PAF-Karachi Institute of Economics & Technology PAF KIET THE CENTER OF EXCELLENCE	General Purpose GPU programming Dr. Ayaz Khan GSSE, PAF-KIET City Campus,

SIMD Optimization to RBF neural Network Using Cuda Framework

`Anees Ahmed (MS-CS) 58875, Arif Sultan (MS-EE) 5906, Waqar Hameed

Table Of Content

Abstract

Single Instruction Multiple Data (SIMD) is a good applicable choice where a grid of data available and we need to apply same computation to all data, like adjusting digital media, scaling digital media and manipulating matrices in Linear Algebra or Statistics or other computational work.

In this paper, we focus on Redial Based Function Network (Neural Network) for function approximation which is an ideal case for SIMD applicability.

We use Nvidia's Cuda framework for implementing SIMD using GPU. Most of the codes are in C/C++

Introduction

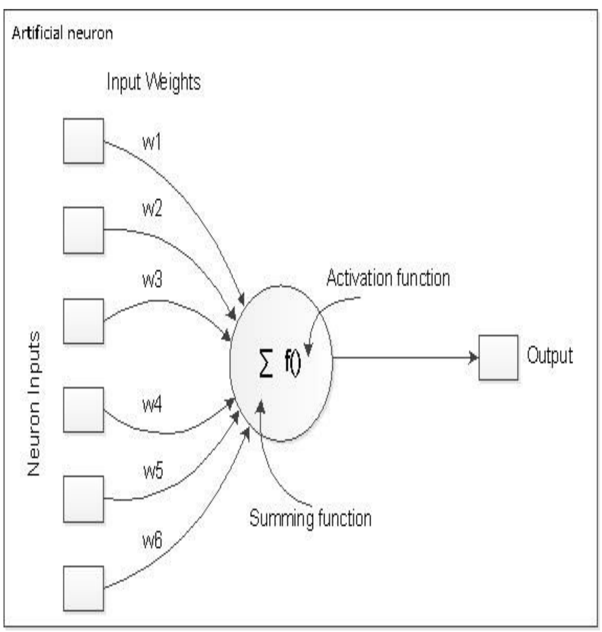
Single Instruction Multiple Data (SIMD) is an approach to parallel computation. It refers to multiple computing or processing units that perform a single operation (computing instruction) on multiple data elements simultaneously. This is also treated as data level parallelism, however, it is different if compare with concurrency. In SIMD only a single process (instruction) is available to all computation unit at a moment (1).

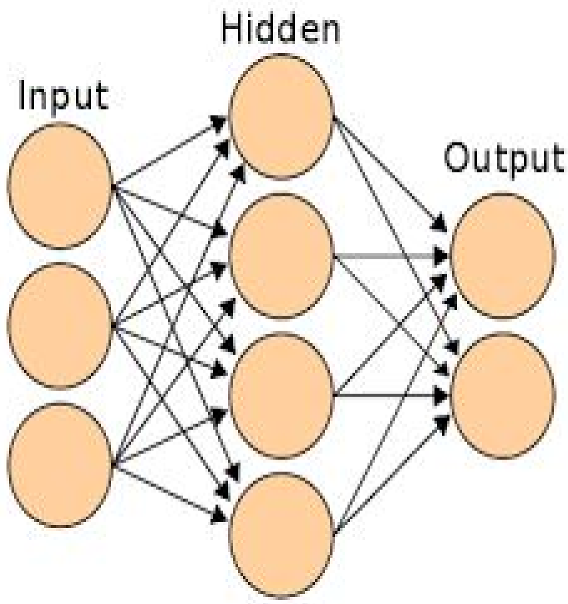
We should not confuse with SIMT which utilizes threads or CPU concurrency that utilizes scheduling and time slicing multiple cores of a CPU.

Using SIMD approach could bring tremendous advantages in computing by reducing computation time especially working with matrics like structure where the parallel calculation is a need and most calculation are not dependent on each other.

Artificial Neural Networks (ANN)

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well (4) .

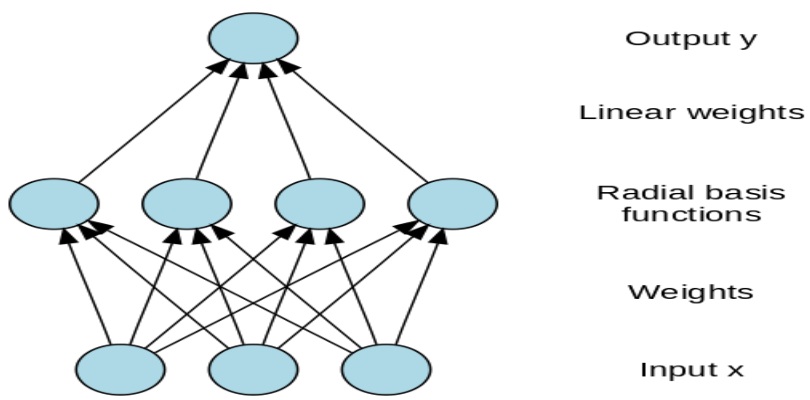




RBF Neural Networks (RBF-NN)

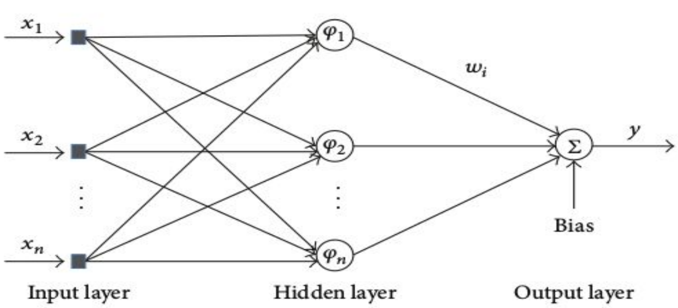
Radial Basis Functions , as a variant of Artificial Neural Network (ANN), start getting attraction in late 80 (1). They are mainly used in pattern recognition techniques but are also used for clustering, functional approximation, spline interpolation etc (2).

An RBF network has two layers of the neural network. The hidden unit implements a radial activated function while output layers of the neural network implement a weighted sum of previous layer output. The output of RBF-NN is linear, while input into the RBF is nonlinear. The nonlinear approximation properties of RBF-NN, we can model complex mappings which perceptron neural networks can only model by means of multiple intermediary layers (4).



The RBF-NN implementation is divided into different steps, like designing the kernel, data pre-processing, training, testing, and approximation.

The architecture consists of multiple layers and input layer, a nonlinear hidden layer and a linear output layer (5)



Kernel Function Design

Description

Our RBF-NN kernel is a fusion of cosine and Euclidean distances. Creating a fusion of both distance function, we get a better result as compare with the conventional approach where mostly a single function is used (5). This fusion is adaptive in nature and provides a robust result during training as an activation function (5).

$$\boxed{\varphi_i (x, x_i) = \alpha_1 \varphi_{i1} (x.x_i) + \alpha_2 \varphi_{i2} (kx - x_i k)}$$

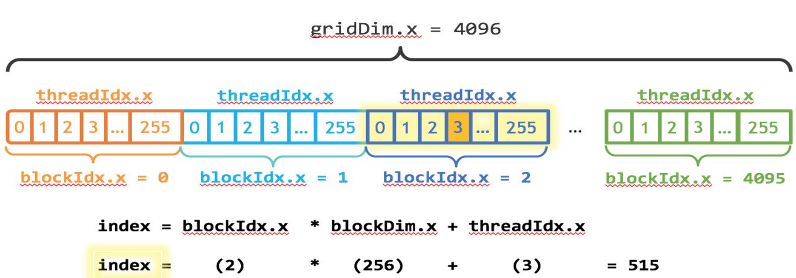
where $\varphi_{i1} (x.x_i)$ and $\varphi_{i2} (kx - x_i k)$ are the cosines and Euclidean kernels.

The kernel is implemented as sequential as following tables and can be modified with SIMD optimization. We can see the approach for optimization is straightforward. The kernel function code using an index that points to a specific thread running parallel with other threads.

SIMD Optimization Approach

The GPU optimization is straightforward. Our optimized kernel function modified to execute instructions over a data element indexed by an indexer. The index position is calculated with the help of Cuda built-in helper variables the provides a specific pointer the particular thread inside the execution block (7).

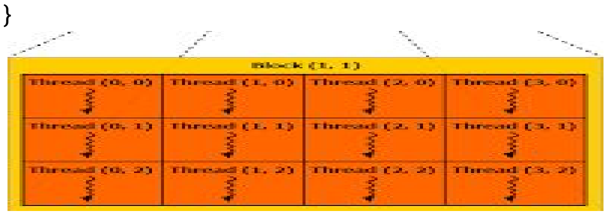
The Cuda Indexing Mechanism



Cuda C Codes

Complte codes can be reviewed at the github repository (8).

Sequential	**SIMD optimized**

void GaussianKernal(float x, float y, int CenterR, int CenterC, float Centers[][121],float* output)
{
 	
//   printf("Gauss Kernel\n\n\n");
    for (int i = 0; i < 121; i++)
    {
        output[i] = exp(-(pow((x - Centers[0][i]), 2) + pow((y - Centers[1][i]), 2))/0.04);
    //   printf("%f\n", output[i]);
    }
 	
}
void CosinKernel(float x,float y, int CenterR, int CenterC,float Centers[][121] , float* output)
{
//   printf("Cosine Kernel\n");
//
 	
    //float output[121];
    float sumCenter[121];
    float intputsq=x*x +y*y;
//   printf("\nMultiplication Kernel\n\n\n");
 	
    for (int i = 0; i < 121; i++)
    {
        float sum = 0.0;
        sum = x * Centers[0][i]+ y * Centers[1][i];
        output[i] = sum;
 	
        sumCenter[i] = sqrt((pow(Centers[0][i], 2) + pow(Centers[1][i], 2))*intputsq);
        output[i] = output[i] / (sumCenter[i]+0.0000000000000001);
        //printf("%f\n", output[i]);
    }
 	
}
	__global__
	void Gauss(float* x,float* y,float* CenterX, float* CenterY,float* output,int N)
	{
	    int i = blockDim.x*blockIdx.x + threadIdx.x;
	    //printf("x= %f, y=%f \n", x[0], y[0]);
	    if (i<N)
	    output[i] = exp(-(pow((x[0]- CenterX[i]), 2) + pow((y[0] - CenterY[i]), 2)) / 0.04);
	    printf("%d: %f\n", i, output[i]);
	
	}
	__global__
	void Coss(float* x, float* y, float* CenterX, float* CenterY, float* output, int N)
	{
	    float sumCenter;
	    float intputsq = x[0]*x[0] + y[0]*y[0];
	        //printf("%f", intputsq);
	// Cuda Helpr
	    int i = blockDim.x*blockIdx.x + threadIdx.x;
	    if (i < N)
	    {
	
	        output[i] = (x[0] * CenterX[i] + y[0] * CenterY[i]) / (sqrt((pow(CenterX[i], 2) + pow(CenterY[i], 2))*intputsq) + 0.0000000000000001);
	
	    }
	}
	

Network Design

Description

The RBF-NN network consists of a large set of neurons. Each neuron is responsible to apply kernel function (guassion, cosine) with each of inputs and obtains a sum of all values for feeding the next layer

SIMD optimization approach.

Unlike a sequential approach where a nested loop is used to sum of the function output for feeding to next layer, The optimized codes call kernel parallel using Cuda helper variables. Each call is runs parallel to the sum operation performed by many threads in the time. A full set is input is collected and send for the paralleled processing so multiple data elements are calculated at the same time with different threads and asynchronously result move back to the host memory.

Cuda C Code

codes are abstracted. For complete codes please review the GitHub repository of the project (8).

```
void train2(int n, float l_rate, float error,  float *RBF_out, float *w)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
            {


            w[i] = w[i] + l_rate*error*RBF_out[i];

  }
__syncthreads();
}
```

Kernel Launch

Description

The RBF-NN network launch requires many steps which include data generation (for testing) and loading into required structures, training using such data via RBF-NN and for training and predicting call the kernel.

Many code blocks require to run on the host in a sequential manner as their parallel implementation does not bring any optimization.

However, many helper functions like blew are optimized the use SIMD and run over the GPU (8).

GPU optimized Helper Functions:

- void OutputNeuron(float* Kernel,float* w, float* output,int N)
- void Multiplication(float* Kernel, float* w, float error, float learningRate, int N)
- void AlphaUpdate(float* KC, float* KG, float* w, float* updateAlpha1, float* updateAlpha2)

References:

1. SIMD architectures | Ars Technica." 21 Mar. 2000, https://arstechnica.com/features/2000/03/simd/. Accessed 7 Dec. 2018.

2. Introduction of the Radial Basis Function (RBF) Networks. Retrieved December 7, 2018, from https://www.researchgate.net/profile/Adrian_Bors/publication/280445892_Introduction_of_the_Radial_Basis_Function_RBF_Networks/links/585f0e4108ae6eb871a31b01/Introduction-of-the-Radial-Basis-Function-RBF-Networks.pdf

3. Haykin, S. (1994) Neural Networks: A Comprehensive Foundation. Upper Saddle River, NJ: Prentice Hall.
4. https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#What%20is%20a%20Neural%20Network
5. A Novel Adaptive Kernel for the RBF Neural Networks, Shujaat Khan · Imran Naseem · Roberto Togneri · Mohammed Bennamoun
6. A Novel Kernel for RBF Based Neural Networks Wasim Aftab, Muhammad Moinuddin, 1 and Muhammad Shafique Shaikh
7. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model Accessed 8 Dec 2018
8. https://github.com/PAF-KIET-GPGPU-Programming-Fall-2018/project-cnn_gpu/blob/master/Code%20Implementations/Cuda%20Implementation/Parallel/project/kernel.cu