ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δ.Π.Μ.Σ. ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ

Παράλληλες Αρχιτεκτονικές Υπολογισμού για Μηχανική Μάθηση

Εξαμηνιαία Εργασία

Λυδία Ιωάννα Κολίτση, Α.Μ. : 03400252
Μανούσος Λιναρδάκης, Α.Μ. : 03400256
Αντώνιος Μπαροτσάκης, Α.Μ. : 03400260
Γεωργία Χατζηγιάννη, Α.Μ. : 03400280

8 Ιουλίου 2025

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

1 Εισαγωγή

Η παρούσα εργασία εξετάζει την παραλληλοποίηση και βελτιστοποίηση της εκπαίδευσης ενός νευρωνικού δικτύου, εστιάζοντας σε αρχιτεκτονικές κοινής μνήμης (CPU) και επιταχυντές (GPU). Το πρόβλημα που επιλύει το εν λόγω νευρωνικό δίκτυο αφορά την αναγνώριση χειρόγραφων ψηφίων από τη βάση δεδομένων MNIST, χρησιμοποιώντας ένα μοντέλο Multi-Layer Perceptron (MLP) που υλοποιήθηκε από το μηδέν με τη χρήση της βιβλιοθήκης numpy. Ως γνωστόν, το πιο υπολογιστικά "ακριβό" τμήμα του κώδικα, το οποίο καταναλώνει το μεγαλύτερο ποσοστό του χρόνου εκτέλεσης, είναι ο αλγόριθμος πολλαπλασιασμού πινάκων (General Matrix Multiply - GEMM). Συνεπώς, η βελτιστοποίηση του GEMM αποτελεί τον κεντρικό πυρήνα αυτής της εργασίας, με σκοπό την επίτευξη σημαντικής επιτάχυνσης (speedup) στη συνολική διαδικασία της εκπαίδευσης. Η υλοποίηση και η πειραματική αξιολόγηση διαρθρώνονται σε δύο κύρια μέρη, την παραλληλοποίηση σε CPU και σε GPU.

Στο πρώτο μέρος, η βελτιστοποίηση αφορά αρχιτεκτονικές κοινής μνήμης με πολλούς πυρήνες. Χρησιμοποιώντας τη βιβλιοθήκη multiprocessing της Python, εφαρμόστηκε μια στρατηγική παραλληλισμού δεδομένων (data parallelism). Συγκεκριμένα, οι υπολογισμοί για κάθε γραμμή του πίνακα εξόδου ανατίθενται ως ανεξάρτητες εργασίες (tasks) σε ένα σύνολο από διεργασίες (processes), οι οποίες εκτελούνται παράλληλα στους διαθέσιμους πυρήνες της CPU. Η προσέγγιση αυτή αξιοποιεί την ικανότητα των σύγχρονων επεξεργαστών να εκτελούν πολλαπλές διεργασίες ταυτόχρονα, διαμοιράζοντας το υπολογιστικό φορτίο.

Στο δεύτερο μέρος, αξιοποιούνται επιταχυντές (GPU), οι οποίοι είναι εξειδικευμένοι για την εκτέλεση μαζικά παράλληλων υπολογισμών με υψηλή αριθμητική ένταση (high operational intensity), όπως ο GEMM. Για την υλοποίηση χρησιμοποιήθηκε η πλατφόρμα CUDA της NVIDIA, μέσω της βιβλιοθήκης numba της Python. Αναπτύχθηκαν τρεις διαδοχικές εκδόσεις του πυρήνα (kernel) GEMM:

1. Μια απλή υλοποίηση για τη διασφάλιση της ορθότητας.
2. Μια βελτιστοποιημένη εκδοχή που εκμεταλλεύεται την ομαδοποιημένη πρόσβαση στη μνήμη (memory coalescing), για την ελαχιστοποίηση της καθυστέρησης κατά τις μεταφορές δεδομένων από την global μνήμη της GPU.
3. Μια ακόμα πιο βελτιστοποιημένη υλοποίηση που αξιοποιεί την ταχεία, προγραμματιζόμενη shared memory των Streaming Multiprocessors (SMs) για την επαναχρησιμοποίηση δεδομένων, η οποία μειώνει δραστικά την ανάγκη για προσβάσεις στην αργή global μνήμη.

Τέλος, οι βελτιστοποιημένοι πυρήνες ενσωματώθηκαν στη συνολική ροή εκπαίδευσης του νευρωνικού δικτύου για να αξιολογηθεί η τελική επιτάχυνση και να σχολιαστεί η κλιμακωσιμότητα (scalability) των λύσεων. Η ανάλυση των αποτελεσμάτων βασίζεται σε μετρικές επίδοσης και θεωρητικές αρχές, όπως ο νόμος του Amdahl και το μοντέλο Roofline, για την ερμηνεία των παρατηρούμενων επιταχύνσεων και των περιοριστικών παραγόντων κάθε αρχιτεκτονικής.

2 Βελτιστοποίηση GEMM

2.1 Παραλληλοποίηση σε CPU

Στο πρώτο μέρος της εργασίας, υλοποιήσαμε μια παράλληλη εκδοχή του αλγορίθμου πολλαπλασιασμού πινάκων (GEMM) για συστήματα κοινής μνήμης, αξιοποιώντας τη βιβλιοθήκη `multiprocessing` της `Python`.

```python
from copy import deepcopy
from multiprocessing import Process, shared_memory, cpu_count
from Multiprocessing.tools import ReleaseSharedMemory, CreateSharedNumpyArray
import numpy as np

def MultiprocessingMatMul(A, B, numberOfProcesses=1):

    def MultiprocessingTargetFunction(A, B, C, iStartIndex, iEndIndex):
        numCols = B.shape[1]
        innerDim = B.shape[0]

        # FILL ME: Perform GEMM for the rows assigned to each process
        # Iterate over the subset of rows assigned to this process
        for i in range(iStartIndex, iEndIndex):
            # Iterate over the columns of the output matrix
            for j in range(numCols):
                # Perform the dot product for the element C[i, j]
                sum_val = 0
                for k in range(innerDim):
                    sum_val += A[i, k] * B[k, j]
                C[i, j] = sum_val
        return

    outArrayShape = (A.shape[0], B.shape[1])

    # Create the shared memory block for the output array C
    CreateSharedNumpyArray(outArrayShape, arrayName='C')
    shm = shared_memory.SharedMemory(name='C')
    C = np.ndarray(outArrayShape, dtype=np.float64, buffer=shm.buf)

    # Initialize the output array to zeros
    C.fill(0)

    # Calculate the number of rows per process
    numRowsPerProcess = A.shape[0] // numberOfProcesses # FILL ME

    # Calculate the remaining rows to be handled by the last process
    lastProcessExtraRows = A.shape[0] % numberOfProcesses # FILL ME: The last process might
    get extra rows, if there are any left

    processes = []
    for i in range(numberOfProcesses):
        # Calculate the start and end row index for each process
        iStartIndex = i * numRowsPerProcess # FILL ME
        iEndIndex = iStartIndex + numRowsPerProcess # FILL ME

        # The last process takes any remaining rows
        if i == numberOfProcesses - 1:
            iEndIndex += lastProcessExtraRows

        # Create a process targeting our function
```

```
        iProcess = Process(target=MultiprocessingTargetFunction,
                           args=(A, B, C, iStartIndex, iEndIndex,))

        processes.append(iProcess)
        iProcess.start()

    # Wait for all processes to finish their execution
    for iProcess in processes:
        iProcess.join()

    # Deepcopy the result from shared memory to a new array
    outC = deepcopy(C)

    # Release the shared memory block
    ReleaseSharedMemory(['C'])

    return outC, 0
```

Η κεντρική ιδέα είναι η κατανομή των γραμμών του πίνακα εξόδου C σε πολλαπλές, ανεξάρτητες διεργασίες. Κάθε διεργασία αναλαμβάνει την ευθύνη για τον υπολογισμό ενός διακριτού "κομματιού" (chunk) γραμμών, επιτρέποντας την ταυτόχρονη εκτέλεσή τους σε διαφορετικούς πυρήνες της CPU.

Αρχικά για να αποφύγουμε τη δαπανηρή αντιγραφή των μεγάλων πινάκων κατά τη δημιουργία κάθε νέας διεργασίας, ο πίνακας εξόδου C δεσμεύεται σε ένα κοινόχρηστο τμήμα μνήμης (shared memory). Η κύρια διεργασία είναι υπεύθυνη για τη δημιουργία αυτού του κοινού χώρου πριν την εκκίνηση των υπολοίπων. Στη συνέχεια, κάθε θυγατρική διεργασία, αντί να δημιουργήσει έναν νέο πίνακα, απλώς συνδέεται σε αυτόν τον προϋπάρχοντα χώρο. Αυτό διασφαλίζει ότι όλες οι διεργασίες εργάζονται πάνω στα ίδια δεδομένα, γράφοντας τα αποτελέσματά τους σε έναν μοναδικό, κεντρικό πίνακα.

Η συνάρτηση-στόχος `MultiprocessingTargetFunction` είναι ο κώδικας που εκτελείται ταυτόχρονα από όλες τις θυγατρικές διεργασίες. Κάθε διεργασία όμως τον εκτελεί με τα δικά της διαφορετικά ορίσματα (iStartIndex, iEndIndex), τα οποία οριοθετούν το ακριβές τμήμα του πίνακα για το οποίο είναι υπεύθυνη.

- Ο εξωτερικός βρόχος `for i in range(iStartIndex, iEndIndex)` κάνει loop μόνο στις γραμμές που της έχουν ανατεθεί. Αυτή ακριβώς είναι η παραλληλοποίηση, κάθε διεργασία δουλεύει στο δικό της, ξεχωριστό κομμάτι.
- Στον μεσαίο βρόχο `for j in range(numCols)`, η διεργασία υπολογίζει για κάθε γραμμή i που έχει αναλάβει, όλα τα στοιχεία αυτής της γραμμής στον πίνακα C. Δηλαδή, αυτή η loop διατρέχει όλες τις στήλες του τελικού πίνακα, από 0 έως numCols-1.
- Ο εσωτερικός βρόχος `for k in range(innerDim)` εκτελεί τον υπολογισμό του εσωτερικού γινομένου (dot product). Για να βρει το στοιχείο $C[i, j]$, πολλαπλασιάζει το κάθε στοιχείο της γραμμής i του πίνακα A με το αντίστοιχο στοιχείο της στήλης j του πίνακα B και τα αθροίζει.

Τέλος, αφού η κύρια διεργασία εκκινήσει όλες τις θυγατρικές, χρησιμοποιεί την εντολή `join()` ως μηχανισμό συγχρονισμού. Η εντολή αυτή την αναγκάζει να αναμένει την ολοκλήρωση όλων των παράλληλων υπολογισμών. Μόνο όταν και η τελευταία θυγατρική διεργασία τελειώσει τη δουλειά της, η κύρια διεργασία συνεχίζει, διασφαλίζοντας ότι το αποτέλεσμα στον κοινόχρηστο πίνακα C είναι πλήρες και έτοιμο για να επιστραφεί.

2.2 Παραλληλοποίηση σε GPU

2.2.1 CudaSimpleMatMul.py

Για την απλή παραλληλοποίηση του GEMM σε GPU, η υλοποίηση ακολουθεί την τυπική ροή ενός προγράμματος CUDA, το οποίο βασίζεται στην αλληλεπίδραση μεταξύ του κεντρικού επεξεργαστή (host) και του επιταχυντή (device). Η στρατηγική παραλληλοποίησης που επιλέχθηκε είναι η αντιστοίχιση ενός νήματος (thread) για τον υπολογισμό ενός στοιχείου του πίνακα εξόδου C. Για να επιτευχθεί αυτό με φυσικό τρόπο για τον δισδιάστατο πίνακα, τα νήματα οργανώνονται σε ένα δισδιάστατο πλέγμα (2D grid), όπου κάθε νήμα λαμβάνει μοναδικές συντεταγμένες (row, col) που αντιστοιχούν απευθείας στο στοιχείο $C[row, col]$ που καλείται να υπολογίσει.

Η διαδικασία ξεκινά με τη μεταφορά των δεδομένων εισόδου στη μνήμη της συσκευής. Οι πίνακες A και B, που αρχικά βρίσκονται στη μνήμη της CPU, αντιγράφονται στην global memory της GPU με τις κλήσεις `cuda.to_device()`. Αυτό το βήμα είναι θεμελιώδες, καθώς η GPU λειτουργεί με το δικό της, ξεχωριστό χώρο μνήμης. Ο χρόνος που απαιτείται για αυτές τις μεταφορές καταγράφεται, διότι αποτελεί ένα σημαντικό, μη υπολογιστικό κόστος (overhead) που επηρεάζει τη συνολική απόδοση, ειδικά σε προβλήματα όπου η μεταφορά δεδομένων είναι χρονοβόρα σε σχέση με τον υπολογισμό.

Στη συνέχεια, ρυθμίζεται η γεωμετρία εκτέλεσης του πυρήνα. Αρχικά, ορίζονται οι διαστάσεις του μπλοκ νημάτων (thread block) σε (32, 32). Ένα μπλοκ είναι μια ομάδα νημάτων που εκτελούνται από κοινού σε έναν Streaming Multiprocessor (SM). Η επιλογή αυτή δημιουργεί μπλοκ των 1024 νημάτων. Δεδομένου ότι το hardware της GPU οργανώνει τα νήματα σε ομάδες των 32, που ονομάζονται warps, η επιλογή διαστάσεων που είναι πολλαπλάσια του 32 (όπως το (32, 32)) εξασφαλίζει την πλήρη και αποδοτική χρήση των πόρων του SM. Έπειτα, υπολογίζεται ο αριθμός των μπλοκ που απαιτούνται για να καλυφθεί ολόκληρος ο πίνακας εξόδου, ορίζοντας τις διαστάσεις του πλέγματος των μπλοκ (grid of blocks). Ο υπολογισμός γίνεται με τη χρήση στρογγυλοποίησης προς τα πάνω (`np.ceil`) της διαίρεσης των διαστάσεων του πίνακα με τις αντίστοιχες διαστάσεις του μπλοκ. Αυτό εγγυάται ότι θα δημιουργηθούν αρκετά νήματα για να καλύψουν πλήρως τον πίνακα, ακόμα και όταν οι διαστάσεις του δεν είναι ακριβή πολλαπλάσια του 32.

Αφού ρυθμίστηκε η γεωμετρία, ακολουθεί η εκκίνηση του πυρήνα `SimpleMatMul2DKernel[...](...)`, η οποία είναι ασύγχρονη. Μέσα στον κώδικα του πυρήνα, κάθε νήμα προσδιορίζει τη μοναδική του θέση στο πλέγμα καλώντας τη συνάρτηση cuda.grid(2), η οποία του επιστρέφει τις global συντεταγμένες (row, col). Αμέσως μετά, εκτελείται ένας έλεγχος ορίων (`if row < C.shape[0] and col < C.shape[1]:`), ο οποίος είναι απαραίτητος για να διασφαλιστεί ότι μόνο τα νήματα που αντιστοιχούν σε πραγματικά στοιχεία του πίνακα C θα προχωρήσουν σε υπολογισμούς. Αυτό αποτρέπει σφάλματα προσπέλασης μνήμης (out-of-bounds access) από τα "περισσευούμενα" νήματα που δημιουργήθηκαν λόγω της στρογγυλοποίησης. Τα νήματα που περνούν τον έλεγχο, εκτελούν τον υπολογισμό του εσωτερικού γινομένου για το στοιχείο $C[row, col]$, διαβάζοντας δεδομένα από τους πίνακες A και B της καθολικής μνήμης και γράφοντας το αποτέλεσμα στον πίνακα C, που επίσης βρίσκεται στην global memory.

Τέλος, μετά την ασύγχρονη εκκίνηση του πυρήνα, ο κώδικας της CPU φτάνει στην εντολή `C = C_global_mem.copy_to_host()`. Αυτή η κλήση είναι κρίσιμης σημασίας, καθώς λειτουργεί ως έμμεσο σημείο συγχρονισμού. Η αντιγραφή των δεδομένων από τη συσκευή (GPU) πίσω στον κεντρικό επεξεργαστή (CPU) δεν μπορεί να ξεκινήσει αν οι υπολογισμοί του πυρήνα δεν έχουν ολοκληρωθεί. Επομένως, η CPU θα "περιμένει" σε αυτό το σημείο μέχρι η GPU να τελειώσει τη δουλειά της, διασφαλίζοντας ότι τα δεδομένα που αντιγράφονται πίσω στον πίνακα C είναι τα τελικά και ορθά αποτελέσματα του πολλαπλασιασμού. Ο χρόνος που απαιτείται για αυτή τη μεταφορά αποτελεί επίσης μέρος του συνολικού overhead της διαδικασίας.

```
1 from numba import cuda
2 import numpy as np
3 from time import perf_counter
```

```
from os import environ

environ["CUDA_VISIBLE_DEVICES"] = "1"

@cuda.jit("void(float64[:,:], float64[:,:],  float64[:,:])")
def SimpleMatMul2DKernel(A, B, C):
    # 2D thread grid
    # get thread coords from cuda.grid
    # (returns absolute position of the current thread in the entire grid of blocks)
    row, col = cuda.grid(2)

    # FILL ME: The code that thread (row,col) will execute (calculating C[row][col])
    # Boundary check: Ensure the thread is within the bounds of the output matrix C.
    # This is important when matrix dimensions are not a multiple of the block dimensions.
    if row < C.shape[0] and col < C.shape[1]:
        # Perform the dot product for C[row, col]
        tmp_sum = 0.0
        # Iterate over the inner dimension (columns of A or rows of B)
        for k in range(A.shape[1]):
            tmp_sum += A[row, k] * B[k, col]
        C[row, col] = tmp_sum

    return


def CudaSimpleMatMul(A, B):

    numRows = A.shape[0]
    numCols = B.shape[1]

    outArrayShape = (numRows, numCols)

    tic = perf_counter()
    # FILL ME : Initialize & transfer A and B to device memory (A_global_mem, B_global_mem)
    A_global_mem = cuda.to_device(A)
    B_global_mem = cuda.to_device(B)
    toc = perf_counter()

    # Allocate device memory for the output matrix C
    C_global_mem = cuda.device_array(outArrayShape, dtype=np.float64)

    transfersTime = toc-tic

    # FILL ME : Create a grid of thread blocks and fire the SimpleMatMul2DKernel kernel. You
    should use a 2D grid (of 2D blocks)
    threadGridDim = 32
    threadsPerBlock = (threadGridDim, threadGridDim)

    # Calculate the number of blocks needed in each dimension.
    # We use ceiling division to ensure we have enough blocks to cover the entire matrix.

    blocksPerGrid_x = int(np.ceil(numRows / threadsPerBlock[0])) # FILL ME
    blocksPerGrid_y = int(np.ceil(numCols / threadsPerBlock[1])) # FILL ME
    blocksPerGrid = (blocksPerGrid_x, blocksPerGrid_y)

    # FILL ME : Invoke kernel #
    # Launch the kernel with the specified grid and block configuration.
    SimpleMatMul2DKernel[blocksPerGrid, threadsPerBlock](A_global_mem, B_global_mem,
    C_global_mem)
```

```
61
62      # Copy the result from device memory back to host memory
63      C = C_global_mem.copy_to_host()
64
65      return C, transfersTime
```

2.2.2 CudaTransposeMatMul.py

Η βελτιστοποίηση του πολλαπλασιασμού πινάκων (GEMM) σε αρχιτεκτονικές GPU αποτελεί ένα κλασικό παράδειγμα του πώς η στρατηγική αντιστοίχισης της εργασίας στα threads μπορεί να επιφέρει δραματικές βελτιώσεις στην απόδοση. Η αρχική, απλή υλοποίηση, αν και λειτουργικά ορθή, αποκαλύπτει σημαντικές αδυναμίες που πηγάζουν από τη μη-βέλτιστη εκμετάλλευση της ιεραρχίας της μνήμης.

Σε αυτή την πρώτη προσέγγιση, το πλέγμα (grid) των threads αντιστοιχίστηκε ώστε η "γρήγορη" διάσταση x να διατρέχει τις γραμμές (row) του πίνακα εξόδου. Δεδομένου ότι οι GPUs εκτελούν εντολές σε ομάδες των 32 threads (warps), αυτή η διάταξη οδηγεί σε warps που αποτελούνται από threads με διαδοχικές τιμές row και την ίδια τιμή col. Κατά τον υπολογισμό $C[row, col] = \sum_k (A[row, k] \cdot B[k, col])$, το πρότυπο πρόσβασης στη μνήμη (memory access pattern) γίνεται ο καθοριστικός παράγοντας της απόδοσης. Για τον πίνακα A, τα threads ενός warp αναζητούν στοιχεία από διαφορετικές γραμμές ($A[row, k]$, $A[row + 1, k]$, ...), τα οποία βρίσκονται σε απομακρυσμένες θέσεις στη μνήμη. Αυτή η κατακερματισμένη πρόσβαση (strided access) είναι εξαιρετικά αναποτελεσματική και υποβαθμίζει το ωφέλιμο εύρος ζώνης της μνήμης. Παρόλο που η πρόσβαση στον πίνακα B είναι ιδανική, καθώς όλα τα threads ζητούν το ίδιο στοιχείο $B[k, col]$ (broadcast), η συνολική απόδοση καθορίζεται από τον πιο αργό κρίκο: την προβληματική πρόσβαση στον A.

Η λύση σε αυτό το αδιέξοδο βρίσκεται στην αναδιάταξη του grid. Αντιστρέψαμε την αντιστοίχιση, ώστε η "γρήγορη" διάσταση x να αντιστοιχεί πλέον στις στήλες (col). Αυτή η φαινομενικά μικρή αλλαγή μετασχηματίζει ριζικά τη συμπεριφορά του πυρήνα. Πλέον, ένα warp αποτελείται από threads που μοιράζονται την ίδια τιμή row και έχουν διαδοχικές τιμές col.

Αυτή η νέα δομή επιλύει ταυτόχρονα και τα δύο προβλήματα μνήμης. Η πρόσβαση στον πίνακα A μετατρέπεται σε μια αποδοτική λειτουργία broadcast, καθώς όλα τα threads του warp αναζητούν το ίδιο στοιχείο $A[row, k]$. Ταυτόχρονα, η πρόσβαση στον πίνακα B γίνεται ιδανική. Τα threads αναζητούν τα στοιχεία $B[k, col]$, $B[k, col + 1]$, ..., τα οποία, λόγω της row-major διάταξης, είναι αποθηκευμένα σε συνεχόμενες θέσεις στη μνήμη. Αυτό επιτρέπει στο hardware να εκτελέσει μια τέλεια συνενωμένη πρόσβαση (coalesced memory access), όπου τα δεδομένα για ολόκληρο το warp ανακτώνται με μία μόνο, ευρεία συναλλαγή μνήμης.

Συμπερασματικά, η μετάβαση από την απλή στην βελτιστοποιημένη υλοποίηση αποδεικνύει ότι η κατανόηση της αρχιτεκτονικής της GPU και η σωστή χαρτογράφηση του προβλήματος στις παράλληλες μονάδες της είναι θεμελιώδους σημασίας. Μετατρέποντας τα πρότυπα πρόσβασης στη μνήμη σε ιδανικά (broadcast και coalesced), επιτυγχάνουμε μια σημαντική επιτάχυνση, αναδεικνύοντας πως στον παράλληλο προγραμματισμό, ο τρόπος διαχείρισης των δεδομένων είναι εξίσου, αν όχι περισσότερο, κρίσιμος από τους ίδιους τους υπολογισμούς.

```
1 from numba import cuda
2 import numpy as np
3 from time import perf_counter
4 from os import environ
5
6 environ["CUDA_VISIBLE_DEVICES"] = "1"
7
8 @cuda.jit("void(float64[:,:], float64[:,:],  float64[:,:])")
9 def TransposeMatMul2DKernel(A, B, C):
```

```
    # 2D thread grid
    # get thread coords from cuda.grid
    # (returns absolute position of the current thread in the entire grid of blocks)

    # FILL ME: Use an inverse/transposed grid for threads
    # The grid's x-dimension is mapped to columns, and y-dimension to rows.
    # This is the opposite of the simple version, and is the conventional mapping.
    # It improves memory coalescing.
    col, row = cuda.grid(2)

    # FILL ME: The code that thread (col,row) will execute --> THE SAME with CudaSimpleMatMul
    # Boundary check to ensure the thread is within the matrix dimensions.
    if row < C.shape[0] and col < C.shape[1]:
        # Perform the dot product for C[row, col]
        tmp_sum = 0.0
        # Iterate over the inner dimension
        for k in range(A.shape[1]):
            tmp_sum += A[row, k] * B[k, col]
        C[row, col] = tmp_sum

    return

def CudaTransposeMatMul(A, B):

    # FILL ME : Use the code from CudaSimpleMatMul. ONLY adjustment: blocksPerGrid_[x, y]
    should be modified from a (row,column) to a (column,row) grid

    numRows = A.shape[0]
    numCols = B.shape[1]

    outArrayShape = (numRows, numCols)

    tic = perf_counter()
    # FILL ME : Initialize & transfer A and B to device memory (A_global_mem, B_global_mem)
    A_global_mem = cuda.to_device(A)
    B_global_mem = cuda.to_device(B)
    toc = perf_counter()

    C_global_mem = cuda.device_array(outArrayShape, dtype=np.float64)

    transfersTime = toc-tic

    # FILL ME : Create a grid of thread blocks and fire the SimpleMatMul2DKernel kernel. You
    should use a 2D grid (of 2D blocks)
    threadGridDim = 32
    threadsPerBlock = (threadGridDim, threadGridDim)

    blocksPerGrid_x = int(np.ceil(numCols / threadsPerBlock[0])) # FILL ME : Transpose
    blocksPerGrid_y = int(np.ceil(numRows / threadsPerBlock[1])) # FILL ME : Transpose
    blocksPerGrid = (blocksPerGrid_x, blocksPerGrid_y)

    # FILL ME : Invoke kernel
    TransposeMatMul2DKernel[blocksPerGrid, threadsPerBlock](A_global_mem, B_global_mem,
    C_global_mem)

    C = C_global_mem.copy_to_host()

    return C, transfersTime
```

2.2.3 CudaSharedMemMatMul.py

Αυτή η υλοποίηση βασίζεται στην τεχνική του tiling (ή block-based matrix multiplication). Ο στόχος της είναι να αντιμετωπίσει το θεμελιώδες "bottleneck" της απόδοσης: την υψηλή καθυστέρηση (latency) της global memory. Η στρατηγική βασίζεται στην αρχή της επαναχρησιμοποίησης δεδομένων (data reuse), ελαχιστοποιώντας τις προσβάσεις στην αργή global memory, αξιοποιώντας την εξαιρετικά γρήγορη, προγραμματιζόμενη κοινόχρηστη μνήμη (shared memory) που είναι διαθέσιμη σε κάθε Streaming Multiprocessor (SM).

Η λογική είναι η εξής: αντί κάθε νήμα να διαβάζει όλα τα δεδομένα που χρειάζεται απευθείας από την global memory, ο πολλαπλασιασμός "σπάει" σε μικρότερους πολλαπλασιασμούς υπο-πινάκων (tiles). Κάθε μπλοκ νημάτων αναλαμβάνει συλλογικά να:

- Φορτώσει ένα tile του πίνακα A και ένα του B από την αργή global memory στη γρήγορη shared memory.
- Εκτελέσει τον πολλαπλασιασμό αυτών των δύο tiles, διαβάζοντας δεδομένα αποκλειστικά από τη shared memory.
- Επαναλάβει τη διαδικασία για όλα τα απαραίτητα ζεύγη tiles, αθροίζοντας τα επιμέρους αποτελέσματα.

Με αυτόν τον τρόπο, κάθε στοιχείο των A και B που φορτώνεται στη shared memory, χρησιμοποιείται πολλαπλές φορές (συγκεκριμένα, `threadGridDim`=32 φορές) από τα νήματα του μπλοκ, μειώνοντας δραστικά τον συνολικό αριθμό προσβάσεων στην καθολική μνήμη. Αναλυτικότερα, ο κώδικας του kernel βασίζεται στις παρακάτω λειτουργίες:

1. Δήλωση και Φόρτωση Shared Memory: Μέσα στον εξωτερικό βρόχο (`for i in range(k_chunks)`), κάθε νήμα του μπλοκ συνεργάζεται για να φορτώσει ένα tile 32x32 του A και ένα του B στις κοινόχρηστες μεταβλητές `sharedMemA` και `sharedMemB`. Κάθε νήμα είναι υπεύθυνο για τη φόρτωση ενός στοιχείου για κάθε πίνακα. Οι προσβάσεις στην καθολική μνήμη ($A[row, global\_A\_col]$ και $B[global\_B\_row, col]$) είναι σχεδιασμένες ώστε να είναι όσο το δυνατόν πιο ομαδοποιημένες (coalesced), μεγιστοποιώντας την αποδοτικότητα της φόρτωσης. Οι έλεγχοι ορίων διασφαλίζουν ότι δεν γίνονται προσβάσεις εκτός των ορίων των πινάκων, γεμίζοντας με μηδενικά όπου χρειάζεται.

2. Συγχρονισμός (`cuda.syncthreads()`): Η κλήση αυτή είναι απολύτως κρίσιμη. Η πρώτη κλήση γίνεται αμέσως μετά τη φόρτωση των tiles και λειτουργεί ως φράγμα (barrier), διασφαλίζοντας ότι όλα τα νήματα του μπλοκ έχουν ολοκληρώσει τη φόρτωση των δεδομένων τους στη shared memory πριν ξεκινήσει ο υπολογισμός. Η δεύτερη κλήση γίνεται στο τέλος του βρόχου, εγγυώμενη ότι όλοι οι υπολογισμοί με το τρέχον tile έχουν ολοκληρωθεί πριν το μπλοκ προχωρήσει για να φορτώσει το επόμενο.

3. Υπολογισμός με Shared Memory: Μεταξύ των δύο κλήσεων συγχρονισμού, κάθε νήμα εκτελεί τον πολλαπλασιασμό των tiles. Ο εσωτερικός βρόχος (`for j in range(threadGridDim)`) διατρέχει τα στοιχεία των tiles που βρίσκονται πλέον στην ταχύτατη shared memory. Κάθε νήμα αθροίζει τα επιμέρους γινόμενα σε μια τοπική μεταβλητή (`tempSum`), η οποία συνήθως αποθηκεύεται σε έναν ακόμη πιο γρήγορο καταχωρητή (register).

4. Εγγραφή Αποτελέσματος: Αφού ολοκληρωθούν όλοι οι κύκλοι (`k_chunks`), το τελικό άθροισμα που έχει υπολογιστεί στο `tempSum` γράφεται μία μόνο φορά από κάθε νήμα στην αντίστοιχη θέση του πίνακα C στην καθολική μνήμη.

Έτσι, η στρατηγική του tiling μετατρέπει ένα πρόβλημα που περιορίζεται από το εύρος ζώνης της μνήμης (memory-bound) σε ένα που περιορίζεται από την υπολογιστική ισχύ (compute-bound), επιτρέποντας στην GPU να επιτύχει πολύ υψηλότερη απόδοση.

```
from numba import cuda, float64
import numpy as np
from time import perf_counter
from os import environ
import math # Added for math.ceil

environ["CUDA_VISIBLE_DEVICES"] = "1"

threadGridDim = 32
threadsPerBlock = (threadGridDim, threadGridDim)

@cuda.jit("void(float64[:,:], float64[:,:],  float64[:,:])")
def SharedMemMatMul2DKernel(A, B, C):
    # 2D thread grid
    # get thread coords from cuda.grid
    # (returns absolute position of the current thread in the entire grid of blocks)
    col, row = cuda.grid(2)

    # Create the shared memory arrays whose shape is dictated by the thread dimensions of each
     block
    sharedMemA = cuda.shared.array((threadGridDim, threadGridDim), dtype=float64)
    sharedMemB = cuda.shared.array((threadGridDim, threadGridDim), dtype=float64)

    if row >= C.shape[0] or col >= C.shape[1]:
        return

    # Indexes on the current thread block will be used for array indexing
    threadX = cuda.threadIdx.x
    threadY = cuda.threadIdx.y
    k_chunks = int(B.shape[0] / threadGridDim)
    if B.shape[0] % threadGridDim != 0:
        k_chunks+=1

    # each thread will compute one element of C
    # to do that it will use the shared memory for
    # computing and storing the sum of the subproducts (dot product)
    tempSum = 0.0
    for i in range(k_chunks):

        # FILL ME: each thread fills an element of sharedMemA and one of sharedMemB. Note:
    beware of the boundaries of the K dimension
        # Calculate global indices for this tile
        global_A_row = row
        global_A_col = i * threadGridDim + threadX

        global_B_row = i * threadGridDim + threadY
        global_B_col = col

        # Load from global memory to shared memory with boundary checks
        if global_A_row < A.shape[0] and global_A_col < A.shape[1]:
            sharedMemA[threadY, threadX] = A[global_A_row, global_A_col]
        else:
            sharedMemA[threadY, threadX] = 0.0

        if global_B_row < B.shape[0] and global_B_col < B.shape[1]:
            sharedMemB[threadY, threadX] = B[global_B_row, global_B_col]
        else:
            sharedMemB[threadY, threadX] = 0.0

```

```
58
59          cuda.syncthreads()
60
61          # FILL ME: Each thread updates its tempSum using the elements of sharedMemA and
        sharedMemB. Note: beware of the boundaries of the K dimension
62          # Multiply tiles from shared memory
63          for j in range(threadGridDim):
64              tempSum += sharedMemA[threadY, j] * sharedMemB[j, threadX]
65
66          cuda.syncthreads()
67      C[row, col] = tempSum
68
69
70  def CudaSharedMemMatMul(A, B):
71
72      # FILL ME : Use the code from CudaTransposeMatMul.#
73      # This part remains identical to previous GPU implementations.
74
75      numRows = A.shape[0]
76      numCols = B.shape[1]
77
78      outArrayShape = (numRows, numCols)
79
80      tic = perf_counter()
81      # FILL ME : Initialize & transfer A and B to device memory (A_global_mem, B_global_mem) #
82      A_global_mem = cuda.to_device(A)
83      B_global_mem = cuda.to_device(B)
84      toc = perf_counter()
85
86      C_global_mem = cuda.device_array(outArrayShape, dtype=np.float64)
87
88      transfersTime = toc-tic
89
90      # FILL ME : Create a grid of thread blocks and fire the SimpleMatMul2DKernel kernel. You
        should use a 2D grid (of 2D blocks) #
91      # The global threadGridDim is already defined as 32 at the top of the file.
92      threadsPerBlock = (threadGridDim, threadGridDim)
93
94      # In the provided kernel, (col, row) = cuda.grid(2), so x-dim maps to columns and y-dim to
         rows.
95      blocksPerGrid_x = int(np.ceil(numCols / threadsPerBlock[0])) # FILL ME : Transpose
96      blocksPerGrid_y = int(np.ceil(numRows / threadsPerBlock[1])) # FILL ME : Transpose
97      blocksPerGrid = (blocksPerGrid_x, blocksPerGrid_y)
98
99      # FILL ME : Invoke kernel #
100     SharedMemMatMul2DKernel[blocksPerGrid, threadsPerBlock](A_global_mem, B_global_mem,
        C_global_mem)
101
102     C = C_global_mem.copy_to_host()
103
104     return C, transfersTime
```

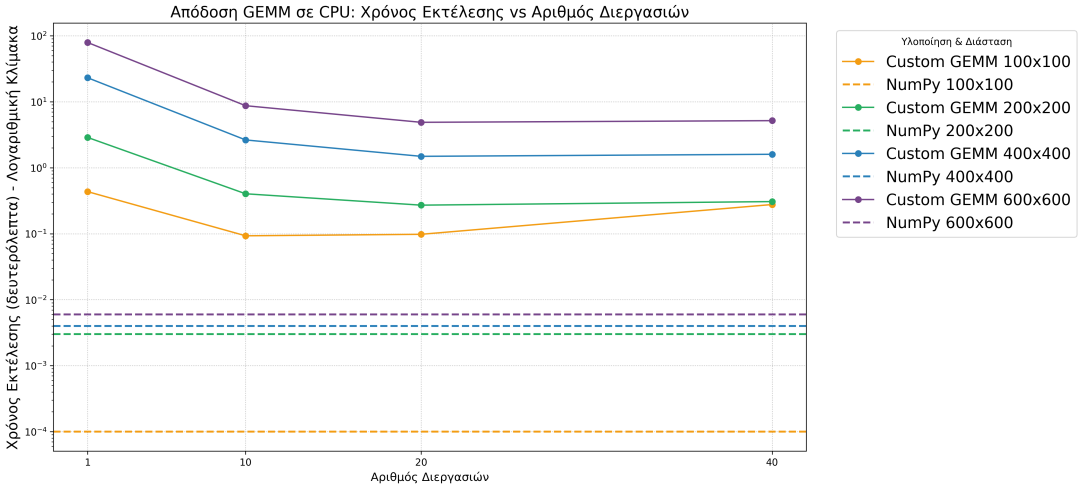
3 Βελτιστοποίηση Νευρωνικού δικτύου

3.1 Σενάριο μετρήσεων και διαγράμματα

3.1.1 Κλιμακωσιμότητα σε CPU

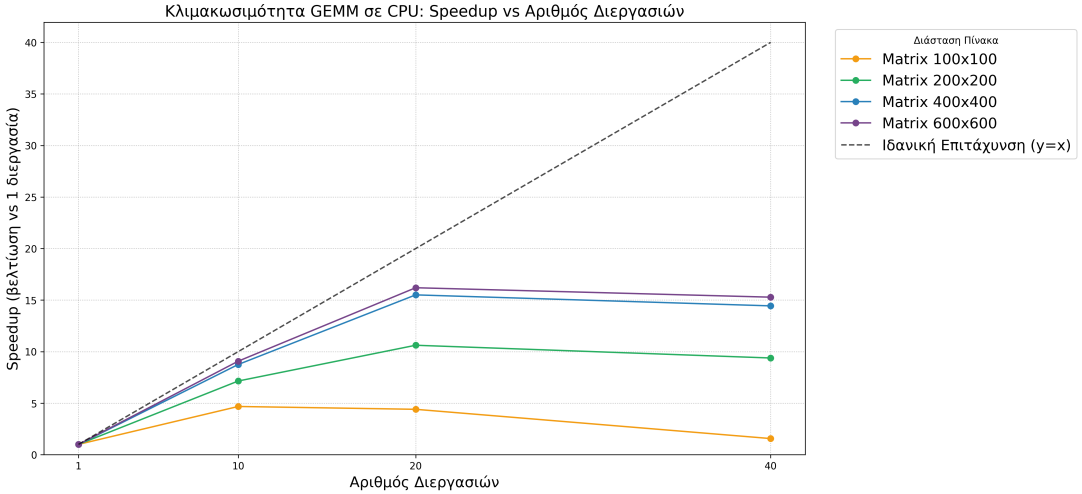
Στο Σχήμα 1, παρατηρούμε ότι ο χρόνος εκτέλεσης της παράλληλης υλοποίησης μειώνεται σημαντικά καθώς αυξάνεται ο αριθμός των διεργασιών, επιβεβαιώνοντας την επιτυχία της παραλληλοποίησης. Ωστόσο, η υλοποίηση `numpy.dot`, η οποία χρησιμοποιεί εσωτερικά βελτιστοποιημένες, προ-μεταγλωττισμένες βιβλιοθήκες, παραμένει τάξεις μεγέθους ταχύτερη.

Σχήμα 1: Απόδοση GEMM σε CPU: Συγκριτικός Χρόνος Εκτέλεσης της παράλληλης υλοποίησης και της numpy.dot για αυξανόμενο αριθμό διεργασιών.



Στο Σχήμα 2, αναλύεται η κλιμακωσιμότητα (scalability). Η καμπύλη του speedup αρχικά αυξάνεται σημαντικά, αλλά στη συνέχεια αποκλίνει ολοένα και περισσότερο, μέχρι που αλλάζει, από αύξουσα σε φθίνουσα. Αυτή η συμπεριφορά είναι αναμενόμενη και εξηγείται από τον νόμο του Amdahl: καθώς προσθέτουμε περισσότερους πυρήνες, το σειριακό τμήμα του προγράμματος (δημιουργία διεργασιών, συγχρονισμός) γίνεται το κυρίαρχο bottleneck, περιορίζοντας την περαιτέρω επιτάχυνση. Επιπλέον, για μεγάλο αριθμό διεργασιών, το κόστος της εναλλαγής περιβάλλοντος (context switching) από το λειτουργικό σύστημα γίνεται σημαντικό.

Σχήμα 2: Κλιμακωσιμότητα GEMM σε CPU: Η επιτάχυνση (speedup) της παράλληλης υλοποίησης σε σχέση με την ιδανική γραμμική επιτάχυνση (y=x).

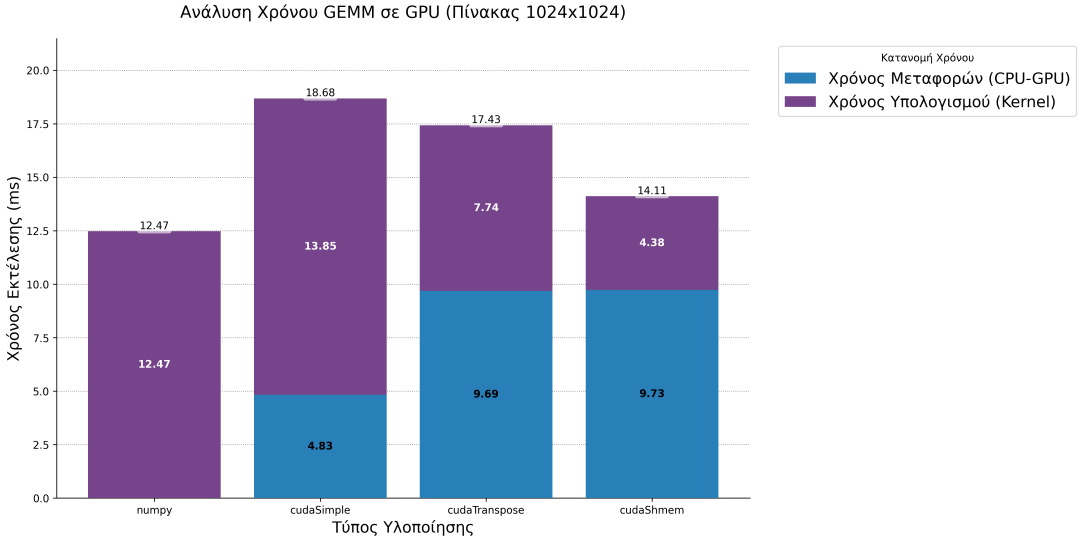


3.1.2 Σύγκριση επιδόσεων με GPU

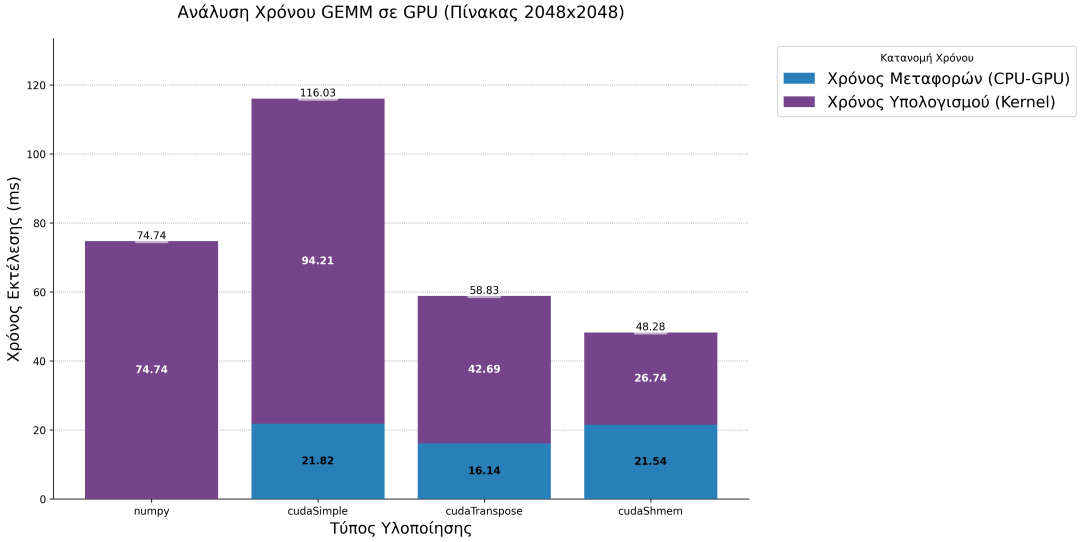
Τα Σχήματα 3,4,5 και 6 παρουσιάζουν την απόδοση των τριών υλοποιήσεων GPU για αυξανόμενο μέγεθος προβλήματος. Όσον αφορά το κόστος των μεταφορών, παρατηρείται ότι στην πρώτη υλοποίηση (1024x1024), ο χρόνος μεταφοράς δεδομένων (μπλε μπάρα) αποτελεί ένα σημαντικό, αν όχι το μεγαλύτερο, μέρος του συνολικού χρόνου. Αυτό αναδεικνύει το "memory wall", όπου η απόδοση περιορίζεται από τη μεταφορά δεδομένων και όχι από τον υπολογισμό.

Η αποτελεσματικότητα των βελτιστοποιήσεων είναι εμφανής, καθότι η μετάβαση από την `CudaSimple` στην `CudaTranspose` και έπειτα στην `CudaShmem` οδηγεί σε δραματική μείωση του χρόνου υπολογισμού (μωβ μπάρα). Η υλοποίηση `CudaShmem` με shared memory είναι σταθερά η ταχύτερη, επιβεβαιώνοντας ότι η επαναχρησιμοποίηση δεδομένων στη γρήγορη shared memory είναι η πιο αποδοτική στρατηγική. Ο λόγος που η υλοποίηση `CudaTranspose` ειναι πιο αποδοτική από την `CudaSimple` αναλύεται στην ενότητα 2.2.2.

Σχήμα 3: Ανάλυση Χρόνου GEMM σε GPU (1024x1024): Σύγκριση απόδοσης και κατανομή χρόνου (μεταφορές vs υπολογισμός) για τις υλοποιήσεις GPU.

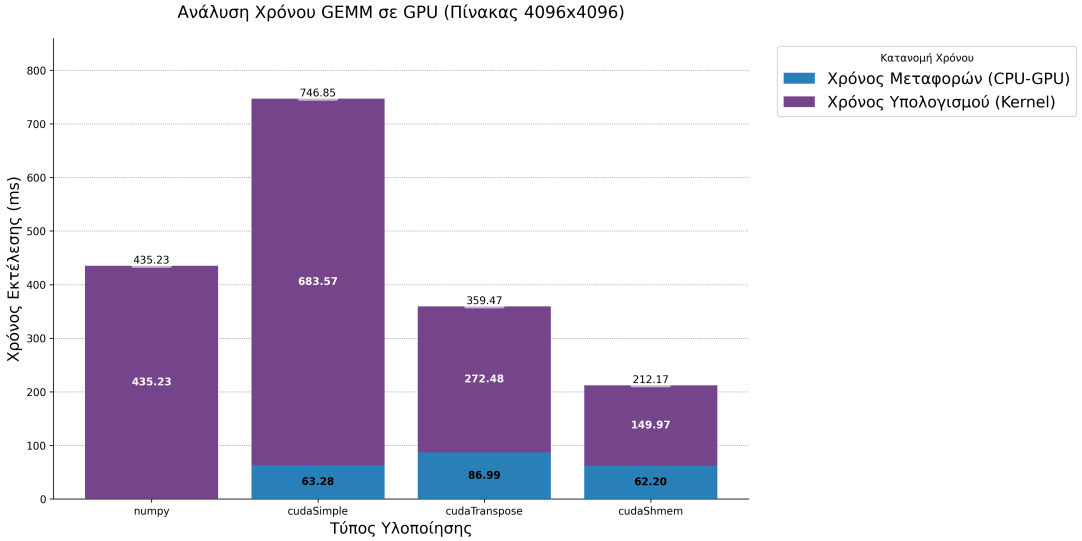


Σχήμα 4: Ανάλυση Χρόνου GEMM σε GPU (2048x2048): Σύγκριση απόδοσης και κατανομή χρόνου (μεταφορές vs υπολογισμός) για τις υλοποιήσεις GPU.

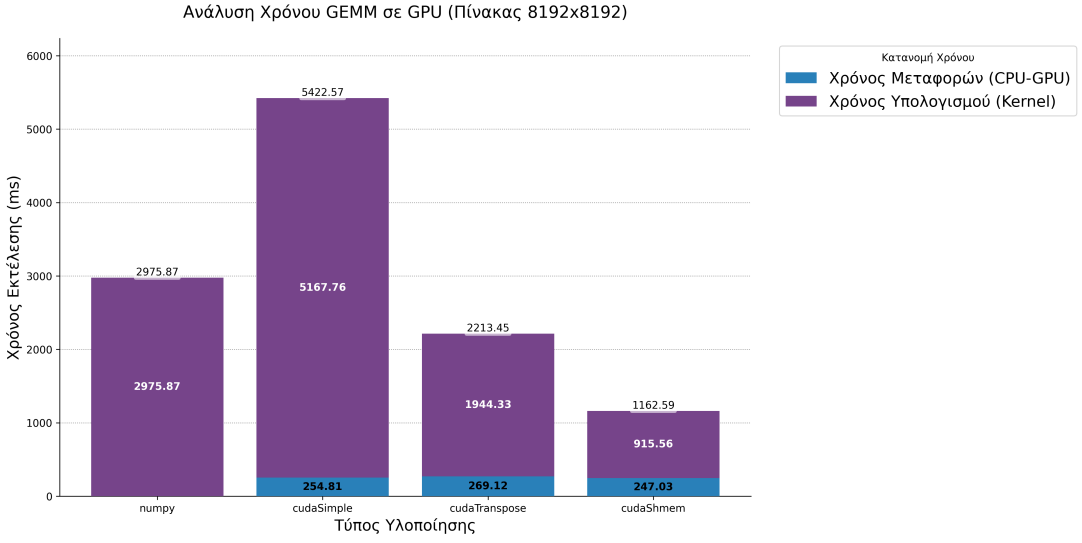


Επιπλέον, καθώς το μέγεθος του πίνακα αυξάνεται, το σχετικό όφελος των βελτιστοποιημένων υλοποιήσεων γίνεται ακόμα μεγαλύτερο. Για μεγάλους πίνακες, ο χρόνος υπολογισμού κυριαρχεί, και η αποδοτικότητα του πυρήνα γίνεται ο καθοριστικός παράγοντας. Η `CudaShmem` επιδεικνύει την καλύτερη κλιμάκωση, καθώς το υπολογιστικό της φορτίο αυξάνεται με ρυθμό $O(N^3)$, ενώ οι προσβάσεις στην global memory μειώνονται σημαντικά.

Σχήμα 5: Ανάλυση Χρόνου GEMM σε GPU (4096x4096): Σύγκριση απόδοσης και κατανομή χρόνου (μεταφορές vs υπολογισμός) για τις υλοποιήσεις GPU.



Σχήμα 6: Ανάλυση Χρόνου GEMM σε GPU (8192x8192): Σύγκριση απόδοσης και κατανομή χρόνου (μεταφορές vs υπολογισμός) για τις υλοποιήσεις GPU.



Για να ποσοτικοποιήσουμε το τελικό όφελος των βελτιστοποιήσεών μας, συγκεντρώσαμε την επιτάχυνση (speedup) κάθε υλοποίησης GPU σε άμεση σύγκριση με την βελτιστοποιημένη `NumPy` που εκτελείται στην CPU (Πίνακας 1).

Παρατηρούμε ότι η `cudaSimple` είναι σταθερά πιο αργή από τη `NumPy`. Αυτή η υλοποίηση θεωρείται "αφελής" (naive) διότι, παρόλο που χρησιμοποιεί χιλιάδες threads, η στρατηγική αντιστοίχισης οδηγεί σε αναποτελεσματικά πρότυπα πρόσβασης στη μνήμη (strided memory access), όπως αναλύθηκε στην ενότητα 2.2.2, ακυρώνοντας έτσι το θεωρητικό όφελος από τον μαζικό παραλληλισμό.

Αντιθέτως, οι βελτιστοποιημένες εκδόσεις `cudaTranspose` και `cudaShmem` καταφέρνουν να ξεπεράσουν την απόδοση της `NumPy` καθώς το μέγεθος του προβλήματος αυξάνεται. Η `cudaTranspose` προσφέρει μια αξιοσημείωτη επιτάχυνση της τάξης του 1.2x-1.3x για μεγάλους πίνακες. Ωστόσο, η υλοποίηση με shared memory (`cudaShmem`) είναι ο αδιαμφισβήτητος νικητής, επιδεικνύοντας μια σαφή τάση κλιμάκωσης: από οριακά πιο αργή για πίνακες 1024x1024, φτάνει να είναι 2.56 φορές ταχύτερη από τη `NumPy` για το μεγαλύτερο πρόβλημα (8192x8192).

Αυτό αποδεικνύει ότι για μεγάλα, compute-bound προβλήματα, η συνδυασμένη στρατηγική της σωστής αντιστοίχισης του grid και της εκμετάλλευσης της ταχείας on-chip μνήμης (shared memory) είναι ο μόνος τρόπος για να ξεκλειδωθεί η πλήρης υπολογιστική ισχύς της GPU και να επιτευχθεί ουσιαστική επιτάχυνση έναντι των σύγχρονων, βελτιστοποιημένων βιβλιοθηκών CPU.

Πίνακας 1: Επιτάχυνση (Speedup) των υλοποιήσεων GPU σε σχέση με τη NumPy

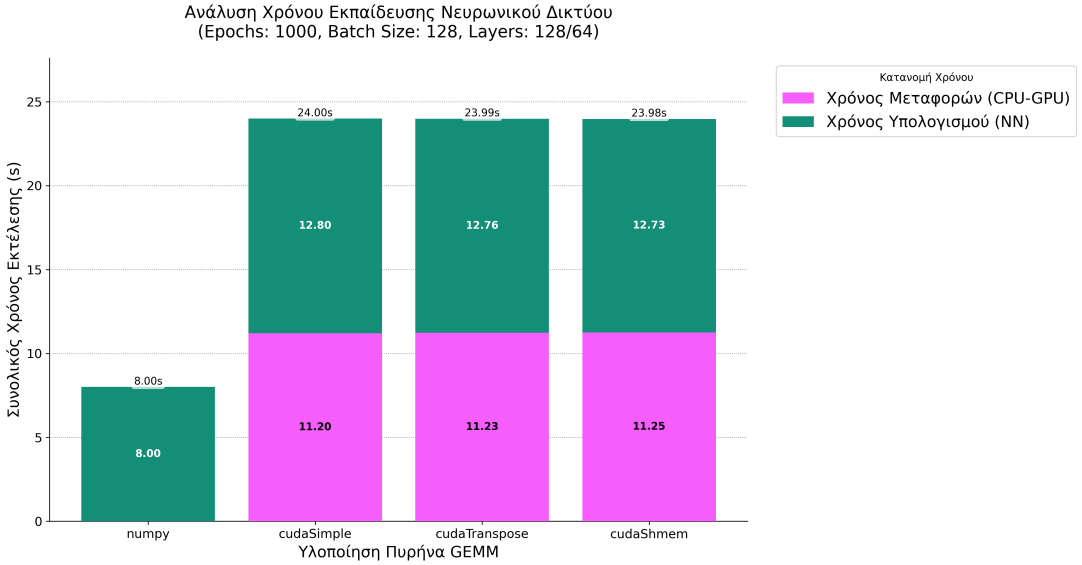
	Διάσταση Πίνακα (N x N)			
Υλοποίηση	**1024**	**2048**	**4096**	**8192**
`cudaSimple`	0.67x	0.64x	0.58x	0.55x
`cudaTranspose`	0.72x	1.27x	1.21x	1.34x
`cudaShmem`	0.88x	1.55x	2.05x	2.56x

4 Εκπαίδευση Νευρωνικού δικτύου

Στο τελευταίο μέρος της εργασίας, ενσωματώσαμε τους βελτιστοποιημένους πυρήνες GEMM στη ροή εκπαίδευσης ενός νευρωνικού δικτύου, για να αξιολογήσουμε την τελική επίδοση σε δύο διαφορετικά σενάρια.

Στο Σχήμα 7, εξετάζεται το σενάριο "validation", με πολλές εποχές (1000) και μικρό μέγεθος batch (128). Σε αυτή την περίπτωση, εκτελούνται πολλοί, μικροί πολλαπλασιασμοί πινάκων. Παρατηρούμε ότι το κόστος μεταφοράς δεδομένων (ροζ μπάρες) είναι πολύ σημαντικό σε σχέση με τον συνολικό χρόνο εκτέλεσης για όλες τις υλοποιήσεις GPU. Όπως φαίνεται αναλυτικά στον Πίνακα 2, το ποσοστό του χρόνου που αναλώνεται στις μεταφορές (Transfer/Total Ratio) αγγίζει το 46% για τις GPU υλοποιήσεις. Ο πυρήνας εκτελείται τόσες πολλές φορές που το αθροιστικό κόστος της μεταφοράς των πινάκων A και B στη GPU σε κάθε κλήση γίνεται το κύριο bottleneck. Αυτό εξηγεί γιατί καμία από τις GPU υλοποιήσεις δεν προσφέρει επιτάχυνση σε σχέση με τη `NumPy` σε αυτό το memory-bound σενάριο.

Σχήμα 7: Ανάλυση Χρόνου Εκπαίδευσης NN (Validation Config): Σύγκριση συνολικού χρόνου εκτέλεσης για το σενάριο με πολλές εποχές και μικρά batches.

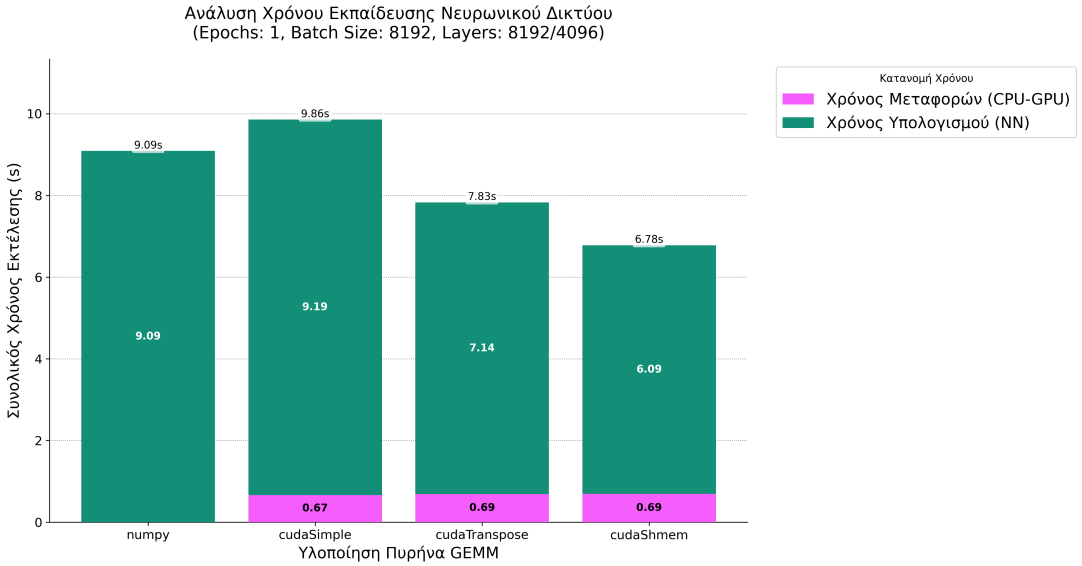


Πίνακας 2: Συγκριτική Απόδοση Εκπαίδευσης NN (Validation Config)

Υλοποίηση	NN Test Accuracy	Transfer/Total Ratio	Speedup to NumPy
`NumPy`	77.5%	0.0	1.0
`cudaSimple`	77.5%	0.467	0.333
`cudaTranspose`	77.5%	0.468	0.334
`cudaShmem`	77.8%	0.469	0.334

Αντίθετα, στο Σχήμα 8, εξετάζεται το σενάριο "high-throughput", με μία μόνο εποχή και πολύ μεγάλο μέγεθος batch (8192). Εδώ, εκτελούνται λίγοι αλλά πολύ μεγάλοι πολλαπλασιασμοί πινάκων. Η εικόνα αντιστρέφεται πλήρως: ο χρόνος υπολογισμού (πράσινες μπάρες) κυριαρχεί, και το κόστος μεταφοράς είναι σχετικά μικρό. Το πρόβλημα γίνεται compute-bound, με το Transfer/Total Ratio να πέφτει κάτω από 11% σε όλες τις GPU υλοποιήσεις (Πίνακας 3). Σε αυτό το σενάριο, η ανωτερότητα του πυρήνα `CudaShmem` είναι εμφανής, καθώς επιτυγχάνει τη μικρότερη συνολική χρονική διάρκεια, προσφέροντας τη μεγαλύτερη επιτάχυνση.

Σχήμα 8: Ανάλυση Χρόνου Εκπαίδευσης NN (High-Throughput Config): Σύγκριση συνολικού χρόνου εκτέλεσης για το σενάριο με μία εποχή και μεγάλα batches.



Πίνακας 3: Συγκριτική Απόδοση Εκπαίδευσης NN (High-Throughput Config)

Υλοποίηση	**NN Test Accuracy**	**Transfer/Total Ratio**	**Speedup to NumPy**
NumPy	11.4%	0.0	1.0
cudaSimple	11.4%	0.068	0.922
cudaTranspose	11.4%	0.088	1.161
cudaShmem	11.4%	0.102	1.341

Συμπερασματικά, η επιλογή της κατάλληλης υλοποίησης εξαρτάται άμεσα από τη φύση του προβλήματος. Για προβλήματα που απαιτούν την εκτέλεση πολλών μικρών πυρήνων, η απόδοση περιορίζεται από το latency και το κόστος μεταφοράς (memory-bound). Για προβλήματα που περιλαμβάνουν μεγάλους, συνεχόμενους υπολογισμούς, η απόδοση καθορίζεται από την υπολογιστική ισχύ του πυρήνα (compute-bound), όπου οι προηγμένες τεχνικές βελτιστοποίησης, όπως το tiling με shared memory, αποδίδουν τα μέγιστα.

5 Συμπεράσματα

Η βελτιστοποίηση του πυρήνα GEMM ανέδειξε τις θεμελιώδεις διαφορές μεταξύ των αρχιτεκτονικών CPU και GPU. Στη CPU, η παραλληλοποίηση μέσω multiprocessing επέτρεψε την αξιοποίηση πολλαπλών πυρήνων, προσφέροντας σημαντική επιτάχυνση που όμως περιορίζεται από το overhead της δημιουργίας και διαχείρισης των διεργασιών, όπως προβλέπει ο νόμος του Amdahl.

Στη GPU, η μετάβαση από την απλή υλοποίηση στην βελτιστοποιημένη με χρήση shared memory κατέδειξε τη σημασία της ιεραρχίας μνήμης. Η τεχνική του tiling μετέτρεψε έναν αλγόριθμο που περιοριζόταν από το εύρος ζώνης της μνήμης (memory-bound) σε έναν που αξιοποιεί την τεράστια υπολογιστική ισχύ της GPU (compute-bound), επιτυγχάνοντας σημαντικά υψηλότερη απόδοση. Η ανάλυση επιβεβαίωσε ότι η κατανόηση της αρχιτεκτονικής, και συγκεκριμένα της ομαδοποιημένης πρόσβασης (memory coalescing) και της επαναχρησιμοποίησης δεδομένων, είναι κρίσιμη για την επίτευξη υψηλής απόδοσης σε επιταχυντές.

Η ενσωμάτωση αυτών των πυρήνων στην εκπαίδευση του νευρωνικού δικτύου κατέδειξε ότι η τελική επίδοση εξαρτάται από τη φύση του φόρτου εργασίας. Σε σενάρια με πολλές κλήσεις μικρών πυρήνων, το κόστος μεταφοράς δεδομένων μεταξύ CPU και GPU κυριαρχεί, καθιστώντας το πρόβλημα memory-bound. Αντίθετα, σε σενάρια με λίγες κλήσεις μεγάλων πυρήνων, ο χρόνος υπολογισμού γίνεται ο καθοριστικός παράγοντας, και οι προηγμένες τεχνικές βελτιστοποίησης, όπως το tiling, αποδίδουν τα μέγιστα.

Συνολικά, η εργασία επιβεβαίωσε ότι ενώ η παραλληλοποίηση σε CPU είναι πιο προσιτή, η αρχιτεκτονική της GPU, όταν αξιοποιείται σωστά, προσφέρει τάξεις μεγέθους υψηλότερη απόδοση για υπολογιστικά έντονους και δομημένους αλγορίθμους όπως ο GEMM. Η κατανόηση της αρχιτεκτονικής του hardware δεν είναι απλώς μια λεπτομέρεια, αλλά ο ακρογωνιαίος λίθος για την επίτευξη υψηλής απόδοσης στον παράλληλο προγραμματισμό.