Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Δ.Π.Μ.Σ. "Επιστήμη Δεδομένων και Μηχανική Μάθηση"

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

Συγγραφείς:
Παναγιώτης Χρονόπουλος 03400240
Ευαγγελία Κοσκινιώτη 03400219
Γιώργος Καλομοίρης 03400215
Ανδρέας Καλογεράς 03400214
Αντώνης Προμπονάς 03400232

Υπεύθυνοι Καθηγητές:
Γκούμας Γεώργιος
Σούντρης Δημήτριος
Κοζύρης Νεκτάριος

Η εργασία κατατέθηκε για το μάθημα:
Παράλληλες Αρχιτεκτονικές Υπολογισμού για Μηχανική Μάθηση

Ιούλιος 2024

1 Εισαγωγή

Σε αυτή την εργασία καλούμαστε να εφαρμόσουμε μια σειρά τεχνικών παραλληλοποίησης στα πλαίσια ενός προβλήματος μηχανικής μάθησης. Συγκεκριμένα, σκοπός μας είναι να επιταχύνουμε την διαδικασία εκπαίδευσης ενός νευρωνικού δικτύου που θα αναγνωρίζει χειρόγραφα αριθμητικά ψηφία, και για να επιτευχθεί αυτή η επιτάχυνση θα επιχειρήσουμε να παραλληλοποιήσουμε την πιο κοστοβόρα πράξη στην εκπαίδευση ενός μοντέλου, τον πολλαπλασιασμό πινάκων. Έτσι, θα πειραματιστούμε με την παραλληλοποίηση τριών παραλλαγών του πολλαπλασιασμού πινάκων, δηλαδή των πράξεων $A \cdot B, A^T \cdot B, A \cdot B^T + C$ με τους A, B, C να είναι οι πίνακες εισόδου. Παρακάτω αναλύουμε την υλοποίηση και τα αποτελέσματα της παραλληλοποίησης αυτών των τριών πράξεων τόσο σε CPU όσο και σε GPU, χρησιμοποιώντας διαφορετικά εργαλεία.

Ζητούμενο 2.1: Παραλληλοποίηση σε CPU

1: OpenMP

Θα ξεκινήσουμε παραλληλοποιώντας τον πολλαπλασιασμό με χρήση CPU και συγκεκριμένα θα αξιοποιήσουμε την βιβλιοθήκη **OpenMP**, η οποία χρησιμοποιείται ειδικά για την υλοποίηση παράλληλων διεργασιών για την γλώσσα C. Χρησιμοποιόντας την εντολή **#pragma omp parallel for** του API όπως φαίνεται στον κώδικα παρακάτω παραλληλοποιούμε τις επαναλήψεις των for loops αναθέτοντας τις επαναλήψεις σε διαφορετικά threads.

```c
#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;
  }
}
```

Παραλληλοποίηση $A \times B$

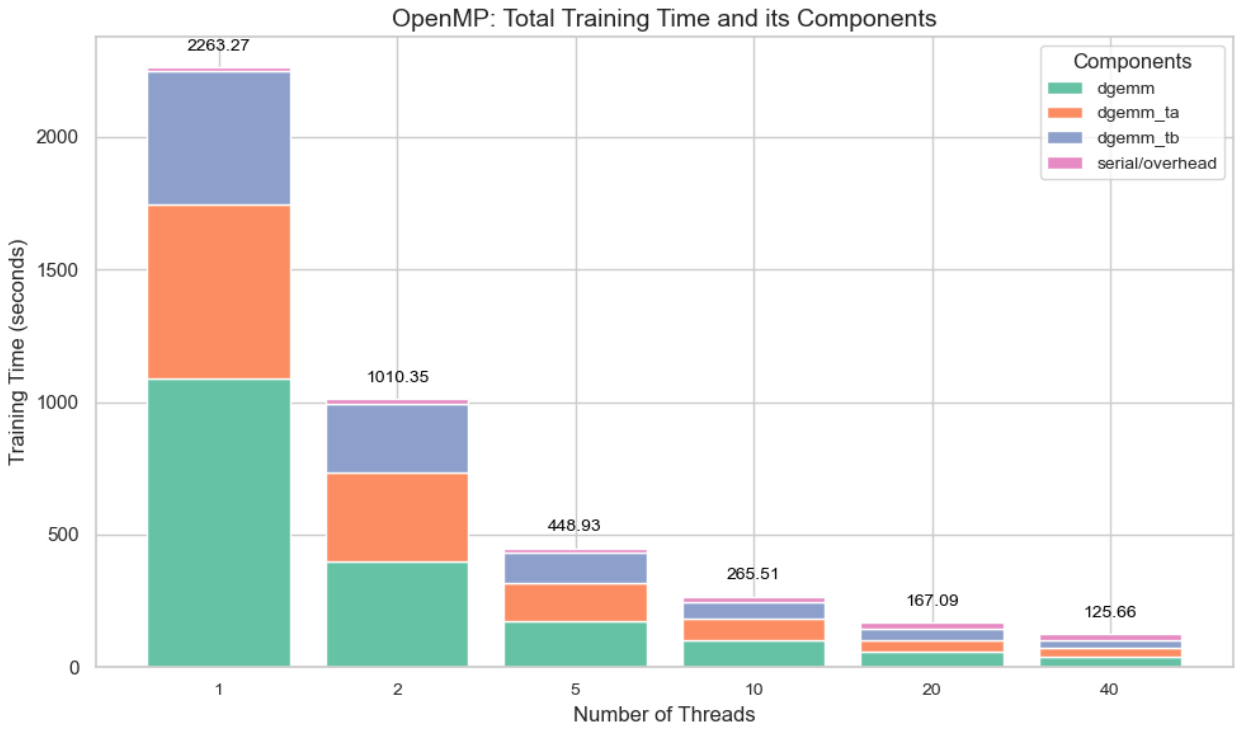
```c
#pragma omp parallel for
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    sum = 0.;
    for (k = 0; k < K; k++)
      sum+=A[k*M+i]*B[k*N+j];
    C[i * N + j] = sum;
  }
}
```

Παραλληλοποίηση $A^\top \times B$

```c
#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[j*K+k];
    D[i*N+j]=sum+C[i*N+j];
  }
}
```

Παραλληλοποίηση $A \times B^\top + C$

Εκτελούμε τον παραπάνω κώδικα για διαφορετικούς αριθμούς threads προκειμένου να μελετήσουμε την επίδραση στην απόδοση του προγράμματος και τα αποτελέσματα φαίνονται στο παρακάτω διάγραμμα.



Συνολικός Χρόνος Εκπαίδευσης με χρήση OpenMP ανάλογα με Threads

Κάθε bar στο παραπάνω bar plot αποτελείται από 4 στοιχεία: τον χρόνο που κατανάλωσε η πράξη $A \cdot B$, τον χρόνο που κατανάλωσε η πράξη $A \cdot B^T$, τον χρόνο που κατανάλωσε η πράξη $A \cdot B^T + C$ και τον χρόνο που χρειάστηκε για τις υπόλοιπες διάφορες λειτουργίες του κώδικα. Στην κουρυφή κάθε bar φαίνεται ο συνολικός χρόνος εκπαίδευσης και παρατηρούμε πως με την αύξηση του αριθμού των threads η εκπαίδευση γίνεται σημαντικά ταχύτερη, εώς και 18 φορές ταχύτερη όταν έχουμε 40 threads. Η δραματικότερη αύξηση παρατηρείται όταν εισάγεται αρχικά η παραλληλοποίηση και ο αριθμός των threads διπλασιάζεται, ο αντίστοιχος χρόνος εκτέλεσης φαίνεται να υποδιπλασιάζεται, μέχρι να φτάσουμε από τα 20 στα 40 threads όπου η επιτάχυνση δείχνει να συγκλίνει. Συνολικά, η εισαγωγή της παραλληλοποίησης μέσω του OpenMP επιταχύνει σημαντικά τη διαδικασία της εκπαίδευσης όπως ήταν αναμενόμενο, και ο μεγαλύτερος αριθμός threads (μέχρι ένα σημείο) μειώνει τον χρόνο εκπαίδευσης ολοένα και περισσότερο.

2: OpenBLAS

Στη συνέχεια θα υλοποιήσουμε την παραλληλοποίηση χρησιμοποιώντας την βιβλιοθήκη **OpenBLAS**, (Basic Linear Algebra Subprograms) μια open-source βιβλιοθήκη που παρέχει ένα σύνολο συναρτήσεων που πραγματοποιούν διάφορες πράξεις (μεταξύ των οποίων είναι και ο πολλαπλασιασμός πίνακων) αποδοτικά μέσω της παραλληλοποίησης. Συγκεκριμένα, θα χρησιμοποιήσουμε τη συνάρτηση **cblas_dgemm** αφού εξετάσουμε προσεκτικά τις παραμέτρους της και τις προσαρμόσουμε κατάλληλα σε κάθε πράξη. Ο τελικός κώδικας φαίνεται παρακάτω.

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

Παραλληλοποίηση $A \times B$

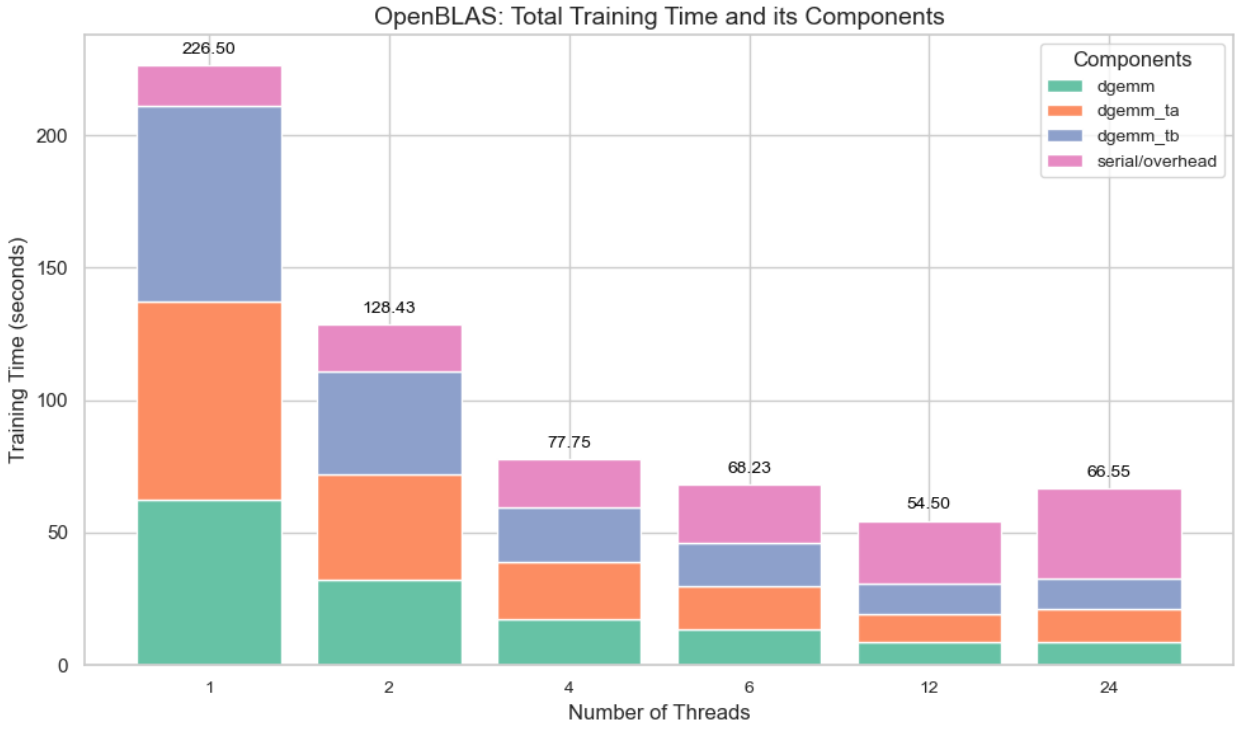
```
cblas_dgemm(CblasRowMajor,
    CblasTrans, CblasNoTrans,
     M, N, K, 1, A, M, B, N,
    0, C, N);
```

Παραλληλοποίηση $A^\top \times B$

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

Παραλληλοποίηση $A \times B^\top + C$

Όπως και στην περίπτωση του OpenMP, πραγματοποιούμε πολλά διαφορετικά πειράματα για διαφορετικό αριθμό threads και παρουσιάζουμε τους χρόνους στο παρακάτω διάγραμμα.



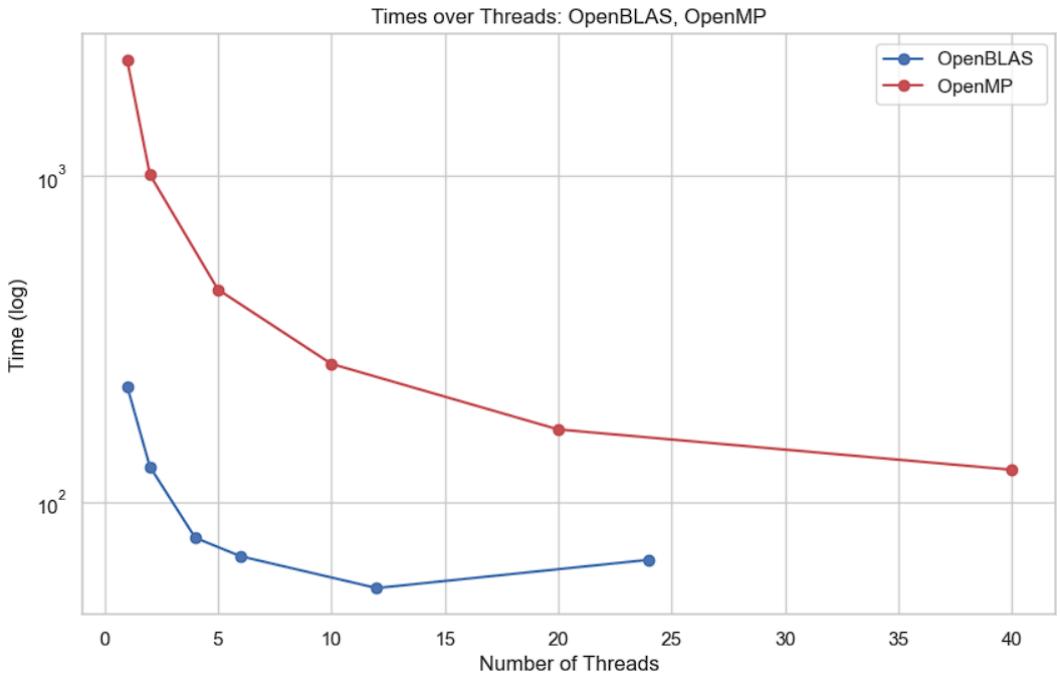
Συνολικός Χρόνος Εκπαίδευσης με χρήση OpenBLAS ανάλογα με Threads

Όπως βλέπουμε ξανά, η παραλληλοποίηση έχει σημαντική επίδραση στον χρόνο εκπαίδευσης και η αύξηση του αριθμού των threads οδηγεί σε ολοένα και μεγαλύτερη μείωση. Ωστόσο, μια πρώτη αξιόλογη παρατήρηση είναι ότι ακόμα και για 1 thread, ο χρόνος εκτέλεσης του OpenBLAS, που περιλαμβάνει ήδη βελτιστοποιημένες εκδοχές των πράξεων, είναι 10 φορές μικρότερος από ότι του OpenMP, και έτσι η μείωση του χρόνου εκτέλεσης συγκλίνει για μικρότερο αριθμό threads. Καταλήγοντας, η χρήση του OpenBLAS έχει πολύ σημαντικά αποτελέσματα στην επιτάχυνση του χρόνου υλοποίησης των πράξεων που μελετάμε, σε μεγαλύτερο βαθμό από το OpenMP.

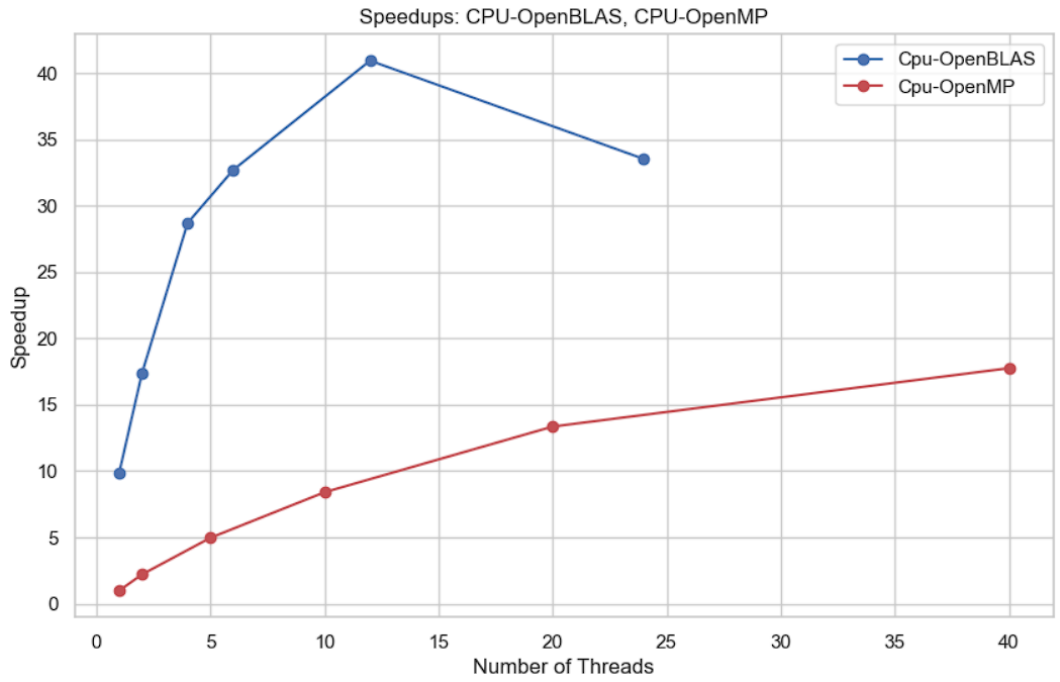
Κλιμακωσιμότητα σε CPU

Σε αυτή την ενότητα θα μελετήσουμε την κλιμακωσιμότητα των τεχνικών παραλληλοποίησης που εξετάσαμε παραπάνω ανάλογα με τον απαραίτητο αριθμό threads. Συγκεκριμένα, θα μελετήσουμε πως μεταβάλλεται το speedup και ο συνολικός χρόνος εκτέλεσης όσο αυξάνεται ο αριθμός των threads στην περίπτωση του OpenMP και του OpenBLAS αντίστοιχα. Για αυτόν τον σκοπό παραθέτουμε τα αντίστοιχα διαγράμματα.

(Σημείωση: Λόγω αλλαγών στο submission script έχουμε παλαιότερη έκδοση όπου τα πλήθη των threads διαφέρουν μεταξύ OpenBLAS και OpenMP)



Χρόνοι Εκπαίδευσης ανάλογα με Threads - OpenMP vs OpenBLAS



Speedup ανάλογα με Threads - OpenMP vs OpenBLAS

Στο διάγραμμα των συνολικών χρόνων εκτέλεσης επιλέξαμε λογαριθμική κλίμακα για τον άξονα του χρόνου καθώς υπήρχε σημαντική διαφορά ανάμεσα στις τάξεις μεγέθους. Και στις δύο περιπτώσεις, με την αρχική αύξηση του αριθμού των threads παρατηρείται μείωση του χρόνου εκτέλεσης και αύξηση του speedup. Ωστόσο, αυτή η τάση φτάνει σε κορεσμό, γρηγορότερα στο OpenBLAS από ότι στο OpenMP, γεγονός που είναι λογικό καθώς η

επιτάχυνση του OpenBLAS ήταν εξαρχής πολύ μεγαλύτερη. Ξανά στην περίπτωση του OpenBLAS παρατηρούμε ότι μετά από ένα σημείο αύξησης του αριθμού των threads ο χρόνος εκτέλεσης αρχίζει να αυξάνεται, γεγονός που σημαίνει ότι αυτή η προσέγγιση δεν είναι άπειρα κλιμακώσιμη, δηλαδή δεν μπορούμε να αυξάνουμε αόριστα τον αριθμό των threads και να παίρνουμε συνέχεια καλύτερους χρόνους εκτέλεσης και καλύτερη επιτάχυνση. Η μεγαλύτερη βελτίωση στον χρόνο παρατηρείται για μικρούς αριθμούς threads και για τις δύο προσεγγίσεις με το OpenBLAS να υπερισχύει σημαντικά σε αυτές τις περιπτώσεις, ωστόσο το OpenMP κλιμακώνει καλύτερα για μεγαλύτερους αριθμούς threads. Τα ίδια συμπεράσματα εξάγονται και για τα speedup, όπου λόγω της κλίμακας φαίνεται καλύτερα η πτώση της επιτάχυνσης στο OpenBLAS για 24 threads. Συνολικά, το συμπέρασμα που μπορούμε να εξάγουμε είναι ότι τόσο το OpenMP όσο και το OpsnBLAS κλιμακώνουν καλά με την αύξηση του αριθμού των threads, με το OpenMP να αυξάνεται με πιο αργό ρυθμό αλλά να έχει σταθερή αύξηση για όλες τις τιμές και το OpenBLAS να έχει πιο απότομη βελτίωση με την αρχική αύξηση του αριθμού των threads και να φθίνει μετά από ένα σημείο. Παρόλα αυτά και οι δύο βιβλιοθήκες μας επιβεβαιώνουν την σημαντική επίδραση της παραλληλοποίησης στην αποδοτικότητα και την ταχύτητα με την οποία πραγματοποιείται ο πολλαπλασιασμός πινάκων.

Ζητούμενο 2.2: Παραλληλοποίηση σε GPU

Σε αυτή την ενότητα θα υλοποιήσουμε τις ίδιες πράξεις με προηγουμένως, αλλά αυτή τη φορά χρησιμοποιώντας το API του CUDA και αξιοποιώντας το για να προγραμματίσουμε παράλληλα χρησιμοποιώντας τη GPU.

1: Shared Memory

Θα ξεκινήσουμε μελετώντας την υλοποίηση κοινής μνήμης. Με τον όρο κοινή μνήμη (**shared memory**) αναφερόμαστε σε μνήμη η οποία βρίσκεται on-chip και στην οποία έχουν πρόσβαση όλα τα νήματα (threads) του κάθε block της GPU. Επειδή είναι on-chip η πρόσβαση είναι πολύ γρηγορότερη από την αντίστοιχη στη καθολική μνήμη, καθιστώντας την ιδανική για σενάρια όπου τα threads εκτελούν πράξεις σε κοινά δεδομένα και πρέπει να παραμένουν ενημερωμένα για τη κατάσταση των πράξεων. Για αυτό το λόγο είναι ιδανική για τον πολλαπλασιασμό πινάκων.

Αρχικά ορίσαμε τις παραμέτρους για την εκτέλεση του CUDA kernel. Το block περιλαμβάνει BLOCK_SIZE x BLOCK_SIZE (16) threads. Το grid ορίστηκε με τέτοιο τρόπο ώστε να εξασφαλίσουμε πως τα blocks θα καλύπτουν όλα τα δεδομένα, ακόμα και εάν το N (πλήθος στηλών του πίνακα C) δεν είναι πολλαπλάσιο του BLOCK_SIZE.

```
dim3 block(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid((N + BLOCK_SIZE - 1)/BLOCK_SIZE, (N + BLOCK_SIZE - 1)/BLOCK_SIZE);
size_t shmem_size = BLOCK_SIZE;
```

Για τις υλοποιήσεις των τριών πράξεων αξιοποιήσαμε τη τεχνική του tiled computation. Και στις τρεις περιπτώσεις ξεκινήσαμε ορίζοντας δύο δισδιάστατα arrays στη κοινή μνήμη μεγέθους TILE_WIDTH * TILE_WIDTH (16) και χρησιμοποιήσαμε τους δείκτες των blocks και των threads ώστε να μπορούμε να προσπελαύνουμε οποιοδήποτε στοιχείο των πινάκων.

```
__shared__ double shared_A[TILE_WIDTH][TILE_WIDTH];
__shared__ double shared_B[TILE_WIDTH][TILE_WIDTH];

int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;

int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;
```

Έπειτα για κάθε περίπτωση προσαρμόσαμε τον κώδικα ώστε οι δείκτες να αντιστοιχούν στα σωστά στοιχεία των πινάκων A και B ανάλογα εάν είναι ανεστραμμένοι ή όχι. Για να εξασφαλίσουμε πως όλα τα νήματα θα

είναι συγχρονισμένα, δηλαδή θα αναμένουν τα υπόλοιπα να φτάσουν στο ίδιο σημείο της εκτέλεσης του κώδικα, ώστε να είναι σωστές οι πράξεις, χρησιμοποιήσαμε την εντολή **__syncthreads()**.

```
double Cvalue = 0.0;

for (int t = 0; t < (K - 1) / TILE_WIDTH + 1; ++t) {
    if (Row < M && t * TILE_WIDTH + tx < K)
        shared_A[ty][tx] = A[Row * K + t * TILE_WIDTH + tx];
    else
        shared_A[ty][tx] = 0.0;

    if (t * TILE_WIDTH + ty < K && Col < N)
        shared_B[ty][tx] = B[(t * TILE_WIDTH + ty) * N + Col];
    else
        shared_B[ty][tx] = 0.0;

    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Cvalue += shared_A[ty][i] * shared_B[i][tx];

    __syncthreads();
}

if (Row < M && Col < N)
    C[Row * N + Col] = Cvalue;
```

Παραλληλοποίηση $A \times B$

```
double Cvalue = 0.0;

for (int t = 0; t < (K - 1) / TILE_WIDTH + 1; ++t) {
    if (Row < M && t * TILE_WIDTH + tx < K)
        shared_A[ty][tx] = A[(t * TILE_WIDTH + tx) * M + Row];
    else
        shared_A[ty][tx] = 0.0;

    if (t * TILE_WIDTH + ty < K && Col < N)
        shared_B[ty][tx] = B[(t * TILE_WIDTH + ty) * N + Col];
    else
        shared_B[ty][tx] = 0.0;

    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Cvalue += shared_A[ty][i] * shared_B[i][tx];

    __syncthreads();
}

if (Row < M && Col < N)
    C[Row * N + Col] = Cvalue;
```

Παραλληλοποίηση $A^\top \times B$

```
double Dvalue = 0.0;

for (int t = 0; t < (K - 1) / TILE_WIDTH + 1; ++t) {
    if (Row < M && t * TILE_WIDTH + tx < K)
        shared_A[ty][tx] = A[Row * K + t * TILE_WIDTH + tx];
    else
        shared_A[ty][tx] = 0.0;

    if (t * TILE_WIDTH + ty < K && Col < N)
        shared_B[ty][tx] = B[Col * K + t * TILE_WIDTH + ty];
    else
        shared_B[ty][tx] = 0.0;

    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Dvalue += shared_A[ty][i] * shared_B[i][tx];

    __syncthreads();
}

if (Row < M && Col < N)
    D[Row * N + Col] = Dvalue + C[Row * N + Col];
```

Παραλληλοποίηση $A \times B^\top + C$

2: cuBLAS

Το **cuBLAS** (CUDA Basic Linear Algebra Subroutines) είναι μια βιβλιοθήκη υλοποιημένη από την NVIDIA, η οποία αξιοποιεί τη δυνατότητα παραλληλισμού των GPUs για να παρέχει ένα σύνολο μεθόδων που εκτελούν βασικές πράξεις γραμμικής άλγεβρας εξαιρετικά γρήγορα. Για τις πράξεις μας θα αξιοποιηθούν οι συναρτήσεις **cublasDgemm** για τον πολλαπλασιασμό και **cublasDgeam** για την πρόσθεση πινάκων αντίστοιχα.

```
cublasHandle_t handle;

cublasCreate(&handle);

double alpha = 1.0;

double beta  = 0.0;


cublasDgemm(handle,
    CUBLAS_OP_N, CUBLAS_OP_N,
     N, M, K, &alpha, B, N, A
    , K, &beta, C, N);
```

Παραλληλοποίηση $A \times B$

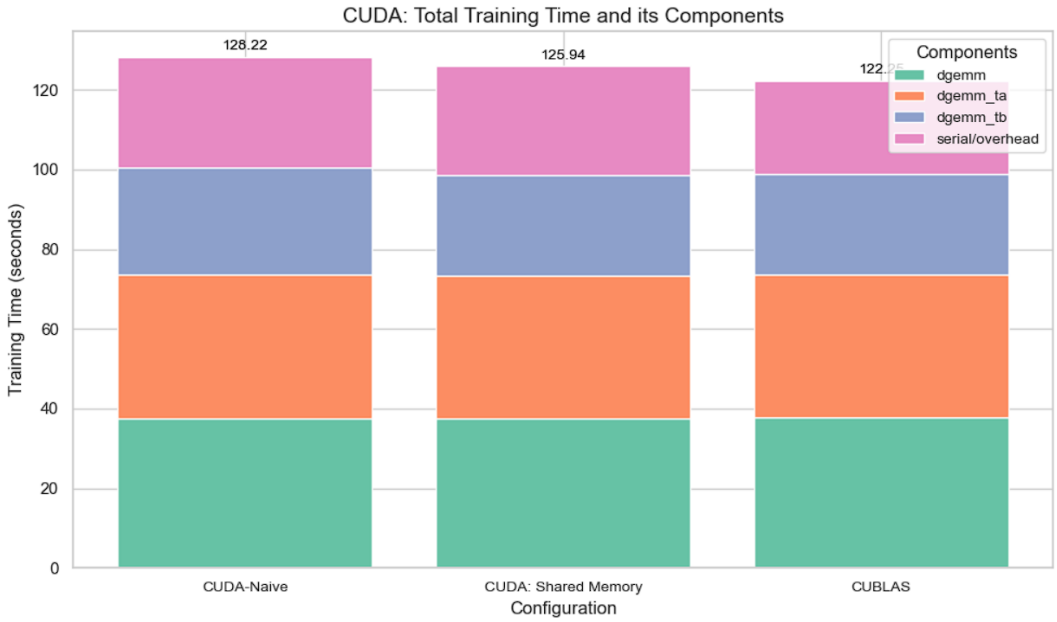
```
cublasHandle_t handle;

cublasCreate(&handle);

double alpha = 1.0;

double beta  = 0.0;


cublasDgemm(handle,
    CUBLAS_OP_N, CUBLAS_OP_T,
     N, M, K, &alpha, B, N, A
    , M, &beta, C, N);
```

Παραλληλοποίηση $A^\top \times B$

```
cublasHandle_t handle;
cublasCreate(&handle);
double alpha = 1.0;
double beta  = 0.0;
cublasDgemm(handle,
    CUBLAS_OP_T, CUBLAS_OP_N,
     N, M, K, &alpha, B, K, A
    , K, &beta, D, N);
beta = 1.0;
cublasDgeam(handle,
    CUBLAS_OP_N, CUBLAS_OP_N,
     N, M, &alpha, C, N, &
    beta, D, N, D, N)
```

Παραλληλοποίηση $A \times B^\top + C$

Παρακάτω παρατίθεται το διάγραμμα των συνολικών χρόνων εκτέλεσης αλλά και των χρόνων που αναλώνονται στις τρεις πράξεις πινάκων. Παρατηρούμε πως δεν υπάρχει ουσιαστική διαφορά, η naive υλοποίηση είναι ελαφρώς πιο αργή, αλλά όχι αρκετά για να εξάγουμε κάποιο ουσιαστικό συμπέρασμα. Ενδεχομένως σε ένα πρόβλημα με περισσότερες πράξεις ή/και περισσότερα δεδομένα να υπήρχε μεγαλύτερο όφελος με τη χρήση κοινής μνήμης και να παρατηρούσαμε διαφορά στους χρόνους.



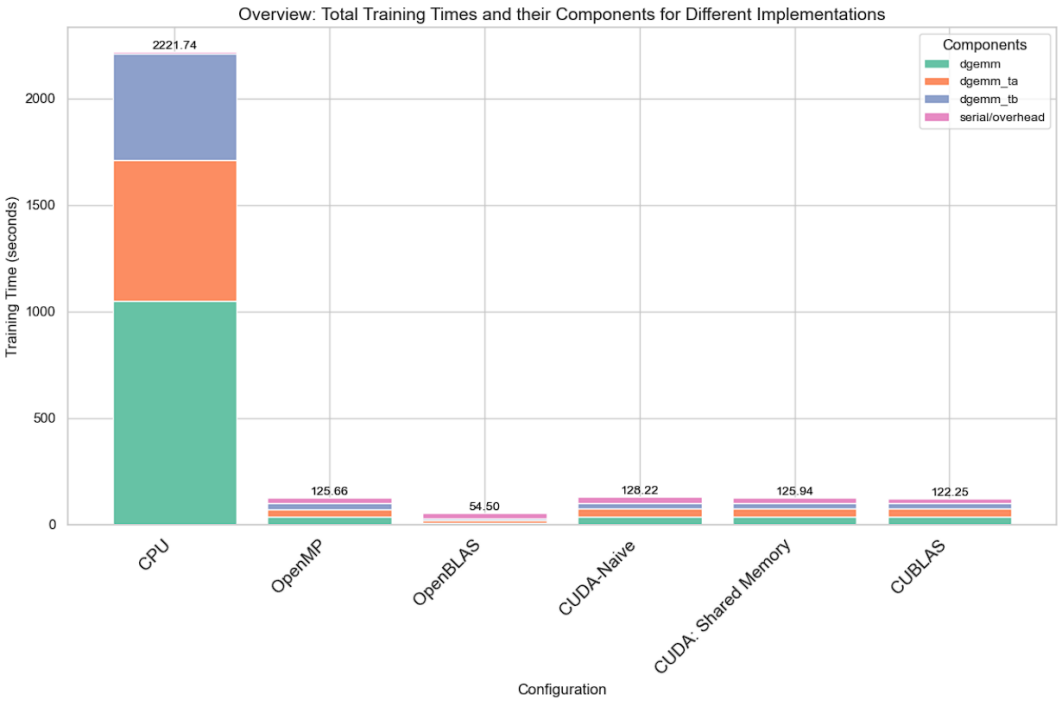
Χρόνοι εκτέλεσης για τις υλοποιήσεις με CUDA

Σύγκριση επιδόσεων σε CPU και GPU

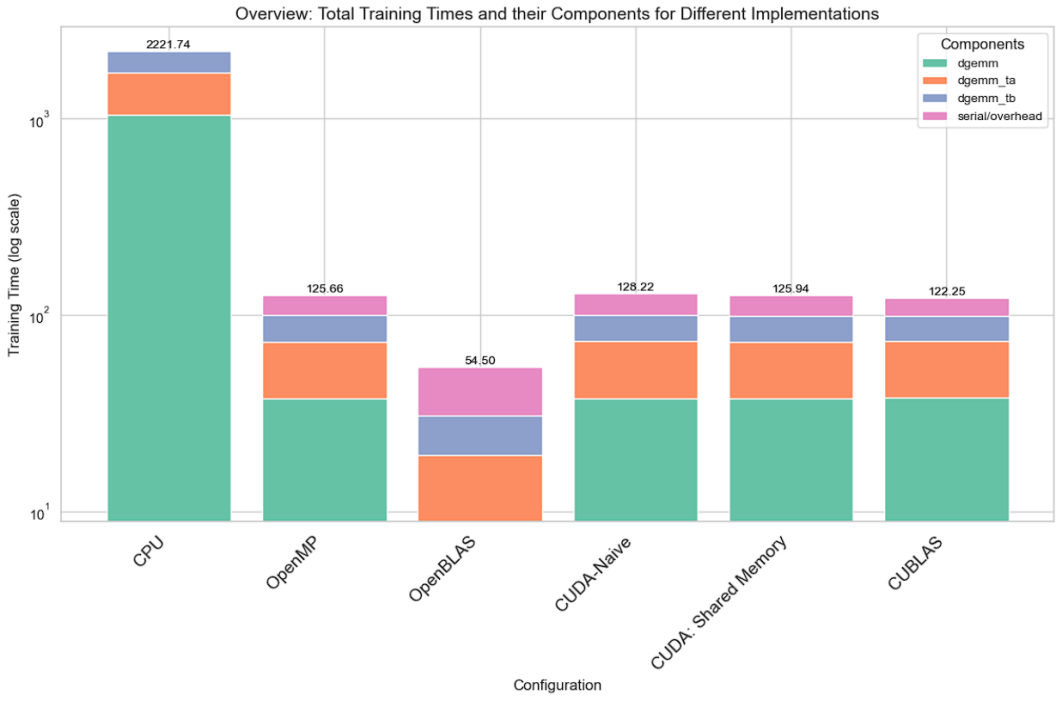
Παρακάτω παρατίθονται συγκεντρωτικά διαγράμματα για τους χρόνους εκτέλεσης καθώς και τους χρόνους που αναλώθηκαν σε πολλαπλασιασμούς πινάκων, σε κανονική και λογαριθμική κλίμακα. Να σημειωθεί πως για τις υλοποιήσεις OpenMP και OpenBLAS επιλέχθηκαν οι χρόνοι των καλύτερων επιδόσεων (για 40 threads και 12 threads αντίστοιχα).

Η καλύτερη παραλληλοποίηση επιτυγχάνεται με το OpenBLAS και μάλιστα με σημαντική διαφορά από τις υπόλοιπες (πάνω από 50% πτώση). Η OpenBLAS είναι βιβλιοθήκη η οποία έχει βελτιστοποιηθεί για παραλληλοποίηση σε CPU και αξιοποιεί στο έπαρκο τους πόρους και την αρχιτεκτονική.

Η χρήση GPU δε φαίνεται να βοήθησε στη παραλληλοποίηση, μάλιστα και οι τρεις υλοποιήσεις πέτυχαν περίπου την ίδια απόδοση με το OpenMP ενώ χειρότερη από το OpenBLAS. Ενδεχομένως το κόστος συγχρονισμού/επικοινωνίας να οδηγεί σε αύξηση του χρόνου εκτέλεσης και αν είχαμε ένα πρόβλημα με περισσότερα δεδομένα ή/και παραμέτρους να παρατηρούσαμε βελτίωση.



Σύγκριση επιδόσεων για τις διαφορετικές υλοποιήσεις



Σύγκριση επιδόσεων για τις διαφορετικές υλοποιήσεις (λογαριθμική κλίμακα)