Παράλληλα και Κατανεμημένα Συστήματα

Ευάγγελος Τσιμπούρης 8257

Εργασία 4 Πρόταση 1

Ευάγγελος Τσιμπούρης
8257

Περιεχόμενα

Ευάγγελος Τσιμπούρης
8257

Γενική Υλοποίηση

1.1 Ζητούμενα

Τα ζητούμενα της εργασίας ήταν να παραλληλοποιηθεί ο αλγόριθμος backpropagation ενός νευρωνικού δικτύου για την αναγνώριση χειρόγραφων αριθμών. Για την υλοποίηση χρησιμοποιήθηκε cuda (έκδοση PyCuda γιατί ο κώδικας ήταν σε python) και pthreads.

1.2 Πρώτη υλοποίηση

Στην σειριακή έκδοση του αρχείου network.py που περιλαμβάνει και την backprop γίνεται χρήση ενός αλγορίθμου Stohastic Gradient Descent.Ο αλγόριθμος αυτός στο κομμάτι που μας ενδιαφέρει ανανεώνει τα weights και biases των νευρώνων.

Η ανανέωση αυτή γίνεται σε «πακέτα» (mini_batches) μεταβλητού μεγέθους. Ο κώδικας προσπελαύνει τα περιεχόμενα x,y των m.batches **σειριακά** και κάνει υπολογισμούς οι οποίοι είναι **ανεξάρτητοι** για κάθε ζεύγος x,y. Οι υπολογισμοί περιλαμβάνουν κυρίως τη χρήση της μεθόδου backprop.

Επομένως η πρώτη σκέψη που έγινε ήταν για κάθε x,y του m.batch η backprop να τρέχει κάθε φορά ως **ξεχωριστό thread**.

Άρα θα έχουμε 5-10 backrop threads να τρέχουν (όσο το δυνατόν) παράλληλα.(5-10 το μέγεθος του m.batch που επιλέχθηκε για τους υπολογισμούς)

Στη συνέχεια η backprop χρησιμοποιεί γραμμικά συστήματα για την εύρεση των nablas (δηλαδή των παραγώγων των weights και biases). Οι πιο «αργοί» υπολογισμοί της backprop βρέθηκε ότι είναι κυρίως τα γινόμενα πινάκων. Σε αρκετά σημεία ο κώδικας υπολογίζει τα γινόμενα με τη χρήση της μεθόδου dot της numpy. Τα μεγέθη των γινομένων των πινάκων είναι του τύπου:

*784x(αριθμός νευρώνων)**dot**(αριθμός νευρώνων)x1*

(Αυτή είναι η μια από τις περιπτώσεις)

Η χειρότερη περίπτωση είναι:

*(784x1)**dot**(1x(αριθμός νευρώνων))*

Οι παραπάνω περιπτώσεις μεταφέρθηκαν στην cuda. Είναι εύκολο να δει κανείς ότι η χειρότερη περίπτωση δημιουργεί ένα πίνακα μεγέθους 784x(αριθμός νευρώνων) ο οποίος στη cuda μπορεί να δημιουργηθεί με **μια** κίνηση!

Συνοπτικά λοιπόν αρχικά έχουμε threads της backprop να τρέχουν παράλληλα. Σε κάθε thread δημιουργήθηκαν cuda kernels που αντικατέστησαν τα «αργά» γινόμενα πινάκων.

1.3 Προβλήματα και αντιμετώπιση

Αν και η αρχική ιδέα δεν άλλαξε κατά τη διάρκεια της υλοποίησης υπήρξαν διάφορα προβλήματα κυρίως στο κομμάτι των matrix multiplication της numpy και της cuda:

1. Το πακέτο numpy για python περιέχει μεθόδους για επιστημονικούς υπολογισμούς. Οι μέθοδοι της numpy όμως είναι optimized και τρέχουν παράλληλα.(Εύκολο να το δεις από το system monitor, όλοι οι 8 πυρήνες ήταν στο 100%). Επομένως να προσπαθήσεις να «κερδίσεις» σε χρόνο τις μεθόδους αυτές δεν είναι εύκολο.

   Αντιμετώπιση: Στην αρχική έκδοση του προγράμματος χρησιμοποιήθηκαν 30-50 νευρώνες. Ανακάλυψα ότι σε αυτά τα μικρά μεγέθη η numpy ήταν «αήττητη» . Επομένως προσάρμοστήκε το πρόγραμμα να τρέχει με μεγάλα μεγέθη όπως 40.000 ενδιάμεσους νευρώνες.

   Ίσως αυτό στην περίπτωση που αναλύουμε εικόνες με 28x28 pixel να είναι υπερβολικό. Εκεί όμως που μπορεί να είναι χρήσιμο είναι αν αναλύαμε εικόνες με μεγαλύτερη ευκρίνεια όπως 128x128 ή 256x256.

   Επομένως η προσαρμογή αυτή να μην είναι τόσο άχρηστη. Δηλαδή ίσως να μπορεί να χρησιμοποιηθεί κάπου και δεν έγινε μόνο για να κερδηθεί σε χρόνους η numpy.

2. Ένας από τους λόγους που τα kernels της py-cuda ήταν πιο αργά από την numpy σε μικρά νούμερα είναι το memory overhead. Ανακαλύφθηκε ότι το memory allocation στην κάρτα γραφικών και η μεταφορά δεδομένων χρειάζεται ένα minimum χρόνο ο οποίος στην περίπτωση μας καθυστερούσε τους υπολογισμούς. Για να το αντιμετωπίσω χρησιμοποιήθηκαν 3 εναλλακτικές υλοποιήσεις άλλα και οι 3 έδιναν παρόμοιους χρόνους. Επομένως και αυτό οδήγησε στην αλλαγή που ανέφερα πιο πάνω.

1.4 Τελική Υλοποιήση

Η τελική έκδοση λοιπόν διατήρησε την αρχική ιδέα αλλά ταυτόχρονα έφερε την αναγκαστική αλλαγή της μεγάλης αύξησης του αριθμού των νευρώνων.

Σημαντικά τμήματα κώδικα

Σε αυτό το κομμάτι αναφέρονται τα βασικά τμήματα του κώδικα που άλλαξαν και τα τμήματα που τμήματα που δημιουργήσαν προβλήματα.

1. **class backprop(threading.thread)**

   Η backprop μετατράπηκε σε ξεχωριστή κλάση. Αυτό ήταν απαραίτητο ώστε να τρέχει ως ξεχωριστό thread και να περιέχει και ταυτόχρονα cuda kernels. (Από ότι βρέθηκε στο web αυτό χρειάζονταν ώστε να έχει το δικό της context).

2. **delta = np.dot(self.weights[-l+1].transpose(), delta) * sp**
   Η γραμμή αυτή αντικαταστάθηκε από ένα cuda kernel για τον υπολγισμό του γινομένου. Κάτω το «ζουμί» του kernel:

```
for(j = 0;j < 10;j++){
                    nabla_b[i] = w[i+j*N]*delta[j] + nabla_b[i];
}
```

   nabla_b το δευτερο στοιχείο της list delta.
   w τα self.weights.

   Σχόλιο: γενικά καλό ειναί να μην υπάρχει for μέσα σε kernel, εδώ χρησιμοποιήθηκε γιατί είναι μόνο 10 επαναλήψεις σταθερά.
   (το *sp έγινε σε μια γραμμή πιο κάτω στο kernel).

3. **nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())**
   Η γραμμή αυτή δίνει και το μεγαλύτερο κέρδος από την παραλληλοποιήση γιατί στο kernel γίνεται σε μια γραμμή!

```
nabla[i*M+j] = delta[i]*a[j];
```

   Σχόλιο: προφανώς η κάρτα γραφικών θα χρειαστεί να τρέξει πολλά threads ανα block ανα grid ωστέ να επιτευχθεί τέλεια παραλληλοποιήση.

Ευάγγελος Τσιμπούρης
8257

4.
```
queue = Queue.Queue()
            .
            .
            .

delta_nabla_w = queue.get()
delta_nabla_b = queue.get()
```

Και μέσα στην backprop:

```
mutex.acquire()
self.queue.put(self.delta_nabla_w)
self.queue.put(self.delta_nabla_b)
mutex.release()
```

Για την αποθήκευση των αποτελεσμάτων των threads της backpropagation χρησιμοποιήθηκε μια ουρά **queue.**
Για να περαστούν τα αποτελέσματα των εκάστοτε threads χρησιμοποιήθηκε **mutex** για την αποφυγή αλλαγή της σειράς w και b και πιθανώς και άλλων προβλημάτων.

5.
```
for x, y in mini_batch:
 gpu_thread = backprop(...args...)
```

Η backprop τρέχει ως ξεχωριστό thread πλέον (το gpu είναι γιατί εμπεριέχει kernels στο εσωτερικό της και όχι γιατί τρέχει εξ ολοκλήρου στην gpu). Το μέγεθος του mini_batch είναι 10 στην default υλοποίηση αλλά πάρθηκαν και χρόνοι και για 5. Το 10 δημιουργούσε overhead επειδή οι πυρήνες είναι 8 και 10 > 8.
Με 5 υπάρχει καλύτερη επιτάχυνση.

6. Η παρακάτω γραμμή του αρχικού κώδικα(και για τα w):

```
nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
```

ήταν αργή και αντικαταστάθηκε από την ισοδύναμη:

```
map(add, nabla_w, delta_nabla_w)
```

Το πρόβλημα ήταν ότι δεν λειτουργούσε καλά για μεγάλους πίνακες, ενώ η map με τον operator add ήταν αρκετά πιο γρήγορη.
Δεν αποτελεί μέρος της παραλληλοποιήσης αλλά ήταν ενοχλητική στις μετρήσεις.

Ευάγγελος Τσιμπούρης
8257

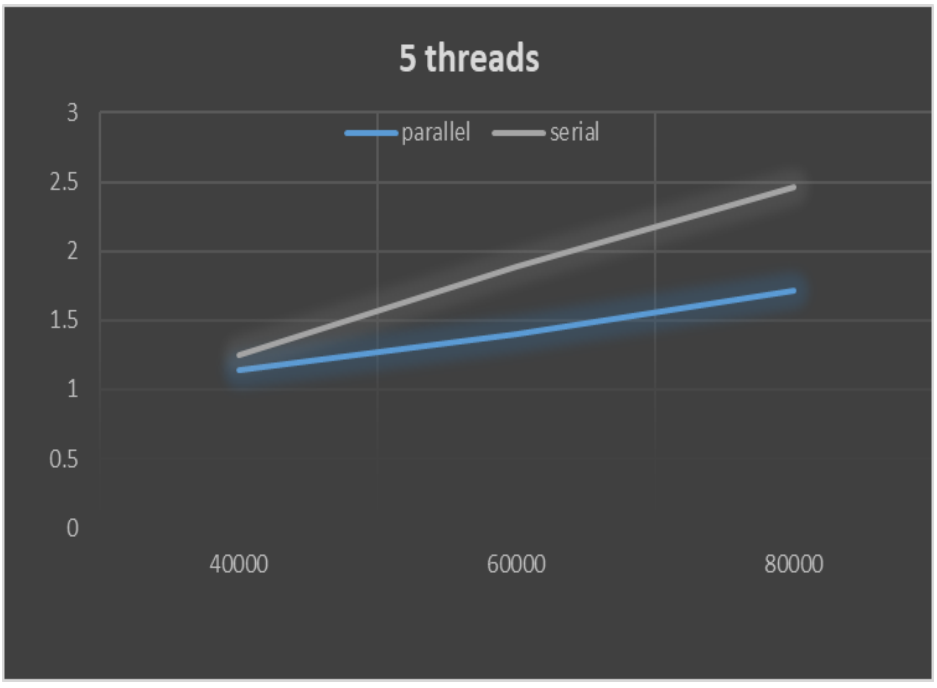
Αποτελέσματα και χρόνοι

Για τα αποτελέσματα χρησιμοποιήθηκε αποκλειστικά το παρακάτω σύστημα:

- i7 4ης γενιας 4-cores (8 με hyperthreading)
- GPU 860M 4GB Nvidia
- 8 GB RAM

Η μνήμη έχει αρκετή σημασία στην συγκεκριμένη υλοποίηση γιατί για κάθε mini batch update δημιουργείται μια queue αρκετά μεγάλη. Επίσης όπως ανέφερα οι ενδιάμεσοι νευρώνες είναι πολλοί με αποτέλεσμα και οι πίνακες να είναι μεγάλοι.

Σχόλια για τους χρόνους

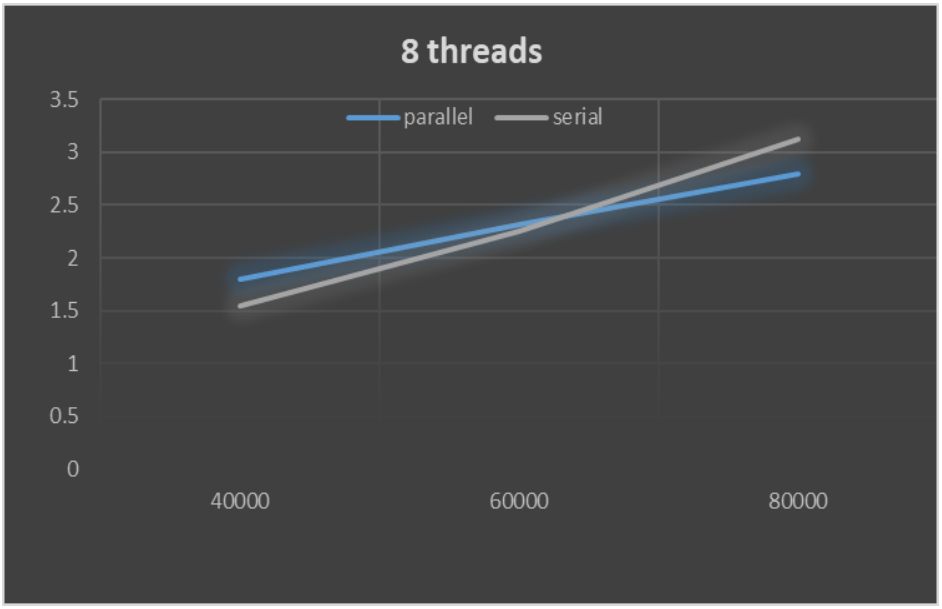


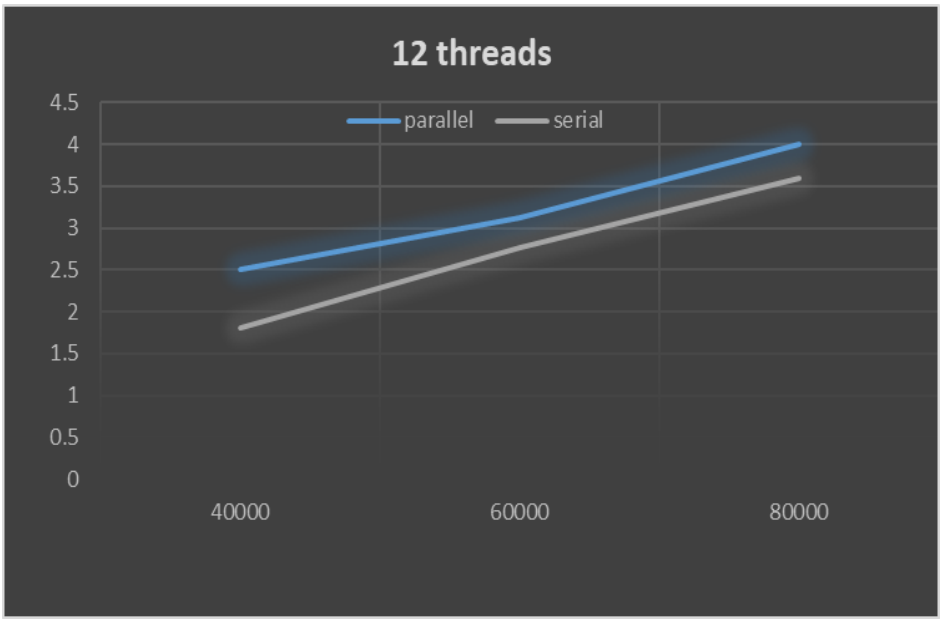
Η βελτίωση είναι ανάλογη του αριθμού των ενδιάμεσων νευρώνων.

Για 80 χιλιάδες η βελτίωση είναι περίπου 32%.

- Οι τιμές του άξονα y είναι σε seconds.
- Ο άξονας x περιέχει τον αριθμό εισόδου των ενδιάμεσων νευρώνων.
- Οι χρόνοι αφορούν την ανανέωση των τιμών ενός mini batch.

Ευάγγελος Τσιμπούρης
8257





Η βελτίωση για 8 threads είναι αρκετά μικρότερη λόγω memory overhead.

Για 80 χιλιάδες η βελτίωση είναι περίπου 9%.

Για 12 threads έχουμε και CPU overhead.

Η υλοποίηση εδώ δεν είναι λειτουργική και δεν φέρνει βελτίωση.

Ευάγγελος Τσιμπούρης
8257

Προβλήματα και βελτιώσεις

- Το κυριότερο πρόβλημα που δεν ξεπεράστηκε ποτέ ήταν το memory overhead που δημιουργούταν κάθε φορά που αρχικοποιούνταν το kernel. Το πρόβλημα αυτό οφείλεται μάλλον στις υλοποιήσεις και το υλικό που βρέθηκε από το internet. Το οποίο μάλλον ήταν ελλιπές η λάθος.(Πιο συγκεκριμένα στην βιβλιογραφία).

- Η χρήση των pthreads για την πλήρη παραλληλοποιήση της backprop για κάθε ζεύγος x,y ενός mini batch έφερε βελτίωση αλλά μόνο για την περίπτωση ενός πολύ μεγάλου δικτύου. Στην ουσία αυτό έγινε ώστε ο χρόνος αρχικοποίησης να καταστεί αμελητέος μπροστά στον καθαρά υπολογιστικό χρόνο.

- Η παραπάνω υλοποίηση δεν έχει ιδιαίτερο νόημα στον συγκεκριμένο αλγόριθμο αλλά θα μπορούσε να χρησιμοποιηθεί σε μεγαλύτερα νευρωνικά δίκτυα.

- Τέλος η υλοποίηση αυτή έγινε έτσι ώστε να καλύπτονται τα ζητούμενα της εργασίας για ταυτόχρονη χρήση pthreads και cuda, και ταυτόχρονα να έχει μια κάποια βελτίωση στους χρόνους για συγκεκριμένες εισόδους.

Βιβλιογραφία

https://wiki.tiker.net/PyCuda/Examples/MultipleThreads

https://wiki.tiker.net/PyCuda/Examples/MatrixmulSimple

https://stackoverflow.com/questions/8340805/pycuda-cuda-some-questions-and-a-simple-code-that-gives-me-error-identifier

https://stackoverflow.com/questions/41743157/understanding-and-optimising-threads-blocks-and-grids-in-pycuda

Κάποιές υλοποιήσεις δεν περιλαμβάνονται στον τελικό κώδικα, απλά δοκιμάστηκαν κάποια στιγμή. Ο τελικός κώδικας περιλαμβάνει μόνο την πιο γρήγορη έκδοση.

Ευάγγελος Τσιμπούρης
8257