0){
            OL2[block_index] = sInput[0] + WL2[block_index*d_L1+d_L1];
            OL2[block_index] = activation_Sigmoid(OL2[block_index]);
        }

    }

}
```

```
__global__ void calc_Error(float *target, int index, float* WL2, float* OL1, float* OL2, float* EL1, float* EL2) {

  int thr_index = blockDim.x * blockIdx.x +threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = thr_index; i < d_L2; i+=stride) {
    EL2[i] = (OL2[i] - target[index*d_L2+i]) * (derivative_Sigmoid(OL2[i])+a);
  }

    __syncthreads();

  for (int i = thr_index; i < d_L1; i+=stride) {
    EL1[i] = 0;
    for (int i2 = 0; i2 < d_L2; i2++) {
      EL1[i] += EL2[i2] * WL2[i2*d_L2+i] * (derivative_Sigmoid(OL1[i])+a);
    }
  }

}

__global__ void trainNN(float* input, float* target, int index, float* WL1, float* WL2, float* OL1, float* OL2, float* EL1, float* EL2)
{
    int i = blockIdx.x;
    int j = threadIdx.x;

    if(i<d_L2 && j<d_L1+1){
        if(j < d_L1)
            WL2[i*d_L1+j] -= a * EL2[i] * OL1[j];
        else
            WL2[i*d_L1+d_L1] -= a * EL2[i];
    }


    if(i<d_L1 && j<d_N+1){
        if(j < d_N)
            WL1[i*d_N+j] -= a * EL1[i] * input[index*d_N+j];
        else
            WL1[i*d_N+d_N] -= a * EL1[i];
    }
}
```

Total execution time 15.8 sec 4x times speed up from CPU. There further optimization that can be done but we will stop here. The complete code can be found in this link.
https://colab.research.google.com/drive/19WIwktQKzfbo19yShuJAJVjrNHY6xGO5?usp=sharing

What Became Possible With GPU Acceleration

As training neural networks on GPU is much faster it gave as the ability to train bigger more complex architectures that would not be feasible other wise, and thanks to that very big advancements in machine learning were made. Today machine learning algorithms are used almost every were. Cloud, driving cars, medicine, voice recognition, image-video editing and processing.