ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ
ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ
ΥΠΟΛΟΓΙΣΤΩΝ

# ΥΠΟΛΟΓΙΣΤΙΚΕΣ ΕΦΑΡΜΟΓΕΣ ΜΕ ΕΠΙΤΑΧΥΝΣΗ GPU ΜΕ PYTHON

## Project για το μάθημα 1ου εξαμήνου
## ΕΙΣΑΓΩΓΗ ΣΤΟΥΣ ΥΠΟΛΟΓΙΣΤΕΣ

**Εργασία των φοιτητών της ΟΜΑΔΑΣ 50**

**ΓΙΩΡΓΟΣ ΞΑΓΟΡΑΚΗΣ - ΑΜ 1092863**

**ΝΑΤΑΛΙΑ ΡΟΥΣΚΑ - ΑΜ 1092581**

**ΙΓΝΑΤΙΟΣ ΤΟΥΒΛΕΛΙΟΣ – ΑΜ 1092865**

**Υπεύθυνος Καθηγητής**
**ΒΑΣΙΛΕΙΟΣ ΠΑΛΙΟΥΡΑΣ**

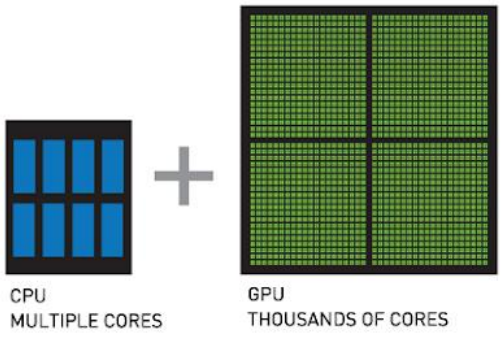
**Ρίο**
**15 Ιανουαρίου 2022**

# 1 Εισαγωγή

Στην παρούσα εργασία θα δείξουμε, μέσω μιας υπολογιστικής εφαρμογής που εκτελεί πληθώρα πολλαπλασιασμών αριθμητικών δεδομένων τύπου float32 σε τετραγωνικό πίνακα μεγάλης διάστασης, πώς αυξάνει το όφελος από τη χρήση της GPU καθώς αυξάνουμε το φορτίο υπολογισμών σε ένα μεγάλο σύνολο δεδομένων, επιταχύνοντας έτσι την εκτέλεση προγραμμάτων σε σχέση με την CPU.

# 2 Εκτέλεση προγραμμάτων σε CPU και GPU

Η Μονάδα Επεξεργασίας Γραφικών (Graphic Processing Unit – GPU) έχει συνήθως πολύ περισσότερους πυρήνες από την Κεντρική Μονάδα Επεξεργασίας (Central Processing Unit – CPU). Ενώ στο παρελθόν οι GPUs σχεδιάζονταν αποκλειστικά για γραφικά υπολογισμών, πλέον χρησιμοποιούνται εκτενώς και για υπολογισμούς γενικού σκοπού. Επιπλέον των γραφικών, οι παράλληλοι υπολογισμοί καθοδηγούμενοι από τις GPUs, χρησιμοποιούνται ευρέως στη μηχανική μάθηση, την βαθιά μάθηση, την τεχνητή νοημοσύνη, την αναλυτική δεδομένων και σε άλλες εργασίες που μπορούν να εκμεταλλευτούν την παράλληλη επεξεργασία [1]. Επομένως, όταν χρησιμοποιούνται για παράλληλη και επίπονη επεξεργασία δεδομένων, οι GPUs συμπεριφέρονται πιο αποτελεσματικά από τις CPUs όσον αφορά στους χρόνους εκτέλεσης των προγραμμάτων, αν και οι GPUs έχουν μικρότερη ταχύτητα ρολογιού σε σχέση με τις CPUs. Στην Εικόνα 1 απεικονίζονται γραφικά οι πυρήνες των CPUs και GPUs [2].



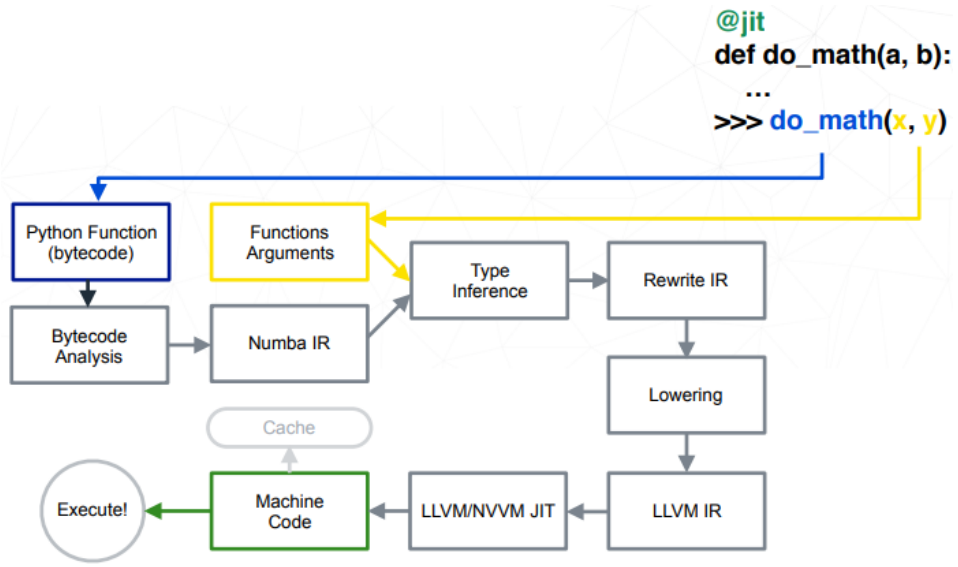

*Εικόνα 1: Πυρήνες Κεντρικής Μονάδας Επεξεργασίας και Μονάδας Επεξεργασίας Γραφικών*

Θα ανέμενε κανείς ότι η εκτέλεση ενός προγράμματος, όπως τα python scripts που αναπτύξαμε στα πλαίσια του μαθήματος, στη GPU θα γίνεται πιο γρήγορα από ότι στην CPU. Κάτι τέτοιο όμως δεν είναι αληθές, καθώς η εκτέλεση στην GPU προϋποθέτει πρώτα τη μεταφορά των δεδομένων στη μνήμη της GPU, στη συνέχεια την εκτέλεση των υπολογισμών και τέλος την επιστροφή του αποτελέσματος. Αν λοιπόν το σύνολο των δεδομένων είναι μικρό ή το υπολογιστικό φορτίο είναι μικρό, δεν γίνεται εκμετάλλευση των δυνατοτήτων της GPU και το αποτέλεσμα είναι η CPU να επιτυγχάνει ταχύτερη εκτέλεση του προγράμματος από την GPU. Το όφελος εμφανίζεται όταν το σύνολο δεδομένων που θα υποστεί επεξεργασία είναι σχετικά μεγάλο και ακόμα περισσότερο όταν το φορτίο επεξεργασίας των δεδομένων είναι αρκετά μεγάλο και μπορεί να εκτελεστεί παράλληλα.

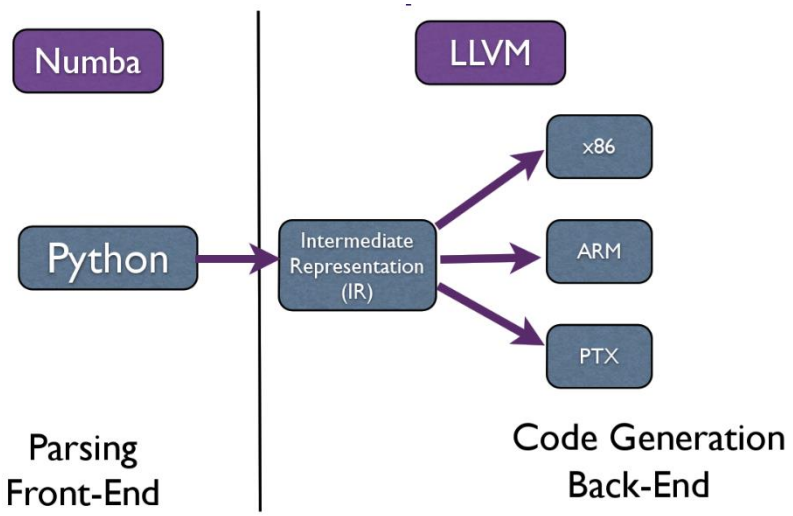
## 3 Numba: A High Performance Python Compiler

Το Numba είναι ένας μεταγλωττιστής ανοικτού κώδικα που μεταγλωττίζει ένα υποσύνολο κώδικα της Python και της NumPy σε ταχύ εκτελέσιμο κώδικα μηχανής [3]. Το Numba μεταφράζει τις συναρτήσεις Python σε βελτιστοποιημένο κώδικα μηχανής κατά τον χρόνο εκτέλεσης χρησιμοποιώντας την τυποποιημένη βιβλιοθήκη μεταγλώττισης LLVM. Οι αλγόριθμοι που χρησιμοποιούν αριθμητικούς υπολογισμούς και έχουν μεταφραστεί με το Numba μπορούν να πλησιάσουν σε ταχύτητα εκτέλεσης την ταχύτητα της γλώσσας προγραμματισμού C και FORTRAN.

Στις παρακάτω δύο εικόνες φαίνονται η ακολουθία δημιουργίας του βελτιστοποιημένου κώδικα μηχανής από τον κώδικα Python και ο διαχωρισμός σε frontend και backend στο Numba.




*Εικόνα 2: Ακολουθία δημιουργίας του βελτιστοποιημένου κώδικα στο Numba*




*Εικόνα 3: Διαχωρισμός Numba σε frontend και backend*

Ο προγραμματιστής χρησιμοποιεί το περιβάλλον της Python, δεν χρειάζεται να αντικαταστήσει τον μεταφραστή Python, ούτε απαιτείται να τρέξει κάποιο επιπλέον βήμα μεταγλώττισης. Απλά χρησιμοποιεί έναν από τους Numba decorators στην συνάρτηση Python που έχει αναπτύξει και ο μεταγλωττιστής Numba κάνει όλη την υπόλοιπη δουλειά.

Ο μεταγλωττιστής Numba είναι σχεδιασμένος να χρησιμοποιείται με τις συναρτήσεις και πίνακες NumPy και δημιουργεί κάποιου τύπου ειδικό κώδικα για διαφορετικούς τύπους δεδομένων πινάκων (array data types) για επίτευξη βέλτιστων επιδόσεων.

# 4 NVIDIA GPUs – Cuda

Οι GPUs της εταιρίας NVIDIA είναι ευρέως διαδεδομένο υλικό που χρησιμοποιείται από πολλούς κατασκευαστές ηλεκτρονικών υπολογιστών, συνδέονται μέσω του PCIe bus και έχουν δική τους μνήμη και πάρα πολλούς πυρήνες οι οποίοι μπορούν να λειτουργούν παράλληλα στην ίδια συνάρτηση λογισμικού.

Η CUDA (Compute Unified Device Architecture), είναι μια παράλληλη πλατφόρμα υπολογισμού και ένα μοντέλο προγραμματισμού που επιτρέπει μεγάλη αύξηση στην επίδοση υπολογισμών εκμεταλλευόμενη τις Nvidia GPUs. Η CUDA έχει αναπτυχθεί από την NVIDIA από το 2006 και χρησιμοποιείται σε εκατομμύρια υπολογιστές που τρέχουν εφαρμογές για περιοχές όπως αστρονομία, βιολογία, χημεία, φυσική, ανάκτηση δεδομένων, τεχνική μάθηση που απαιτούν μεγάλη υπολογιστική δύναμη [*7*] CUDA, https://en.wikipedia.org/wiki/CUDA[7].

Η CUDA Python της Nvidia παρέχει μια προγραμματιστική διεπαφή εφαρμογών (application programming interface – API) για υπάρχουσες βιβλιοθήκες και εργαλειοθήκες, ώστε να διευκολύνει την ανάπτυξη εφαρμογών που χρησιμοποιούν την επιτάχυνση που προσφέρουν οι Nvidia GPUs. Αν και η Python είναι πολύ δημοφιλής γλώσσα, καθώς χρησιμοποιεί μεταφραστή είναι σχετικά αργή και μη κατάλληλη για υπολογισμούς υψηλών επιδόσεων. Οπότε με την χρήση της CUDA, οι εφαρμογές που γράφονται σε python μπορούν να εκμεταλλευτούν την ταχύτητα που προσφέρουν οι GPUs της Nvidia [8].

# 5 Ο decorator vectorize()

Οι decorators είναι ένα ισχυρό και χρήσιμο εργαλείο της Python καθώς επιτρέπουν στους προγραμματιστές να τροποποιούν την συμπεριφορά μια συνάρτησης ή μιας κλάσης. Οι decorators «τυλίγουν» μια υπάρχουσα συνάρτηση και δίνουν τη δυνατότητα να επεκταθεί η συμπεριφορά της χωρίς να την τροποποιούν μόνιμα [4].

Ο decorator vectorize() στο Numba, επιτρέπει στις συναρτήσεις Python που έχουν ως είσοδο ορίσματα βαθμωτά μεγέθη να χρησιμοποιηθούν ως παγκόσμιες συναρτήσεις NumPy (NumPy universal functions – ufuncs). Με τη χρήση του decorator vectorize, ο μεταφραστής Numba μπορεί να μεταφράσει μια συνάρτηση Python σε μια ufunc που λειτουργεί πάνω σε πίνακες (arrays) NumPy τόσο γρήγορα σαν να ήταν γραμμένη σε γλώσσα C. Επίσης με τον decorator vectorize, ο προγραμματιστής γράφει την συνάρτησή του σαν να λειτουργεί σε βαθμωτά μεγέθη

εισόδου αντί για πίνακες (arrays). Ο Numba αναλαμβάνει τη δημιουργία του απαιτούμενου κώδικα επαναλήψεων (loop) φροντίζοντας για την αποτελεσματική εκτέλεση της επαναληπτικής διαδικασίας στα πραγματικά δεδομένα πινάκων [5].

Παρακάτω δίνεται ένα παράδειγμα χρήσης του decorator vectorize με μία υπογραφή. Τα ορίσματα εισόδου είναι δύο και έχουν τύπο float64, ενώ η συνάρτηση επιστρέφει ένα αποτέλεσμα πάλι τύπου float64.

```
From numba import vectorize, float64
@vectorize([float64(float64, float64)])
def f(x, y):
     return x + y
```

Ο decorator vectorize() υποστηρίζει διαφορετικές επιλογές για την μονάδα και τον τρόπο επεξεργασίας που θα εκτελεστεί μια συνάρτηση ufunc, οι οποίες φαίνονται στον παρακάτω πίνακα.

| **Επιλογή** | **Μονάδα και τρόπος επεξεργασίας** |
|---|---|
| cpu | Single-threaded CPU |
| parallel | Multi-core CPU |
| cuda | CUDA GPU |

Αν στο παραπάνω παράδειγμα χρήσης επιλεγεί η εκτέλεση στην CUDA GPU η δήλωση του decorator πρέπει να γίνει ως εξής:

```
from numba import vectorize, float64
@vectorize([float64(float64, float64)], 'cuda')
def f(x, y):
     return x + y
```

Μία γενική οδηγία είναι χρησιμοποιείται διαφορετική επιλογή για διαφορετικά μεγέθη δεδομένων και διαφορετικούς υπολογιστικούς αλγόριθμους. Η επιλογή 'cpu' λειτουργεί ικανοποιητικά για μικρά μεγέθη δεδομένων (μικρότερα του 1KB) και αλγόριθμους χαμηλών υπολογιστικών απαιτήσεων. Η επιλογή αυτή έχει το μικρότερο επιπλέον κόστος επεξεργασίας. Η επιλογή 'parallel' λειτουργεί καλά για μεσαίου μεγέθους δεδομένα (περίπου λιγότερα από 1MB), καθώς για να γίνει εκμετάλλευση της παράλληλης επεξεργασίας με νήματα (threads) εισάγεται μια επιπλέον καθυστέρηση η οποία δεν είναι συμφέρουσα σε μικρά μεγέθη δεδομένων. Τέλος, η επιλογή 'cuda' λειτουργεί καλά για μεγάλου μεγέθους δεδομένα (περίπου περισσότερα από 1MB) και αλγόριθμους υψηλών υπολογιστικών απαιτήσεων, καθώς η μεταφορά δεδομένων

προς και από την μνήμη της GPU προσθέτει σημαντικό χρόνο και για να συμφέρει θα πρέπει ο χρόνος εκτέλεσης να είναι στο CPU να είναι ιδιαίτερα μεγάλος.

# 6 Υπολογιστική Εφαρμογή με Επιτάχυνση GPU

Η υπολογιστική εφαρμογή που αναπτύχθηκε στην παρούσα εργασία βασίστηκε στη λογική του παραδείγματος που υπάρχει στην αναφορά [9] και χρησιμοποιεί τον decorator vectorize().

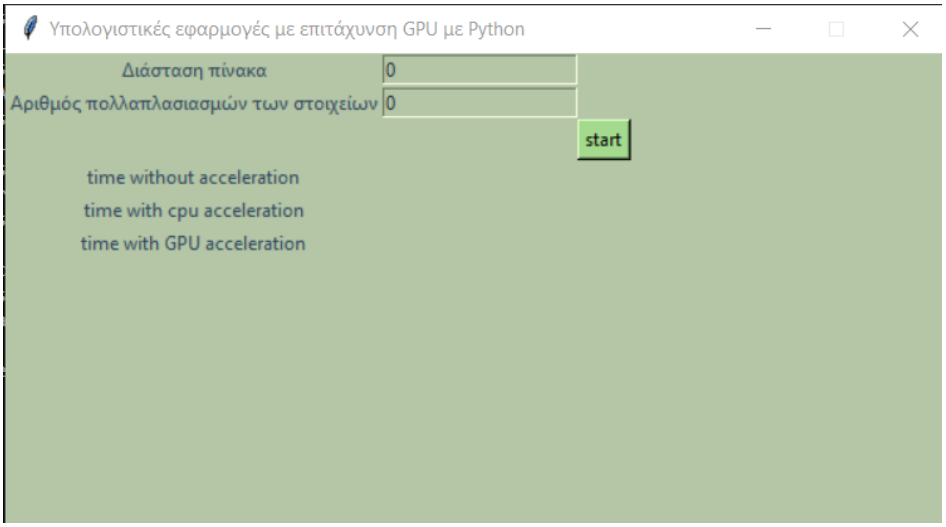
Ως υπολογιστική εφαρμογή αναπτύχθηκε ένα πρόγραμμα που συσσωρεύει το άθροισμα των γινομένων των στοιχείων δύο τετραγωνικών πινάκων διάστασης στο αντίστοιχο στοιχείο ενός άλλου τετραγωνικού πίνακα ίδιας διάστασης. Το μέγεθος NXN των πινάκων καθορίζεται από την παράμετρο N και το πλήθος των επαναλήψεων του υπολογισμού των γινομένων και αντίστοιχα των αθροίσεων που θα γίνουν καθορίζεται από την παράμετρο M. Στην παρακάτω εικόνα εμφανίζονται τα βήματα της εφαρμογής:

ΒΗΜΑ 1: Διάβασμα δεδομένων εισόδου: Παράμετροι N και M
ΒΗΜΑ 2: Δημιουργία τετραγωνικού πίνακα A μεγέθους NXN με στοιχεία $A[i,j], i,j = 1, \dots, N$.
ΒΗΜΑ 3: Γέμισμα του πίνακα με τυχαίους αριθμούς τύπου float32 ομοιόμορφα κατανεμημένων από το 50 έως το 100.
ΒΗΜΑ 4: Δημιουργία τετραγωνικού πίνακα A μεγέθους NXN με στοιχεία $B[i,j], i,j = 1, \dots, N$.
ΒΗΜΑ 5: Γέμισμα του πίνακα με τυχαίους αριθμούς τύπου float32 ομοιόμορφα κατανεμημένων από το 50 έως το 100.
ΒΗΜΑ 6: ***Υπολογισμός στην CPU*** του τετραγωνικού πίνακα F μεγέθους NXN με στοιχεία $F[i,j] = \sum_{m=1}^{M} A[i,j] * B[i,j]$ και καταγραφή και εκτύπωση του χρόνου εκτέλεσης του υπολογισμού.
ΒΗΜΑ 7: ***Υπολογισμός στην CPU με τον decorator vectorize()*** του τετραγωνικού πίνακα F μεγέθους NXN ως εξής: $F[i,j] = \sum_{m=1}^{M} A[i,j] * B[i,j]$ και καταγραφή και εκτύπωση του χρόνου εκτέλεσης του υπολογισμού.
ΒΗΜΑ 8: ***Υπολογισμός στην GPU με τον decorator vectorize()*** του τετραγωνικού πίνακα F μεγέθους NXN ως εξής: $F[i,j] = \sum_{m=1}^{M} A[i,j] * B[i,j]$ και καταγραφή και εκτύπωση του χρόνου εκτέλεσης του υπολογισμού.

*Εικόνα 4: Βήματα υπολογιστικής εφαρμογής*

# 7 Γραφικό Interface tkinter και Χρήση Threading

Στην παρακάτω εικόνα παρουσιάζεται το παράθυρο του γραφικού interface της εφαρμογής που αναπτύχθηκε με τη βοήθεια της βιβλιοθήκης tkinter.




*Εικόνα 5: Παράθυρο γραφικού interface της υπολογιστικής εφαρμογής*

Στο πρώτο κουτί εισαγωγής στοιχείων, ο χρήστης εισάγει τις διαστάσεις του τετραγωνικού πίνακα και στο δεύτερο πόσες φορές θα επαναλαμβάνεται ο πολλαπλασιασμός μεταξύ των στοιχείων των δύο πινάκων.

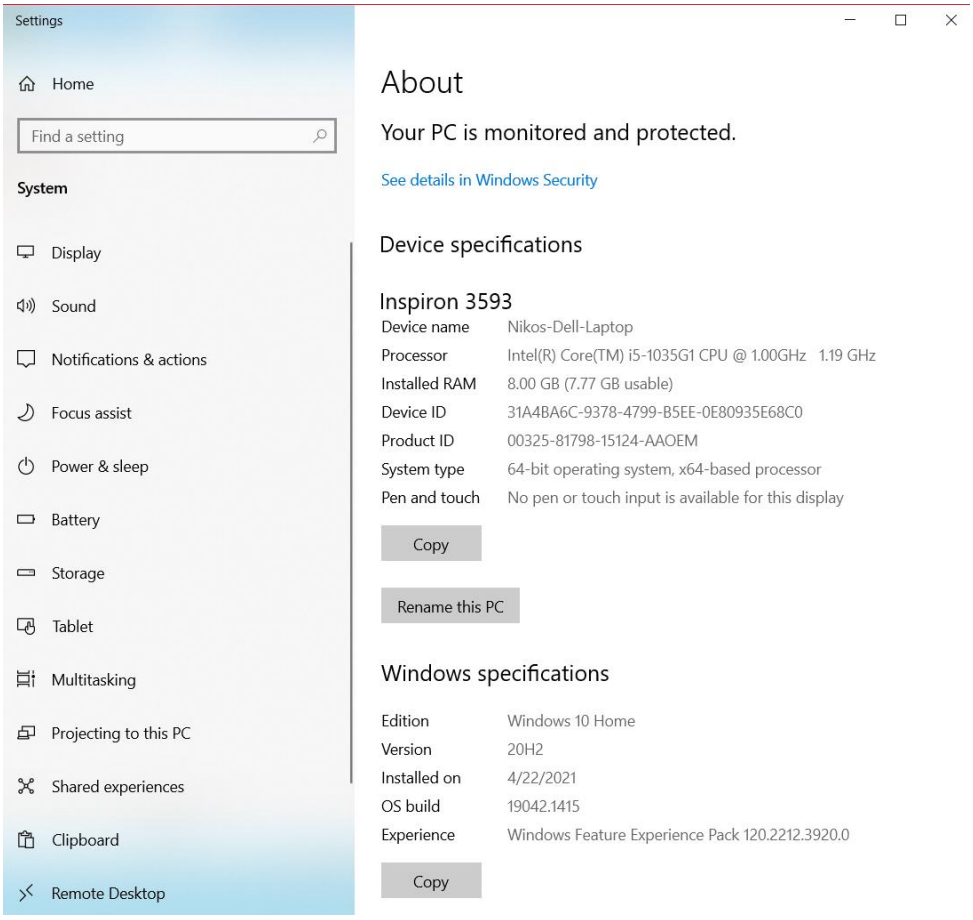
Με το κουμπί (start) ο χρήστης ξεκινάει την εκτέλεση του προγράμματος, δημιουργούνται οι 2 τετραγωνικοί πίνακες A και B και καλούνται με τη σειρά οι τρεις υπολογιστικές συναρτήσεις που αντιστοιχούν στα βήματα 6, 7 και 8 που αναφέρθηκαν στην προηγούμενη παράγραφο, δημιουργώντας κάθε φορά τον νέο τετραγωνικό πίνακα με τα αποτελέσματα.

Μετά την ολοκλήρωση κάθε υπολογιστικής συνάρτησης, εμφανίζονται στο ίδιο παράθυρο οι αντίστοιχοι χρόνοι εκτέλεσης με ακρίβεια 8 δεκαδικών ψηφίων.

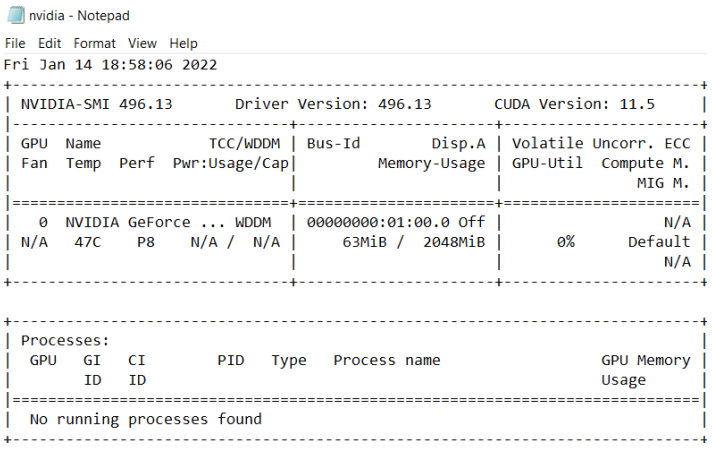
Στο πρόγραμμα χρησιμοποιήθηκε επίσης και η βιβλιοθήκη threading έτσι ώστε το παράθυρο να συνεχίζει να λειτουργεί ενόσω ο υπολογιστής καλεί με τη σειρά τις συναρτήσεις υπολογισμού και εκτελεί τους πολλαπλασιασμούς των στοιχείων των πινάκων.

## 8 Περιβάλλον Ανάπτυξης και Εγκατάστασης λογισμικού

Ο υπολογιστής στον οποίο αναπτύξαμε και τρέξαμε την υπολογιστική εφαρμογή είναι ένα DELL laptop Intel Core i5 που τρέχει Windows 10 και έχει δύο GPU, η μία είναι NVIDIA. Τα βασικά χαρακτηριστικά του υλικού και του λειτουργικού συστήματος όπως φαίνονται από τα System Properties είναι τα εξής:




Κάποια χαρακτηριστικά της GPU NVIDIA, όπως Driver, και CUDA versions, μνήμη, θερμοκρασία κ.α. εμφανίζονται παρακάτω μέσω εκτέλεσης της εντολής nvidia-smi (NVIDA System Management Interface) στο command line shell του Anaconda.




Για να εγκαταστήσουμε το περιβάλλον ανάπτυξης της εφαρμογής εκτελούμε με τη σειρά τα παρακάτω βήματα:

1. Αρχικά κατεβάζουμε και εγκαθιστούμε το περιβάλλον Anaconda από τον σύνδεσμο https://www.anaconda.com/products/individual για να αποκτήσουμε πρόσβαση στα conda packages στα οποία περιέχεται ο μεταγλωττιστής Numba.
2. Στην συνέχεια στο command prompt shell του Anaconda εκτελούμε την παρακάτω εντολή για να εγκατασταθεί στον υπολογιστή ο Numba

```
conda install numba
```

3. Στην συνέχεια, πάλι στο command prompt shell του Anaconda εκτελούμε την παρακάτω εντολή για να επιτραπεί η χρήση της GPU από τη numba μέσω του target=cuda

```
conda install cudatoolkit
```

Το πρόγραμμα python `50_code.py` που αναπτύξαμε εκτελείται μέσα από το command prompt shell του Anaconda με την παρακάτω εντολή:

```
python 50_code.py
```

Στο παράθυρο με το command prompt shell του Anaconda από το οποίο γίνεται η εκτέλεση του προγράμματος εμφανίζονται διάφορες πληροφορίες καθώς εξελίσσεται η εκτέλεση του προγράμματος, όπως οι πίνακες εισόδου A και B, κάποιο στοιχείων των πινάκων με τα αποτελέσματα των πράξεων και οι χρόνοι εκτέλεσης των υπολογιστικών συναρτήσεων.

## 9 Αποτελέσματα – Μετρήσεις – Συμπεράσματα

Για να δείξουμε πότε κερδίζουμε από την επιτάχυνση στην CPU και στην GPU εκτελέσαμε το πρόγραμμα αρκετές φορές με αυξανόμενο υπολογιστικό φορτίο κάθε φορά. Θέσαμε ως διάσταση των τετραγωνικών πινάκων N=1000, που αντιστοιχεί σε μέγεθος πινάκων ίσο με 4 Mbytes, διότι κάθε πίνακας έχει 1 εκατομμύριο στοιχεία ενώ κάθε στοιχείο είναι τύπου float32 με μέγεθος ίσο 4 bytes.

Η αύξηση του υπολογιστικού φορτίου ελέγχθηκε μέσω της αύξησης του αριθμού των επαναλήψεων των πολλαπλασιασμών M. Πχ για M=100 έχουμε 100 φορές επανάληψη των πολλαπλασιασμών των στοιχείων των πινάκων και επομένως 100 εκατομμύρια πολλαπλασιασμούς. Επιπλέον σε κάθε επανάληψη γίνονται και 1 εκατομμύριο προσθέσεις, άρα συνολικά 100 εκατομμύρια προσθέσεις.

*Πίνακας 1: Αποτελέσματα Μετρήσεων*

| **M = Αριθμός πολ/μών float32 (millions)** | **Χρόνος εκτέλεσης CPU (seconds)** | **Χρόνος εκτέλεσης CPU με επιτάχυνση vectorize (seconds)** | **Επιτάχυνση CPU (φορές γρηγορότερα)** | **Χρόνος εκτέλεσης GPU με επιτάχυνση vectorize (seconds)** | **Επιτάχυνση GPU (φορές γρηγορότερα)** |
|---|---|---|---|---|---|
| 100 | 0.337744 | 0.053578 | 6.30378 | 0.851047 | 0.39686 |
| 500 | 1.671494 | 0.529422 | 3.15721 | 0.866178 | 1.92973 |
| 1000 | 3.411271 | 2.512819 | 1.35755 | 1.010324 | 3.37641 |
| 1500 | 5.084771 | 1.706936 | 2.97889 | 0.885602 | 5.74160 |
| 2000 | 6.972906 | 4.903417 | 1.42205 | 1.020302 | 6.83416 |
| 2500 | 8.972863 | 2.88873 | 3.10616 | 1.03636 | 8.65806 |
| 3000 | 11.610839 | 7.942279 | 1.46190 | 1.125201 | 10.31890 |
| 3500 | 12.444238 | 4.064483 | 3.06170 | 1.024262 | 12.14947 |
| 4000 | 15.197093 | 4.649553 | 3.26851 | 0.986644 | 15.40281 |
| 4500 | 16.773789 | 5.22045 | 3.21309 | 0.978701 | 17.13883 |
| 5000 | 18.626811 | 5.822392 | 3.19917 | 0.990838 | 18.79905 |
| 10000 | 37.306349 | 11.687285 | 3.19205 | 1.063750 | 35.07060 |

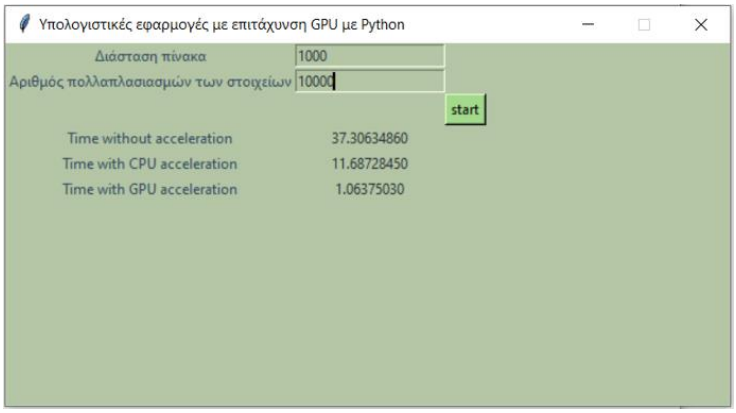
Όπως παρατηρούμε από τον πίνακα το όφελος από την επιτάχυνση μέσω της χρήσης του decorator vectorize() στην GPU αρχίζει να εμφανίζεται από την τιμή M=500 (εκατομμύρια πολλαπλασιασμοί) με επιτάχυνση 1.92 φορές πιο γρήγορα, ενώ όσο αυξάνουμε το πλήθος των πολλαπλασιασμών, τόσο πιο πολύ κερδίζουμε σε χρόνο εκτέλεσης με όφελος 35.07 φορές γρηγορότερα όταν γίνονται 10 δισεκατομμύρια πολλαπλασιασμοί. Αντίθετα, σε πολύ χαμηλό φορτίο (M=100) η χρήση GPU δεν προσφέρει κάποιο κέρδος.

Αντίστοιχα, μέσω της χρήσης του decorator vectorize() στην CPU υπάρχει ένα μικρό σχετικό κέρδος, με την επιτάχυνση να κυμαίνεται από 1.35 έως 6.30 φορές γρηγορότερα, χωρίς να υπάρχει κάποια συσχέτιση με την αύξηση των υπολογισμών. Μάλιστα η μεγαλύτερη επιτάχυνση επιτυγχάνεται με το μικρότερο υπολογιστικό φορτίο.

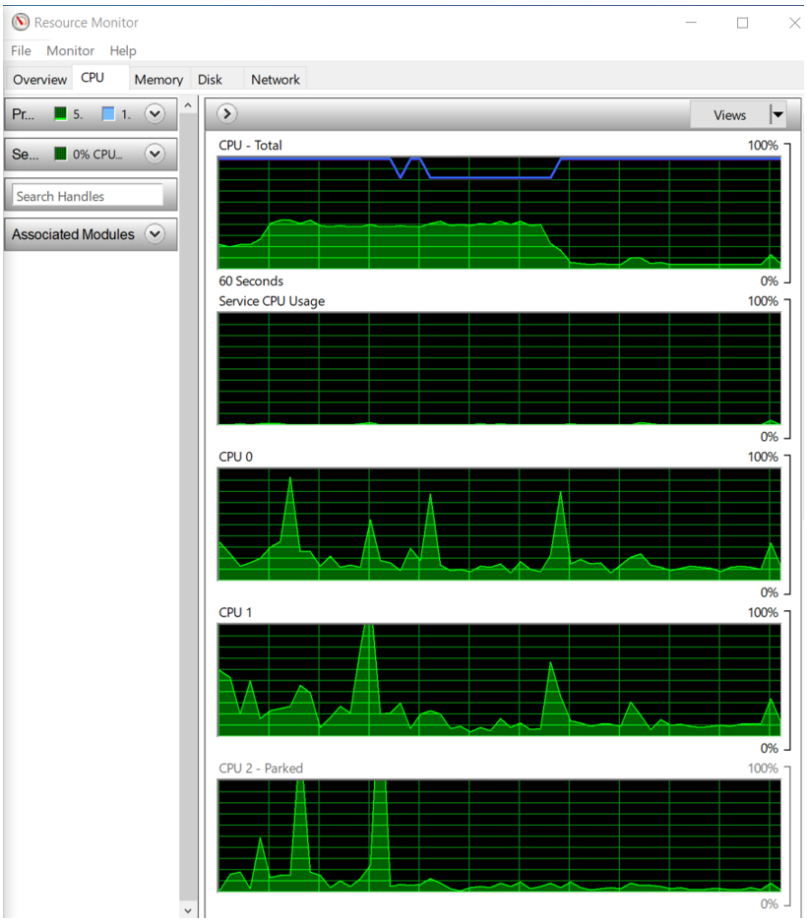
Στα ακόλουθα 3 σχήματα εμφανίζουμε παράθυρα από το τρέξιμο της εφαρμογής για N=1000 και M=10000.

Στο πρώτο παράθυρο φαίνεται το γραφικό interface με τα δεδομένα εισόδου και τα αποτελέσματα με τους 3 χρόνους εκτέλεσης (εκτέλεση στη CPU χωρίς επιτάχυνση, εκτέλεση στη

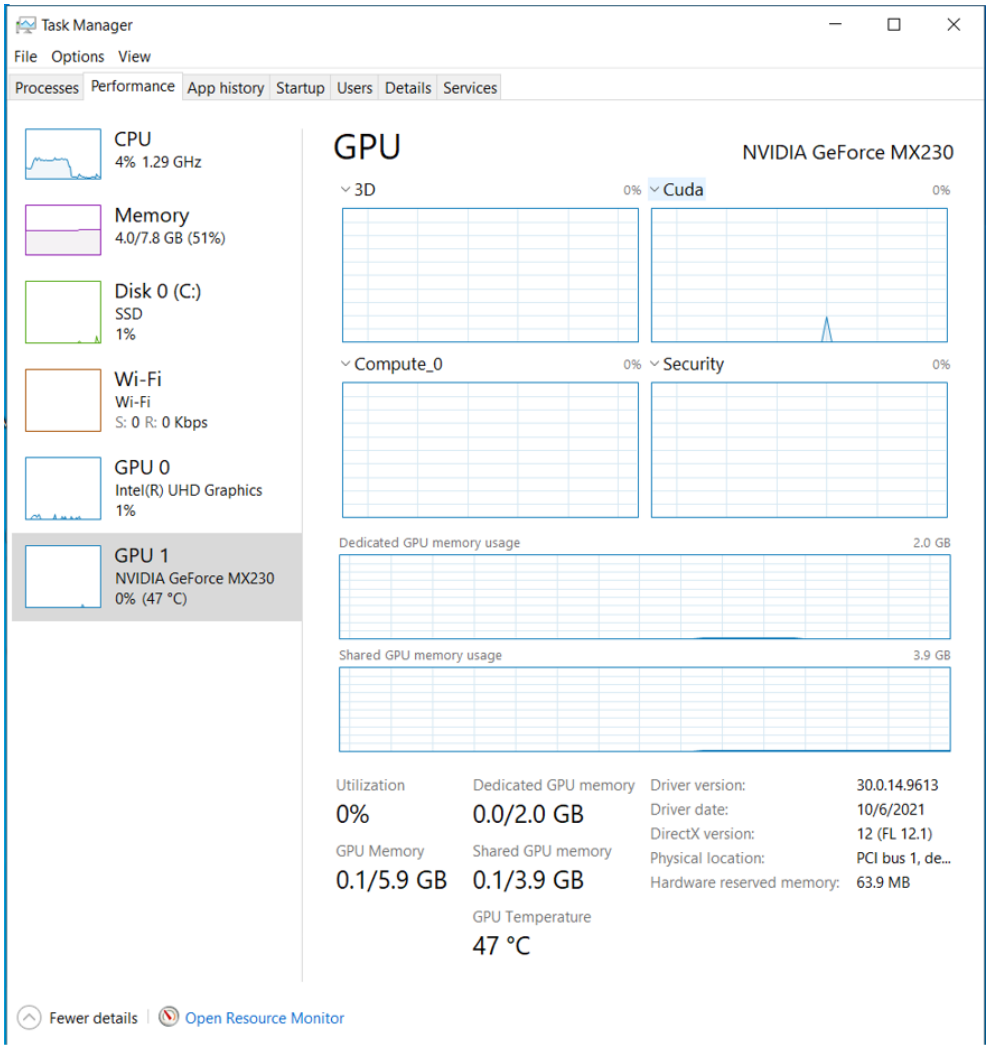
CPU με επιτάχυνση μέσω του decorator vectorize() και εκτέλεση στη GPU με επιτάχυνση μέσω του decorator vectorize()).




Στο επόμενο παράθυρο εμφανίζεται ένα screenshot από το Resource Monitor του task manager των Windows 10 κατά την εκτέλεση της υπολογιστικής εφαρμογής. Σε αυτό φαίνεται η χρησιμοποίηση στους δύο πυρήνες του υπολογιστή όταν εκτελούνται οι πολλαπλασιασμοί στη CPU.




Τέλος, στο ακόλουθο παράθυρο εμφανίζεται ένα screenshot από τον task manager των Windows 10 κατά την εκτέλεση της υπολογιστικής εφαρμογής. Σε αυτό φαίνεται η μικρής διάρκειας κορυφή που αντιστοιχεί στο ποσοστό χρησιμοποίησης της NVIDIA GPU όταν εκτελούνται οι πολλαπλασιασμοί.




Παραθέτουμε τον Πίνακα 2, ο οποίος προέκυψε από μετρήσεις που έγιναν με τα ίδια αριθμητικά δεδομένα N,M στον υπολογιστή με τα παρακάτω χαρακτηριστικά:

| | |
|---|---|
| Device name | LAPTOP-N4HCS4G7 |
| Processor | Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz |
| Installed RAM | 16.0 GB (15.8 GB usable) |
| System type | 64-bit operating system, x64-based processor |

*Πίνακας 2*

| **M = Αριθμός πολ/μών float32 (millions)** | **Χρόνος εκτέλεσης CPU (seconds)** | **Χρόνος εκτέλεσης CPU με επιτάχυνση vectorize (seconds)** | **Επιτάχυνση CPU (φορές γρηγορότερα)** | **Χρόνος εκτέλεσης GPU με επιτάχυνση vectorize (seconds)** | **Επιτάχυνση GPU (φορές γρηγορότερα)** |
|---|---|---|---|---|---|
| 100 | 0.207349 | 0.048034 | 4.316713 | 1.217511 | 0.170306 |
| 500 | 0.998294 | 0.393028 | 2.540007 | 0.889345 | 1.122505 |
| 1000 | 2.055196 | 0.845397 | 2.431042 | 0.169512 | 12.124192 |
| 1500 | 3.035847 | 1.266719 | 2.396622 | 0.955048 | 3.178738 |
| 2000 | 3.986097 | 1.706408 | 2.335958 | 0.763182 | 5.222997 |
| 2500 | 4.952822 | 2.157844 | 2.295264 | 0.838420 | 5.907328 |
| 3000 | 6.169792 | 2.634153 | 2.342230 | 0.215939 | 28.571921 |
| 3500 | 6.962209 | 3.041681 | 2.288935 | 0.813349 | 8.559928 |
| 4000 | 7.988715 | 3.483274 | 2.293450 | 0.845348 | 9.450209 |
| 4500 | 9.036920 | 3.883669 | 2.326903 | 0.895893 | 10.087053 |
| 5000 | 9.948075 | 4.369441 | 2.276739 | 0.856740 | 11.611545 |
| 10000 | 20.011320 | 8.689642 | 2.302893 | 0.934810 | 21.406831 |

Συμπερασματικά καταλήγουμε ότι οι χρόνοι εκτέλεσης της εφαρμογής με τα ίδια αριθμητικά δεδομένα M,N είναι παρόμοιοι στους δύο υπολογιστές, γεγονός που σημαίνει ότι επιτελείται ο κύριος στόχος της εργασίας να αναδειχθεί το όφελος επιτάχυνσης που προσφέρει η GPU για εφαρμογές με μεγάλο υπολογιστικό φορτίο.

## 10 Βιβλιογραφικές Αναφορές

[1] Mantas Levinas, A Complete Introduction to GPU Programming With Practical Examples in CUDA and Python, September 30, 2021, https://blog.cherryservers.com/introduction-to-gpu-programming-with-cuda-and-python

[2] Nickson Joram, Executing a Python Script on GPU Using CUDA and Numba in Windows 10, 30/4/2021, https://medium.com/geekculture/executing-a-python-script-on-gpu-using-cuda-and-numba-in-windows-10-1a1b10c29c9

[3] Numba: A High Performance Python Compiler, https://numba.pydata.org/

[4] Decorators in Python, 16/12/2021, https://www.geeksforgeeks.org/decorators-in-python/

[5] Creating NumPy universal functions: The vectorize() decorator, https://numba.pydata.org/numba-doc/latest/user/vectorize.html

[6] PyHEP 2021: CUDA and Python with Numba, video at https://www.youtube.com/watch?v=xes5ri5ccWY

[7] CUDA, https://en.wikipedia.org/wiki/CUDA

[8] GPU-Accelerated Computing with Python, https://developer.nvidia.com/how-to-cuda-python

[9] Mark Harris, Numba: High-Performance Python with CUDA Acceleration, 19/9/2013,https://developer.nvidia.com/blog/numba-python-cuda-acceleration/