Λογισμικό και προγραμματισμός υψηλής επίδοσης

Εργασία 2

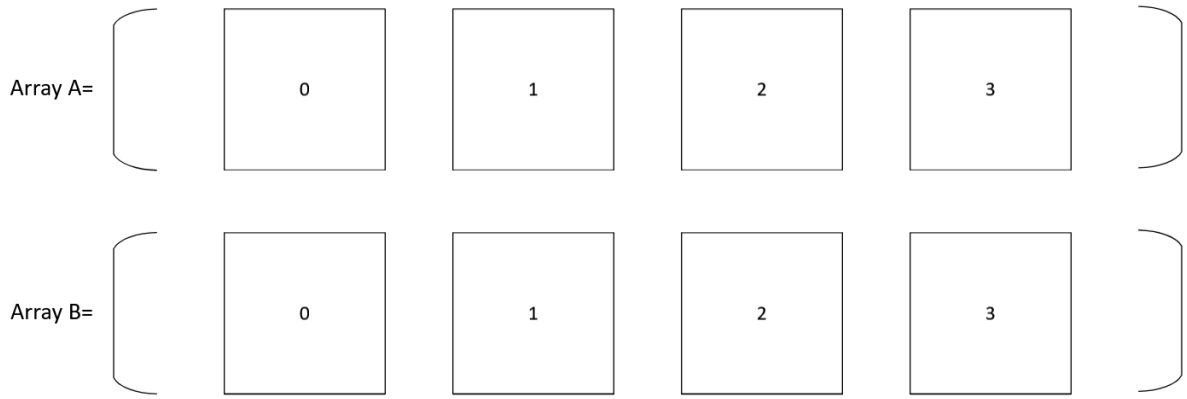
Περιεχόμενα

Question 1

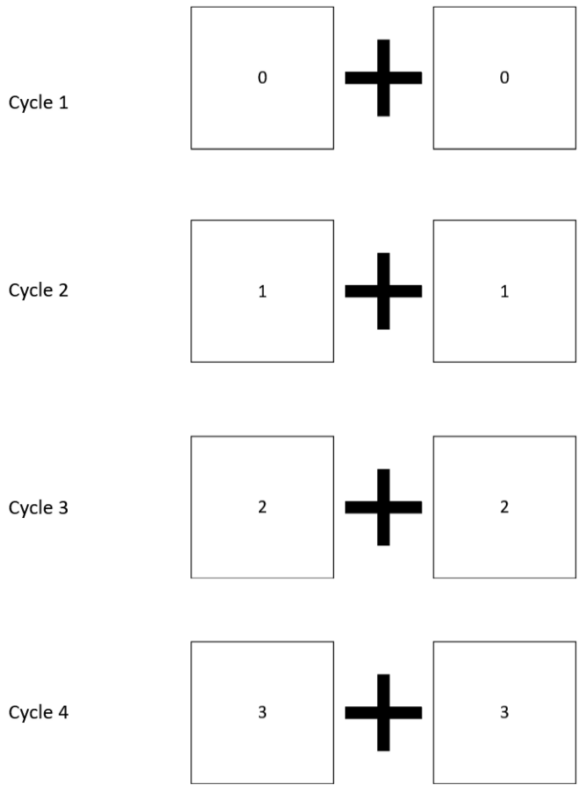
α)

Με όλες τις υλοποιήσεις σκοπεύουμε να χρησιμοποιήσουμε simd για την επιτάχυνση των υπολογισμών. Για να κατανοήσουμε την λειτουργία που θέλουμε να κάνουμε, ας δούμε ένα παράδειγμα:

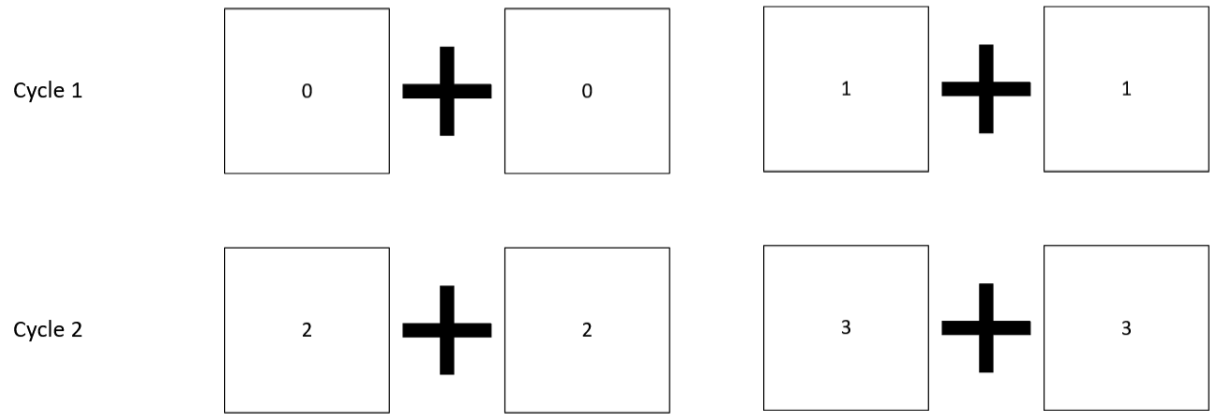
Έστω ότι έχουμε 2 array A,B με 4 ίδια στοιχεία το καθένα 0, 1, 2, 3 και θέλουμε να τα προσθέσουμε. Κανονικά, σε έναν κύκλο cpu κάνουμε πρώτα την πρόσθεση των 2 στοιχείων 0+0, στον επόμενο κάνουμε των στοιχείων 1+1 κλπ. Με την χρήση simd, μπορούμε μέσα σε 1 cpu cycle να κάνουμε πολλαπλές πράξεις μαζί, όπως 0+0 και 1+1, αρά επιταχύνουμε την διαδικασία χωρίς να χρειαστούμε παραπάνω επεξεργαστικούς πόρους.



Εικόνα 1: Δείχνει τα αρχικά array A, B.



Εικόνα 2: Κλασσικός τρόπος πράξεων, 1 ανά κύκλο ρολογιού.



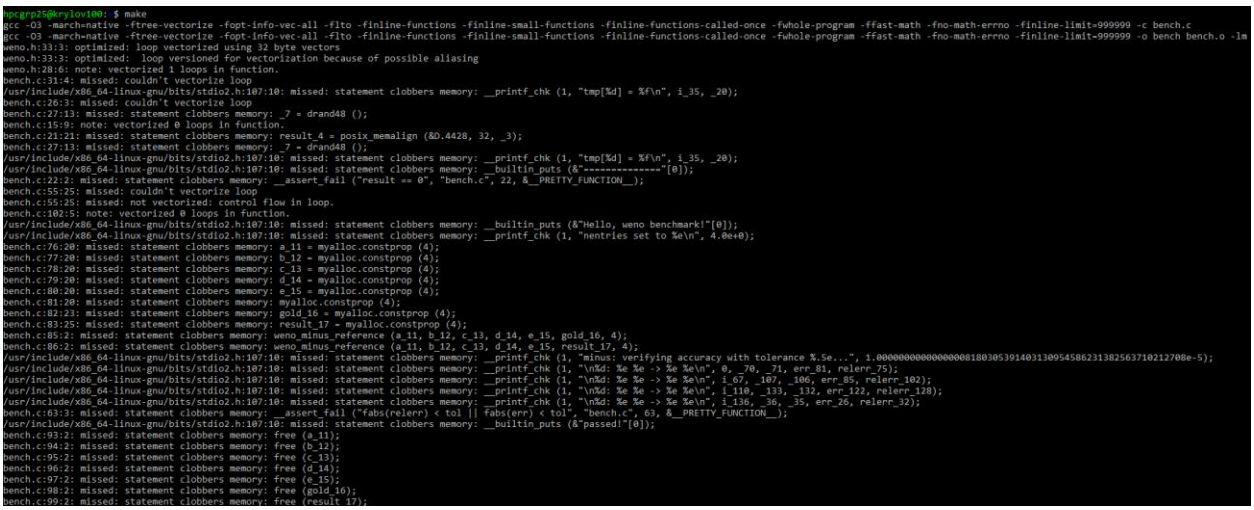
Εικόνα 3: Simd πράξεις, πολλαπλές ανά κύκλο ρολογιού.

Automatic vectorization

Για να ενεργοποιήσουμε το automatic vectorization, αρκεί να χρησιμοποιήσουμε τα κατάλληλα flag. Αναλυτικότερα, θέτουμε το μέγιστο δυνατό optimization (level 3) επιτρέποντας inlining, loop unrolling και vectorization. Επιπλέον, φροντίζουμε με το native ότι χρησιμοποιείτε η βέλτιστη τεχνική για την αρχιτεκτονική μας (ο xeon του server συγκεκριμένα είναι x86) και αφήνουμε την διαδικασία αυτόματα στον compiler με την ftree vectorize.

```
CC = gcc

# Best optimization, inlining included
CFLAGS = -O3 -march=native -ftree-vectorize -fopt-info-vec-all \
         -flto -finline-functions -finline-small-functions \
         -finline-functions-called-once -fwhole-program \
         -ffast-math -fno-math-errno \
         -finline-limit=999999

LIBS = -lm

all: bench

bench: bench.o
     $(CC) $(CFLAGS) -o $@ $^ $(LIBS)

# Compile the bench
bench.o: bench.c weno.h
     $(CC) $(CFLAGS) -c bench.c

clean:
     rm -f *.o bench
```

Το αποτέλεσμα που λαμβάνουμε είναι:



Μπορούμε να διακρίνουμε ότι το κύριο loop που κάνει τις πράξεις (το weno minus reference) έγινε vectorized με επιτυχία. Τα υπόλοιπα error είναι από τις πράξεις μνήμης και i/o διαδικασιών, τα οποία δεν γίνονται vectorize άρα ο compiler μας τα δείχνει ως σφάλμα, ενώ στην πραγματικότητα η διαδικασία μας είναι σωστή. Αξίζει να σημειωθεί ότι ο compiler δεν έκανε vectorize το loop μας χωρίς την ffast και το inline, μάλλον επειδή θεωρούσε ότι θα προκαλέσει σφάλμα η' πως είναι ασύμφορο να γίνει. Το συμπέρασμα είναι ότι προφανώς η αυτόματη διαδικασία δεν είναι η βέλτιστη.

Omp vectorization

Οι αλλαγές που χρειάζονται είναι ελάχιστες, αρκεί να προσθέσουμε στην εντολή omp parralel for το simd και επιπλέον επιλέξαμε να κάνουμε inline την συνάρτηση για να μην έχουμε overhead, δηλαδή αντί να την καλούμε απλά γράψαμε κατευθείαν τις πράξεις μέσα στο loop.

```
#pragma once

#include <omp.h>
#include "weno.h"

void weno_minus_openmp(const float * const a, const float * const b,
const float * const c,
                    const float * const d, const float * const e, float
* const out,
                    const int NENTRIES)
{
    #pragma omp parallel for simd
    for(int i = 0; i < NENTRIES; ++i) {
        const float is0 = a[i]*(a[i]*(float)(4./3.)  -
```

```
b[i]*(float)(19./3.)  + c[i]*(float)(11./3.)) +
                        b[i]*(b[i]*(float)(25./3.)  -
c[i]*(float)(31./3.)) +
                        c[i]*c[i]*(float)(10./3.);

        const float is1 = b[i]*(b[i]*(float)(4./3.)  -
c[i]*(float)(13./3.)  + d[i]*(float)(5./3.))  +
                        c[i]*(c[i]*(float)(13./3.)  -
d[i]*(float)(13./3.)) +
                        d[i]*d[i]*(float)(4./3.);

        const float is2 = c[i]*(c[i]*(float)(10./3.) -
d[i]*(float)(31./3.)  + e[i]*(float)(11./3.)) +
                        d[i]*(d[i]*(float)(25./3.)  -
e[i]*(float)(19./3.)) +
                        e[i]*e[i]*(float)(4./3.);

        const float is0plus = is0 + (float)WENOEPS;
        const float is1plus = is1 + (float)WENOEPS;
        const float is2plus = is2 + (float)WENOEPS;

        const float alpha0 = (float)(0.1)/((float)(is0plus*is0plus));
        const float alpha1 = (float)(0.6)/((float)(is1plus*is1plus));
        const float alpha2 = (float)(0.3)/((float)(is2plus*is2plus));

        const float alphasum = alpha0 + alpha1 + alpha2;
        const float inv_alpha = (float)1/alphasum;

        const float omega0 = alpha0 * inv_alpha;
        const float omega1 = alpha1 * inv_alpha;
        const float omega2 = (float)1 - omega0 - omega1;

        out[i] = omega0*((float)(1.0/3.)*a[i] - (float)(7./6.)*b[i] +
(float)(11./6.)*c[i]) +
                 omega1*(-(float)(1./6.)*b[i] + (float)(5./6.)*c[i] +
(float)(1./3.)*d[i]) +
                 omega2*((float)(1./3.)*c[i]  + (float)(5./6.)*d[i] -
(float)(1./6.)*e[i]);
    }
}
```

Avx vectorization

Χρησιμοποιούμε εντολές για 256bit register, άρα μπορούμε να κάνουμε 8 πράξεις ταυτόχρονα. Αρχικά, πριν από τους υπολογισμούς φορτώνουμε στους καταχωρητες τις τιμές των σταθερών μας ώστε να είναι έτοιμες για χρήση.

```
// initialize
    const __m256 v_4o3  = _mm256_set1_ps(4.0f/3.0f);
    const __m256 v_19o3 = _mm256_set1_ps(19.0f/3.0f);
    const __m256 v_11o3 = _mm256_set1_ps(11.0f/3.0f);
    const __m256 v_25o3 = _mm256_set1_ps(25.0f/3.0f);
    const __m256 v_31o3 = _mm256_set1_ps(31.0f/3.0f);
    const __m256 v_10o3 = _mm256_set1_ps(10.0f/3.0f);
    const __m256 v_13o3 = _mm256_set1_ps(13.0f/3.0f);
    const __m256 v_5o3  = _mm256_set1_ps(5.0f/3.0f);
    const __m256 v_eps  = _mm256_set1_ps(WENOEPS);
    const __m256 v_one  = _mm256_set1_ps(1.0f);
    const __m256 v_alpha0 = _mm256_set1_ps(0.1f);
    const __m256 v_alpha1 = _mm256_set1_ps(0.6f);
    const __m256 v_alpha2 = _mm256_set1_ps(0.3f);
```

Συνεχίζοντας, υλοποιούμε ένα loop ώστε να παίρνουμε τα δεδομένα από τα array στους καταχωρητες, συγκεκριμένα 8 στοιχεία κάθε φορά και κάνουμε vectorize τις πράξεις για τα stencil.

```
#pragma omp parallel for
    for (int i = 0; i < NENTRIES; i += 8) {
        // Load all data
        __m256 va = _mm256_loadu_ps(&a[i]);
        __m256 vb = _mm256_loadu_ps(&b[i]);
        __m256 vc = _mm256_loadu_ps(&c[i]);
        __m256 vd = _mm256_loadu_ps(&d[i]);
        __m256 ve = _mm256_loadu_ps(&e[i]);

// Calculations
        __m256 is0 = _mm256_mul_ps(va, _mm256_mul_ps(va, v_4o3));
        is0 = _mm256_sub_ps(is0, _mm256_mul_ps(va, _mm256_mul_ps(vb,
v_19o3)));
        is0 = _mm256_add_ps(is0, _mm256_mul_ps(va, _mm256_mul_ps(vc,
v_11o3)));
        __m256 tmp = _mm256_mul_ps(vb, _mm256_mul_ps(vb, v_25o3));
        tmp = _mm256_sub_ps(tmp, _mm256_mul_ps(vb, _mm256_mul_ps(vc,
v_31o3)));
        is0 = _mm256_add_ps(is0, tmp);
        is0 = _mm256_add_ps(is0, _mm256_mul_ps(_mm256_mul_ps(vc, vc),
v_10o3));


        __m256 is1 = _mm256_mul_ps(vb, _mm256_mul_ps(vb, v_4o3));
        is1 = _mm256_sub_ps(is1, _mm256_mul_ps(vb, _mm256_mul_ps(vc,
v_13o3)));
        is1 = _mm256_add_ps(is1, _mm256_mul_ps(vb, _mm256_mul_ps(vd,
v_5o3)));
```

```
        tmp = _mm256_mul_ps(vc, _mm256_mul_ps(vc, v_13o3));
        tmp = _mm256_sub_ps(tmp, _mm256_mul_ps(vc, _mm256_mul_ps(vd,
v_13o3)));
        is1 = _mm256_add_ps(is1, tmp);
        is1 = _mm256_add_ps(is1, _mm256_mul_ps(_mm256_mul_ps(vd, vd),
v_4o3));


        __m256 is2 = _mm256_mul_ps(vc, _mm256_mul_ps(vc, v_10o3));
        is2 = _mm256_sub_ps(is2, _mm256_mul_ps(vc, _mm256_mul_ps(vd,
v_31o3)));
        is2 = _mm256_add_ps(is2, _mm256_mul_ps(vc, _mm256_mul_ps(ve,
v_11o3)));
        tmp = _mm256_mul_ps(vd, _mm256_mul_ps(vd, v_25o3));
        tmp = _mm256_sub_ps(tmp, _mm256_mul_ps(vd, _mm256_mul_ps(ve,
v_19o3)));
        is2 = _mm256_add_ps(is2, tmp);
        is2 = _mm256_add_ps(is2, _mm256_mul_ps(_mm256_mul_ps(ve, ve),
v_4o3));
```

Παρακάτω, για να αποφευχθεί η αριθμητική αστάθεια κατά τη διάρκεια υπολογισμού των βαρών, στους δείκτες ομαλότητας προστίθεται μια μικρή σταθερά (η epsilon). Με τον τρόπο αυτό, αποφεύγονται πιθανά σφάλματα διαίρεσης με το μηδέν και εξασφαλίζουμε ότι το πρόγραμμα παραμένει σταθερό κατά τη διάρκεια της εκτέλεσης. Μόλις προστεθεί το epsilon, οι δείκτες ομαλότητας τετραγωνίζονται ως μέρος της διαδικασίας κανονικοποίησης.

```
// Add epsilon and compute squares
        is0 = _mm256_add_ps(is0, v_eps);
        is1 = _mm256_add_ps(is1, v_eps);
        is2 = _mm256_add_ps(is2, v_eps);
        is0 = _mm256_mul_ps(is0, is0);
        is1 = _mm256_mul_ps(is1, is1);
        is2 = _mm256_mul_ps(is2, is2);


        is0 = _mm256_div_ps(v_alpha0, is0);
        is1 = _mm256_div_ps(v_alpha1, is1);
        is2 = _mm256_div_ps(v_alpha2, is2);


        __m256 alpha_sum = _mm256_add_ps(_mm256_add_ps(is0, is1),
is2);
        __m256 inv_sum = _mm256_div_ps(v_one, alpha_sum);
```

Τέλος, υπολογίζουμε τα βάρη και κάνουμε τους τελικούς υπολογισμούς.

```
__m256 omega0 = _mm256_mul_ps(is0, inv_sum);
        __m256 omega1 = _mm256_mul_ps(is1, inv_sum);
        __m256 omega2 = _mm256_sub_ps(v_one, _mm256_add_ps(omega0,
omega1));

        // Final reconstruction
        __m256 u0 = _mm256_mul_ps(_mm256_set1_ps(1.0f/3.0f), va);
        u0 = _mm256_sub_ps(u0,
_mm256_mul_ps(_mm256_set1_ps(7.0f/6.0f), vb));
        u0 = _mm256_add_ps(u0,
_mm256_mul_ps(_mm256_set1_ps(11.0f/6.0f), vc));

        __m256 u1 = _mm256_mul_ps(_mm256_set1_ps(-1.0f/6.0f), vb);
        u1 = _mm256_add_ps(u1,
_mm256_mul_ps(_mm256_set1_ps(5.0f/6.0f), vc));
        u1 = _mm256_add_ps(u1,
_mm256_mul_ps(_mm256_set1_ps(1.0f/3.0f), vd));

        __m256 u2 = _mm256_mul_ps(_mm256_set1_ps(1.0f/3.0f), vc);
        u2 = _mm256_add_ps(u2,
_mm256_mul_ps(_mm256_set1_ps(5.0f/6.0f), vd));
        u2 = _mm256_sub_ps(u2,
_mm256_mul_ps(_mm256_set1_ps(1.0f/6.0f), ve));

        // Combine results
        __m256 result = _mm256_mul_ps(omega0, u0);
        result = _mm256_add_ps(result, _mm256_mul_ps(omega1, u1));
        result = _mm256_add_ps(result, _mm256_mul_ps(omega2, u2));

        _mm256_storeu_ps(&out[i], result);
```

β)

Αρχικά, για όλες τις υλοποιήσεις απενεργοποιούμε το debug mode του αρχικού bench ώστε να έχουμε το κανονικό dataset. Σε αυτό παρατηρούμε ότι περιλαμβάνει 2 διαφορετικά, το peak like και το stream like όπου το πρώτο δοκιμάζει την απόδοση δεδομένων που βρίσκονται στην cache (είναι περίπου 32kb, άρα χωράει στην cache) και το δεύτερο δοκιμάζει την απόδοση για δεδομένα που βρίσκονται στον δίσκο (είναι 512mb). Η κάθε δοκιμή υλοποιείτε 4 φορές, άρα μπορούμε να δούμε ανά μέσο όρο αν η υλοποίηση μας είναι αποδοτική. Όλα τα παραπάνω λεγόμενα βρίσκονται εντός του αρχικού αρχείου bench.

```
printf("Hello, weno benchmark!\n");
     const int debug = 1;
```

```
    if (debug)
    {
        benchmark(argc, argv, 4, 1, 1, "debug");
        return 0;
    }
```

Omp bench

Για την omp υλοποίηση, κάνουμε πλέον reference το κατάλληλο αρχείο ώστε να τρέξουν οι μετρήσεις και επιπλέον μετράμε τον χρόνο εκτέλεσης. Στην συνέχεια, καλούμε το αρχικό bench και μετράμε πάλι τον χρόνο εκτέλεσης ενώ ταυτόχρονο συγκρίνουμε τις τιμές των αποτελεσμάτων μας στις πράξεις. Εφόσον έχουν μικρό σφάλμα, τότε θεωρούμε ότι η διαδικασία ολοκληρώθηκε με επιτυχία.

```
// Run original
    double t1 = get_wtime();
    weno_minus_reference(a, b, c, d, e, gold, NENTRIES);
    double t2 = get_wtime();
    printf("Reference time: %f seconds\n", t2-t1);

    // Run OMP
    t1 = get_wtime();
    weno_minus_openmp(a, b, c, d, e, result, NENTRIES);
    t2 = get_wtime();
    printf("OpenMP time: %f seconds\n", t2-t1);

    const double tol = 1e-5;
    printf("Verifying accuracy with tolerance %.5e...", tol)
```

Παρατηρούμε τα εξής:

Peak like:

Run	Αρχικός κώδικας	Omp	Απόδοση
0	0.002506	0.034104	Omp πιο αργό
1	0.002870	0.004798	Omp 1.7x πιο γρήγορο
2	0.002876	0.002380	Omp 1.2x πιο γρήγορο
3	0.002864	0.000071	Omp 40x πιο γρήγορο

Stream like:

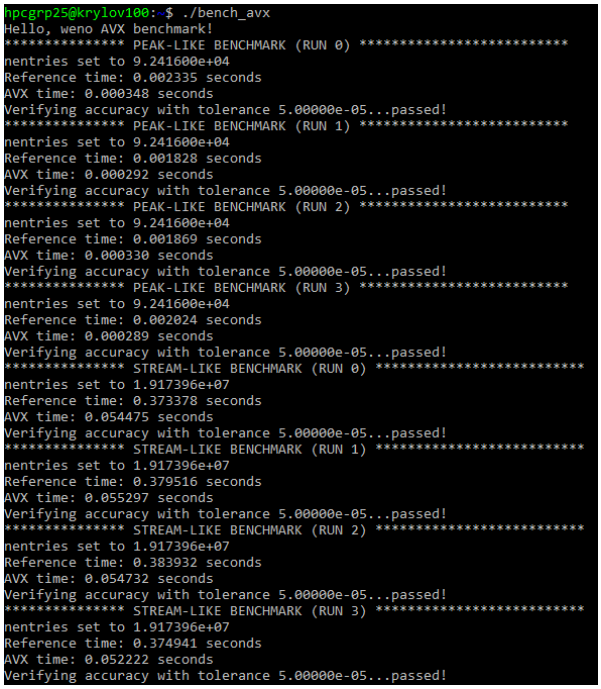
Run	Αρχικός κώδικας	Omp	Απόδοση
0	0.538961	0.027125	Omp 20x πιο γρήγορο
1	0.371902	0.020740	Omp 18x πιο γρήγορο
2	0.368252	0.016525	Omp 22x πιο γρήγορο
3	0.455793	0.020485	Omp 22x πιο γρήγορο

Από τα αποτελέσματα καταλαβαίνουμε ότι η omp υλοποίηση για cache δεν επιφέρει σταθερή αύξηση απόδοσης, αντιθέτως μπορεί να είναι και πιο αργή από την σειριακή. Αυτό συμβαίνει λόγο του overhead δημιουργίας νημάτων και ανάθεσης εργασιών σε αυτά, καθώς και από τον os scheduler την συγκεκριμένη χρονική στιγμή. Επαναλαμβάνοντας το πείραμα πολλαπλές φορές, βρήκαμε ότι ανά μέσο όρο το speedup είναι οριακά καλύτερο από την σειριακή έκδοση. Αντιθέτως, παρατηρούμε σταθερή αύξηση απόδοσης στην περίπτωση των δεδομένων στην μνήμη. Η σειριακή έκδοση πρέπει να περιμένει τα δεδομένα από την μνήμη, ενώ η παράλληλη κάνει το ίδιο αλλά με πολλαπλά thread και συνεπώς δεν περιορίζεται τόσο (δηλαδή αξιοποιεί περισσότερο bandwidth της μνήμης). Επιπλέον, αφού η ταχύτητα της μνήμης είναι σαφώς πιο αργή από της cache, το overhead μας επηρεάζει ελάχιστα.

Avx bench

Είναι ακριβώς το ίδιο με το omp, η μόνη αλλαγή είναι ότι αυξήσαμε ελάχιστα το περιθώριο σφάλματος στον έλεγχο.

```
// Run original
    double t1 = get_wtime();
    weno_minus_reference(a, b, c, d, e, gold, NENTRIES);
    double t2 = get_wtime();
    printf("Reference time: %f seconds\n", t2-t1);

    // Run AVX
    t1 = get_wtime();
    weno_avx(a, b, c, d, e, result, NENTRIES);
    t2 = get_wtime();
    printf("AVX time: %f seconds\n", t2-t1);

    const double tol = 5e-5;
    printf("Verifying accuracy with tolerance %.5e...", tol);
    check_error(tol, gold, result, NENTRIES);
    printf("passed!\n");
```



Παρατηρούμε τα εξής:

Peak like:

Run	Αρχικός κώδικας	Avx	Απόδοση
0	0.002335	0.000348	Avx 6.7x πιο γρήγορο
1	0.001828	0.000292	Avx 6.3x πιο γρήγορο
2	0.001869	0.000330	Avx 5.7x πιο γρήγορο
3	0.002024	0.000289	Avx 7.0x πιο γρήγορο

Stream like:

Run	Αρχικός κώδικας	Avx	Απόδοση
0	0.373378	0.054475	Avx 6.9x πιο γρήγορο
1	0.379516	0.055297	Avx 6.9x πιο γρήγορο
2	0.383932	0.054732	Avx 7.0x πιο γρήγορο
3	0.374941	0.052222	Avx 7.2x πιο γρήγορο

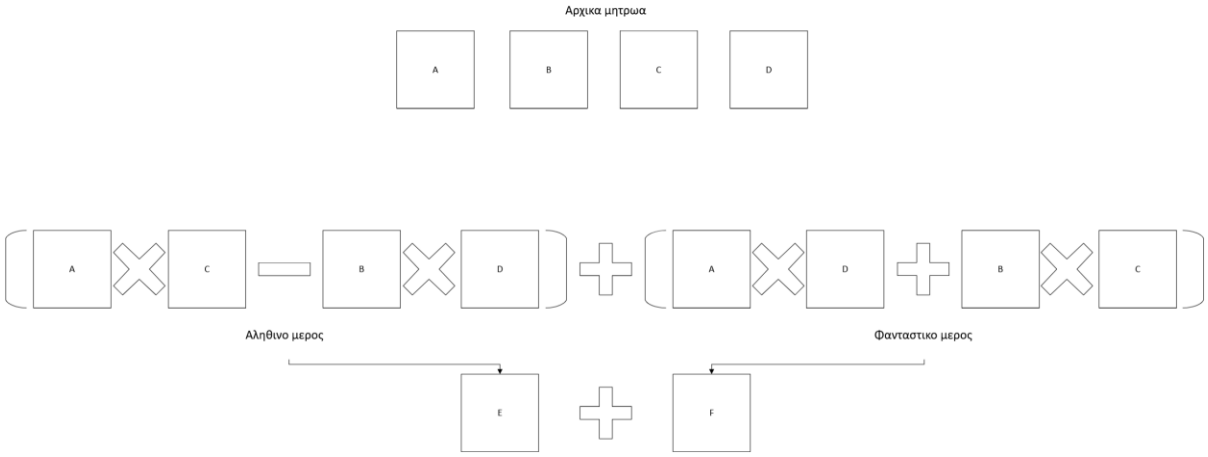
Και για τα 2 test, έχει σταθερή αύξηση απόδοσης. Καταλαβαίνουμε ότι το vectorization δεν υποφέρει από θέματα scheduler και overhead όπως το omp, άρα ακόμα και στην cache προσφέρει σταθερή απόδοση. Στην περίπτωση της

μνήμης, επωφελείται από το γεγονός ότι κάνει 8 πράξεις ανά κύκλο αντί για 1. Είναι αρκετά πιο αργό από το omp για τον απλό λόγο ότι παραμένει single threaded, άρα είναι φυσικά αδύνατο να έχει το ίδιο throughput. Συμπερασματικά, η βέλτιστη υλοποίηση θα ήταν ο συνδυασμός omp και avx για να πέτυχουμε το μέγιστο δυνατό speedup και στις 2 περιπτώσεις.

Question 2

a)

Για την συγκεκριμένη ερώτηση, ουσιαστικά απλά ακολουθούμε κανονικές πράξεις με μητρώα. Από την σχέση (1) που δίνεται, παρατηρούμε πως υπολογίζεται το πραγματικό και το φανταστικό μέρος.



Εικόνα 4: Υπολογισμός σύνθετων μητρώων.

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define N 1024

#define cudaCheckError() {                                          \
    cudaError_t e=cudaGetLastError();                               \
    if(e!=cudaSuccess) {                                            \
        printf("CUDA Error %s:%d: %s\n", __FILE__, __LINE__,        \
                cudaGetErrorString(e));                             \
        exit(EXIT_FAILURE);                                         \
    }                                                               \
}
```

```
// Total run time
double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
}

// define to which matrixes to write to
__global__ void complexMatMulKernel(
    const float* A, const float* B,
    const float* C, const float* D,
    float* E, float* F,
    int n)
{
// Assigning each thread an element to calculate
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

// Check that its inside the matrix
    if (row < n && col < n)
    {
// Initialize to zero before calculations
        float sumAC = 0.0f;
        float sumBD = 0.0f;
        float sumAD = 0.0f;
        float sumBC = 0.0f;

        for (int k = 0; k < n; ++k)
        {
// retrieve values
            float a = A[row * n + k];
            float b = B[row * n + k];
            float c = C[k * n + col];
            float d = D[k * n + col];

// Calculate A,B,C,D
            sumAC += a * c;
            sumBD += b * d;
            sumAD += a * d;
            sumBC += b * c;
        }

// Calculate and store in E,F
        E[row * n + col] = sumAC - sumBD;
        F[row * n + col] = sumAD + sumBC;
    }
}
```

```
// Random matrix values
void initializeMatrix(float* mat, int n)
{
    for (int i = 0; i < n * n; ++i)
    {
        mat[i] = static_cast<float>(rand()) / RAND_MAX;
    }
}

int main()
{
    double start_time = get_time();

    srand(time(NULL));
    size_t size = N * N * sizeof(float);

// Starting in cpu before transfering to gpu
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);
    float* h_D = (float*)malloc(size);
    float* h_E = (float*)malloc(size);
    float* h_F = (float*)malloc(size);

    initializeMatrix(h_A, N);
    initializeMatrix(h_B, N);
    initializeMatrix(h_C, N);
    initializeMatrix(h_D, N);

// Tranfering to gpu
    float *d_A, *d_B, *d_C, *d_D, *d_E, *d_F;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);
    cudaMalloc((void**)&d_D, size);
    cudaMalloc((void**)&d_E, size);
    cudaMalloc((void**)&d_F, size);
    cudaCheckError();

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_D, h_D, size, cudaMemcpyHostToDevice);
    cudaCheckError();

// Block size
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
```

```
                  (N + dimBlock.y - 1) / dimBlock.y);

    complexMatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, d_D,
d_E, d_F, N);
    cudaCheckError();
    cudaDeviceSynchronize();
    cudaCheckError();

    cudaMemcpy(h_E, d_E, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_F, d_F, size, cudaMemcpyDeviceToHost);
    cudaCheckError();

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_D);
    cudaFree(d_E);
    cudaFree(d_F);
    cudaCheckError();

    free(h_A);
    free(h_B);
    free(h_C);
    free(h_D);
    free(h_E);
    free(h_F);

    double end_time = get_time();
    printf("Matrix size: %d x %d\n", N, N);
    printf("Total execution time: %.4f seconds\n", (end_time -
start_time) / 1000000.0);

    return 0;
}
```

β)

Αρχικά, η όλη λογική των υπολογισμών γίνεται με την συνάρτηση complexMatMulKernel. Ορίζουμε σε ποια μητρώα θέλουμε να γράψουμε και αναθέτουμε σε κάθε thread το αντίστοιχο στοιχείο που είναι υπεύθυνο να υπολογίσει. Συνεχίζοντας, ελέγχουμε ότι είμαστε εντός των διαστάσεων του μητρώου (δηλαδή έχουμε ακόμα στοιχεία να υπολογίσουμε) και αρχικοποιουμε τις μεταβλητές όπου θα αποθηκεύουμε τα αποτελέσματα των πράξεων. Επιπλέον, τώρα πρέπει να ανακτήσουμε τις πραγματικές τιμές των στοιχείων από τις αντίστοιχες θέσεις και υπολογίζουμε τα γινόμενα των συνδυασμών μητρώων που έχουμε (AC, BD, AD, BC) ώστε να υπολογίσουμε τα τελικά μητρώα E και F χρησιμοποιώντας την σχέση 1 της εκφώνησης. Το σημαντικότερο

κομμάτι είναι η αρχικοποίηση των μητρώων στην cpu (δηλαδή στον Host) καθώς αυτή είναι υπεύθυνη για την διαχείριση της gpu και γενικότερα της διαχείρισης όλης της διαδικασίας. Αφού γίνει αυτό με επιτυχία, τότε μόνο γίνεται να μεταφέρουμε τα δεδομένα στην gpu (device) ώστε τελικά να κάνει τους υπολογισμούς, δηλαδή να τρέξει την συνάρτηση που αναφέρθηκε παραπάνω. Τέλος, χωρίζουμε το αρχικό πρόβλημα (στην συγκεκριμένη περίπτωση τον υπολογισμό μητρώων διαστάσεων NxN=1024x1024) σε 32x32 block, δηλαδή σε 1024 thread/block καθώς αυτό είναι το μέγιστο που υποστηρίζει η v100. Μέσω διάφορων πειραμάτων, διαπιστώσαμε ότι έχει ελάχιστη διαφορά με 16x16 block size για σχετικά μικρές διαστάσεις (κάτω από 4096x4096) αλλά σχετικά εμφανή σε μεγαλύτερες.

γ)

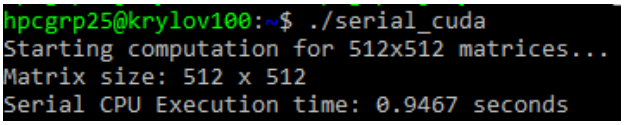
Για την σειριακή έκδοση, υλοποιούμε ακριβώς τον ίδιο κώδικα απλά χωρίς το cuda μέρος, οι πράξεις είναι ιδίες.

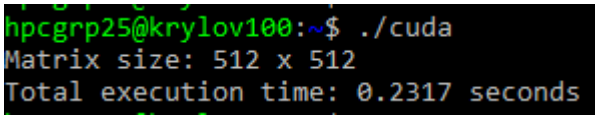
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define N 1024

double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
}

void initializeMatrix(float* mat, int n) {
    for (int i = 0; i < n * n; ++i) {
        mat[i] = (float)rand() / RAND_MAX;
    }
}

void complex_matrix_multiply(const float* A, const float* B, const
float* C, const float* D,
                             float* E, float* F, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            float sumAC = 0.0f;
            float sumBD = 0.0f;
            float sumAD = 0.0f;
            float sumBC = 0.0f;

            for (int k = 0; k < n; k++) {
                float a = A[i * n + k];
```

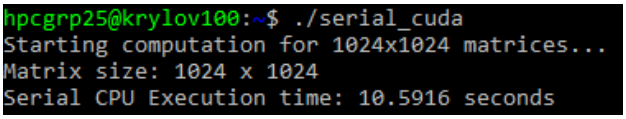
```
                float b = B[i * n + k];
                float c = C[k * n + j];
                float d = D[k * n + j];

                sumAC += a * c;
                sumBD += b * d;
                sumAD += a * d;
                sumBC += b * c;
            }

            E[i * n + j] = sumAC - sumBD;
            F[i * n + j] = sumAD + sumBC;
        }
    }
}

int main() {
    srand(time(NULL));
    size_t size = N * N * sizeof(float);

    float* A = (float*)malloc(size);
    float* B = (float*)malloc(size);
    float* C = (float*)malloc(size);
    float* D = (float*)malloc(size);
    float* E = (float*)malloc(size);
    float* F = (float*)malloc(size);

    if (!A || !B || !C || !D || !E || !F) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    initializeMatrix(A, N);
    initializeMatrix(B, N);
    initializeMatrix(C, N);
    initializeMatrix(D, N);

    printf("Starting computation for %dx%d matrices...\n", N, N);
    double start_time = get_time();

    complex_matrix_multiply(A, B, C, D, E, F, N);

    double end_time = get_time();
    double execution_time = (end_time - start_time) / 1000000.0;

    printf("Matrix size: %d x %d\n", N, N);
    printf("Serial CPU Execution time: %.4f seconds\n",
execution_time);
```

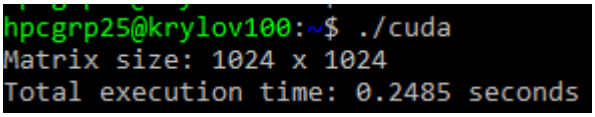
```
    free(A); free(B); free(C); free(D); free(E); free(F);
    return 0;
}
```

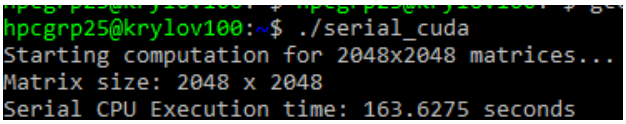
Πραγματοποιούμε πειράματα για N=512,1024, 2048, 4096 (για την cpu υλοποίηση έως 2048, καθώς ο χρόνος εκτέλεσης είναι πολύ μεγάλος):

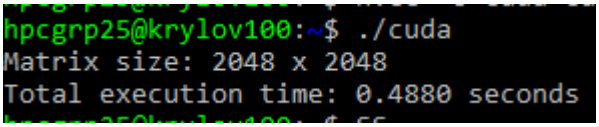


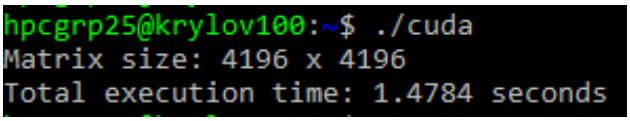


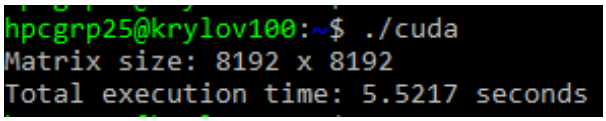






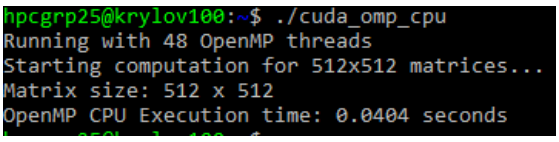


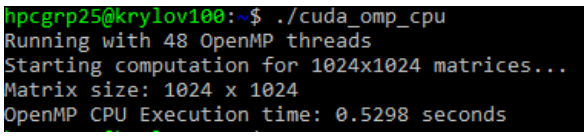


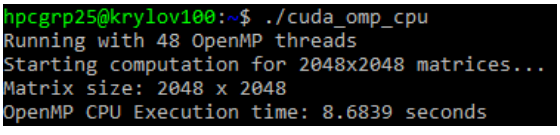


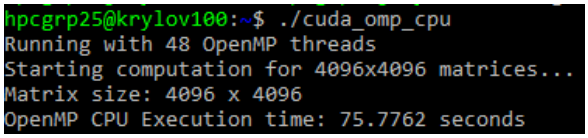
Παρατηρούμε ότι για μικρές διαστάσεις οι διαφορές δεν είναι δραματικές (περίπου 4x speedup), αλλά όσο μεγαλώνουν γίνονται μεγαλύτερες (περίπου 40x πιο γρήγορη η gpu για 1024x1024) έως το σημείο όπου απλά οι διαφορές είναι ασύγκριτες. Ένα πρόβλημα μισού δευτερολέπτου μεταφράζεται σε λεπτά για την cpu.

Για μια πιο δίκαια σύγκριση και λαμβάνοντας υπόψιν ότι οι επεξεργαστές xeon υποφέρουν με single core performance, μετατρέπουμε τον σειριακό κώδικα σε παράλληλο και έχουμε:









Σε μικρές διαστάσεις η cpu είναι αρκετά πιο γρήγορη, αφού δεν έχει καθυστερήσεις μεταφορών δεδομένων Host->Device και το αντίθετο. Όσο μεγαλώνουν οι διαστάσεις, πάλι βλέπουμε ότι η cpu δεν συγκρίνεται με την απόδοση της gpu σε καμία περίπτωση.