Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης (Α.Π.Θ.)

ΗΥ3603 Παράλληλα και Διανεμημένα Συστήματα

Εργασία 4: Εύρεση πλήθους τριγώνων μη κατευθυνόμενου απλού γράφου.

Αντωνιάδης Δημήτριος (8462): akdimitri@auth.gr

23 Σεπτεμβρίου 2019

Περιεχόμενα

1 Εισαγωγή.

Το παρόν έγγραφο αποτελεί την αναφορά της τέταρτης ($4_{ης}$) εργασίας του μαθήματος *Παράλληλα και Διανεμημένα Συστήματα*. Στο πλαίσιο της εργασίας αυτής υλοποιήθηκε παράλληλος αλγόριθμος σε CUDA, ο οποίος διαβάζει σε αραιή μορφή τον πίνακα γειτνίασης A ενός απλού, μη κατευθυνόμενου γράφου $G(V, E)$ και υπολογίζει τον αριθμό των τριγώνων που σχηματίζονται σε αυτόν.

Ο πηγαίος κώδικας της εργασίας και λοιπά αρχεία περιλαμβάνονται στον παρακάτω σύνδεσμο:

https://github.com/akdimitri/CUDA-Parallel-Triangle-Counting

Παραπάνω έγινε μία σύντομη περιγραφή της εργασίας, στην επόμενη ($2_{η}$) ενότητα αναλύονται οι αλγόριθμοι που υλοποιήθηκαν. Στην τρίτη ($3_{η}$) ενότητα παρουσιάζονται τα αρχεία του repository που περιλαμβάνει η εργασία και ο τρόπος μεταγλώττισης του προγράμματος. Στην τέταρτη ($4_{η}$) ενότητα παρουσιάζονται τα αποτελέσματα των αλγορίθμων από τις δοκιμές. Τέλος, στην πέμπτη ($5_{η}$) ενότητα συνοψίζονται τα συμπεράσματα της εργασίας.

2 Αλγόριθμοι εύρεσης πλήθους τριγώνων n_T μη κατευθυνόμενου απλού γράφου $G(V, E)$.

Σύμφωνα με την εκφώνηση, η εύρεση του πλήθους των τριγώνων ενός μη κατευθυνόμενου απλού γράφου μπορεί να γίνει με τον παρακάτω αλγόριθμο.

Algorithm 1: TRIANGLES counting.

Input: A adjacency matrix of graph $G(V, E)$
Output: n_T graph's number of triangles

1 $C = ((A * A) \odot A$ `/* ⊙ is Hadamard product */`
2 $n_T = \frac{1}{6} \sum_{ij} C_{ij}$

2.1 Αλγόριθμος σε CUDA με τη χρήση της βιβλιοθήκης CUSPARSE.

Αρχικά, έγινε προσπάθεια να υλοποιηθεί ο παραπάνω αλγόριθμος σε CUDA με τη χρήση της βιβλιοθήκης CUSPARSE. Πιο συγκεκριμένα, το πρόγραμμα διάβαζε τον πίνακα A σε αραιή μορφή COO (Coordinate list), τον μετέτρεπε σε μορφή CSR (Compressed sparse row) με τη χρήση της συνάρτησης cusparseXcoo2csr(...) και πραγματοποιούσε τον πολλαπλασιασμό $A * A$ με τη χρήση της συνάρτησης cusparseScsrgemm2(...). Το αποτέλεσμα του πολλαπλασιασμού ήταν ο πίνακας B σε μορφή CSR. Στη συνέχεια ο πίνακας αυτός μετατρεπόταν σε μορφή COO και διατηρούνταν μόνο τα στοιχεία τα οποία είχαν κοινές συντεταγμένες με τον πίνακα A.

Ωστόσο, ο αλγόριθμος αυτός απορρίφθηκε γρήγορα διότι η διαδικασία του πολλαπλασιασμού ήταν ιδιαίτερα χρονοβόρα και δεν παρατηρούνταν σημαντική επιτάχυνση έναντι του αντίστοιχου χρόνου που απαιτούσε ο αλγόριθμος στο MATLAB.

Για την βελτίωση της ταχύτητας εκτέλεσης του προγράμματος δοκιμάσθηκαν και παραλλαγές του αλγορίθμου 1.

Πιο αναλυτικά δοκιμάσθηκαν οι δύο παρακάτω αλγόριθμοι.

Algorithm 2: TRIANGLES counting.

Input: A adjacency matrix of graph $G(V, E)$, L Lower triangular part of matrix A
Output: n_T graph's number of triangles

1 $C = ((A * A) \odot L$ `/* ⊙ is Hadamard product */`
2 $n_T = \frac{1}{3} \sum_{ij} C_{ij}$

Algorithm 3: TRIANGLES counting.

Input: L Lower triangular part of graph's $G(V, E)$ adjacency matrix A
Output: n_T graph's number of triangles

1 $C = ((L * L) \odot L$ `/* ⊙ is Hadamard product */`
2 $n_T = \frac{1}{2} \sum_{ij} C_{ij}$

Στους παραπάνω αλγορίθμους έγινε χρήση του κάτω(άνω) τριγωνικού πίνακα του A. Παρόλο αυτά, ο αλγόριθμος με τη χρήση της βιβλιοθήκης CUSPARSE δεν εμφάνισε αξιόλογη απόδοση και απορρίφθηκε. Η αργή εκτέλεση του οφείλονταν κυρίως στο γεγονός ότι ο πολλαπλασιασμός $A * A$ κατανάλωνε άσκοπο χρόνο για περιττές πράξεις, δηλαδή για τον υπολογισμό στοιχείων τα οποία θα απορρίπτονταν αργότερα από το γινόμενο Hadamard. Ένας ακόμη λόγος που απορρίφθηκε ο αλγόριθμος που έκανε χρήση της βιβλιοθήκης CUSPARSE, ήταν υπερβολική μνήμη που καταλάμβαναν τα στοιχεία. Για τη χρήση των συναρτήσεων απαιτούνταν πέρα από τους πίνακες με τις συντεταγμένες των στοιχείων και πίνακες με τις τιμές τους. Οι πίνακες με τις τιμές ήταν άχρηστοι εφόσον ήταν γνωστό ότι όλες οι τιμές των μη μηδενικών στοιχείων των πινάκων ισούταν με 1.

2.2 Σειριακός Αλγόριθμος.

Εφόσον παρατηρήθηκε ότι ο πολλαπλασιασμός είναι μία χρονοβόρα διαδικασία επιχειρήθηκε να υπολογισθούν μόνο τα γινόμενα των *χρήσιμων* στοιχείων. Χρήσιμα στοιχεία είναι τα στοιχεία που έχουν τιμή 1 στον πίνακα A.

Έστω ο πίνακας γειτνίασης A μεγέθους N του απλού μη κατευθυνόμενου γράφου $G(V, E)$.

$$A = \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

Εφόσον ο πίνακας έχει μόνο τιμές 0 και 1 αρκεί να αποθηκεύσουμε μόνο τις συντεταγμένες των στοιχείων με τιμή 1. Ο A έχει $NNZ = 14$ μη μηδενικά στοιχεία.

Ο πίνακας σε μορφή CSR αποθηκεύεται ως εξής (η αρίθμηση των πινάκων ξεκινάει από το 0) [1]:

$$Rows = \begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 6 & 8 & 11 & 14 \\ \hline \end{array}$$

$$Cols = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 3 & 0 & 2 & 3 & 4 & 1 & 4 & 0 & 1 & 4 & 1 & 2 & 3 \\ \hline \end{array}$$

Ο πίνακας Rows περιέχει στη θέση i τη θέση του πρώτου στοιχείου της γραμμής i στον πίνακα Cols. Ο πίνακας Cols περιέχει τις αντίστοιχες στήλες των στοιχείων κάθε γραμμής με τιμή 1.

Δηλαδή, το πρώτο στοιχείο της γραμμής 0 βρίσκεται στη θέση 0 του πίνακα Cols. Η τιμή στη θέση 0 του πίνακα Cols είναι 1. Συνεπώς αυτό σημαίνει ότι στη θέση (0,1) του πίνακα, υπάρχει στοιχείο με τιμή 1.

ή για παράδειγμα,

Το στοιχείο στη θέση 2 του πίνακα Rows είναι το 6. Αυτό σημαίνει ότι η γραμμή 2 αποθηκεύει τα στοιχεία της στον πίνακα Cols από τη θέση 6 και μετά. Στη θέση 6 του Cols η τιμή είναι ένα, άρα το στοιχείο (2,1) έχει τιμή 1.

Εφόσον είναι γνωστές οι συντεταγμένες των μη μηδενικών τιμών, είναι δυνατό να υπολογίσουμε το γινόμενο $A * A$ μόνο για τις συντεταγμένες αυτές.

Τα στοιχεία με τιμή 1 έχουν τις παρακάτω συντεταγμένες:
(0,1) (0,2)
(1,0) (1,2) (1,3) (1,4)
(2,1) (2,4)
(3,0) (3,1) (3,4)
(4,1) (4,2) (4,3)

Ας δούμε για παράδειγμα το στοιχείο στη θέση (3,4). Για να υπολογισθεί ο πολλαπλασιασμός $A * A$ στη θέση (3,4) αρκεί να πολλαπλασιασθεί η γραμμή 3 με τη στήλη 4. Εφόσον ο πίνακας είναι συμμετρικός αρκεί να υπολογισθεί το γινόμενο της γραμμής 3 με τη γραμμή 4. Δεδομένου ότι τα στοιχεία είναι αποθηκευμένα σε μορφή CSR, αρκεί να βρεθεί το σύνολο των κοινών στοιχείων της γραμμής 3 και 4 στον πίνακα Cols.

Δηλαδή:

$$\begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 0 & 0 & 1 \\ \hline \end{array} * \begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline 1 \\ \hline 1 \\ \hline 0 \\ \hline \end{array} = 1 * 0 + 1 * 1 + 0 * 1 + 0 * 1 + 1 * 0 = 1$$

ή

Στήλες με μη μηδενικά στοιχεία της γραμμής 3:

0	1	4

Στήλες με μη μηδενικά στοιχεία της γραμμής 4:

1	2	3

΄Οπως φαίνεται παραπάνω έχουν ένα κοινό στοιχείο το 1. ΄Αρα το αποτέλεσμα στη θέση (3,4) του γινομένου $A * A$ θα είναι ίσο με 1.

Επομένως προκειμένου να υπολογισθεί πιο γρήγορα η έκφραση $(A * A) \odot A$ αρκεί για κάθε μη μηδενικό στοιχείο (i, j) του πίνακα A να βρεθούν τα κοινά στοιχεία των γραμμών i, j στον πίνακα Cols.

Algorithm 4: TRIANGLES counting - Serial Algorithm .

Input: A adjacency matrix of graph $G(V, E)$ in CSR format.

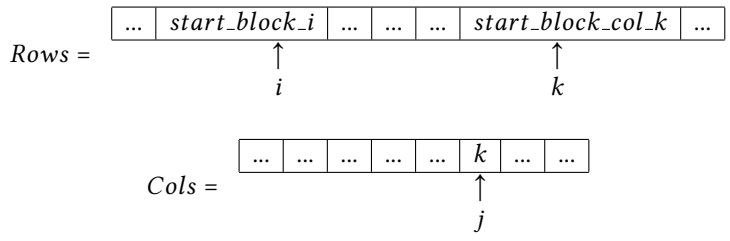
- $Rows[N + 1]$
- $Cols[NNZ]$

Output: n_T graph's number of triangles

```
1  sum = 0 for i ← 0 to N − 1 do
2      start_block_i = Rows(i)
3      stop_block_i = Rows(i + 1) − 1
4      for j ← start_block_i to stop_block_i do
5          k = Cols[j]
6          start_block_col_k = Rows(k)
7          stop_block_col_k = Rows(k + 1) − 1
8          matches_found =
             compare_blocks(Cols, start_block_i, stop_block_i, start_block_col_k, stop_block_col_k)
9          sum = sum + matches_found
10 n_T = sum/6
```

Ουσιαστικά ο παραπάνω αλγόριθμος ελέγχει για κάθε στοιχείο (i, k) αν το block που περιγράφει τη γραμμή i στον πίνακα Cols έχει κοινά στοιχεία με το block που περιγράφει τη γραμμή k στον πίνακα Cols. Τελικά, ο αλγόριθμος αθροίζει το σύνολο των κοινών στοιχείων που βρίσκει και το διαιρεί με 6. Έτσι προκύπτει ο αριθμός των τριγώνων του γράφου.

Η συνάρτηση *compare_blocks* δέχεται ως ορίσματα τον πίνακα Cols, τη θέση του πρώτου στοιχείου της γραμμή i στον πίνακα Cols η οποία περιγράφεται στον παραπάνω αλγόριθμο από τη μεταβλητή *start_block_i*. Ακόμη, δέχεται ως όρισμα τη θέση του τελευταίου στοιχείου της γραμμής i στον πίνακα Cols η οποία περιγράφεται από τη μεταβλητή *stop_block_i*. Η μεταβλητή k περιγράφει την τεταγμένη ενός στοιχείου στη γραμμή i στο οποίο υπάρχει μη μηδενική τιμή και επειδή A συμμετρικός η στήλη k θα ισούται με τη γραμμή k. Αντίστοιχα ορίζονται οι μεταβλητές *start_block_col_k* και *stop_block_col_k*.



Algorithm 5: COMPARE blocks.

Input:

- $Cols[NNZ]$
- $start\_block\_i$
- $stop\_block\_i$
- $start\_block\_col\_k$
- $stop\_block\_col\_k$

Output: $matches\_found$

```
1  i = start_block_i
2  k = start_block_col_k
3  matches_found = 0
4  while (i <= stop_block_i) && (k <= stop_block_col_k) do
5      if Cols[i] == Cols[k] then
6          i++
7          k++
8          matches_found++
9      else if Cols[i] < Cols[k] then
10         i++
11     else
12         k++
13 return matches_found
```

Ο παραπάνω αλγόριθμος μπορεί να γίνει ακόμη πιο αποδοτικός αν σκεφτεί κανείς τη σχέση

$$(A * A) \odot L$$

ή

$$(A * A) \odot U$$

Οι παραπάνω σχέσεις μπορούν να υλοποιηθούν με τη χρήση μίας εντολής If η οποία Θα επιτρέπει την εκτέλεση του αλγορίθμου μόνο για $k > i$, δηλαδή μόνο για τα στοιχεία που βρίσκονται στον κάτω(άνω) τριγωνικό. Με τον τρόπο αυτό θα εκτελεσθούν ακριβώς οι μισές συγκρίσεις εφόσον ο A είναι συμμετρικός.

2.3 Παράλληλος αλγόριθμος σε CUDA.

Δεδομένου ότι έχει υλοποιηθεί ο σειριακός αλγόριθμος τώρα πρέπει ο αλγόριθμος αυτός να μετατραπεί σε παράλληλο. Αυτό που μπορεί να παραλληλοποιηθεί είναι το εξωτερικό For loop του σειριακού αλγορίθμου και η διαδικασία σύγκρισης.

Οι κάρτες γραφικών CUDA διαθέτουν έναν αριθμό Streaming Multiprocessors. Κάθε Streaming Multiprocessor μπορεί να διαχειρίζεται threads από μόνο ένα block. Ακόμη ο μέγιστος αριθμός threads που μπορεί να διαχειρίζεται ο Streaming Multiprocessor είναι συγκεκριμένος.

Επομένως αποφασίστηκε να καλείται ένα block για κάθε γραμμή του πίνακα. Κάθε block αποφασίστηκε να έχει το μέγιστο δυνατό αριθμό threads που μπορεί να διαχειριστεί κάθε φορά ένας Streaming Multiprocessor. Αυτός ο αριθμός ονομάζεται wrap size.

Τώρα κάθε thread ενός block αποτελεί ένα στοιχείο της γραμμής $blockIdx.x$. Άρα κάθε thread ελέγχει το σύνολο των κοινών στοιχείων που έχει η γραμμή $blockIdx.x$ στον πίνακα $Cols$ με τη στήλη(γραμμή) που του αντιστοιχεί στον πίνακα $Cols$.

Algorithm 6: MAIN function Triangles counting - Parallel Algorithm .

Input: A adjacency matrix of graph $G(V, E)$ in CSR format.

- $Rows[N + 1]$
- $Cols[NNZ]$

Output: n_T graph's number of triangles

```
1 ...
2 total_sum_d = 0   /* assuming variable total_sum at device has been set to 0 */
3 kernel«< N, wrap_size »>( Rows, Cols, total_sum_d)
4 ...
5 n_T = total_sum/3
```

Η συνάρτηση **kernel** υλοποιεί τον παράλληλο kernel σε CUDA όπως φαίνεται στον αλγόριθμο παρακάτω.

Algorithm 7: KERNEL function Triangles counting - Parallel Algorithm .

Input: A adjacency matrix of graph $G(V, E)$ in CSR format.

- $Rows[N + 1]$
- $Cols[NNZ]$

Output: $total\_sum\_d$: 3 times graph's number of triangles

```
1  i = blockIdx.x
2  start_block_i = Rows[i]
3  stop_block_i = Rows[i + 1] − 1
4  j = start_block_i + threadIdx.x
5  while j <= stop_block_i do
6      temp = 0
7      k = Cols[j]
8      if k > blockIdx.x then
9          start_block_col_k = Rows[k]
10         stop_block_col_k = Rows[k + 1] − 1
11         tepm =
             compare_blocks(Cols, start_block_i, stop_block_i, start_block_col_k, stop_block_col_k)
12         atomicAdd(total_sum_d, temp)
13     j = j + blockDim.x
```

Ουσιαστικά αυτό που πραγματοποιεί ο παραπάνω αλγόριθμος είναι να εκτελεί παράλληλα σε πρώτο επίπεδο τη διαδικασία για κάθε γραμμή σε ξεχωριστό Streaming Multiprocessor και σε επόμενο επίπεδο να εκτελεί παράλληλα κάθε thread τη διαδικασία σύγκρισης του block της γραμμής i με αυτό της στήλης k των μη μηδενικών στοιχείων.

2.4 Παράλληλος αλγόριθμος σε CUDA με τη χρήση Shared Memory.

Δεδομένου ότι εντός ενός block με threads διαβάζεται σε κάθε σύγκριση το block με τα στοιχεία της γραμμής, το block με τα στοιχεία της γραμμής του πίνακα $Cols$ μπορεί να φορτωθεί στη shared memory.

Στην περίπτωση αυτή ο αλγόριθμος είναι ίδιος με τον παραπάνω με μία μικρή τροποποίηση στο Loop και στα ορίσματα της compare_blocks.

Algorithm 8: KERNEL function Triangles counting - Parallel Algorithm - Shared Memory.

Input: A adjacency matrix of graph $G(V, E)$ in CSR format.

- $Rows[N + 1]$
- $Cols[NNZ]$

Output: $total\_sum\_d$: 3 times graph's number of triangles

```
1  i = blockIdx.x
2  start_block_i = Rows[i]
3  stop_block_i = Rows[i + 1] − 1
4  nnz_at_row = stop_block_i − start_block_i
5  j = threadIdx.x
6  while j < nnz_at_row do
7      shared[j] = Cols[start_block_i + j]
8      j = j + blockDim.x
9  __syncthreads()
10 j = start_block_i + threadIdx.x
11 while j <= stop_block_i do
12     temp = 0
13     k = Cols[j]
14     if k > blockIdx.x then
15         start_block_col_k = Rows[k]
16         stop_block_col_k = Rows[k + 1] − 1
17         tepm =
             compare_blocks(Cols, shared, nnz_at_row, start_block_col_k, stop_block_col_k)
18         atomicAdd(total_sum_d, temp)
19     j = j + blockDim.x
20     __syncthreads()
```

Η συνάρτηση compare_blocks(..) είναι σχεδόν πανομοιότυπη. Μπορεί να βρεθεί στο αρχείο ./code/CUDA parallel shared memory/main.cu.

3 Περιεχόμενα φακέλου εργασίας και Μεταγλώττιση.

Στο φάκελο της εργασίας περιλαμβάνονται δύο φάκελοι. Ο φάκελος code και ο φάκελος matlab.

```
Repository
├── code
│   ├── sequential
│   │   └── main.c
│   ├── CUDA parallel
│   │   └── main.cu
│   ├── CUDA parallel shared memory
│   │   └── main.cu
│   ├── data.h
│   └── CUSPARSE parallel
│       ├── main.cu
│       ├── tools.cu
│       ├── tools.h
│       └── cusparse_script.m
└── matlab
    └── script.m
```

Στο φάκελο matlab περιλαμβάνεται το αρχείο script.m. Το αρχείο αυτό υπολογίζει τον αριθμό των τριγώνων για έναν πίνακα A και υπολογίζει και το χρόνο εκτέλεσης του αλγορίθμου στο MATLAB. Επιπλέον, τυπώνει σε αρχείο τον πίνακα A σε μορφή CSR. Πιο συγκεκριμένα, δημιουργεί δύο αρχεία, ένα αρχείο που περιέχει τον πίνακα Cols και ένα αρχείο που περιέχει τον πίνακα Rows.

Στο φάκελο code περιλαμβάνεται το αρχείο data.h. Το αρχείο αυτό περιλαμβάνει το path για τα αρχεία με τους πίνακες $Cols$ και $Rows$ καθώς και για τις μεταβλητές N(μέγεθος πλευράς πίνακα) και NNZ(πλήθος μη μηδενικών στοιχείων πίνακα).

Στο φάκελο sequential περιλαμβάνεται ο σειριακός αλγόριθμος ενώ στους υπόλοιπους περιλαμβάνεται ο παράλληλος και ο παράλληλος με τη χρήση της shared memory αντίστοιχα.

Στο φάκελο CUSPARSE parallel περιλαμβάνεται η υλοποίηση σε CUSPARSE όπως αυτή περιγράφτηκε στην αρχή της ενότητας 2. Το αρχείο cusparse_script.m χρησιμοποιείται για την εξαγωγή των πινάκων σε μορφή COO. Το πρόγραμμα αυτό δεν κάνει χρήση του αρχείου data.h. Η εκτέλεση του προγράμματος σε CUSPARSE απαιτεί και δύο ορίσματα, τον αριθμό των γραμμών και τον αριθμό των μη μηδενικών στοιχείων.

Compilation για το σειριακό αλγόριθμος μπορεί να κάνει κανείς με τη χρήση της εντολής:

- gcc main.c -o main

Compilation για τους παράλληλους αλγορίθμους μπορεί να κάνει κανείς με τη χρήση της εντολής:

- nvcc main.cu -o main

Compilation για τον αλγόριθμο CUSPARSE μπορεί να κάνει κανείς με τη χρήση της εντολής:

- nvcc main.cu tools.cu -lcusparse -lcudart -o main

4 Αποτελέσματα.

Για τη μελέτη των αλγορίθμων χρησιμοποιήθηκαν τα παρακάτω δεδομένα:

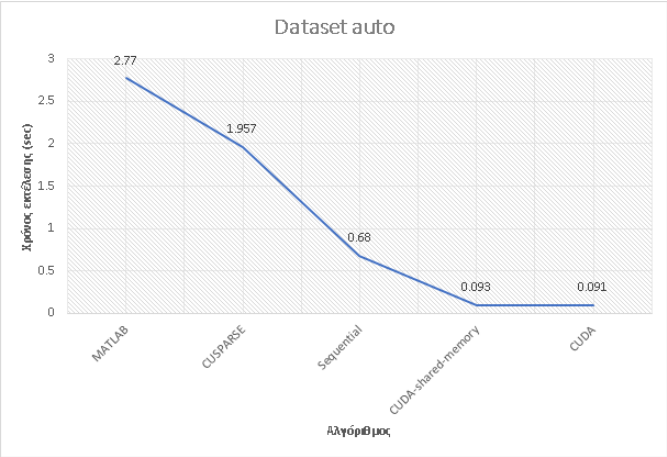
- auto : https://sparse.tamu.edu/DIMACS10/auto
- great-britain_osm : https://sparse.tamu.edu/DIMACS10/great-britain_osm
- delaunay_n22 : https://sparse.tamu.edu/DIMACS10/delaunay_n22
- delaunay_n23 : https://sparse.tamu.edu/DIMACS10/delaunay_n23

Οι δοκιμές έγιναν σε υπολογιστή με επεξεργαστή **Intel® Core™ i5-4690K** ο οποίος έχει ταχύτητα ρολογιού 3.50GHz με 4 πυρήνες. Η κάρτα γραφικών που χρησιμοποιήθηκε ήταν η **GeForce GTX 650 Ti**.

Στοιχεία των datasets.

Dataset	Number of Rows	Non Zero Elements	Density	Triangles
auto	448695	6629222	$3.29 * 10^{-5}$	6245184
great-britain_osm	7733822	16313034	$2.72 * 10^{-7}$	10908
delaunay_n22	4194304	25165738	$1.43 * 10^{-6}$	8436672
delaunay_n23	8388608	50331568	$7.15 * 10^{-7}$	16873359

Αρχικά παρουσιάζονται οι χρόνοι εκτέλεσης των αλγορίθμων που παρουσιάσθηκαν παραπάνω για το dataset auto.

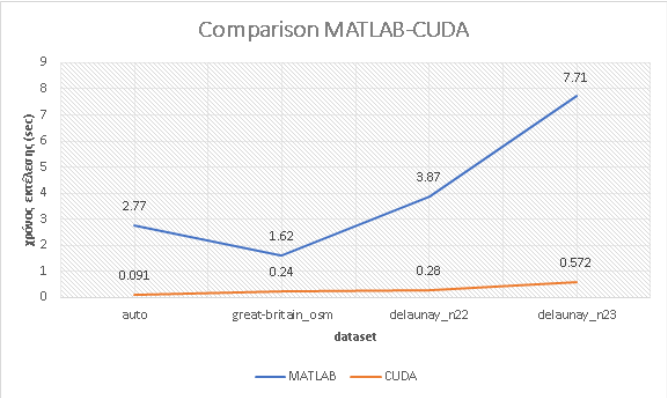


Σχήμα 1: Χρόνοι εκτέλεσης αλγορίθμων στο dataset auto.

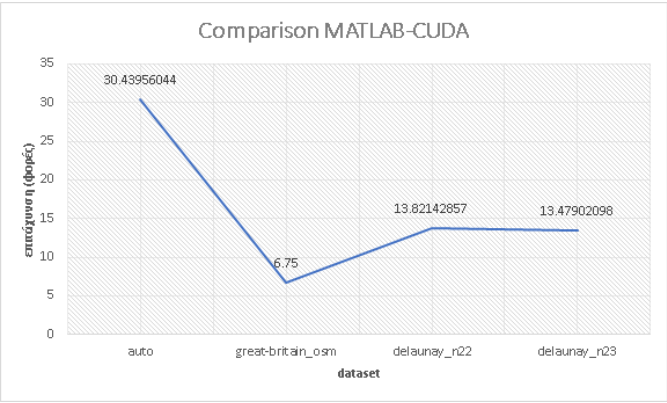
΄Οπως φαίνεται παραπάνω ο αλγόριθμος στο MATLAB είναι ο πιο αργός, ενώ ο αλγόριθμος σε CUSPARSE είναι μόλις **1.41** φορές πιο γρήγορος από αυτόν σε MATLAB. Ο σειριακός αλγόριθμος που υλοποιήθηκε είναι **4.07** φορές πιο γρήγορος από αυτόν σε MATLAB. Οι παράλληλοι αλγόριθμοι σε CUDA τόσο με τη χρήση Shared Memory όσο και χωρίς παρουσιάζουν ίδιο περίπου χρόνο εκτέλεσης.

Το γεγονός αυτό οφείλεται στο μικρό μέγεθος των στοιχείων που φορτώνεται στη Shared Memory σε κάθε νέο block. Ο χρόνος που κερδίζεται από τη χρήση της Shared Memory χάνεται από τη διαδικασία φόρτωσης των δεδομένων σε αυτή. Η επιτάχυνση που παρουσιάζουν οι αλγόριθμοι σε CUDA είναι περίπου **30** φορές πιο γρήγοροι από αυτόν στο MATLAB και **7.4** φορές πιο γρήγοροι από τον σειριακό.

Στη συνέχεια παρουσιάζονται οι χρόνοι εκτέλεσης και των τεσσάρων datasets στο MATLAB και στην CUDA.



Σχήμα 2: Χρόνοι εκτέλεσης MATLAB/CUDA.



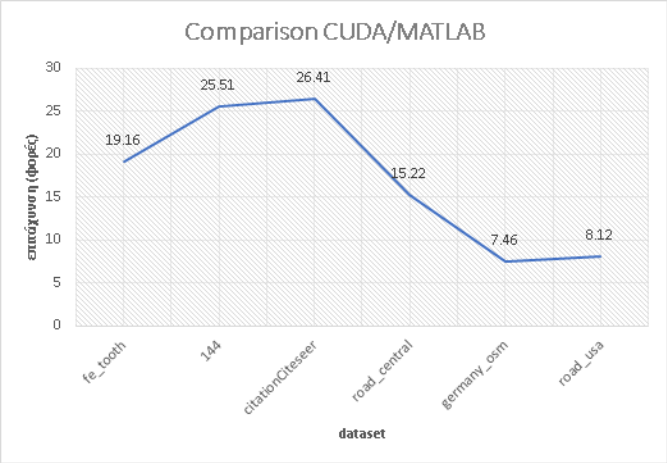
Σχήμα 3: Επιτάχυνση αλγορίθμου MATLAB/CUDA.

΄Οπως γίνεται αντιληπτό από τα παραπάνω διαγράμματα η επιτάχυνση κυμαίνεται από **6.75** φορές έως και **30.43** φορές.

Προφανώς η **ορθότητα του αλγορίθμου** επιβεβαιώνεται από το σωστό υπολογισμό των τριγώνων ο οποίος συμπίπτει με τον υπολογισμό των τριγώνων στο MATLAB.

΄Αλλα Datasets που μελετήθηκαν:

Dataset	MATLAB (sec)	CUDA (sec)	SPEED UP	N	NNZ
fe_tooth [2]	0.23	0.012	19.16	78136	905182
144 [3]	0.74	0.029	25.51	144649	2148786
citationCiteseer [4]	3.54	0.134	26.41	268495	2313294
road_central [5]	7.46	0.49	15.22	14081816	33866826
germany_osm [6]	2.8	0.375	7.46	11548845	24738362
road_usa [7]	6.09	0.75	8.12	23947347	57708624



Σχήμα 4: Επιτάχυνση αλγορίθμου MATLAB/CUDA.

Οι διαφορές που εντοπίζονται στους χρόνους επιτάχυνσης οφείλονται στο utilazation που πραγματοποιείται κάθε φορά. Δηλαδή, ένα πυκνότερο dataset με μέσο όρο μη μηδενικών στοιχείων ανα γραμμή κοντά στο 32 (όσος και ο βέλτιστος αριθμός παράλληλων thread) επιτυγχάνει καλύτερο utilization και συνεπώς καλύτερη απόδοση. Αντίθετα, ένας πολύ αραιός πίνακας με μέσο όρο 2 στοιχεία ανα γραμμή παρουσιάζει κάκιστο utilization καθώς τα υπόλοιπα 30 threads παραμένουν αναξιοποίητα σε κάθε επανάληψη, άρα και κακή επιτάχυσνη.

5 Επίλογος.

Στο έγγραφο αυτό παρουσιάσθηκαν αλγόριθμοι εύρεσης του πλήθους των τριγώνων ενός απλού μη κατευθυνόμενου γράφου. Ο βέλτιστος αλγόριθμος που υλοποιήθηκε, υλοποιήθηκε σε CUDA και κατάφερε να επιτύχει βελτίωση του χρόνου εκτέλεσης από **6.75** έως **30.43** φορές, έναντι του αλγορίθμου 1 στο MATLAB. Στις δοκιμές που πραγματοποιήθηκαν μελετήθηκαν πίνακες μεγέθους από $\mathbf{2^{16}}$ έως και $\mathbf{2^{25}}$ με $\mathbf{2^{20}}$ έως $\mathbf{2^{26}}$ μη μηδενικά στοιχεία.

Αναφορές

[1] Wikipedia Sparse Matrix. https://en.wikipedia.org/wiki/Sparse_matrix.

[2] fe_tooth graph. https://sparse.tamu.edu/DIMACS10/fe_tooth.

[3] 144 graph. https://sparse.tamu.edu/DIMACS10/144.

[4] citationCiteseer graph. https://sparse.tamu.edu/DIMACS10/citationCiteseer.

[5] road_central graph. https://sparse.tamu.edu/DIMACS10/road_central.

[6] germany_osm graph. https://sparse.tamu.edu/DIMACS10/germany_osm.

[7] road_usa graph. https://sparse.tamu.edu/DIMACS10/road_usa.