Neural Network Training Acceleration with PSO Algorithm on a GPU Using OpenCL

Jiajun Li
School of Microelectronics
Tianjin University
Tianjin, China
lijiajun5678919@tju.edu.cn

Qiang Liu
School of Microelectronics
Tianjin University
Tianjin, China
qiangliu@tju.edu.cn

ABSTRACT

Neural networks and deep learning currently provide the promising solutions to many practical problems. One of the difficulties in building neural network models is the training process that requires to find an optimal solution for the network weights. The Particle Swarm Optimization (PSO) algorithm has been recently applied to neural network training due to its global search. However, the PSO algorithm suffers from large execution time. In this paper, a parallel design of the PSO algorithm is proposed, using OpenCL language on a GPU. To improve the performance, fine memory allocation is considered for the parallel particle processing and an efficient parallel reduction scheme based on local and global reduction is proposed. By fully utilizing the processing power of the GPU, the OpenCL PSO implementation accelerates the neural network training by up to 35 times, compared to the multithreaded C++ implementation on a CPU.

1. INTRODUCTION

Artificial neural networks (ANNs) are information processing systems with great ability of learning complex relationships from a set of data. Nowadays, ANNs are very popular in many fields. However, ANN still face some difficulties, one of which is the training of ANNs. In practice, the trained neural model can make accurate prediction only if (1) the training data contain sufficient information and (2) a good training algorithm is exploited. These two conditions make the Universal Approximation Theorem feasible. Sufficient training data can be provided by users, while a suitable training algorithm is hard to find to reach a satisfactory performance. In general, the training algorithms, including Quasi-Newton Methods, Non-linear least squares and Conjugate Gradient, could converge quickly and reach the locally optimal solutions. There is no guarantee that the globally optimal solution is obtained since these algorithms can only reflect the local property of the objective functions. As a result, various global optimization algorithms are more and more favorable.

The particle swarm optimization (PSO) is a non-deterministic optimization algorithm [1]. It has great adaptability in dealing with different optimization problems in a wide range of applications. However, as the algorithm simulates the movement of particle swarm over a very large number of iterations, PSO requires a large amount of execution time.

Recently, heterogeneous computing composed of CPUs and GPUs has shown powerful computation capability. In order to take full advantage of the heterogeneous computing systems, a unified programming paradigm is required. Open Computing Language (OpenCL) is an industry standard framework for programming on the computing platform composed of a combination of CPUs, GPUs, FPGAs and other processors [2]. By standardizing the programming model, developers can focus on the application development.

In this paper, a parallel design of the PSO algorithm is proposed on the platform with a CPU and a GPU using OpenCL. The parallelism of the algorithm, the computing structure of GPU as well as the execution model of OpenCL are considered together. To maximize the parallelism and avoid frequent global communication, multiple groups of particle swarm are implemented, each group containing local reduction for each iteration, while global reduction and update among different groups are executed seldom. All computation tasks involved in PSO are organized in five kernels and executed on GPU to avoid the frequent data transfers between CPU and GPU. The intermediate computation results are stored in the private and local memories as much as possible. The fine memory management allows us to exploit the high bandwidth of the memory hierarchy, with respect to data sharing among the kernels. The main contributions of the paper are summarized as follows.

- A parallel implementation of the PSO algorithm is designed using OpenCL on GPU. The implementation parallelizes the algorithm at the level of particles and divides the iteration space into multiple subgroups with local reduction followed by global reduction. In this way, the communication among parallel units is minimized. In addition, a parallel reduction scheme is also proposed.

- Variables involved in the parallelized particle processing are analyzed individually and mapped onto different levels of the memory hierarchy, to feed data efficiently to the parallel units.

- The implementation is applied to accelerate neural network training which is one of the barriers for practical applications of ANNs. The results show that the proposed implementation accelerates the training up to 35 times over the multithreaded C++ implementation on a CPU.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

The remainder of this paper is organized as follows. Section II introduces related works. Section III discusses the ANN training problem. Then Section IV briefly introduces the PSO algorithm and Section V presents the proposed OpenCL PSO implementation. Section VI evaluates the proposed implementation. Finally, Section VII concludes the paper with future works.

2. RELATED WORKS

GPU has been a promoted parallel computing device to accelerate applications. In [3], Travelling Salesman Problem was accelerated on GPU using the CUDA programming model and gained 5 times speedup. In [4], Monte-Carlo uncertainty analysis of the multiline-TRL VNA-calibration algorithm was implemented using OpenCL on GPU, leading to about 40 times speed improvement over the C++ implementation on CPU.

Various PSO algorithms have been accelerated on GPU. In [5], PSO was used to solve the Multidimensional Knapsack Problem with CUDA on GPU and obtains a speedup of 9.6 times. In [6], the authors used OpenCL to accelerate the serial OBL-based PSO K-means, and a maximum speedup of 5 times is achieved on GPU. In [7], the multi-swarm PSO algorithm was implemented on an accelerated processing unit fusing the CPU and GPU together on a single die using OpenCL, achieving a speedup of 1.29 times. In [8], the OpenCL-based PSO algorithm was implemented to solve the Quadratic Assignment Problem. Because updating the best solutions was executed at the host side, only a speedup of 7.56 is achieved. The designs above did not significantly accelerate the PSO algorithms, because they failed to minimize the communication between the host and the device. In [9], the PSO algorithm was accelerated on GPU using the CUDA programming model and achieved 10 times speedup. In [10], fine grained parallelization of high-dimension PSO is implemented on GPU using CUDA, where a thread corresponds to a given dimension of a particle.

In this paper, the PSO implementation is used to accelerate the training of ANNs. This will benefit a number of applications, such as ANNs trained by PSO were used to investigate and optimize the influence of variables of methyl orange adsorption [11], to estimate the parameters of photovoltaic models [12], and to model the EM behavior of microwave circuits [13]. The accelerated training process could lead to more accurate ANN models within limited time, and thus the optimization and estimation will be more effective.

3. PROBLEM FORMULATION

ANN is comprised of a large number of neurons connecting with each other. The connection between each pair of neurons has a weight. One of the most important steps in building ANN models is the training, *i.e.,* finding the weights of the network. The training faces several challenges such as slow convergence, local minimum, high complexity, etc.

The PSO design proposed in this paper is applicable to train various ANNs such as multi-layer perceptron (MLP) networks, recurrent networks and wavelet networks. Here we use the MLP ANN to discuss the problem because it is the most commonly used neural network [14]. The classical MLP network has three layers. The first layer has l input neurons, the second layer has p hidden neurons and the third layer has q output neurons.

The input to each neuron at the hidden layer is a weighted sum of the input parameters $i = (i_1, i_2, \cdots, i_l)$,

$$h_m = \sum_{j=0}^{l-1} w_{jm}^{12} \times i_j, 0 \le m \le p-1 \tag{1}$$

and the outputs of the network are the weighted sum of outputs $F(h_m)$ from the hidden neurons with the activation function $F(x) = \frac{1}{1+e^{-x}}$.

$$o_t = \sum_{m=0}^{p-1} w_{mt}^{23} \times F(h_m), 0 \le t \le q-1 \tag{2}$$

$\boldsymbol{w} = \left[w_{11}^{12} w_{12}^{12} \dots w_{qp}^{12}\right]$ is the network weight vector. The training process of the ANN is to find $\boldsymbol{w}$ such that the training error is minimized. The normalized training error is defined as [15]

$$E_{Tr}(w) = \sqrt{\frac{1}{size(Tr) \times q} \sum_{k \in Tr} \sum_{j=1}^{q} \left| \frac{o_j(i_k, w) - d_{jk}}{d_{\max j} - d_{\min j}} \right|^2} \tag{3}$$

where Tr represents a set of training data, d_{jk} is the expected output in the training data, $d_{\max j}$ and $d_{\min j}$ are the maximum and minimum values of d_{jk}. As a result, the ANN training is actually an unconstrained optimization problem as below. The PSO algorithm introduced in the next section can solve the problem effectively.

$$\min E_{Tr}(w) \tag{4}$$

4. PSO ALGORITHM FOR ANN TRAINING AND DESIGN CHALLENGES

The PSO algorithm simulates the movement of a swarm of particles towards an optimal solution in a multidimensional search space. Instead of using evolutionary computational algorithms, each particle in the search space is dynamically adjusted according to its own moving experience and the other particles' moving experience [1].

In the PSO algorithm, particle i has position $\boldsymbol{x_i} \in \mathbb{R}^n$ and velocity $\boldsymbol{v_i} \in \mathbb{R}^n$. Also, particle i has a fitness value $f(\boldsymbol{x_i})$, where $f : \mathbb{R}^n \to \mathbb{R}$ is the cost function which must be minimized, and the corresponding position is recorded as $\boldsymbol{pbest}_i$ called private best fitness. The whole particle swarm has the best fitness value whose position is labelled as $\boldsymbol{gbest}$, called global best fitness.

In this work, position $\boldsymbol{x_i}$ of particle i represents one possible solution of the weight vector $\boldsymbol{w}$ of ANN. Given $\boldsymbol{w}$ is d dimensions, $\boldsymbol{x_i}, \boldsymbol{v_i} \in \mathbb{R}^d$. Then the particles are processed in the following three steps.

Firstly, particles' velocity $\boldsymbol{v_i}$ and position $\boldsymbol{x_i}$ are updated according to Eqs. (5) and (6)

$$\begin{aligned} \boldsymbol{v_i} = \alpha * \boldsymbol{v_i} + c_1 * rand() * (\boldsymbol{pbest_i} - \boldsymbol{x_i}) \\ + c_2 * rand() * (\boldsymbol{gbest} - \boldsymbol{x_i}) \end{aligned} \tag{5}$$

$$\boldsymbol{x_i} = \boldsymbol{x_i} + \boldsymbol{v_i} \tag{6}$$

where c_1 and c_2 are two positive constants, $rand()$ is random number in the range [0,1], and α is the inertia weight. In this paper, α is 1.0, and c_1 and c_2 are 2.0.

Secondly, all particles' fitness is evaluated using the cost function. For the ANN training, the fitness evaluation function is Eq. (3).

Thirdly, the particle' private best fitness $\boldsymbol{pbest_i}$ is found and recorded. Then, global best fitness $\boldsymbol{gbest}$ can be found and updated. The PSO algorithm is an iterative process. At each iteration, if the global best fitness meets the criteria or the maximum number of iterations is reached, the algorithm stops and outputs the solution; otherwise the particles are updated and the particle swarm continues movement.

Although enabling global search, the PSO algorithm converges slowly due to the stochastic population-based nature. It is thought that the PSO algorithm is suitable to parallel execution. However, there are three main questions needed to be addressed in order to achieve efficient parallel implementation of the PSO algorithm.

(1) The first one is in which level to perform parallelization. In other words, how the workload is partitioned among parallel threads. Fine grained parallelization makes the computation task of each thread simple, but may introduce large amount of communications among parallel threads. In contrast, coarse grained parallelization assigns more workload to each thread, requiring more computing and storage resources per thread, although inter-thread communication is reduced. A tradeoff should be considered.

(2) The increased number of parallel threads enables better performance, provided those threads can efficiently access to data. Herein lies the second challenge. OpenCL supports a memory hierarchy. Each processing element (PE) has a private memory. The private memory has a bandwidth of about 8 TB/s. Each computing unit (CU), composed of several PEs, has a local memory which is only open to the threads mapped to the CU. The bandwidth of local memory is about 1.5 TB/s. Lastly, the whole device has a global memory which can be accessed by all CUs and is also used to communicate with the host. The bandwidth of global memory is 200 GB/s. As a result, it is suggested that the data should be stored as close as possible to PE where the data are processed, as well as considering data sharing to reduce data transfer.

(3) The parallel PSO algorithm involves the reduction step, which produces the global best fitness $\boldsymbol{gbest}$ based on the results of individual particles. The traditional sequential reduction significantly degrades the performance. Therefore, the third question stands in how to design parallel reduction to best utilize the available parallel computing resources and avoid frequent global reduction.

In this paper, we propose a parallel PSO design addressing the above issues. The details will be presented in the next section.

5. OPENCL IMPLEMENTATION OF PSO

The overall OpenCL implementation of the PSO algorithm is shown in Fig. 1. Before looking at the details, the execution model of OpenCL is briefly introduced. In OpenCL the computation platform contains a host and several acceleration devices [2]. In this paper, the device is a GPU, which is composed of multiple compute units (CUs). Each CU contains multiple processing elements (PEs), where the computation is actually executed. During execution, a host program runs on the host and sends kernels to execute on the device. When a kernel is submitted for execution, an index space is defined. A work item (thread) is instantiated to execute each point in the index space. Each work item executes the same code as expressed in the kernel. Work items are grouped into work groups, and the work items within a work group are executed concurrently on the PEs of a single CU. Based on this execution model, the parallel PSO is designed as follows.

The initialization and final output parts are executed on a host (a CPU), while the middle part of the diagram is mapped on a GPU. The middle part implements five kernels corresponding to the operations involved in the PSO algorithm, including random number generation, particle update, fitness evaluation and the best fitness selection. Decomposition of the operations into different kernels leads to modularized design and enables easy extension, such as using different random number generators and cost functions. The first three operations execute on individual particles, and thus we assign one work item to perform each operation per particle. That is, we parallelize the algorithm at the particle level.

The work items are divided into multiple work groups. The number of work items in a work group should be carefully designed. In OpenCL, the most basic unit of scheduling is called wavefront. The number of work items in a wavefront is 32 in NVIDIA GPU. To avoid divergence, the wavefronts in a work group should be unabridged, *i.e.,* the number of work items in a work group is multiples of 32 in NVIDIA GPU. According to [16], when the work group contains 256 work items, the utilization of the used device can reach one hundred percent.

The best fitness selection is a reduction operation, which is divided into two phases implemented in two kernels. In the first phase, the particle with the best fitness is found within each work group. This phase is called local reduction. The local reduction with the first three operations iterates hundreds of times and obtains the best particle of each work group. In the second phase, the best particles of different work groups are compared to search the best one within the whole particle swarm. The second phase is called global reduction. The two-phase implementation avoids frequent global reduction. Both local reduction and global reduction are parallelized, where each work item is assigned to compare one pair of values. The designs of the parallel reduction scheme and the kernels are presented in the next in detail, together with the fine manipulation of memory objects.

5.1 Random Number Generator

As shown in Eq. (5), random numbers are needed in the particles' velocity updating. To reduce data transfers between the host and the device, a random number generator is implemented as a kernel and is executed by each work item on GPU. Since there is not $rand()$ function in OpenCL, the linear congruential method [17] for random number generation is implemented. The implementation has the advantages in computing speed, portability and low deviation rate. Its formulas are shown as follows.

$$a_k = (\beta a_{k-1} + c) \bmod M \tag{7}$$

$$r_k = \frac{a_k}{M}, k = 1, 2, \ldots \tag{8}$$

In this work, β, c and M are 21403, 2531011 and 2147483648, respectively. r_k is the random number. The seed a_k is stored in local memory because it is used by this kernel only. The generated random numbers are stored in global memory so as to be accessed by the next kernel.

5.2 Particles' Position and Velocity Update

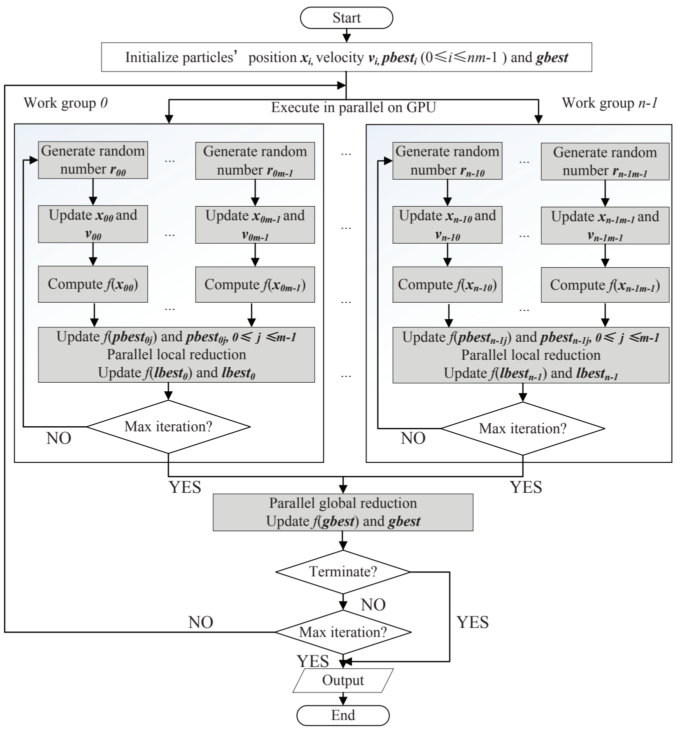
Start
Initialize particles' position x_i, velocity v_i, $pbest_i$ ($0 \leq i \leq nm-1$) and $gbest$
Work group 0
Execute in parallel on GPU
Work group n-1
Generate random number r_{00}
Generate random number r_{0m-1}
Update x_{00} and v_{00}
Update x_{0m-1} and v_{0m-1}
Compute $f(x_{00})$
Compute $f(x_{0m-1})$
Update $f(pbest_{0j})$ and $pbest_{0j}$, $0 \leq j \leq m-1$
Parallel local reduction
Update $f(lbest_0)$ and $lbest_0$
Max iteration?
NO
Generate random number r_{n-10}
Generate random number r_{n-1m-1}
Update x_{n-10} and v_{n-10}
Update x_{n-1m-1} and v_{n-1m-1}
Compute $f(x_{n-10})$
Compute $f(x_{n-1m-1})$
Update $f(pbest_{n-1j})$ and $pbest_{n-1j}$, $0 \leq j \leq m-1$
Parallel local reduction
Update $f(lbest_{n-1})$ and $lbest_{n-1}$
Max iteration?
NO
YES
YES
Parallel global reduction
Update $f(gbest)$ and $gbest$
Terminate?
NO
YES
Max iteration?
NO
YES
Output
End

Figure 1: The proposed parallel PSO implementation.

The updating of the particles' position and velocity as shown in (5) and (6) is in the second kernel. Each work item updates single particle's position and velocity. As shown in Fig. 1, the PSO implementation contains two levels of iteration loops, the inner level within each work group for local reduction and the outer level for global reduction. As a result, the updating operation has slight modification, which is when iterating from the outer loop into the inner loop the global best $\boldsymbol{gbest}$ is used and within the inner loop the local best $\boldsymbol{lbest}$ is used in (5).

The memory objects for the computation are managed as below. The host first transmits the initial particles to the global memory of GPU. Particles' position $\boldsymbol{x}_i$, private best position $\boldsymbol{pbest}_i$, local best position $\boldsymbol{lbest}$ and global best position $\boldsymbol{gbest}$ are stored in the global memory. This is because $\boldsymbol{x}_i$ is required and $\boldsymbol{pbest}_i$ is generated by the fitness evaluation kernel, and $\boldsymbol{lbest}$ and $\boldsymbol{gbest}$ are generated by the local and global reduction kernels, respectively. In contrast, particle's velocity $\boldsymbol{v}_i$ is only used by this kernel and stored in the local memory. Due to the limited space of the private memory and the scalability requirement, $\boldsymbol{v}_i$ is not stored in the private memory.

5.3 Fitness Evaluation

The formula of the cost function has been introduced in Eq. (3). The complex computation is decomposed into simple arithmetic operations according to Eqs. (1)-(3). The training data $Tr = \{(i_1, d_1), (i_2, d_2), \ldots\}$ of the ANN are transmitted from the host to the global memory first, and then moved to the local memory for fast access and being accessed by multiple work items. The variables $\boldsymbol{h}$, $\boldsymbol{o}$, and intermediate computation result $\boldsymbol{y}$, which contains the result of the absolute operation $|\cdot|$ in Eq. (3), are stored in the private memory. The obtained fitness value $f(\boldsymbol{x}_i)$, *i.e.*, $E_{Tr}(\boldsymbol{w})$, is stored in the global memory.

5.4 Parallel Local Reduction

This kernel launches one work item per particle in a work group and searches for the best within a work group. The kernel contains three parts. The first is to update the private best $\boldsymbol{pbest}_i$ and the corresponding fitness value $f(\boldsymbol{pbest}_i)$. The second part is the best fitness selection of the work group. The best fitness values $f(\boldsymbol{pbest}_i)$ of particles in a work group are first transferred from the global memory to the local memory and then parallel reduction executes.

A parallel reduction scheme was proposed in [18]. The scheme is that for 2^n data being reduced 2^{n-1} work items operate and work item i compares datum i and datum $2^{n-1} + i$. The parallel reduction structure avoids wavefront divergence, because the consecutive work items operate in the same way and strided accesses to the memory is achieved. However, when the total number of data under reduction is not a power of two, a number of large values need to be padded, wasting computation power. We propose a new scheme shown in Fig. 2, where five data are reduced to produce the minimum one. In the scheme work item i makes a comparison between datum i and datum $n-1-i$, where n is the number of data under reduction. If the number of data at a step is odd, the data in the middle is stored without operation. In this way, unnecessary operations are avoided. This scheme can deal with any numbers of data while there is no wavefront divergence.

The last part updates the local best of each work group, and the results are stored in the global memory.

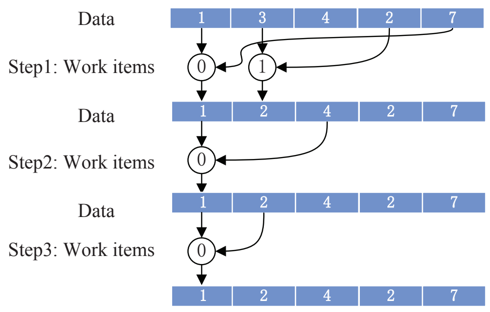

Figure 2: Example of our parallel reduction scheme.

5.5 Parallel Global Reduction

The final kernel performs the global best fitness selection. The best fitness values $f(\boldsymbol{lbest}_i)$ of the work groups are transmitted to the local memory of the first work group, and then the parallel reduction as presented above is executed to obtain the global best fitness. Afterwards, the particles' global best position $\boldsymbol{gbest}$ is renewed and stored in the global memory.

So far, the parallel PSO design is introduced. In the next, we evaluate the design for training ANNs.

6. EXPERIMENTAL RESULTS

The GPU used in this work is NVIDIA GT630. It is equipped with the NVIDIA graphic processor Kepler GK108 containing 384 PEs. The CPU is Intel Pentium G2030 with a clock frequency of 3.0GHz and dual-core double threads. To compare with, the C++ version of the PSO algorithm is also implemented and executed on the CPU. The C++ version is optimized into multithread using OpenMP in Intel Parallel Studio. In the optimized C++ version the *for* loops in objective function and particles' position and velocity update are mapped into two threads. The experimental results include three parts. The first part evaluates the performance of the OpenCL implementation for training various ANNs while compared with the multithreaded C++ version. The second part compares our design to an existing parallel PSO implementation. The last part applies our PSO design to a real application.

6.1 Performance Evaluation

In the first experiment, the ANN under training has 50 weights, and thus the particles' dimension is $d = 50$. The number of training data is 50. The results, shown in Table 1, are the execution time when the maximum number of iterations of the PSO, which is 100000, is reached. What can be seen from the table is that the OpenCL implementation on the CPU+GPU is significantly faster than the multithreaded C++ implementation on the CPU. The speedup increases almost linearly as the number of particles increases.

In the second experiment, the number of particles is fixed to 1024. The particles' dimension varies according to the architecture of the ANNs under training. Again, the maximum number of iterations is 100000. The results are shown in Table 2. The superior performance of the OpenCL implementation can also be seen. The speedup is up to 50 times compared to the multithreaded C++ implementation. Unlike the previous experiment, as the number of dimension

Table 1: Comparison of C++ and OpenCL implementations with different numbers of particles.

particles	C++ (sec)	OpenCL (sec)	speedup
64	386.76	97.97	3.95
128	771.05	103.81	7.43
256	1543.93	117.38	13.15
512	3086.09	137.19	22.5
1024	6218.58	203.47	30.56

Table 2: Comparison of C++ and OpenCL implementations for training different ANNs

ANN	Dimension	C++ (sec)	OpenCL (sec)	speedup
3-8-1	32	3926.11	145.50	26.98
7-8-1	64	6643.08	215.32	30.85
4-6-12	96	15367.46	481.38	31.92
6-8-10	128	17308.09	529.50	32.69
15-32-1	512	68065.32	1904.16	35.75

increases, the execution time of the OpenCL implementation increases linearly, and thus the speedup over the multithreaded C++ version is almost the same.

In the third experiment, the proposed parallel PSO design is compared with a parallel PSO implementation, in which there is no local reduction and global reduction is executed at each iteration. In this experiment, the number of particles is fixed to 1024, and the particles' dimension is 240. The number of training data is 100. The ANN has 6 inputs, 2 outputs and 30 hidden neurons. Both implementations are compared in terms of execution time and training error after 1000 iterations. The results in Table 3 show that our parallel PSO design is up to 4 times faster than the traditional parallel PSO implementation, and achieves the training error one order of magnitude better.

6.2 Comparison with Existing Parallel PSO Implementation

In this part, the proposed implementation is compared with the existing parallel PSO implementation [10]. In [10], each work item is instantiated for one dimension of a particle and the work group represents a particle, while in our implementation each work item is assigned to one particle.

Table 4 shows the results. The comparison is made in the same conditions except the GPU devices. The GPU used in [10] is NVIDIA GTX 460 which has a computing power of 1.36 TFLOPS while the capability of the GPU we used is 692.7 GFLOPS. Two cost functions used in [10] are deployed, $f_1(x) = \sum_{i=1}^{n} x_i^2$ and $f_2(x) = 100\left(x_{i-1} - x_i^2\right)^2 + (x_{i-1} - 1)^2$. The results show that our proposed implementation achieves better performance on less powerful device. Note that the results of the implementation [10] are cited from the published paper.

6.3 Real Application

To evaluate the performance of the proposed design in practice, the electromagnetic (EM) behavior modeling of an ultra-wideband (UWB) antenna [13] is deployed in this experiment. The model has six input variables and two output variables. A training data set with 5000 data samples is

Table 3: Comparison of our parallel PSO implementation to the parallel PSO implementation, in which there is no local reduction and global reduction is executed at each iteration.

Versions	Execute time (*sec*)	Training error
Parallel PSO without local reduction	99.14	0.37056
Our parallel PSO	25.53	0.05198

Table 4: Comparison of the parallel PSO implementation [10] and ours.

Implementation	f_1 (*sec*)	f_2 (*sec*)
Ours	4.58	6.9
[10]	6.36	9.19

used. The target neural model has 30 hidden neurons. The number of particles is 1024. The proposed OpenCL design and the multithreaded C++ implementation are applied and compared in terms of execution time and training error after 2000 iterations. The results are shown in Table 5. The proposed OpenCL implementation is over 33 times faster than the multithreaded C++ implementation while the training errors are close.

7. CONCLUSIONS

In this paper, a parallel implementation of the PSO algorithm using OpenCL on GPU is introduced. The implementation achieves performance improvement up to 35 times compared to the multithreaded C++ version for training various ANNs. Future work considers mapping the OpenCL design onto FPGAs to see what performance improvement can be achieved from the adding extra device to the heterogeneous computing platform.

8. REFERENCES

[1] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[2] Aaftab Munshi, Benedict Gaster, Timothy Mattson, James Fung, and Ginsburg Dan. OpenCL programming guide. 2011.

[3] Olfa Bali, Walid Elloumi, Pavel Krmer, and Adel M. Alimi. GPU particle swarm optimization applied to travelling salesman problem. In *Mcsoc*, 2015.

[4] Pawel Linczuk, Piotr Zdunek, Pawel Barmuta, Marcin Kotz, and Arkadiusz Lewandowski. GPU implementation of multiline TRL calibration for efficient Monte-Carlo uncertainty analysis. In *International Conference on Microwave, Radar and Wireless Communications*, pages 1–4, 2016.

[5] D Zan and J Jaros. Solving the multidimensional knapsack problem using a CUDA accelerated PSO. In *IEEE Congress on Evolutionary Computation*, pages 2933–2939, 2014.

[6] Qingyu Zhai, Dongfeng Yuan, Haixia Zhang, and Kai Gao. Parallelization of OBL based PSO K-means algorithm using OpenCL architecture. In *International Conference on Natural Computation*, pages 714–719, 2014.

Table 5: ANN training for UWB EM behavior modeling.

Implementation	Execute time (*sec*)	Training error
OpenCL	2435.36	1.52%
C++	82726.95	1.49%

[7] Wayne Franz, Parimala Thulasiraman, and Ruppa K. Thulasiram. *Optimization of an OpenCL-Based Multi-swarm PSO Algorithm on an APU*. 2014.

[8] Piotr Szwed, Wojciech Chmiel, and Piotr Kaduczka. *OpenCL Implementation of PSO Algorithm for the Quadratic Assignment Problem*. Springer International Publishing, 2015.

[9] Feng Chen, Yu Bo Tian, and Min Yang. Research and design of parallel particle swarm optimization algorithm based on CUDA. *Computer Science*, (47):280–287, 2014.

[10] R. M. Calazan, N. Nedjah, and L. d. M. Mourelle. Parallel GPU-based implementation of high dimension particle swarm optimizations. In *2013 IEEE 4th Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4, Feb 2013.

[11] Shilpi Agarwal, Inderjeet Tyagi, Vinod Kumar Gupta, M. Ghaedi, M. Masoomzade, A. M. Ghaedi, and B. Mirtamizdoust. Kinetics and thermodynamics of methyl orange adsorption from aqueous solutions–artificial neural network-particle swarm optimization modeling. *Journal of Molecular Liquids*, 218:354–362, 2016.

[12] Jieming Ma, Ka Lok Man, T. O Ting, Nan Zhang, Sheng Uei Guan, and Prudence W. H Wong. Accelerating parameter estimation for photovoltaic models via parallel particle swarm optimization. In *International Symposium on Computer, Consumer and Control*, pages 175–178, 2014.

[13] F. Feng, C. Zhang, J. Ma, and Q. J. Zhang. Parametric modeling of EM behavior of microwave components using combined neural networks and pole-residue-based transfer functions. *IEEE Transactions on Microwave Theory and Techniques*, 64(1):60–77, Jan 2016.

[14] S. Razavi and B.A. Tolson. A new formulation for feedforward neural networks. *Neural Networks, IEEE Transactions on*, 22(10):1588–1598, Oct 2011.

[15] Qi Jun Zhang, K. C Gupta, and V. K Devabhaktuni. Artificial neural networks for RF and microwave design - from theory to practice. *IEEE Transactions on Microwave Theory & Techniques*, 51(4):1339–1350, 2003.

[16] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Elsevier, MK, 2012.

[17] Ke Ding and Ying Tan. Comparison of random number generators in particle swarm optimization algorithm. In *IEEE Congress on Evolutionary Computation*, pages 2664–2671, 2014.

[18] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. 2012.