Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
ΔΠΜΣ Επιστήμη Δεδομένων και Μηχανική Μάθηση

Επιτάχυνση εκπαίδευσης νευρωνικού δικτύου σε αρχιτεκτονικές κοινής μνήμης με OpenMP και CUDA

Αλέξανδρος Βυθούλκας
alvythoulkas@protonmail.com
03400045

Δημήτρης Ζερκελίδης
dimzerkes@gmail.com
03400049

Μαρία Καϊκτζόγλου
makaiktzoioglou@gmail.com
03400052

Θάνος Πάντος
pantos.thn@gmail.com
03400026

Σωτήρης Πελέκης
pelekhs@gmail.com
03400069

Σπύρος Πούρος
spyros.poulos@gmail.com
03400071

2020
Ιούνιος

Περιεχόμενα

1 Σκοπός της Εργασίας

Σκοπός της εργασίας είναι η εφαρμογή μεθόδων παραλληλοποίησης για την εκπαίδευση νευρωνικών δικτύων με χρήση διαφόρων μεθόδων σε CPU και GPU. Συγκεκριμένα, εξετάζεται το πρόβλημα αναγνώρισης χειρόγραφων αριθμητικών ψηφίων από τη βάση δεδομένων MNIST, η οποία περιλαμβάνει 70.000 εικόνες ψηφίων διαστάσεων 28x28 οι οποίες ισοδυναμούν με διανύσματα μήκους 784 χαρακτηριστικών ως είσοδο της αρχιτεκτονικής βαθιάς μάθησης. Δεδομένου ότι ο πολλαπλασιασμός πινάκων αποτελεί την πιο κοστοβόρα διαδικασία κατά την εκπαίδευση νευρωνικών δικτύων είναι πολύ σημαντικό να επιδιωχθεί η μέγιστη δυνατή επιτάχυνση και βελτιστοποίηση της και αυτό είναι το αντικείμενο της εργασίας αυτής. Ως γνωστόν, η πράξη του πολλαπλασιασμού πινάκων είναι "embarassingly parallel" δεδομένου ότι τα στοιχεία του τελικού πίνακα εξόδου μπορούν να υπολογιστούν εντελώς ανεξάρτητα μεταξύ τους. Η παραλληλοποίηση αυτή υλοποιείται σε CPU και GPU τροποποιώντας αντίστοιχα τον κώδικα (linalg.c & linalg.cu) που δίνεται στα πλαίσια τις εργασίας, ενώ εξετάζονται διάφοροι τρόποι βελτιστοποίησης σε κάθε περίπτωση. Όσον αφορά τη CPU εξετάζεται η απόδοση α) του προγραμματιστικού μοντέλου OpenMP συμπληρώνοντας γραμμές κώδικα σε μια απλή υλοποίηση του πολλαπλασιασμού πινάκων σε C και β) της βιβλιοθήκης παραλληλοποίησης OpenBlas και πιο συγκεκριμένα της συνάρτησης πολλαπλασιασμού πινάκων cblas_dgemm. Στην περίπτωση της GPU εξετάζονται αντίστοιχα η επίδοση α) μιας η υλοποίησης σε Cuda με αποδοτική αξιοποίηση της shared memory και β) η υλοποίηση με χρήση της βιβλιοθήκης cuBlas και πιο συγκεκριμένα των συναρτήσεων cublasDgemm (matrix multiplication) cublasDgeam (matrix addition). Στα πλαίσια λοιπόν της εργασίας εξετάζεται η βελτιστοποίηση, με χρήση παραλληλοποίησης, των παρακάτω 3 πράξεων πολλαπλασιασμού πινάκων οι οποίες αποτελούν τον πυρήνα των forward και backward steps της εκπαίδευσης ενός νευρωνικού δικτύου.

$$\mathbf{C}_{M\times N} = \mathbf{A}_{M\times K} \cdot \mathbf{B}_{K\times N} \tag{1}$$

$$\mathbf{C}_{M\times N} = \mathbf{A}^{\top}_{K\times M} \cdot \mathbf{B}_{K\times N} \tag{2}$$

$$\mathbf{D}_{M\times N} = \mathbf{A}_{M\times K} \cdot \mathbf{B}^{\top}_{N\times K} + \mathbf{C}_{M\times N} \tag{3}$$

2 Παραλληλοποίηση σε CPU

Σε αυτό το σημείο γίνεται η διαδικασία παραλληλοποίησης του πολλαπλασιασμού πινάκων με χρήση της CPU. Αυτό θα επιτευχθεί με χρήση των δύο παρακάτω εργαλείων:

- Προγραμματιστικό μοντέλο OpenMP
- Ειδική βιβλιοθήκη OpenBLAS

Μέσω της χρήσης των παραπάνω μεθόδων παραλληλοποίησης επιτυγχάνεται το υπολογιστικό κόστος του πολλαπλασιασμού πινάκων να κατανέμεται σε επιμέρους threads ενός μεγαλύτερου συστήματος με περισσότερους πυρήνες.

2.1 OpenMP

Το προγραμματιστικό μοντέλο του OpenMP έχει αναπτυχθεί για τις γλώσσες προγραμματισμού C, C++ και Fortran. Χρησιμοποιώντας τις κατάλληλες εντολές, ο χρήστης έχει τη δυνατότητα να επιλέξει την περιοχή προς παραλληλοποίηση, τον αριθμό των threads που θα αναλάβουν την παράλληλη υλοποίηση όπως επίσης και τα δικαιώματα προσβασιμότητας των threads στις διάφορες μεταβλητές της περιοχής προς παραλληλοποίηση. Στην ενότητα αυτή, λοιπόν προχωρήσαμε σε τροποποίηση του κώδικα των 3 συναρτήσεων dgemm, dgem_ta,

dgem_tb με προσθήκη των κατάλληλων εντολών **pragma omp parallel for**, όπως φαίνεται στις παρακάτω γραμμές κώδικα. Η εντολή αυτού του τύπου χρησιμοποείται στα πλαίσια του OpenMP ειδικά για να παραλληλοποιεί for loops.

2.1.1 $C = A \cdot B$

Στον παρακάτω κώδικα έχει προστεθεί η εντολή **pragma omp parallel for private(sum, j, k)** στη δοθείσα συνάρτηση dgemm. Με την εντολή αυτή επιτυγχάνουμε το "μοιρασμα" του εξωτερικού i-loop, που αφορά το iteration πάνω στις M γραμμές του A, στα επιμέρους threads. Το j-loop αφορά το iteration πάνω στις N στήλες του B ενώ το k-loop πάνω στις στήλες του A και ταυτόχρονα στις ισάριθμες γραμμές του B. Το scheduling αφήνεται στο default του λειτουργικού συστήματος, καθώς δεν ορίζουμε explicitly κάποιον scheduler. Κάθε thread λοιπόν αναλαμβάνει την εκτέλεση ενός υποσυνόλου των iterations που αντιστοιχούν στο loop αυτό. Είναι πολύ σημαντικό οι counters των εσωτερικών loops να οριστούν ως private προκειμένου να διασφαλιστεί ότι το iteration πάνω σε αυτούς θα είναι αποκλειστικό για κάθε thread και δε θα αλλάζει η τιμή του counter από άλλο thread. Δηλαδή θέλουμε ένα thread κατα τη διάρκεια ενός i-iteration να αναλαμβάνει των υπολογισμό που αντιστοιχεί σε μία ολόκληρη γραμμή του πίνακα A (i-loop) και σε όλες τις στήλες του πίνακα B (j-loop) συσσωρεύοντας το αποτέλεσμα του αθροίσματος γινομένων της κάθε στήλης στη μεταβλητή sum μέχρι να προσπελάστεί ολόκληρη η i-γράμμη του A και η αντίστοιχη j-στήλη του B κοινού μήκους K (k-loop) ώστε να αποθηκευτεί το αποτέλεσμα στο αντίστοιχο c_{ij}. Άλλο ένα πολύ σημαντικό στοιχείο είναι ότι οι πίνακες, παρότι δισδιάστατοι, στη μνήμη είναι αποθηκευμένοι σειριακά και κατά γραμμή στη μνήμη, γεγονός που σημαίνει ότι πρέπει να φροντίσουμε για τη σωστή προσπέλαση τους. Έχοντας λοιπόν χαρακτηριστικά το i να καθορίζει για ποια γραμμή του A μιλάμε αρκεί να πολλ/ζουμε το δείκτη αυτό με το πλήθος στηλών K του A προκειμένου να προσπερνάμε τις αντίστοιχες θέσεις μνήμης που αντιστοιχούν σε κάθε μία απο τις προηγούμενες γραμμές του. Για αυτό και καταλήγουμε στην εκφραση A[i * K + k] οπού το k loopάρει πάνω στη ι-γραμμή. Κάνοντας το ίδιο για τον πίνακα B πρέπει να σκεφτούμε ότι προσπελαύνουμε τα στοιχεία του κατά στήλη στο εσωτερικό κάθε k-iteration που σημαίνει ότι πρέπει να κάνουμε την αντίστοιχη διαδικασία προσπερνώντας N στοιχεία για κάθε προηγούμενη γραμμή. Έτσι καταλήγουμε στην έκφραση B[k * N + j]. Κάθε φορά που υπολογίζεται ένα εσωτερικό γινόμενο i-γραμμής του A και j-στήλης του B, στην ολοκλήρωση κάθε k-loop, είναι απαραίτητο το αποτέλεσμα να τοποθετηθεί στο αντίστοιχο c_{ij}, πριν ξαναναρχικοποιηθεί στο 0 το sum και πάμε στο επόμενο loop, και αυτό επιτυγχάνεται με χρήση της έκφρασης C[i * N + j] = sum. Η λογική είναι ίδια ως προς το ότι για να προσπελάσουμε το i,j στοιχείο του C πρέπει να προσπερνάμε τα N στοιχεία που είναι αποθηκευμένα για κάθε γραμμή στη μνήμη. Ο j-δείκτης μεταβάλλεται μία φορά στο τέλος κάθε k-loop πηγαίνοντας μας για την ίδια i-γραμμή του A στην επόμενη j-στήλη του B κ.ο.κ. Η διαδικασία αυτή επαναλαμβάνεται για όλες τις στήλες του B και σ αυτό το σημείο τελειώνει το κάθε i-iteration.

```
    ...
    #else
    int i, j, k;
    double sum;
    #pragma omp parallel for private(sum, j, k)
    for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        sum = 0.;
        for (k = 0; k < K; k++)
            sum += A[i * K + k] * B[k * N + j];
        C[i * N + j] = sum;
        }
    }
    #endif
```

2.1.2 $C = A^T \cdot B$

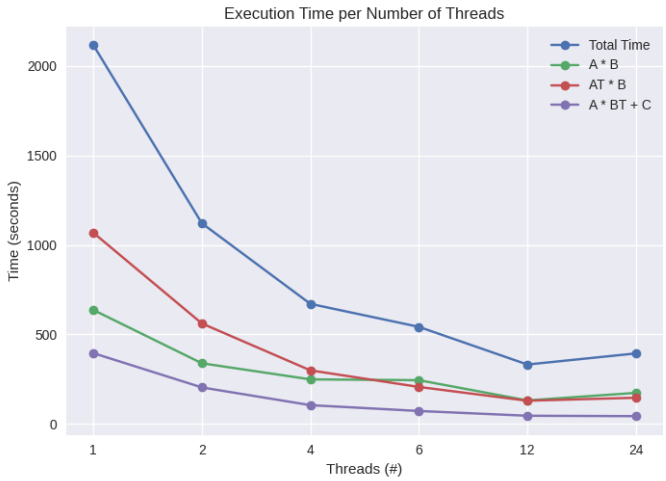
Στον παρακάτω κώδικα έχει γίνει ακριβώς η ίδια προσθήκη παραλληλοποίησης **pragma omp parallel for private(sum, j, k)** στη δοθείσα συνάρτηση dgemm_ta. Η λογική προφανώς είναι ακριβώς η ίδια αλλά αυτή τη φορά το μόνο που αλλάζει είναι ότι πρέπει το k πρέπει να κάνει iteration πάνω στην i-στήλη του και αυτό απεικονίζεται μόνο σε τροποποίηση της έκφρασης δεικτών του A σε A[k * M + i]. Σε αυτή την περίπτωση, κάθε φορά που αυξάνει το k κατά 1 πρέπει να γίνεται προσπέρασμα στις M στήλες του για να κρατείται το επόμενο στοιχείο της στήλης ενώ το i υποδεικνύει ότι κρατείται η i-οστή στήλη του κάθε φορά αναλόγως. Τα υπόλοιπα εκτυλίσσονται παρομοίως.

```
...
#else
int i, j, k;
double sum;
#pragma omp parallel for private(sum, j, k)
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        sum = 0.;
        for (k = 0; k < K; k++)
            sum += A[k * M + i] * B[k * N + j];
        C[i * N + j] = sum;
        }
      }
#endif
```

2.1.3 $D = A \cdot B^T + C$

Σε αυτή την περίπτωση η αναστροφή θα πρέπει να εφαρμοστεί στον B πίνακα ώστε να μπορέσει να γίνει η πράξη του πολλαπλασιασμού. Αντίστοιχα με την προηγούμενη ενότητα, η αναστροφή επιτυγχάνεται στην ίδια λογική με αλλαγή της έκφρασης δεικτών σε B[j * K + k]. Μία ακόμα προσθήκη είναι ότι στο τελικό αποτέλεσμα προστίθεται και η αντίστοιχη τιμή του πίνακα C. Η παραλληλοποίηση, όπως και στις δύο προηγούμενες περιπτώσεις επιτυγχάνεται με τη χρήση του **pragma omp parallel for private(sum, j, k)** στη δοθείσα συνάρτηση dgemm_tb.

```
...
#else
int i, j, k;
double sum;
#pragma omp parallel for private(sum, j, k)
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        sum = 0.;
        for (k = 0; k < K; k++)
            sum += A[i * K + k] * B[j * K + k];
        D[i * N + j] = sum + C[i * N + j];
    }
  }
#endif
```



Σχήμα 1: Χρόνος Εκτέλεσης ανά αριθμό νημάτων

Στην παραπάνω εικόνα μπορούμε να παρατηρήσουμε τον συνολικό χρόνο εκτέλεσης που απαιτείται για την κάθε πράξη ανά αριθμό χρησιμοποιούμενων νημάτων. Είναι ξεκάθαρο πως εξετάζοντας τον χρόνο εκτέλεσης ξεχωριστά για κάθε πράξη αλλά και για όλες μαζί συνολικά όσο αυξάνεται ο αριθμός των νημάτων ο χρόνος εκτέλεσης μειώνεται σημαντικά. Κατά τα λοιπά βλέπουμε πως υπάρχει μία τάση ομαλοποίησης του χρόνου απόκρισης από τα 4 νήματα και πάνω.

2.2 OpenBLAS

Η ανοιχτού κώδικα βιβλιοθήκη OpenBLAS παρέχει έτοιμες συναρτήσεις για την παραλληλοποίηση πράξεων γραμμικής άλγεβρας, όπως πρόσθεση και πολλαπλασιασμός διανυσμάτων και πινάκων. Σε αυτήν την ενότητα θα παραλληλοποιήσουμε τις παραπάνω συναρτήσεις μέσω έτοιμων και βέλτιστων υλοποιήσεων που παρέχονται από την OpenBLAS. Τα παρακάτω αποτελεί παράδειγμα για πίνακες $\mathbf{A_{M\times K}}, \mathbf{B_{K\times N}}$

Η συνάρτηση που θα μας βοηθήσει να να εφαρμόσουμε τον αλγόριθμο GeMM είναι η εξής:

```
    cblas_dgemm(CBLAS_LAYOUT layout,
                CBLAS_TRANSPOSE TransA,
                CBLAS_TRANSPOSE TransB,
                const int M,
                const int N,
                const int K,
                const double alpha,
                const double * A,
                const int lda,
                const double * B,
                const int ldb,
                const double beta,
                double * C,
                const int ldc
                )
```

Συγκεκριμένα υλοποιεί την πράξη alpha*A*B + beta*C αποθηκεύοντας το αποτέλεσμα στον πίνακα C. Στον πίνακα 1 βλέπουμε το νόημα των παραμέτρων που δέχεται η συνάρτηση.

2.2.1 $C = A \cdot B$

Για την υλοποίηση του πολλαπλασιασμού

$$\mathbf{C_{M\times N}} = \mathbf{A_{M\times K}} \cdot \mathbf{B_{K\times N}} \tag{4}$$

καλούμε τη συνάρτηση:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            M, N, K, 1, A, K, B, N, 0, C, N);
```

Όπου περνάμε με σειρά, τις γραμμές του πίνακα A, τις στήλες του πίνακα B και την κοινή στήλη K. Έπειτα, βάζουμε την τιμή 1 γιατί θέλουμε το $\cdot B$. Ακολουθεί ο πίνακας A με το το leading dimension του που είναι η στήλη K(εφόσον οι πίνακες μας είναι RowMajor), ο πίνακας B με το leading dimension του επίσης που είναι το N. Επιπλέον, βάζουμε beta= 0 καθώς δε θέλουμε να κάνουμε scale τον πίνακα C και μας ενδιαφέρει να του δώσουμε απλώς το αποτέλεσμα A*B. Τελευταία παράμετρος είναι το leading dimension του C που είναι το N.

2.2.2 $C = A^T \cdot B$

Η επόμενη πράξη που θα περιγράψουμε με τη χρήση OpenBlas είναι ο πολλαπλασιασμός

$$\mathbf{C_{M\times N}} = \mathbf{A^{\top}_{M\times K}} \cdot \mathbf{B_{K\times N}} \tag{5}$$

Για αυτήν την περίπτωση, όπου ο A είναι transposed θα καλέσουμε την παρακάτω συνάρτηση.

```
cblas_dgemm(CblasRowMajor, CblasTrans,
            CblasNoTrans, M, N, K, 1, A, M, B,
            N, 0, C, N);
```

Παρατηρούμε πως άλλαξε η 2η παράμετρος σε σχέση με την 1η περίπτωση και έγινε CblasTrans για να δηλώσει ότι θέλουμε τον transposed A. Στη συνέχεια οι άλλες παράμετροι ακολουθούν την ίδια λογική με την πρώτη περίπτωση.

2.2.3 $D = A \cdot B^T + C$

Η τελευταία πράξη που θέλουμε να υλοποιήσουμε είναι η

$$\mathbf{D_{M\times N}} = \mathbf{A_{M\times K}} \cdot \mathbf{B^{\top}_{K\times N}} + \mathbf{C_{M\times N}} \tag{6}$$

Αυτή πραγματοποιείται ακολούθως:

```
memcpy (D, C, M * N * sizeof (double));
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, M, N, K, 1.0,
A, K, B, K, 1.0, D,  N);
```

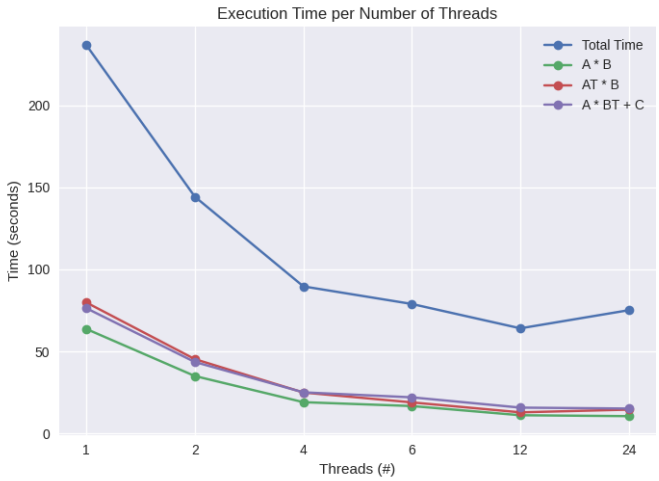
Πρώτη μας κίνηση είναι με την εντολή memcpy να μεταφέρουμε τον πίνακα C, στον D , ο οποίος είναι μεγέθους $M \times N$ με τιμές double. Επομένως, έχουμε ήδη υλοποιήσει το πρώτο μέρος της ζητούμενης πράξεις. Στη συνέχεια με τη συνάρτηση της openblas cblas_dgemm όπου θέτουμε alpha = 1.0 και beta = 1.0 για να υπολογίσουμε την πράξη $D = 1*A\times B^{\top}+1*D$,όπου το D είναι ίσο με το C, εφόσον προηγουμένως το περάσαμε με την εντολή memcpy.

Όρισμα	Περιγραφή
layout	CblasRowMajor / CblasColumnMajor (Δείχνει αν ο πίνακας είναι αποθηκευμένος κατά γραμμή ή κατα στήλες αντίστοιχα.)
TransA	CblasNoTrans / CblasTrans(Δείχνει αν ο πίνακας A είναι transposed ή όχι.)
TransB	CblasNoTrans / CblasTrans (Δείχνει αν ο πίνακας B είναι transposed ή όχι.)
M	γραμμές του A και του C πίνακα
N	στήλες για B και C πίνακα
K	στήλες του A και γραμμές του B(κοινό στοιχείο για να γίνεται ο πολλαπλασιασμός πινάκων)
alpha	Πραγματική τιμή που κάνει scale τον πίνακα A*B
A	πινακας A
lda	Κύρια διάσταση του πίνακα A ή ο αριθμός των στοιχείων μεταξύ διαδοχικών σειρών. Για row major πίνακες είναι ο αριθμός στηλών.
B	πίνακας B
ldb	Κύρια διάσταση του πίνακα B ή ο αριθμός των στοιχείων μεταξύ διαδοχικών σειρών. Για row major πίνακες είναι ο αριθμός στηλών.
beta	Πραγματική τιμή που κάνει scale τον πίνακα C
C	Πίνακας C
ldc	Κύρια διάσταση του πίνακα C ή ο αριθμός των στοιχείων μεταξύ διαδοχικών σειρών. Για row major πίνακες είναι ο αριθμός στηλών.

Πίνακας 1: Περιγραφή συνάρτησης cblas/Dgemm

2.2.4 Αποτελέσματα

Στη συνέχεια ακολουθούν οι επιδόσεις της εκπαίδευσης του νευρωνικού δικτύου για σειριακή εκτέλεση και για παράλληλη εκτέλεση των πολλαπλασιασμών πινάκων OpenMP και OpenCUBLAS. Έχουν χρησιμοποιηθεί όπως θα φαίνεται 1, 2, 4, 6, 12, 24 threads. Σε αυτά τα δύο διαγράμματα αποτυπώνεται ο συνολικός χρόνος εκτέλεσης της εκπαίδευσης και η επιτάχυνση της παράλληλης εκτέλεσης των πολλαπλασιασμών πινάκων.



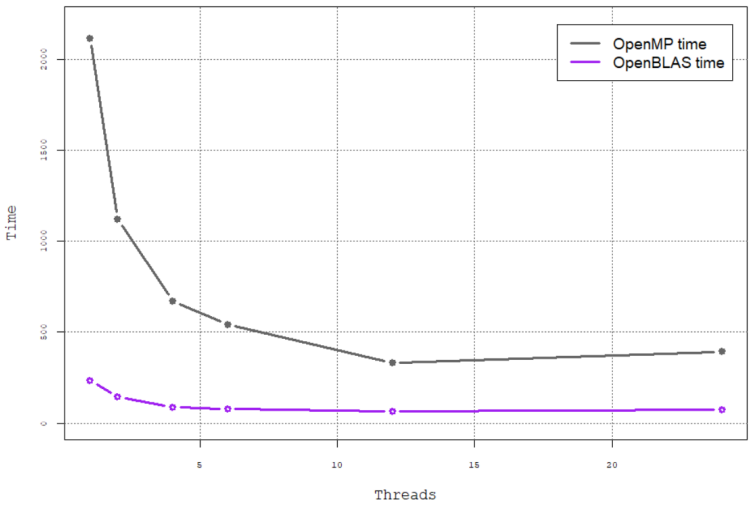
Σχήμα 2: Execution Time

Από την παραπάνω γραφική παράσταση, παρατηρούμε πως οι χρόνοι απόκρισης για την κάθε πράξη μειώνονται σε σημαντικό βαθμό όσο γίνεται χρήση περισσοτέρων νημάτων, ενώ από την χρήση 4 νημάτων και άνω υπάρχει μία "σταθεροποίηση" του χρόνου εκτέλεσης για την κάθε πράξη.

Από τη σύγκριση των 2 μεθόδων OpenMP και OpenBLAS παρατηρούμε πως η υλοποίηση με OpenBLAS είναι πολυ πιο γρήγορη από αυτή του OpenMP τόσο σειριακά όσο και παράλληλα. Αυτό είναι λογικό καθώς το OpenBLAS είναι μια βελτιστοποιημένη μέθοδος του πολλαπλασιασμού πινάκων ανάλογα με τη μορφή που δέχεται ως όρισμα στις συναρτήσεις του.

2.2.5 Κλιμακωσιμότητα

Παρακάτω φαίνεται η επίδοση του training στο νευρωνικό δίκτυο για σειριακή εκτέλεση αλλά και 2, 4, 6, 12 και 24 threads. Στο Σχ. 3 παρουσιάζεται ο συνολικός χρόνος εκτέλεσης ανά thread και το αντίστοιχο speedup ανά thread και για τις 2 μεθοδολογίες.

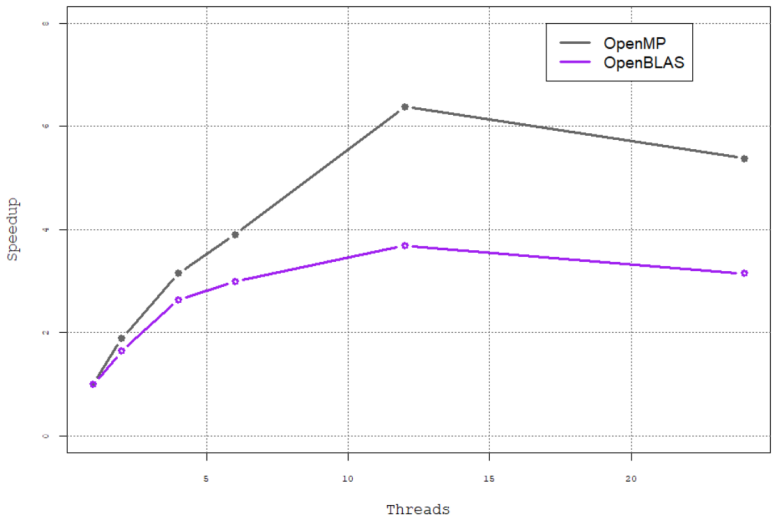


Σχήμα 3: Κλιμάκωση χρόνων εκτέλεσης ως προς τον αριθμό των νημάτων για OpenMP, OpenBLAS

Στην περίπτωση του OpenMP η αύξηση του αριθμού threads οδηγεί αρχικά σε υποδιπλασιασμό του χρόνου από το 1 στα 2 threads. Ο χρόνος εκτέλεσης συνεχίζει να ακολουθεί

φθίνουσα πορεία ωστόσο με μειούμενο ρυθμό όσο αυξάνουμε τον αριθμό των threads εως τα 12. Στη συνέχεια από τα 12 εως τα 24 threads παρατηρούμε μια μικρή αύξηση στο χρόνο πράγμα που υποδεικνύει. Παρόμοιες συμπεριφορές παρατηρούνται και στο OpenBLAS με ωστόσο σε πολύ χαμηλότερους χρόνους εν γένει. Αυτή η συμπεριφορά είναι αναμενόμενη μετά από ένα σημείο καθώς παρ' ότι ένα πρόβλημα μπορεί να είναι παραλληλοποιήσιμο υπάρχουν και άλλα στοιχεία τα οποία επηρεάζουν τον "επιτρεπόμενο" βαθμό παραλληλοποίησης. Τα στοιχεία αυτά είναι τα εξής:

- **Race conditions**: Οσο ο αριθμός νημάτων αυξάνεται υπερβολικά δημιουργείται συμφόρηση στο memory bus με αποτέλεσμα τα threads να ανταγωνίζονται μεταξύ τους για διαδικασίες όπως είναι το διάβασμα από την κύρια μνήμη.
- **Thread synchronisation overheads**: Ο συγχρονισμός των επιπλέον νημάτων αρχίζει να εισάγει επιπλέον overhead στους υπολογισμούς το οποίο δεν αντισταθμίζεται από την επιπλέον παραλληλοποίηση.



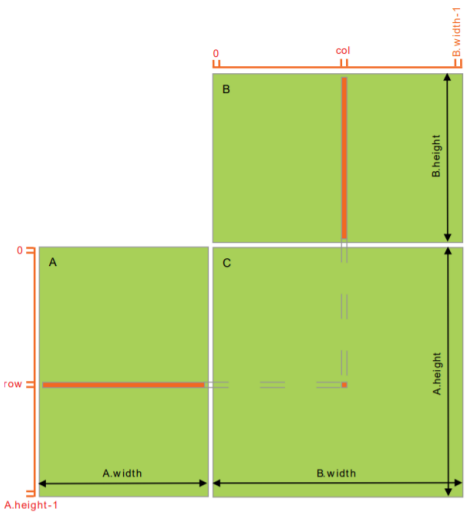
Σχήμα 4: Κλιμάκωση speedup ως προς τον αριθμό των νημάτων για OpenMP, OpenBLAS

Στο συγκεκριμένο διάγραμμα μελετάμε το speedup ή επιτάχυνση της κάθε μεθοδολογίας για σειριακή εκτέλεση και νήματα 2,4,6,12,24. Ως επιτάχυνση ορίζουμε το πηλίκο του χρόνου της σειρακής εκτέλεσης ως προς της παράλληλης. Παρατηρούμε και σε αυτό το διάγραμμα πως μετά από τα 12 thread υπάρχει μείωση του χρόνου και δε βελτιώνονται οι 2 υλοποιήσεις. Ενώ μέχρι τα 12 νήματα παρατηρούμε μεγάλη βελτίωση.

3 Παραλληλοποίηση σε GPU

Στο δεύτερο κομμάτι της παρούσας εργαστηριακής άσκησης θα γίνει χρήση των επιταχυντών (GPUs) για την παραλληλοποίηση των πολλαπλασιασμών. Ως μαθηματική πράξη, ο πολλαπλασιασμός πινάκων είναι εύκολα παραλληλοποιήσιμη και για αυτό το λόγο το μοντέλο **SIMT** (SIngle Instruction Multiple Threads) που ακολουθούν οι GPUs είναι κατάλληλο για την εκτέλεση τέτοιων πράξεων. Πρακτικά, η ύπαρξη πολλών πυρήνων καθιστά δυνατή την παραλληλοποίηση της πράξης αυτής σε μεγαλύτερο βαθμό από την ίδια περίπτωση με χρήση της CPU, αναθέτοντας ένα στοιχείο του τελικού πίνακα, δηλαδή το εσωτερικό γινόμενο μιας γραμμής με μία στήλη, σε κάθε νήμα. Αυτή η δυνατότητα των GPUs τις κάνει τόσο δημοφιλείς στον κλάδων των Νευρωνικών Δικτύων, δεδομένης της ανάγκης να διαχειριστούν τέτοιου είδους πράξης,

Στην συνέχεια, θα αναλυθεί η παραλληλοποίηση του πολλαπλασιασμού πινάκων με GPUs τόσο με την Shared Memory όσο και με την βιβλιοθήκη cuBLAS. Για την υλοποίηση αυτού του ερωτήματος υπήρξε αρχικός κώδικας που μας δόθηκε από το εργαστήριο. Ο κώδικας αυτός αποτελεί μία naive υλοποίηση του πολλαπλασιασμού πινάκων σε GPUs. Σύμφωνα με τις πρακτικές του υπολογισμού της πράξης πολλαπλασιασμού πινάκων κάθε thread στη δοθείσα naive υλοποίηση είναι υπεύθυνο για τον υπολογισμό ενός στοιχείου. Στο Σχ. 5 απεικονίζεται σχηματικά η διαδικασία που ακολουθεί η naive υλοποίηση που μας δόθηκε. Εφόσον έχουμε αποθηκεύσει τον πίνακα στην global μνήμη της GPU, κάθε thread είναι υποχρεωμένο να διαβάζει τα δεδομένα που χρειάζεται (στοιχεία κάθε γραμμής του A και της αντίστοιχης στήλης του B) για την εκάστοτε πράξη από αυτή τη μνήμη με αποτέλεσμα να πραγματοποιούνται πολλές προσβάσεις, οι οποίες χαρακτηρίζονται από χαμηλές ταχύτητες, πράγμα που μειώνει ιδιαίτερα την αποδοτικότητα του αλγορίθμου. Στη βελτιστοποίηση αυτή καλούμαστε να την κάνουμε σε πρώτη φάση με υλοποίηση σχετικού κώδικα καθώς και με χρήση της βιβλιοθήκης cuBLAS η οποία υλοποιεί μια ακόμη πιο προηγμένη παραλλαγή της.

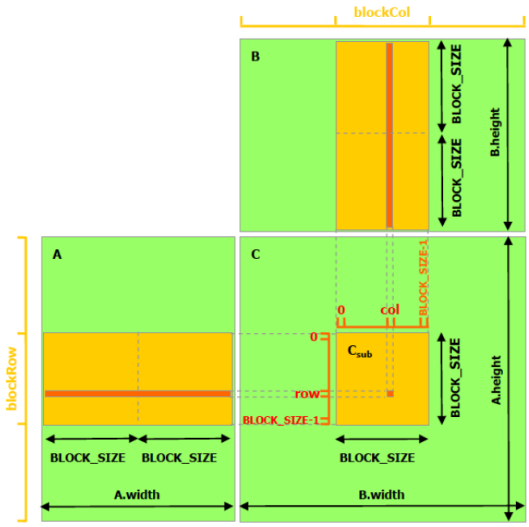


Σχήμα 5: Naive υλοποίηση του πολλαπλασιασμού πινάκων A,B στη GPU. [Πηγή: NVIDIA CUDA C Programming Guide]

3.1 Shared Memory

Εμείς καλούμαστε να βελτιστοποιήσουμε την υλοποίηση των ίδιων πράξεων με τη χρήση της shared memory, η οποία είναι προσβάσιμη από όλα τα threads ενός thread block, όπως φαίνεται στο Σχ. 6 καθώς οι προσβάσεις σε αυτή τη μνήμη χαρακτηρίζονται από ιδιαίτερα χαμηλά latencies. Στις παρακάτω επεξηγήσεις γίνεται η θεώρηση της πράξης τετραγωνικών πινάκων $C = A \cdot B$ Προτιμάται λοιπόν σε αυτή την περίπτωση να επιστρατεύουμε σε μία πρώτη φάση τα threads ενός thread block, που αφορούν τον υπολογισμό ενός υποπίνακα του C μεγέθους BLOCK_DIMY x BLOCK_DIMX, να φορτώσουν στη shared memory τις αντίστοιχες γραμμές του A και στήλες του B. Σε κάθε thread block λοιπόν, κάθε thread φέρνει από την κύρια μνήμη στην κοινή μνήμη της GPU ένα στοιχείο του A και ένα στοιχείο του B που αντιστοιχεί στις συντεταγμένες του εντός του πίνακα C. Στη συνέχεια, εφόσον όλα τα threads του block έχουν μεταφέρει τα στοιχεία στη shared memory τα threads συγχρονίζονται και ξεκινούν τον τελικό υπολογισμό με τα απαιτούμενα στοιχεία να βρίσκονται ήδη στη shared memory και να μη χρειάζεται να γίνεται καμία αναφορά στην κύρια μνήμη. Αυτή η διαδικασία επαναλαμβάνεται για όλους τους υποπίνακες μεγέθους BLOCK_DIMY x BLOCK_DIMX - tiles - submatrices των A, B που αντιστοιχούν στην αντίστοιχο υπολογιζόμενο tile του C. Η διαδικασία αυτή αφενός παραλληλοποιείται περαιτέρω ανάλογα με το

πλήθος των SMs της GPU ωστόσο το τμήμα που απευθύνεται στον υπολογισμό του κάθε tile του C υποχρεωτικά αναλαμβάνεται από τα threads του ίδιου thread block. Αυτό που πρέπει να κρατήσουμε και που σίγουρα δημιουργεί ένα overhead στην υλοποίηση αυτή είναι ότι κάθε thread block έχει την υποχρέωση να συγχρονίζει τα threads του $2 \cdot \frac{N}{BlockDiM}$ φορές κατά προσέγγιση, όπου N η διαστάση των 2 τετραγωνικών στην περίπτωση μας πινάκων A, B. Ο παράγοντας 2 οφείλεται στο ότι απαιτείται ένας συγχρονισμός που σηματοδοτεί ότι τα στοιχεία βρίσκονται στη shared memory και άλλος ένας που σηματοδοτεί ότι το τμήμα του υπολογισμού που αφορά προς συζήτηση tile του C και τα αντίστοιχα tiles των A,B έχει πραγματοποιηθεί. Στη συνέχεια το thread block περνάει στα επόμενα tiles των A,B που αφορούν τις επόμενες επιμέρους αθροίσεις του ίδιου tile του C. Στη συνέχεια παρατίθεται ο κώδικας της σχετικής υλοποίησης.



Σχήμα 6: Πολλαπλασιασμός πινάκων A, B με χρήση της κοινής μνήμης έτσι ώστε να αποθηκεύει υποπίνακες των A,B. [Πηγή: NVIDIA CUDA C Programming Guide]

3.1.1 $C = A \cdot B$

Παρακάτω είναι το kernel που υλοποιεί τον κλασσικό πολλαπλασιασμό πινακων $\times B$. Επίσης, στο τέλος παραθέτουμε και το τρόπο με τον οποίο καλούμε το kernel με τα grid και block sizes. Η μεθοδολογία που ακολουθήθηκε είναι αυτή που περιγράψαμε παραπάνω.

```
__global__ void dgemm_optimized(const double *A, const double *B,
                                double *C,
                                const int M, const int N, const int K) {
    const int BLOCK_SIZE = 16;
    double sum = 0.;

    // Initialize A and B submatrices
    __shared__ double A_sub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_sub[BLOCK_SIZE][BLOCK_SIZE];

    // Calculate thread row and col on C matrix
    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y,
        col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    // Get A and B submatrices values into shared memory
    for (int m = 0; m < K / BLOCK_SIZE; ++m) {
        if (row < M && m * BLOCK_SIZE + threadIdx.x < K)
            A_sub[threadIdx.y][threadIdx.x] = A[(row * K) +
                (m * BLOCK_SIZE) + threadIdx.x];
        if (col < N && m * BLOCK_SIZE+threadIdx.y < K)
            B_sub[threadIdx.y][threadIdx.x] = B[((m * BLOCK_SIZE)
                + threadIdx.y) * N + col];

    // Sync threads to have full submatrices
       __syncthreads();

    // Calculate C submatrix value for current thread
       for (int k = 0; k < BLOCK_SIZE; ++k)
           sum += A_sub[threadIdx.y][k] * B_sub[k][threadIdx.x];

    // Sync threads again
       __syncthreads();
    }

    // Write C matrix value
    if (row < M && col < N)
       C[(row * N) + col] = sum;
.
.
.
#elif defined(_GPU_GEMM_OPT)
  dim3 block(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE,
            (M + BLOCK_SIZE - 1) / BLOCK_SIZE);
  dgemm_optimized<<<grid, block>>>(A, B, C, M, N, K);
  checkCudaErrors(cudaPeekAtLastError());
  checkCudaErrors(cudaDeviceSynchronize());
#endif
}
```

3.1.2 $C = A^T \cdot B$

Σε αυτήν την πράξη κατά αναλογία με τα παραπάνω κάναμε κάποιες αλλαγές στον τρόπο που γεμίζουμε τον A_sub και τη μεταβλητή sum.

```
__global__ void dgemm_ta_optimized(const double *A, const double *B,
                                   double *C,
                                   const int M, const int N, const int K) {

    const int BLOCK_SIZE = 16;
    double sum = 0.;

    // Initialize A and B submatrices
    __shared__ double A_sub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_sub[BLOCK_SIZE][BLOCK_SIZE];

    // Calculate thread row and col on C matrix
    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y,
        col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    // Get A and B submatrices values into shared memory
    for (int m = 0; m < K / BLOCK_SIZE; m++) {

        if (row < M && (((m * BLOCK_SIZE) + threadIdx.x) < K))
            A_sub[threadIdx.y][threadIdx.x] = A[row + ((m * BLOCK_SIZE +
                threadIdx.x) * M)];

        if (col < N && (m * BLOCK_SIZE + threadIdx.y < K))
            B_sub[threadIdx.y][threadIdx.x] = B[(m * BLOCK_SIZE +
                threadIdx.y) * N + col];

    // Sync threads to have full submatrices
        __syncthreads();

    // Calculate C submatrix value for current thread
       for (int k = 0; k < BLOCK_SIZE; k++)
           sum += A_sub[threadIdx.y][k] * B_sub[k][threadIdx.x];

    // Sync threads again
        __syncthreads();
    }

    // Write C matrix value
    if (row < M && col < N)
        C[(row * N) + col] = sum;
}

.
.
.
#elif defined(_GPU_GEMM_OPT)
  dim3 block(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE,
            (M + BLOCK_SIZE - 1) / BLOCK_SIZE);
  dgemm_optimized<<<grid, block>>>(A, B, C, M, N, K);
  checkCudaErrors(cudaPeekAtLastError());
  checkCudaErrors(cudaDeviceSynchronize());
#endif
```

3.1.3 $D = A \cdot B^T + C$

Παρόμοια και σε αυτήν την πράξη αλλάζουμε τον τρόπο με τον οποίο γεμίζουμε τον πίνακα B_sub και το sum, ενώ μετά από το συγχρονισμό των νημάτων μέσω του synchtreads(), περνάμε το τελικό αποτέλεσμα στον πίνακα D.

```
__global__ void dgemm_tb_optimized(const double *A, const double *B, const double *C,
                                   double *D,
                                   const int M, const int N, const int K) {

    const int BLOCK_SIZE = 16;
    double sum = 0.;

    // Initialize A and B submatrices
    __shared__ double A_sub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_sub[BLOCK_SIZE][BLOCK_SIZE];

    // Calculate thread row and col on C matrix
    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y,
        col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    // Get A and B submatrices values into shared memory
    for (int m = 0; m < K / BLOCK_SIZE; m++) {

        if (row < M && m * BLOCK_SIZE + threadIdx.x < K)
            A_sub[threadIdx.y][threadIdx.x] = A[(row * K) +
                (m * BLOCK_SIZE) + threadIdx.x];

        if (col < N && m * BLOCK_SIZE + threadIdx.y < K)
            B_sub[threadIdx.y][threadIdx.x] = B[threadIdx.y +
                (m * BLOCK_SIZE) + (col * K)];

    // Sync threads to have full submatrices
       __syncthreads();

    // Calculate C submatrix value for current thread
       for (int k = 0; k < BLOCK_SIZE; k++)
           sum += A_sub[threadIdx.y][k] * B_sub[k][threadIdx.x];

    // Sync threads again
       __syncthreads();
    }

    // Write C matrix value
    if (row < M && col < N){
       sum += C[row * N + col];
       D[row * N + col] = sum;}
.
.
.
#elif defined(_GPU_GEMM_OPT)
  dim3 block(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE,
            (M + BLOCK_SIZE - 1) / BLOCK_SIZE);
  dgemm_tb_optimized<<<grid, block>>>(A, B, C, D, M, N, K);

  checkCudaErrors(cudaPeekAtLastError());
  checkCudaErrors(cudaDeviceSynchronize());
#endif

}
```

3.2 cuBLAS

Η βιβλιοθήκη cuBLAS είναι μια υλοποίηση της βιβλιοθήκης BLAS (Basic Linear Algebra Subprograms) κατάλληλη για την CUDA. Αποτελείται από παράλληλες συναρτήσεις CUDA που υλοποιούν βασικές πράξεις γραμμικής άλγεβρας όπως ανάγνωση και εγγραφή πινάκων,

πολλαπλασιασμό διανυσμάτων με πίνακα, πολλαπλασιασμό πινάκων κ.ά.. Η βιβλιοθήκη χωρίζει τις πράξεις της σε 3 επίπεδα:

1. Πράξεις με βαθμωτά μεγέθη και διανύσματα,
2. Κατάταξη των πράξεων μεταξύ πινάκων και διανυσμάτων,
3. Πράξεις μεταξύ πινάκων.

Δεδομένων των παραπάνω, υπολογίζουμε πως θα παρουσιάζει καλύτερα αποτελέσματα στην GPU όσον αφορά την παραλληλοποίηση των πράξεων αυτών.

Παρακάτω ακολουθεί η υλοποίηση του νευρωνικού δικτύου με τη χρήση της βιβλιοθήκης cuBLAS. Οι συναρτήσεις της λειτουργούν αναμένοντας τους πίνακες σε column-major order, το οποίο θα το αναλύσουμε αργότερα στα παραδείγματα της άσκησης.

Η συνάρτηση που θα μας βοηθήσει να να εφαρμόσουμε τον αλγόριθμο GeMM στην Cuda είναι η παρακάτω:

```
cublasStatus_t cublasDgemm(cublasHandle_t handle,
    cublasOperation_t transa, cublasOperation_t transb,
    int m, int n, int k,
    const double *alpha,
    const double *A, int lda,
    const double *B, int ldb,
    const double *beta,
    double *C, int ldc)
```

Τα ορίσματα και η λειτουργία της περιγράφονται στον πίνακα 2.

Όρισμα	Περιγραφή
handle	handle για το context της Cuda
transa	Δηλώνει την πράξη op(A),η οποία για CUBLAS_OP_N δίνει op(A)=A αλλιώς αν CUBLAS_OP_T δίνει $op(A) = A^\top$
transb	Δηλώνει την πράξη op(B),η οποία για CUBLAS_OP_N δίνει op(B)=B αλλιώς αν CUBLAS_OP_T δίνει $op() = ^\top$
m	γραμμές του op(A) και του C πίνακα
n	στήλες για op(B) και C πίνακα
k	στήλες του op(A) και γραμμές του op(B)(κοινό στοιχείο για να γίνεται ο πολλαπλασιασμός πινάκων)
alpha	Πραγματική τιμή που κάνει scale τον πίνακα A*B
A	πινακας A
lda	Κύρια διάσταση του πίνακα A ή ο αριθμός των στοιχείων μεταξύ διαδοχικών σειρών. Για row major πίνακες είναι ο αριθμός στηλών.
B	πίνακας B
ldb	Κύρια διάσταση του πίνακα B ή ο αριθμός των στοιχείων μεταξύ διαδοχικών σειρών. Για row major πίνακες είναι ο αριθμός στηλών.
beta	Πραγματική τιμή που κάνει scale τον πίνακα C
C	Πίνακας C
ldc	Κύρια διάσταση του πίνακα C ή ο αριθμός των στοιχείων μεταξύ διαδοχικών σειρών. Για row major πίνακες είναι ο αριθμός στηλών.

Πίνακας 2: Περιγραφή συνάρτησης cublasDgemm

Όπως ήδη αναφέρθηκε η cublas λειτουργεί με τη διαφορά από το openblas ότι δέχεται τους πίνακες σε column-major order. Επομένως, χρησιμοποιήσαμε τις οδηγίες που μας δίνονται στη δομή του πρόγραμματος για να εκτελέσουμε τους παρακάτω υπολογισμούς.

3.2.1 $C = A \cdot B$

Για τον πολλαπλασιασμό $A \times B$ θα υπολογίσουμε το $(A^\top \times B^\top)^\top$. Αυτό μας συμφέρει καθώς θα περάσουμε τους πίνακες σε transposed μορφή όπως δηλαδή τους περιμένει η cublas, ενώ το αποτέλεσμα που θα λάβουμε θα είναι και αυτό σε μορφή column-major με αποτέλεσμα να μας γυρίσει τον πίνακα $A \times B = C$. Ο συγκεκριμένος πολλαπλασιασμός γίνεται με την εντολή:

```
double alpha = 1;
double beta = 0;
cublasDgemm(cublas_handle, CUBLAS_OP_N,
            CUBLAS_OP_N, N, M, K, &alpha, B,
            N, A, K, &beta, C, N);
```

3.2.2 $C = A^T \cdot B$

Στη συγκεκριμένη υλοποίηση μας βολεύει να κάνουμε τον εξής μετασχηματισμό:
$A^T \times B = A^T \times (B^T)^T = A^{'} \times (B')^T = (B' \times (A')^T)^T$

```
double alpha = 1;
double beta = 0;
cublasDgemm(cublas_handle, CUBLAS_OP_N,
            CUBLAS_OP_T, N, M, K, &alpha,
            B, N, A, M, &beta, C, N);
```

3.2.3 $D = A \cdot B^T + C$

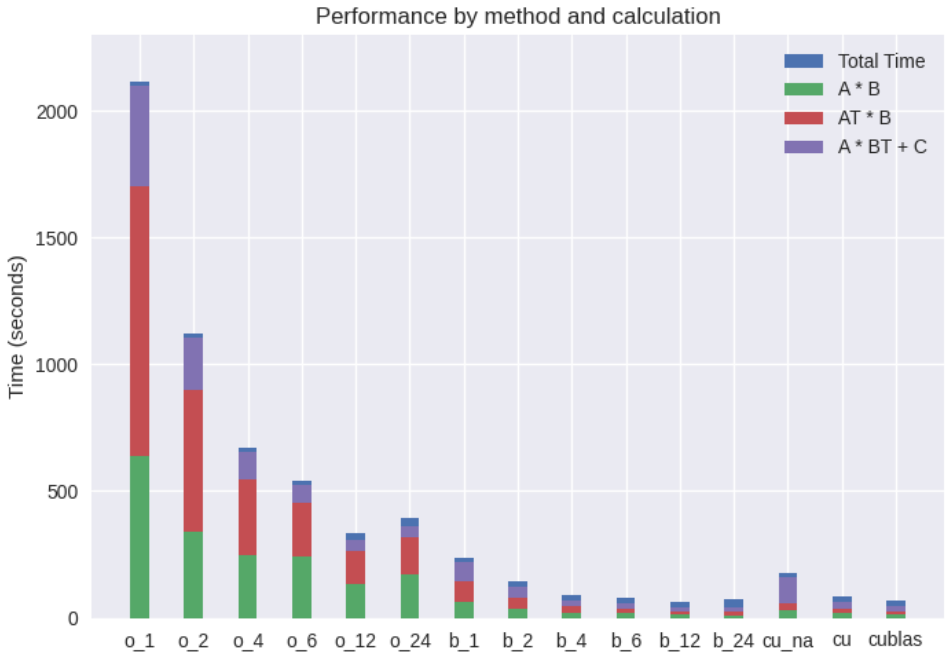
Για την τελική περίπτωση θα χρειαστούμε τη βοήθεια δύο συναρτήσεων. Η μια είναι η γνωστή cublasDgemm και η άλλη είναι η cublasDgeamm, που υλοποιεί την πράξη $C = \alpha \times A + \beta \times B$. Η δεύτερη χρησιμοποιείται ουσιαστικά για να προσθέσουμε τον πίνακα C στο γινόμενο $\times B^T$, που είναι αποθηκευμένο στον πίνακα D από την εντολή cublasDgemm. Ο μετασχηματισμός που χρησιμοποιήσαμε σε αυτήν την περίπτωση είναι: $C = A \times B^T = (A^T)^T \times B^T = (A')^T \times B' = ((B')^T \times A')^T$ για να μας διευκολύνει λειτουργικά με τη συνάρτηση cublasDgemm και στη συνέχεια κάναμε $D = C + D$ με την cublasDgeam.

```
double alpha = 1;
double beta = 0;
cublasDgemm(cublas_handle, CUBLAS_OP_T, CUBLAS_OP_N,
            N, M, K, &alpha, B, K, A, K, &beta, D, N);
cublasDgeam(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N,
            N, M, &alpha, C, N, &alpha, D, N, D, N);
```

3.3 Αποτελέσματα - Σύγκριση GPU-CPU

Παρακάτω φαίνονται οι γραφικές παραστάσεις για κάθε μέθοδο υλοποίησης είτε σε GPU είτε σε CPU και αναγράφεται ο συνολικός χρόνος εκτέλεσης αλλά και ο χρόνος εκτέλεσης των επιμέρους λειτουργιών. Με το αρχικό "o" συμβολίζουμε τις υλοποιήσεις openmp και διπλά βάζουμε τον αριθμό των threads που χρησιμοποίησε. Το "o_1" συμβολίζει το σειριακό χρόνο με ένα νήμα. Στη συνέχεια με "b" συμβολίζουμε τις εκτελέσεις με openBLAS και τέλος

τα labels ”cu_na” , ”cu” και ”cublas” αναφέρονται αντίστοιχα στις υλοποιήσεις cuda χωρίς shared memory, cuda με shared memory και τέλος με την έτοιμη βιβλιοθήκη CuBLAS.



Σχήμα 7: Σύγκριση όλων των μεθόδων CPU και GPU

Από το Σχήμα 5, είναι εμφανές πόσο σημαντική είναι η παραλληλοποίηση των πολλαπλασιασμών πινάκων. Παρατηρούμε, πως για τη CPU μέχρι τα 12 νήματα υπάρχει μεγάλη βελτίωση, ενώ στα 24 ανεβαίνει ο χρόνος. Γεγονός, που σημαίνει πως για τόσα πολλά threads, αυτά ανταγωνίζονται για πόρους μεταξύ τους και έτσι υπάρχει καθυστέρηση. Όσο αφορά τη σύγκριση μεταξύ CPU και GPU βλέπουμε πως γενικά η GPU πετυχαίνει πολύ καλύτερα αποτελέσματα, με καλύτερη υλοποίηση αυτή του Cublas. Ωστόσο το Cublas με το openBLAS και χρήση 12 νημάτων είναι τα καλύτερα σε απόδοση και πολύ κοντά μεταξύ τους. Αυτο μπορεί να αποδωθεί στη μεταφορά δεδομένων από τη CPU στη GPU. Εντυπωσιακό, είναι κατά πόσο μειώνεται ο χρόνος εκτέλεσης όταν χρησιμοποιούμε shared memory αντί για naive υλοποίηση, καθώς υπάρχει μείωση παραπάνω από το μισό. Πολύ σημαντικό είναι, τέλος να παρατηρήσουμε ότι μια διαδικασία training ενος νευρωνικού δικτύου αποτελείται στο μεγαλύτερο με διαφορά ποσοστό της (>99%) από της πράξεις πολλαπλασιασμού πινάκων, γεγονός που δικαιολογεί και το ερευνητικό ενδιαφέρον της επιτάχυνσης των πράξεων αυτών.