2017

Υλοποίηση Νευρωνικού Δικτύου χρησιμοποιώντας Pthreads & Cuda

4Η ΕΡΓΑΣΙΑ ΣΤΑ ΠΑΡΑΛΛΗΛΑ ΚΑΙ ΔΙΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

ΦΙΛΙΠΠΟΣ ΧΡΗΣΤΟΥ 8276
ΣΤΑΥΡΟΣ ΑΝΤΩΝΙΑΔΗΣ 8279

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ | email : fchristou@auth.gr & antoniads@auth.gr

Πίνακας Περιεχομένων

Εισαγωγή

Σκοπός αυτής της εργασίας είναι η παραλληλοποίηση του αλγορίθμου BackPropagation, ο οποίος αποτελεί μέρος του νευρωνικού δικτύου. Η δουλειά που έγινε βασίστηκε στον κώδικα, οποίος μπορεί να γίνει διαθέσιμος μέσω της εντολής git :

`git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git [1]`

Ο αρχικός κώδικας είναι γραμμένος σε python και η περιγραφή του μπορεί να βρεθεί στο http://neuralnetworksanddeeplearning.com/ [2] .

Το πρότζεκτ αυτό πρώτα πρώτα έχει ως σκοπό την παραλληλοποίηση του BP αλγορίθμου με την χρήση Pthreads & CUDA και αφετέρου την επικοινωνία μεταξύ python και C, που χρειάζεται για την πραγματοποίηση του νευρωνικού δικτύου. Αυτή η επικοινωνία και η μεταφορά δεδομένων μεταξύ της python και C έγινε χρησιμοποιώντας την βιβλιοθήκη *ctypes*. Ο σκοπός αυτού του πρότζεκτ δεν είναι η υπολογιστική βελτιστοποίηση του νευρωνικού δικτύου, καθότι για αυτό θα έπρεπε να χρησιμοποιηθούν άλλοι αλγόριθμοι και τεχνικές, αλλά η επίδειξη των Pthreads και Cuda μαζί και η σύνδεση αυτών με την python.

Επεξήγηση Αρχείων Project

Στον φάκελο του πρότζεκτ βρίσκονται 11 αρχεία . Ο μεγάλος αριθμός αυτός, εξυπηρετεί στις περισσότερες δοκιμές για την καλύτερη λήψη συμπερασμάτων.

- network_parallel.py : Είναι το αντίστοιχο network.py του αρχικού κώδικα του git , το οποίο τροποποιήθηκε ώστε να επιτρέπει να καλούνται συναρτήσεις της C.
- runPy.py : Το scriptάκι για την άμεση εκτέλεση του νευρωνικού δικτύου.

- measurements_script.sh : bash script για την αυτόματη λήψη μετρήσεων. Ο αριθμός που αναγράφεται για κάθε μέτρηση αποτελεί μέσο όρο 10 μετρήσεων
- BP_serial.c : Είναι το σειριακό πρόγραμμα σε C, το οποίο υλοποιεί τόσο την συνάρτηση backprop() του αρχικού κώδικα git , όσο και την συνάρτηση update_mini_batch(). Αυτός ο κώδικας θα αποτελέσει το κατά κυρίως σημείο αναφοράς μας ως προς την βελτιστοποίηση των χρόνων.
- BP_serial_wMain.c : Είναι εξίσου ο ίδιος κώδικας με παραπάνω , με μόνη διαφορά ότι περιέχεται main() συνάρτηση και τρέχει ανεξάρτητα της python. Αυτό το αρχείο χρησιμοποιήθηκε για καλύτερο debugging αλλά και για πιο γρήγορα & έγκυρα αποτελέσματα. Πάλι θα αποτελέσει το σημείο αναφοράς μας ως προς την βελτιστοποίηση των χρόνων.
- BP_wPthreads.c : Είναι ο παραπάνω σειριακός κώδικας που υφιστάμενος παραλληλοποίηση με Pthreads.
- BP_wPthreads_wMain: Είναι ο ίδιος κώδικας με τον BP_wPthreads.c , αλλά με main().
- BP_Cuda.cu : Είναι ο σειριακός κώδικας σε C υφιστάμενος παραλληλοποίηση με Cuda.
- BP_wCuda_wMain.cu : Ο κώδικας BP_Cuda.cu με main()
- BP_PthreadsCuda.cu : Ο σειριακός κώδικας υφιστάμενος παραλληλοποίηση τόσο με Pthreads όσο και με Cuda.
- BP_wPthreadsCUDA_wMain.cu : Ο κώδικας BP_PthreadsCuda.cu αλλά με main για να είναι ανεξάρτητος από την python.

Note : Τόσο για τις δοκιμές που έγιναν στον κώδικα με main() , όσο και στις δοκιμές που έγιναν μέσω Python, η ψευδοτυχαία συνάρτηση που χρησιμοποιήθηκε αρχικοποιούνταν σε όλες τις περιπτώσεις με το ίδιο seed (και ούτε γινόταν shuffle() στα δεδομένα εκπαίδευσης). Κάνοντας αυτές τις κινήσεις , σημαίνει ότι έπρεπε να βγάζω τα ίδια αποτελέσματα (τελικός πίνακας biases,weights, αριθμός επιτυχίας ανά epoch, …) για όλες τις περιπτώσεις , όπως και έγινε. Επιπλέων, τα αποτελέσματα μας συμφωνούσαν και με τα αποτελέσματα του εξ'αρχής σειριακού προγράμματος python network.py. Το παραπάνω πιστοποιεί την εγκυρότητα των αποτελεσμάτων για εμάς τους developers.

Compiling & building libraries & running

Η μεταγλώττιση των αρχείων γίνεται ως εξής :

BP_serial_wMain.c → gcc BP_serial.c -lm -o bp_serial.out

BP_wPthreads_wMain.c → gcc -pthreads BP_wPthreads.c -lm -o bp_pthreads.out

BP_wPthreadsCuda_wMain.cu , BP_wCuda_wMain.cu → nvcc –default-stream per-thread $(FILENAME) -lm -o $(FILEOUT)

Η εκτέλεση αυτών γίνεται δίνοντας ως ορίσματα :

1. eta : σταθερά που καθορίζει το βήμα στην Gradient Descent
2. batch_size : σταθερά που καθορίζει το μέγεθος των μικρότερων mini_batches που χωρίζει το training data εξαιτίας της Stochastic Gradient Descent
3. Νευρώνες στο πρώτο επίπεδο
4. Νευρώνες στο δεύτερο επίπεδο
5. Νευρώνες στο τρίτο επίπεδο
6. Νευρώνες στο τέταρτο επίπεδο

7. … κτλ κτλ

Για να τρέξουμε ολόκληρο το νευρωνικό δίκτυο χρειαζόμαστε τα εξής αρχεία που μπορούν να βρεθούν στην ιστοσελίδα του Git [1] : mnist_loader.py & mnist.pkl.gz , όπου το 2^{ο} είναι τα δεδομένα εκπαίδευσης και testing ενώ το πρώτο είναι ο τρόπος φόρτωσης αυτών. Εξαιτίας του μεγάλου μεγέθους τους 2^{ου} δεν επισυνάπτονται ως μέρος του project.

Για να τρέξουμε το νευρωνικό δίκτυο εκτελούμε τις εντολές που βρίσκονται στο runPy.py. Πρώτα όμως πρέπει να έχουμε δημιουργήσει την βιβλιοθήκη που θα χρησιμοποιήσει η python για να καλέσει τις συναρτήσεις της C. Αυτό γίνεται ως εξής :

Πρώτα πρέπει να δώσουμε τιμή στην environment variable LD_LIBRARY_PATH. Αυτό μπορεί να γίνει μέσω της εντολής : export LD_LIBRARY_PATH=/path/to/directory/…

Έπειτα για το χτίσιμο των βιβλιοθηκών χρησιμοποιούμε τις παραπάνω εντολές :

BP_serial.c → gcc -fPIC -shared -o concBP.so BP_serial.c

BP_wPthreads.c → gcc -fPIC -pthread -shared -o concBP.so BP_wPthreads.c

BP_Cuda.cu → nvcc --default-stream per-thread -Xcompiler -fPIC -shared -o concBP.so BP_Cuda.cu

BP_PthreadsCuda.cu → nvcc --default-stream per-thread -Xcompiler -fPIC -shared -o concBP.so \
BP_PthreadCuda.cu

Από την python στην C

Ο κώδικας εντός του αρχείου network.py τροποποιήθηκε ώστε να γίνει ικανός να περάσει δεδομένα στη C. Οι βασικές τροποποιήσεις που έγιναν ήταν η μετατροπή των πολυδιάστατων δομών (weights, biases, training_data, …) σε μονοδιάστατες δομές χρησιμοποιώντας την εντολή flatten(). Ο κώδικας της python συνδέεται με την C , μέσω της C συνάρτηση parallelBP().

Έπειτα εξετάζοντας τον σειριακό κώδικα σε python , τόσο της συνάρτηση backprop όσο και της update_mini_batch() έγινε συγγραφή του αντίστοιχου κώδικα σε C. Αξίζει να αναφερθεί ότι ο κώδικας σε C αποδείχτηκε πολλάκις αργότερος από τον αντίστοιχο κώδικα σε python. Αυτό συνέβη διότι η βιβλιοθήκη numpy της python , καθώς και αρκετές άλλες λειτουργίες της (κυρίως με πίνακες), δρουν παράλληλα και βέλτιστα εκμεταλλεύοντας τους πιο προηγμένους αλγορίθμους και δίνοντας δυνατότητα να μοιραστεί η δουλειά σε διαφορετικούς πυρήνες επεξεργασίας. Έτσι η ανωτερότητα της python , έναντι του απλού προγράμματος σε C που συγγράφηκε είναι δεδομένη (όπως και αποδεικνύεται από τα πειράματα). Για αυτόν τον λόγο θα αποτέλεσει η C σημείο αναφοράς μας και όχι η python.

Σύντομη περιγραφή του Νευρωνικού Δικτύου

Πριν μπούμε στο κυρίως θέμα , χρειάζεται να θυμηθούμε λείαν συντόμως τον τρόπο λειτουργείας του νευρωνικού δικτύου και συνεπώς και του BP αλγορίθμου.

Το νευρωνικό δίκτυο αποτελείται από επίπεδα νευρώνων. Κάθε επίπεδο ενώνεται με το επόμενο μέσο μιας παραμέτρου βάρους (weight parameter). Έτσι, δημιουργείται ο τρισδιάστατος πίνακας weights , ποτ είναι τέτοιος ώστε :

- weights[0][][] είναι ένας πίνακας που συνδέει τα πρώτα 2 επίπεδα.
- weights[i][j][k] είναι το βάρος που συνδέει το κ-οστό νευρώνα του I επιπέδου με τον j-οστό νευρώνα του i+1 επιπέδου.

Επίσης, κάθε νευρώνας από το δεύτερο επίπεδο και μετά διαθέτει ακόμη μια παράμετρο που ονομάζεται πόλωση (bias). Έτσι, ο πίνακας biases δημιουργείται.

Ο σκοπός των νευρωνικών δικτύων είναι η εύρεση των καταλληλότερων παραμέτρων (weights & biases), ώστε το σύστημα να έχει την επιθυμητή συμπεριφορά. Οι βέλτιστοι παράμετροι βρίσκονται με το να ενημερώνονται κατ'επανάληψη, αφού πρώτα έχουν αρχικοποιηθεί σε μια αυθαίρετα τυχαία τιμή.

Ο αλγόριθμος καταφέρνει αυτήν την βελτίωση των παραμέτρων ώστε το σύστημα να πλησιάζει περισσότερο την επιθυμητή συμπεριφορά χρησιμοποιώντας τον αλγόριθμο Stochastic Gradient Descent. Η υλοποίηση του SGD γίνεται μέσω του BP αλγορίθμου και πάει κάπως έτσι :

1. Διαιρούμε το αρχικό σύνολο εκπαίδευσης (training data) σε μικρότερα τμήματα (mini batches) και έπειτα παίρνουν τον μέσο όρο του Gradient Descent σε αυτά. Έπειτα με βάση της μερικές παραγώγους της συνάρτηση κόστους που υπολογίζουμε μπορούμε να ενημερώσουμε τις παραμέτρους weights-biases.
2. Συνεχίζουμε αυτήν την διαδικασία μέχρι να εξαντληθούν όλα τα training data.
3. Εφόσον έχουμε τελειώσει με όλα τα mini batches του training data , τότε έχει περάσει μια εποχή (epoch)
4. Επαναλαμβάνουμε την παραπάνω διαδικασία συνεχίζοντας στην $2^{η}$ εποχή και ούτω το κάθε εξής . Επαναλαμβάνουμε για όσες εποχές θέλουμε μέχρι να είμαστε ευχαριστημένοι με την εκπαίδευση του συστήματος και την συμπεριφορά του.

Ας όμως σταθούμε στο βήμα νούμερο 1. Ο υπολογισμός του SGD γίνεται παίρνοντας σειριακά ένα ένα κάθε ζευγος του training data (raw data input και desirable output) μέχρις ότου τελειώσει το mini batch που έχουμε. Για κάθε ζεύγος training data εφαρμόζουμε τον αλγόριθμο BackPropagation.

Τα κύρια στοιχεία του BP αλγορίθμου είναι. Πρώτα, υπολογίζουμε την έξοδο για κάθε νευρώνα και για κάθε επίπεδο, βασιζόμενοι στην raw data input. Αυτή η διαδικασία λέγεται *feedforward.* Έπειτα, πάμε ανάποδα και υπολογίζουμε για κάθε επίπεδο τις μερικές παραγώγους της συνάρτησης κόστους ως προς τα weights & biases, αρχίζοντας από το τελευταίο προς το πρώτο.

Αυτές οι μερικές παράγωγοι έπειτα αθροίζονται και λαμβάνεται ο μέσος όρους τους, έτσι ώστε να γίνει η ενημέρωση των παραμέτων weights & biases.

Παραλληλοποίηση με Pthreads

Πρώτα, προσπαθούμε να κάνουμε παραλληλοποίηση μέσω της CPU. Ο τρόπος υλοποίησης είναι σχετικά απλός. Αντί να υπολογίζουμε το BP σειριακά για κάθε ζεύγος του training data , διαιρούμε (το

ήδη διαιρεμένο mini batch από το συνολικό training data) σε ακόμα μικρότερες ομάδες chunks. Κάθε chunk ανατίθεται σε ένα διαφορετικό thread και έτσι ο υπολογισμός γίνεται παράλληλα. Φυσικά, ο βέλτιστος αριθμός των chunk θα είναι ο ίδιος με τον αριθμό των logical CPU cores. Για αυτόν τον λόγο χρησιμοποιήθηκε η εντολή *const int Ncores = sysconf(_SC_NPROCESSORS_ONLN) .*

Όλη αυτή ανάθεση εργασίας στα pthreads συμβαίνει στην parallelBP() , ενώ το υπόλοιπο πρόγραμμα μένει ουσιαστικά το ίδιο.

Η βελτίωση που κατορθώθηκε ήταν αναμενόμενα πολύ καλή , και συνάρτηση των πυρήνων που διαθέτει το υπολογιστικό σύστημα. Νούμερα θα επιδειχθούν στην ενότητα Μετρήσεις.

Παραλληλοποίηση με CUDA

Η CUDA όπως γνωρίζουμε εκμεταλλεύεται για να πετύχει βελτίωση του χρόνου υπολογισμού την GPU. Απαραίτητο για να συμβεί αυτό είναι να βρεθεί μια λειτουργία η οποία να επαναλαμβάνεται και να είναι ανεξάρτητη η μια επανάληψη από την προηγούμενη. Στον κώδικα C βρέθηκαν 3 τέτοια σημεία και σε αυτά έγινε παραλληλοποίηση με CUDA.

Πρωτού προχωρήσουμε να γίνει σαφές ότι για έγινε χρήση Cuda Version 9.0. Και όπως αναφέρεται στο https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/ [3] , από την Cuda 7 και μετά δίνεται η δυνατότητα κάθε νήμα του προγράμματος να καλεί τον cuda kernel ασύγχρονα και ανεξάρτητα από τα άλλα νήματα. Αυτό αρκεί εφόσον το αρχείο .cu έχει μεταγλωττιστεί με την εντολή `--default-stream per-thread.`

Συνεπώς δεν χρειάζεται να λάβουμε κάποιο μέτρο για τον συγχρονισμό (ή μάλλον τον άσυγχρονισμό) των διαφορετικών νημάτων που παράγουμε ως προς τα cuda kernels , καθώς αυτό γίνεται αυτόματα.

Η παραλληλοποίηση που έγινε με CUDA συνέβη στα 3 παρακάτω μέρη ως εξής :

1. Μέσα στην backpropagation για τον πολλαπλασιασμό πινάκων :

Ο πολλαπλασιασμός πινάκων με CUDA συμβαίνει στην συνάρτηση kernel matrix_cuda_mullt(). Ο kernel αυτός κάνει χρήση της shared memory για γρηγορότερη προσπέλαση των στοιχείων. Κάθε thread υπολογίζει ένα στοιχείο του νέου πίνακα. Σε περίπτωση που οι πίνακες είναι πολλοί μεγάλοι χρησιμοποιείται η τεχνική του blocking. Δηλαδή, μεταφέρεται στην shared memory ένας υποπίνακας του πραγματικού και την επόμενη φορά ένας άλλος υποπίνακας, ώσπου τελικά μεταφερθούν όλα τα απαραίτητα στοιχεία στην shared memory και γίνουν οι δέουσες πράξεις ώστε να παράξει το κάθε thread το στοιχείο του τελικού πίνακα.

Επίσης για την γρηγορότερη προσπέλαση της global memory έχουμε φροντίσει έτσι ώστε το δεύτερο όρισμα-πίνακας του πολλαπλασιασμού B να παιρνάει στο device ήδη ανεστραμμένος. Αυτό διότι γνωρίζουμε ότι στον πολλαπλασιασμό ο δεύτερος πίνακας προσπελάζεται ανά στήλες. Με αυτόν τον τρόπο οι στήλες είναι σε συνεχόμενες θέσεις μνήμης και άρα προσπελάζεται πιο γρήγορα.

Όπως παρατήρησα από τον matrix_cuda_mullt() kernel η απόδοσή του είναι πολύ καλή για τον πολλαπλασιασμό διασδιάστατων πινάκων. Όταν όμως εμπλέκονται πίνακες γραμμές η απόδοση μειώνεται κατά πολύ. Τόσο πολύ που για μικρό αριθμό στοιχείων ο kernel επιβαρύνει το σύστημα (εξαιτίας των cudaMemcpy που περνάει τα δεδομένα από τον host στο device). Για την καταπολέμηση αυτού του προβλήματος θεωρήθηκε η σταθερά *cuda_threshold* η οποία υποδεικνύει το κατώφλι

υπολογισμών που από αυτούς και πάνω ανατίθενται οι πράξεις στην GPU αλλιώς γίνονται τοπικά στο host machine σειριακά. Για τον κώδικα πολλαπλασιασμού πινάκων με CUDA και shared memory ακολουθήθηκε η λογική που περιγράφεται στο εξής link : http://www.techdarting.com/2014/03/matrix-multiplication-in-cuda-using.html [4]

2. Μέσα στην backpropagation για τον υπολογισμό του Hadamard product και για τις εξισώσεις BP2 BP3 που γίνεται λόγος στο chapter 2 του M. Nielsen [2] :

Εδώ είναι και η απλούστερη περίπτωση CUDA, όπου κάθε thread κάνει μια και μόνο πράξη και έπειτα επιστρέφονται τα δεδομένα στο host machine. Ο kernel που υλοποιεί το παραπάνω είναι ο cuda_BP2_BP3().

3. Για την ενημέρωση των παραμέτρων weights & biases στην parallelBP() :

Ο kernel για την Τρίτη περίπτωση είναι ο cuda_update(). Αυτή η περίπτωση είναι ιδιαίτερη. Αυτό διότι είναι η μόνη περίπτωση στην οποία επιτρέπεται να γίνει το κάλεσμα του kernel ασύγχρονα (non-blocking) και αυτό διότι είναι η μόνη περίπτωση όπου τα δεδομένα είναι πλήρως ανεξάρτητα μεταξύ τους (ο υπολογισμός των weights & biases για το κάθε επίπεδο είναι ανεξάρτητος από τα άλλα επίπεδα). Έτσι μπορούμε να κάνουμε χρήση των **cuda streams**. Για κάθε επίπεδο τόσο για τον υπολογισμό των νέων weights όσο και για τον υπολογισμό των νέων biases δημιουργήθηκε ένα cuda stream, το οποίο

a. Αντιγράφει ασύγχρονα μέσω της cudaMemcpyAsync τα δεδομένα από τον host στο device.
b. Καλεί τον cuda kernel
c. Αντιγράφει πάλι ασύγχρονα μέσω την cudaMemcpyAsunc τα τελικά δεδομένα από το device στον host.

Το πλεονέκτημα του ότι χρησιμοποιούνται τα non-default cuda streams είναι ότι ο κάθε kernel δεν χρειάζεται να περιμένει τον προηγούμενο ώστε να αρχίσει να δουλεύει και ότι μπορεί εφόσον υπάρχει δυνατότητα στην GPU ανάθεσης πόρων να τρέχουν παράλληλα. Επίσης χρησιμοποιώντας την cudaMemcpyAsync δίνουμε την δυνατότητα στην GPU να μεταφέρει παράλληλα από και προς τον host/device περισσότερα δεδομένα.

Pthreads & CUDA

Για την παραλληλοποίηση Pthreads & CUDA μαζί δεν χρειάζεται να κάνουμε τίποτα παραπάνω από το να ενώσουμε τους δύο κώδικες και να προσθέσουμε την επικεφαλίδα #include <pthread.h> στο τελικό αρχείο .cu

Μετρήσεις

Οι μετρήσεις έγιναν για τα προγράμματα που περιέχουν main και κατά συνέπεια ισχύουν αναλογικά και για ολόκληρο το νευρωνικό δίκτυο, καθ΄΄ότι όσο πιο γρήγορα τρέχει η BP τόσο πιο γρήγορα θα τρέχει και το νευρωνικό δίκτυο.

Για να μην μπερδευόμαστε με άλλες παραμέτρους του συστήματος , η σταθερά βήματος eta (η) κρατήθηκε σταθερή και ίση με 3.

No	Mini batch size	Neurons per layer (from first layer to last layer)	Serial (sec)	Pthreads (sec)	Cuda (sec)	Pthreads & Cuda (sec)
1	10	784 30 10	0.007403	0.007684	0.091389	0.079124
2	10	783 783 10	0.153091	0.121160	0.192084	0.177922
3	10	16384 783 10	3.151033	2.403637	2.030224	2.123741
4	10	16384 5000 10	20.57875	14.647388	12.460009	12.246172
5	100	783 783 10	1.418194	0.481671	1.276519	0.602825
6	1000	783 783 10	14.103560	3.927255	12.127905	4.479007
7	1000	16384 30 10	11.211456	3.143376	12.162338	6.250459
8	1000	16384 783 10	294.6669	80.833212	186.060544	67.660768
9	100	783 2000 1000 500 10	0.935271	0.317074	0.994061	0.473501
10	100	2000 1000 500 10	6.021721	1.998673	4.595179	2.062877

Από τον παραπάνω πίνακα μπορούμε να παρατηρήσουμε τα εξής πράγματα :

- Όσο μικρότερο το batch size τόσο χαμηλότερη η βελτίωση που έχουμε με Pthreads. Αυτό είναι λογικό αφού η διαμοίραση της δουλειάς γίνεται με βάση το batch size. Όταν έχουμε μεγαλύτερο batch size περισσότερη δουλειά θα ανατεθεί σε κάθε thread ξεχωριστά και συνεπώς θα έχουμε καλύτερη βελτίωση.
- Όσο μεγαλύτερα του νούμερα των νευρώνων ανά επίπεδο τόσο καλύτερη η βελτίωση με CUDA. Εξίσου το παραπάνω είναι λογικό διότι για μικρά νούμερα νευρώνων το offset που προσδίδει η CUDA (για την αντιγραφή δεδομένων από και προς την GPU) είναι συγκρίσιμο με την ταχύτητα εκτέλεσης. **Συνεπώς για μικρό αριθμό νευρώνων δεν ωφελεί η CUDA**. Βλέπουμε και από της μετρήσεις ότι οι χρόνοι της CUDA είναι αρχικά χειρότεροι από τον σειριακό και όσο αυξάνουμε τους νευρώνες ανά επίπεδο γίνονται καλύτεροι και από τα Pthreads (No4).
- Τέλος παρατηρούμε ότι για να υπάρχει βελτίωση ως προς την χρήση και CUDA και Pthreads απαιτείται να έχουμε και μεγάλο batch size αλλά και πολλούς νευρώνες ανά επίπεδο (No7).

Εν κατακλείδι

Τέλος , βλέπουμε πως τόσο η CUDA όσο και τα Pthreads είναι ένας πολύ καλός τρόπος παραλληλοποίησης. Αυτό όμως δε σημαίνει ότι πρέπει να τα χρησιμοποιούμε και πάντα. Κάθε φορά πρέπει ανάλογα με το πρόβλημα που έχουμε και τις εισόδους που σκοπεύουμε να βάλουμε να γράφουμε και την αντίστοιχη παραλληλοποίηση.

Αναφορές

[1] «GitHub,» [Ηλεκτρονικό]. Available: https://github.com/mnielsen/neural-networks-and-deep-learning.git.

[2] M. Nielsen, «Neural Networks and Deep Learning,» [Ηλεκτρονικό]. Available: http://neuralnetworksanddeeplearning.com/.

[3] M. Harris, «NVIDIA ACCELERATED COMPUTING,» 22 1 2015. [Ηλεκτρονικό]. Available: https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[4] M. Doijade, «TechDarting,» [Ηλεκτρονικό]. Available: http://www.techdarting.com/2014/03/matrix-multiplication-in-cuda-using.html.