Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

Χειμερινό Εξάμηνο 2024-2025

2ο Σετ Εργαστηριακών Ασκήσεων

Ον/νυμο: Παναγιώτης Καλοζούμης Α.Μ.: 1084560

Ον/νυμο: Δημήτριος Στασινός Α.Μ.: 1084643

Ερώτημα 1: SIMD και WENO5

Στόχος αυτού του ερωτήματος είναι να δοκιμάσουμε διάφορες μεθόδους διανυσματοποίησης των πράξεων του WENO5. Συγκεκριμένα, θα συγκρίνουμε:

- Την αρχική εκτέλεση του WENO, χωρίς καμία βελτιστοποίηση
- Μια εκδοχή με αυτόματη διανυσματοποίηση από τον compiler
- Μια εκδοχή με διανυσματοποίηση από το OpenMP
- Μια υλοποίηση με διανυσματικούς καταχωρητές AVX μεγέθους 512

Αυτόματη διανυσματοποίηση

Αρχικά θέλουμε να ελέγξουμε πώς η αυτόματη διανυσματοποίηση επηρεάζει την απόδοση του προγράμματος. Για να βελτιστοποιήσουμε μόνο το κομμάτι κώδικα που αφορά το WENO, χωρίς να αναγκαστούμε να δημιουργήσουμε ξεχωριστό source αρχείο, χρησιμοποιούμε τα pragma βελτιστοποίησης που παρέχει ο GCC:

```
#pragma GCC push_options
#pragma GCC optimize ("-ftree-vectorize")

void weno_minus_reference(...)
{
    for (int i=0; i<NENTRIES; ++i)
        out[i] = weno_minus_core(a[i], b[i], c[i], d[i], e[i]);
}

#pragma GCC pop_options
```

Με την επιλογή `GCC push_options` ο compiler αποθηκεύει σε μια στοίβα τα flags που του έχουμε περάσει, ώστε να βελτιστοποιήσει το ακόλουθο κομμάτι κώδικα με διαφορετικό τρόπο. Έπειτα χρησιμοποιούμε το flag `-ftree-vectorize`, το οποίο

ενεργοποιεί την αυτόματη διανυσματοποίηση μόνο για το συγκεκριμένο for loop. Ο compiler δεν επιχειρεί να διανυσματοποιήσει κανένα άλλο κομμάτι κώδικα. Έπειτα επαναφέρουμε τα flags του compiler με την επιλογή `pop_options`.

Παρατηρήσαμε ότι για να γίνει επιτυχώς η διανυσματοποίηση του for loop, πρέπει και το συνολικό αρχείο να γίνει compile με τουλάχιστον O1. Αν ούτε το αρχείο, ούτε το κομμάτι κώδικα (μέσω pragma) δεν έχουν κάποιο optimization level, τότε δε γίνεται καν προσπάθεια για διανυσματοποίηση, ανεξαρτήτως του flag `-ftree-vectorize`. Για να μπορούμε να βλέπουμε τα μηνύματα του compiler για τη διανυσματοποίηση, χρησιμοποιούμε το flag `-fopt-info-vect-all` κατά το compilation, το οποίο δείχνει όλες τις επιτυχείς και αποτυχημένες προσπάθειες διανυσματοποίησης. Μερικές από τις παρατηρήσεις μας φαίνονται στη συνέχεια:

- Αν το συνολικό αρχείο δεν έχει καθόλου optimization, ενώ το κομμάτι κώδικα έχει ένα οποιοδήποτε optimization level από O1 και πάνω (μέσω του pragma), δεν πραγματοποιείται διανυσματοποίηση:

```
weno.h:35:2: missed: couldn't vectorize loop

weno.h:36:12: missed: statement clobbers memory: _14 =
weno_minus_core (_12, _10, _8, _6, _4);

weno.h:31:6: note: vectorized 0 loops in function.

weno.h:36:12: missed: statement clobbers memory: _14 =
weno_minus_core (_12, _10, _8, _6, _4);
```

- Αν το κομμάτι κώδικα έχει optimization level O2/O3 και το αρχείο έχει O1, επίσης δεν πραγματοποιείται διανυσματοποίηση:

```
weno.h:35:2: missed: couldn't vectorize loop

weno.h:36:12: missed: not vectorized: relevant stmt not supported:
_14 = weno_minus_core (_12, _10, _8, _6, _4);

weno.h:31:6: note: vectorized 0 loops in function.
```

Τελικά, αποφασίσαμε να κάνουμε compile το αρχείο με O1 και να χρησιμοποιήσουμε μόνο το `#pragma GCC optimize("-fopt-info-vec")` στο κομμάτι κώδικα. Να σημειωθεί ότι ακόμα κι' έχουμε χρησιμοποιήσει όλα τα κατάλληλα flag για να γίνει διανυσματοποίηση, ο compiler δεν μπορεί να είναι σίγουρος ότι ο κώδικας όντως μπορεί να εκτελεστεί διανυσματικά χωρίς παρενέργειες. Αν η συνάρτηση που προσπαθούμε να διανυσματοποιήσουμε χρησιμοποιεί δείκτες, τότε ο compiler υποθέτει ότι οι περιοχές μνήμης δεν είναι ανεξάρτητες, αλλά έχουν μερική επικάλυψη. Ο compiler αποφεύγει την εφαρμογή βελτιστοποιήσεων σε τέτοιες περιπτώσεις, καθώς μπορεί να οδηγήσει σε λανθασμένα αποτελέσματα. Έτσι, στα μηνύματα του compiler παρατηρούμε ότι δημιουργούνται δύο εκδοχές του κώδικα, μια όπου η διανυσματική εκτέλεση είναι ασφαλής, και μια που δεν είναι:

```
weno.h:35:2: optimized: loop vectorized using 16 byte vectors
weno.h:35:2: optimized:  loop versioned for vectorization because
of possible aliasing
weno.h:31:6: note: vectorized 1 loops in function.
```

Αν είμαστε σίγουροι ότι οι πίνακές μας όντως δεν επικαλύπτονται, τότε μπορούμε να επιτρέψουμε στον compiler να εκμεταλλευτεί αυτές τις βελτιστοποιήσεις μέσω της λέξης restrict:

```
void weno_minus_reference(
      const float* restrict const a,
      const float* restrict const b,
      const float* restrict const c,
      const float* restrict const d,
      const float* restrict const e,
      float* restrict const out,
      const int NENTRIES
)
```

Αν εφαρμόσουμε όλα τα παραπάνω, παρατηρούμε ότι η διανυσματοποίηση του for loop στη γραμμή 35 του weno.h γίνεται επιτυχώς, με διανυσματικούς καταχωρητές των 128 bit (4 float):

```
gcc -O1 -fopt-info-vec-all -o bench.o -c bench.c

weno.h:35:2: optimized: loop vectorized using 16 byte vectors
weno.h:31:6: note: vectorized 1 loops in function.
```

Κάτι τελευταίο που πρέπει να επισημάνουμε είναι ότι, ακόμα κι' αν η διανυσματοποίηση γίνει επιτυχώς από τον compiler, δεν είναι εγγυημένο ότι το πρόγραμμα θα εκτελεστεί σωστά. Αν η μνήμη που χρησιμοποιείται δεν είναι ευθυγραμμισμένη, τότε για την αρχικοποίηση ενός διανύσματος απαιτούνται πολλαπλές εντολές φόρτωσης από τη μνήμη, μειώνοντας σημαντικά το όφελος των διανυσματικών εντολών. Δεσμεύοντας μνήμη που είναι ευθυγραμμισμένη στο πλήθος bytes των καταχωρητών που χρησιμοποιούμε (π.χ. 16 για καταχωρητές των 128 bits), εξασφαλίζουμε ότι η φόρτωση στους καταχωρητές γίνεται από σωστές θέσεις, και σε ένα μόνο βήμα. Έχουμε χρησιμοποιήσει ευθυγράμμιση στα 64 bytes, καθώς καλύπτει όλες τις περιπτώσεις για πιθανά μεγέθη διανυσμάτων (από 128 έως και 512 bits) που θα χρησιμοποιήσουμε για τις υλοποιήσεις μας:

```
posix_memalign((void **)&tmp, 64, sizeof(float) * NENTRIES);
```

Διανυσματοποίηση με OpenMP

Για τη διανυσματοποίηση με OpenMP χρησιμοποιούμε την οδηγία `#pragma omp simd`. Ομοίως με πριν, θα χρειαστεί το πρόγραμμα να γίνει compile με τουλάχιστον O1 και με το `-fopenmp` flag για να πετύχει η διανυσματοποίηση. Επίσης όλοι οι δείκτες που χρησιμοποιούμε έχουν restrict και η μνήμη είναι ευθυγραμμισμένη. Για να εξασφαλίσουμε ότι η αυτόματη διανυσματοποίηση που εφαρμόσαμε προηγουμένως δεν επηρεάζει κάπως τη διανυσματοποίηση του OpenMP, κάνουμε compile την υλοποίηση με OpenMP ως διαφορετικό translation unit, δηλαδή χωρίσαμε τον κώδικα σε header και source και απλά συμπεριλαμβάνουμε το header file στον κώδικα του benchmark.

```
#pragma omp simd
for (int i = 0; i < NENTRIES; ++i)
    out[i] = weno_minus_core_simd(a[i], b[i], c[i], d[i], e[i]);
```

Παρατηρούμε ότι η διανυσματοποίηση γίνεται επιτυχώς:

```
gcc -O1 -fopenmp -fopt-info-vec -o weno_simd.o -c weno_simd.c

weno_simd.c:39:58: optimized: loop vectorized using 16 byte
vectors
```

Διανυσματοποίηση με AVX-512

Για τη διανυσματοποίηση με AVX/SSE χρησιμοποιήσαμε καταχωρητές μεγέθους 512 (AVX-512). Η βασική ιδέα πίσω από αυτή την υλοποίηση είναι ότι φορτώνουμε σε 5 διανυσματικούς καταχωρητές va, vb, vc, vd και ve κάθε φορά τα επόμενα 16 στοιχεία (float) των αντίστοιχων πινάκων a, b, c, d και e, αλλάζοντας και το βήμα του for loop. Έπειτα, εντός της συνάρτησης weno_minus_core οι πράξεις γίνονται ακριβώς όπως και στον αρχικό κώδικα, αλλά αντί για απλές μεταβλητές χρησιμοποιούμε τα διανύσματα αυτά. Κάθε κλήση της συνάρτησης παράγει τα 16 επόμενα αποτελέσματα ως ένα διάνυσμα, το οποίο έπειτα αποθηκεύεται στις αντίστοιχες θέσεις του πίνακα out. Η αξιοποίηση των διανυσματικών καταχωρητών και του αντίστοιχου συνόλου εντολών επιτυγχάνεται μέσω intrinsics, φορτώνοντας το header “x86intrin.h”.

Για την υλοποίησή μας έχουν λάβει υπ’ όψη τα παρακάτω:

- Οι πίνακες από τους οποίους διαβάζουμε είναι, όπως και πριν, ευθυγραμμισμένοι, ώστε να αξιοποιούμε τις εντολές που προσφέρονται για εγγραφή και ανάγνωση από ευθυγραμμισμένη μνήμη (_mm_load_ps/ _mm_store_ps), επιτυγχάνοντας έτσι καλύτερη απόδοση. Αν η μνήμη δεν ήταν ευθυγραμμισμένη στα 64 bytes, οι εντολές αυτές θα οδηγούσαν σε σφάλμα
- Εκτός από την ευθυγράμμιση στα 64 bytes, πρέπει και οι ίδιοι οι πίνακες να έχουν μέγεθος πολλαπλάσιο των 64 bytes (ισοδύναμα, 16 floats). Αυτό απαιτείται διότι τα διανύσματά μας διαβάζουν κάθε φορά 16 float από τον πίνακα, άρα υπάρχει περίπτωση να διαβάσουμε μνήμη εκτός ορίων. Σε έναν πραγματικό υπολογισμό, η ιδανική λύση θα ήταν να επεκτείνουμε τους πίνακες στο επόμενο πολλαπλάσιο των 16 στοιχείων και απλά να αγνοούμε τα επιπλέον αποτελέσματα. Ωστόσο, για τις ανάγκες του benchmark, αρκεί απλά να κόψουμε στοιχεία από τον πίνακα μέχρι να φτάσουμε το αμέσως προηγούμενο πολλαπλάσιο του 16. Αυτό κάνει η παρακάτω σχέση, την οποία τροποποιήσαμε για το πρόβλημά μας:

```
const int NENTRIES = 16 * (N[i] / 16);
```

- Εντός της συνάρτησης χρειάζεται συχνά να κάνουμε πράξεις μεταξύ των μεταβλητών και κάποιων αριθμητικών σταθερών. Γι’ αυτές τις πράξεις, ορίζουμε διανύσματα μεγέθους 16 floats που έχουν παντού το ίδιο στοιχείο (με την _mm_set1_ps). Επειδή η χρήση σταθερών είναι πολύ συχνή, και πολλές απ’ αυτές επαναχρησιμοποιούνται, ορίζουμε τις πιο βασικές απ’ αυτές στην αρχή της συνάρτησης, ώστε να γλυτώνουμε περιττές εντολές. Ενδεικτικά:

```
__m512 _4_div_3  = _mm512_set1_ps(4.f/3.f);
__m512 _19_div_3 = _mm512_set1_ps(19.f/3.f);
__m512 _11_div_3 = _mm512_set1_ps(11.f/3.f);
...
```

Για παρόμοιους λόγους με το OpenMP, έτσι και εδώ η υλοποίηση βρίσκεται σε διαφορετικό translation unit. Για λόγους ισοδυναμίας όλων των υλοποιήσεων, και αυτή

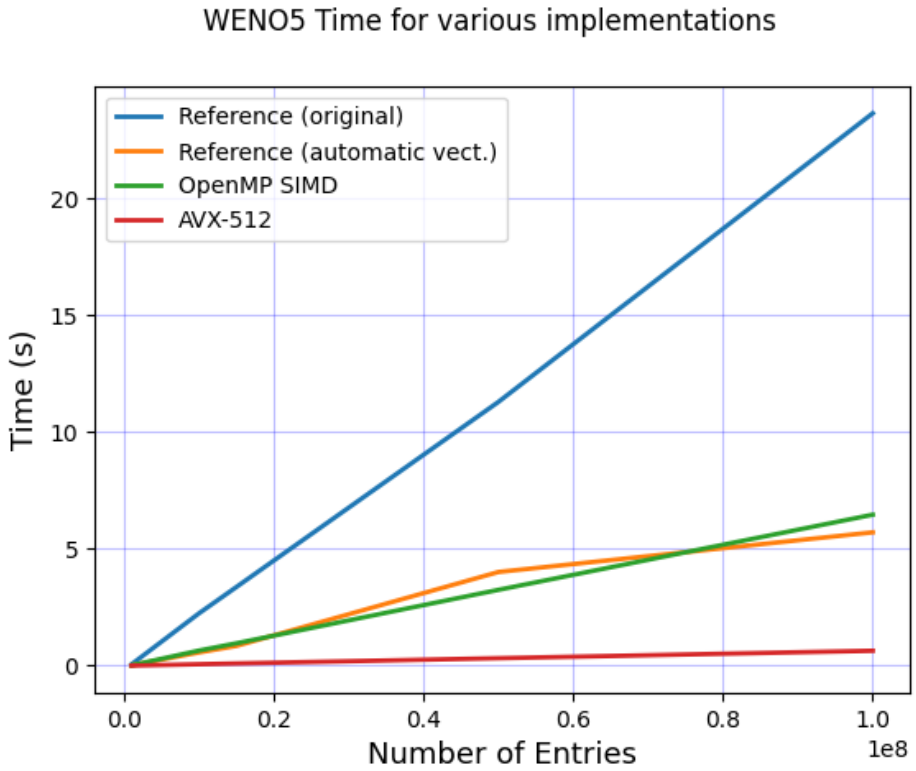
έγινε compile με O1, αν και δεν είναι αναγκαίο για την ρητή διανυσματοποίηση που κάναμε εμείς:

```
gcc -O1 -fopt-info-vec -o weno_avx.o -c weno_avx.c
```

Πειραματική σύγκριση

Για την αξιολόγηση της υλοποίησης μας, υλοποιήσαμε επιπλέον ένα Python script που καλεί πολλές φορές το benchmark πρόγραμμα. Εντός του benchmark, τρέχουμε την αρχική εκδοχή και τις 3 βελτιστοποιήσεις που κάναμε. Κάθε εκτέλεση γίνεται 8 φορές με τα ίδια δεδομένα. Συγκεντρώνουμε τους χρόνους από όλες τις εκτελέσεις για κάθε υλοποίηση σε κατάλληλο πίνακα, ο οποίος αντιγράφεται πίσω στο script μέσω διαμοιραζόμενης μνήμης. Έπειτα, έχοντας όλους τους χρόνους για τα διαφορετικά μεγέθη, δημιουργούμε τα κατάλληλα διαγράμματα σε Python. Οι δοκιμές έγιναν για μεγέθη [1e6, 1e7, 1.5e7 ,0.5e8, 1e8].

Παρακάτω φαίνεται το διάγραμμα που συγκρίνει τους χρόνους αυτούς για τις διάφορες υλοποιήσεις, συναρτήσει του μεγέθους του προβλήματος:



Όπως ήταν αναμενόμενο, σε κάθε περίπτωση έχουμε πολύ καλύτερη απόδοση από τον αρχικό κώδικα. Παρατηρούμε ότι η αυτόματη διανυσματοποίηση του compiler και η διανυσματοποίηση του OpenMP οδηγεί σε περίπου την ίδια απόδοση, χωρίς να είναι κάποιο αυστηρά καλύτερο. Αυτό είναι αναμενόμενο, καθώς το OpenMP SIMD στην πραγματικότητα αξιοποιεί την αυτόματη διανυσματοποίηση, παρέχοντας απλά επιπλέον πληροφορίες στον compiler, οι οποίες δεν προκύπτουν από την στατική ανάλυση, για το πώς να βελτιστοποιήσει καλύτερα τον κώδικα. Δραματική βελτίωση παρατηρούμε με τη χρήση AVX-512. Σε αυτή την υλοποίηση οι καταχωρητές έχουν πολύ μεγαλύτερο εύρος. Επίσης, με την ρητή διανυσματοποίηση μπορούμε να ελέγχουμε ακριβώς πώς βελτιστοποιείται κάθε κομμάτι των υπολογισμών, πετυχαίνοντας καλύτερη απόδοση

εκεί όπου η αυτόματη διανυσματοποίηση δεν μπορούσε να βοηθήσει. Παρακάτω φαίνονται οι τιμές του γραφήματος και αριθμητικά, σε δευτερόλεπτα:

Entries	Reference (original)	Reference (automatic vect.)	OpenMP SIMD	AVX-512
1000000	0,061194	0,015161	0,016455	0,006117
10000000	2,241656	0,567626	0,649308	0,064154
15000000	3,370572	0,847319	0,960767	0,09722
50000000	11,29755	4,013808	3,249333	0,322038
1E+08	23,65858	5,712242	6,468461	0,640865

Ερώτημα 2: CUDA και Πολλαπλασιασμός Μιγαδικών Μητρώων

Για τον πολλαπλασιασμό μιγαδικών μητρώων, πρώτα αρχικοποιούμε τα μητρώα A, B, C, D στον host με τυχαίες τιμές. Για ελαφρώς γρηγορότερη αρχικοποίηση, χρησιμοποιούμε 4 νήματα, όπου κάθε νήμα αναλαμβάνει την ανάθεση τιμών σε ένα από τα μητρώα:

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        initialize_matrix_(A, N);

        #pragma omp task
        initialize_matrix_(B, N);

        #pragma omp task
        initialize_matrix_(C, N);

        #pragma omp task
        initialize_matrix_(D, N);
    }
}
```

Για να εξασφαλίσουμε ότι οι τιμές των μητρώων είναι όντως τυχαίες, κάθε νήμα αρχικοποιεί ένα seed που εξαρτάται από το ID του μέσα στην παράλληλη περιοχή:

```
srand(time(NULL) + 1000 * omp_get_thread_num());
```

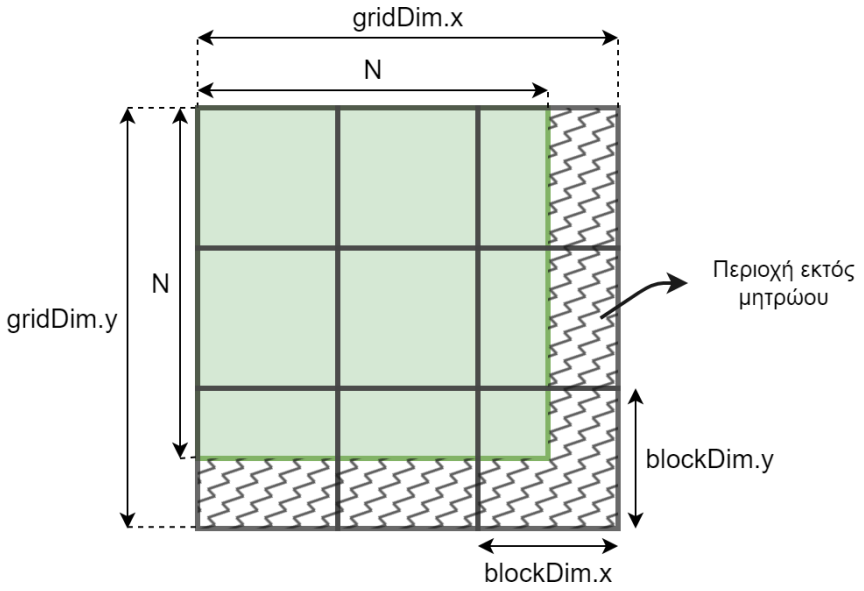
Για να σιγουρευτούμε για την ορθότητα των υπολογισμών που εκτελούμε στην GPU, καθώς επίσης και για να αξιολογήσουμε την υλοποίησή μας, φτιάξαμε και μια ισοδύναμη υλοποίηση του μιγαδικού πολλαπλασιασμού μητρώων στην CPU την οποία παραλληλοποιήσαμε, με νήματα του OpenMP.

Για την εκτέλεση των πράξεων στη GPU ακολουθήσαμε δύο προσεγγίσεις. Η πρώτη μας υλοποίηση ήταν μια naïve, προφανής λύση που χρησιμοποιεί πολλαπλούς kernel και έχει μεγαλύτερες απαιτήσεις σε μνήμη. Στη συνέχεια βελτιώσαμε τη λύση αυτή ώστε να είναι πιο αποδοτική και να εκτελεί όλες τις απαιτούμενες πράξεις σε ένα μόνο kernel launch, χωρίς να απαιτεί επιπλέον μνήμη. Στη συνέχεια παρουσιάζουμε και συγκρίνουμε και τις δύο υλοποιήσεις ώστε να τονίσουμε τη σημασία της σωστής χρήσης της GPU.

Και για τις δύο υλοποιήσεις, πριν μπορέσουμε να τρέξουμε κάποια συνάρτηση στη GPU, θα χρειαστεί να ορίσουμε το πλέγμα του προβλήματος για το kernel launch. Για προγραμματιστική ευκολία αποφασίσαμε να χρησιμοποιήσουμε δισδιάστατο πλέγμα και μπλοκ, καθώς απεικονίζονται καλά πάνω στα μητρώα μας και διευκολύνουν τη διαχείριση των νημάτων. Κάθε μπλοκ έχει σταθερή διάσταση 16x16. Ένα τέτοιο μπλοκ έχει συνολικά 256 νήματα, το όποιο είναι πολλαπλάσιο του 32. Αυτό είναι ιδανικό καθώς οι περισσότερες NVIDIA GPU εκτελούν υπολογισμούς παράλληλα σε ομάδες 32 νημάτων (warps) . Αυτό εξασφαλίζει ότι τα περισσότερα μπλοκ θα είναι πλήρως απασχολημένα, εκτός ενδεχομένως από μερικά, για λόγους που θα εξηγήσουμε στη συνέχεια. Έπειτα, χρησιμοποιούμε τόσα μπλοκ σε κάθε διάσταση ώστε να καλύψουμε πλήρως το μητρώο μας. Χρησιμοποιούμε τον παρακάτω τύπο για κάθε διάσταση, έστω για τη x:

$$gridx(N) = \frac{N + blockx - 1}{blockx}$$

Αυτός ο τύπος εξασφαλίζει ότι για μέγεθος N μεταξύ δύο πολλαπλασίων του blockx θα χρησιμοποιούμε το ίδιο πλήθος μπλοκ στη διάσταση x. Για παράδειγμα, από N=17, έως N=32 θα χρησιμοποιηθούν 2 μπλοκ (το εύρος τιμών είναι 2 έως 2.93, αλλά λόγω της ακέραιας διαίρεσης γίνεται 2). Με αυτό τον συνδυασμό μεγέθους πλέγματος και μπλοκ εξασφαλίζουμε ότι κάθε νήμα θα απεικονιστεί σε μια θέση του μητρώου, ώστε να υπολογίσει το αντίστοιχο αποτέλεσμα. Προφανώς, για N από 17 έως και 31, δηλαδή για N που δεν είναι πολλαπλάσιο του block.x = 16, τα περιθωριακά μπλοκ θα καλύπτουν και μια περιοχή εκτός του μητρώου. Τα νήματα που είναι εκτός του μητρώου θα χρειαστεί να παραμείνουν αδρανή, καθώς δεν μπορούν να εκτελέσουν κάποιον υπολογισμό. Επομένως, για N που δεν είναι πολλαπλάσιο του 16 μειώνεται η αξιοποίηση της GPU:



Για δεδομένο kernel launch, κάθε νήμα εκτελεί ένα αντίγραφο του kernel. Για να ξέρει κάθε νήμα ποιο στοιχείο του αποτελέσματος να υπολογίσει, χρειάζεται να πάρουμε τη θέση του στο πλέγμα, μέσω της θέσης του μπλοκ όπου ανήκει και της θέσης του μέσα στο μπλοκ:

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;

int pos = i * N + j;
```

Επομένως, ένα νήμα μπορεί να εκτελέσει πράξεις μόνο αν το (i, j) είναι στο [0, N-1] x [0, N-1]:

```
if (i < N && j < N)
{
    ...
}
```

Στη συνέχεια αναλύουμε τις δύο υλοποιήσεις μας.

Naïve υλοποίηση

Αρχικά δεσμεύουμε στη GPU τη μνήμη που απαιτείται για τα μητρώα A, B, C, D καθώς επίσης και τα αποτελέσματα E, F. Έπειτα αντιγράφουμε προς τη GPU τη μνήμη που αρχικοποιήθηκε στη CPU.

Παρατηρούμε ότι για τον υπολογισμό των E, F θα χρειαστεί πρώτα να υπολογίσουμε τα γινόμενα AC, BD, AD, BC. Επομένως, δεσμεύουμε επιπλέον μνήμη και γι' αυτά τα μητρώα, αρχικοποιώντας την έπειτα στο 0 για να μπορούν να εκτελεστούν σωστά οι πράξεις.

Για τους υπολογισμό απλά υπολογίζουμε 4 γινόμενα, μια αφαίρεση και μια πρόσθεση μητρώων. Όλες αυτές οι πράξεις μητρώων υλοποιούνται ως διαφορετικά kernel:

```
multiply_matrix<<<grid, block>>>(devAC, devA, devC, N);
multiply_matrix<<<grid, block>>>(devBD, devB, devD, N);
multiply_matrix<<<grid, block>>>(devAD, devA, devD, N);
multiply_matrix<<<grid, block>>>(devBC, devB, devC, N);

sub_matrix<<<grid, block>>>(devE, devAC, devBD, N);
add_matrix<<<grid, block>>>(devF, devAD, devBC, N);
```

Έπειτα οι πράξεις γίνονται κανονικά, για τη μια θέση που έχει αναλάβει κάθε νήμα.

Πολλαπλασιασμός (γραμμή * στήλη):

```
for (int k = 0; k < N; k++)
    R[i * N + j] += M1[i * N + k] * M2[k * N + j];
```

Πρόσθεση/ Αφαίρεση:

```
R[pos] = M1[pos] + M2[pos];
```

Μόλις ολοκληρωθούν όλοι οι υπολογισμοί, αντιγράφουμε τα E και F πίσω στη CPU και απελευθερώνουμε τη μνήμη που δεσμεύσαμε.

Αποδοτική υλοποίηση

Αρχικά δεσμεύουμε στη GPU τη μνήμη που απαιτείται για τα μητρώα A, B, C, D καθώς επίσης και τα αποτελέσματα E, F. Έπειτα αντιγράφουμε προς τη GPU τη μνήμη που αρχικοποιήθηκε στη CPU.

Στην προηγούμενή μας υλοποίηση προσπαθούσαμε να υπολογίσουμε, μέσω πολλαπλών kernel launches, κάθε ενδιάμεσο αποτέλεσμα που χρειαζόμαστε για να κάνουμε τις τελικές προσθέσεις. Για μια πιο αποδοτική υλοποίηση, κάθε νήμα μπορεί να αναλάβει τον πλήρη υπολογισμό μιας θέσης του τελικού αποτελέσματος, από την αρχή μέχρι το τέλος, χωρίς να χρειαστεί να τρέξουν άλλοι kernel. Ουσιαστικά, κάθε νήμα θα πρέπει να κάνει 4

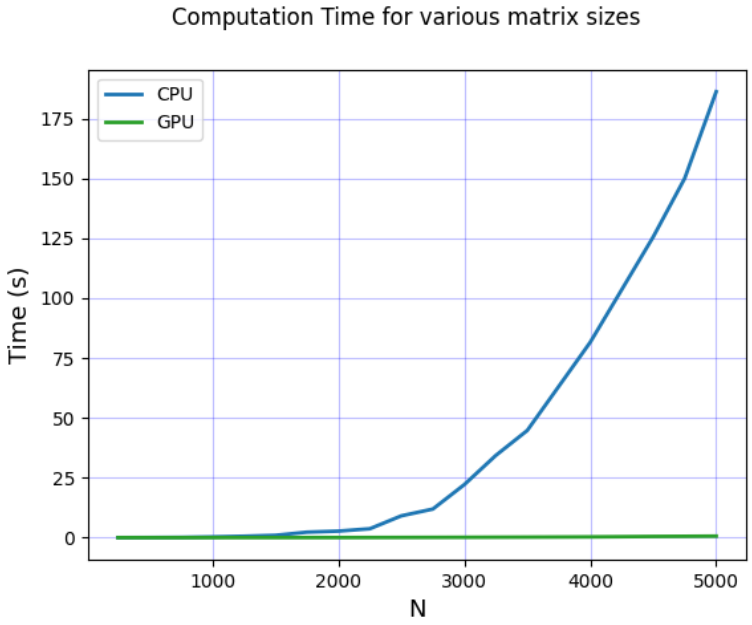
πολλαπλασιασμούς γραμμή-στήλη, μια αφαίρεση (για το E) και μια πρόσθεση (για το F). Με αυτόν τον τρόπο γλιτώνουμε το overhead πολλαπλών κλήσεων και τη μνήμη που απαιτείται για τον πλήρη υπολογισμό των ενδιάμεσων αποτελεσμάτων:

```
float resAC = 0.0f, resBD = 0.0f, resAD = 0.0f, resBC = 0.0f;

for (int k = 0; k < N; k++)
{
    resAC += A[i * N + k] * C[k * N + j];
    resBD += B[i * N + k] * D[k * N + j];
    resAD += A[i * N + k] * D[k * N + j];
    resBC += B[i * N + k] * C[k * N + j];
}

E[i*N + j] = resAC - resBD;
F[i*N + j] = resAD + resBC;
```

Πειραματική σύγκριση

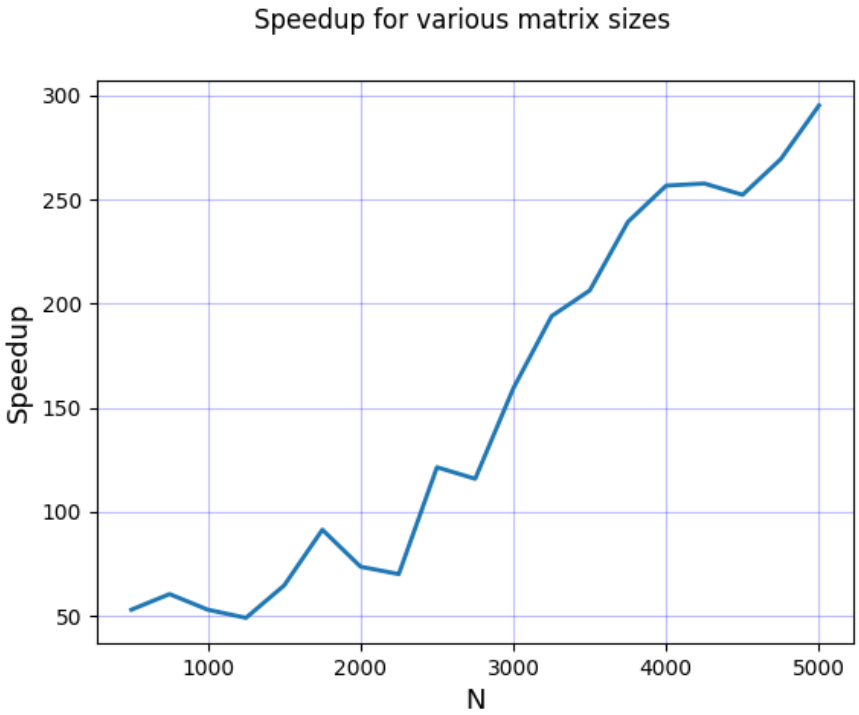
Για την αξιολόγηση της υλοποίησης μας, υλοποιήσαμε επιπλέον ένα Python script που καλεί πολλές φορές το CUDA πρόγραμμα με διαφορετικές τιμές του N. Οι χρόνοι εκτέλεσης για τις διάφορες διεργασίες (αρχικοποίηση μητρώων, πράξεις CPU, πράξεις GPU, κ.τ.λ.) στέλνονται πίσω στο script μέσω διαμοιραζόμενης μνήμης. Έπειτα, έχοντας όλους τους χρόνους για τα διαφορετικά μεγέθη, δημιουργούμε τα κατάλληλα διαγράμματα σε Python.

Αρχικά, θέλουμε να συγκρίνουμε τους χρόνους εκτέλεσης μεταξύ CPU και της αποδοτικής μας υλοποίησης στη GPU. Παρακάτω φαίνεται το διάγραμμα που συγκρίνει τους χρόνους αυτούς για διάφορες τιμές του N:

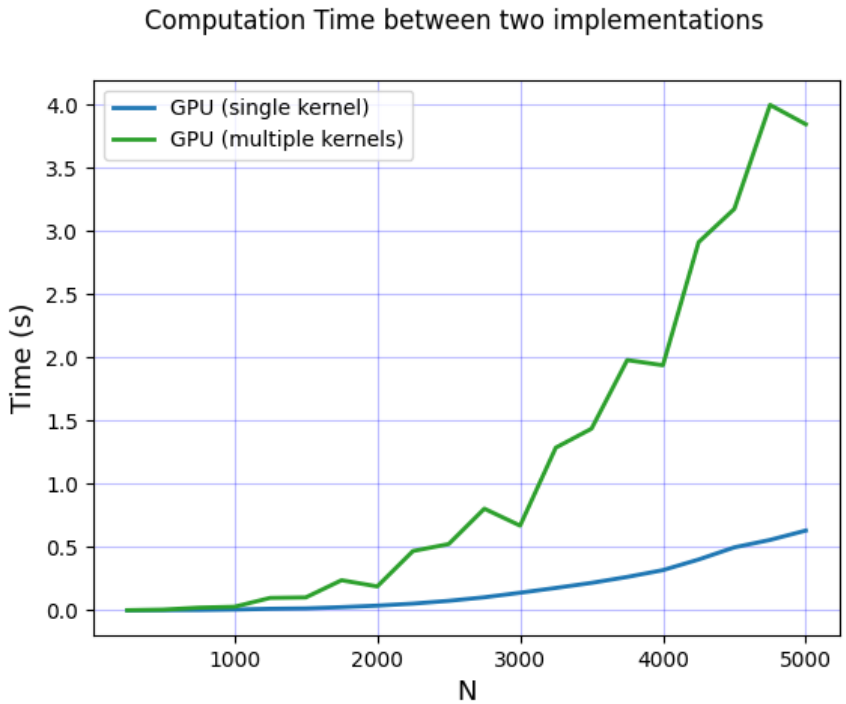


Το διάγραμμα δείχνει ξεκάθαρα ότι ο υπολογισμός στη GPU είναι δραματικά ταχύτερος από τη CPU, καθώς για τα μεγέθη μητρώων που δοκιμάσαμε, ο χρόνος της GPU μεταβάλλεται ελάχιστα, ενώ η CPU γίνεται όλο και πιο αργή. Γίνεται προφανής, επομένως, η τεράστια σημασία της χρήσης GPU σε εφαρμογές υψηλών επιδόσεων, ιδιαίτερα για υπολογισμούς μεγάλης κλίμακας που μπορούν να επωφεληθούν από μεγάλο βαθμό παραλληλίας.

Για να ποσοτικοποιήσουμε το όφελος της GPU, υπολογίζουμε την επιτάχυνση σε σχέση με τη CPU. Στο επόμενο γράφημα φαίνεται η επιτάχυνση καθώς το μέγεθος του πίνακα αυξάνεται. Παρατηρούμε ότι για μικρά μεγέθη η επιτάχυνση είναι μικρότερη, αν και αισθητή, πιθανώς λόγω του περιττού overhead της παραλληλίας για τόσο μικρά στιγμιότυπα, αλλά καθώς τα δεδομένα μεγαλώνουν, η GPU προσφέρει τεράστιο κέρδος απόδοσης:



Για το παραπάνω πρόβλημα, μπορούμε επίσης να συγκρίνουμε την απόδοση των δύο υλοποιήσεών μας, ώστε να αναδείξουμε τη σημασία της σωστής χρήσης της GPU:

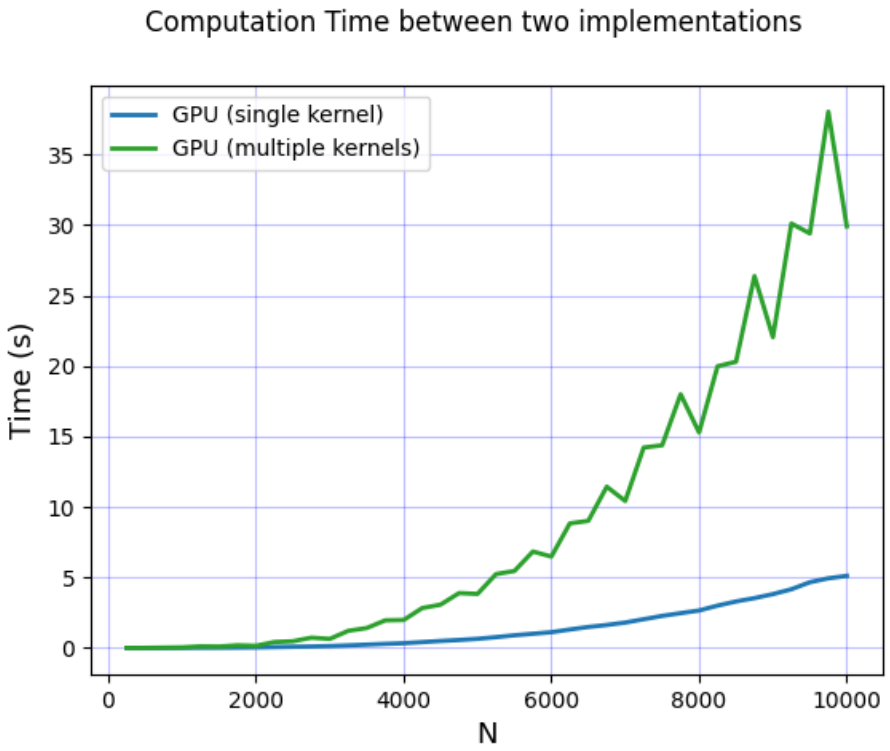


Είναι ξεκάθαρο πως η υλοποίηση σε ένα kernel launch αξιοποιεί πολύ καλύτερα την GPU, καθώς όλο το αποτέλεσμα υπολογίζεται παράλληλα από την αρχή. Όλες οι πράξεις όλων των kernel launches συγχωνεύονται σε ένα kernel launch, γλυτώνοντας έτσι τις περιττές κλήσεις.

Παρακάτω φαίνονται οι χρόνοι σε δευτερόλεπτα για όλες τις δοκιμές που εκτελέστηκαν στα παραπάνω διαγράμματα:

N	Init	CPU	GPU (naive)	GPU	Comparison
250	0,007	0,021	0,001	0	0
500	0,019	0,053	0,005	0,001	0
750	0,056	0,121	0,021	0,002	0
1000	0,062	0,318	0,028	0,006	0,001
1250	0,12	0,589	0,098	0,012	0,001
1500	0,154	0,97	0,102	0,015	0,001
1750	0,257	2,285	0,238	0,025	0,002
2000	0,322	2,725	0,189	0,037	0,002
2250	0,43	3,715	0,469	0,053	0,003
2500	0,487	9,105	0,524	0,075	0,004
2750	0,616	11,934	0,804	0,103	0,005
3000	0,716	22,151	0,67	0,139	0,006
3250	0,892	34,33	1,287	0,177	0,007
3500	1,017	44,773	1,437	0,217	0,008
3750	1,131	63,172	1,98	0,264	0,009
4000	1,282	81,615	1,939	0,318	0,02
4250	1,518	103,575	2,913	0,402	0,011
4500	1,675	125,615	3,176	0,498	0,013
4750	1,835	150,056	3,999	0,557	0,014
5000	2,11	186,235	3,846	0,631	0,019

Για μεγέθη N μεγαλύτερα του 5000 γίνεται ιδιαίτερα δύσκολο να συνεχίσουμε να συγκρίνουμε την απόδοση της GPU με τη CPU, καθώς οι χρόνοι CPU γίνονται υπερβολικά μεγάλοι. Μπορούμε, ωστόσο, να συνεχίσουμε να συγκρίνουμε τις δύο υλοποιήσεις μας στη GPU. Παρακάτω φαίνεται διάγραμμα των χρόνων εκτέλεσης στη GPU για μέγεθος πίνακα μέχρι 10.000, όπου για τόσο μεγάλα στιγμιότυπα η διαφορά μεταξύ των δύο υλοποιήσεων είναι ακόμα πιο αισθητή:



Παρακάτω φαίνονται σε αριθμητική μορφή οι χρόνοι του διαγράμματος σε δευτερόλεπτα:

N	Init	GPU (naive)	GPU	N	Init	GPU (naive)	GPU
250	0,008	0,001	0	**5250**	2,27	5,238	0,765
500	0,02	0,005	0,001	**5500**	2,463	5,459	0,895
750	0,037	0,02	0,002	**5750**	2,812	6,839	1,006
1000	0,079	0,034	0,006	**6000**	3,047	6,481	1,114
1250	0,115	0,113	0,011	**6250**	3,127	8,834	1,307
1500	0,176	0,098	0,015	**6500**	3,554	9,015	1,485
1750	0,252	0,186	0,023	**6750**	5,351	11,439	1,628
2000	0,359	0,151	0,036	**7000**	4,139	10,424	1,8
2250	0,4	0,42	0,062	**7250**	5,67	14,215	2,036
2500	0,537	0,474	0,083	**7500**	4,765	14,375	2,272
2750	0,582	0,726	0,102	**7750**	4,904	17,995	2,466
3000	0,641	0,642	0,132	**8000**	5,075	15,285	2,655
3250	0,889	1,214	0,182	**8250**	5,688	19,981	3,002
3500	1,036	1,412	0,233	**8500**	6,025	20,302	3,296
3750	1,08	1,956	0,287	**8750**	6,438	26,379	3,534
4000	1,412	1,982	0,333	**9000**	6,739	22,048	3,818
4250	1,353	2,826	0,414	**9250**	7,276	30,108	4,163
4500	1,72	3,072	0,494	**9500**	9,188	29,404	4,66
4750	2,003	3,891	0,561	**9750**	7,838	38,04	4,94
5000	2,125	3,833	0,645	**10000**	8,345	29,911	5,111

Περιβάλλον Υλοποίησης

Παραθέτουμε πληροφορίες για το περιβάλλον όπου υλοποιήθηκαν οι λύσεις της εργαστηριακής άσκησης:

Χαρακτηριστικό	**Τιμή**
Operating System	Microsoft Windows 11 Home (x64), Version 24H2, Build 26100.3037 WSL (Windows Subsystem for Linux): Ubuntu 24.04 LTS
CPU model	AMD Ryzen 7 8845HS @ 3.8 GHz
Physical CPU cores	8
Logical CPU cores	16
RAM	32 GB DDR5
GPU	NVIDIA GeForce RTX 4060 Laptop GPU
VRAM	8188 MB
L1 data cache	8 x 32 KB (8-way, 64-byte line)
L2 cache	8 x 1024 KB (8-way, 64-byte line)
L3 cache	16 MB (16-way, 64-byte line)
Instruction Sets	MMX (+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AES, AVX, AVX2, AVX512 (DQ, BW, VL, CD, IFMA, VBMI, VBMI2, VNNI, BITALG, VPOPCNTDQ, BF16), FMA3, SHA