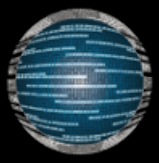


Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πανεπιστήμιο Πατρών
23 Νοεμβρίου 2020

Παράλληλος Προγραμματισμός
1ο *Εργαστήριο*

Ανδρέας Καρατζάς

Περιεχόμενα

1

Ο αλγόριθμος K-Means

1.1 Εισαγωγή

Για την 1η εργαστηριακή άσκηση, υλοποιήθηκε ο αλγόριθμος K-Means. Ο K-Means είναι ένας αλγόριθμος που χρησιμοποιείται για συσταδοποίηση στο χώρο του Big Data. ο λόγος είναι ότι έχει πολύ καλή απόδοση. Η πολυπλοκότητά του είναι $O(iterations \cdot dataset \cdot K \cdot dimentions)$. Ο ψευδοκώδικας που θα μπορούσε να περιγράψει τον K-Means είναι ο Αλγόριθμος 1.1.

Αλγόριθμος 1.1 K-Means

```
1  input: Vec[N][Nv], Center[Nc][Nv], Classes[N], double threshold
2  output: Center
3  begin
4          while ∑_{i=0}^{N} ∑_{j=0}^{Nc} euclidean distance(Vec[i], Center[j]) > threshold:
5                  Classes ← j | ⇒ argmin euclidean distance(Vec[i], Center[j]) ∀i ∈ N & j ∈ Nc
6                  foreach Center[i]:
7                          Center[i] ← Vec[j] ∀j | ⇒ Vec[j] ∈ Classes[i]
8          return Center
9  end
```

Στα πλαίσια της εργασίας έγιναν αρκετές βελτιστοποιήσεις. Ωστόσο, ο αλγόριθμος σχεδιάστηκε σε ένα γενικό πλαίσιο και χωρίς τοπικότητα, δηλαδή δεν έγιναν παραδοχές. Για τον έλεγχο του αλγορίθμου οπτικά, φτιάχτηκε και ένα πρόγραμμα σε γλώσσα Python, το οποίο θα αναλυθεί σε επόμενο κεφάλαιο. Στα επόμενα κεφάλαια, θα αναλυθεί η δομή της εργασίας και θα απαντηθούν τα ζητούμενα που τίθενται στην εκφώνηση. Η εργασία υλοποιήθηκε σε `C++ 17` με σκοπό την δημοσίευση του κώδικα στην πλατφόρμα του GitHub χρησιμοποιώντας τον IDE του Visual Studio 2019 και το μεταγλωττιστή Intel 19.1. Το σύστημα στο οποίο υλοποιήθηκε ο αλγόριθμος έχει λειτουργικό Windows 10, επεξεργαστή Intel Core i7 9ης γενιάς και μνήμη RAM των 16 GB. Για την εκτέλεση του κώδικα σε λειτουργικό Unix, συντάχθηκε ένα αρχείο makefile το οποίο βρίσκεται στο ίδιο directory με τα αρχεία κώδικα.

1.2 Ρυθμίζοντας το Visual Studio 2019

Αρχικά, σε περίπτωση που το μηχάνημα στο οποίο θα ελεγχθεί η εργασία διαθέτει επεξεργαστή Intel, προτείνεται η εγκατάσταση του Compiler της Intel, καθώς παρέχει επιπλέον βελτιστοποίηση κώδικα για επεξεργαστές της αρχιτεκτονικής τους.

Έπειτα, θα πρέπει να γίνει cloning του repository από το GitHub. Εναλλακτικά, μπορούν να προστεθούν χειροκίνητα τα αρχεία που βρίσκονται στο συμπιεσμένο. Στο συμπιεσμένο υπάρχει και ένα αρχείο *makefile* που μπορεί να αξιοποιηθεί ιδιαιτέρως σε περιβάλλον Unix. Στο περιβάλλον Visual Studio 2019, θα πρέπει να γίνουν αλλαγές στις ρυθμίσεις του Project επιλέγοντας `Project > Properties` και ακολουθώντας τα παρακάτω βήματα:

- `Configuration Properties > General > Platform Toolset` → `Intel C++ Compiler 19.1`
- `Configuration Properties > General > C++ Language Standard` → `ISO C++ 17 Standard (std::c++17)`
- `Configuration Properties > C++ > Optimization > Optimization` → `Highest Optimizations (/O3)`
- `Configuration Properties > C++ > Optimization > Inline Function Expansion` → `Any Suitable (/Ob2)`
- `Configuration Properties > C++ > Optimization > Enable Intrinsic Functions` → `Yes (/Oi)`
- `Configuration Properties > C++ > Optimization > Favor Size Or Speed` → `Favor fast code (/Ot)`
- `Configuration Properties > C++ > Optimization > Enable Fiber-Safe Optimizations` → `Yes (/GT)`
- `Configuration Properties > C++ > Optimization [Intel C++] > Optimize For Windows Application` → `Yes (/GA)`
- `Configuration Properties > C++ > Code Generation > Enable String Pooling` → `Yes (/GF)`
- `Configuration Properties > C++ > Code Generation > Enable C++ Exceptions` → `Yes (/EHsc)`
- `Configuration Properties > C++ > Code Generation > Floating Point Model` → `Fast=2 (/fp:fast=2)`
- `Configuration Properties > C++ > Code Generation > Enable Floating Point Exceptions` → `No`
- `Configuration Properties > C++ > Code Generation[Intel C++] > Floating Point Expression Evaluation` → `Double (/fp:double)`
- `Configuration Properties > C++ > Code Generation[Intel C++] > Intel Processor-Specific Optimization` → `Same as the host processor performing the compilation (/QxHost)`

- `Configuration Properties > C++ > Language > C++ Language Standard` → `ISO C++ 17 Standard (/std:c++17)`
- `Configuration Properties > C++ > Language[Intel C++] > C/C++ Language Support` → `C++ 17 Support (/Qstd=c++17)`

1.3 Η δομή της εργασίας

Ο κώδικας χωρίστηκε σε header files. Στα διάφορα header files δηλώθηκαν συναρτήσεις και μεταβλητές, που ορίζονται στα αντίστοιχα CPP αρχεία κώδικα. Το repository του αλγορίθμου βρίσκεται σε αυτό το link. Στο repository υπάρχει ο φάκελος k-means στον οποίο υπάρχουν όλα τα αρχεία που γράφηκαν στα πλαίσια της εργασίας, μαζί με τα αρχεία `k-means.vcxproj` και `k-means.vcxproj.filters`, τα οποία είναι αρχεία του Visual Studio.

1.3.1 Ο φάκελος data

Στον φάκελο data αποθηκεύονται τα δεδομένα όπως αυτά υπολογίστηκαν από τον αλγόριθμο K-Means. Τα δεδομένα αυτά αποτυπώνουν το τυχαίο dataset, αλλά και τις αλλαγές που υπάρχουν στις μεταβλητές Classes και Centers. Ο χρήστης θα πρέπει να μετακινεί ό,τι αρχεία βρίσκονται μέσα στο συγκεκριμένο φάκελο. Αν και έγινε προσπάθεια σύνταξης κώδικα για να επιλυθεί το παραπάνω πρόβλημα χρησιμοποιώντας τη βιβλιοθήκη filesystem, ο compiler δεν κατάφερε να παράξει εκτελέσιμο.

1.3.2 `Common.h`

Στην κεφαλίδα `Common.h` δηλώνονται οι απαραίτητες βιβλιοθήκες για την εκτέλεση του προγράμματος, αλλά και σταθερές που καθορίζουν τη λειτουργία του αλγορίθμου. Αναλύοντας:

- `VERBOSITY`: Η σταθερά χρησιμεύει κατά την κλήση της συνάρτησης `kmeans_progress` που είναι δηλωμένη στην κεφαλίδα `Interface`. Η σταθερά αυτή μπορεί να πάρει τιμή 0, 1, 2 ή 3. Όσο πιο μεγάλη τιμή έχει, τόσο μεγαλύτερη θα είναι και η συχνότητα εκτύπωσης των αποτελεσμάτων του kmeans κατά τη διαδικασία της βελτιστοποίησης των συστάδων
- `TEST_MODE`: Η σταθερά αυτή χρησιμεύει κατά τη διαδικασία του testing του προγράμματος. Όταν αρχικοποιείται με 1, το πρόγραμμα χρησιμοποιεί σύνολο δεδομένο με πληθάριθμο 30 αντί για $100,000$ που ορίζεται μέσα στην εκφώνηση. Επίσης, οι διαστάσεις είναι 2 και τα clusters 3. Όταν είναι αρχικοποιημένη με 0, τότε το πρόγραμμα λειτουργεί με τα δεδομένα της εκφώνησης

- `MAX_LIMIT`: Η σταθερά αυτή καθορίζει το μέγιστο που μπορεί να πάρει οποιοδήποτε από τα στοιχεία του συνόλου δεδομένων μας (μεταβλητή `Vec`) έτσι ώστε να λυθεί το πρόβλημα της υπερχείλισης δεδομένων τύπου float. Σε περίπτωση που γίνεται testing, η σταθερά παίρνει χαμηλότερη τιμή έτσι ώστε να διευκολύνει τον προγραμματιστή

1.3.3 `Vec.h` και `Vec.cpp`

Στο αρχείο `Vec.h` γίνεται η δήλωση της συνάρτησης `vec_init` η οποία καλείται στην αρχή του προγράμματός μας για να γεμίσει τη μεταβλητή Vec που αποτελεί το dataset του αλγορίθμου. Στο αρχείο `Vec.cpp`, η παραπάνω συνάρτηση εμπλουτίζεται με τον απαραίτητο κώδικα.

1.3.4 `Center.h` και `Center.cpp`

Στο αρχείο `Center.h` γίνεται η δήλωση της συνάρτησης `init_centers` η οποία καλείται στην αρχή του προγράμματός μας για να αρχικοποιηθούν τα clusters (μεταβλητή Center[1]). Στο αρχείο `Center.cpp`, η παραπάνω συνάρτηση εμπλουτίζεται με τον απαραίτητο κώδικα.

1.3.5 `Class.h` και `Class.cpp`

Στο αρχείο `Class.h` γίνεται η δήλωση της συνάρτησης `compute_classes` η οποία καλείται κάθε φορά που πρέπει να αντιστοιχιστούν τα δεδομένα του αλγορίθμου (dataset) με το σωστό cluster (μεταβλητή `old_Center`). Στο αρχείο `Class.cpp`, η παραπάνω συνάρτηση εμπλουτίζεται με τον απαραίτητο κώδικα.

1.3.6 `Optimize.h` και `Optimize.cpp`

Στο αρχείο `Optimize.h` γίνεται η δήλωση της συνάρτησης `optimize_center` η οποία καλείται σε κάθε επανάληψη του αλγορίθμου και επιστρέφει τα καινούργια διανύσματα - συστάδες (μεταβλητή new_Center). Η συνάρτηση αυτή καλείται μετά την αντιστοίχιση δεδομένων με συστάδες, για να βελτιστοποιηθούν τα διανύσματα των συστάδων. Στο αρχείο `Optimize.cpp`, η παραπάνω συνάρτηση εμπλουτίζεται με τον απαραίτητο κώδικα.

1.3.7 `Distance.h` και `Distance.cpp`

Στο αρχείο `Distance.h` γίνεται η δήλωση των παρακάτω συναρτήσεων:

- `eucl_diff`: Η συνάρτηση αυτή δέχεται στην είσοδό της 2 διανύσματα διαστάσεων N_v και επιστρέφει την Ευκλείδεια απόστασή τους

[1] Στο πρόγραμμα αναφέρεται ως `old_Center`

- `convergence`: Η συνάρτηση αυτή δέχεται στην είσοδό της τις συστάδες από δύο διαδοχικές επαναλήψεις και επιστρέψει την αθροιστική Ευκλείδεια απόστασή τους. Η συνάρτηση καλείται για να αποφανθεί ο αλγόριθμος σύγκλισης
- `normalize_convergence`: Η συνάρτηση αυτή δέχεται την προηγουμένως υπολογισμένη αθροιστική Ευκλείδεια απόσταση δύο διαδοχικών συστάδων και την κανονικοποιεί σύμφωνα με το μαθηματικό τύπο:

$$\frac{|A - B|}{\max A, B}$$

  Ωστόσο, αν ο αλγόριθμος είναι ακόμα στις πρώτες επαναλήψεις, όπου η Ευκλείδεια απόσταση αλλάζει ραγδαία, τότε η κανονικοποιημένη τιμή θεωρείται ίση με 1 (μέγιστη κανονικοποιημένη τιμή). Τέλος, για να λυθεί το πρόβλημα του υπολογιστικού σφάλματος λόγω έλλειψης επαρκούς ακρίβειας σε αριθμούς κινητής υποδιαστολής, εισάγεται ένας επιπλέον κανόνας: Αν η μη κανονικοποιημένη αθροιστική Ευκλείδεια απόσταση δύο διαδοχικών συνόλων συστάδων είναι μικρότερη του 100, τότε θεωρείται ότι η επιτάχυνση των κέντρων είναι σχεδόν μηδενική[2] και άρα ο αλγόριθμος έχει συγκλίνει

Στο αρχείο `Distance.cpp`, οι συναρτήσεις αυτές εμπλουτίζονται με τον αντίστοιχο κώδικα.

1.3.8 `Interface.h` και `Interface.cpp`

Στο αρχείο `Interface.h` γίνεται η δήλωση των παρακάτω συναρτήσεων:

- `multidimentional_float_vector_interface`: Η συνάρτηση αυτή εκτυπώνει σε περιβάλλον CLI περιεχόμενα οποιασδήποτε μεταβλητής τύπου `std::vector<std::array<float, Nv>>`. Στο πρόγραμμα, χρησιμοποιείται για την εκτύπωση των περιεχομένων των μεταβλητών:
  - `Vec`,
  - `old_Center` και
  - `new_Center`
- `multidimentional_int_array_interface`: Η συνάρτηση αυτή εκτυπώνει σε περιβάλλον CLI περιεχόμενα οποιασδήποτε μεταβλητής τύπου `std::array<std::vector<int>, Nc>`. Στο πρόγραμμα, χρησιμοποιείται για την εκτύπωση των περιεχομένων της μεταβλητής `Classes`
- `progress_interface` και `kmeans_progress`: Η συνάρτηση `progress_interface` καλείται από τη συνάρτηση `kmeans_progress`, η οποία πληροφορεί το χρήστη για τη σχετική πρόοδο του αλγορίθμου

[2] Η επιτάχυνση των κέντρων είναι πολύ μικρή σε αυτήν την περίπτωση καθώς η μέση μετατόπιση σημείου μεταξύ των επαναλήψεων αυτών είναι μικρότερη του $\frac{100}{100 \cdot 1{,}000} = 0.001$.

- `kmeans_termination`: Η συνάρτηση αυτή καλείται στο τέλος του αλγορίθμου και εκτυπώνει τους χρόνους (benchmark) του αλγορίθμου και επιβεβαιώνει τη σύγκλισή του

Στο αρχείο `Interface.cpp`, οι συναρτήσεις αυτές εμπλουτίζονται με τον αντίστοιχο κώδικα.

1.3.9 `Benchmark.h` και `Benchmark.cpp`

Στο αρχείο `Benchmark.h` δηλώνονται οι συναρτήσεις:

- `init_benchmark`: Η συνάρτηση αυτή επιστρέψει ένα σημείο στο χρόνο (checkpoint), οπουδήποτε το χρειάζεται ο προγραμματιστής, για να μετρήσει χρόνο (benchmarking)
- `bench_convergence`: Η συνάρτηση αυτή θα μπορούσε να διαγραφεί, καθώς έχει ίδιο σώμα με τη συνάρτηση init_benchmark. Ωστόσο, λόγω της διαφοράς στο όνομα, βοηθάει τον προγραμματιστή να ξεχωρίζει το σκοπό της καθεμίας και συνεπώς το που πρέπει να κληθεί η καθεμία
- `bench_loop`: Η συνάρτηση αυτή μετράει το συνολικό χρόνο που βρίσκεται «εγκλωβισμένος» ο αλγόριθμος μέσα στην επανάληψη βελτιστοποίησης
- `terminate_bench`: Η συνάρτηση αυτή θα μπορούσε να διαγραφεί, καθώς έχει ίδιο σώμα με τη συνάρτηση init_benchmark και τη `bench_convergence`. Ωστόσο, λόγω της διαφοράς στο όνομα, βοηθάει τον προγραμματιστή να ξεχωρίζει το σκοπό της καθεμίας και συνεπώς το που πρέπει να κληθεί η καθεμία
- `benchmark_results`: Η συνάρτηση αυτή χρησιμοποιείται στο τέλος του προγράμματος κι επιτελεί συνολικό benchmarking του αλγορίθμου

Στο αρχείο `Benchmark.cpp`, οι συναρτήσεις αυτές εμπλουτίζονται με τον αντίστοιχο κώδικα.

1.3.10 `Validation.h` και `Validation.cpp`

Στο αρχείο `Validation.h` δηλώνονται οι συναρτήσεις:

- `export_multidimentional_float_vector`: Η συνάρτηση αυτή αποθηκεύει σε αρχείο CSV τα περιεχόμενα οποιασδήποτε μεταβλητής τύπου `std::vector<std::array<float, Nv>>`. Στο πρόγραμμα, χρησιμοποιείται για την εκτύπωση των περιεχομένων των μεταβλητών:
  - `Vec`,
  - `old_Center` και
  - `new_Center`

- `export_multidimentional_integer_array`: Η συνάρτηση αυτή αποθηκεύει σε αρχείο CSV τα περιεχόμενα οποιασδήποτε μεταβλητής τύπου `std::array<std::vector<int>, Nc>`. Στο πρόγραμμα, χρησιμοποιείται για την εκτύπωση των περιεχομένων της μεταβλητής `Classes`
- `track_kmeans_progress` και `export_kmeans_progress`: Η συνάρτηση `track_kmeans_progress` ενημερώνει τη μεταβλητή `kmeans_progress` την οποία αποθηκεύει σε αρχείο TXT η συνάρτηση `export_kmeans_progress`. Τα δεδομένα που εξάγονται αφορούν τη πρόοδο που κάνει ο αλγόριθμος K-Means από επανάληψη σε επανάληψη

Ο σκοπός αυτών των συναρτήσεων είναι εμφανής κατά το testing του προγράμματος. Τα δεδομένα που εξήχθησαν στο φάκελο `data` μετά από την εκτέλεση του αλγορίθμου, τα επεξεργάζεται το πρόγραμμα σε Python, το οποίο επιστρέφει τη γεωμετρική εικόνα της πορείας του αλγορίθμου. Στο αρχείο `Validation.cpp`, οι συναρτήσεις αυτές εμπλουτίζονται με τον αντίστοιχο κώδικα.

1.3.11 `kmeans.h` και `kmeans.cpp`

Στο αρχείο `kmeans.cpp` δηλώνονται όλες οι επιπλέον (custom) κεφαλίδες του προγράμματος. Στο αρχείο `kmeans.cpp` είναι γραμμένος ο driver του προγράμματος (συνάρτηση main).

1.4 Από το testing, στο scaling

Το πρόγραμμα όπως έχει ανέβει στο GitHub είναι αρχικοποιημένο για testing. Για να μπορέσει ο εκάστοτε χρήστης να δει την καθαρή απόδοση του προγράμματος[3], θα πρέπει να γίνει η ακόλουθη επεξεργασία:

- `Common.h > line 27 > TEST_MODE` ← 0
- `kmeans.cpp > lines [18, 19]` ← Comment
- `kmeans.cpp > lines [23, 24]` ← Comment
- `kmeans.cpp > line 36` ← Comment
- `kmeans.cpp > lines [38, 39]` ← Comment
- `kmeans.cpp > lines [41, 42]` ← Comment
- `kmeans.cpp > line 47` ← Comment
- `kmeans.cpp > lines [52, 53]` ← Comment

[3] Το πρόγραμμα φαίνεται αν είναι αποδοτικό ή όχι, αν οι μεταβλητές `N`, `Nv` και `Nc` αρχικοποιηθούν όπως περιγράφεται στο τέταρτο κομμάτι της εργασίας.

Σε περίπτωση που ο tester επιθυμεί να κάνει profiling ή να χρησιμοποιήσει την `time` λειτουργικού Unix, τότε θα πρέπει να κάνει επιπλέον των παραπάνω και τις παρακάτω αλλαγές:

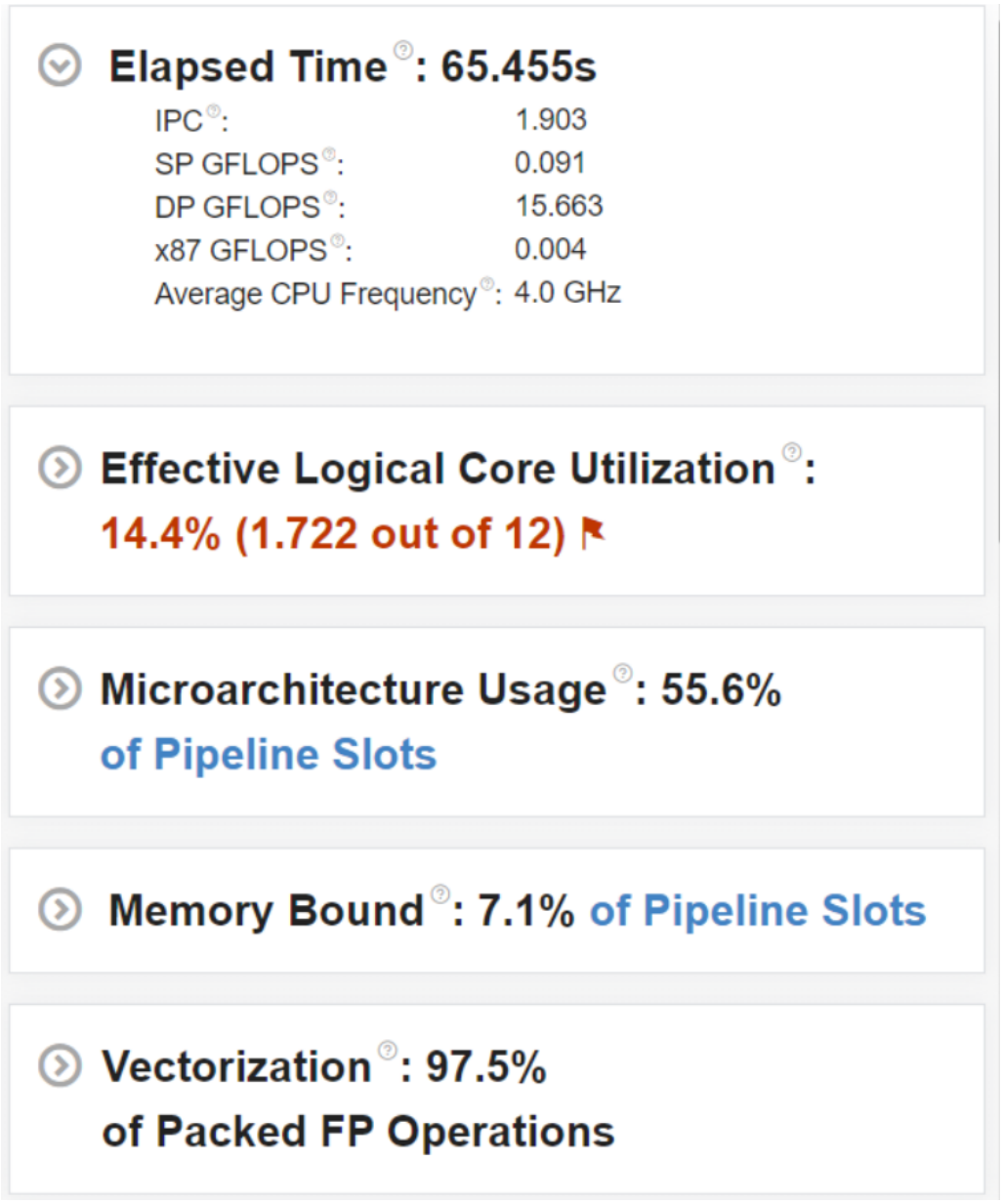
- `kmeans.cpp > lines [12, 14]` ← Comment
- `kmeans.cpp > line 30` ← Comment
- `kmeans.cpp > lines [33, 35]` ← Comment
- `kmeans.cpp > lines [49, 51]` ← Comment

1.5 Τα αποτελέσματα του Intel VTUNE Profiler

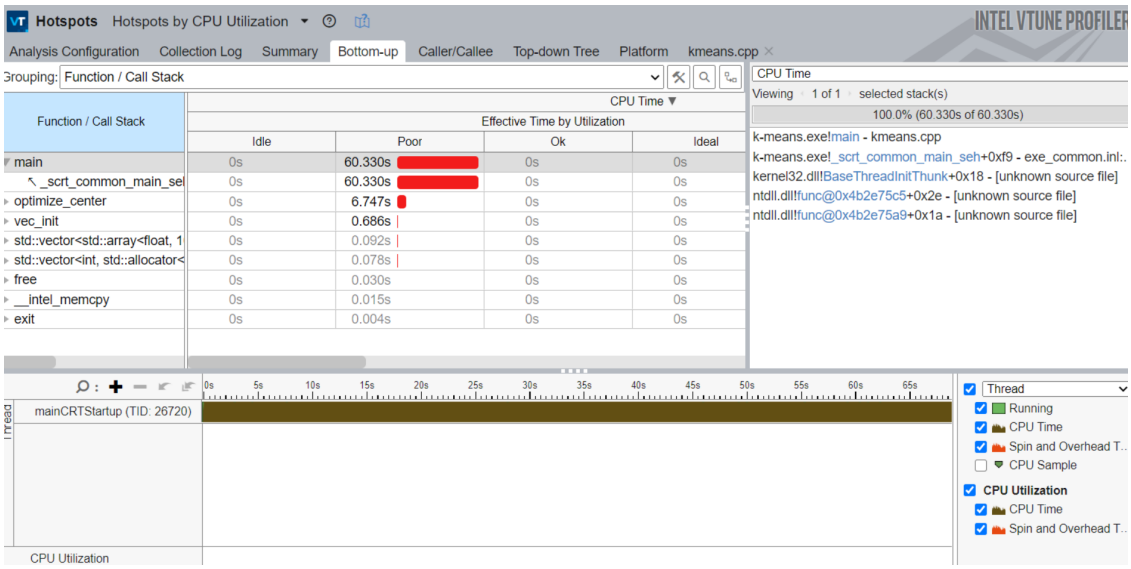
Το αποτέλεσμα του Intel VTUNE Profiler φαίνεται στο Σχήμα 1.1.

Το πρώτο στοιχείο που φαίνεται είναι ο χρόνος που χρειάστηκε το πρόγραμμα για την εκτέλεσή του. Ο χρόνος είναι μέσα στα επιθυμητά πλαίσια όπως αυτά ορίζονται στην εκφώνηση. Η κόκκινη ένδειξη δείχνει το πόση παραλληλοποίηση έγινε στον αλγόριθμο. Επειδή η άσκηση δε ζητούσε παραλληλοποίηση, το πρόγραμμα είναι 1 - *threaded*. Θετικά είναι τα ποσοστά *Memory Bound* που δείχνει την αποδοτικότητα των I/Os που έγιναν κατά την εκτέλεση του προγράμματος, και το Vectorization, το οποίο επιτρέπει block επεξεργασία.

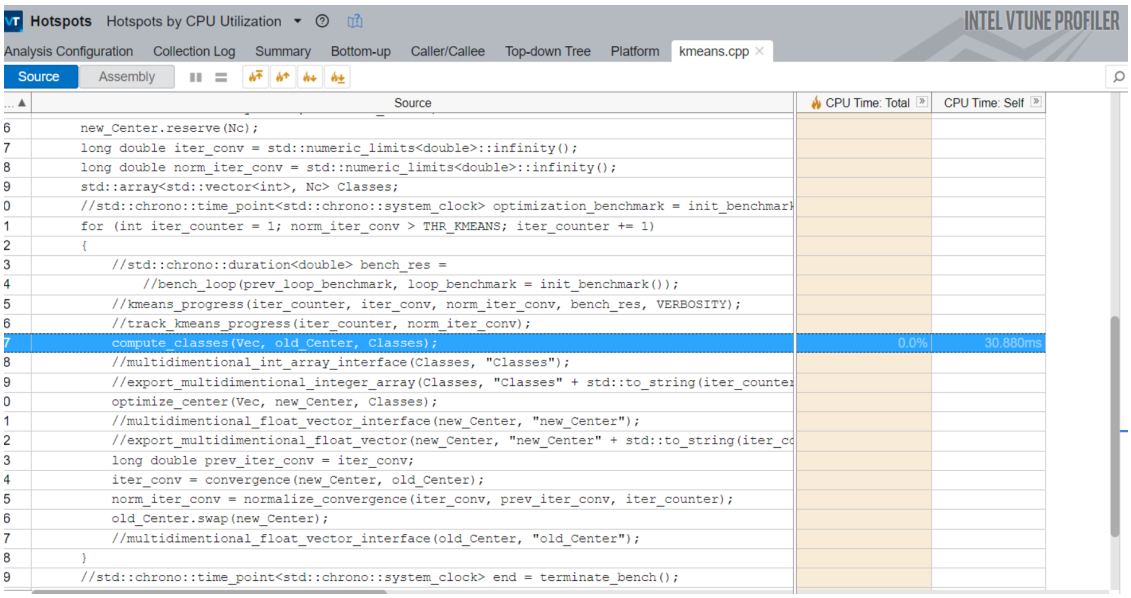
Μια λεπτομερέστερη εικόνα του προγράμματος δίνεται στα σχήματα 1.2 και 1.3, όπου φαίνεται ανά συνάρτηση η κατανάλωση χρόνου. Βλέπουμε ότι ο Profiler δίνει την main που είναι λογικό, καθώς αποτελεί το driver του προγράμματος. Ωστόσο, στο Σχήμα 1.3 ξεκαθαρίζει λίγο παραπάνω η κατανομή και πλέον δίνεται η συνάρτηση `compute_classes`, η οποία καταναλώνει σχεδόν το μισό χρόνο του προγράμματος. Αυτό είναι απολύτως λογικό, καθώς καλείται σε κάθε επανάληψη του K-Means και έχει περισσότερες πράξεις από κάθε άλλη συνάρτηση εντός της επανάληψης.



Σχήμα 1.1: Η ανάλυση του Intel VTUNE Profiler



Σχήμα 1.2: Η ανάλυση της *main*



Σχήμα 1.3: Το «hotspot» του προγράμματος

2

Το πρόγραμμα επικύρωσης σε Python

2.1 Περιγραφή

Για τον ευκολότερο έλεγχο του προγράμματος συντάχθηκε και ένα πρόγραμμα σε Python με περιβάλλον *PyCharm*. Το πρόγραμμα υπάρχει στο φάκελο `kmeans-visualize`. Το πρόγραμμα χρειάζεται τις βιβλιοθήκες:

- NumPy
- Pandas
- Matplotlib

Προαιρετικά, ο προγραμματιστής μπορεί να εγκαταστήσει και το πακέτο SciPy. Έπειτα, ο προγραμματιστής μετακινεί τα αρχεία επικύρωσης που δημιουργήθηκαν στο φάκελο `data` από τον αλγόριθμο K-Means στο φάκελο που υπάρχει και το Python Script. Τέλος, ο προγραμματιστής εκτελεί το Python Script.

Στο GitHub υπάρχουν και οδηγίες για το πως μπορεί ο προγραμματιστής να μετατρέψει αυτό το «stream» αρχείων PNG που δημιουργούνται μετά από την εκτέλεση του Script σε αρχείο GIF (βλέπε αρχείο «instructions»).

3

Ο κώδικας C++

3.1 Σχολιασμός

Στην αρχή η ιδέα ήταν να χρησιμοποιηθούν αποκλειστικά πίνακες για την αποθήκευση όλων των δεδομένων. Ωστόσο, αυτό αποδείχθηκε εξαιρετικά απαιτητικό ως προς το stack του προγράμματος. Έτσι, έγινε η αλλαγή σε δυναμικούς πίνακες που αποτελούνται από (στατικούς) πίνακες. Το αποτέλεσμα ήταν ότι κάθε φορά ήταν επαρκές να υπάρχει στη μνήμη ένα διάνυσμα 1000 στοιχείων, που θα αποτελούσε το διάνυσμα προς επεξεργασία από τη μεταβλητή `Vec`, ένα διάνυσμα επίσης 1000 στοιχείων που θα αποτελούσε κάποιο από τα κέντρα της μεταβλητής `old_Center` ή `new_Center` (ανάλογα με το σημείο εκτέλεσης) και οι δείκτες προς όλους τους πίνακες οι οποίοι όμως είναι αποθηκευμένοι στο heap. Στο φάκελο `buggy` υπάρχουν ενδεικτικά κάποιες ενδιάμεσες εκδόσεις που ικανοποιούν το ζητούμενο της εργασίας 4. Ωστόσο, δεν υπάρχουν σχόλια στις εκδόσεις αυτές, καθώς είναι κάποιες ειδικά έχουν αρκετά λανθασμένη αρχιτεκτονική.

3.2 Τα αρχεία

Κώδικας 3.2.1: Η κεφαλίδα Common.h

```cpp
/**
 * Common.h
 *
 * In this header file, we define the constants
 * used throughout the K-means algorithm. We also
 * include all the header files necessary to make
 * the implementation work.
 */

#pragma once

#include <iostream>                             ///< std::cout
#include <vector>                               ///< std::vector
#include <iomanip>                              ///< std::setw
#include <limits>                               ///< std::numeric_limits
#include <chrono>                               ///< std::chrono
#include <array>                                ///< std::array
#include <algorithm>                            ///< std::find
#include <utility>                              ///< std::pair
#include <string>                               ///< std::string
#include <cstdlib>                              ///< std::abs
#include <random>                               ///< std::random_device
#include <fstream>                              ///< std::ofstream
#include <numeric>                              ///< std::iota

constexpr int VERBOSITY = 3;                    ///< Sets K-means' iterface verbosity level
constexpr int TEST_MODE = 1;                    ///< Set it equal to 1 to enter test mode, otherwise set it to 0
constexpr int MAX_LIMIT = TEST_MODE == 1 ? 31 : 63; ///< Upper bound used in the randomly generated dataset of reals
constexpr int N = TEST_MODE == 1 ? 20 : 100000;     ///< Elements in our dataset
constexpr int Nv = TEST_MODE == 1 ? 2 : 1000;       ///< Dimentions of each element
constexpr int Nc = TEST_MODE == 1 ? 3 : 100;        ///< Number of clusters, also known as `K`
constexpr double THR_KMEANS = 0.000001;             ///< Threshold used to indicate K-means convergence
```

Κώδικας 3.2.2: Η κεφαλίδα Vec.h

```cpp
/**
 * Vec.h
 *
 * In this header file, we define a function that
 * executes only in the beginning of the algorithm.
 * This function is used to generate a random dataset
 * for the K-means algorithm.
 */

#pragma once

#include "Common.h"

void vec_init(std::vector<std::array<float, Nv>>& Vec);
```

Κώδικας 3.2.3: Το αρχείο Vec.cpp

```cpp
#include "Vec.h"

/**
 * Initializes `Vec` variable.
 *
 * @param[in, out] Vec the random dataset generated for clustering using the K-means algorithm.
 *
 * @see Common.h
 *
 * @remark [<random> Engines and Distributions](https://docs.microsoft.com/en-us/cpp/standard-library/random?view=msvc-160#engdist)
 */
void vec_init(std::vector<std::array<float, Nv>>& Vec)
{
	std::random_device rd_Vec;                                          ///< Initializes a random generator
	std::mt19937 mt_Vec(rd_Vec());                                          ///< Uses the mt19937 engine
	std::uniform_real_distribution<float> dist_Vec(0.0, MAX_LIMIT + 0.0);       ///< Generates a uniform distribution bounded by the `MAX_LIMIT` set in `Common.h` to prevent number overflow
	for (int i = 0; i < N; i += 1)
	{
		std::array<float, Nv> Elements;                                     ///< Declares a vector `Elements` that temporarily holds the vector that is to be inserted to `Vwc`
		for (int j = 0; j < Nv; j += 1)
		{
			Elements[j] = dist_Vec(mt_Vec);                             ///< Updates contents of `Elements`
		}
		Vec.emplace_back(Elements);                                     ///< Moves `Elements` to `Vec`
	}
}
```

Κώδικας 3.2.4: Η κεφαλίδα Center.h

```cpp
/**
 * Center.h
 *
 * In this header file, we define a function that helps
 * in the random initialization of the variable `Center`.
 * This function is only used in the beginning of the
 * K-means algorithm.
 */

#pragma once

#include "Common.h"
#include "Distance.h"

void init_centers(const std::vector<std::array<float, Nv>>& Vec, std::vector<std::array<float, Nv>>& old_Center);
```

Κώδικας 3.2.5: Το αρχείο Center.cpp

```cpp
#include "Center.h"

/**
 * Initializes `Center` variable.
 *
 * @param[in] Vec the dataset generated.
 * @param[in, out] old_Center the random centroids generated in the beginning of the K-means algorithm.
 *
 * @remark [<random> Engines and Distributions](https://docs.microsoft.com/en-us/cpp/standard-library/random?view=msvc-160#engdist)
 */
void init_centers(const std::vector<std::array<float, Nv>>& Vec, std::vector<std::array<float, Nv>>& old_Center)
{
	std::vector<int> Random(N);                                                                     ///< Declares our random `Vec` generator vector
	std::iota(Random.begin(), Random.end(), 0);                                                     ///< Initializes vector with all possible indeces of `Vec`
	std::shuffle(Random.begin(), Random.end(), std::mt19937{ std::random_device{}() });             ///< Shuffles `Random` vector
	for (int i = 0; i < Nc; i += 1)
	{
		std::array<float, Nv> Element;                                                          ///< Declares a vector `Elements` that temporarily holds the vector chosen from `Vec`
		Element = Vec.at(Random.at(i));                                                         ///< Copies contents of the random `Vec` element
		old_Center.emplace_back(Element);                                                       ///< Moves `Element` variable to the `old_Center` variable
	}
}
```

Κώδικας 3.2.6: Η κεφαλίδα Class.h

```h
/**
 * Class.h
 *
 * In this header file, we a function that assigns
 * the vectors found in the variable `Vec` to a
 * corresponding `Class` or more commonly known as cluster.
 */

#pragma once

#include "Common.h"
#include "Distance.h"

void compute_classes(const std::vector<std::array<float, Nv>>& Vec, const std::vector<std::array<float, Nv>>& old_Center, std::array<std::vector<int>, Nc>& Classes);
```

Κώδικας 3.2.7: Το αρχείο Class.cpp

```cpp
#include "Class.h"

/**
 * Computes `Class` variable.
 *
 * @param[in] Vec the dataset generated.
 * @param[in] old_Center the current centroids.
 * @param[in, out] Classes the classes corresponding to the dataset.
 *
 * @note This function calls `eucl_diff` from Distance.h
 *
 * @see [K-means clustering Algorithm](https://medium.com/@jaredchilders_38839/k-means-clustering-algorithm-4334db89bdf3)
 */
void compute_classes(const std::vector<std::array<float, Nv>>& Vec, const std::vector<std::array<float, Nv>>& old_Center, std::array<std::vector<int>, Nc>& Classes)
{
	for (int i = 0; i < Nc; i += 1)                                                 ///< Clears contents of `Classes` variable
	{
		Classes[i].clear();                                                     ///< Prevents garbage processing
		Classes[i].reserve((int) N/Nc);                                         ///< Optimizes array with high probability
	}
	int argmin_idx = -1;                                                            ///< Initializes \argmin index
	long double argmin_val = std::numeric_limits<long int>::max() + 0.0;            ///< Initializes \argmin value
	for (int i = 0; i < N; i += 1)                                                  ///< Loop through `Vec`
	{
		for (int j = 0; j < Nc; j += 1)                                         ///< Loop through `old_Center`
		{
			long double temp_eucl_dist = eucl_diff(Vec.at(i), old_Center.at(j));    ///< Compute Euclidean distance between parsed `Vec` element and parsed `old_Center` element
			if (argmin_val > temp_eucl_dist)                                ///< Checks if this Euclidean distance is so far the smallest
			{
				argmin_val = temp_eucl_dist;                            ///< Updates \argmin value
				argmin_idx = j;                                         ///< Updates \argmin index
			}
		}
		Classes[argmin_idx].emplace_back(i);                                    ///< Assigns `Vec` element to a cluster
		argmin_idx = -1;                                                        ///< Sets \argmin index for the next loop
		argmin_val = std::numeric_limits<unsigned long int>::max();             ///< Sets \argmin value for the next loop
	}
}
```

Κώδικας 3.2.8: Η κεφαλίδα Optimize.h

```h
/**
 * Optimize.h
 *
 * In this header file, we define a function which
 * computes the error of the `Center` variable with
 * respect to the `Vec` variable.
 */

#pragma once

#include "Common.h"

void optimize_center(const std::vector<std::array<float, Nv>>& Vec, std::vector<std::array<float, Nv>>& new_Center, const std::array<std::vector<int>, Nc>& Classes);
```

Κώδικας 3.2.9: Το αρχείο Optimize.cpp

```cpp
#include "Optimize.h"

/**
 * Computes `Center` variable after `Classes` correction.
 *
 * @param[in] Vec the dataset.
 * @param[in] Classes the classes corresponding to the dataset.
 * @param[in, out] new_Center the new clusters of the K-means algorithm.
 */
void optimize_center(const std::vector<std::array<float, Nv>>& Vec, std::vector<std::array<float, Nv>>& new_Center, const std::array<std::vector<int>, Nc>& Classes)
{
	new_Center.clear();                                                             ///< Clears previously computed data
	for (int i = 0; i < Nc; i += 1)                                                 ///< Loops through all clusters
	{
		std::array<float, Nv> Element = { 0 };                                  ///< Initializes `Element` array which holds a cluster vector
		for (int j = 0; j < Classes[i].size(); j += 1)                          ///< Loops through `Vec` vectors
		{
			for (int k = 0; k < Nv; k += 1)                                 ///< Loops through vector elements
			{
				Element[k] += (float)(Vec.at(Classes[i].at(j))[k] / Classes[i].size());  ///< Computes vector element
			}
		}
		new_Center.emplace_back(Element);                                       ///< Updates `new_Center` variable
	}
}
```

Κώδικας 3.2.10: Η κεφαλίδα Distance.h

```h
/**
 * Distance.h
 *
 * In this header file, we define some functions which
 * help with the computations needed to define whether
 * K-means has converged or not.
 */

#pragma once

#include "Common.h"

long double eucl_diff(const std::array<float, Nv>& src, const std::array<float, Nv>& dst);
long double convergence(const std::vector<std::array<float, Nv>>& curr_Center, const std::vector<std::array<float, Nv>>& prev_Center);
long double normalize_convergence(const long double curr_iter_conv, const long double prev_iter_conv, const int iter_counter);
```

Κώδικας 3.2.11: Το αρχείο Distacne.cpp

```cpp
#include "Distance.h"

/**
 * Computes Euclidean distance between 2 arrays of size `Nv`.
 *
 * @param[in] A the first array.
 * @param[in] B the second array.
 *
 * @return Euclidean distance.
 */
long double eucl_diff(const std::array<float, Nv>& A, const std::array<float, Nv>& B)
{
	long double eucl_diff = 0;
	for (int i = 0; i < Nv; i += 1)                                                    ///< Loops through all `Nv` elements of the given arrays
	{
		long double point_diff = A[i] - B[i];                                      ///< Computes element-wise difference
		if (point_diff > std::numeric_limits<float>::max())                        ///< Checks for data overflow
		{
			point_diff = std::numeric_limits<float>::max();                    ///< Prevents data overflow
		}
		long double point_diff_squared = point_diff * point_diff;                  ///< Squares difference
		eucl_diff += point_diff_squared;                                           ///< Adds difference to `eucl_diff`
	}
	return eucl_diff;
}

/**
 * Checks if K-means has converged.
 *
 * @param[in] curr_Center the current `Center` variable .
 * @param[in] prev_Center the old `Center` variable.
 *
 * @return Euclidean distance between the 2 vectors.
 *
 * @note if the Euclidean distance between 2 points is not greater than 1.0, then it is normalized to 0.
 *               This is to optimize speed, prevent overfitting, and solve the precision error carried throughout the $Nc \times Nv = 1{,}000{,}000$ additions.
 */
long double convergence(const std::vector<std::array<float, Nv>>& curr_Center, const std::vector<std::array<float, Nv>>& prev_Center)
{
	long double convergence_sum = 0;                                           ///< Initializes `convergence_sum` variable used to store the total additive difference between `curr_Center` and `prev_Center`
	for (int i = 0; i < Nc; i += 1)                                                    ///< Loops through all centers
	{
		long double tmp_eucl_d = eucl_diff(curr_Center.at(i), prev_Center.at(i));  ///< Computes element-wise Euclidean distance
		convergence_sum += tmp_eucl_d > 1.0 ? tmp_eucl_d : 0.0;                    ///< Optimizes Convergence if error is close to defined threshold
		if (convergence_sum > std::numeric_limits<double>::max())                  ///< Checks for number overflow
		{
			convergence_sum = std::numeric_limits<double>::max();              ///< Prevents number overflow
			break;                                                             ///< Breaks loop because `convergence_sum` is maxed
		}
	}
	return convergence_sum;
}

/**
 * Normalizes the error between current `Center` variable and old `Center` variable.
 *
 * @param[in] curr_iter_conv the current `Center` error.
 * @param[in] prev_iter_conv the old `Center` error.
 * @param[in] iter_counter the loop number.
 *
 * @return normalized Euclidean distance between centers.
 *
 * @note The computation of the normalization formula is explained below:
 *               if the current loop error is less or equal to the number of clusters, then $normalized = 0.0$
 *               if the current iteration of the optimization loop is less or equal to 2, then $normalized = 1.0$
 *               else, $normalized = \frac{|curr_iter_conv - prev_iter_conv|}{\arg\max curr_iter_conv, prev_iter_conv}$
 */
long double normalize_convergence(const long double curr_iter_conv, const long double prev_iter_conv, const int iter_counter)
{
	return curr_iter_conv > Nc && iter_counter > 2 ? std::abs(curr_iter_conv - prev_iter_conv) / std::max(curr_iter_conv, prev_iter_conv) :
		iter_counter <= 2 ? 1.0 : 0.0;
}
```

Κώδικας 3.2.12: Η κεφαλίδα Interface.h

```cpp
/**
 * interface.h
 *
 * In this header file, we define some functions which
 * implement a basic User Interface (UI) throughout the
 * execution of the K-means algorithm.
 */

#pragma once

#include "Common.h"

void multidimentional_float_vector_interface(const std::vector<std::array<float, Nv>>& Obj, std::string Obj_name);
void multidimentional_int_array_interface(const std::array<std::vector<int>, Nc>& Obj, std::string Obj_name);
void progress_interface(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark);
void kmeans_progress(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark, const int verbose);
void kmeans_termination(const std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_results, const long double iter_conv);
```

Κώδικας 3.2.13: Το αρχείο Interface.cpp

```cpp
#include "Interface.h"

/**
 * Prints the contents of a vector of arrays.
 *
 * @param[in] Obj the vector of arrays.
 * @param[in] obj_name the name of the variable.
 *
 * @note This function helps in printing the contents of the variables `Vec`, `old_Centers` and `new_Centers`.
 */
void multidimentional_float_vector_interface(const std::vector<std::array<float, Nv>>& Obj, std::string Obj_name)
{
	for (int i = 0; i < Obj.size(); i += 1)
	{
		for (int j = 0; j < Obj.at(i).size(); j += 1)
		{
			std::cout << std::setw(11) << Obj_name << "[" << i << "][" << j << "]:\t" << std::setw(18) << Obj.at(i)[j] << " ";
		}
		std::cout << std::endl;
	}
	std::cout << std::endl << std::endl;
}

/**
 * Prints the contents of an array of vectors.
 *
 * @param[in] Obj the array of vectors.
 * @param[in] obj_name the name of the variable.
 *
 * @note This function helps in printing the contents of the variable `Classes`.
 */
void multidimentional_int_array_interface(const std::array<std::vector<int>, Nc>& Obj, std::string Obj_name)
{
	for (int i = 0; i < Obj.size(); i += 1)
	{
		for (int j = 0; j < Obj[i].size(); j += 1)
		{
			std::cout << Obj_name << "[" << i << "][" << j << "]: " << std::setw(4) << Obj[i].at(j) << " ";
			if (j % 6 == 0 && j != 0)
			{
				std::cout << std::endl;
			}
		}
		std::cout << std::endl << std::endl;
	}
	std::cout << std::endl << std::endl;
}

/**
 * Prints the progress of the K-means algorithm.
 *
 * @param[in] iter_counter the current loop number.
 * @param[in] iter_conv the current loop error.
 * @param[in] norm_iter_conv the current normalized loop error.
 * @param[in] loop_benchmark the current loop benchmark
 */
void progress_interface(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark)
{
	std::cout << "Iteration: " << std::setw(2) << iter_counter
		<< "\tError: " << std::setw(std::numeric_limits<float>::max_digits10) << iter_conv
		<< "\tNormalized error: " << std::setw(std::numeric_limits<float>::max_digits10) << norm_iter_conv
		<< "\tLoop benchmark: " << std::setw(std::numeric_limits<double>::max_digits10) << loop_benchmark.count();
	TEST_MODE == 1 ? (std::cout << std::endl << std::endl << std::endl) : std::cout << std::endl;
}

/**
 * Calls `progress_interface` function to print K-means progress.
 *
 * @param[in] iter_counter the current loop number.
 * @param[in] norm_iter_conv the current normalized loop error.
 * @param[in] iter_conv the current loop error.
 * @param[in] loop_benchmark the current loop benchmark
 * @param[in] verbose the frequency used for activating the interface
 *                if equal to 0, then `progress_interface` is called every 4 loops
 *                if equal to 1, then `progress_interface` is called every 3 loops
 *                if equal to 2, then `progress_interface` is called every 2 loops
 *                if equal to 3, then `progress_interface` is called in every loop
 *
 * @see `progress_interface`
 */
void kmeans_progress(const int iter_counter, const long double iter_conv, const long double norm_iter_conv, const std::chrono::duration<double> loop_benchmark, const int verbose)
{
	switch (verbose)
	{
	case 0: if (iter_counter % 4 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	case 1: if (iter_counter % 3 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	case 2: if (iter_counter % 2 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	case 3: if (iter_counter % 1 == 0) { progress_interface(iter_counter, iter_conv, norm_iter_conv, loop_benchmark); }
		  break;
	default:
		break;
	}
}

/**
 * Prints the resuts of the K-means algorithm.
 *
 * @param[in] bench_results the benchmarking results.
 * @param[in] iter_conv the final normalized K-means error.
 *
 * @see `Benchmark.h`
 */
void kmeans_termination(const std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_results, const long double iter_conv)
{
	std::cout << std::endl
		<< std::setw(18) << "[TOTAL TIME] " << std::setw(15) << bench_results.first.count()
		<< " {IN SECONDS}" << std::endl
		<< std::setw(18) << "[K-MEANS BENCH] " << std::setw(15) << bench_results.second.count()
		<< " {IN SECONDS}" << std::endl
		<< std::setw(18) << "[CONVERGENCE] " << std::setw(15) << iter_conv << std::endl;
}
```

Κώδικας 3.2.14: Η κεφαλίδα Benchmark.h

```h
/**
 * Benchmark.h
 *
 * In this header file, we define some functions that help us
 * benchmark some crucial parts of the K-means algorithm. Using
 * these functions, we can benchmark the K-means runtime, as well as
 * how long did it take for our implementation to converge.
 */

#pragma once

#include "Common.h"

std::chrono::time_point<std::chrono::system_clock> init_benchmark(void);
std::chrono::time_point<std::chrono::system_clock> bench_convergence(void);
std::chrono::duration<double> bench_loop(std::chrono::time_point<std::chrono::system_clock>, std::chrono::time_point<std::chrono::system_clock>);
std::chrono::time_point<std::chrono::system_clock> terminate_bench(void);
std::pair<std::chrono::duration<double>, std::chrono::duration<double>> benchmark_results(std::chrono::time_point<std::chrono::system_clock> start, std::chrono::time_point<std::chrono::system_clock> loop_benchmark, std::chrono::time_point<std::chrono::system_clock> end);
```

Κώδικας 3.2.15: Το αρχείο Benchmark.cpp

```cpp
#include "Benchmark.h"

/**
 * Launches benchmark.
 *
 * @return the starting timepoint of the K-means algorithm.
 */
std::chrono::time_point<std::chrono::system_clock> init_benchmark(void)
{
        std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_clock::now();
        return start;
}

/**
 * Launches K-means optimization loop benchmark.
 *
 * @return the starting timepoint of the K-means optimization loop.
 *
 * @see kmeans.cpp
 */
std::chrono::time_point<std::chrono::system_clock> bench_convergence(void)
{
        std::chrono::time_point<std::chrono::system_clock> loop_benchmark = std::chrono::system_clock::now();
        return loop_benchmark;
}

/**
 * Computes per-loop benchmark.
 *
 * @param[in] startpoint loop starting time point
 * @param[in] endpoint loop ending time point
 *
 * @return optimization loop benchmark.
 *
 * @see kmeans.cpp
 */
std::chrono::duration<double> bench_loop(std::chrono::time_point<std::chrono::system_clock> startpoint, std::chrono::time_point<std::chrono::system_clock> endpoint)
{
        return endpoint - startpoint;
}

/**
 * Stops K-means benchmark.
 *
 * @return the ending timepoint of the K-means algorithm.
 */
std::chrono::time_point<std::chrono::system_clock> terminate_bench(void)
{
        std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now();
        return end;
}

/**
 * Computes K-means benchmarking results.
 *
 * @param[in] start the starting timepoint of the K-means algorithm.
 * @param[in] loop_benchmark the starting timepoint of the K-means optimization loop.
 * @param[in] end the ending timepoint of the K-means algorithm.
 *
 * @return a pair with the total execution time of the K-means algorithm in seconds
 *                and the total time in seconds that the algorithm needed to converge.
 */
std::pair<std::chrono::duration<double>, std::chrono::duration<double>> benchmark_results(
        const std::chrono::time_point<std::chrono::system_clock> start,
        const std::chrono::time_point<std::chrono::system_clock> loop_benchmark,
        const std::chrono::time_point<std::chrono::system_clock> end)
{
        std::chrono::duration<double> elapsed_seconds = end - start;
        std::chrono::duration<double> k_means_benchmark = end - loop_benchmark;
        std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_res;
        bench_res.first = elapsed_seconds;
        bench_res.second = k_means_benchmark;
        return bench_res;
}
```

Κώδικας 3.2.16: Η κεφαλίδα Validation.h

```h
/**
 * Validation.h
 *
 * In this header file, we define some functions which
 * help the developer validate the data computed.
 * This is done by exporting the data to CSV files.
 */

#pragma once

#include "Common.h"

void export_multidimentional_float_vector(const std::vector<std::array<float, Nv>> Obj, std::string filename);
void export_multidimentional_integer_array(const std::array<std::vector<int>, Nc> Obj, std::string filename);
void track_kmeans_progress(int iter_counter, long double norm_iter_conv);
void export_kmeans_progress(std::string filename);
```

Κώδικας 3.2.17: Το αρχείο Validation.cpp

```cpp
#include "Validation.h"

static std::vector<std::pair<int, long double>> kmeans_progress;                                    ///< Variable used to store K-means progress

/**
 * Exports the contents of a vector of arrays to a CSV file.
 *
 * @param[in] Obj the vector of arrays.
 * @param[in] filename the name of file.
 *
 * @note This function helps in exporting the contents of the variables `Vec`, `old_Centers` and `new_Centers`.
 */
void export_multidimentional_float_vector(const std::vector<std::array<float, Nv>> Obj, std::string filename)
{
	std::ofstream export_stream;                                                                    ///< Defines an output file stream
	export_stream.open("./data/" + filename + ".csv");                                              ///< Associates `export_stream` with a CSV file named after the `filename` variable
	for (int i = 0; i < Obj.size(); i += 1)                                                             ///< Loops through `Obj`
	{
		for (int j = 0; j < Obj.at(i).size(); j += 1)                                               ///< Loops through the elements of the array inside `Obj` that is being parsed
		{
			export_stream << Obj.at(i)[j] << (j < Obj.at(i).size() - 1 ? "," : "");                 ///< Exports element of that array to the `export_stream` file stream
		}
		export_stream << std::endl;
	}
	export_stream.close();                                                                          ///< Closes file stream
}

/**
 * Exports the contents of an array of vectors into a CSV file.
 *
 * @param[in] Obj the array of vectors.
 * @param[in] filename the name of file.
 *
 * @note This function helps in printing the contents of the variable `Classes`.
 */
void export_multidimentional_integer_array(const std::array<std::vector<int>, Nc> Obj, std::string filename)
{
	std::ofstream export_stream;                                                                    ///< Defines an output file stream
	export_stream.open("./data/" + filename + ".csv");                                              ///< Associates `export_stream` with a CSV file named after the `filename` variable
	for (int i = 0; i < Obj.size(); i += 1)                                                             ///< Loops through `Obj`
	{
		for (int j = 0; j < Obj[i].size(); j += 1)                                                  ///< Loops through the elements of the vector inside `Obj` that is being parsed
		{
			export_stream << Obj[i].at(j) << (j < Obj.at(i).size() - 1 ? "," : "");                 ///< Exports element of that vector to the `export_stream` file stream
		}
		export_stream << std::endl;
	}
	export_stream.close();                                                                          ///< Closes file stream
}

/**
 * Updates `kmeans_progress` variable, which will be used to export data about K-means progress.
 *
 * @param[in] iter_counter the current loop number.
 * @param[in] norm_iter_conv the current normalized loop error.
 */
void track_kmeans_progress(int iter_counter, long double norm_iter_conv)
{
	kmeans_progress.push_back(std::make_pair(iter_counter, norm_iter_conv));
}

/**
 * Exports the contents of the `kmeans_progress` variable into a CSV file.
 *
 * @param[in] filename the name of file.
 */
void export_kmeans_progress(std::string filename)
{
	std::ofstream export_stream;                                                                    ///< Defines an output file stream
	export_stream.open("./data/" + filename + ".txt");                                              ///< Associates `export_stream` with a CSV file named after the `filename` variable
	for (int i = 0; i < kmeans_progress.size(); i += 1)                                             ///< Loops through `kmeans_progress`
	{
		export_stream <<
			"Iteration: " << kmeans_progress.at(i).first <<
			"Convergence: " << kmeans_progress.at(i).second << std::endl;                           ///< Exports an element of `kmeans_progress` vector to the `export_stream` file stream
	}
	export_stream.close();                                                                          ///< Closes file stream
}
```

Κώδικας 3.2.18: Η κεφαλίδα kmeans.h

```
/**
 * kmeans.h
 *
 * In this header file, we include all the
 * custom made header files used in the
 * K-means algorithm.
 */

#pragma once

#include "Vec.h"
#include "Center.h"
#include "Class.h"
#include "Optimize.h"
#include "Distance.h"
#include "Interface.h"
#include "Benchmark.h"
#include "Validation.h"
```

Κώδικας 3.2.19: Το αρχείο kmeans.cpp

```
#include "kmeans.h"

/**
 * Implements K-means clustering algorithm.
 *
 * @return 0, if the executable was terminated normally.
 *
 */
int main(void)
{
	std::chrono::time_point<std::chrono::system_clock> loop_benchmark, prev_loop_benchmark, start;		///< Initializes loop-wise benchmark variables
	std::cout.precision(std::numeric_limits<float>::max_digits10);		///< Sets output precision for long doubles
	prev_loop_benchmark = start = init_benchmark();		///< Starts custom benchmark
	std::vector<std::array<float, Nv>> Vec;		///< Initializes the `Vec` variable
	Vec.reserve(N);		///< Reserves `N` blocks of memory to increase speed
	vec_init(Vec);		///< Fills `Vec` with random real numbers
	multidimentional_float_vector_interface(Vec, "Vec");		///< Outputs `Vec` contents
	export_multidimentional_float_vector(Vec, "Vec");		///< Exports `Vec` contents to CSV file
	std::vector<std::array<float, Nv>> old_Center;		///< Initializes `old_Center` variable, which holds centroids
	old_Center.reserve(Nc);		///< Reserves `Nc` blocks of memory to increase speed
	init_centers(Vec, old_Center);		///< Fills `old_Center` with random vectors from `Vec`
	multidimentional_float_vector_interface(old_Center, "old_Center");		///< Outputs `old_Center` contents
	export_multidimentional_float_vector(old_Center, "old_Center");		///< Exports `old_Center` contents to CSV file
	std::vector<std::array<float, Nv>> new_Center;		///< Initializes the `new_Center` variable, which holds optimized centroids
	new_Center.reserve(Nc);		///< Reserves `Nc` blocks of memory to increase speed
	long double iter_conv = std::numeric_limits<double>::infinity();		///< Initializes `iter_conv` variable, which holds the sum of distances betweens `Vec` and `new_Center`
	long double norm_iter_conv = std::numeric_limits<double>::infinity();		///< Initializes `norm_iter_conv` variable, which holds the normalized value of `iter_conv`
	std::array<std::vector<int>, Nc> Classes;		///< Initializes `Classes` variable, which holds all cluster pointers
	std::chrono::time_point<std::chrono::system_clock> optimization_benchmark = init_benchmark();		///< K-means optimization loop benchmark
	for (int iter_counter = 1; norm_iter_conv > THR_KMEANS; iter_counter += 1)		///< K-means optimazation loop
	{
		std::chrono::duration<double> bench_res =
			bench_loop(prev_loop_benchmark, loop_benchmark = init_benchmark());		///< Benchmarks loop
		kmeans_progress(iter_counter, iter_conv, norm_iter_conv, bench_res, VERBOSITY);		///< Outputs K-means progress
		track_kmeans_progress(iter_counter, norm_iter_conv);		///< Stores K-means progress
		compute_classes(Vec, old_Center, Classes);		///< Re-computes `Classes` contents
		multidimentional_int_array_interface(Classes, "Classes");		///< Outputs `Classes` contents
		export_multidimentional_integer_array(Classes, "Classes" + std::to_string(iter_counter));		///< Exports `Classes` contents to CSV file
		optimize_center(Vec, new_Center, Classes);		///< Computes error of current clusters and stores it to the variable `new_Center`
		multidimentional_float_vector_interface(new_Center, "new_Center");		///< Outputs `new_Center` contents
		export_multidimentional_float_vector(new_Center, "new_Center" + std::to_string(iter_counter));		///< Exports `new_Center` contents to CSV file
		long double prev_iter_conv = iter_conv;		///< Stores previous `iter_conv` to use it to normalize convergence
		iter_conv = convergence(new_Center, old_Center);		///< Computes convergence of current K-means loop
		norm_iter_conv = normalize_convergence(iter_conv, prev_iter_conv, iter_counter);		///< Normalizes `iter_conv` to avoid overfitting
		old_Center.swap(new_Center);		///< Swaps `old_Center` with `new_Center` to start the next loop
		multidimentional_float_vector_interface(old_Center, "old_Center");		///< Outputs `old_Center` contents
	}
	std::chrono::time_point<std::chrono::system_clock> end = terminate_bench();		///< Initializes the endpoint of K-means benchmark
	std::pair<std::chrono::duration<double>, std::chrono::duration<double>> bench_res =
		benchmark_results(start, optimization_benchmark, end);		///< Stores benchmark results
	kmeans_termination(bench_res, norm_iter_conv);		///< Outputs benchmark results
	export_kmeans_progress("kmeans_results");		///< Exports K-means progress into a CSV file
	return 0;
}
```