TECHNISCHE UNIVERSITÄT DRESDEN

KOMPLEXPRAKTIKUM PARALLELES RECHNEN

Timo Nicolai
Studiengang: Informationssystemtechnik
Matrikelnummer: 4048209

14. Juni 2018

Betreuer
Dipl.-Ing. Oliver Knodel

INHALTSVERZEICHNIS

1 EINLEITUNG

Dieser Beleg beschreibt die Implementierung von Lloyds k-means Vektorquantisierungs-Algorithmus sowohl in serieller als auch parallelisierter Form.

Die Parallelisierung wurde zum einen durch parallele Nutzung mehrerer Prozessor-Kerne mittels `OpenMP` Quellcode-Annotationen und zum anderen durch Auslagerung parallelisierbarer Performance-kritischer Abschnitte auf eine `NVIDIA`-GPU mittels `CUDA` erreicht. Beide Methoden werden im Folgenden mit Hinblick auf Besonderheiten in der Implementierung und Performance-Gewinnen in Abhängigkeit der Problemgröße und genutzten Hardware-Ressourcen mit der seriellen Implementierung gegenübergestellt.

2 DER K-MEANS ALGORITHMUS

2.1 BESCHREIBUNG DES ALGORITHMUS

Unter der Bezeichnung k-means sind mehrere Algorithmen bekannt, die alle zum Ziel haben, auf effizientem Wege eine annähernde Lösung für das folgende Problem zu finden:

Eine Menge von Vektoren ist derart in k Partitionen zu zerlegen, dass die Summe der quadratischen euklidische Distanzen aller Vektoren in jeder Partition zum jeweiligen Partitions-Schwerpunkt minimiert wird. D.h. sei M eine Menge von Vektoren $x \in \mathbb{R}^n$, dann sind Partitionen $P = \{P_i \mid i = 1, 2, \ldots, k\}$ von M mit Schwerpunkten $S = \{\mu_i \mid i = 1, 2, \ldots, k\}$ gesucht, sodass der folgende Ausdruck minimal wird:

$$\sum_{k} \sum_{x \in P_k} \|x - \mu_k\|^2$$

Der hier betrachtete Algorithmus ist wohl der bekannteste und einfachste unter diesen Algorithmen und wurde erstmals von Lloyd vorgeschlagen [1]. Er basiert auf der wiederholten Zuweisung von Vektoren zu den Partitionen mit den ihnen am nähsten gelegenen Schwerpunkten und folgender Neuberechnung dieser Schwerpunkte und ist hier als Algorithmus 1 dargestellt.

Algorithmus 1 Llyods k-means Algorithmus

```
1:  procedure K-MEANS(x)
2:      // Initialisiere Schwerpunkte mit zufälligen Vektoren
3:      for i = 1...k do
4:          P_i := ∅
5:          μ_i := ein zufälliges x_j ∈ M
6:      end for
7:      // Berechne (annähernd) ideale Partitionen iterativ
8:      while Änderungen an P do
9:          // Weise Vektoren den Partitionen mit nächstem Schwerpunkt zu
10:         for x_i ∈ M do
11:             j := argmin_j ||x_i − μ_j||
12:             P_j := P_j ∪ {x_i}
13:         end for
14:         // Aktualisiere Partitions-Schwerpunkte
15:         for i = 1...k do
16:             μ_i := |P_i|^−1 Σ_{x_j ∈ P_i} x_j
17:         end for
18:     end while
19: end procedure
```

Es ist zusätzlich zu beachten, dass es mehrere denkbare Bedingungen für den den Abbruch der Hauptschleife in den Zeilen 8 bis 18 geben kann. Die in Algorithmus 1 gezeigte triviale Abbruchbedingung greift dann, wenn in einer Iteration kein Vektor die Partition wechselt. In diesem Fall wurde eine stationäre Lösung gefunden und die Berechnung kann abgebrochen

werden. Auch möglich ist es, von vorneherein ein Limit für die Anzahl der Iterationen festzulegen bei dessen Überschreitung abgebrochen wird auch wenn keine stationäre Lösung gefunden wurde. Eine weitere Möglichkeit ist es, dann abzubrechen wenn sich in einer Iteration keiner der Partitions-Schwerpunkte um mehr als einen Schwellwert ϵ von seinem vorherigen Wert entfernt. In den folgenden Implementierungen wird stets mit einem Iterations-Limit von 100 gearbeitet (das praktischen Untersuchungen nach jedoch selten erreicht wird).

Außerdem ist in Algorithmus 1 die Behandlung leer verbliebener Partitionen nach der Schleife in den Zeilen 10 bis 13 nicht dargestellt. Dieser Fall sollte in einer "vernünftigen" Implementierung für die meisten Inputs eher die Ausnahme als die Regel darstellen, muss aber trotzdem berücksichtigt werden um die Korrektheit des Algorithmus zu garantieren. Der in den folgenden Implementierungen (mit Ausnahme der `CUDA C` Implementierung) gewählte Ansatz ist es, iterativ alle leeren Partitionen mit dem Vektor aus der jeweils momentan größten Partition, der von deren Schwerpunkt am weitesten entfernt ist, zu füllen.

Der Algorithmus wie er hier dargestellt ist, kann auf Mengen von Vektoren beliebiger Dimensionen angewandt werden. Im Folgenden wird allerdings nur noch die Anwendung des k-means Algorithmus zur Segementierung von Farbbildern betrachtet (und somit der Fall $x \in F \subset \mathbb{R}^3$ unter der Annahme, dass M die Menge aller Pixel eines Bildes in `RGB`-Vektordarstellung und F der `RGB`-Farbraum ist, d.h. $x = (r, g, b)^T$ mit $r, g, b \in [0, 255]$).

2.2 SEQUENTIELLE IMPLEMENTIERUNG

Listing 2.1 zeigt eine sequentielle Implementierung des k-means Algorithmus in C. Die Funktion `kmeans` in Zeile 42 zerlegt die `n_pixels` in `pixels` gespeicherten Pixel in `n_centroids` Partitionen mit den Schwerpunkten `centroids`. Die Funktion weist zu diesem Zweck jedem Pixel `pixels[i]` in `labels[i]` den Index der Partition zu der dieser Pixel gehört zu. `struct pixel` repräsentiert dabei einen Punkt im `RGB`-Farbraum und ist wie in Listing 2.2 gezeigt definiert.

Listing 2.1: Sequentielle k-means Implementierung in C (Ausschnitt `kmeans.c`)

```
1  #include <float.h>
2  #include <math.h>
3  #include <omp.h>
4  #ifdef PROFILE
5  #include <stdio.h>
6  #endif
7  #include <stdlib.h>
8  #include <time.h>
9
10 #include "kmeans_config.h"
11 #include "kmeans.h"
12
13 // compute euclidean distance between two pixel values
14 static inline double pixel_dist(struct pixel p1, struct pixel p2)
15 {
16     double dr = p1.r - p2.r;
17     double dg = p1.g - p2.g;
18     double db = p1.b - p2.b;
19
20     return sqrt(dr * dr + dg * dg + db * db);
21 }
22
23 // find index of centroid with least distance to some pixel
24 static inline size_t find_closest_centroid(
25     struct pixel pixel, struct pixel *centroids, size_t n_centroids)
26 {
27     size_t closest_centroid = 0u;
28     double min_dist = DBL_MAX;
29
30     for (size_t i = 0; i < n_centroids; ++i) {
31         double dist = pixel_dist(pixel, centroids[i]);
32
```

```
        if (dist < min_dist) {
            closest_centroid = i;
            min_dist = dist;
        }
    }

    return closest_centroid;
}

void kmeans_c(struct pixel *pixels, size_t n_pixels,
              struct pixel *centroids, size_t n_centroids,
              size_t *labels)
{
#ifdef PROFILE
    clock_t exec_begin;
    double exec_time_kernel1;
    double exec_time_kernel2;
    double exec_time_kernel3;
#endif

    // seed rand
    srand(time(NULL));

    // allocate auxiliary heap memory
    struct pixel *sums = malloc(n_centroids * sizeof(struct pixel));
    size_t *counts = malloc(n_centroids *  sizeof(size_t));

    // randomly initialize centroids
    for (size_t i = 0u; i < n_centroids; ++i) {
        centroids[i] = pixels[rand() % n_pixels];

        struct pixel tmp = { 0.0, 0.0, 0.0 };
        sums[i] = tmp;

        counts[i] = 0u;
    }

    // repeat for KMEANS_MAX_ITER or until solution is stationary
    for (int iter = 0; iter < KMEANS_MAX_ITER; ++iter) {
        int done = 1;

        // reassign points to closest centroids
#ifdef PROFILE
        exec_begin = clock();
#endif
        for (size_t i = 0u; i < n_pixels; ++i) {
            struct pixel pixel = pixels[i];

            // find centroid closest to pixel
            size_t closest_centroid =
                find_closest_centroid(pixel, centroids, n_centroids);

            // if pixel has changed cluster...
            if (closest_centroid != labels[i]) {
                labels[i] = closest_centroid;

                done = 0;
            }

            // update cluster sum
            struct pixel *sum = &sums[closest_centroid];
            sum->r += pixel.r;
            sum->g += pixel.g;
            sum->b += pixel.b;

            // update cluster size
            counts[closest_centroid]++;
        }
#ifdef PROFILE
        exec_time_kernel1 = (double) (clock() - exec_begin) / CLOCKS_PER_SEC;
#endif

        // repair empty clusters
```

```
106 #ifdef PROFILE
107         exec_begin = clock();
108 #endif
109         for (size_t i = 0u; i < n_centroids; ++i) {
110             if (counts[i])
111                 continue;
112 
113             done = 0;
114 
115             // determine largest cluster
116             size_t largest_cluster = 0u;
117             size_t largest_cluster_count = 0u;
118             for (size_t j = 0u; j < n_centroids; ++j) {
119                 if (j == i)
120                     continue;
121 
122                 if (counts[j] > largest_cluster_count) {
123                     largest_cluster = j;
124                     largest_cluster_count = counts[j];
125                 }
126             }
127 
128             // determine pixel in this cluster furthest from its centroid
129             struct pixel largest_cluster_centroid = centroids[largest_cluster];
130 
131             size_t furthest_pixel = 0u;
132             double max_dist = 0.0;
133             for (size_t j = 0u; j < n_pixels; ++j) {
134                 if (labels[j] != largest_cluster)
135                     continue;
136 
137                 double dist = pixel_dist(pixels[j], largest_cluster_centroid);
138 
139                 if (dist > max_dist) {
140                     furthest_pixel = j;
141                     max_dist = dist;
142                 }
143             }
144 
145             // move that pixel to the empty cluster
146             struct pixel replacement_pixel = pixels[furthest_pixel];
147             centroids[i] = replacement_pixel;
148             labels[furthest_pixel] = i;
149 
150             // correct cluster sums
151             sums[i] = replacement_pixel;
152 
153             struct pixel *sum = &sums[largest_cluster];
154             sum->r -= replacement_pixel.r;
155             sum->g -= replacement_pixel.g;
156             sum->b -= replacement_pixel.b;
157 
158             // correct cluster sizes
159             counts[i] = 1u;
160             counts[largest_cluster]--;
161         }
162 #ifdef PROFILE
163         exec_time_kernel2 = (double) (clock() - exec_begin) / CLOCKS_PER_SEC;
164 #endif
165 
166         // average accumulated cluster sums
167 #ifdef PROFILE
168         exec_begin = clock();
169 #endif
170         for (int j = 0; j < n_centroids; ++j) {
171             struct pixel *centroid = &centroids[j];
172             struct pixel *sum = &sums[j];
173             size_t count = counts[j];
174 
175             centroid->r = sum->r / count;
176             centroid->g = sum->g / count;
177             centroid->b = sum->b / count;
178 
```

```
179             sum->r = 0.0;
180             sum->g = 0.0;
181             sum->b = 0.0;
182
183             counts[j] = 0u;
184         }
185 #ifdef PROFILE
186         exec_time_kernel3 = (double) (clock() - exec_begin) / CLOCKS_PER_SEC;
187 #endif
188
189         // break if no pixel has changed cluster
190         if (done)
191             break;
192     }
193 #ifdef PROFILE
194     printf("Total kernel execution times:\n");
195     printf("Kernel 1 (reassigning points to closest centroids): %.3e\n",
196            exec_time_kernel1);
197     printf("Kernel 2 (repairing empty clusters): %.3e\n",
198            exec_time_kernel2);
199     printf("Kernel 3 (average accumulated centroids): %.3e\n",
200            exec_time_kernel3);
201 #endif
202
203     free(sums);
```

Listing 2.2: struct pixel (Ausschnitt `kmeans.h`)

```
4  struct pixel
5  {
6      double r, g, b;
7  };
```

Will man die Funktion nutzen um ein Farbbild zu segmentieren, muss man also zunächst die Pixel des Bildes in ein eindimensionales Array vom Typ `struct pixel[]` transformieren und der Funktion `kmeans` übergeben. Danach kann dann der jeweils i-te Pixel durch den in `centroids[labels[i]]` gespeicherten Schwerpunkt der Partitionen der er zugeteilt wurde ersetzt und anschließend das zweidimensionale Bild rekonstruiert werden.
In den Zeilen 61-68 findet die zufällige Initialisierung der Partitions-Schwerpunkte statt. Die Hauptschleife in den Zeilen 71-192 iteriert solange, bis keine Neuverteilung von Pixeln auf Partitionen mehr stattfindet oder `KMEANS_MAX_ITER` Iterationen erreicht worden sind. Dabei werden in den Zeilen 78-100 alle Pixel den Partitionen mit dem ihnen am nächsten liegenden Schwerpunkt zugewiesen. Gleichzeitig werden die Summen aller Pixel in jeder Partition im Array `sums` und die Größen der Partitionen im Array `counts` festgehalten. In den Zeilen 109-161 wird die (verhältnismäßig aufwändige aber in den meisten Anwendungsfällen selten auftretende) Umverteilung von Pixeln auf eventuell leer gebliebene Partitionen vorgenommen. In Zeilen 170-184 werden die neuen Schwerpunkte über die in `sums` und `counts` gepeicherten Werte berechnet.

2.2.1 PROFILING

Vor der Parallelisierung der Implementierung ist es sinnvoll zunächst zu bestimmen welche Codeabschnitte am meisten zur Gesamtlaufzeit des Algorithmus beitragen. Diese sollten dann entsprechend bei der Parallelisierung priorisiert werden (sofern sie sinnvoll parallelisierbar sind). Wird das gezeigte Programm mit gesetztem `PROFILE` Symbol kompiliert, so wird für die drei wichtigsten Programmabschnitte (zuweisen von Pixeln zu Partitionen, Reparatur leerer Partitionen und Neuberechnung der Partitions-Schwerpunkte) jeweils die CPU-Laufzeit (hier aufgrund fehlender Parallelisierung quasi identisch zur "wall-clock-time") aufgezeichnet und auf der Konsole ausgegeben (z.B. mittels `make profile`). Auf ein Beispielbild (`images/profile_image.jpg`, `512×512 Pixel`) angewandt ergeben sich so beispielsweise folgende Werte:

```
Kernel 1 (reassigning points to closest centroids): 4.809e-03
Kernel 2 (repairing empty clusters): 1.000e-06
Kernel 3 (average accumulated centroids): 1.000e-06
```

Es ist hier nicht überraschend, dass die Neuzuweisung von Pixeln an die Partitionen mit den ihnen am nächsten liegenden Schwerpunkten um einige Größenordnungen mehr Zeit beansprucht als die beiden anderen Abschnitte. Die Reparatur findet für typische Inputs sehr selten bzw. gar nicht statt und die Neuberechnung der Schwerpunkte ist mit wenig Rechenaufwand verbunden. Diese beiden Abschnitte sind für die erfolgreiche Parallelisierung demnach quasi irrelevant.

3 PARALLELISIERUNG

3.1 OPENMP

3.1.1 IMPLEMENTIERUNG

Listing 3.1 zeigt die mittels `OpenMP` parallelisierte Variante der in Listing 2.1 gezeigten C-Implementierung. Modifizierte Zeilen sind farbig hervorgehoben. Es mussten nur einige wenige Änderungen vorgenommen werden um eine performante Parallelisierung zu erreichen. Hier zeigt sich eine große Stärke von OpenMP: existierender Code kann teilweise (fast) ohne Änderungen am Code selbst nur durch die Nutzung von `#pragma` Direktiven parallelisiert werden. Natürlich muss für maximale Ausnutzung der dem Algorithmus inhärenten Parallelisierbarkeit eine günstige Code-Struktur vorliegen. Dies ist hier aber prinzipiell schon gegeben, da die Performance-kritische Schleife in Zeilen 78-100 in Listing 2.1 bereits in geigneter Form vorliegt. Hier können die Berechnungen für jedes Pixel unabhängig voneinander erfolgen, es muss lediglich der Zugriff auf die Variable `done`, `sums` und `counts` synchronisiert werden. Für erstere genügt eine `#pragma atomic write` Direktive, für die beiden Arrays kann die `reduction` Klausel [1] genutzt werden. Hierfür muss dann allerdings `sums` als Array von Fließkommazahlen realisiert werden. (jeweils drei aufeinander folgende ersetzen ein `struct pixel`) da Reduktionen nicht auf nutzerdefinierte Typen angewandt werden können.
Andere Abschnitte des Algorithmus können potentiell ebenfall parallelisiert werden, allerdings mit vergleichbar kleinem Performance-Gewinn. Bei der Reparatur leerer Partitionen wurde ebenfalls über die einzelnen Pixel parallelisiert, dieser Codeabschnitt wird jedoch nur selten (wenn überhaupt) aufgerufen. Eine Parallelisierung bei Neuberechnung der Schwerpunkte hat für realistische Input-Größen keinen starken Einfluss auf die Ausführungszeit des Programms ist hier aber ebenfalls einfach möglich.

Listing 3.1: Parallelisierte k-means Implementierung mit `OpenMP` (Ausschnitt `kmeans.c`)

```
207 void kmeans_omp(struct pixel *pixels, size_t n_pixels,
208                 struct pixel *centroids, size_t n_centroids,
209                 size_t *labels)
210 {
211     // seed rand
212     srand(time(NULL));
213
214     // allocate auxiliary heap memory
215     double *sums = malloc(3 * n_centroids * sizeof(double));
216     size_t *counts = malloc(n_centroids * sizeof(size_t));
217
218     // randomly initialize centroids
219     for (size_t i = 0u; i < n_centroids; ++i) {
220         centroids[i] = pixels[rand() % n_pixels];
221
222         double *sum = &sums[3 * i];
223         sum[0] = sum[1] = sum[2] = 0.0;
224
225         counts[i] = 0u;
226     }
227
228     // repeat for KMEANS_MAX_ITER or until solution is stationary
229     for (int iter = 0; iter < KMEANS_MAX_ITER; ++iter) {
```

[1] Seit OpenMP Version 4.5 (z.B. von GCC ab Version 6.1 unterstützt) auf Arrays anwendbar.

```
        int done = 1;

        // reassign points to closest centroids
        #pragma omp parallel for \
            reduction(+ : sums[:(3 * n_centroids)], counts[:n_centroids])
        for (int i = 0; i < n_pixels; ++i) {
            struct pixel pixel = pixels[i];

            // find centroid closest to pixel
            int closest_centroid =
                find_closest_centroid(pixel, centroids, n_centroids);

            // if pixel has changed cluster...
            if (closest_centroid != labels[i]) {
                labels[i] = closest_centroid;

                #pragma omp atomic write
                done = 0;
            }

            // update cluster sum
            double *sum = &sums[3 * closest_centroid];
            sum[0] += pixel.r;
            sum[1] += pixel.g;
            sum[2] += pixel.b;

            // update cluster size
            counts[closest_centroid]++;
        }

        // repair empty clusters
        for (size_t i = 0u; i < n_centroids; ++i) {
            if (counts[i])
                continue;

            done = 0;

            // determine largest cluster
            size_t largest_cluster = 0u;
            size_t largest_cluster_count = 0u;
            for (size_t j = 0u; j < n_centroids; ++j) {
                if (j == i)
                    continue;

                if (counts[j] > largest_cluster_count) {
                    largest_cluster = j;
                    largest_cluster_count = counts[j];
                }
            }

            // determine pixel in this cluster furthest from its centroid
            struct pixel largest_cluster_centroid = centroids[largest_cluster];

            size_t furthest_pixel = 0u;
            double max_dist = 0.0;

            #pragma omp parallel for
            for (size_t j = 0u; j < n_pixels; ++j) {
                if (labels[j] != largest_cluster)
                    continue;

                double dist = pixel_dist(pixels[j], largest_cluster_centroid);

                #pragma omp critical
                {
                    if (dist > max_dist) {
                        furthest_pixel = j;
                        max_dist = dist;
                    }
                }
            }

            // move that pixel to the empty cluster
```

```
303             struct pixel replacement_pixel = pixels[furthest_pixel];
304             centroids[i] = replacement_pixel;
305             labels[furthest_pixel] = i;
306
307             // correct cluster sums
308             double *sum = &sums[3 * i];
309             sum[0] = replacement_pixel.r;
310             sum[1] = replacement_pixel.g;
311             sum[2] = replacement_pixel.b;
312
313             sum = &sums[3 * largest_cluster];
314             sum[0] -= replacement_pixel.r;
315             sum[1] -= replacement_pixel.g;
316             sum[2] -= replacement_pixel.b;
317
318             // correct cluster sizes
319             counts[i] = 1u;
320             counts[largest_cluster]--;
321         }
322
323         // average accumulated cluster sums
324         #pragma omp parallel for
325         for (int j = 0; j < n_centroids; ++j) {
326             struct pixel *centroid = &centroids[j];
327             double *sum = &sums[3 * j];
328             size_t count = counts[j];
329
330             centroid->r = sum[0] / count;
331             centroid->g = sum[1] / count;
332             centroid->b = sum[2] / count;
333
334             sum[0] = sum[1] = sum[2] = 0.0;
335             counts[j] = 0u;
336         }
337
338         // break if no pixel has changed cluster
339         if (done)
340             break;
341     }
342
343     free(sums);
344     free(counts);
345 }
```

3.1.2 PERFORMANCE

Abbildung 3.1 und Abbildung 3.2 visualisieren die Ausführungszeiten des seriellen sowie des mit `OpenMP` parallelisierten k-means Algorithmus (hier bei Nutzung aller Prozessor-Kerne).
Für alle auf der horizontalen Achse abgetragenen Bild-Dimensionen wurden beide Implementierungen auf je 100 verschiedene, zufällig generierte Farbbilder angewandt. Die Programme wurden auf einer Maschine mit einem `Intel Core i5D-4690` Prozessor mit vier Prozessor-Kernen getestet. Die Boxplots lassen erkennen, in welchem Bereich sich die Ausführungszeiten jeweils befinden und wie groß ihre Streuung ist. Die gestrichtelte Linie verläuft durch die Mediane der Ausführungszeiten.
Abbildung 3.3 zeigt für dieselben Bild-Dimensionen jeweils den Durchschnitt des durch die Nutzung von `OpenMP` erreichten Speedup bei Variation der genutzten Prozessor-Kerne[2].
Es lässt sich erkennen, dass für angemessene Problemgrößen ein Speedup, der ungefähr im Bereich der Anzahl der zur Verfügung stehenden Prozessorkerne liegt, erreicht werden konnte. Dabei fällt der Geschwindigkeitsgewinn mit steigender Prozessoranzahl jedoch immer kleiner aus. Gleichzeitig ist bei Nutzung von mehreren Kernen die Schwankung der Ausführungszeiten größer.

[2] Jeweils mittels `omp_set_num_threads()` konfiguriert.

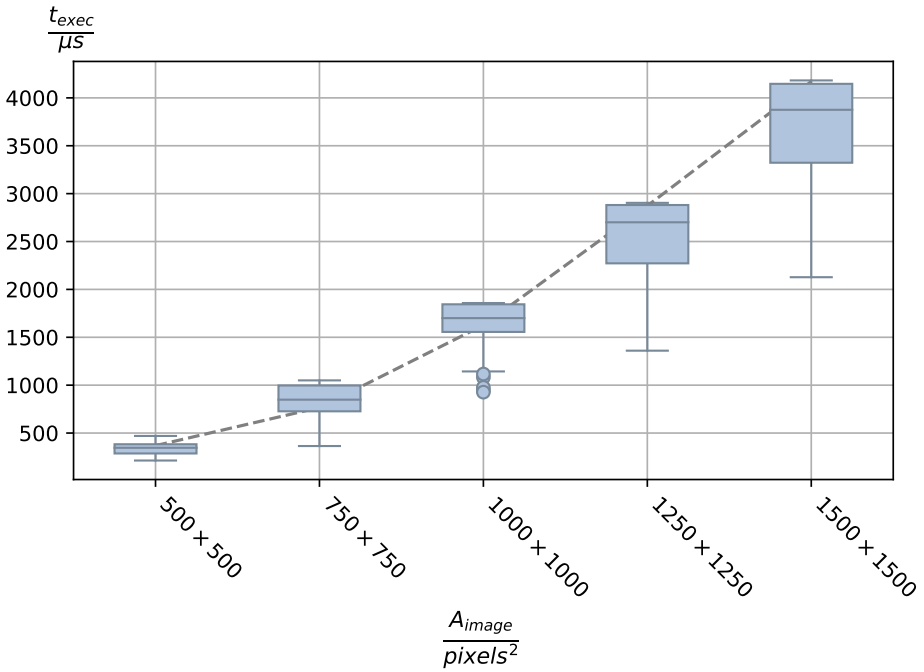


Abbildung 3.1: Serielle C Implementierung - Ausführungszeiten
(je 100 Datenpunkte pro Bild-Dimension)

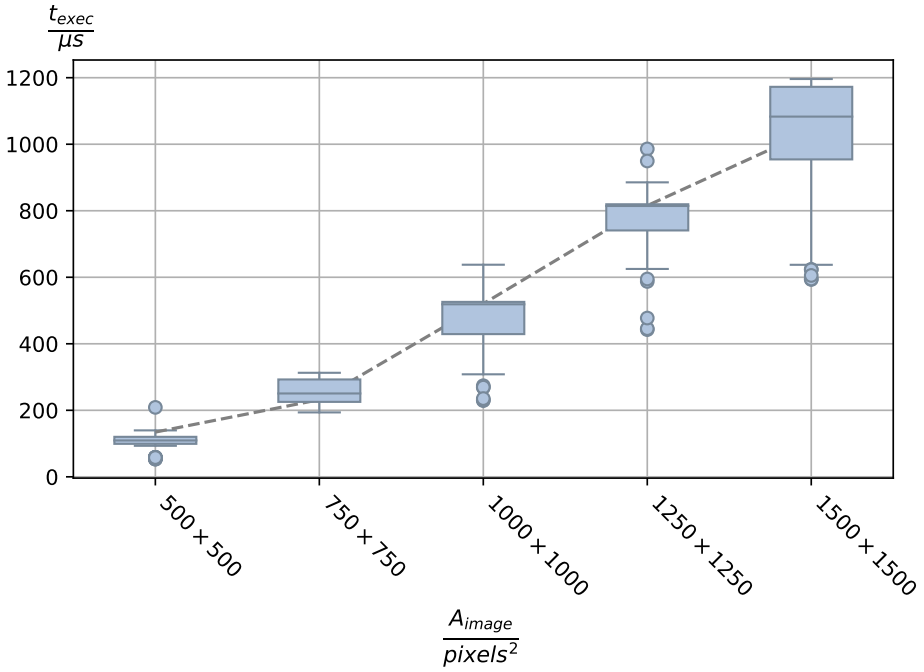


Abbildung 3.2: Parallelisierte OpenMP Implementierung - Ausführungszeiten
(vier Prozessorkerne, je 100 Datenpunkte pro Bild-Dimension)

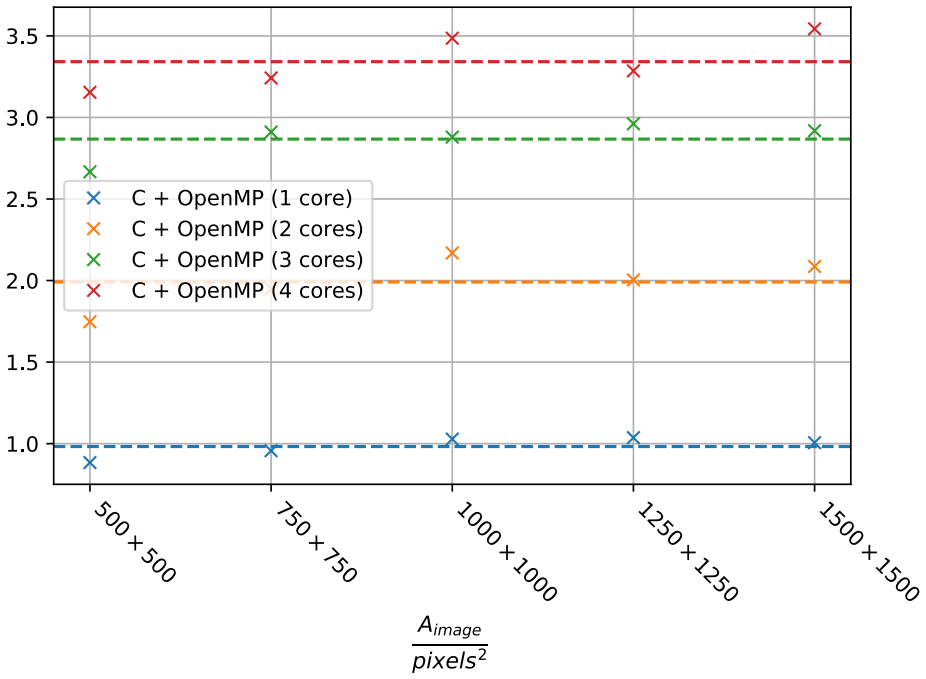


Abbildung 3.3: Speedup C vs. C + OpenMP

3.2 CUDA

3.2.1 IMPLEMENTIERUNG

Listing 3.2 zeigt die mittels `CUDA` parallelisierte Version des Programms. Im Gegensatz zur `OpenMP` Variante unterscheidet diese sich deutlich vom reinen C-Code. Die Performance-kritische Zuweisung von Pixeln zu Partitionen sowie die Neuberechnung der Schwerpunkte sind mittels auf der Grafikkarte (nachfolgend `Device`) ausgeführten `Kerneln` realisiert (letzterer dient hauptsächlich dazu, unnötiges Übertragen von Zwischenergebnissen von Device zu Host zu vermeiden).

Listing 3.2: Parallelisierte k-means Implementierung mit `CUDA` (kmean.cu)

```
1  #include <assert.h>
2  #include <float.h>
3  #include <math.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  extern "C" {
9  #include "kmeans.h"
10 }
11 #include "kmeans_config.h"
12
13 /* Helper Functions ************************************************************/
14
15 #define cudaAssert(code, file, line) do {                 \
16   if (code != cudaSuccess) {                              \
17     fprintf(stderr, "A CUDA error occurred: %s (%s:%d)\n", \
18             cudaGetErrorString(code), file, line);        \
19     exit(code);                                           \
20   }                                                       \
21 } while(0)
22
23 #define cudaCheck(code) do { cudaAssert(code, __FILE__, __LINE__); } while(0)
```

```
24
25  /* CUDA kernels ***************************************************************/
26
27  // reassign points to closest centroids (#threads must be a power of two)
28  __global__
29  static void reassign(struct pixel *pixels, size_t n_pixels,
30                       struct pixel *centroids, size_t n_centroids,
31                       size_t *labels, struct pixel *sums, size_t *counts,
32                       int *empty, int *done)
33  {
34      // index alias
35      size_t tid = threadIdx.x;
36      size_t bid = blockIdx.x;
37
38      // set up shared and global memory
39      extern __shared__ char shared[];
40
41      struct pixel *shared_pixels = (struct pixel *) shared;
42
43      size_t shared_counts_offs = blockDim.x * sizeof(struct pixel);
44      shared_counts_offs += sizeof(size_t) - sizeof(struct pixel) % sizeof(size_t);
45
46      size_t *shared_counts = (size_t *) &shared[shared_counts_offs];
47
48      if (tid < n_centroids) {
49        shared_pixels[tid] = centroids[tid];
50
51        struct pixel tmp = { 0.0, 0.0, 0.0 };
52        sums[n_centroids * bid + tid] = tmp;
53        counts[n_centroids * bid + tid] = 0u;
54      }
55
56      __syncthreads();
57
58      // begin reassignment
59      size_t index = bid * blockDim.x + tid;
60      size_t closest_centroid = 0u;
61
62      if (index >= n_pixels) {
63         closest_centroid = n_centroids + 1u;
64      } else {
65          // find closest centroid
66          double min_dist = DBL_MAX;
67
68          struct pixel *p = &pixels[index];
69          for (size_t j = 0u; j < n_centroids; ++j) {
70              struct pixel *c = &shared_pixels[j];
71
72              double dr = p->r - c->r;
73              double dg = p->g - c->g;
74              double db = p->b - c->b;
75
76              double dist = sqrt(dr * dr + dg * dg + db * db);
77
78              if (dist < min_dist) {
79                  closest_centroid = j;
80                  min_dist = dist;
81              }
82          }
83
84          // if pixel has changed cluster...
85          if (closest_centroid != labels[index]) {
86              labels[index] = closest_centroid;
87
88              *done = 0;
89          }
90      }
91
92      // perform cluster wise tree-reduction to obtain cluster sums / counts
93      for (size_t j = 0u; j < n_centroids; ++j) {
94          if (j == closest_centroid) {
95              shared_pixels[tid] = pixels[index];
96              shared_counts[tid] = 1u;
```

```
            empty[j] = 0u;
        } else {
            struct pixel tmp = { 0.0, 0.0, 0.0 };
            shared_pixels[tid] = tmp;
            shared_counts[tid] = 0u;
        }

        __syncthreads();

        for (size_t dist = blockDim.x >> 1; dist > 0u; dist >>= 1) {
            if (tid < dist) {
                struct pixel *shared_sum1 = &shared_pixels[tid];
                struct pixel *shared_sum2 = &shared_pixels[tid + dist];

                shared_sum1->r += shared_sum2->r;
                shared_sum1->g += shared_sum2->g;
                shared_sum1->b += shared_sum2->b;

                shared_counts[tid] += shared_counts[tid + dist];
            }

            __syncthreads();
        }

        if (tid == 0) {
            struct pixel *sum = &sums[n_centroids * bid + j];
            struct pixel *shared_sum = &shared_pixels[0];

            sum->r += shared_sum->r;
            sum->g += shared_sum->g;
            sum->b += shared_sum->b;

            counts[n_centroids * bid + j] += shared_counts[0];
        }
    }
}

// reduce per-block cluster sums and counts
__device__
static void _reduce(size_t n_blocks, size_t n_centroids,
                    struct pixel *sums, size_t *counts)
{
    size_t index = blockIdx.x * blockDim.x + threadIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    // reduce per-block clusters sums / counts
    for (size_t dist = (n_blocks * n_centroids) >> 1;
         dist >= n_centroids; dist >>= 1) {

        for (size_t i = index; i < dist; i += stride) {
            struct pixel *sum1 = &sums[i];
            struct pixel *sum2 = &sums[i + dist];

            sum1->r += sum2->r;
            sum1->g += sum2->g;
            sum1->b += sum2->b;

            counts[index] += counts[i + dist];
        }

        __syncthreads();
    }
}

__global__
static void reduce(size_t n_blocks, size_t n_centroids,
                   struct pixel *sums, size_t *counts)
{
    _reduce(n_blocks, n_centroids, sums, counts);
}

// re-calculate centroids
__global__
```

```
static void average(size_t n_blocks, struct pixel *centroids, size_t n_centroids,
                    struct pixel *sums, size_t *counts, int reduce)
{
    size_t index = blockIdx.x * blockDim.x + threadIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    if (reduce)
        _reduce(n_blocks, n_centroids, sums, counts);

    // compute new centroids
    for (size_t i = index; i < n_centroids; i += stride) {
        struct pixel *c = &centroids[i];
        struct pixel *sum = &sums[i];
        size_t count = counts[i];

        c->r = sum->r / count;
        c->g = sum->g / count;
        c->b = sum->b / count;
    }
}

/* Main Function ***************************************************************/

extern "C" void kmeans_cuda(struct pixel *pixels, size_t n_pixels,
                            struct pixel *centroids, size_t n_centroids,
                            size_t *labels)
{
    // number of blocks to be used on device
    size_t n_blocks_reassign =
        (n_pixels + KMEANS_CUDA_BLOCKSIZE - 1u) / KMEANS_CUDA_BLOCKSIZE;

    // round upwards to power of two
    size_t n_blocks_reassign_log2 = 0u;
    while(n_blocks_reassign >>= 1)
        ++n_blocks_reassign_log2;

    n_blocks_reassign = 1 << (n_blocks_reassign_log2 + 1u);

    size_t n_block_reduce =
        (((n_centroids * n_blocks_reassign) >> 1) + KMEANS_CUDA_BLOCKSIZE - 1u) /
        KMEANS_CUDA_BLOCKSIZE;

    // reassignment step shared memory size
    size_t shm_slots;
    if (n_centroids > KMEANS_CUDA_BLOCKSIZE)
        shm_slots = n_centroids;
    else
        shm_slots = KMEANS_CUDA_BLOCKSIZE;

    size_t shm_reassign = shm_slots * (sizeof(struct pixel) + sizeof(size_t));
    shm_reassign += sizeof(size_t) - sizeof(struct pixel) % sizeof(size_t);

    // initialize centroids with random pixels
    srand(time(NULL));

    for (size_t i = 0u; i < n_centroids; ++i)
        centroids[i] = pixels[rand() % n_pixels];

    // initialize device memory
    size_t pixels_sz = n_pixels * sizeof(struct pixel);
    size_t centroids_sz = n_centroids * sizeof(struct pixel);
    size_t labels_sz = n_pixels * sizeof(size_t);

    struct pixel *pixels_dev;
    struct pixel *centroids_dev;
    size_t *labels_dev;

    cudaCheck(cudaMalloc(&pixels_dev, pixels_sz));
    cudaCheck(cudaMalloc(&centroids_dev, centroids_sz));
    cudaCheck(cudaMalloc(&labels_dev, labels_sz));

    cudaCheck(cudaMemcpy(pixels_dev, pixels, pixels_sz,
                         cudaMemcpyHostToDevice));
```

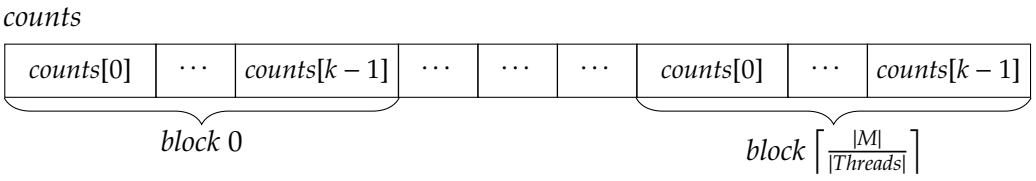
```
243
244     cudaCheck(cudaMemcpy(centroids_dev, centroids, centroids_sz,
245                          cudaMemcpyHostToDevice));
246
247     cudaCheck(cudaMemcpy(labels_dev, labels, labels_sz,
248                          cudaMemcpyHostToDevice));
249
250     // allocate and initialize auxiliary memory
251     size_t sums_sz = n_centroids * sizeof(struct pixel);
252     size_t counts_sz = n_centroids * sizeof(size_t);
253     size_t empty_sz = n_centroids * sizeof(int);
254     size_t sums_dev_sz = n_blocks_reassign * n_centroids * sizeof(struct pixel);
255     size_t counts_dev_sz = n_blocks_reassign * n_centroids * sizeof(size_t);
256
257     struct pixel *sums, *sums_dev;
258     size_t *counts, *counts_dev;
259     int *empty, *empty_dev;
260     int done, *done_dev;
261
262     sums = (struct pixel *) malloc(sums_sz);
263     counts = (size_t *) malloc(counts_sz);
264     empty = (int *) malloc(empty_sz);
265
266     cudaCheck(cudaMalloc(&sums_dev, sums_dev_sz));
267     cudaCheck(cudaMalloc(&counts_dev, counts_dev_sz));
268     cudaCheck(cudaMalloc(&empty_dev, empty_sz));
269     cudaCheck(cudaMalloc(&done_dev, sizeof(int)));
270
271     for (size_t i = 0u; i < n_centroids; ++i) {
272         struct pixel tmp = { 0.0, 0.0, 0.0 };
273         sums[i] = tmp;
274         counts[i] = 0u;
275     }
276
277     // repeat for KMEANS_MAX_ITER or until solution is stationary
278     for (int iter = 0; iter < KMEANS_MAX_ITER; ++iter) {
279         for (size_t i = 0u; i < n_centroids; ++i)
280             empty[i] = 1;
281
282         done = 1;
283
284         cudaCheck(cudaMemcpy(empty_dev, empty, empty_sz,
285                              cudaMemcpyHostToDevice));
286
287         cudaCheck(cudaMemcpy(done_dev, &done, sizeof(int),
288                              cudaMemcpyHostToDevice));
289
290         // reassign points to closest centroids
291         reassign<<<n_blocks_reassign, KMEANS_CUDA_BLOCKSIZE, shm_reassign>>>(
292             pixels_dev, n_pixels, centroids_dev, n_centroids, labels_dev,
293             sums_dev, counts_dev, empty_dev, done_dev
294         );
295
296         cudaCheck(cudaPeekAtLastError());
297
298         cudaCheck(cudaMemcpy(empty, empty_dev, empty_sz,
299                              cudaMemcpyDeviceToHost));
300
301         cudaCheck(cudaMemcpy(&done, done_dev, sizeof(int),
302                              cudaMemcpyDeviceToHost));
303
304         // check whether empty clusters need to be repaired
305         int repair = 0;
306         for (size_t i = 0u; i < n_centroids; ++i) {
307             if (!empty[i])
308                 continue;
309
310             // reduce in separate kernel
311             reduce<<<n_block_reduce, KMEANS_CUDA_BLOCKSIZE>>>(
312                 n_blocks_reassign, n_centroids, sums_dev, counts_dev
313             );
314
315             cudaCheck(cudaMemcpy(sums, sums_dev, sums_sz,
```

```
                                cudaMemcpyDeviceToHost));

            cudaCheck(cudaMemcpy(counts, counts_dev, counts_sz,
                                cudaMemcpyDeviceToHost));

            done = 0;
            repair = 1;
            break;
        }

        // repair empty clusters (on host)
        if (repair) {
            for (size_t i = 0u; i < n_centroids; ++i) {
                if (!empty[i])
                    continue;

                done = 0;

                // determine largest cluster
                size_t largest_cluster = 0u;
                size_t largest_cluster_count = 0u;
                for (size_t j = 0u; j < n_centroids; ++j) {
                    if (j == i)
                        continue;

                    if (counts[j] > largest_cluster_count) {
                        largest_cluster = j;
                        largest_cluster_count = counts[j];
                    }
                }

                // determine pixel in this cluster furthest from its centroid
                struct pixel *largest_cluster_centroid =
                    &centroids[largest_cluster];

                size_t furthest_pixel = 0u;
                double max_dist = 0.0;
                for (size_t j = 0u; j < n_pixels; ++j) {
                    if (labels[j] != largest_cluster)
                        continue;

                    struct pixel *p = &pixels[j];

                    double dr = p->r - largest_cluster_centroid->r;
                    double dg = p->g - largest_cluster_centroid->g;
                    double db = p->b - largest_cluster_centroid->b;

                    double dist = sqrt(dr * dr + dg * dg + db * db);

                    if (dist > max_dist) {
                        furthest_pixel = j;
                        max_dist = dist;
                    }
                }

                // move that pixel to the empty cluster
                struct pixel replacement_pixel = pixels[furthest_pixel];
                centroids[i] = replacement_pixel;
                labels[furthest_pixel] = i;

                // correct cluster sums
                sums[i] = replacement_pixel;

                struct pixel *sum = &sums[largest_cluster];
                sum->r -= replacement_pixel.r;
                sum->g -= replacement_pixel.g;
                sum->b -= replacement_pixel.b;

                // correct cluster sizes
                counts[i] = 1u;
                counts[largest_cluster]--;
            }

```

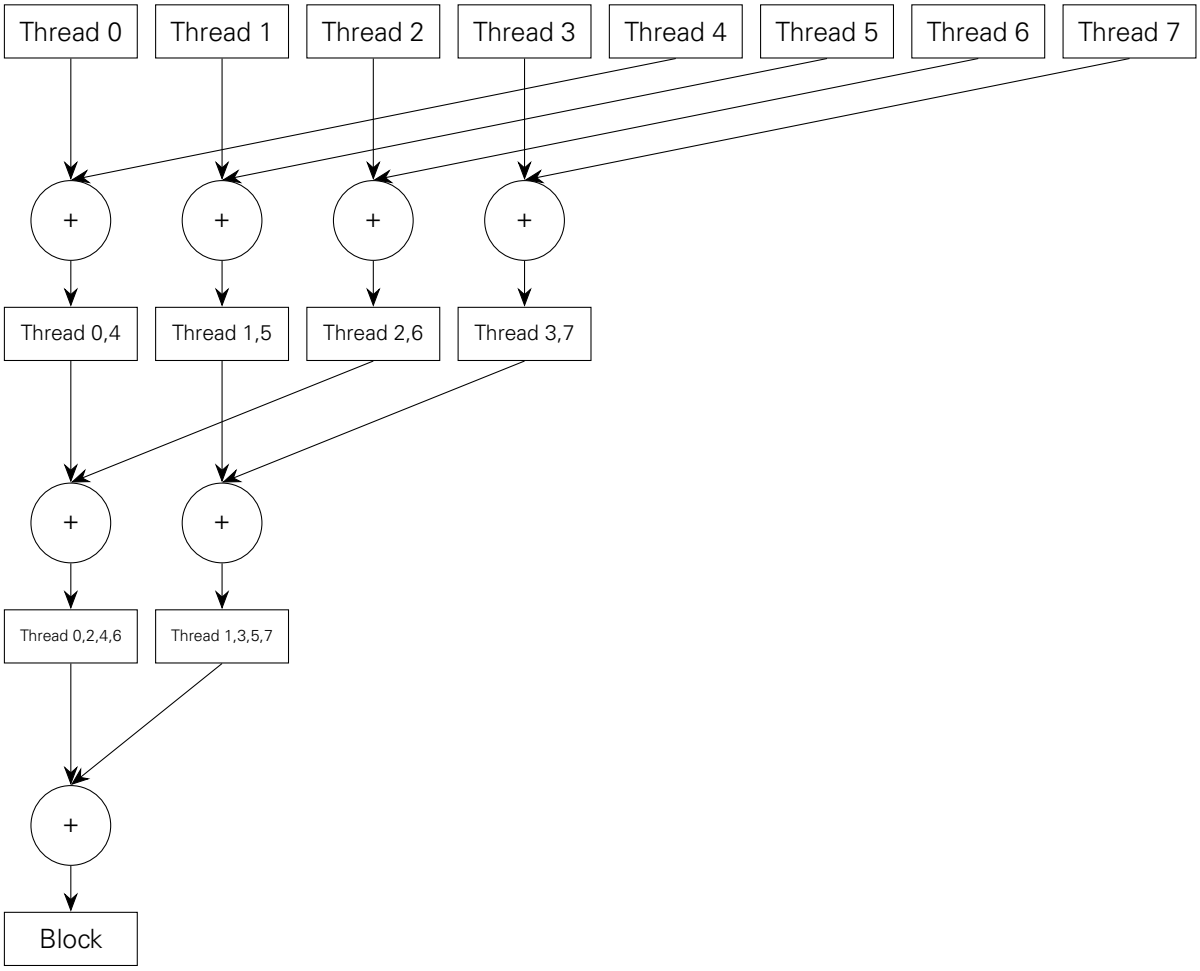
```
389             cudaCheck(cudaMemcpy(sums_dev, sums, sums_sz,
390                                  cudaMemcpyHostToDevice));
391
392             cudaCheck(cudaMemcpy(counts_dev, counts, counts_sz,
393                                  cudaMemcpyHostToDevice));
394         }
395
396         // re-calculate centroids
397         average<<<n_block_reduce, KMEANS_CUDA_BLOCKSIZE>>>(
398             n_blocks_reassign, centroids_dev, n_centroids,
399             sums_dev, counts_dev, !repair
400         );
401
402         cudaCheck(cudaPeekAtLastError());
403         cudaCheck(cudaDeviceSynchronize());
404
405         // break if no pixel has changed cluster
406         if (done)
407             break;
408     }
409
410     // copy device memory back to host
411     cudaCheck(cudaMemcpy(pixels, pixels_dev, pixels_sz,
412                          cudaMemcpyDeviceToHost));
413
414     cudaCheck(cudaMemcpy(centroids, centroids_dev, centroids_sz,
415                          cudaMemcpyDeviceToHost));
416
417     cudaCheck(cudaMemcpy(labels, labels_dev, labels_sz,
418                          cudaMemcpyDeviceToHost));
419
420     // free host and device memory
421     free(sums);
422     free(counts);
423
424     cudaCheck(cudaFree(pixels_dev));
425     cudaCheck(cudaFree(centroids_dev));
426     cudaCheck(cudaFree(labels_dev));
427     cudaCheck(cudaFree(sums_dev));
428     cudaCheck(cudaFree(counts_dev));
429 }
```

Die Parallelisierung ist hier von der Struktur her sehr ähnlich zu der bereits mit `OpenMP` realisierten. Problematisch ist hier nur, dass `CUDA` keine zu `OpenMP` äquivalente Syntax zur automatischen Reduktion von Arrays bietet. Diese musste dementsprechend in den `reassign` und `average` Kerneln explizit implementiert werden.
Der `reassign` Kernel lädt zunächst die Schwerpunkte in ein `shared memory` Segment, um den wiederholten lesenden Zugriff aller Threads in diesem Block auf die Schwerpunkte zu beschleunigen. In den Zeilen 59-90 wird anschließend äquivalent zur `OpenMP`-Implementierung pro Thread der zu einem Pixel nächste Schwerpunkt ermittelt. In Zeile 93 beginnt dann die Reduktion welche die `sums` und `counts` Arrays aktualisiert und die Nutzung kritischer Abschnitte umgeht. Dabei wird der gleiche Reduktionsschritt sequentiell auf die einzelnen Partition angewandt. Anschließend haben die Bereiche globalen Device-Speichers auf die `sums` bzw. `counts` (im Kernel, nicht zu verwechseln mit den gleichnamigen Host-Zeigern) zeigen folgenden Inhalt[3]:

[3] Die Gauss-Klammern symbolisieren hier ein Aufrunden auf die nächsthöhere Zweierpotenz



D.h. es existiert dann in diesen Speicherbereichen pro Block für den der Kernel ausgeführt wurde aneinandergereiht Arrays, deren Dimensionen denen der Host-Speicherbereiche auf die die Host-Zeiger `sums` und `counts` zeigen entsprechen und die jeweils den Beitrag aller in dem jeweiligen Block betrachteten Pixel zu diesen Arrays darstellen.
Die Reduktion erfolgt dabei für beide Arrays in $log_2(|Threads|)$ Schritten. Zunächst werden in den Zeilen 93-101 für jeden Thread genau dann, wenn der für den zum Thread gehörigen Pixel bestimmte nächstgelegene Schwerpunkt in der aktuell betrachteten Partition liegt, dessen Wert und eine eins (für die Bestimmung der Partitionsgröße) im `shared memory` Segment abgelegt. Anschließend werden diese Werte nach folgendem Schema schrittweise addiert um die entsprechenden Werte für den gesamten Block zu ermitteln (hier beispielsweise für acht Threads pro Block):



Im `average` Kernel wird die Reduktion schließlich abgeschlossen, indem diese Arrays nach dem gleichen Schema in der `_reduce` Device-Funktion aufsummiert werden. Anschließend sind die Partitions-Summen und -Größen ermittelt mit deren Hilfe dann in den Zeilen 180-188 die neuen Partitions-Schwerpunkte berechnet werden.
Die Reparatur leerer Partitionen ist in den Zeilen 305-394 komplett mittels Host-Code implemen-

tiert. Der ansonsten im `average` Kernel stattfindende letzte Teil der Reduktion wird in diesem Fall in einem separaten Kernel durchgeführt. Anschließend erfolgt der zur seriellen Implementierung äquivalente Reparatur-Schritt (hierbei müssen auch die Speicherinhalte der `sums` und `counts` Arrays zu Beginn und Ende zwischen Host und Device synchronisiert werden). Da dieser Abschnitt Performance-unkritisch ist, ist die genaue Implementierung hier nicht entscheidend und der Entwicklungsaufwand der hier für eine Parallelisierung notwendig wäre nicht gerechtfertigt. Insbesondere sollte auch der Overhead durch den Launch des zusätzlichen Kernels nicht ins Gewicht fallen, da dieser nur stattfindet, wenn tatsächlich leere Partitionen vorliegen.

3.2.2 PERFORMANCE

Der gewählte Ansatz umgeht zwar den Einsatz atomarer Operation und kritischer Abschnitte, die seriell abgearbeitet werden müssen und damit zu starken Performance-Einbußen führen, resultiert aber trotzdem durch die vielen benötigten Thread-Synchronisations-Punkte nicht in einer optimalen GPU Auslastung. Mit `NVIDIA Nsight` lässt sich ermitteln, dass rund die Hälfte aller auftretenden Instruction Stalls durch Thread-Synchronisation bedingt sind.
Außerdem limitiert der relativ hohe Shared Memory Bedarf zu Beginn jedes Reduktionsschrittes die Anzahl parallel ausführbarer Warps pro Streaming Multiprocessor, wie in Abbildung 3.5 (ebenfalls mit Nsight ermittelt) zu erkennen ist.

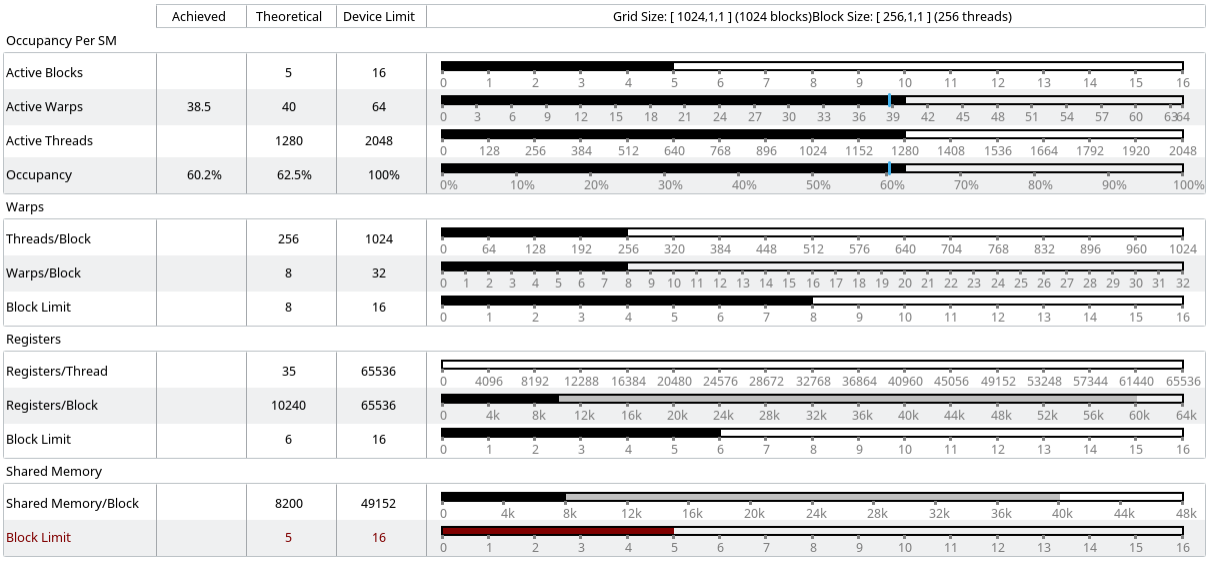


Abbildung 3.4: Shared Memory Ausnutzung

Die Analyse mit `Nsight` zeigt aber auch (Abbildung 3.5), dass die Auslastung der 13 zur Verfügung stehenden Streaming Multiprocessors nahezu ideal ist.
Trotz suboptimaler Ausnutzung der Device-Hardware wurden akzeptable Ergebnisse bezüglich der Ausführungszeit erzielt. Dies ist in Abbildung 3.6 gezeigt, die für eine Reihe von Problemgrößen den Median der Ausführungszeit (je 100 Testläufe) für alle besprochenen Implementierungen gegenüberstellt. Erkennbar ist, dass der Einsatz von `CUDA` hier einen ungefähr mit der `OpenMP` Implementierung vergleichbaren Performance-Gewinn liefert.

Abbildung 3.5: Streaming Multiprocessor Auslastung

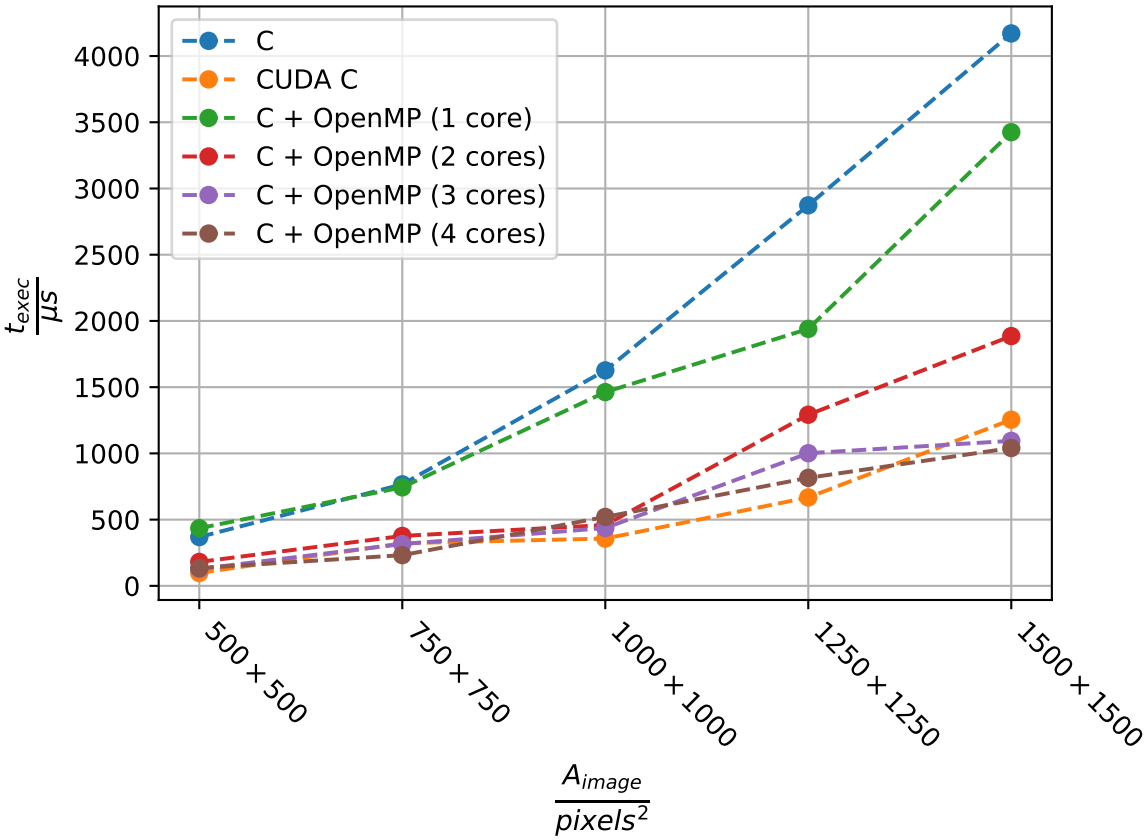


Abbildung 3.6: Übersichts-Vergleich Ausführungszeiten (Median aus je 100 Testläufen)

4 NUTZUNG DES REPOSITORYS

Eine Anmerkung zu Beginn: Während der Entwicklung der `CUDA`-Implementierung kam es zu mehreren Zeitpunkten zu Programm-Fehlschlägen die durch Treiber-Probleme hervorgerufen wurden. Sollte z.B. das `demo`-Programm mit einem durch `CUDA` hervorgerufenen `unknown error` abstürzen, hilft unter Umständen ein Neustart oder im schlimmsten Fall eine Neuinstallation des `NVIDIA`-Treibers (mittels des `CUDA-9.2`-Installers) ab.
Zudem ist die Kompilation aufgrund der Nutzung einer `OpenMP 4.5` Features unter Ubuntu nur mit `gcc-8` möglich (obwohl `OpenMP 4.5` eigentlich ab `gcc-6` unterstützt sein sollte). Hier muss eventuell das Makefile manuell modifiziert werden.

4.1 DEMO

Wird aus dem Wurzelverzeichnis des Repositorys `make demo` ausgeführt, werden die verschiedenen Implementierungen auf das Beispiel-Bild `images/demo_image.jpg` angewandt und die Ergebnisse dargestellt. Da die initialen Partitions-Schwerpunkte zufällig gewählt werden, kann es unter Umständen vorkommen, dass die Ergebnisse leicht unterschiedlich ausfallen (auch in Abhängigkeit der maximalen Anzahl von Iterationen). Abbildung 4.1 zeigt ein Beispiel.
Mittel `make repairtest` kann außerdem für die verschiedenen Implementierungen der Mechanismus zur Reparatur leerer Partitionen überprüft werden. Dabei wird der Algorithmus für zwei Partitionen auf ein überwiegend einfarbiges Bild angewandt. Hier ist die Wahrscheinlichkeit, dass Anfangs beide (zufällig gewählte) Partitions-Schwerpunkte identisch sind entsprechend hoch. Bei korrektem Reparatur-Mechanismus sollten im Ergebnis dennoch immer zwei farblich unterschiedliche Partitionen zu sehen sein, wie in Abbildung 4.2 gezeigt.

Abbildung 4.1: Beispielhafte Demo Ergebnisse

4.2 BENCHMARK

Bei Ausführung von `make benchmark` werden die Implementierungen hingegen entsprechend der im `Makefile` gesetzten Parameter auf zufällig generierte Bilder verschiedener Größen mit variierenden Partitions-Größen angewandt und jeweils die Ausführungszeiten[1] in `.csv` Datein im `benchmarks` Verzeichnis protokolliert (zur Neugenerierung von Benchmark-Daten müssen

[1] "wall-clock-time", gemessen mit `omp_get_wtime()`

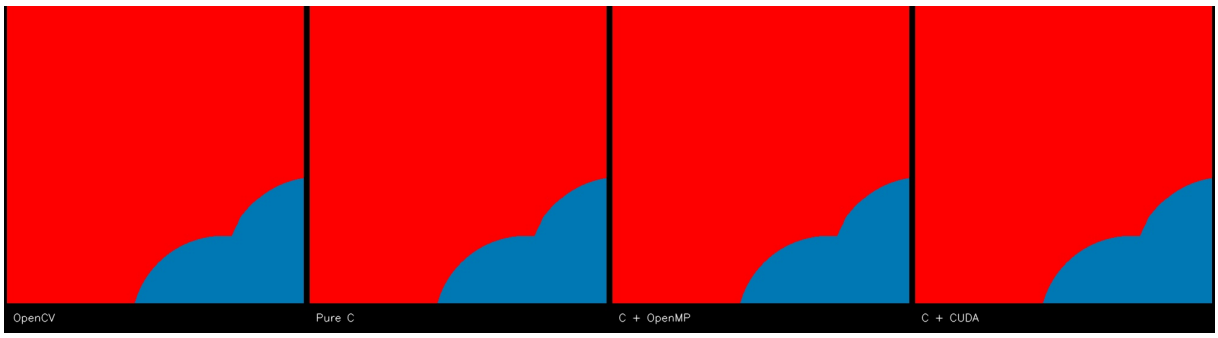


Abbildung 4.2: Beispielhafte `make repairtest` Ausgabe

dabei vorher existierende `.csv` Dateien gelöscht werden). Mit dem Python script `tool/plot` können die Ergbenisse visualisiert werden, dazu muss `./tool/plot benchmarks` ausgerufen werden (passiert bei Ausführung von `make benchmark` automatisch). Die resultierenden Grafiken finden sich unter `report/resources` und über diesen Bericht verteilt.

5 QUELLCODE

Alle genutzten Quellcode- und Build-Dateien:

Listing 5.1: `kmeans.h`

```h
#ifndef KMEANS_H
#define KMEANS_H

struct pixel
{
    double r, g, b;
};

void kmeans_c(struct pixel *pixels, size_t n_pixels,
              struct pixel *centroids, size_t n_centroids,
              size_t *labels);

void kmeans_omp(struct pixel *pixels, size_t n_pixels,
                struct pixel *centroids, size_t n_centroids,
                size_t *labels);

void kmeans_cuda(struct pixel *pixels, size_t n_pixels,
                 struct pixel *centroids, size_t n_centroids,
                 size_t *labels);

#endif
```

Listing 5.2: `kmeans_config.h`

```h
#pragma once

#ifndef KMEANS_MAX_ITER
  #define KMEANS_MAX_ITER 100
#endif
#ifndef KMEANS_CUDA_BLOCKSIZE
  #define KMEANS_CUDA_BLOCKSIZE 256
#endif
```

Listing 5.3: `kmeans.c`

```c
#include <float.h>
#include <math.h>
#include <omp.h>
#ifdef PROFILE
#include <stdio.h>
#endif
#include <stdlib.h>
#include <time.h>

#include "kmeans_config.h"
#include "kmeans.h"

// compute euclidean distance between two pixel values
static inline double pixel_dist(struct pixel p1, struct pixel p2)
{
    double dr = p1.r - p2.r;
    double dg = p1.g - p2.g;
```

```
    double db = p1.b - p2.b;

    return sqrt(dr * dr + dg * dg + db * db);
}

// find index of centroid with least distance to some pixel
static inline size_t find_closest_centroid(
    struct pixel pixel, struct pixel *centroids, size_t n_centroids)
{
    size_t closest_centroid = 0u;
    double min_dist = DBL_MAX;

    for (size_t i = 0; i < n_centroids; ++i) {
        double dist = pixel_dist(pixel, centroids[i]);

        if (dist < min_dist) {
            closest_centroid = i;
            min_dist = dist;
        }
    }

    return closest_centroid;
}

void kmeans_c(struct pixel *pixels, size_t n_pixels,
              struct pixel *centroids, size_t n_centroids,
              size_t *labels)
{
#ifdef PROFILE
    clock_t exec_begin;
    double exec_time_kernel1;
    double exec_time_kernel2;
    double exec_time_kernel3;
#endif

    // seed rand
    srand(time(NULL));

    // allocate auxiliary heap memory
    struct pixel *sums = malloc(n_centroids * sizeof(struct pixel));
    size_t *counts = malloc(n_centroids *  sizeof(size_t));

    // randomly initialize centroids
    for (size_t i = 0u; i < n_centroids; ++i) {
        centroids[i] = pixels[rand() % n_pixels];

        struct pixel tmp = { 0.0, 0.0, 0.0 };
        sums[i] = tmp;

        counts[i] = 0u;
    }

    // repeat for KMEANS_MAX_ITER or until solution is stationary
    for (int iter = 0; iter < KMEANS_MAX_ITER; ++iter) {
        int done = 1;

        // reassign points to closest centroids
#ifdef PROFILE
        exec_begin = clock();
#endif
        for (size_t i = 0u; i < n_pixels; ++i) {
            struct pixel pixel = pixels[i];

            // find centroid closest to pixel
            size_t closest_centroid =
                find_closest_centroid(pixel, centroids, n_centroids);

            // if pixel has changed cluster...
            if (closest_centroid != labels[i]) {
                labels[i] = closest_centroid;

                done = 0;
            }
```

```
91
92              // update cluster sum
93              struct pixel *sum = &sums[closest_centroid];
94              sum->r += pixel.r;
95              sum->g += pixel.g;
96              sum->b += pixel.b;
97
98              // update cluster size
99              counts[closest_centroid]++;
100         }
101 #ifdef PROFILE
102         exec_time_kernel1 = (double) (clock() - exec_begin) / CLOCKS_PER_SEC;
103 #endif
104
105         // repair empty clusters
106 #ifdef PROFILE
107         exec_begin = clock();
108 #endif
109         for (size_t i = 0u; i < n_centroids; ++i) {
110             if (counts[i])
111                 continue;
112
113             done = 0;
114
115             // determine largest cluster
116             size_t largest_cluster = 0u;
117             size_t largest_cluster_count = 0u;
118             for (size_t j = 0u; j < n_centroids; ++j) {
119                 if (j == i)
120                     continue;
121
122                 if (counts[j] > largest_cluster_count) {
123                     largest_cluster = j;
124                     largest_cluster_count = counts[j];
125                 }
126             }
127
128             // determine pixel in this cluster furthest from its centroid
129             struct pixel largest_cluster_centroid = centroids[largest_cluster];
130
131             size_t furthest_pixel = 0u;
132             double max_dist = 0.0;
133             for (size_t j = 0u; j < n_pixels; ++j) {
134                 if (labels[j] != largest_cluster)
135                     continue;
136
137                 double dist = pixel_dist(pixels[j], largest_cluster_centroid);
138
139                 if (dist > max_dist) {
140                     furthest_pixel = j;
141                     max_dist = dist;
142                 }
143             }
144
145             // move that pixel to the empty cluster
146             struct pixel replacement_pixel = pixels[furthest_pixel];
147             centroids[i] = replacement_pixel;
148             labels[furthest_pixel] = i;
149
150             // correct cluster sums
151             sums[i] = replacement_pixel;
152
153             struct pixel *sum = &sums[largest_cluster];
154             sum->r -= replacement_pixel.r;
155             sum->g -= replacement_pixel.g;
156             sum->b -= replacement_pixel.b;
157
158             // correct cluster sizes
159             counts[i] = 1u;
160             counts[largest_cluster]--;
161         }
162 #ifdef PROFILE
163         exec_time_kernel2 = (double) (clock() - exec_begin) / CLOCKS_PER_SEC;
```

```
#endif

        // average accumulated cluster sums
#ifdef PROFILE
        exec_begin = clock();
#endif
        for (int j = 0; j < n_centroids; ++j) {
            struct pixel *centroid = &centroids[j];
            struct pixel *sum = &sums[j];
            size_t count = counts[j];

            centroid->r = sum->r / count;
            centroid->g = sum->g / count;
            centroid->b = sum->b / count;

            sum->r = 0.0;
            sum->g = 0.0;
            sum->b = 0.0;

            counts[j] = 0u;
        }
#ifdef PROFILE
        exec_time_kernel3 = (double) (clock() - exec_begin) / CLOCKS_PER_SEC;
#endif

        // break if no pixel has changed cluster
        if (done)
            break;
    }
#ifdef PROFILE
    printf("Total kernel execution times:\n");
    printf("Kernel 1 (reassigning points to closest centroids): %.3e\n",
           exec_time_kernel1);
    printf("Kernel 2 (repairing empty clusters): %.3e\n",
           exec_time_kernel2);
    printf("Kernel 3 (average accumulated centroids): %.3e\n",
           exec_time_kernel3);
#endif

    free(sums);
    free(counts);
}

void kmeans_omp(struct pixel *pixels, size_t n_pixels,
                struct pixel *centroids, size_t n_centroids,
                size_t *labels)
{
    // seed rand
    srand(time(NULL));

    // allocate auxiliary heap memory
    double *sums = malloc(3 * n_centroids * sizeof(double));
    size_t *counts = malloc(n_centroids * sizeof(size_t));

    // randomly initialize centroids
    for (size_t i = 0u; i < n_centroids; ++i) {
        centroids[i] = pixels[rand() % n_pixels];

        double *sum = &sums[3 * i];
        sum[0] = sum[1] = sum[2] = 0.0;

        counts[i] = 0u;
    }

    // repeat for KMEANS_MAX_ITER or until solution is stationary
    for (int iter = 0; iter < KMEANS_MAX_ITER; ++iter) {
        int done = 1;

        // reassign points to closest centroids
        #pragma omp parallel for \
            reduction(+ : sums[:(3 * n_centroids)], counts[:n_centroids])
        for (int i = 0; i < n_pixels; ++i) {
            struct pixel pixel = pixels[i];
```

```
237
238             // find centroid closest to pixel
239             int closest_centroid =
240                 find_closest_centroid(pixel, centroids, n_centroids);
241
242             // if pixel has changed cluster...
243             if (closest_centroid != labels[i]) {
244                 labels[i] = closest_centroid;
245
246                 #pragma omp atomic write
247                 done = 0;
248             }
249
250             // update cluster sum
251             double *sum = &sums[3 * closest_centroid];
252             sum[0] += pixel.r;
253             sum[1] += pixel.g;
254             sum[2] += pixel.b;
255
256             // update cluster size
257             counts[closest_centroid]++;
258         }
259
260         // repair empty clusters
261         for (size_t i = 0u; i < n_centroids; ++i) {
262             if (counts[i])
263                 continue;
264
265             done = 0;
266
267             // determine largest cluster
268             size_t largest_cluster = 0u;
269             size_t largest_cluster_count = 0u;
270             for (size_t j = 0u; j < n_centroids; ++j) {
271                 if (j == i)
272                     continue;
273
274                 if (counts[j] > largest_cluster_count) {
275                     largest_cluster = j;
276                     largest_cluster_count = counts[j];
277                 }
278             }
279
280             // determine pixel in this cluster furthest from its centroid
281             struct pixel largest_cluster_centroid = centroids[largest_cluster];
282
283             size_t furthest_pixel = 0u;
284             double max_dist = 0.0;
285
286             #pragma omp parallel for
287             for (size_t j = 0u; j < n_pixels; ++j) {
288                 if (labels[j] != largest_cluster)
289                     continue;
290
291                 double dist = pixel_dist(pixels[j], largest_cluster_centroid);
292
293                 #pragma omp critical
294                 {
295                     if (dist > max_dist) {
296                         furthest_pixel = j;
297                         max_dist = dist;
298                     }
299                 }
300             }
301
302             // move that pixel to the empty cluster
303             struct pixel replacement_pixel = pixels[furthest_pixel];
304             centroids[i] = replacement_pixel;
305             labels[furthest_pixel] = i;
306
307             // correct cluster sums
308             double *sum = &sums[3 * i];
309             sum[0] = replacement_pixel.r;
```

```
            sum[1] = replacement_pixel.g;
            sum[2] = replacement_pixel.b;

            sum = &sums[3 * largest_cluster];
            sum[0] -= replacement_pixel.r;
            sum[1] -= replacement_pixel.g;
            sum[2] -= replacement_pixel.b;

            // correct cluster sizes
            counts[i] = 1u;
            counts[largest_cluster]--;
        }

        // average accumulated cluster sums
        #pragma omp parallel for
        for (int j = 0; j < n_centroids; ++j) {
            struct pixel *centroid = &centroids[j];
            double *sum = &sums[3 * j];
            size_t count = counts[j];

            centroid->r = sum[0] / count;
            centroid->g = sum[1] / count;
            centroid->b = sum[2] / count;

            sum[0] = sum[1] = sum[2] = 0.0;
            counts[j] = 0u;
        }

        // break if no pixel has changed cluster
        if (done)
            break;
    }

    free(sums);
    free(counts);
}
```

Listing 5.4: `kmeans_wrapper.h`

```
#pragma once

#include <omp.h>
#include <opencv2/core/core.hpp>

extern "C" {
#include "kmeans.h"
}

class KmeansWrapper
{
public:
    virtual void exec(cv::Mat const &image, size_t n_centroids) = 0;

    cv::Mat get_result() { return result; };
    double get_exec_time() { return _exec_time; };

    virtual ~KmeansWrapper() {}

protected:
    void start_timer() { _start_time = omp_get_wtime(); };
    void stop_timer() { _exec_time = (double) (omp_get_wtime() - _start_time); }
    cv::Mat result;

private:
    double _start_time;
    double _exec_time;
};

class KmeansOpenCVWrapper : public KmeansWrapper
{
public:
```

```
33      KmeansOpenCVWrapper() {}
34      void exec(cv::Mat const &image, size_t n_clusters);
35  };
36
37  class KmeansCWrapper : public KmeansWrapper
38  {
39  public:
40      KmeansCWrapper(
41          void (*impl)(struct pixel *, size_t, struct pixel *, size_t, size_t *),
42          int cores = 1) : impl(impl), cores(cores) {}
43
44      void exec(cv::Mat const &image, size_t n_clusters);
45
46  protected:
47      void (*impl)(struct pixel *, size_t, struct pixel *, size_t, size_t *);
48      int cores;
49  };
50
51  class KmeansCUDAWrapper : public KmeansCWrapper
52  {
53  public:
54      KmeansCUDAWrapper() : KmeansCWrapper(kmeans_cuda) {}
55  };
56
57  class KmeansOMPWrapper : public KmeansCWrapper
58  {
59  public:
60      KmeansOMPWrapper(int cores = 4) : KmeansCWrapper(kmeans_omp, cores) {}
61  };
62
63  class KmeansPureCWrapper : public KmeansCWrapper
64  {
65  public:
66      KmeansPureCWrapper() : KmeansCWrapper(kmeans_c) {}
67  };
```

Listing 5.5: `kmeans_wrapper.cc`

```
1   #include <omp.h>
2   #include <opencv2/core/core.hpp>
3
4   #include "kmeans_config.h"
5   #include "kmeans_wrapper.h"
6
7   void KmeansCWrapper::exec(cv::Mat const &image, size_t n_centroids) {
8
9       size_t n_pixels = image.rows * image.cols;
10
11      // allocate memory
12      std::vector<pixel> pixels(n_pixels);
13      for (int y = 0; y < image.rows; ++y) {
14          for(int x = 0; x < image.cols; ++x) {
15              int idx = y * image.cols + x;
16              pixels[idx].r = image.at<cv::Vec3b>(y, x)[2];
17              pixels[idx].g = image.at<cv::Vec3b>(y, x)[1];
18              pixels[idx].b = image.at<cv::Vec3b>(y, x)[0];
19          }
20      }
21
22      std::vector<pixel> centroids(n_centroids);
23      for (size_t i = 0; i < n_centroids; ++i) {
24          centroids[i].r = 0.0;
25          centroids[i].g = 0.0;
26          centroids[i].b = 0.0;
27      }
28
29      std::vector<size_t> labels(n_pixels);
30
31      // perform calculations
32      if (cores)
33          omp_set_num_threads(cores);
```

```
34
35      start_timer();
36      impl(&pixels[0], n_pixels, &centroids[0], n_centroids, &labels[0]);
37      stop_timer();
38
39      // rebuild image from results
40      result = cv::Mat(image.size(), image.type());
41      for (int y = 0; y < image.rows; ++y) {
42          for (int x = 0; x < image.cols; ++x) {
43              int label = labels[y * image.cols + x];
44              result.at<cv::Vec3b>(y, x)[2] = centroids[label].r;
45              result.at<cv::Vec3b>(y, x)[1] = centroids[label].g;
46              result.at<cv::Vec3b>(y, x)[0] = centroids[label].b;
47          }
48      }
49  }
50
51  void KmeansOpenCVWrapper::exec(cv::Mat const &image, size_t n_centroids) {
52
53      // construct input data points vector
54      cv::Mat data_points(image.rows * image.cols, 3, CV_32F);
55      for (int y = 0; y < image.rows; ++y) {
56          for (int x = 0; x < image.cols; ++x) {
57              for (int channel = 0; channel < 3; ++channel)
58                  data_points.at<float>(y *  image.cols + x, channel) =
59                      image.at<cv::Vec3b>(y, x)[channel];
60          }
61      }
62
63      // allocate space for labels
64      cv::Mat labels;
65
66      // transform cluster centers into **double format
67      cv::Mat centroids;
68
69      // specify termination criteria
70      cv::TermCriteria term(CV_TERMCRIT_ITER, KMEANS_MAX_ITER, 0);
71
72      // perform calculations
73      start_timer();
74      cv::kmeans(data_points, n_centroids, labels, term, 1,
75                 cv::KMEANS_RANDOM_CENTERS, centroids);
76      stop_timer();
77
78      // rebuild image from results
79      result = cv::Mat(image.size(), image.type());
80      for (int y = 0; y < image.rows; ++y) {
81          for (int x = 0; x < image.cols; ++x) {
82              int idx = labels.at<int>(y * image.cols +  x, 0);
83              for (int i = 0; i < 3; ++i)
84                  result.at<cv::Vec3b>(y, x)[i] = centroids.at<float>(idx, i);
85          }
86      }
87  }
```

Listing 5.6: `kmeans_benchmark.cc`

```
1   #include <cstring>
2   #include <fstream>
3   #include <vector>
4
5   #include <opencv2/opencv.hpp>
6
7   #include "kmeans_wrapper.h"
8
9   static int parse_intarg(char const *arg)
10  {
11      int res = 0;
12      try {
13          size_t idx;
14          res = std::stoi(arg, &idx);
```

```
15
16          if (idx != strlen(arg))
17              throw std::invalid_argument("trailing garbage");
18
19      } catch (std::exception const &e) {
20          throw std::invalid_argument(
21              std::string("malformed integer param: ") + e.what());
22      }
23
24      return res;
25  }
26
27  int main(int argc, char **argv)
28  {
29      std::vector<std::pair<std::string, KmeansWrapper *>> wrappers;
30
31      int const dim_min = parse_intarg(argv[1]);
32      int const dim_max = parse_intarg(argv[2]);
33      int const dim_step = parse_intarg(argv[3]);
34
35      int const clusters_min = parse_intarg(argv[4]);
36      int const clusters_max = parse_intarg(argv[5]);
37      int const clusters_step = parse_intarg(argv[6]);
38
39      int const n_exec = parse_intarg(argv[7]);
40
41      std::string csvdir(argv[8]);
42      if (csvdir.back() != '/')
43          csvdir += '/';
44
45      KmeansOpenCVWrapper opencv_wrapper;
46      wrappers.push_back(std::make_pair("OpenCV", &opencv_wrapper));
47
48      KmeansPureCWrapper pure_c_wrapper;
49      wrappers.push_back(std::make_pair("C", &pure_c_wrapper));
50
51      KmeansOMPWrapper omp_wrapper_single(1);
52      KmeansOMPWrapper omp_wrapper_double(2);
53      KmeansOMPWrapper omp_wrapper_triple(3);
54      KmeansOMPWrapper omp_wrapper_quad(4);
55      wrappers.push_back(std::make_pair("OpenMP_single", &omp_wrapper_single));
56      wrappers.push_back(std::make_pair("OpenMP_double", &omp_wrapper_double));
57      wrappers.push_back(std::make_pair("OpenMP_triple", &omp_wrapper_triple));
58      wrappers.push_back(std::make_pair("OpenMP_quad", &omp_wrapper_quad));
59
60      KmeansCUDAWrapper cuda_wrapper;
61      wrappers.push_back(std::make_pair("CUDA", &cuda_wrapper));
62
63      for (size_t i = 0u; i < wrappers.size(); ++i) {
64          std::string &name = std::get<0>(wrappers[i]);
65          KmeansWrapper *wrapper = std::get<1>(wrappers[i]);
66
67          std::string outfile(name + ".csv");
68          std::string csvfile(csvdir + outfile);
69
70          std::ifstream is(csvfile);
71          if (is.good())
72              continue;
73
74          std::cout << "creating: " << csvfile << '\n';
75
76          std::ofstream os(csvfile);
77          os << "dim,clusters,time\n";
78
79          for (int dim = dim_min; dim <= dim_max; dim += dim_step) {
80              std::cout << dim << "x" << dim << "...\n";
81
82              cv::Mat image = cv::Mat::zeros(dim, dim, CV_8UC3);
83              cv::randu(image, cv::Scalar(0, 0, 0), cv::Scalar(255, 255, 255));
84
85              for (int c = clusters_min; c <= clusters_max; c += clusters_step) {
86                  for (int i = 0; i <= n_exec; ++i) {
87                      wrapper->exec(image, c);
```

```
                    double t = wrapper->get_exec_time();
                    os << dim << ',' << c << ',' << t << '\n';
                }
            }
        }
    }
}
```

Listing 5.7: `kmeans_demo.cc`

```
#include <cstring>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

#include <opencv2/opencv.hpp>

#include "kmeans_wrapper.h"

int main(int argc, char **argv)
{
    cv::Mat image;
    int n_clusters;

    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " IMAGE CLUSTERS RESULT_OUT\n";
        return -1;
    }

    // load image
    image = cv::imread(argv[1]);
    if (image.empty()) {
        std::cerr << "Failed to load image file '" << argv[1] << "'\n";
        return -1;
    }

    // parse number of clusters
    try {
        size_t idx;
        n_clusters = std::stoi(argv[2], &idx);

        if (idx != strlen(argv[2]))
            throw std::invalid_argument("trailing garbage");

    } catch (std::exception const &e) {
        std::cerr << "Failed to parse number of clusters: " << e.what() << '\n';
        return -1;
    }

    // initialize implementation variants
    std::vector<std::pair<char const *, KmeansWrapper *>> impl;

    KmeansOpenCVWrapper opencv_wrapper;
    impl.push_back(std::make_pair("OpenCV", &opencv_wrapper));

    KmeansPureCWrapper pure_c_wrapper;
    impl.push_back(std::make_pair("Pure C", &pure_c_wrapper));

    KmeansOMPWrapper omp_c_wrapper;
    impl.push_back(std::make_pair("C + OpenMP", &omp_c_wrapper));

    KmeansCUDAWrapper cuda_c_wrapper;
    impl.push_back(std::make_pair("C + CUDA", &cuda_c_wrapper));

    // setup results display window
    const int margin = 10;

    cv::Mat win_mat(
        cv::Size((image.cols + margin) * impl.size(), image.rows + 50),
        CV_8UC3, cv::Scalar(0, 0, 0)
```

```
    );

    int offs = margin / 2;
    for (auto const &pane: impl) {
        char const *title = std::get<0>(pane);

        // get result and execution time for each wrapper
        KmeansWrapper *wrapper = std::get<1>(pane);
        wrapper->exec(image, n_clusters);
        cv::Mat result = wrapper->get_result();
        //double exec_time = wrapper->get_exec_time();

        result.copyTo(win_mat(cv::Rect(offs, 0, image.cols, image.rows)));

        // construct subtitles
        std::ostringstream subtext;
        subtext << title; //<< " (" << std::setprecision(2) << exec_time
                          //    << " sec.)";

        cv::putText(win_mat, subtext.str(),
                    cvPoint(offs + margin, image.rows + 30),
                    cv::FONT_HERSHEY_SIMPLEX, 0.6, cv::Scalar(255,255,255),
                    1.0, CV_AA);

        offs += image.cols + margin;
    }

    image.release();

    // display window
    std::string disp_title(argv[1]);
    cv::namedWindow(disp_title, cv::WINDOW_AUTOSIZE);
    cv::imshow(disp_title, win_mat);
    cv::imwrite(argv[3], win_mat);
    try {
        cv::waitKey(0);
        cv::destroyWindow(disp_title);
    } catch (cv::Exception const &e) {
        // demo window was closed externally
    }

    win_mat.release();

    return 0;
}
```

Listing 5.8: `Makefile`

```
# directories ######################################################################

C_SRC_DIR=c/src
C_OBJ_DIR=c/obj
C_INCLUDE_DIR=c/include

CPP_SRC_DIR=cpp/src
CPP_OBJ_DIR=cpp/obj
CPP_INCLUDE_DIR=cpp/include

CONFIG_DIR=config
BENCHMARK_OUT_DIR=benchmarks
BUILD_DIR=build
IMAGE_DIR=images
REPORT_DIR=report
REPORT_AUX_DIR=$(REPORT_DIR)/aux
REPORT_PDF_DIR=$(REPORT_DIR)/pdf
REPORT_RESOURCE_DIR=$(REPORT_DIR)/resources
REPORT_TEX_DIR=$(REPORT_DIR)/tex

# parameters #######################################################################

PROFILE_IMAGE=$(IMAGE_DIR)/profile_image.jpg
```

```
PROFILE_CLUSTERS=5

BENCHMARK_DIM_MIN=500
BENCHMARK_DIM_MAX=1500
BENCHMARK_DIM_STEP=250
BENCHMARK_CLUSTER_MIN=5
BENCHMARK_CLUSTER_MAX=5
BENCHMARK_CLUSTER_STEP=1
BENCHMARK_N_EXEC=100
BENCHMARK_PLOT=tool/plot.py

DEMO_IMAGE=$(IMAGE_DIR)/demo_image.jpg
DEMO_CLUSTERS=5
DEMO_RESULT_OUT=$(REPORT_RESOURCE_DIR)/demo_results.jpg

REPAIRTEST_IMAGE=$(IMAGE_DIR)/repairtest_image.bmp
REPAIRTEST_CLUSTERS=2
REPAIRTEST_RESULT_OUT=$(REPORT_RESOURCE_DIR)/repairtest_results.jpg

# compilation settings ########################################################

CC_C=gcc
CC_CUDA=nvcc
CC_CPP=g++

C_CFLAGS=-std=c99 -Wall -g -O3 -I$(C_INCLUDE_DIR) -I$(CONFIG_DIR) -fopenmp
CPP_CFLAGS=-std=c++11 -Wall -g -O0 -I$(CONFIG_DIR) -I$(C_INCLUDE_DIR) \
           -I$(CPP_INCLUDE_DIR) `pkg-config --cflags opencv`
CUDA_CFLAGS=-I$(CONFIG_DIR) -I$(C_INCLUDE_DIR)

LCV=`pkg-config --libs opencv`
LOMP=-fopenmp
LCUDA=-L/usr/local/cuda/lib64 -lcudart

# build sources ###############################################################

all: $(BUILD_DIR)/demo $(BUILD_DIR)/profile $(BUILD_DIR)/benchmark

$(BUILD_DIR)/demo: $(CPP_OBJ_DIR)/kmeans_demo.o \
 $(C_OBJ_DIR)/kmeans.o $(C_OBJ_DIR)/kmeans_cuda.o \
 $(CPP_OBJ_DIR)/kmeans_wrapper.o
	$(CC_CPP) -o $@ $^ $(LCV) $(LOMP) $(LCUDA)

$(BUILD_DIR)/profile: $(CPP_OBJ_DIR)/kmeans_profile.o \
  $(C_OBJ_DIR)/kmeans_profile.o $(CPP_OBJ_DIR)/kmeans_wrapper.o
	$(CC_CPP) -o $@ $^ $(LCV) $(LOMP)

$(BUILD_DIR)/benchmark: $(CPP_OBJ_DIR)/kmeans_benchmark.o \
  $(C_OBJ_DIR)/kmeans.o $(C_OBJ_DIR)/kmeans_cuda.o \
  $(CPP_OBJ_DIR)/kmeans_wrapper.o
	$(CC_CPP) -o $@ $^ $(LCV) $(LOMP) $(LCUDA)

$(C_OBJ_DIR)/kmeans_cuda.o: $(C_SRC_DIR)/kmeans.cu \
  $(C_INCLUDE_DIR)/kmeans.h $(CONFIG_DIR)/kmeans_config.h
	$(CC_CUDA) -c -o $@ $< $(CUDA_CFLAGS)

$(C_OBJ_DIR)/kmeans_profile.o: $(C_SRC_DIR)/kmeans.c \
  $(C_INCLUDE_DIR)/kmeans.h $(CONFIG_DIR)/kmeans_config.h
	$(CC_C) -c -o $@ $< $(C_CFLAGS) -DPROFILE

$(C_OBJ_DIR)/%.o: $(C_SRC_DIR)/%.c \
  $(C_INCLUDE_DIR)/kmeans.h $(CONFIG_DIR)/kmeans_config.h
	$(CC_C) -c -o $@ $< $(C_CFLAGS)

$(CPP_OBJ_DIR)/%.o: $(CPP_SRC_DIR)/%.cc \
  $(C_INCLUDE_DIR)/kmeans.h $(CONFIG_DIR)/kmeans_config.h \
  $(CPP_INCLUDE_DIR)/kmeans_wrapper.h
	$(CC_CPP) -c -o $@ $< $(CPP_CFLAGS)

# build report ################################################################

define run_pdflatex
  pdflatex -halt-on-error -shell-escape -output-directory $(REPORT_AUX_DIR) $< > /dev/null
```

```
endef

report: $(REPORT_PDF_DIR)/report.pdf

$(REPORT_PDF_DIR)/report.pdf: $(REPORT_TEX_DIR)/report.tex $(REPORT_RESOURCE_DIR)/*
	$(call run_pdflatex)
	$(call run_pdflatex)
	-@mv $(REPORT_AUX_DIR)/report.pdf $(REPORT_PDF_DIR)

# PHONY rules ###################################################################

.PHONY: demo, profile, benchmark, clean

demo: $(BUILD_DIR)/demo $(DEMO_IMAGE)
	./$(BUILD_DIR)/demo $(DEMO_IMAGE) $(DEMO_CLUSTERS) $(DEMO_RESULT_OUT)

repairtest: $(BUILD_DIR)/demo $(REPAIRTEST_IMAGE)
	./$(BUILD_DIR)/demo $(REPAIRTEST_IMAGE) $(REPAIRTEST_CLUSTERS) \
	$(REPAIRTEST_RESULT_OUT)

profile: $(BUILD_DIR)/profile $(PROFILE_IMAGE)
	./$(BUILD_DIR)/profile $(PROFILE_IMAGE) $(PROFILE_CLUSTERS)

benchmark: $(BUILD_DIR)/benchmark $(BENCHMARK_PLOT)
	./$(BUILD_DIR)/benchmark \
	$(BENCHMARK_DIM_MIN) $(BENCHMARK_DIM_MAX) $(BENCHMARK_DIM_STEP) \
	$(BENCHMARK_CLUSTER_MIN) $(BENCHMARK_CLUSTER_MAX) $(BENCHMARK_CLUSTER_STEP) \
	$(BENCHMARK_N_EXEC) $(BENCHMARK_OUT_DIR)
	./$(BENCHMARK_PLOT) $(BENCHMARK_OUT_DIR)

clean:
	rm $(C_OBJ_DIR)/*.o 2> /dev/null || true
	rm $(CPP_OBJ_DIR)/*.o 2> /dev/null || true
	rm $(BUILD_DIR)/* 2> /dev/null || true
	rm $(REPORT_AUX_DIR)/* 2> /dev/null || true
	rm $(REPORT_PDF_DIR)/* 2> /dev/null || true
```

LITERATURVERZEICHNIS

[1] Lloyd., S. P. (1982). "Least squares quantization in PCM". IEEE Transactions on Information Theory

QUELLCODEVERZEICHNIS