Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Δ.Π.Μ.Σ. Επιστήμης Δεδομένων
και Μηχανικής Μάθησης

Παράλληλες Αρχιτεκτονικές Υπολογισμού για Μηχανική Μάθηση
Επιτάχυνση εκπαίδευσης νευρωνικού δικτύου σε αρχιτεκτονικές κοινής μνήμης με OpenMP και CUDA

July 28, 2021

1. Εισαγωγή

Στα πλαίσια αυτής της εργασίας θα καταπιαστούμε με την παραλληλοποίηση του αλγορίθμου πολλαπλασιασμού πινάκων (GEneral Matrix Multiply, GEMM), ο οποίος καταναλώνει μεγάλο ποσοστό του χρόνου εκτέλεσης της εκπαίδευσης νευρωνικών δικτύων. Ξεκινώντας από μία απλοϊκή αρχική υλοποίηση, στο τέλος της εργασίας θα έχουμε πετύχει μία αρκετά αποδοτική υλοποίηση του αλγορίθμου GEMM για CPUs και GPUs, η οποία θα οδηγήσει σε επιτάχυνση της εκπαίδευσης του νευρωνικού δικτύου.
Όσον αφορά την παραλληλοποίηση σε CPU, αρχικά θα τρέξουμε τον κώδικα σειριακά, ώστε να έχουμε μια Baseline της επιτάχυνσης που θα επιτύχουμε. Στη συνέχεια, θα παραλληλοποιήσουμε τα αντίστοιχα `for` loops που χρησιμοποιούνται για την υλοποίηση του πολλαπλασιασμού πινάκων με την χρήση της βιβλιοθήκης **openMP**. Επιπλέον, θα υλοποιήσουμε τις ίδιες πράξεις με τη βοήθεια της βιβλιοθήκης **openBLAS**, και συγκεκριμένα της συνάρτησης `cblas_dgemm()`.
Όσον αφορά την παραλληλοποίηση σε GPU, στην αρχή ομοίως με την CPU θα τρέξουμε τις πράξεις με έναν αφελή τρόπο, στην πορεία θα βελτιστοποιήσουμε και τις τρεις παραλλαγές πολλαπλασιασμού με χρήση κοινής μνήμης (shared memory), και επιπλέον θα τις υλοποιήσουμε με τη χρήση της βιβλιοθήκης **cuBLAS**.
Όλες οι υλοποιήσεις της CPU έτρεξαν στα μηχανήματα της ουράς termis του εργαστηρίου.

2. Παραλληλοποίηση σε CPU

2.1 Αφελής υλοποίηση

Στην αρχή, παραθέτουμε την απλή υλοποίηση, η οποία υπάρχει ήδη στο αρχείο linalg.c, για τις τρείς παραλλαγές πολλαπλασιασμού πινάκων που ζητούνται, δηλαδή: $C = A \cdot B$, $C = A^\top \cdot B$ και $D = A \cdot B^\top + C$.

```
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;
    }
  }
```

```
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;
    }
  }
```

```
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;
    }
  }
```

2.2 Παραλληλοποίηση με openMP

Για να παραλληλοποιήσουμε τις τρεις συναρτήσεις πολλαπλασιασμού πινάκων, προσθέσαμε τη δομή του openMP πριν το for loop.

```
#pragma omp parallel for
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;
    }
  }
```

```
#pragma omp parallel for
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;
    }
  }
```

```
#pragma omp parallel for
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;
    }
  }
```

Στην πορεία, σκεφτήκαμε και μια εναλλακτική υλοποίηση, η οποία αντιστρέφει τα for loops από ijk όπως ήταν αρχικά, σε ikj. Αυτή η υλοποίηση όπως αναφέρεται και σε αυτό το link είναι πολύ ταχύτερη λόγω πιο contiguous κατανομή της μνήμης, εξήγηση σε αυτό το link.

```
// C = A x B
#pragma omp parallel for
for(i=0; i < M; i++){
    for(k=0; k < K; k++){
      for(j=0; j < N; j++){
        C[i*N+j] += A[i*K+k]*B[k*N+j];
   }
  }
 }
```

```
// C = A' x B
#pragma omp parallel for
  for(i=0; i < M; i++){
    for(k=0; k < K; k++){
      for(j=0; j < N; j++){
        C[i*N+j] += A[i*K+k]*B[k*N+j];
   }
  }
 }
```

```
// D = A x B' + C
#pragma omp parallel for
for(i = 0; i < M; i++){
 for(k = 0; k < K; k++){
  for(j = 0; j < N; j++){
   D[i * N + j] += A[i * K + k] * B[j * K + k];
  }
 }
}
#pragma omp parallel for
for(i = 0; i < M; i++){
 for(j = 0; j < N; j++){
  D[i * N + j] += C[i * N + j];
 }
}
```

Επομένως, έχοντας αυτές τις δυο υλοποιήσεις πρέπει να αναλύσουμε ποια απ'τις δύο θα κρατήσουμε για κάθε παραλλαγή. Αφού εκτελέσαμε και τις δυο περιπτώσεις, για αριθμό threads: 1,2,4,6,12 και 24, στο διάγραμμα 1 φαίνονται τα αποτελέσματα για τους χρόνους.

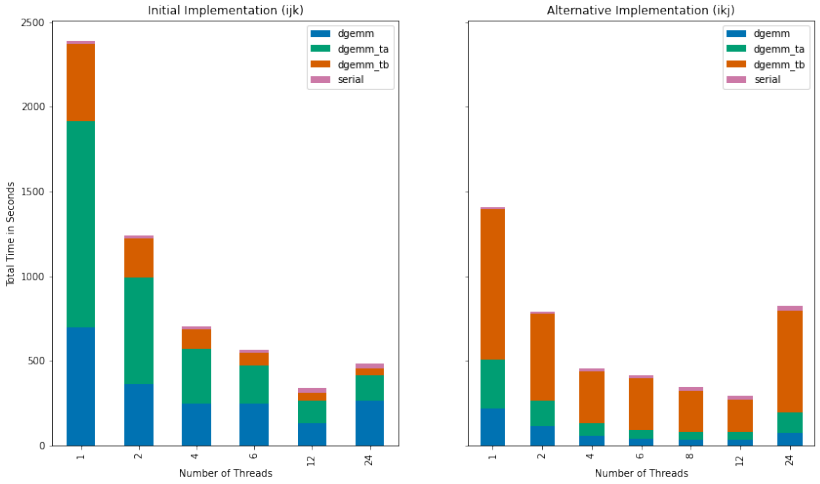

Fig. 1: Σύγκριση δυο υλοποιήσεων - όλα τα threads

Παρατηρούμε ότι η εναλλακτική υλοποίηση, δηλαδή η σειρά των for loops να είναι ikj αντί για ijk της αρχικής, είναι πιο αποτελεσματική σε όλες τις περιπτώσεις threads εκτός από τα 24. Παρόλα αυτά, και οι δυο υλοποιήσεις πετυχαίνουν ελάχιστο χρόνο στα 12 threads. Άλλη μια σημαντική διαφορά που παρατηρούμε στην εναλλακτική υλοποίηση, είναι ότι καθώς στον πολλαπλασιασμό $D = A \cdot B^{\top} + C$ προσθέσαμε και δεύτερο loop, αυτό καθιστά την υλοποίηση μη αποδοτική για αυτή την παραλλαγή. Επομένως, για την τελική μας υλοποίηση παραλληλοποίησης μέσω του OpenMP κρατήσαμε την εναλλακτική υλοποίηση (ikj) για τις πρώτες 2 παραλλαγές του πολλαπλασιασμού, ενώ για την τελευταία παραλλαγή κρατήσαμε την αρχική υλοποίηση. Στο διάγραμμα 2 φαίνεται η σύγκριση των δύο εναλλακτικών υλοποιήσεων.

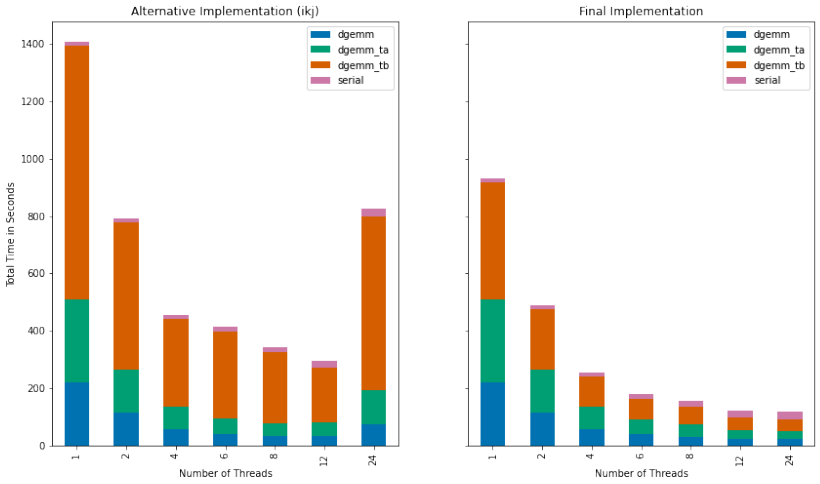

Fig. 2: Σύγκριση δυο υλοποιήσεων - όλα τα threads

Κάτι που μπορεί να μην φαίνεται καθαρά στο διάγραμμα είναι ότι στην τελική υλοποίηση, τα 24 threads είναι λίγο ταχύτερα από ότι τα 12 threads, μόλις κατά 3 δευτερόλεπτα (122 sec τα 12 threads, 118 sec τα 24). Επομένως η τελική υλοποίηση βοήθησε στο να επιτύχουμε λίγο καλύτερο scalability, εφόσον σε όλες τις υπόλοιπες υλοποιήσεις τα 24 threads έπαιρναν περισσότερο χρόνο από ότι τα 12.

2.3 Παραλληλοποίηση με openBLAS

Ο πιο αποδοτικός τρόπος για να εκμεταλλευτούμε την μέγιστη δυνατή υπολογιστική ισχύ μιας υπολογιστικής μονάδας (CPU) ειναι να χρησιμοποιήσουμε βιβλιοθήκες που έχουν γραφτεί ειδικά για μία συγκεκριμένη αρχιτεκτονική σε χαμηλότερο προγραμματιστικό επίπεδο από τη γλώσσα C, δηλαδή στη γλώσσα assembly. Με αυτό τον τρόπο μπορεί μεν η υλοποίηση μας να περιορίζεται σε μία συγκεκριμένη αρχιτεκτονική CPU (π.χ. Intel x86), όμως έχουμε τη δυνατότητα να χρησιμοποιήσουμε συγκεκριμένες δυνατότητες που μας προσφέρονται από την εκάστοτε πλατφόρμα όπως για παράδειγμα την τεχνολογία AVX (Advanced Vector Extensions), η οποία επιταχύνει σημαντικά τους υπολογισμούς.

Στην βιβλιοθήκη OpenBLAS λοιπόν κάθε αλγόριθμος βασικών πράξεων γραμμικής άλγεβρας ξαναγράφεται σε γλώσσα assembly εκμεταλλευόμενος τις ειδικές δυνατότητες της CPU στην οποία εκτελείται, επομένως περιμένουμε να δούμε σημαντικά καλύτερη ταχύτητα εκτέλεσης των πράξεων γραμμικής άλγεβρας με τη χρήση της βιβλιοθήκης OpenBLAS, σε σχέση με την αντίστοιχη υλοποίηση σε γλώσσα C και την παραλληλοποίηση με τη βοήθεια του openmp.

Όσον αφορά την παραλληλοποίηση με τη χρήση της βιβλιοθήκης openBLAS, η δουλειά μας ήταν αρκετά απλή αφού χρειάστηκε απλά να διαβάσουμε το documentation της εντολής `cblas_dgemm()` και να κάνουμε την κατάλληλη παραλλαλαγή πολλαπλασιασμού που ζητείται. Έτσι έχουμε:

- για τον $C = A \cdot B$:

```
float alpha = 1.0;
float beta = 0.0;
cblas_dgemm(CblasRowMajor,CblasNoTrans, CblasNoTrans,
M,N,K,alpha, A, K, B, N, beta, C, N);
```

- Για τον $C = A^\top \cdot B$:

```
float alpha = 1.0;
float beta = 0.0;
cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans,
M, N, K, alpha, A, M, B, N, beta, C, N);
```

- Και τέλος για τον $D = A \cdot B^\top + C$:

```
float alpha = 1.0;
float beta = 1.0;
int i,j,k;
cblas_dgemm(CblasRowMajor,CblasNoTrans, CblasTrans,
M,N,K, alpha, A,K,B,K, beta,D , N);
for (i=0; i < M;i++){
  for(j=0; j < N;j++){
D[i*N + j] += C[i*N + j]; }}
```

3. Παραλληλοποίηση σε GPU

Καθώς ο κώδικας για τις υλοποιήσεις των πολλαπλασιασμών με χρήση shared memory είναι αρκετά μεγάλος, δεν τον παραθέτουμε εφόσον υπάρχουν όλα στον φάκελο μας στο scirouter: `/home/parml/parml02/ex1_omp_cuda/src/linalg.cu`.
Όλα τα runs πραγματοποιήθηκαν στο μηχάνημα gold2 του εργαστηρίου. Στο διάγραμμα 3 παραθέτουμε τους χρόνους τους οποίους πήραμε για τις τρεις διαφορετικές υλοποιήσεις, δηλαδή την αφελή υλοποίηση, την υλοποίηση με χρήση κοινής μνήμης καθώς και με την χρήση της βιβλιοθήκης cuBLAS.

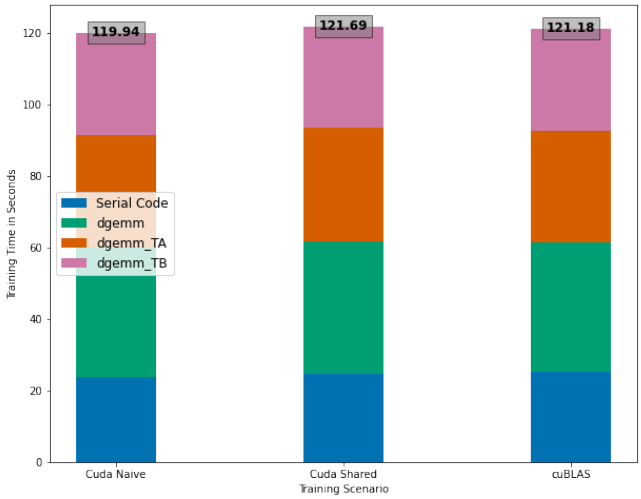

Fig. 3: Χρόνος Εκτέλεσης για τρεις υλοποιήσεις σε CUDA.

Παρατηρούμε ότι οι χρόνοι που πήραμε δεν είναι οι αναμενόμενοι καθώς θα περιμέναμε σίγουρα η υλοποίηση με χρήση κοινής μνήμης αλλά και η υλοποίηση με τις συναρτήσεις της βιβλιοθήκης cuBLAS να είναι πολύ πιο γρήγορες. Επομένως, συμπεραίνουμε ότι για να πετύχουμε καλύτερη επιτάχυνση στην GPU δεν είναι τόσο απλό, και πιθανώς να χρειάζονται και διαφορετικά (μεγαλύτερα) problem sizes για να φανεί αυτό.

4. Πειράματα και Μετρήσεις επιδόσεων

4.1 Κλιμακωσιμότητα σε CPU

Στο διάγραμμα 4 παρουσιάζεται ένα διάγραμμα χρόνου εκτέλεσης καθώς μεταβάλλεται ο αριθμός των νημάτων, για openMP καθώς και για openBLAS. Παρατηρούμε πως το openBLAS πετυχαίνει τον ελάχιστο χρόνο στα 12 threads, ενώ στα 24 threads αυξάνεται λίγο ο χρόνος.
Μεταβαίνοντας από 12 σε 24 πυρήνες φαίνεται ότι η επιτάχυνση δεν αυξάνεται καθόλου, ενώ στις περιπτώσεις των dgemm_ta και dgemm_tb μειώνεται. Αυτό συμβαίνει επειδή οι 12 επιπλέον πυρήνες παρέχονται μέσω της τεχνολογίας hyperthreading. Στην αρχιτεκτονική αυτή κάθε επεξεργαστικός πυρήνας, γίνεται αντιληπτός από το λειτουργικό σύστημα ως δυο λογικοί πυρήνες (συγκεκριμένα ένας πραγματικός επεξεργαστικός πυρήνας και ένας επιπλέον ”εικονικός”), και διαμοιράζει το φόρτο ανάμεσά τους όταν παρίσταται η ανάγκη. Όμως στην πραγματικότητα μόνο ένας λογικός/εικονικός πυρήνας ανά πραγματικό επεξεργαστικό πυρήνα μπορεί να εκτελεί εντολές κάθε χρονική στιγμή. Όπως είναι αναμενόμενο λοιπόν δε μπορούμε να παραλληλοποιήσουμε επιπλέον το πρόγραμμά μας με τη χρήση των hyperthreads. Ταυτόχρονα επειδή από τη χρήση των hyperthreads προστίθεται επιπλέον περιπλοκότητα (overhead), ενώ ταυτόχρονα αυξάνει και ο ανταγωνισμός προς την κρυφή (cache) μνήμη του εκάστοτε πυρήνα παρατηρούμε μία υποβάθμιση της επιτάχυνσης με αποτέλεσμα την πιο αργή εκτέλεση του κώδικα.

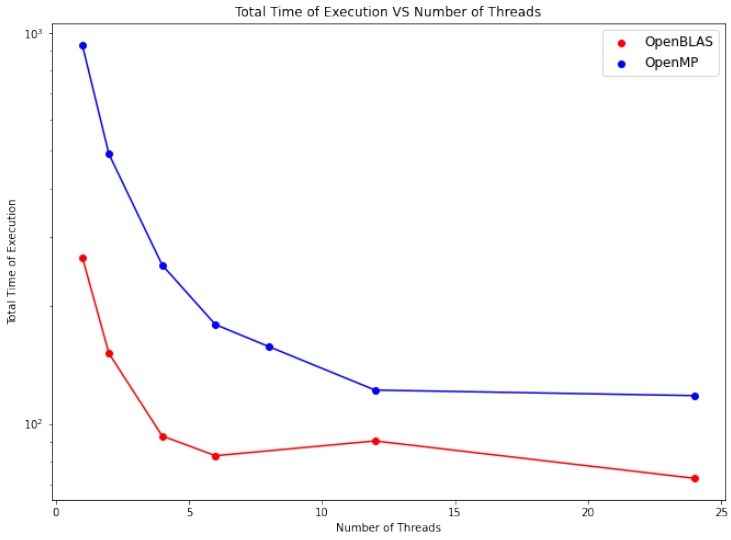

Fig. 4: Συνολικός Χρόνος vs Αριθμός Νημάτων (ημιλογαριθμική κλίμακα).

Στο διάγραμμα 5 παρουσιάζεται ένα διάγραμμα επιτάχυνσης (speedup) καθώς μεταβάλλεται ο αριθμός των νημάτων, για openMP και για openBLAS.

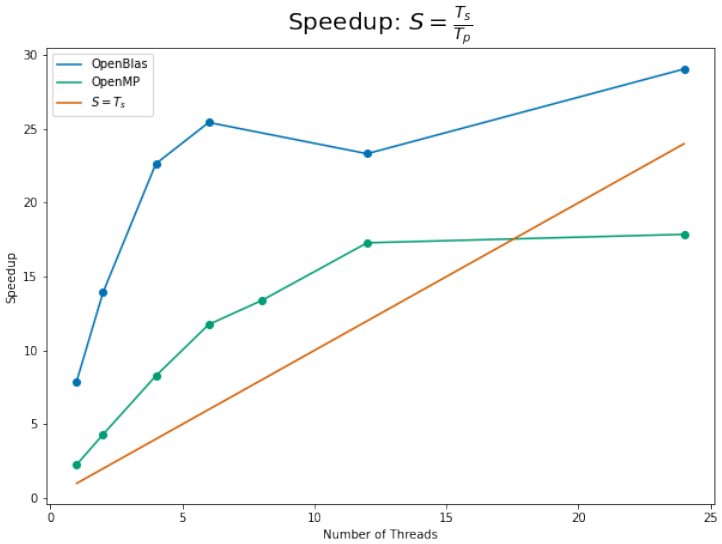

Fig. 5: Επιτάχυνση vs Αριθμός Νημάτων.

Εδώ είναι πολύ σημαντικό να εξηγήσουμε γιατί και οι δύο καμπύλες επιτάχυνσης είναι superlinear. Ως σειριακό χρόνο στον υπολογισμό της επιτάχυνσης, θέσαμε τον χρόνο που εκτελέστηκε με την αφελή υλοποίηση χωρίς καμία παραλληλοποίηση.

- Όσον αφορά το openBLAS, είναι αναμενόμενο να είναι superlinear καθώς αυτή η βιβλιοθήκη πραγματοποιεί βελτιστοποιημένη υλοποίηση όπως αναφέρθηκε παραπάνω. Επομένως, η επιτάχυνση φαίνεται superlinear καθώς προκύπτει από τον συνδυασμό της επίδρασης της παραλληλοποίησης, του cache management και του tiling. Προκειμένου να καταστεί ξεκάθαρη η επιτάχυνση εξαιτίας της παραλληλοποίησης θεωρούμε ως σειριακό χρόνο αυτόν που έχουμε για 1 μόνο thread, όπως φαίνεται και στο διάγραμμα 6.
- Όσον αφορά το OpenMP, όπως εξηγήσαμε και στην ενότητα 2.2, έχει γίνει εναλλαγή των for loops από ijk σε ikj, άρα πρακτικά εξετάζουμε διαφορετική υλοποίηση, γι'αυτό η σύγκριση δεν γίνεται σε ίδια βάση. Για να υπολογίσουμε την πραγματική του επιτάχυνση, θα έπρεπε να τρέξουμε την ikj υλοποίηση σειριακά και με βάση αυτόν το χρόνο να υπολογίσουμε το speedup. Στο διάγραμμα 6, φαίνεται το speedup της αρχικής αφελούς ijk υλοποίησης, όπου έχει παραλληλοποιηθεί με openMP. Απο αυτό το διάγραμμα παρατηρούμε πως σε περίπτωση που είχαμε κάνει αυτήν την υλοποίηση θα είχαμε sublinear επιτάχυνση, όπως είναι αναμενόμενο. Συνοψίζοντας, η τελική μας υλοποίηση στο openMP επιταχύνεται τόσο από την παραλληλοποίηση του openMP αλλά και λόγω της βελτιστοποίησης που συντελέστηκε στον κώδικα.

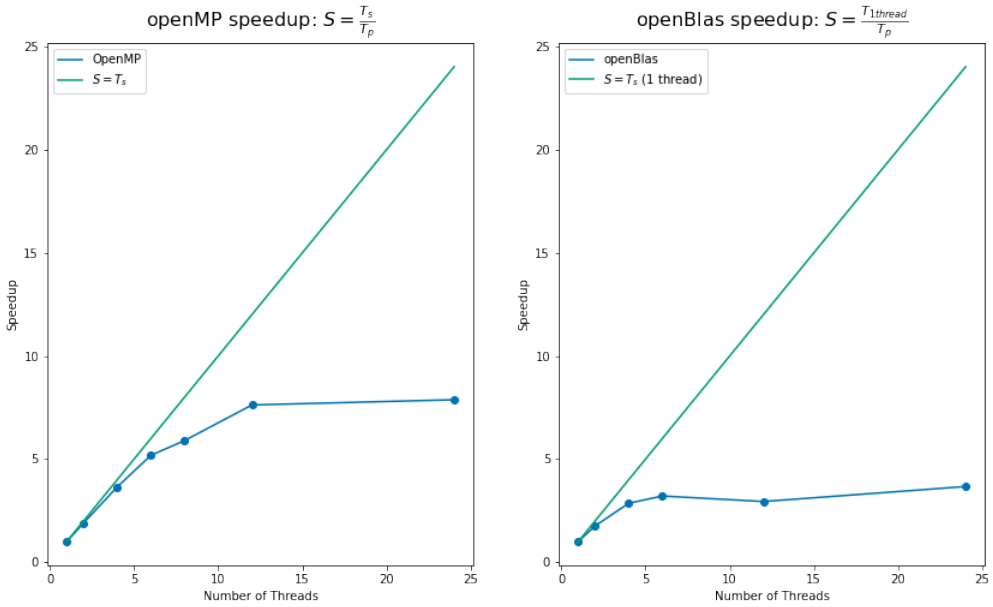

Fig. 6: Επιτάχυνση των openMP (ijk) και openBLAS με τους αντίστοιχους σειριακούς τους χρόνους εκτέλεσης.

4.2 Σύγκριση Επιδόσεων σε CPU και GPU

Στο παρακάτω διάγραμμα 7 φάινονται όλες οι υλοποιήσεις που εκτελέστηκαν στα πλαίσια της εργασίας, ενώ στο διάγραμμα 8 έχουμε κρατήσει τις καλύτερες. Ο χρόνος του serial code προκύπτει άμα απο τον ολικό χρόνο αφαιρέσουμε τον χρόνο του dgemm, dgemm_TA, dgemm_TB.

Αξίζει να σημειωθεί ότι το OpenMP επιτυγχάνει χρόνους πολύ κοντινούς σε αυτούς του OpenBLAS ενώ αρκετά αξιοπερίεργο είναι το γεγονός ότι το OpenBLAS φαίνεται πως χρειάζεται αισθητά περισσότερο χρόνο για την εκτέλεση του μη-παραλληλοποιήσιμου κώδικα. Τέλος, όπως έχει αναφερθεί και παραπάνω αναμενόταν πως θα υπήρχε μεγαλύτερο speedup με την χρήση του cuBLAS επομένως φαίνεται πως τα χαρακτηριστικά του μηχανήματος δεν ευνοούν την συγκεκριμένη περίπτωση χρήσης.

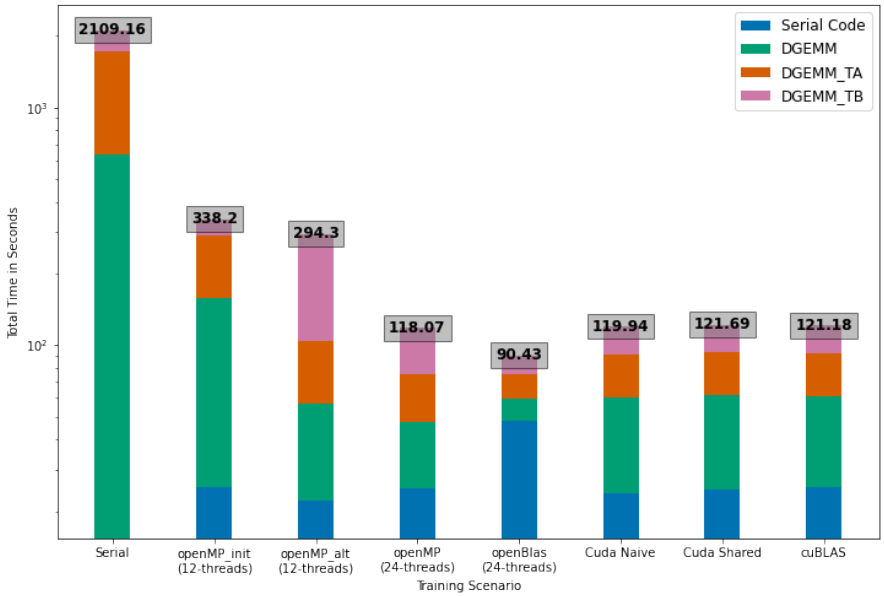

Fig. 7: Σύγκριση Επιδόσεων μεταξύ CPU & GPU. (semi-log scale)

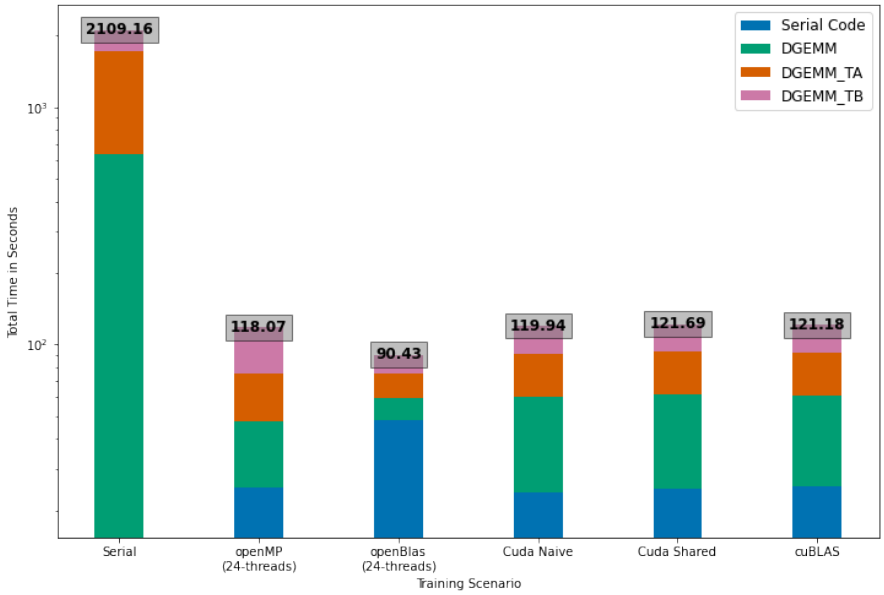

Fig. 8: Σύγκριση Επιδόσεων μεταξύ CPU & GPU. (semi-log scale)