Exercise 2: Solutions

Hardware Accelerators: K-means Clustering

Prof. Dr. Matthias S. Müller
Dr. Christian Terboven
Dr. Sandra Wienke
Julian Miller

Task 1

Task 1.1: Preparations

- This exercise can be done on the RWTH cluster environment CLAIX
- Access via lecture project lect0053
  - Register for an account “Hochleistungsrechnen RWTH Aachen” at the selfservice (http://www.rwth-aachen.de/selfservice)
  - Sent your TIM ID to contact@hpc.rwth-aachen.de in case you are no member yet
- Any problems with the cluster environment?

Task 1.2: Implementation of K-means

- Implement the proposed k-means algorithm as denoted in the code by TODO: task 1.2.

```
void k_means(int niters, point_t *points, point_t *centroids,
             int *assignment, point_t *result, int n, int k) {
  for (int iter = 0; iter < niters; ++iter) {
    // determine nearest centroids
    for (int i = 0; i < n; ++i) {
      double optimal_dist = DBL_MAX;      // Calculate Euclidean
distance to each centroid and
         determine the closest mean
      for (int j = 0; j < k; ++j) {
        double dist = (points[i].x - centroids[j].x) *
                      (points[i].x - centroids[j].x) +
                      (points[i].y - centroids[j].y) *
                      (points[i].y - centroids[j].y);
        if (dist < optimal_dist) {
          optimal_dist = dist;
          assignment[i] = j;
    } } }
```

Task 1.2: Implementation of K-means

```
    // Calculate new positions of centroids
    int count[k];
    double sum_x[k];
    double sum_y[k];
    for (j = 0; j < k; ++j) {
      count[j] = 0;
      sum_x[j] = 0.0;
      sum_y[j] = 0.0;
    }
    for (i = 0; i < n; ++i) {
      count[assignment[i]]++;
      sum_x[assignment[i]] += points[i].x;
      sum_y[assignment[i]] += points[i].y;
    }
    for (j = 0; j < k; ++j) {
      if (count[j] != 0.0) {
        centroids[j].x = sum_x[j] / count[j];
        centroids[j].y = sum_y[j] / count[j];
} } } }
```

```
$ make run-small
Executing k-means clustering with
20 iterations, 1000 points, and 5
centroids...
Time Elapsed: 0.000235 s
```

Task 1.3: Deploy: Check for Correctness

- Verify that the results obtained in task 1.2 are correct (template: TODO: task 1.3).
  - Write initial position of centroids to result before execution the algorithm:

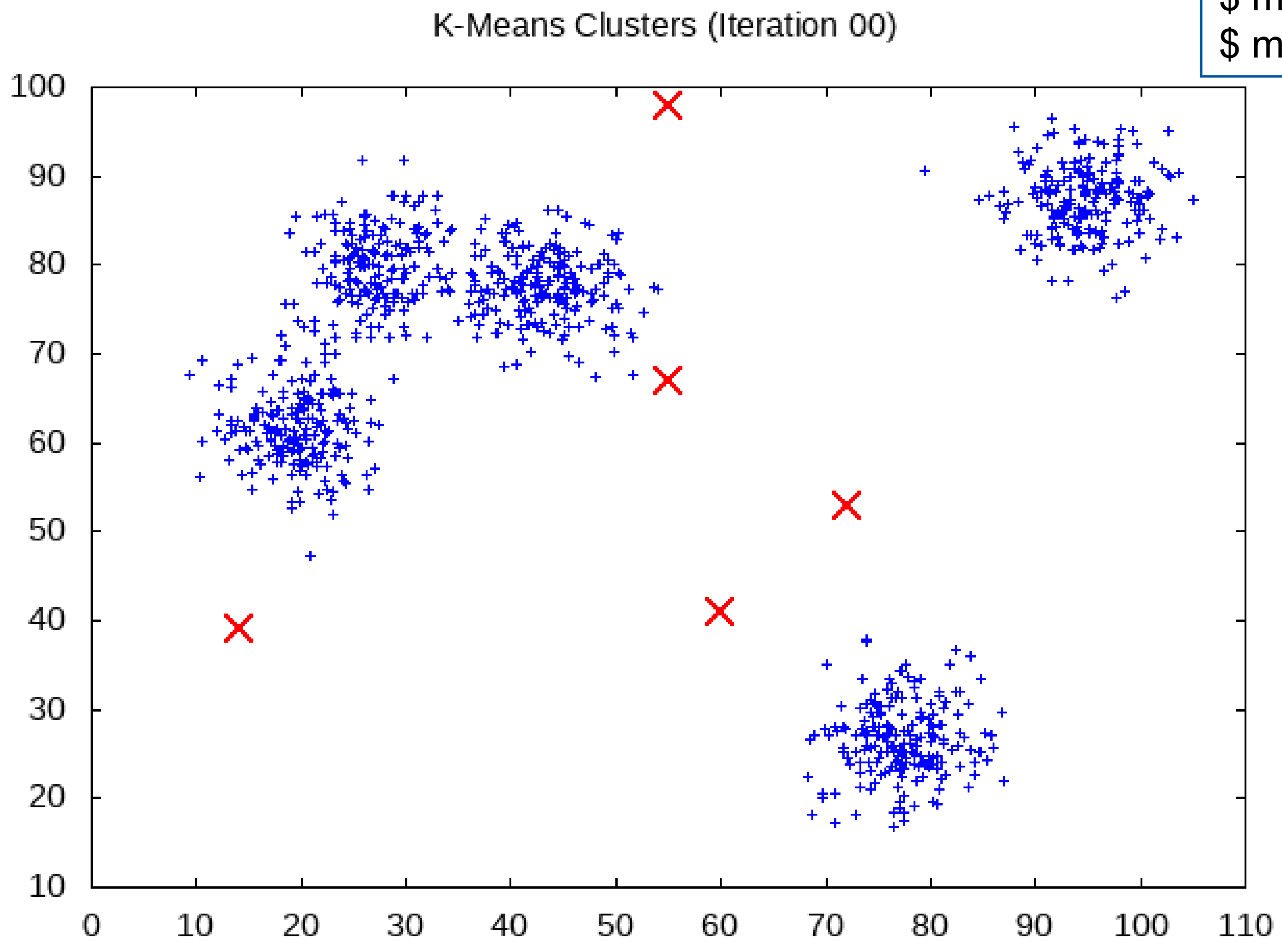
```
for (int i = 0; i < k; ++i) {
  result[i].x = centroids[i].x;
  result[i].y = centroids[i].y;
}
```

  - Store results after each iteration:

```
for (int j = 0; j < k; ++j) {
  if (count[j] != 0) {
    centroids[j].x = sum_x[j] / count[j];
    centroids[j].y = sum_y[j] / count[j];
  }
  result[(iter + 1) * k + j].x = centroids[j].x;
  result[(iter + 1) * k + j].y = centroids[j].y;
} }
```

Task 1.3: Deploy: Check for Correctness

```
$ module load MISC gnuplot
$ make vis-small
```



Task 1.4: Optimization of K-means

- Optimize the serial implementation.
- Remember AoS vs. SoA

```
struct point_t {
  double x;
  double y;
};
```

```
struct point_aos_t {
  double *x;
  double *y;
};
```

structure of array vs array of structure

locality of reference!

When CPU processes an instruction or data,
it fetches them from cache.
When we use AoS and load sth from cache,
data are loaded as a whole structure.
When we use SoA, CPU can read just *x of
each structures.
If we have to use every element of a structure, AoS can be useful
because the whole data of each structure can be loaded.
Otherwise, SoA will enhance the speed

Depending on the usage, we should choose one of the structures
properly to enhance the performance

Address	0	8	16	24	32	40
AoS	x[0]	y[0]	x[1]	y[1]	x[2]	y[2]
SoA	x[0]	x[1]	x[2]	y[0]	y[1]	y[2]

Task 1.4: Optimization of K-means

- Change all accesses from *[i].x to *->x[i]

```
int main(int argc, const char* argv[]) {
  ...
  // Initialize points and centroids in SoA format
  point_soa_t points_soa;
  points_soa.x = (double*) malloc(n*sizeof(double));
  points_soa.y = (double*) malloc(n*sizeof(double));
  for (int i = 0; i < (niters + 1) * k; ++i) {
    result[i].x = -1.0;
    result[i].y = -1.0;
  }
  for (int i = 0; i < k; ++i) {
    result->y[i] = centroids->x[i];
    result->y[i] = centroids->y[i];
  }
  ...
}
```

```
$ make run-small
Executing k-means clustering with
20 iterations, 1000 points, and 5
centroids...
Time Elapsed (AoS): 0.000235 s
Time Elapsed (SoA): 0.000235 s
```

Task 1.4: Optimization of K-means

- Why is there no performance difference?
  - Optimal coalescing: all data read is requested

- Would the following data structure change things?
  - Yes, to compute distance only x and y are needed
  - Degree of coalescing = $\frac{\#\text{bytes requested}}{\#\text{bytes read}} = \frac{16 \text{ bytes}}{24 \text{ bytes}} = \frac{2}{3}$

```
struct point_t {
  double x;
  double y;
  int assignment;
};
```

Address	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68
AoS	x[0]		y[0]		a[0]		x[1]		y[1]		a[1]		x[2]		y[2]		a[2]	
SoA	x[0]		x[1]		x[2]		y[0]		y[1]		y[2]		a[0]	a[1]	a[2]			

Task 2

Preparations

- Frontend for development and short test
  - `$ ssh login18-g-1.hpc.itc.rwth-aachen.de`
  - Load NVIDIA compiler
    - `$ module load cuda/112`
  - List loaded modules
    - `$ module list`

- Backend for performance measurements
  - `$ sbatch kmeans_gpu.sh`

Task 2.1: Assess: Investigation of Parallelism

- Identify the hotspot of the algorithm of the optimized and checked serial version (task 1.4) and evaluate which parts are suitable for the GPU. Investigate which steps of the algorithm can be parallelized and how it can be achieved. Reason about dependencies.
  - Profiler shows 100% time spent in `kmeans()`
  - 3 nested loops:
    - `for (iter = 0; iter < NO_ITER; ++iter) {`
      - Outer loop over iterations not parallelizable
    - `for (i = 0; i < n; ++i) {`
      - 2nd loop parallelizable (no data dependencies)
    - `for (j = 0; j < k; ++j) {`
      - 3rd loop parallelizable with a reduction on `optimal_dist`

Task 2.2: Assess: Performance Modeling of K-Means

- Model the execution time t_{GPU} of the hotspot on the V100 GPU based on the performance model introduced in the lecture. What limits the execution time of the hotspot?
  - No data dependencies in 2nd loop
  - Many computations to offload

Task 2.2: Assess: Performance Modeling of K-Means

- n*k operations per iteration with
  - Simplification: leave out update to `optimal_dist`
  - 7 DP operations
  - 4 READS from main memory (down to 0 reads if `points` and `centroids` can be cached)

```
for (int iter = 0; iter < niters; ++iter) {
   for (int i = 0; i < n; ++i) {
     double optimal_dist = DBL_MAX;
     for (int j = 0; j < k; ++j) {
       double dist = (points[i].x - centroids[j].x) *
                     (points[i].x - centroids[j].x) +
                     (points[i].y - centroids[j].y) *
                     (points[i].y - centroids[j].y);
       if (dist < optimal_dist) {
         optimal_dist = dist;
         assignment[i] = j;
   } } }
```

Task 2.2: Assess: Performance Modeling of K-Means

- `Count`, `sum_x` and `sum_y` can be (for reasonable large k) cached
- 2n READ of `points` and n READ of `assignment`
- 4k WRITE to `centroids` and `result`
- 2n DP operations (additions on `sum_x` and `sum_y`) and 2k DP operations (divisions for calculation of `centroids`)

```
int count[k];
double sum_x[k];
double sum_y[k];
for (j = 0; j < k; ++j) {
  count[j] = 0;
  sum_x[j] = 0.0; sum_y[j] = 0.0;
}
for (i = 0; i < n; ++i) {
  count[assignment[i]]++;
  sum_x[assignment[i]] += points[i].x;
  sum_y[assignment[i]] += points[i].y;
}
for (int j = 0; j < k; ++j) {
  if (count[j] != 0) {
    centroids[j].x = sum_x[j] / count[j];
    centroids[j].y = sum_y[j] / count[j];
  }
  result[(iter + 1) * k + j].x = centroids[j].x;
  result[(iter + 1) * k + j].y = centroids[j].y;
} }
```

Task 2.2: Assess: Performance Modeling of K-Means

- Total kernel：
  - n*k*7 + 2k +2n DP operations
  - n*k*4 + 2n + 4k DP READs/ WRITEs and n Integer READs
  - Dominated by main loop over n and k
  - Operational intensity of main loop: 7 Flops/32 byte

Task 2.2: Assess: Performance Modeling of K-Means

- Performance Model
  - $t_{kernel} = \max(t_{compute}, t_{memory})$
  - $t_{compute} = \frac{arithmetic\ operations\ [Flop]}{P_{max}}$
    - No. DP operations: no_iter * n * k * 7 Flop
    - P_{max} = 2560 cores * 1.3 GHz * 2 = 6656 GFlop/s
  - $t_{memory} = \frac{data\ transfers(LOAD, STORE)\ [words]}{b_s}$
    - Data transfers: no_iter * n * k * 4 Words
    - b_s measured with BabelStream benchmark: 865 GB/s

Task 2.2: Assess: Performance Modeling of K-Means

- Performance Model
  - $t_{GPU} = t_{H2D} + t_{kernel} + t_{D2H}$
  - $t_{data} = t_{H2D} + t_{D2H}$
    - H2D: points (2n doubles), centroids (2k doubles), assigments (n integers), result (2k (no_iter+1) doubles)
    - D2H: result (2k (no_iter+1) doubles)
    - b_{PCI} measured: 12 GB/s
    - $t_{data} = 2\alpha + \frac{2(n+k+k(no\_iter+1)) * 8\ bytes + n * 4\ bytes}{12\ GB/s} + \frac{2k(no\_iter+1) * 8\ bytes}{12\ GB/s}$

Task 2.3: Port K-means to GPU

- Offload the identified hotspot in task 2.1 to a GPU while taking care of the required data.

```
int main(int argc, const char* argv[]) {
  ...
  // Allocate memory for GPU
  point_t *d_points = 0; point_t *d_centroids = 0;
  int *d_assignments = 0; point_t *d_result = 0;
  cudaMalloc((void**)&d_points, N * sizeof(point_t));
  cudaMalloc((void**)&d_centroids, K * sizeof(point_t));
  cudaMalloc((void**)&d_assignments, N * sizeof(int));
  cudaMalloc((void**)&d_result, (niters + 1) * k * sizeof(point_t));
  // Copy data to GPU
  double runtime_all = get_time();
  cudaMemcpy(d_points, h_points, N * sizeof(point_t), cudaMemcpyHostToDevice);
  cudaMemcpy(d_centroids, h_centroids, K * sizeof(point_t), cudaMemcpyHostToDevice);
  cudaMemcpy(d_assignments, h_assignments, N * sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_result, result, (niters + 1) * k * sizeof(point_t),
cudaMemcpyHostToDevice);
```

Task 2.3: Port K-means to GPU

```
 // Apply k-means algorithm
 double runtime_kernel = get_time();
 k_means<<<(n + THREADSPERBLOCK - 1)/THREADSPERBLOCK,
            THREADSPERBLOCK>>>(d_points, d_centroids, d_assignment
            d_result, n, k);
 cudaDeviceSynchronize();
 runtime_kernel = get_time() - runtime_kernel;

 // Copy results back to host
cudaMemcpy(result, d_result, (niters + 1) * k * sizeof(point_t),
           cudaMemcpyDeviceToHost);
 runtime_all = get_time() - runtime_all;

 // Free memory
 cudaFree(d_result);
 cudaFree(d_assignment);
 cudaFree(d_centroids);
 cudaFree(d_points);
```

Task 2.3: Port K-means to GPU

- Intuitive solution

```
__global__ void k_means(point_t *points,
                        point_t *centroids,
                        int *assignment) {
  int iter, j;
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  for (iter = 0; iter < NO_ITER; ++iter) {
    if (tid < n) {
      double optimal_dist = DBL_MAX;
      // Calculate Euclidean distance to each centroid and
         determine the closest mean
      for (j = 0; j < K; ++j) {
        double dist = (points[tid].x - centroids[j].x) *
                      (points[tid].x - centroids[j].x) +
                      (points[tid].y - centroids[j].y) *
                      (points[tid].y - centroids[j].y);
        if (dist < optimal_dist) {
          optimal_dist = dist;
          assignment[tid] = j;
    } } }
```

Task 2.3: Port K-means to GPU

```
    // Calculate new positions of centroids
    if (tid < k) {
      int count = 0;
      double sum_x = 0.0;
      double sum_y = 0.0;
      for (int j = 0; j < n; ++j) {
        if (assignment[j] == tid) {
          sum_x += points[j].x;
          sum_y += points[j].y;
          count++;
        }
      }
      if (count != 0.0) {
        centroids[tid].x = sum_x / count;
        centroids[tid].y = sum_y / count;
      }
      result[(iter + 1) * k + tid].x = centroids[tid].x;
      result[(iter + 1) * k + tid].y = centroids[tid].y;
} } }
```

```
$ make run-small
Executing k-means clustering with
20 iterations, 1000 points, and 5
centroids...
Time Elapsed (kernel): 0.000440 s
Time Elapsed (total): 0.000529 s
```

- Wrong results! Why?

Task 2.3: Port K-means to GPU

- Two separate kernels are required for synchronization

```
__global__ void calc_distances(point_t *points, point_t *centroids,
                               int *assignment, int n, int k) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < n) {
    double optimal_dist = DBL_MAX;
    for (int j = 0; j < k; ++j) {
      double dist = (points[tid].x - centroids[j].x) *
                    (points[tid].x - centroids[j].x) +
                    (points[tid].y - centroids[j].y) *
                    (points[tid].y - centroids[j].y);
      if (dist < optimal_dist) {
        optimal_dist = dist;
        assignment[tid] = j;
} } } }
```

Task 2.3: Port K-means to GPU

```
__global__ void update_centroids(int iter, point_t *points, point_t *centroids,
                                  int *assignment, point_t *result, int n, int k)
{
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < k) {
    int count = 0;
    double sum_x = 0.0;
    double sum_y = 0.0;
    for (int j = 0; j < n; ++j) {
      if (assignment[j] == tid) {
        sum_x += points[j].x;
        sum_y += points[j].y;
        count++;
    } }
    if (count != 0.0) {
      centroids[tid].x = sum_x / count;
      centroids[tid].y = sum_y / count;
    }
    result[(iter + 1) * k + tid].x = centroids[tid].x;
    result[(iter + 1) * k + tid].y = centroids[tid].y;
} }
```

```
$ make run-small
Executing k-means clustering with
20 iterations, 1000 points, and 5
centroids...
Time Elapsed (kernel): 0.001929 s
Time Elapsed (total): 0.002001 s
```

Task 2.4: Optimize and Parallelize K-means on the GPU

- Optimize the algorithm and parallelize it to utilize the GPU as much as possible. Keep the data layout optimizations from task 1.4 in mind.
  - Similar findings here: optimal coalescing
  - Distribute updating of centroids over multiple thread blocks to prevent serialization
  - Updating of centroids could be done in shared memory

Task 2.4: Optimize and Parallelize K-means on the GPU

- Move assignment into first kernel

```
__global__ void assign_clusters(point_t *points, point_t *centroids, point_t *sums,
                                int *counts, int n, int k) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= n) return;
    const double x = points[idx].x;
    const double y = points[idx].y;
    double optimal_dist = DBL_MAX;
    int assignment;
    for (int j = 0; j < k; ++j) {
        const double dist = (x - centroids[j].x) * (x - centroids[j].x) +
                            (y - centroids[j].y) * (y - centroids[j].y);
        if (dist < optimal_dist) {
            optimal_dist = dist;
            assignment = j;
        }
    }
    atomicAdd(&sums[assignment].x, x);
    atomicAdd(&sums[assignment].y, y);
    atomicAdd(&counts[assignment], 1);
}
```

Task 2.4: Optimize and Parallelize K-means on the GPU

- Start k blocks with one thread each to avoid branch divergence
  - compute_new_means<<<k, 1>>>(d_centroids, d_sums, d_counts, d_result, k, iter);

```
__global__ void compute_new_means(point_t *centroids, point_t *sums, const int *counts,
                                  point_t *result, int k, int iter) {
    if (threadIdx.x > 0) return;
    const int cluster = blockIdx.x;
    if (counts[cluster] != 0.0) {
        centroids[cluster].x = sums[cluster].x / counts[cluster];
        centroids[cluster].y = sums[cluster].y / counts[cluster];
    }
    result[(iter + 1) * k + cluster].x = centroids[cluster].x;
    result[(iter + 1) * k + cluster].y = centroids[cluster].y;
}
```

```
$ make run-small
Executing k-means clustering with
20 iterations, 1000 points, and 5
centroids...
Time Elapsed (kernel): 0.000440 s
Time Elapsed (total): 0.000510 s
```

Task 2.5: Deploy: Verify Results

- Check you results for correctness and evaluate the actual performance with the performance estimated with your model in task 3.1. How close is you implementation to the theoretical peak performance for that hotspot based on the model? Evaluate reasons for potential differences between the modeled and the achieved performance.
  - Exact same results

Task 2.5: Deploy: Verify Results

- Performance Comparison (large data set)
  - $t_{data} = t_{total} - t_{kernel} =$ 2.932300 s – 2.926127 s ≈ 6.2 ms
    - Modelled: $t_{data} = 2\alpha + \frac{2(n+k+k(no\_iter+1)) * 8\ bytes + n * 4\ bytes}{12\ GB/s} + \frac{2k(no\_iter+1) * 8\ bytes}{12\ GB/s} \approx \frac{2(1000000+5000+10000(50+1)) * 8\ bytes + 1000000 * 4\ bytes}{12\frac{GB}{s}} \approx$ 2.35 ms
  - $t_{kernel} =$ 2.926127
    - Modelled: $t_{kernel} = \max(t_{compute}, t_{memory})$
    - $t_{compute} = \frac{arithmetic\ operations}{P_{max}} = \frac{50 * 1000000 * 5000 * 7\ \text{Flop}}{6656\ \text{GFlop/s}} =$ 262.9 ms
    - $t_{memory} = \frac{data\ transfers(LOAD, STORE)}{b_s} = \frac{50 * 1000000 * 5000 * 4 * 8\ \text{bytes}}{865\ GB/s} =$ 9.25 s
- Lots of caching

Task 2.6: Performance Comparison

- Compare your obtained performance results of the k-means algorithm on the GPU to the ones on the CPU (task 1.4). Is this problem suitable for the GPU (meaning: is the algorithm accelerated by the usage of the GPU)? Justify your answer.
  - $t_{CPU} = 126.705629$ s
  - $t_{GPU} = 2.932300$ s (0.42 s for the optimized version)
  - Speedup of ~43
- Is this comparison fair? Justify your answer.
  - No, use all 48 cores on CPU node
  - $t_{CPU} = 5.427817$ s -> 322.4 Gflop/s ($P_{max,CPU} = 2150$ Gflop/s)
  - $t_{GPU} = 2.932300$ s -> 596.8 Gflop/s ($P_{max,GPU} = 6656$ Gflop/s)
  - Speedup of ~1.85
- Optimized Version
  - $t_{GPU} = 0.425751$ s -> 4110.4 Gflop/s ($P_{max,GPU} = 6656$ Gflop/s)
  - Speedup of ~12.75

Thank you for your kind attention.