Εθνικο Μετσοβιο Πολυτεχνειο

Επιστημη Δεδομενων και Μηχανικη Μαθηση

Παράλληλες Αρχιτεκτονικές Υπολογισμού για Μηχανική Μάθηση

ΕΡΓΑΣΙΑ

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

Βλάχος Ιωάννης	ΑΜ: 03400087	ioannisvlahos@mail.ntua.gr
Ορφανουδάκης Φίλιππος Σκόβελεφ	ΑΜ: 03400107	philipposorfanoudakis@mail.ntua.gr
Πανόπουλος Ιωάννης	ΑΜ: 03003050	ioannispanop@mail.ntua.gr
Παπαγεωργίου Δημήτριος	ΑΜ: 03400088	dimitriospapageorgiou@mail.ntua.gr

Αθήνα, Ιούνιος 2021

Περιεχόμενα

1. Εισαγωγή

Στα πλαίσια της άσκησης μας ζητείται να εκπαιδεύσουμε ένα νευρωνικό δίκτυο με 3 κρυφά επίπεδα, με στόχο την κατηγοριοποίηση χειρόγραφων ψηφίων του συνόλου δεδομένων MNIST. Η εκπαίδευση ενός νευρωνικού δικτύου, περιλαμβάνει αρκετούς πολλαπλασιασμούς πινάκων. Όπως γνωρίζουμε η απλή υλοποίηση του πολλαπλασιασμού πινάκων έχει $O(n^3)$ πολυπλοκότητα, και αυτό οφείλεται στα εμφωλευμένα loops που περιλαμβάνει. Επίσης η απλή υλοποίηση οδηγεί σε αρκετά cache misses τα οποία επιβαρύνουν την απόδοση και τον χρόνο εκτέλεσης. Για αυτούς τους λόγους, βασικό ζητούμενο της άσκησης είναι η αποδοτικότερη υλοποίηση του αλγορίθμου πολλαπλασιασμού πινάκων (GEneral Matrix Multiply, GEMM) με κύριο εργαλείο, την παραλληλοποίηση. Πιο συγκεκριμένα, θα ασχοληθούμε με την βελτιστοποίηση του αλγορίθμου τόσο σε CPU όσο και σε GPU.

1.1 CPU

Η παραλληλοποίηση σε CPU γίνεται είτε με το προγραμματιστικό μοντέλο OpenMP, ή με τη χρήση της βιβλιοθήκης OpenBLAS. Το μηχάνημα που χρησιμοποιήθηκε αποτελείται από δύο επεξεργαστές Intel Xeon X5650 (6 πυρήνες + 2 hyper-threads ανά πυρήνα, συνολικά 12 πυρήνες + 24 hyper-threads), στο οποίο έχουμε πρόσβαση από την ουρά termis.

1.2 GPU

Το παρόν κομμάτι της αναφοράς αφορά στην παραλληλοποίηση της διαδικασίας εκπαίδευσης ενός νευρωνικού δικτύου σε GPUs. Το κομμάτι που παραλληλοποιείται μέσω CUDA ή cuBLAS είναι οι πολλαπλασιασμοί πινάκων, ενώ τα πειράματα εκτελούνται πάνω στο μηχάνημα dungani που αποτελείται από έναν επεξεργαστή Intel i7-4820K (4 πυρήνες + 2 hyper-threads ανά πυρήνα, συνολικά 8 hyper-threads) και μία κάρτα γραφικών NVIDIA Tesla K40, στο οποίο έχουμε πρόσβαση από την ουρά serial.

1.3 Κώδικας προς Βελτιστοποίηση

Μας δίνονται 3 τμήματα κώδικα, καθένα εκ των οποίων περιέχει μιας μορφής πολλαπλασιασμό πινάκων A και B με την naive υλοποίηση τους. Η naive υλοποίηση αποτελείται από 3 loops. Το πρώτο loop σκανάρει τις γραμμές του A, το δεύτερο σκανάρει τις στήλες του B και το τρίτο παίρνει κάθε στοιχείο της στήλης του B. Πιο συγκεκριμένα τα τμηματα κώδικα αναφέρονται στον φάκελο που μας δίνεται ως **dgemm**, **dgemm_ta** και **dgemm_tb**. Πιο συγκεκριμένα :

- **dgemm**: $A \cdot B$
- **dgemm_ta**: $A^T \cdot B$
- **dgemm_tb**: $A \cdot B^T + C$

1.4 Έλεγχος Ορθότητας

Πριν προχωρήσουμε στην ανάλυση και επεξήγηση της μεθοδολογίας που εφαρμόσαμε, να αναφέρουμε πως όλα τα προγράμματα μας, έχουν περάσει τον έλεγχο ορθότητας που έχει τεθεί και παράγουν σωστά αποτελέσματα.

2. Κώδικας και Υλοποίηση

2.1 OpenMP

Όπως αναλύσαμε και στην προηγούμενη ενότητα τα τμήματα κώδικα προς βελτιστοποίηση αποτελούνται απο εμφωλευμένα for loops. Η οδηγία στο OpenMP που παραλληλοποιεί τα for loops είναι η **#pragma omp parallel for**. Επίσης, καθως παραλληλοποιούμε τα for loops θέλουμε για κάθε στοιχείο του νέου πίνακά μας, που είναι το γινόμενο, να συσσωρεύουμε όλα τα επιμέρους αθροίσματα. Κάθε thread που έχει αναλάβει τον υπολογισμό ενός από τα πολλά αθροίσματα που χρειάζεται για τον υπολογισμό ενός στοιχείου γινομένου C_{ij}, προσθέτει το αποτέλεσμα σε μια κοινή μεταβλητή. Για αυτό το σκοπό το OpenMP προσφέρει την οδηγία reduction(+:sum). Εμείς θα πειραματιστούμε με αυτές τις 2 εντολές για να βρούμε τελικά την υλοποίηση που μας επιστρέφει τους καλύτερους χρόνους.

Οι δοκιμές που πραγματοποιήθηκαν είναι οι εξής :

- #pragma omp parallel colapse(3), δηλαδή unroll τα εμφωλευμένα loops και έπειτα παραλληλοποίηση τους
- #pragma omp parallel for shared(i,j) private(k) reduction(+:sum), δηλαδή παραλληλοποίηση του 2ου εσωτερικού loop.
- #pragma omp parallel for shared(A, B, C,i) private(j, k, sum), δηλαδή παραλληλοποίηση του 1ου εσωτερικού for loop. Πλέον κάθε thread κοιτάει μια στήλη του πίνακα B και επομένως αναλαμβάνει μια στήλη του γινομένου.
- #pragma omp parallel for shared(A, B, C) private(i, j, k, sum), πλέον κάθε thread κοιτάει μια γραμμή του A και επομένως αναλαμβάνει μια γραμμή του γινομένου.

Από αυτές τις 4 δοκιμές τα καλύτερα αποτελέσματα από άποψη χρόνου, σημειώθηκαν στην τελευταία υλοποίηση, δηλαδή στην παραλληλοποίηση γραμμής. Οι αντίστοιχες υλοποιήσεις ειναι οι εξής :

dgemm:

```
#pragma omp parallel for shared(A, B, C) private(i, j, k, sum)
  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;
    }
  }
```

dgemm_ta:

```
#pragma omp parallel for shared(A, B, C) private(i, j, k, 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;
    }
  }
```

dgemm_tb:

```
#pragma omp parallel for shared(A, B, C, D) private(i, j, k, sum)
  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];
    }
  }
```

Ως shared θέτουμε μόνο τους πίνακες A,B,C,D, οι οποίοι πρέπει να υπάρχουν σε κάθε thread και στη συνέχεια τις υπόλοιπες μεταβλτητες τις θέτουμε ως private έτσι ώστε να αποφύγουμε τα conflicts και κάθε thread να αναλάβει αποκλειστικά μια γραμμή του γινομένου.

2.2 OpenBLAS

Για την αξιοποίηση της βιβλιοθήκης OpenBLAS, χρησιμοποιούμε την συνάρτηση cblas_dgemm. Αυτή δέχεται τα εξής ορίσματα:

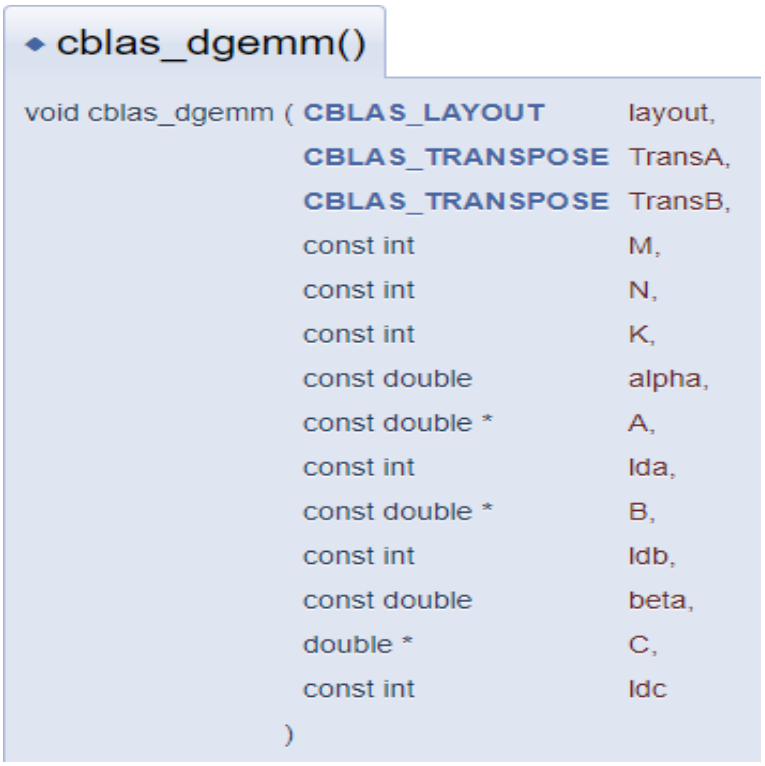

Figure 2.1: OpenBlas Documentation

Αν ο πίνακας που δίνεται ως όρισμα δεν είναι ανάστροφος, τότε δίνεται το όρισμα CblasNoTrans, αλλιώς δίνεται το όρισμα CblasTrans. Το leading dimension κάθε ορίσματος είναι ο αριθμός των στηλών του πίνακα στη μορφή που δίνεται, δηλαδή είτε σε κανονική μορφή είτε ανάστροφος. Πιο συγκεκριμένα, οι υλοποιήσεις μας είναι οι παρακάτω :

dgemm:

```
cblas_dgemm(CblasRowMajor,
              CblasNoTrans,
              CblasNoTrans,
              M,
              N,
              K,
              1, // alpha = 1
              A,
              K, // Leading dimension (number of columns) of A = M x K
              B,
              N, // Leading dimension (number of columns) of B = K x N
              0, // beta = 0
              C,
              N // Leading dimension (number of columns) of C = M x N
 );
```

dgemm_ta:

```
cblas_dgemm(CblasRowMajor,
              CblasTrans,
              CblasNoTrans,
              M,
              N,
              K,
              1, // alpha = 1
              A,
              M, // Leading dimension (number of columns) of A = M x K
              B,
              N, // Leading dimension (number of columns) of B = K x N
              0, // beta = 0
              C,
              N // Leading dimension (number of columns) of C = M x N
 );
```

dgemm_tb:

```
cblas_dgemm(CblasRowMajor,
              CblasNoTrans,
              CblasTrans,
              M,
              N,
              K,
              1, // alpha = 1
              A,
              K, // Leading dimension (number of columns) of A = M x K
              B,
              K, // Leading dimension (number of columns) of B = K x N
              1, // beta = 1
              C,
              N // Leading dimension (number of columns) of C = M x N
 );
 memcpy(D, C, M * N * sizeof(double));
```

Στην τελευταία υλοποίηση ο B^T έχει διαστάσεις $N \times K$, άρα το leading term θα είναι ο αριθμός των στηλών δηλαδή K. Συγκεκριμένα η τελευταία υλοποίηση απαιτεί την εξής πράξη πινάκων :

$D = A \cdot B^T + C$. Η cblas_dgemm αποθηκεύει το αποτέλεσμα της στον πίνακα C, και στο τέλος αντιγράφουμε το αποτέλεσμα μας στον πίνακα D.

2.3 CUDA

Η υλοποίηση της άσκησης αποτελείται από την χρήση κοινής μνήμης στο περιβάλλον CUDA. Παρακάτω εξετάζονται οι ιδέες που χρησιμοποιήθηκαν.

Στην υλοποίηση μας συμμετέχουν πολλά threads για τον υπολογισμό ενός αποτελέσματος του πίνακα C. Συγκεκριμένα, πολλά threads υπολογίζουν τα μερικά αθροίσματα ενός tile των πινάκων εισόδου και εξόδου. Όμως ένα thread τη φορά είναι επιφορτισμένο για την ανάθεση της τελικής τιμής σε κάθε κελί του C πίνακα που καλούμαστε να υπολογίσουμε.

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
int local_tidx = threadIdx.x;
int local_tidy = threadIdx.y;
```

Κάθε block περιέχει μέρος των πινάκων A, B (tile) και εντός αυτού θα γίνεται ο υπολογισμός των μερικών αθροισμάτων για ένα μέρος του συνολικού αθροίσματος των θέσεων C[i][j] του εν λόγω κομματιού.

Σε ό,τι αφορά το kernel launch configuration, έχουμε:

- BLOCK SIZE = 16
- grid μεγέθους $N \times M$ blocks
- κάθε block είναι διδιάστατο μεγέθους $BLOCK\_SIZE \times BLOCK\_SIZE$
- Η κοινή μνήμη κάθε block έχει μέγεθος 2 × BLOCK_SIZE × BLOCK_SIZE × sizeof(double), ώστε να χωράνε τα περιεχόμενα των πινάκων A, B για ένα tile τύπου double.

```
dim3 block(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE,
            (M + BLOCK_SIZE - 1) / BLOCK_SIZE);
size_t shmem_size = 2 * BLOCK_SIZE * BLOCK_SIZE * sizeof(double);
dgemm_tb_optimized<<<grid, block, shmem_size>>>(A, B, C, D, M, N, K,
   BLOCK_SIZE);
checkCudaErrors(cudaPeekAtLastError());
checkCudaErrors(cudaDeviceSynchronize());
```

Στη συνέχεια δουλεύουμε στην 3η διάσταση μεγέθους K, στην οποία υπολογίζονται τα αθροίσματα. Σε κάθε block φορτώνεται ένα μέρος του πίνακα A και πίνακα B κατά γραμμή και κατά στήλη αντίστοιχα. Διαπερνάμε με τόσα blocks (που ισοδυναμούν με tiles) όσα χρειάζεται για να υπολογιστεί το άθροισμα των γινομένων. Σε κάθε block φορτώνουμε στην κοινή μνήμη στοιχεία των σειρών του A και των στηλών του B που θέλουμε. Ύστερα, υπολογίζουμε τα μερικά αθροίσματα για τα περιεχόμενα που υπάρχουν σε ένα block μόνο για την **κοινή μνήμη**. Πριν και μετά από αυτόν τον υπολογισμό συγχρονίζουμε τα threads ώστε να μην έχουμε προβλήματα στην προσπέλαση της μνήμης από άλλα threads που ενδεχομένως δεν έχουν ολοκληρώσει τους υπολογισμούς τους. Τελικά καλύπτουμε επαναληπτικά όλη την κοινή διάσταση των πινάκων, κατά την οποία πολλαπλασιάζουμε.

```
#pragma unroll
    for (int current_block = 0; current_block < (K - 1) / SHARED_MEM_DIM +
   1; current_block++)
```

```
    {
      // Column j of matrix A (sliding offset per tile)
      j = current_block * blockDim.x + threadIdx.x;
      // Row i of matrix B (sliding offset per tile)
      i = current_block * blockDim.y + threadIdx.y;
      // Load A[row][j] to shared mem from global mem
      partials_A[local_tidy * SHARED_MEM_DIM + local_tidx] = A[row * K + j];
      // Load B[i][col] to shared mem from global mem
      partials_B[local_tidy * SHARED_MEM_DIM + local_tidx] = B[i * N + col];
      // Synchronize before computation so that shared memory is full !
      __syncthreads();
      // Calculate **partial** dot products ONLY from shared memory !
      // In a single tile, SHARED_MEM_DIM * SHARED_MEM_DIM partial
      // results will be computed.
      // ( & unroll loop for performance)
#pragma unroll
      for (int k = 0; k < SHARED_MEM_DIM; k++)
      {
        // No shared memory bank conflict!
        temp += partials_A[local_tidy * SHARED_MEM_DIM + k] * partials_B[k *
    SHARED_MEM_DIM + local_tidx];
      }
      // Synchronize before next iteration so that previous
      // values of shared memory are not overwritten BEFORE
      // another thread has not used them yet!
      // (for whatever reason)
      __syncthreads();
    }
```

Τέλος, αφού έχουμε διατρέξει όλα τα απαραίτητα blocks για να υπολογίσουμε K γινόμενα και να τα αθροίσουμε, αποθηκεύουμε το αποτέλεσμα στο αντίστοιχο κελί του C στην global memory.

```
C[row * N + col] = temp;
```

Για τις παραλλαγές του πολλαπλασιασμού πινάκων, αρκεί να αλλάξουμε τον τρόπο που διατρέχουμε τους πίνακες A, B ανάλογα με τον πολλαπλασιασμό που θέλουμε να εκτελέσουμε. Για παράδειγμα, στην περίπτωση του $C = A^T B$, πρέπει να διατρέξουμε και τους 2 πίνακες ανά γραμμή, ώστε ο A να θεωρηθεί ανάστροφος. Αλλάζουμε έτσι τις πρώτες γραμμές του εξωτερικού for loop. Η υλοποίηση λοιπόν για το dgemm_ta είναι :

```
extern __shared__ double partials[];
  // Shared storage for a single tile of A
  double *partials_A = partials;
  // Shared storage for a single tile of B
  double *partials_B = partials_A + SHARED_MEM_DIM * SHARED_MEM_DIM;

  // Get row and column of **output C**
  // (We utilize one thread per output value of C)
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int local_tidx = threadIdx.x;
  int local_tidy = threadIdx.y;

  if (row < M && col < N)
  {
    double temp = 0;
    int i, j;

    // For all "imaginary" blocks (tiles) that are required
    // to traverse dimension K (dimension of dot product)
    // ( & unroll loop for performance)
```

```
#pragma unroll
    for (int current_block = 0; current_block < (K - 1) / SHARED_MEM_DIM + 1; current_block++)
    {

      // Row j of matrix A (sliding offset per tile)
      j = current_block * blockDim.x + threadIdx.x;
      // Row i of matrix B (sliding offset per tile)
      i = current_block * blockDim.y + threadIdx.y;

      // Load A[j][row] to shared mem from global mem
      // row === column of transposed matrix A
      partials_A[local_tidy * SHARED_MEM_DIM + local_tidx] = A[j * M + row];

      // Load B[i][col] to shared mem from global mem
      partials_B[local_tidy * SHARED_MEM_DIM + local_tidx] = B[i * N + col];

      // Synchronize before computation
      // so that shared memory is full !
      __syncthreads();

      // Calculate **partial** dot products ONLY from shared memory !
      // In a single tile, SHARED_MEM_DIM * SHARED_MEM_DIM partial
      // results will be computed.
      // ( & unroll loop for performance)
#pragma unroll
      for (int k = 0; k < SHARED_MEM_DIM; k++)
      {

        // No shared memory bank conflict!
        temp += partials_A[local_tidy * SHARED_MEM_DIM + k] * partials_B[k * SHARED_MEM_DIM + local_tidx];
      }

      // Synchronize before next iteration so that previous
      // values of shared memory are not overwritten BEFORE
      // another thread has not used them yet!
      // (for whatever reason)
      __syncthreads();
    }

    // Now that all tiles in the dimension of dot product have been
    // **accumulated** on 'temp', we can store one result per thread to
    // outpur array.

    // Store result in C
    C[row * N + col] = temp;
  }
```

Και για το dgemm_tb :

```
 extern __shared__ double partials[];
  // Shared storage for a single tile of A
  double *partials_A = partials;
  // Shared storage for a single tile of B
  double *partials_B = partials_A + SHARED_MEM_DIM * SHARED_MEM_DIM;

  // Get row and column of **output C**
  // (We utilize one thread per output value of C)
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int local_tidx = threadIdx.x;
```

```
  int local_tidy = threadIdx.y;

  if (row < M && col < N)
  {
    double temp = 0;
    int i, j;

    // For all "imaginary" blocks (tiles) that are required
    // to traverse dimension K (dimension of dot product)
    // ( & unroll loop for performance)
#pragma unroll
    for (int current_block = 0; current_block < (K - 1) / SHARED_MEM_DIM + 1; current_block++)
    {

      // Column j of matrix A (sliding offset per tile)
      j = current_block * blockDim.x + threadIdx.x;
      // Row i of matrix B (sliding offset per tile)
      i = current_block * blockDim.y + threadIdx.y;

      // Load A[row][j] to shared mem from global mem
      partials_A[local_tidy * SHARED_MEM_DIM + local_tidx] = A[row * K + j];

      // Load B[i][col] to shared mem from global mem
      partials_B[local_tidy * SHARED_MEM_DIM + local_tidx] = B[col * K + i];

      // Synchronize before computation
      // so that shared memory is full !
      __syncthreads();

      // Calculate **partial** dot products ONLY from shared memory !
      // In a single tile, SHARED_MEM_DIM * SHARED_MEM_DIM partial
      // results will be computed.
      // ( & unroll loop for performance)
#pragma unroll
      for (int k = 0; k < SHARED_MEM_DIM; k++)
      {

        // No shared memory bank conflict!
        temp += partials_A[local_tidy * SHARED_MEM_DIM + k] * partials_B[k * SHARED_MEM_DIM + local_tidx];
      }

      // Synchronize before next iteration so that previous
      // values of shared memory are not overwritten BEFORE
      // another thread has not used them yet!
      // (for whatever reason)
      __syncthreads();
    }

    // Now that all tiles in the dimension of dot product have been
    // **accumulated** on 'temp', we can store one result per thread to
    // outpur array.

    // Store result in D
    D[row * N + col] = C[row * N + col] + temp;
  }
```

Για να τρέξουμε τις optimized versions, αλλάζουμε την μεταβλητή GEMM_OPTIMIZED σε 1 στο αρχείο Makefile.gpu, ώστε να γίνεται compile και run με τις βέλτιστες υλοποιήσεις.

2.4 cuBLAS

Εξετάζουμε στη συνέχεια το documentation του cuBLAS, όπως επίσης και την χρήση των έτοιμων συναρτήσεων.

```
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)
```

Figure 2.2: cublasDgemm

Στην βιβλιοθήκη cuBLAS οι πίνακες ειναι αποθηκευμένοι κατά στήλες, ενώ οι πράξεις γίνονται σε ανάστροφους πίνακες. Έτσι πρέπει να φέρουμε τους πίνακες σε κατάλληλη μορφή για να δοθούν σωστά τα ορίσματα στην cublasDgemm.

Για τον απλό πολλαπλασιασμό $A \times B$, η ζητούμενη μορφή είναι η

$$C = A \times B = (A^T)^T \times (B^T)^T = (A')^T \times (B')^T = (B' \times A')^T$$

Έτσι λοιπόν σε αυτή την περίπτωση, τα ορίσματα θα είναι:

```
cublasHandle_t handle;
cublasCreate(&handle);
const double a1 = 1;
const double b1 = 0;
cublasDgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N,
              N, M, K, //m, n, k (because B*A)
              &a1,     //*alpha
              B, N,    //lda, rows of B^T if no trans, cols of B^T otherwise
              A, K,    //ldb
              &b1,     //*beta
              C, N);   //ldc, no transform on C^T
cublasDestroy(handle);
```

Δίνουμε δηλαδή τους πίνακες χωρίς να είναι ανεστραμμένοι, αλλά πρώτα τον B και μετά τον A και με ld τις στήλες των πινάκων.

Στην περίπτωση του dgemm_ta η μετατροπή των εισόδων είναι η εξής :

$$C = A^T \times B = A' \times (B^T)^T = A' \times (B')^T = (B' \times (A')^T)^T$$

και τα ορίσματα είναι τα εξής :

```
cublasHandle_t handle;
  cublasCreate(&handle);
  const double a2 = 1;
  const double b2 = 0;
  cublasDgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T,
              N, M, K, //m, n, k
              &a2,     //*alpha
              B, N,    //lda
              A, M,    //ldb
```

```
10              &b2,     //*beta
11              C, N);   //ldc
12  cublasDestroy(handle);
```

Για την περίπτωση του dgemm_tb αρχικά έχουμε την εξής μετατροπή :

$$D = A \times B^T + C = (A^T)^T \times B^T + C = (A')^T \times B' + C = ((B')^T \times A')^T + C$$

Για το πρώτο όρισμα της πρόσθεσης χρησιμοποιούμε την εξής υλοποίηση :

```
1  cublasHandle_t handle;
2   cublasCreate(&handle);
3   const double a3 = 1;
4   const double b3 = 0;
5   cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
6               N, M, K, //m, n, k
7               &a3,     //*alpha
8               B, K,    //lda
9               A, K,    //ldb
10              &b3,     //*beta
11              D, N);   //ldc
12  cublasDestroy(handle);
```

Και τέλος για να προσθέσουμε τον D στο αποτέλεσμα χρησιμοποιούμε την cublasDgeam, η οποία είναι παρόμοια με την παραπάνω αλλά κάνει πρόσθεση με αντίστοιχους συντελεστές. Έτσι, έχοντας αποθηκεύσει το αποτέλεσμα στον D, προσθέτουμε σε αυτόν τον C.

```
1  cublasHandle_t handle1;
2  cublasCreate(&handle1);
3  const double a4 = 1;
4  const double b4 = 1;
5  cublasDgeam(handle1, CUBLAS_OP_N, CUBLAS_OP_N,
6              M, N,  //m, n
7              &a4,   //*alpha
8              C, M,  //lda
9              &b4,   //*beta
10             D, M,  //ldb
11             D, M); //ldc
12 cublasDestroy(handle1);
```

3. Πειράματα και Αποτελέσματα

3.1 Παραλληλοποίηση σε CPU

Έχοντας τρέξει τις παραπάνω υλοποιήσεις, προκύπτουν οι παρακάτω χρόνοι αναλυτικά ανά μέθοδο και αριθμό νημάτων.

	n_threads	DGEMM	DGEMM_TA	DGEMM_TB	TOTAL
Serial	1	1046.15	1377.70	453.41	**2891.63**
OpenMP	1	664.15	1129.59	414.92	2224.86
	2	342.80	597.26	210.09	1167.09
	4	249.55	311.82	104.26	683.28
	6	242.27	222.67	72.22	554.93
	12	130.32	132.86	45.55	**333.53**
	24	250.90	145.63	41.76	466.15
OpenBLAS	1	67.06	83.58	79.02	243.45
	2	35.51	44.61	42.11	140.87
	4	18.80	24.21	22.84	84.72
	6	16.62	18.58	21.40	75.68
	12	11.46	12.73	15.16	**59.55**
	24	10.62	14.80	14.15	66.48

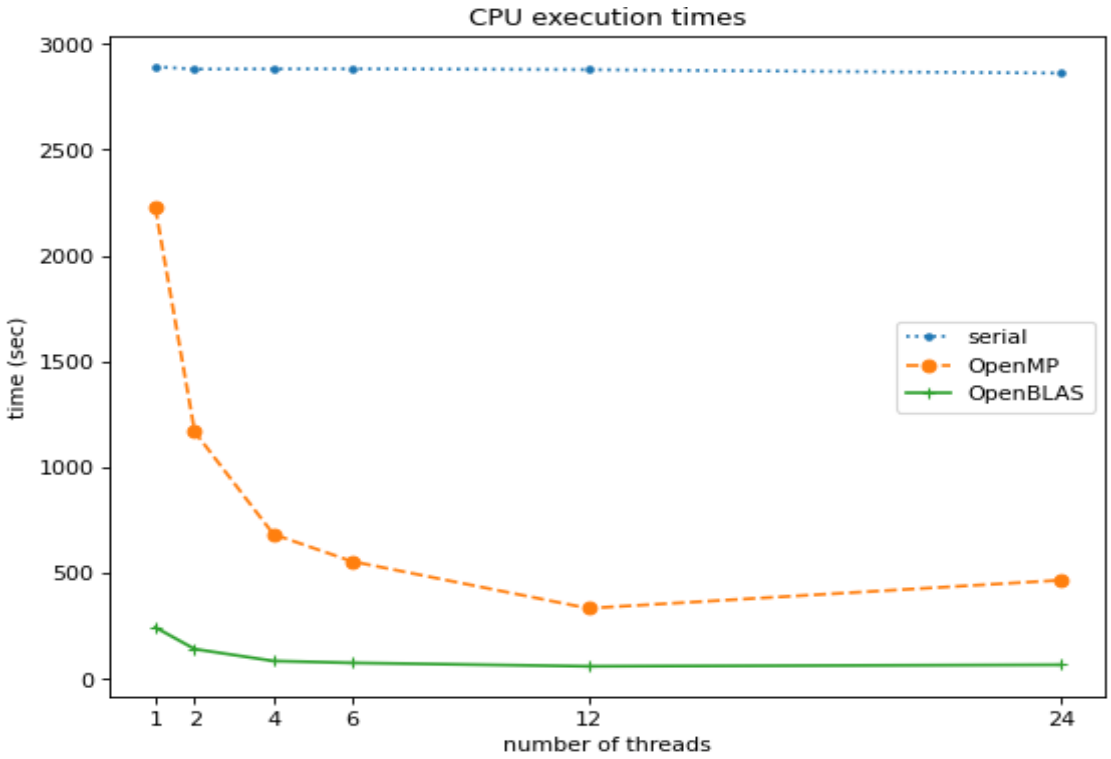

Figure 3.1: CPU χρόνοι εκτέλεσης

Διαιρώντας τους χρόνους της σειριακής υλοποίησης με τους χρόνους της κάθε μεθόδου λαμβάνουμε τα speed ups.

	n_threads	speed-up
Serial	1	1
OpenMP	1	1.299
	2	2.477
	4	4.231
	6	**5.21**
	12	8.669
	24	6.203
OpenBLAS	1	11,877
	2	20.526
	4	34.131
	6	**38.20**
	12	48.55
	24	43.496

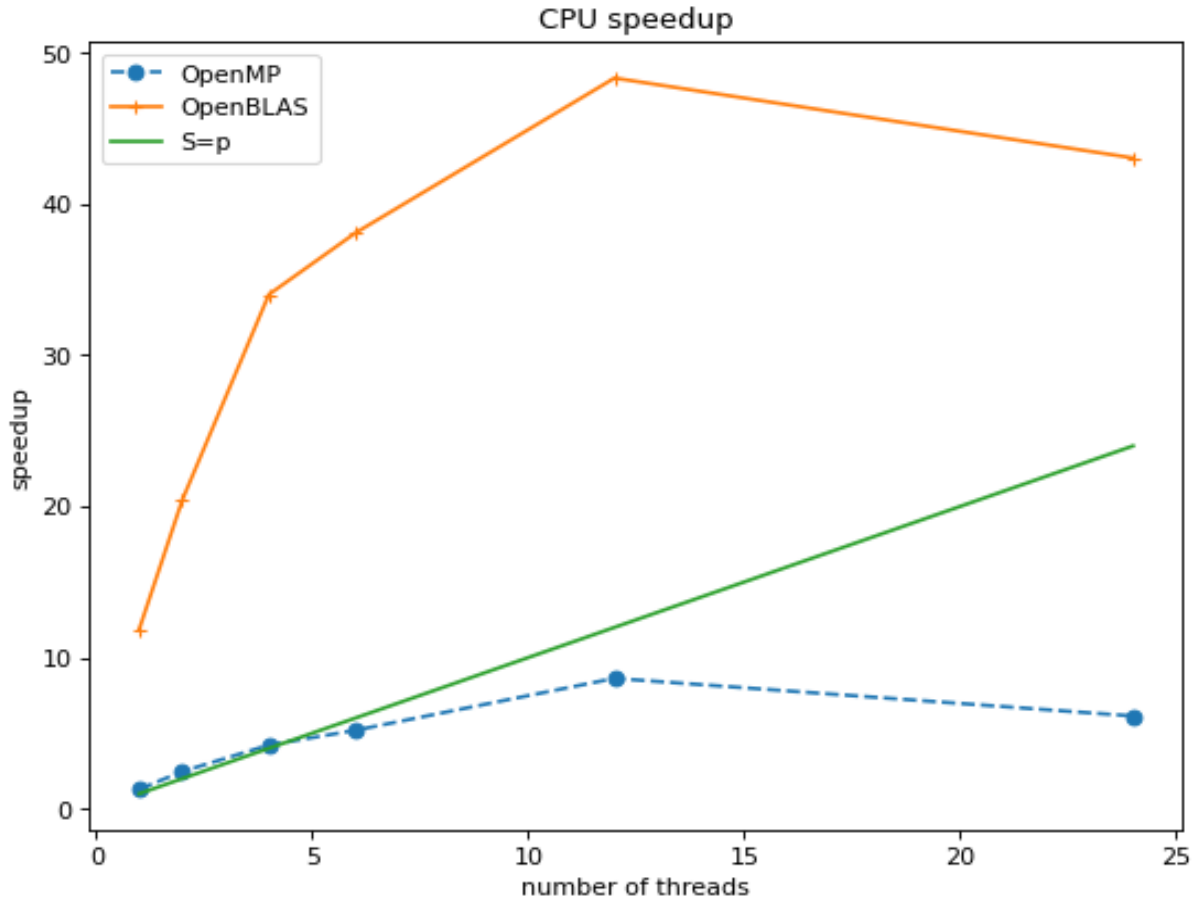

Figure 3.2: CPU speedups

3.2 Παραλληλοποίηση σε GPU

Οι παραπάνω υλοποιήσεις έδωσαν τους εξής χρόνους εκτέλεσης για τα ζητούμενα της εργασίας. Οι κατηγορίες είναι η naive υλοποίηση, η υλοποίηση optimized CUDA και αυτή της βιβλιοθήκης cuBLAS.

	DGEMM	DGEMM_TA	DGEMM_TB	TOTAL
Naive	463.83	625.11	306.96	1406.92
Opt. CUDA	513.09	154.53	57.53	741.56
cuBLAS	512.17	154.64	57.56	740.72

Και το αντίστοιχο SpeedUp που λαμβάνουμε είναι :

- **CUDA** : 0.527
- **cuBLAS** : 0.526

3.3 Σύγκριση CPU - GPU

Αναλυτικότερα τα αποτελέσματα των πινάκων φαίνονται παρακάτω. Οι χρόνοι κάθε σεναρίου παρουσιάζονται ανά στήλη και ανά κατηγορία.

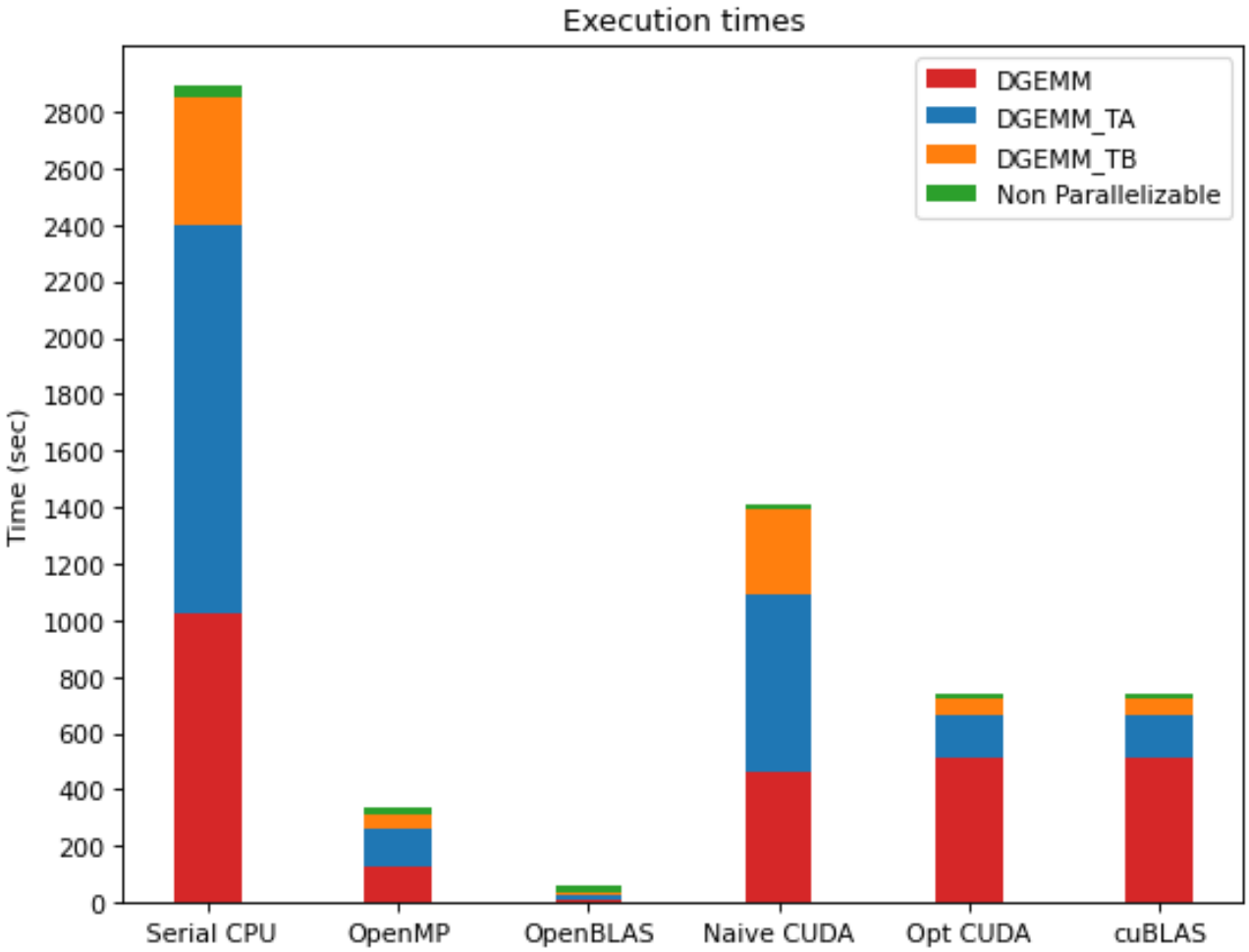

Figure 3.3: Χρόνοι εκτέλεσης

Συνολικά, παρατηρούμε ότι οι βελτιστοποιημένες εκτελέσεις σημειώνουν πολύ καλύτερους χρόνους συγκριτικά με τις αντιστοιχες Serial και Naive, επομένως ο στόχος της άσκησης επετεύχθη.Όσον αφορά τις διαφορετικές πράξεις πινάκων, παρατηρούμε ότι ο DGEMM_TB και ακολούθως ο DGEMM_TA έχουν τους καλύτερους χρόνους εκτέλεσης και στις 4 περιπτώσεις παραλληλοποίησης. Επιπρόσθετα, ο DGEMM φαίνεται να αποδίδει χειρότερα στις

βελτιστοποίησεις με CUDA και cuBLAS.

Τέλος, η βιβλιοθήκη OpenMP και OpenBLAS πετυχαίνουν καλύτερους χρόνους, με την υλοποίηση της έτοιμης βιβλιοθήκης να ξεχωρίζει.