Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

Χειμερινό Εξάμηνο 2024-2025

3ο Σετ Εργαστηριακών Ασκήσεων

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

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

OpenMP και Πολλαπλασιασμός Μιγαδικών Μητρώων

Η υλοποίηση που ακολουθήσαμε για τον πολλαπλασιασμό μιγαδικών μητρώων στη GPU με OpenMP είναι παρόμοια με την υλοποίηση που κάναμε στο CUDA. 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.

Για την εκτέλεση των πράξεων στη GPU τροποποιήσαμε την αποδοτική μας υλοποίηση από το CUDA. Πριν εκτελεστεί η οποιαδήποτε πράξη στη GPU, πρέπει να ορίσουμε το περιβάλλον των δεδομένων με την οδηγία `#pragma omp target data`. Οι πίνακες A, B, C, D αντιγράφονται πλήρως στη GPU και παραμένουν εκεί (to), καθώς δεν τροποποιούνται εντός της GPU. Τα αποτελέσματα των πινάκων E, F επιστρέφονται από την GPU (from), χωρίς να χρειαστεί να μεταφερθεί η αρχική τους τιμή από τη CPU στη GPU:

```
#pragma omp target data map(to: A[0:N*N], B[0:N*N], C[0:N*N],
D[0:N*N]) map(from: E[0:N*N], F[0:N*N])
```

Για να ορίσουμε κώδικα που τρέχει στη GPU χρησιμοποιούμε το `#pragma omp target`. Η παραλληλοποίηση του for loop στη GPU γίνεται με το `#pragma omp teams`, το οποίο ορίζει ομάδες νημάτων, όπου κάθε ομάδα τρέχει σε ένα μπλοκ της GPU. Με την οδηγία `distribute` κατανέμουμε τις επαναλήψεις στα νήματα των ομάδων. Το teams distribute είναι ιδανικό για παραλληλοποίηση σε GPU, καθώς εκτελεί ιεραρχική παραλληλία (πρώτα σε ομάδες, μετά σε νήματα), η οποία ταιριάζει καλά με την ιεραρχική δομή της GPU. Επίσης χρησιμοποιούμε `collapse(2)` για να δημιουργήσουμε έναν ενιαίο χώρο επαναλήψεων για τα πρώτα 2 επίπεδα του for, επιτυγχάνοντας καλύτερη απόδοση:

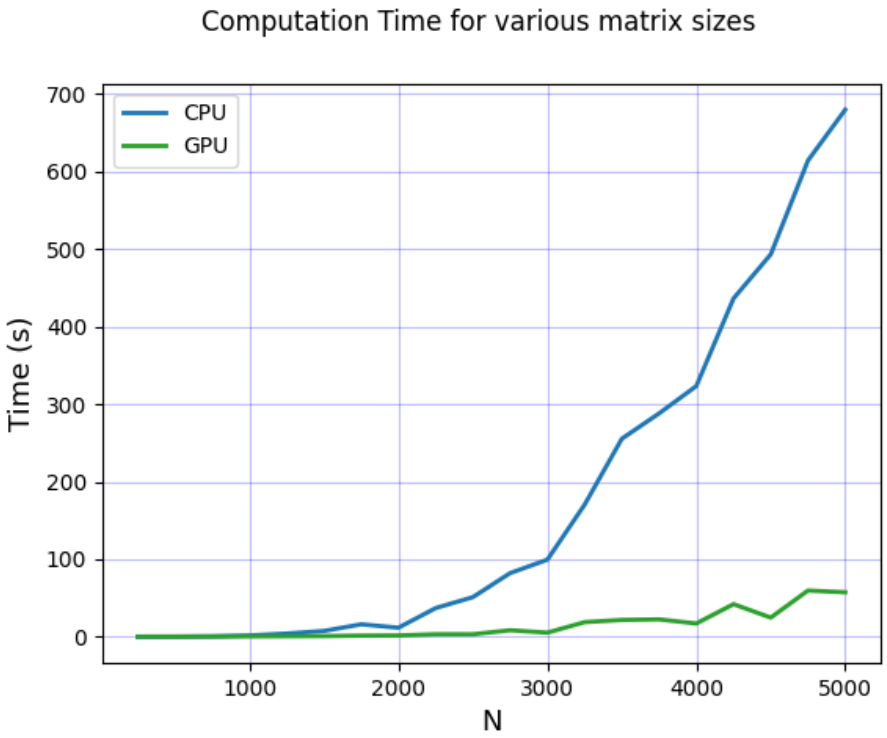
```
#pragma omp target teams distribute parallel for collapse(2)
```

Εντός του for χρησιμοποιούμε την ίδια λογική με το CUDA. Δεδομένου ότι κάθε νήμα της GPU έχει αναλάβει ένα ζεύγος (i j), κάθε νήμα θα υπολογίζει την αντίστοιχη θέση των μητρώων E και F. Κάθε νήμα θα πρέπει να κάνει 4 πολλαπλασιασμούς γραμμή-στήλη, μια αφαίρεση (για το E) και μια πρόσθεση (για το F):

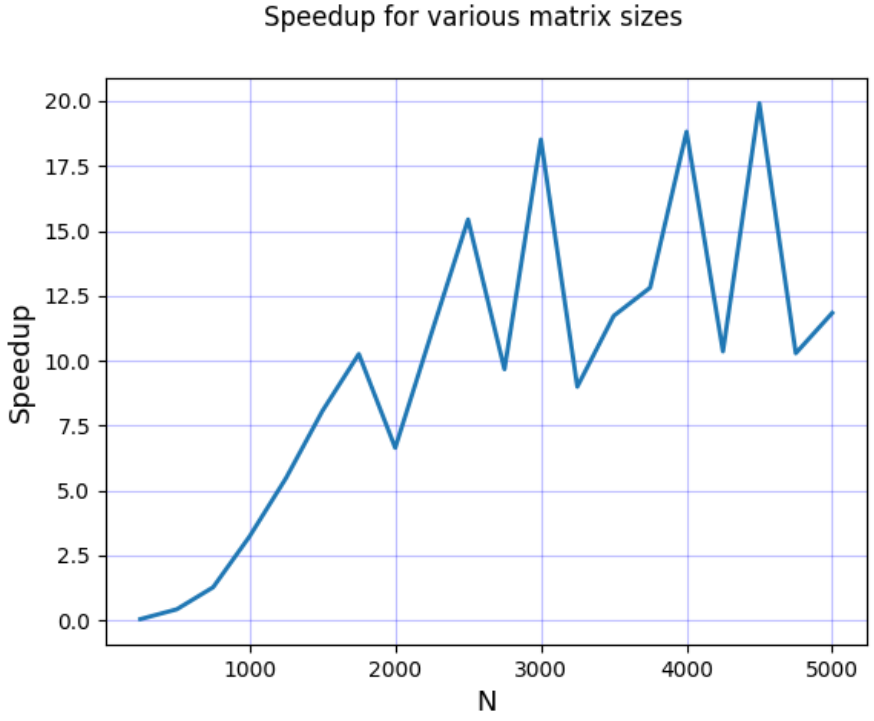
```
float resAC = 0.0f, resBD = 0.0f, resAD = 0.0f, resBC = 0.0f;

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;
```

Εκτέλεση δοκιμών

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



Παρατηρούμε ότι ο υπολογισμός στη GPU είναι πολύ καλύτερος από τη σειριακή εκτέλεση στη CPU, καθώς για τα μεγέθη μητρώων που δοκιμάσαμε, ο χρόνος της GPU μεταβάλλεται ελάχιστα, ενώ η CPU γίνεται όλο και πιο αργή. 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, αλλά καθώς τα δεδομένα μεγαλώνουν, η GPU προσφέρει μεγαλύτερο κέρδος απόδοσης:



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

N	Init	CPU	GPU	Comparison
250	0,008	0,017	0,348	0
500	0,026	0,143	0,335	0
750	0,047	0,53	0,415	0,001
1000	0,094	1,763	0,545	0,001
1250	0,144	4,186	0,764	0,001
1500	0,204	7,589	0,941	0,002
1750	0,273	16,137	1,573	0,002
2000	0,357	11,763	1,77	0,004
2250	0,538	37,075	3,346	0,007
2500	0,57	51,183	3,315	0,006
2750	0,677	82,181	8,495	0,008
3000	0,818	99,486	5,372	0,014
3250	0,938	170,458	18,938	0,017
3500	1,14	255,482	21,782	0,009
3750	1,272	288,132	22,486	0,021
4000	1,43	323,405	17,183	0,027
4250	1,636	436,36	42,121	0,013
4500	1,848	493,315	24,769	0,023
4750	2,019	614,281	59,707	0,016
5000	2,232	679,822	57,424	0,018

Περιβάλλον Υλοποίησης

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