Παράλληλα και κατανεμημένα συστήματα Υπολογιστών
Άσκηση 4

Τομέας Ηλεκτρονικής και Υπολογιστών

Τμήμα ΗΜΜΥ 7ο Εξάμηνο Α.Π.Θ. 2016-2017

Λαμπρίδης Αλέξανδρος 6965 (alamprid@ece.auth.gr)

Μάστορας Ραφαήλ Ευάγγελος 7918 (rmastora@ece.auth.gr)

Οκτώβριος 2017

Περιεχόμενα.

Εισαγωγή.

Η αναφορά αυτή είναι για την τέταρτη άσκηση στο μάθημα Παράλληλα και Διανεμημένα συστήματα υπολογιστών. Στα πλαίσια της τέταρτης εργασίας δόθηκε το πρόβλημα της επιτάχυνσης της εκπαίδευσης ενός νευρωνικού δικτύου. Επιπρόσθετα η εκπαίδευση ενός νευρωνικού δικτύου γίνεται με διάφορους αλγόριθμους. Σαν υλοποίηση επιλέχθηκε ο αλγόριθμος back-propagation.

Ο αλγόριθμος back-propagation συνοπτικά ακολουθεί την εξής μέθοδο:

1. Τα βάρη των νευρώνων και των εισόδων στους νευρώνες αρχικοποιούνται με μια τυχαία τιμή
2. Εισάγονται στο νευρωνικό οι τιμές των εισόδων και υπολογίζεται η έξοδος από το νευρωνικό
3. Υπολογίζεται το σφάλμα της εξόδου από το νευρωνικό και ξεκινάει η διαδικασία διόρθωσης των βαρών από το επίπεδο εξόδου (output layer)
4. Οι διορθώσεις των βαρών του επιπέδου εξόδου και το σφάλμα πέρνα στα εσωτερικά επίπεδα μέχρι και το επίπεδό εισόδου όπου σε κάθε επίπεδο επαναλαμβάνεται η ίδια διαδικασία διόρθωσης.
5. Τα παραπάνω βήματα επαναλαμβάνονται τόσες εποχές όσες χρειάζονται για φτάσει το σφάλμα εξόδου για τις εισόδους σε ικανοποιητικό βαθμό.

Από την περιγραφή της παραπάνω μεθόδου είναι προφανές ότι η διαδικασία της εκπαίδευσης είναι μια χρονοβόρα διαδικασία η οποία εξαρτάται από το συγκεκριμένο πρόβλημα που καλείται να αντιμετωπίσει το νευρωνικό σύστημα.

Τα μέρη του αλγόριθμού back-propagation μπορούν να παραλληλοποιηθούν με διάφορους τρόπους. Αρχικά λόγω της δομής αλγορίθμου η παραλληλοποίηση γίνεται στο κάθε επίπεδο αφού για να υπολογιστούν τα βάρη των νευρώνων ενός επίπεδου χρειάζονται τα βάρη των νευρώνων του προηγούμενου επιπέδου. Οπότε οι μέθοδοι παραλληλοποίησης που προτείνονται στην δημοσίευση Parallelization of a Back propagation Neural Network on a Cluster Computer (1) είναι σε επίπεδο του νευρωνικού ή σε νευρώνες ενός επιπέδου. Οι δύο στρατηγικές έχουν κοινό σημείο ότι αφού έχει ολοκληρωθεί το η διαδικασία για την διόρθωση των βαρών στο προηγούμενο επίπεδο είναι δυνατόν για το επόμενο να διαμοιραστεί και να γίνει παράλληλα. Τέλος στην συγκεκριμένη υλοποίηση παρουσιάζεται η στρατηγική παραλληλισμού σε επίπεδο (exemplar Parallelism).

Η στρατηγική παραλληλοποίησης exemplar Parallelism περιληπτικά είναι η εξής:

For epochs

For each worker
do Send Neural Net

Gather new Neural Net weights

Apply Weight Change

Η στρατηγική αυτή έχει σαν κεντρική ιδέα ότι το νευρωνικό δίκτυο εκπαιδεύεται σε διάφορες εισόδους και χρησιμοποιώντας ένα μέσο όρο διορθώνονται τα βάρη του νευρωνικού. Στην επόμενη εποχή συνεχίζει το διορθωμένο νευρωνικό, μετεκπαιδεύεται για διάφορες εισόδους και επαναλαμβάνεται η ίδια διαδικασία. Αυτή η τεχνική μπορεί να γίνει με αρκετούς τρόπους. Ζητούμενο της εργασίας είναι να χρησιμοποιηθεί το πραγματιστικό πρότυπο threads και CUDA.

Τέλος στην αναφορά αυτή περιέχεται η ανάλυση της υλοποίησης, ο έλεγχος ορθότητας, τα χρονικά αποτελέσματα και οτιδήποτε χρησιμοποιήθηκε για την εκπόνηση αυτής της εργασίας.

Περιγραφή της υλοποίησης.

Η στρατηγική παραλληλοποίησης που επιλέχθηκε (exemplar Parallelism) μπορεί να υλοποιηθεί με αρκετούς τρόπους. Σαν τεχνοτροπίες υλοποιήθηκαν το προγραμματιστικό πρότυπο pthreads και η CUDA.

Αρχικά αυτή η στρατηγική μπορεί να χωριστεί στο κομμάτι όπου εκπαιδεύεται το νευρωνικό για τις διάφορες εισόδους και στο κομμάτι όπου οι γίνονται οι υπολογισμοί για την διόρθωση των βαρών του νευρωνικού δικτύου. Οπότε το μέρος το οποίο εκπαιδεύει τα νευρωνικά υλοποιείται από πολλά thread που είναι αδρανή σε ένα threadpool (Α και Β) και το κομμάτι στο οποίο γίνονται οι υπολογισμοί για την διόρθωση των βαρών του νευρωνικού γίνεται από την GPU (Γ). Η παραπάνω διαδικασία αναλύεται με τον παρακάτω ψευδοκώδικα.

```
int main(int argc, char *argv[])
{
   //Init Thread Pool
   ThreadPool tp(NUM_THREADS-1);

      //Epochs
   for(int i=0;i<EPOCHS;i++)
   {
      // Loop to send out work to thread pool
      for(int j=1;j<NUM_THREADS;j++)
      {

       if(j!=0){
         Task* t = new Task(&train_wrapper, (void*) neural_net);
         tp.add_task(t);
       }

      }
            //Do work to cover
            train_wrapper, (void*) neural_net
```

(Α)

```
      //Synch method
      while(flag)
        if(pthread_mutex_trylock(&mutexBusy)==0)
                  thread_mutex_unlock (&mutexBusy);

      netMatrix[0].gatherErrors2(netMatrix,j+1);
```

(Β)

```
      //With Cuda
      netMatrix[0].applyBatchCumulations(0.2f,0.1f);
```

(Γ)

```
   }
}
```

Στις παρακάτω ενότητες ακολουθεί η ανάλυση του σειριακού κώδικα, η λογική της pool thread και τέλος η CUDA. Ο κώδικα ο οποίος υλοποίει το neural net είναι από το project https://github.com/danielrioss/bpnet_wpage.

Συνοπτική επεξήγηση του applyBatchCumulations.

Η υλοποίηση αυτή αποτελεί την σειριακή έκδοση της συνάρτησης που υπολογίζει και ανανεώνει τα βάρη. Η συνάρτηση αυτή προϋποθέτει να έχει πραγματοποιηθεί η εκπαίδευση του νευρωνικού και να έχουν κρατηθεί τα αθροίσματα των errors για κάθε νευρώνα του δικτύου. Έπειτα όταν κληθεί υπολογίζει και ανανεώνει τα βάρη του νευρωνικού με βάση τον μέσο όρο error για κάθε νευρώνα μέσω των παρακάτω σχέσεων.

Ανανέωση του κέρδους των βαρών για κάθε νευρώνα

$$\sum neuron \rightarrow wgain\,[i] -= alpha * neuron \rightarrow errorGain[i]/numberOfBatches$$

Ανανέωση των βαρών και delta

$$udelta = alpha * (neuron \rightarrow errorWeight[i]/numberOfBatches) + neuron \rightarrow deltavalues[i][j] * momentum$$

$$\sum neuron \rightarrow weights\,[i][j] += udelta$$

$$neuron \rightarrow deltavalues[i][j] = udelta$$

Αυτός ο αλγόριθμος υλοποιήθηκε από τους συγγραφείς αυτής της εργασίας και χρησιμοποιήθηκε σαν παράδειγμα ο εξής κώδικας http://www.ccodechamp.com/c-program-of-multilayer-perceptron-net-using-backpropagation/

Συνοπτική επεξήγηση pool thread (A+B).

Η δημιουργία μιας pool thread, επειδή η εκπαίδευση κάθε νευρωνικού επαναλαμβάνεται για κάθε εποχή, είναι απαραίτητη αφού έτσι μηδενίζεται το κόστος δημιουργίας νέων thread για κάθε εποχή. Ο κώδικας που χρησιμοποιήθηκε είναι από το project(https://github.com/bilash/threadpool).

Ο κώδικας υλοποιεί μια pool thread η οποία δεν έχει κάποιο μηχανισμό συγχρονισμού. Ο μηχανισμός συγχρονισμού είναι απαραίτητος για να προχώρηση η ροή του προγράμματος από το (A) στο βήμα (B) διότι πρέπει όλα τα thread να έχουν εκπαιδεύσει τα νευρωνικά δίκτυα ώστε να μπορούν να συγκεντρωθούν τα διάφορα βάρη. Ο μηχανισμός δημιουργήθηκε με μια μεταβλητή mutex. Κάθε φορά που ένα thread αναλαμβάνει δουλειά (task) και την φέρνει εις πέρας μειώνει ένα counter και ενεργοποιεί σε ένα πίνακα ένα δείκτη ώστε το main thread να

γνωρίζει ποιό thread έχει ολοκληρώσει το task και είναι έτοιμο για να μεταφέρει τα error. Αυτός ο μηχανισμός εξασφαλίζει ότι θα έχουν υλοποιηθεί όλα τα task της pool thread ώστε στην συνέχεια θα περάσει στον υπολογισμό του μέσου όρου (Γ).

Συνοπτική επεξήγηση του KernelapplyBatchCumulations (Γ).

Στη συνάρτηση αυτή ετοιμάζονται όλα τα απαραίτητα δεδομένα για την κλήση των συναρτήσεων που εκτελούνται στην κάρτα γραφικών. Συγκεκριμένα είναι αναγκαίο να διαμορφωθούν τα στοιχεία των error καθώς και τα βάρη των νευρώνων μαζί με τις υπόλοιπες παραμέτρους στην μορφή μονοδιάστατων πινάκων. Αρχικά αποφασίζεται το μέγεθος κάθε πίνακα ώστε όλοι πίνακες που συμμετέχουν στην ίδια πράξη να έχουν σωστή και γρήγορη προσπέλαση. Έπειτα δημιουργούνται οι πίνακες αυτοί και λαμβάνουν τα στοιχεία από τις δομές των νευρώνων και των επιπέδων. Στο σημείο αυτό υπάρχει έλεγχος ώστε τα κελιά του πίνακα που δημιουργήθηκαν και δεν αντιστοιχούν σε πραγματικές τιμές να μηδενίζονται.

Μετά το σημείο αυτό οι πίνακες μεταφέρονται στην κάρτα γραφικών με τις συναρτήσεις cudaMemcopy και παραμένουν εκεί μέχρι να ολοκληρωθεί η αλλαγή των βαρών. Η λογική που υλοποιήθηκε η συνάρτηση αυτή επιτρέπει μια ελάχιστη χρήση των συναρτήσεων μεταφοράς δεδομένων από και προς την κάρτα γραφικών καθώς είναι αρκετά χρονοβόρα διαδικασία.

Στην συνέχεια μέσω της συνάρτησης gridDimensions επιστρέφονται κατάλληλα μεγέθη grid και block για το πρόβλημα ώστε να γίνεται βέλτιστη κατανομή των threads στην κάρτα γραφικών και όλα τα layers να ανανεώνονται συγχρόνως ενώ παράλληλα αποφεύγονται λάθη διευθυνσιοδότησης .

Ύστερα καλούνται οι πυρήνες cuda_layer_comulations και cuda_layer_comulations2 οι οποίοι υλοποιούν την μεταβολή των βαρών ανάλογα με τα error που προέκυψαν για το τελευταίο batch με βάση τους παρακάτω τύπους

Πυρήνας cuda cuda_layer_comulations:

$$\sum wgain\,[index] -= alpha * errorGain[index]/numberOfBatches$$

Πυρήνας cuda_layer_comulations2:

$$udelta = alpha * (errorWeight\,[index]/numberOfBatches) + deltavalues[index] * momentum$$

$$\sum weights\,[index] += udelta$$

$$deltavalues[index] = udelta$$

Το index που χρησιμοποιείται σε αυτές τις πράξεις βρέθηκε ύστερα από θεωρητική μελέτη του προβλήματος και φαίνεται παρακάτω

$$index = blockDim.y \\ * \left(blockIdx.y + gridDim.y * (threadIdx.x + blockIDx.x * blockDim.x)\right) \\ + threadIdx.y$$

Μόλις ολοκληρωθεί η επεξεργασία των δεδομένων από την κάρτα γραφικών επιστρέφονται τα δεδομένα και ανανεώνονται στις δομές του νευρωνικού.

Αποτελέσματα και περιγραφή ελέγχου ορθότητας.

Έλεγχος ορθότητας.

Ως έλεγχος ορθότητας χρησιμοποιήθηκε ο πίνακας αλήθειας της XOR για διάφορα μεγέθη εισόδων. Συγκεκριμένα στο παρόν πρόγραμμα δίνεται από τον χρήστη ο αριθμός των ψηφίων για τα οποία επιθυμεί να εκπαιδευτεί το νευρωνικό ώστε να παράγει το αποτέλεσμα της XOR. Έτσι για παράδειγμα για 2 εισόδους αρχικά δημιουργείται η κατάλληλη αλληλουχία ψηφίων

$$input = \begin{bmatrix} 0 & , & 0 \\ 0 & , & 1 \\ 1 & , & 0 \\ 1 & , & 1 \end{bmatrix}$$

στον πίνακα input και τα αντίστοιχα αποτελέσματα της XOR μεταξύ των ψηφίων της ίδιας σειράς στον πίνακα desiredout.

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} XOR \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = desiredout = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Τα στοιχεία αυτά δίνονται στο νευρωνικό για εκπαίδευση και μετά από ένα μεγάλο αριθμό εποχών αναμένεται η συνάρτηση propagate(input[i]) να επιστρέφει τη σωστή έξοδο που αντιστοιχίζεται στο desiredout[i]. Παρακάτω παραθέτονται τα αποτελέσματα της εκπαίδευσης του νευρωνικού για μέγεθος 3 εισόδων και στο αρχείο tests.txt παραθέτονται και για 2 και 4 εισόδους τα αποτελέσματα.

```
./backpropagation 3 2 3 1000000 4 2 0
Train net duration : 9.79216
TESTED PATTERN 0 DESIRED OUTPUT: 0 NET RESULT: 0.00321255
TESTED PATTERN 1 DESIRED OUTPUT: 1 NET RESULT: 0.998837
TESTED PATTERN 2 DESIRED OUTPUT: 1 NET RESULT: 0.998586|
TESTED PATTERN 3 DESIRED OUTPUT: 0 NET RESULT: 0.00321386
TESTED PATTERN 4 DESIRED OUTPUT: 1 NET RESULT: 0.998819
TESTED PATTERN 5 DESIRED OUTPUT: 0 NET RESULT: 0.00342432
TESTED PATTERN 6 DESIRED OUTPUT: 0 NET RESULT: 0.00322475
TESTED PATTERN 7 DESIRED OUTPUT: 1 NET RESULT: 0.99841
square_error= 6.25608e-06
```

Τα παραπάνω αποτελέσματα πάρθηκαν χρησιμοποιώντας την CPU για την ανανέωση των βαρών, ωστόσο και με τη χρήση της GPU επιστρέφονται τα ίδια αποτελέσματα ,έχοντας όμως επιβαρυμένους χρόνους για το μικρό μέγεθος των παραδειγμάτων αυτών. Η συνάρτηση της CUDA συνιστάται να χρησιμοποιηθεί για μεγάλο αριθμό hidden layer και νευρώνων καθώς τότε θα φανεί σημαντική βελτίωση σε σχέση με την CPU.

Σε αυτό το σημείο χρειάζεται να αναφερθεί ότι για μεγαλύτερο αριθμό ψηφίων είναι αναγκαίο να βρεθεί η κατάλληλη τοπολογία του δικτύου καθώς και να αυξηθεί σημαντικά ο αριθμός των εποχών προκειμένου το νευρωνικό να συγκλίνει στην επιθυμητή λύση. Η διαδικασία αυτή είναι ιδιαίτερα χρονοβόρα και απαιτεί πολλές δοκιμές για διάφορες παραμέτρους.

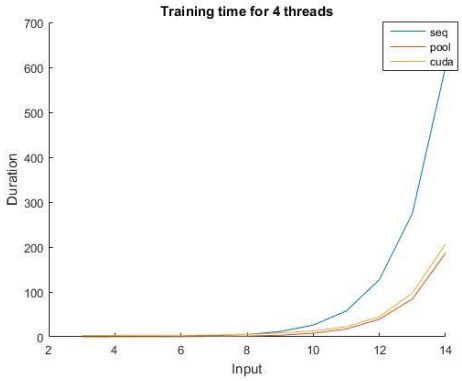
Χρονικά αποτελέσματα.

Σε αυτήν την ενότητα παρουσιάζονται τα χρονικά αποτελέσματα. Οι δοκιμές έγιναν για μικρό αριθμό εποχών, χωρίς να είναι απαραίτητα ο σωστός συνδυασμός τοπολογίας και αριθμός εποχών ώστε να το νευρωνικό δίκτυο να επιστρέφει σωστό αποτέλεσμα. Αυτή η προσέγγιση γίνεται ώστε να φανεί η ταχύτητα της εκπαίδευσης και να καταστεί δυνατό να εξαχθούν συμπεράσματα. Η ορθότητα της υλοποίησης περιγράφεται και αποδεικνύεται στην παράγραφο «έλεγχο ορθότητας».

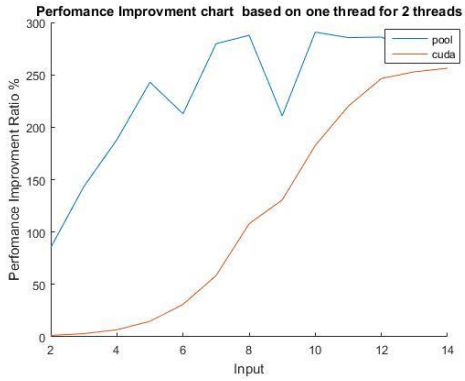
Τα διαγράμματα που ακολουθούν είναι δύο τύπων. Ο πρώτος τύπος εμφανίζει τον απόλυτο χρόνο εκπαίδευσης και τρείς χρωματισμένες καμπύλες. Η καμπύλη *sequential* είναι η καμπύλη του σειριακού αλγόριθμου. Η καμπύλη *pool* αναφέρεται στο κομμάτι του κώδικα (Α και Β) που χρησιμοποιεί pthreads και η apply batch (Γ) γίνεται σειριακά με την συνάρτηση applyBatchCumulations. Η καμπύλη *cuda* αναφέρεται σε στο κομμάτι του κώδικα (Α και Β) που χρησιμοποιεί pthreads και η apply batch (Γ) γίνεται με τον kernelapplyBatchCumulations που χρησιμοποιεί CUDA. Επίσης σαν μεταβλητές στο πρόβλημα χρησιμοποιούνται ο αριθμός των εισόδων και ο αριθμός του συνόλου των νευρώνων. Στον αριθμό των νευρώνων συμπεριλαμβάνεται και ο αριθμός των hidden layer. ο Τέλος οι περισσότερες δοκιμές έγιναν για 10.000 εποχές.

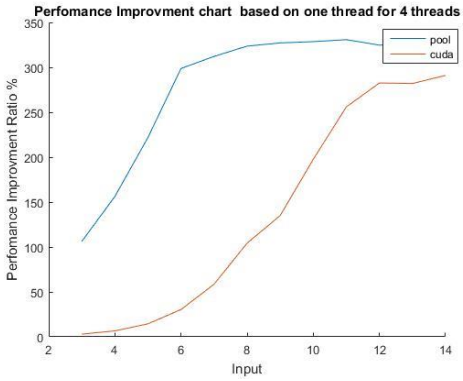
Χρονικά αποτελέσματα για μεταβαλλόμενο μέγεθος εισόδου.

Παρακάτω βλέπουμε την χρονική διάρκεια των train για 10.000 εποχές*$2^{inputsize}$ με μεταβαλλόμενο τον αριθμό των εισόδων στο σύστημα για 2 και 4 threads.



(Στα παραπάνω διαγράμματα ο κάθετος άξονας είναι σε δευτερόλεπτα)



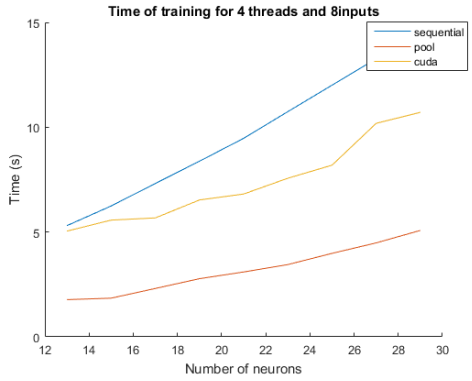


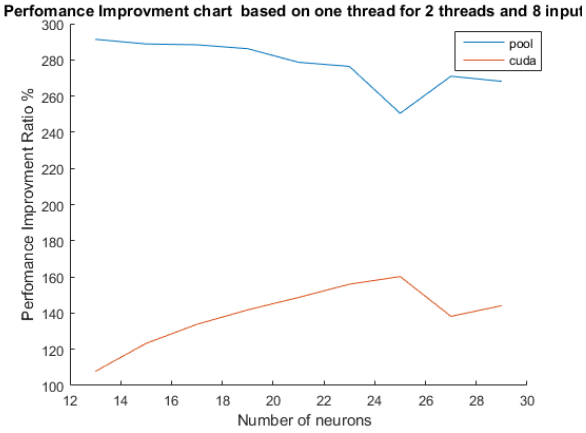
Όπως φαίνεται τόσο η υλοποίηση με pool thread (καμπύλη *pool*) όσο και η υλοποίηση pool thread μαζί με CUDA (καμπύλη *cuda*) δείχνουν βελτίωση σε σχέση με τον σειριακό κώδικα. Η βελτίωση αυτή ξεπερνάει το 250% όσο ανεβαίνουμε πάνω από 12 εισόδους (144 patterns) όπως μπορεί κανείς να δει στα παρακάτω διαγράμματα.

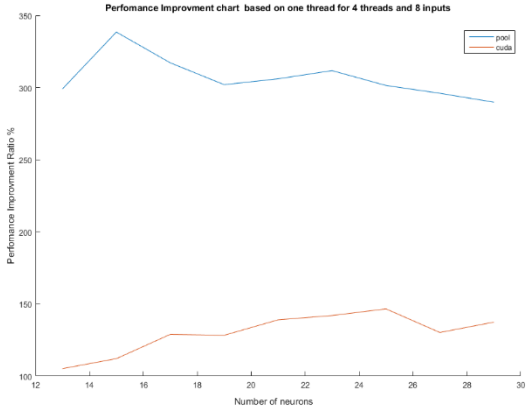
Ειδικά στην υλοποίηση που περιέχει cuda μπορεί να επιβαρύνει αρχικά την επίδοση, ωστόσο όπως φαίνεται ακολουθεί περίπου εκθετική αύξηση της επίδοσης σε σχέση με τον σειριακό και αναμένεται να ξεπερνάει την επίδοση της poolthread για μεγαλύτερες εισόδους όπως επίσης και για πολυπλοκότερο νευρωνικό δίκτυο.

Χρονικά αποτελέσματα για μεταβαλλόμενο μέγεθος νευρωνικού δικτύου.

Στα παρακάτω διαγράμματα παρουσιάζονται τα χρονικά αποτελέσματα για διαφορετικό μέγεθος του νευρωνικού δικτύου. Φαίνεται από τα παρακάτω διαγράμματα το overhead που υπάρχει στην μεταφορά των δεδομένων από την “cpu” στην “gpu”. Άλλο ένα στοιχείο που εμφανίζεται είναι το εξής: δεν ισχύει η υπόθεση ότι η συγκεκριμένη υλοποίηση CUDA είναι γρηγορότερη για πολυπλοκότερο νευρωνικό δίκτυο.

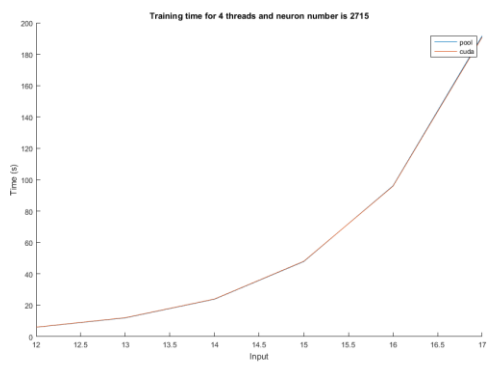


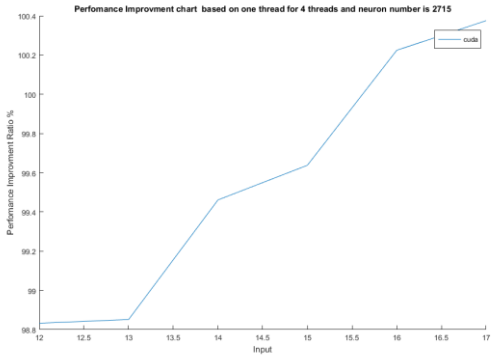




Από τα δύο τελευταία διαγράμματα εμφανίζεται πως όσο αυξάνεται η πολυπλοκότητα του νευρωνικού με την ίδια είσοδο δεν υπάρχει αντίστοιχη αύξηση της επίδοσης του κώδικα POOL με τον κώδικα CUDA. Το οποίο ερμηνεύεται ως εξής: Για πολυπλοκότερο νευρωνικό δίκτυο χωρίς να αυξάνεται ο αριθμός των εισόδων ο φόρτος του train (A) είναι μικρότερος από το φόρτο εργασία του applybatch με Cuda για αυτό και η βελτίωση δεν είναι η αναμενόμενη.

Στα επόμενα διαγράμματα παρουσιάζονται τα αποτελέσματα της εκπαίδευσης ενός πολύπλοκου νευρωνικού με σύνολο περίπου 2715 νευρώνες και μεγάλο αριθμό εισόδων.





Από τα παραπάνω διαγράμματα φαίνεται πως για μεγάλο φόρτο εργασίας η απόδοση του κώδικα POOL και του κώδικα CUDA είναι περίπου ίδια. Συνεπώς η συγκεκριμένη παραλληλοποίηση του κομματιού applykernel με CUDA δεν επαρκεί για να εμφανιστεί σημαντική βελτίωση και αντιθέτως επιφέρει πολύ μεγάλο overhead στα μικρότερα προβλήματα.

Βιβλιογραφία

(1) Parallelization of a Backpropagation Neural Network on a Cluster Computer Mark Pethick, Michael Liddle, Paul Werstein, and Zhiyi Huang Department of Computer Science University of Otago Dunedin, New Zealand. (http://www.cs.otago.ac.nz/staffpriv/hzy/papers/pdcs03.pdf)