Software & Programming of HPC Systems

ΕΡΓΑΣΙΑ 2: SIMD and CUDA

Φλουρής Παντελής, up1093507

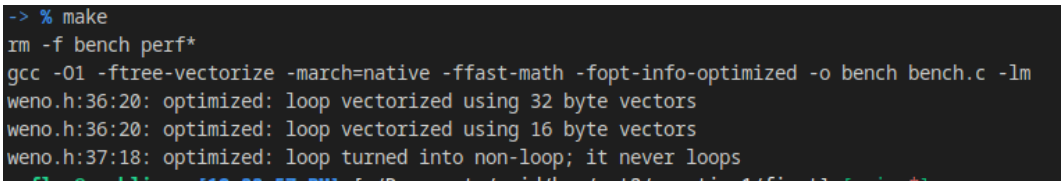
Κολτσάκης Χρυσάφης, up1084671

1. SIMD and WENO5

A)

Η ενεργοποίηση των GCC flags **-O1 -march=native -ftree-vectorize** στον αρχικό κώδικα δεν παρείχε καμια βελτιστοποίηση. Αυτο εύκολα λύθηκε με μετατροπή της weno_minus_core σε static inline, επιτρέποντας στον GCC να την κανει vectorize. Δίνουμε στα pointers που περνώνται στην weno_minus_reference το attribute __restrict__, βοηθώντας τον compiler να καταλάβει οτι ειναι aligned.

Μετά απο αυτή την αλλαγή, ο gcc δημιουργεί δυο εκδόσεις: Μια χρησιμοποιώντας SSE και μία χρησιμοποιώντας AVX, πετυχαίνοντας έτσι μεγαλύτερη συμβατότητα με παλαιότερα συστήματα.



Η χρηση OpenMP vectorization ηταν πιο απλή αλλα λιγότερο αποδοτική. Στον αρχικό πηγαίο κώδικα, καναμε μόνο αυτές τις αλλαγές:

- Προσθήκη <omp.h>
- #pragma omp declare simd στις συναρτησεις weno_minus_reference και weno_minus_core

- #pragma omp simd aligned(a, b, c, d, e, out:32) στο for loop (32 διότι οι μεταβλητές ειναι aligned στα 32 bytes).

Αξίζει να σημειωθεί οτι η χρηση των OpenMP directives στον κώδικα του πρώτου υποερωτήματος (με τα compiler vectorizations) δεν παρείχε καμια βελτίωση).

```
gcc -O1 -fopenmp-simd -fopenmp -ffast-math -fopt-info-vec-optimized -o bench bench.c -lm
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 32 byte vectors
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 32 byte vectors
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 32 byte vectors
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 32 byte vectors
weno.h:41:53: optimized: loop vectorized using 16 byte vectors
weno.h:41:53: optimized: loop vectorized using 64 byte vectors
weno.h:41:53: optimized: loop vectorized using 32 byte vectors
weno.h:41:53: optimized: loop vectorized using 64 byte vectors
weno.h:41:53: optimized: loop vectorized using 32 byte vectors
weno.h:7:27: optimized: loop vectorized using 16 byte vectors
weno.h:7:27: optimized: loop vectorized using 32 byte vectors
weno.h:7:27: optimized: loop vectorized using 16 byte vectors
weno.h:7:27: optimized: loop vectorized using 32 byte vectors
weno.h:7:27: optimized: loop vectorized using 32 byte vectors
weno.h:7:27: optimized: loop vectorized using 64 byte vectors
weno.h:7:27: optimized: loop vectorized using 64 byte vectors
```

Για την παραλληλοποίηση με AVX intrinsics, σβησαμε την συνάρτηση weno_minus_core και τοποθετήσαμε όλο τον κώδικα απευθείας μεσα στην weno_minus_reference. Αυτο το βήμα δεν είναι απαραίτητο, αλλα το βρήκαμε πιο εύκολο.

Ο βρόγχος επανάληψης γινεται με βήματα των 8, και αρχικοποιούνται σε καταχωρητές 8 τιμές του κάθε float array και μερικοί καταχωρητές με 8 αντίγραφα του κάθε αριθμού που χρειαζόμαστε για τις πράξεις.

```
__m256 a_ = _mm256_load_ps(a+i);
__m256 b_ = _mm256_load_ps(b+i);
__m256 c_ = _mm256_load_ps(c+i);
__m256 d_ = _mm256_load_ps(d+i);
__m256 e_ = _mm256_load_ps(e+i);

__m256 coef1 = _mm256_set1_ps(4.0f/3.0f);
__m256 coef2 = _mm256_set1_ps(19.0f/3.0f);
__m256 coef3 = _mm256_set1_ps(11.0f/3.0f);
__m256 coef4 = _mm256_set1_ps(25.0f/3.0f);
__m256 coef5 = _mm256_set1_ps(31.0f/3.0f);
__m256 coef6 = _mm256_set1_ps(10.0f/3.0f);
__m256 coef7 = _mm256_set1_ps(13.0f/3.0f);
__m256 coef8 = _mm256_set1_ps(5.0f/3.0f);
```

Στην συνέχεια υπολογίζονται και αποθηκεύονται σε AVX καταχωρητές οι τιμές is0, is1, is2 με χρήση fmadd και fmsub όπου ειναι εφικτό:

```
__m256 A = _mm256_mul_ps(a_, coef1);
A = _mm256_fmsub_ps(b_, coef2, A);
A = _mm256_fmadd_ps(c_, coef3, A);
A = _mm256_add_ps(a_, A);

__m256 B = _mm256_mul_ps(b_, coef4);
B = _mm256_fmsub_ps(c_, coef5, B);

__m256 C = _mm256_mul_ps(c_, coef6);
C = _mm256_mul_ps(c_, C);

__m256 is0 = _mm256_add_ps(A, B);
is0 = _mm256_add_ps(is0, C);
```

Συνεχίζουμε την αντιστοιχη διαδικασία για το υπόλοιπο του κώδικα. Χρησιμοποιούμε την εντολη _mm256_rcp_ps στα σημεία που χρειαζόμαστε κάποιο αντίστροφο.

Στο τέλος αποθηκεύουμε τα αποτελέσματα στην μνήμη.

B)

Αποτελέσματα με αρχικό bench:

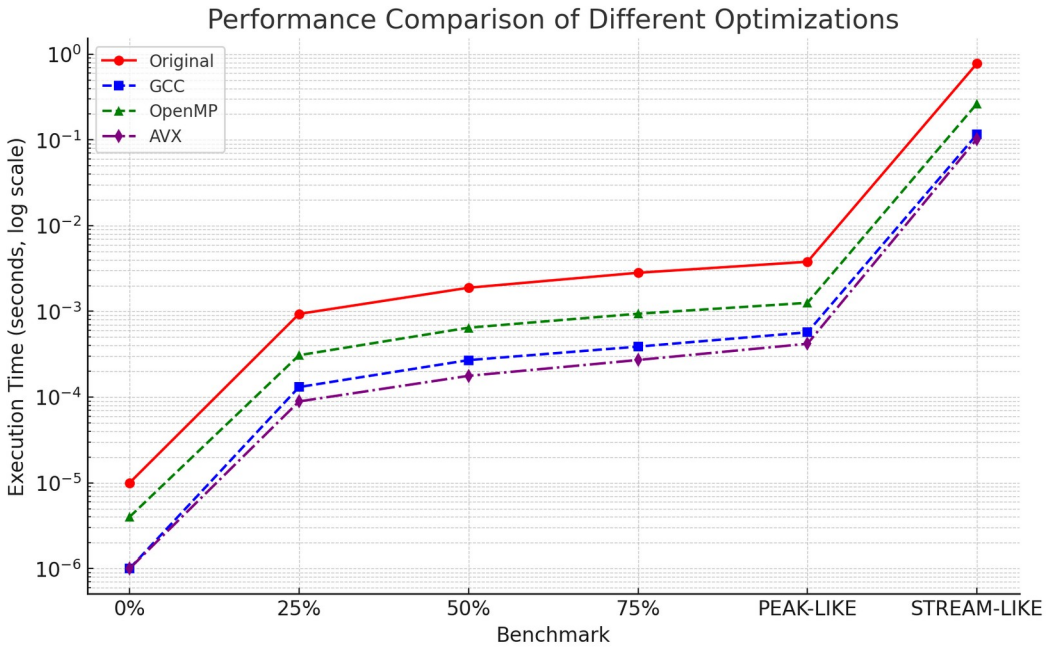
Benchmark	original	GCC	OpenMP	AVX
PEAK-LIKE	0.004017	0.000549	0.001237	0.000363
STREAM-LIKE	0.778927	0.120399	0.262018	0.101510

Οπως ειναι προφανές, οι ταχύτητες όταν τα δεδομένα βρισκονται στην κρυφή μνήμη ειναι πολύ γρήγορες. Η βελτίωση ειναι επίσης μεγαλύτερη όταν τα δεδομένα βρισκονται στην κρυφή μνήμη, καθώς όταν τα δεδομένα πρέπει να φορτωθούν απο την κύρια μνήμη εισάγεται μια μεγάλη καθυστέρηση που δεν μπορούμε να αποφύγουμε.

Η πιο γρήγορη μέθοδος ειναι η χρήση AVX intrinsics, με αμέσως επόμενη σε ταχύτητα τα GCC optimizations. Το vectorization με OpenMP ειναι σημαντικά και σταθερά πιο αργο, με 4x βελτίωση από τον αρχικό κώδικα, συγκριτκά με >10x βελτίωση απο τις άλλες μεθόδους για το PEAK-LIKE benchmark, και 6-7x βελτίωση για το STREAM-LIKE.

Επεκτείναμε το benchmark βάζοντας διαφορετικά nentries – ntimes, με στόχο προσομοίωσης διαφορετικών ποσοστών χρήσης cache.

Benchmark	original	GCC	OpenMP	AVX
0%	0.000010	0.000001	0.000004	0.000001
25%	0.000934	0.000131	0.000308	0.000089
50%	0.001887	0.000270	0.000645	0.000177
75%	0.002820	0.000388	0.000939	0.000271
PEAK-LIKE	0.003791	0.000570	0.001258	0.000419
STREAM-LIKE	0.780143	0.115920	0.265023	0.102822



Παρατηρούμε οτι η βελτίωση στην απόδοση ειναι σταθερή ανεξαρτητα ποσοστού χρήσης cache, δείχνοντας την αξία αυτής της τεχνικής.

2. CUDA and Complex Matrix Multiplication

Εκτελέσαμε τους υπολογισμούς με 4 διαφορετικούς τρόπους:

- CUDA with Global Memory (cuda_global.h)
- CUDA with Shared Memory (cuda_shared.h)
- CUDA with cuBLAS Library (cuda_cublas.h)
- CPU (cpu.h)

Η κάθε υλοποίηση περιέχει τις συναρτήσεις της στα αντοίστιχα header files.

Στην main αρχικά δεσμεύεται μνήμη για τα μητρώα (χρησιμοποιείται μονοδιάστατος πίνακας για ευχρηστία και καλύτερη τοπικότητα μνήμης), και αρχικοποιούνται κατάλληλα τα A,B,C,D.

Στην συνέχεια δεσμεύεται μνήμη και στην κάρτα γραφικών για τα A, B, C, D, και αντιγράφονται τα περιεχόμενα απο την CPU στην GPU.

Στην συνέχεια εκτελούνται με την σειρά υπολογισμοί για τις διαφορετικές μεθόδους (ενδιάμεσα τίθενται 0 οι τιμές στα μητρώα αποτελεσμάτων για αποφυγή λαθών). (computeX()).

Τέλος, συγκρίνονται τα αποτελέσματα μεταξύ των διαφόρων μεθόδων για να επιβεβαιωθούν τα σωστά αποτελέσματα.

Λογω της βιβλιοθήκης cuBLAS, η οποία περιμένει column-major indexing, αυτό χρησιμοποιήσαμε για όλες τις μεθόδους.

A. cuda_global.h

Η “χαζή” μέθοδος, που χρησιμοποιεί την global μνήμη της GPU.

Στην συνάρτηση computeCUDAGlobal, που δέχεται ως ορίσματα τα μητρώα στην CPU στα οποία θα αποθηκευτούν τα αποτελέσματα, ορίζουμε τον αριθμό των blocks μεγέθους 8x8 (πολλαπλάσιο του warp size ετσι ώστε να μην εχουμε ανενεργά threads) ετσι ωστε να υπολογίζονται όλα τα δεδομένα. Η καλύτερη απόδοση συμβαίνεια για blocks 8x8,

Στην συνέχεια καλούνται με την σειρά οι απαραίτητες συναρτήσεις matMult / vecAdd / VecSub για τον υπολογισμό της ζητούμενης πράξης. Στο τέλος, αποθηκεύονται τα τελικά αποτελέσματα στις μεταβλητές του επεξεργαστή.

B. cuda_shared.h

Μια πιο “εξυπνη” λύση, καθώς χρησιμοποιεί shared μνήμη, μειώνοντας την καθυστέρηση και χρησιμοποιώντας πιο ορθά την μνήμη της GPU.

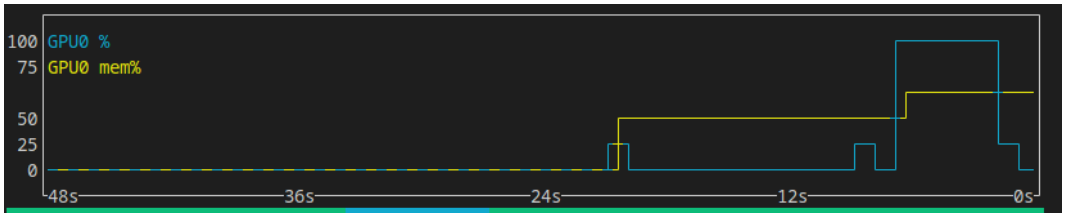
Η λειτουργία ειναι παρομοια με την global, με την διαφορά οτι χρησιμοποιείται shared μνήμη.

Για κάποιο λόγο, για μεγάλες τιμες του N, η shared και η global υλοποιήσεις αποτυγχάνουν κάποιες φορές, επιστρέφοντας χωρίς αποτελέσματα. Υποψιαζόμαστε οτι σε περίπτωση που η VRAM δεν αρκεί, απλά επιστρέφει χωρίς αποτέλεσμα. Η παρακάτω υλοποίηση είναι ***και γρηγορότερη*** *και* ***λειτουργεί ανεξάρτητως μεγέθους****.*

C. cuda_cublas.h

Η καλύτερη λύση, καθώς χρησιμοποιεί την optimized βιβλιοθήκη cuBLAS για τους υπολογισμούς. Η διαφορά γίνεται πιο εμφανής με μεγαλύτερα N. Μεσω της μεθόδου cublasDgemm γίνονται οι ιαπαραίτητες πράξεις και τα δεδομένα αντιγράφονται στην CPU.

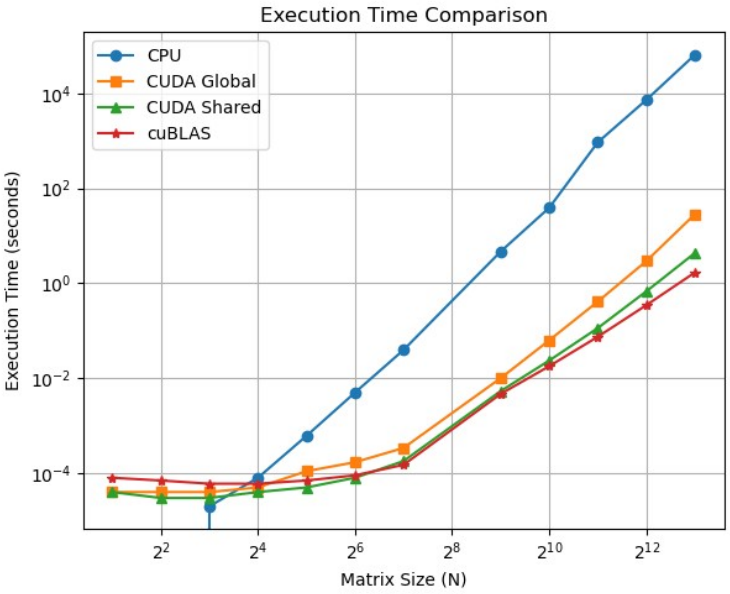
Επιχειρήσαμε να αξιοποιήσουμε streams για τον παράλληλο υπολογισμό των E και F, αλλά η απόδοση παρέμεινε η ίδια. Κοιτάζοντας την χρήση της GPU μέσω NVTOP κατα την διάρκεια λειτουργείας, είναι αρκετά κοντά στο 100%. Αυτο δείχνει οτι και με χρήση πολλαπλών streams δεν θα υπήρχε παραπάνω βελτίωση, αφου η GPU λειτουργεί ηδη κοντά στο έπακρο των δυνατοτήτων της.

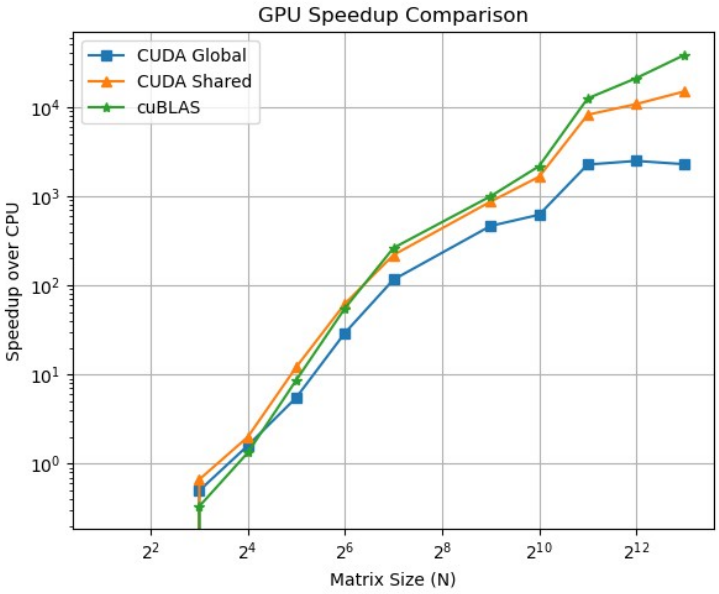


D. cpu.h

Σε αυτό το αρχείο, πέραν των συναρτήσεων για τους υπολογισμούς (που είναι αρκετά παρόμοιες με την CUDA υλοποίης, απλώς για CPU), εχουμε τοποθετήσει και τις συναρτήσεις για initialization και cleanup.

Ε. Αποτελέσματα





Παρατηρούμε γραμμική επιτάχυνση για όλες τις υλοποιήσεις συγκριτικά με τον CPU υπολογισμό, αλλα για N = 2^11 και μετά παρατηρείται η επιτάχυνση να επιβραδύνει και για τις 3 υλοποιήσεις. Η CUDA Global μαλιστα μοιάζει να σταματά να επιταχύνει περαιτέρω. Θα θέλαμε να τρέξουμε την σύγκριση αυτή και για μεγαλύτερα μεγέθη, αλλά η επεξεργασία CPU ειναι πολύ χρονοβόρα.

Ως τελικό συμπέρασμα αυτής της εργασίας έχουμε οτι η παραλληλοποίηση με GPUs είναι ένα εξαιρετικά δυνατό εργαλείο, και η χρήση εξειδικευμένων βιβλιοθηκών όπου αυτές υπάρχουν είναι επιθυμητή.