Parallel Neural Network Training with OpenCL

Nenad Krpan, Domagoj Jakobović
Faculty of Electrical Engineering and Computing
Unska 3, Zagreb, Croatia
Email: nenadkrpan@gmail.com, domagoj.jakobovic@fer.hr

***Abstract*—This paper describes the parallelization of neural network training algorithms on heterogeneous architectures with graphical processing units (GPU). The algorithms used for training are particle swarm optimization and backpropagation. Parallel versions of both methods are presented and speedup results are given as compared to the sequential version. The efficiency of parallel training is investigated in regards to various neural network and training parameters.**

I. Introduction

The application of neural networks can greatly benefit from implementations that take advantage of parallelism, i.e. where the neural network output is computed in parallel [1], [2], [3]. However, the neural network learning process also requires much time and needs high-performance machines for industrial applications. This is especially true for networks with numerous neurons and synaptic connections and for training sets with large number of examples.

This paper discusses the parallelization of neural network training process on heterogeneous architectures using OpenCL. We present parallel implementations of two training algorithms: the backpropagation algorithm and training with particle swarm optimization (PSO). Backpropagation is probably the most used algorithm for training, whereas PSO is increasingly gaining popularity for this purpose in recent years. The parallel versions are compared to their serial counterparts and speedup and efficiency measures are reported.

A special care is given to the influence that neural network and learning parameters, such as the number of layers, neurons and learning samples, have on parallel algorithm performance. Furthermore, the specifics of the OpenCL architecture are taken into account so as to carefully devise the parallelization method that would be efficient in different conditions and thus usable to potential researchers. The results show that parallel implementations have a clear advantage over original algorithms, which can be obtained with widely available hardware.

The remainder of this paper is organized as follows: Section II describes the neural network training process with the two algorithms. Section III presents parallel models of the training algorithms for the target OpenCL platform. Section IV gives experimental results and efficiency analysis, and short conclusions are given in Section V.

II. Training Neural Networks

Training neural networks includes finding a suitable set of neuron weights, given a network structure, that best describe the given set of examples (learning test cases, learning samples). Neural networks can be trained with almost any optimization method, but several algorithms have proven especially efficient for this purpose. Among those, the backpropagation algorithm and particle swarm optimization may well be the most prominent ones in recent years.

A. Backpropagation

The main goal of network training is to reduce the amount of difference between the actual and the desired network output. The backpropagation algorithm uses the output error to adapt the weights of the output layer, hidden layers and finally the input layer. The update scheme uses the derivative of the activation function and thus requires the function to be differentiable. After the procedure is performed for each training sample, the process is repeated until a predefined rate of convergence is attained [4], [5], [6].

The error of a single output neuron j is defined as $e_j = d_j - y_j$, where d_j is the desired and y_j the actual output. The local derivative of an output neuron is defined as

$$\delta_j = e_j \cdot \varphi'_j \ , \tag{1}$$

where φ'_j is the derivative of the activation function, and the local derivative of a neuron j in the hidden layer is

$$\delta_j = \varphi'_j \sum_k \delta_k \cdot w_{jk} \ , \tag{2}$$

where k indicates all the output neurons the neuron j is connected to. The neuron weights are then updated with the predefined learning rate η by

$$\Delta w_{ji} = \eta \cdot \delta_j \cdot y_i. \tag{3}$$

B. Particle Swarm Optimization

PSO is an optimization algorithm (a metaheuristic) inspired by the flocking of birds and similar representations of organisms in large groups [7], [5], [8]. It is an example of a larger group of algorithms implementing *swarm intelligence* to solve non-linear optimization problems. PSO simulates the movement of a group of potential solutions, or particles, in problem domain according to simple rules governing their relative velocity.

In each iteration, the algorithm updates the position of every particle (its value in problem domain) and its speed taking into account:

- previous speed of the particle;

- previous best position found by this particle;
- best position found by all particles so far.

These components are combined in such a way to maintain the exploratory nature of the algorithm on one hand, and to preserve convergence properties on the other.

The dimensionality of the problem domain (the space the particles move in) is equal to the number of weights in the neural network. Each particle represents a potential solution, i.e. a complete set of weights and biases (if included) of the neural network. Particle fitness f is usually obtained by evaluating the corresponding network on a number of test cases and calculating the mean squared error as:

$$f = \frac{1}{N \cdot O} \sum_{i=1}^{N} \sum_{j=1}^{O} (y_{ij} - d_{ij})^2 \tag{4}$$

where N represents the number of learning samples, O is the number of neurons in the output layer, whereas y_ij and d_ij is the actual and the desired output of j-th output neuron for i-th learning sample. After the evolution process, the best evolved particle is used to define the weights for the resulting neural network.

III. Parallelization using OpenCL

A. OpenCL programming model

OpenCL [9], [10], [11] is an open standard which defines a parallel programming framework for programs that execute across heterogeneous platforms. Programs that conform to the standard may be executed on CPU, graphical processing units (GPU) and other devices that have interfaces to the standard specification. Processing elements in OpenCL are the *compute device* (such as GPU), which may have one or more *compute units* (e.g. a GPU multiprocessor), which in turn may consist of one or more *processing elements* (e.g. a GPU scalar processor). The processing elements are accessed by a *host*, which may be a computer system where OpenCL programs are initiated.

The programming elements in OpenCL are based on *work items* or *kernels*, which are equivalent to tasks in parallel programming model, and which execute on a processing element. A set of multiple work items that are processed (in parallel) by a compute unit is called a *workgroup*.

OpenCL memory model distinguishes different levels of memory:

- *global memory* is a multipurpose memory on the compute device that is the largest and the slowest one;
- *constant memory* is a part of the global memory that remains unchanged during the kernel execution;
- *local memory* is assigned to a compute unit and is smaller and faster than the global memory;
- *private memory* is the smallest and the fastest (e.g. registers), and is accessible only from the processing element it belongs to.

The global memory is the only way of communication between the host and the compute device, whereas the other types of memory may be used for internal computation. In case more private memory is requested than physically available, global memory will be used instead.

B. Training with parallel Backpropagation

The neural network structure in this work is presumed to match the multilayered perceptron. The algorithms can also be used for different models, but possibly with different performance results.

In our parallel backpropagation variant, first the network output is computed layer by layer at a time. Each layer is handled with a set of work items, where each work item computes the output of a single neuron. The computation of the output of the next layer is possible only when the previous layer has been computed, so this is done sequentially for each layer.

The same mechanism is used when local derivatives are calculated in the opposite direction - layer by layer are processed in parallel, and finally a single kernel is executed which updates the weights of all neurons. The number of times N_k that the kernel execution is started in each backpropagation iteration is thus

$$N_k = S \cdot (2 \cdot (L - 1) + 1), \tag{5}$$

where L is the number of layers and S the number of samples. Since every kernel execution induces a slight overhead, this technique is not expected to be efficient for small networks.

C. Training with parallel PSO

In parallel PSO implementation, the task of calculating the network output for every learning sample and every particle is represented as an individual OpenCL kernel function (unlike in the backpropagation algorithm). In the training process there are S samples and N particles, where each particle contains a set of all weights in a network (a particle is equivalent to a single network). In each iteration of the algorithm $N \cdot S$ work items can be executed in parallel (for all the particles and all the samples). The algorithm is briefly presented as follows:

```
copy samples from host to device;
repeat
  copy weights to device;
  compute outputs for all particles;
  compute MSE of all particles;
  copy MSEs to host;
  update particle positions and speeds;
until termination criterion is satisfied;
```

The samples are copied onto the computing device (the GPU) only once in the beginning as they remain unchanged during the learning. The network weights (particles) are modified in each iteration and thus have to be transferred to the device each time. The MSEs of all particles are also calculated on the device and are then copied to the host machine. The host machine maintains and updates the current speed and position of every particle, since the complexity of this operation is negligible compared to the rest of the algorithm.

When calculating network output on the device, a single work item determines the output values of a single network layer in parallel, based on the previous layer output. From this it follows that every work item has to allocate the amount of memory twice the size of the largest network layer.

Based on this approach, we developed and tested two variants of parallel training algorithm: the first variant uses the *private memory* for calculation, which can only be allocated statically inside the kernel. Since OpenCL program requires compilation before every kernel execution, the exact amount of memory doesn't need to be predefined.

The second variant uses the *local memory* for output calculation. Work items are organized in workgroups so that every work item in the same group computes outputs for the same particle, but for different input sample. The total amount L of local memory per workgroup (in bytes) can thus be determined by

$$L = 4 \cdot (N_w + 2 \cdot G \cdot W_{Lmax}) \tag{6}$$

where N_w is the number of weights, G is the number of work items in a group and W_{Lmax} is the size of the largest network layer. This number must be taken into account beforehand, and this approach cannot be used for very large networks.

IV. Results and Analysis

In this section the performance analysis for both presented methods is reported. The target platform for maximum performance is the GPU, but since the OpenCL programming model allows portability to different platforms, this allows the same OpenCL program to be executed on a CPU as a computing device. Beside the GPU results we also include measurements of the same program executed on a central processor with multiple cores.

The tested GPU platform is NVIDIA GeForce GTS 250 and the CPU is AMD Phenom II x6 1055T (6 cores). Regardless of the target platform, for speedup measurement all OpenCL execution times are compared to the serial algorithm implementation on a single CPU core.

A. Parallel backpropagation algorithm

For the backpropagation algorithm the following annotations for tested platforms are used in the results:

- **CPU**: sequential algorithm, executed on CPU (on a single core);
- **OpenCL GPU**: OpenCL implementation on GPU;
- **OpenCL CPU**: OpenCL implementation on CPU.

The parallel BP algorithm was tested on a network with 9 input neurons, 3 hidden layers and a single output. The algorithm efficiency proved to be very low for smaller networks; only when the number of neurons in a hidden layer is larger than 400, the algorithm is faster than the serial version. The runtimes of single algorithm iteration are shown in Fig. 1 and the corresponding speedup is given in Fig. 2. If the number of neurons is even more increased, the algorithm exhibits useful speedups, as shown in Fig. 3.

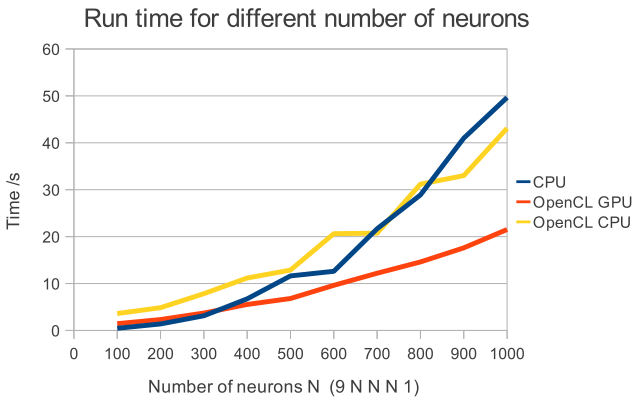

Fig. 1. Parallel BP, run time for different number of neurons

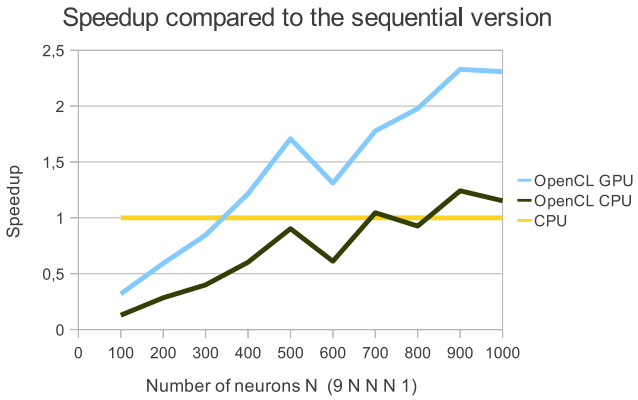

Fig. 2. Parallel BP, speedup for different number of neurons

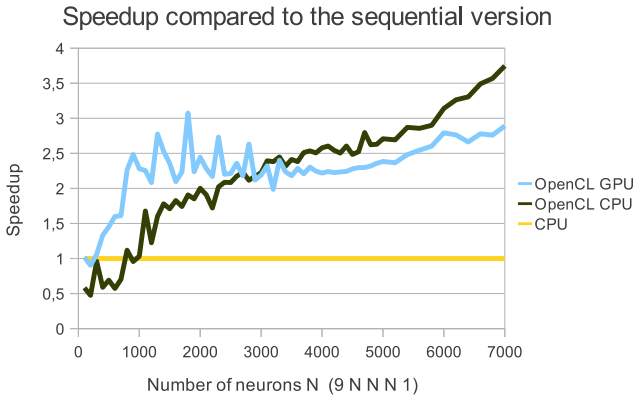

Fig. 3. Parallel BP, speedup for larger number of neurons

B. Parallel PSO algorithm

The parallel PSO algorithm performance is reported with the following annotations:

- **CPU**: sequential PSO, executed on CPU (on a single core);
- **local GPU**: OpenCL implementation on GPU, variant with local memory;
- **private GPU**: OpenCL implementation on GPU, variant with private memory;
- **local CPU**: OpenCL implementation on CPU, variant with local memory;
- **private CPU**: OpenCL implementation on CPU, variant with private memory.

Note that the OpenCL 'local memory', when deployed on a CPU, actually represents the RAM and is equivalent to the OpenCL global memory. This may be implementation dependent, but in practice both local and private memory can be expected to be implemented as global memory on CPU devices. The population size in PSO algorithm is 64 particles.

We found the parallel PSO implementation much more adaptive to different network sizes; the initial results are obtained on a network with number of neurons 9-15-6-1 in 4 layers. The following experiments were performed to test the algorithm efficiency when changing the number of samples, the number of neurons and the number of layers.

The influence of the number of samples is measured by executing a single PSO iteration and averaging over 10 repetitions. The runtimes are given in Fig. 4 and the speedup in Fig. 5. When both the network and the number of samples is relatively small, the parallel GPU implementation is faster than sequential PSO for more than 30 samples, and OpenCL CPU version for more than 120 samples. If the number of samples is even larger, the speedup is clearly visible in Fig. 6. It can be seen from the results that the speedup of the CPU variant of parallel PSO is close to ideal, since the tested CPU contains 6 cores. At about 30000 samples (not shown here) the GPU variant with local memory reaches the maximum speedup value of 32, which does not increase with additional samples.

In the next experiment the number of samples was set to 10000 and the number of neurons in a single hidden layer was varied. The speedup results for this experiment are shown in Fig. 7 and Fig. 8. These results show great variability in speedup for parallel PSO with local GPU memory; remember that this version requires the amount of memory that is equal to twice the number of neurons in the largest layer. With the increase of the layer size, the number of available work units that can be executed in parallel decreases, which causes this version to be slower than other OpenCL variants for more than 90 neurons in the largest layer (this value of course depends on the GPU characteristics).

The last experiment included 10000 learning samples, variable number of hidden layers and 10 neurons per layer. The speedup results for this case are given in Fig. 9. All the OpenCL variants except the GPU local memory maintain a

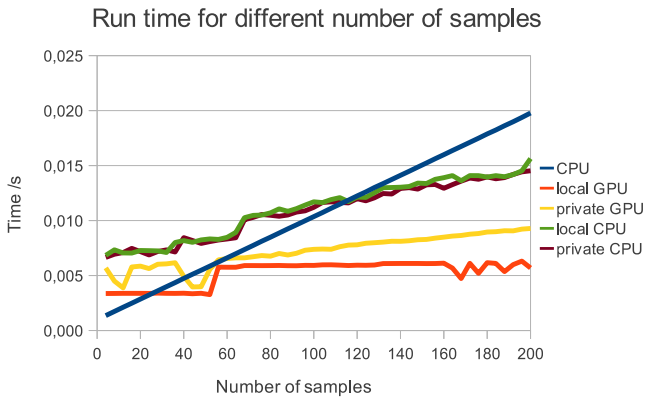

Fig. 4. Parallel PSO, run time for different number of samples

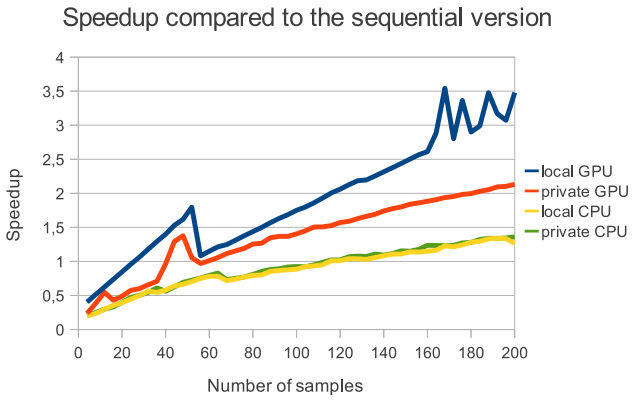

Fig. 5. Parallel PSO, speedup for different number of samples

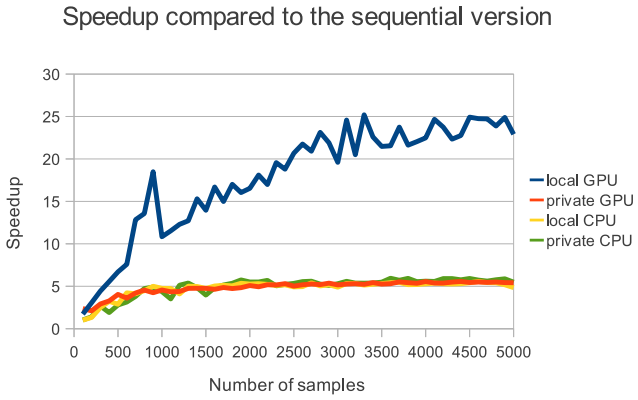

Fig. 6. Parallel PSO, speedup for larger number of samples

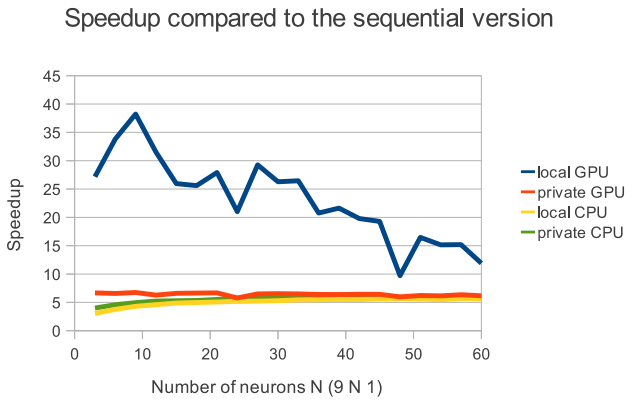

Fig. 7. Parallel PSO, speedup for different number of neurons

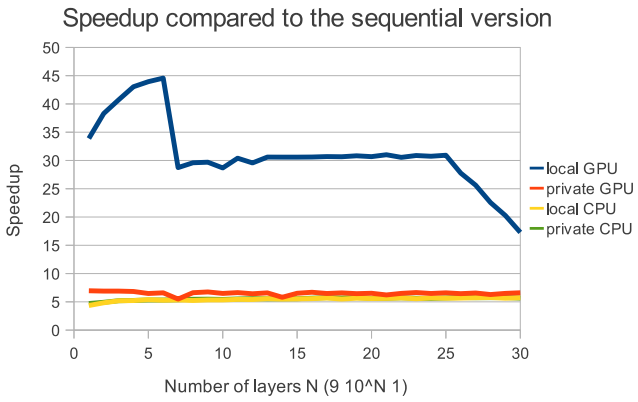

Fig. 9. Parallel PSO, speedup for different number of hidden layers

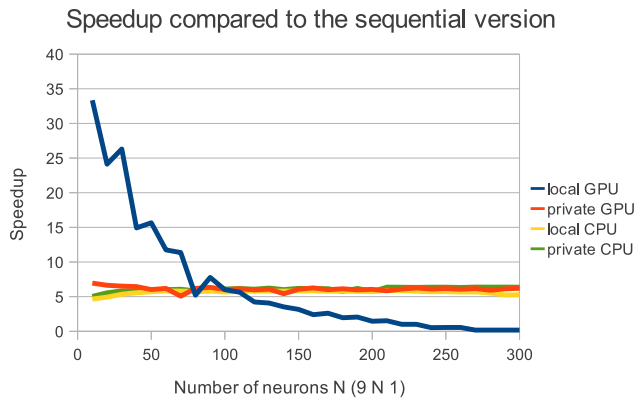

Fig. 8. Parallel PSO, speedup for larger number of neurons

stable speedup of about 6. The version with local memory, on the other hand, shows great variability depending on the number of layers and the required amount of memory. As the memory footprint is increased, less work groups can fit in a compute unit at the same time, thus lowering the parallelism. Finally, with more than 25 layers the needed amount of memory causes the decrease in the number of available work items per workgroup, which is visible as a speedup decrease.

V. Conclusions

This paper describes two implementations of parallel neural network training algorithms with OpenCL. The programming model allows the implementations to be deployed on different platforms; although the maximum efficiency is obtained on a GPU device, the program can also be executed on a general purpose CPU.

Training with parallel backpropagation proved to be efficient only with large networks and is not recommended for smaller networks and smaller number of samples. Parallel training with PSO, on the other hand, is much more applicable in various conditions regarding the number of samples and network size and structure. Two versions of parallel PSO are described, which differ by the type of memory primarily used for calculation. The local memory version shows the best overall speedup results, but may be highly dependent on the network parameters. The version with private memory exhibits a smaller speedup, but is less sensitive to parameter changes.

References

[1] A. Pétrowski, G. Dreyfus, and C. Girault, "Performance analysis of a pipelined backpropagation parallel algorithm," *IEEE Transactions on Neural Networks*, 1993.

[2] R. Uetz and S. Behnke, "Large-scale object recognition with cuda-accelerated hierarchical neural networks," in *Proceedings of the 1st IEEE International Conference on Intelligent Computing and Intelligent Systems*, 2009.

[3] X. Sierra-Canto, F. Madera-Ramírez, and V. Uc-Cetina, "Parallel training of a back-propagation neural network using cuda," in *Proceedings of the Ninth International Conference on Machine Learning and Applications*, 2010.

[4] B. D. Bašić, M. Čupić, and J. Šnajder, "Umjetne neuronske mreže," Faculty of Electrical Engineering and Computing, Zagreb, 2008.

[5] J. Fulcher and L. C. Jain, *Computational Intelligence: A Compendium*. Springer, 2008.

[6] S. Lončarić, "Neuronske mreže - predavanja," Faculty of Electrical Engineering and Computing, Zagreb, 2010.

[7] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of IEEE International Conference on Neural Networks*, 1995.

[8] M. Čupić, "Prirodom inspirirani optimizacijski algoritmi," Faculty of Electrical Engineering and Computing, Zagreb, 2010.

[9] Khronos OpenCL Working Group, *The OpenCL Specification version 1.1*, Khronos Group, 2010.

[10] (2011, June) OpenCL and the AMD APP SDK. [Online]. Available: http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx

[11] *OpenCL Programming Guide for the CUDA Architecture*, NVIDIA, 2010.