May 14, 2018

Abstract

1 Maximum performance of the kmeans kernel

The kernel of the `kmeans` algoirthm is in the `kmeans_clustering` function.

The function is split into three main parts:

- Initialization (happens once)
- In the main loop:
  - Calculating membership to existing clusters for each point.
  - Update clusters.

We can safely ignore initialization for our performance consideration which leaves the main loop.

1.1 The Main Loop

We can split the main loop again into three somewhat distinct parts.

- In a parallel loop:
  - Find the nearest cluster for each point.
  - Collect partial updates on the position of clusters.
- Reduce partial information and update positions of clusters

For our analysis we assume 5 clusters and 34 features which matches the default of kmeans and our input datasets.

The only significant contributors here are the loops to find the nearest cluster and updating the cluster information. For this reason we will look only at these inner loops and not at the rest of the main loop.

1.1.1 Main loop:Finding the nearest cluster

Listing 1: Inlined representation of find_nearest_point

```
for (i=0; i<nclusters; i++) {
        float dist=0;
        for (j=0; j<nfeatures; j++)
                dist += (pt1[j]-clusters[i][j]) * (pt1[j]-clusters[i][j]);
        if (dist < min_dist) {
                min_dist = dist;
                index    = i;
        }
}
```

The kernel here is: `dist += (pt1[j]-pt2[j]) * (pt1[j]-pt2[j])`.

We get 8 [1] byte of memory bandwith and three floating point operations. Memory bandwith consists of 2 loads for each iteration. Operations are two additions and one multiplication. [2] [3]

- Read: `pt1[j]`
- Read: `clusters[i][j]`

Immediatly obvious from the snippet above we get at least $nclusters * nfeatures$ iterations. With one call to `find_nearest_point` for each point per iteration of the main loop this gives us:

$$iterations = n * nclusters * nfeatures = n * 170$$

[1] Actually 12. But we can ignore the store of dist as it will be held in a register.

[2] Treating subtraction as addition.

[3] While there are two subtractions in the code gcc will eliminate one through common subexpression elimination.

Even for our smallest data sample which has 100 points this already gives us 17000 iterations and will dominate the computational demands for any reasonably large size of n.

If we stop here our Arithmethic Intensity is 3 FP operations per 8 byte. Giving an AI of $\frac{0{,}375Ops}{Byte}$ [4]

1.1.2 Main Loop:Updating cluster information.

Listing 2: Updating (partial) cluster information

```
for (j=0; j<nfeatures; j++)
        partial_new_centers[tid][index][j] += feature[i][j];
```

This happens again for each point in the dataset for each iteration of the Main loop. This gives us for iterations:

$$iterations = n * nfeatures = 34$$

While this is not as big an contributor as the loop finding the nearest cluster it still contributes a noteworthy amount of computational demands.

In particular for each iteration here we have two loads (8 Byte) and one addition. [5]

1.1.3 Main Loop:Reduction loop one

Listing 3: Reduction pt1

```
/* let the main thread perform the array reduction */
for (i=0; i<nclusters; i++) {
        for (j=0; j<nthreads; j++) {
                new_centers_len[i] += partial_new_centers_len[j][i];
                partial_new_centers_len[j][i] = 0.0;
                for (k=0; k<nfeatures; k++) {
                        new_centers[i][k] += partial_new_centers[j][i][k];
                        partial_new_centers[j][i][k] = 0.0;
                }
        }
}
```

The amount of iterations is static in regards to the number of items we work on. It is given by $iterations = clusters * threads * features$. In our case when using all cares we get $iterations = 5 * 80 * 34 = 13600$. Which however is neglible for nontrivial datasets so we don't include it in our arithmetic intensity computation.

1.1.4 Main Loop:Reduction loop two

[4] Although in practice caches will usually hold both pt1 and the clusters

[5] Although it is fair to assume that partial_new_centers[tid][index][j] will be cached.

Listing 4: Reduction pt2

```
/* replace old cluster centers with new_centers */
for (i=0; i<nclusters; i++) {
        for (j=0; j<nfeatures; j++) {
                if (new_centers_len[i] > 0)
                        clusters[i][j] = new_centers[i][j] / new_centers_len[i];
                new_centers[i][j] = 0.0;    /* set back to 0 */
        }
        new_centers_len[i] = 0;    /* set back to 0 */
}
```

With at most $clusters * features = 5 * 34 = 170 = iterations$ this loop has no significant impact on overalls performance. For this reason we will ignore it.

1.2 kmeans: Roofline Model

Table 1 gives us the AI for the inner main loop based on it's two essential parts according to the roofline model.

It is however important to point out this does assume NO caching. In practice most of the memory pressure will be absorbed by the cache.

According to the roofline model this gives us a demand of

$$\frac{1280GFlops}{0,31Flop/Byte} = 4129Gb/s$$

on the memory system for 80 Threads.

Table 1: Arithmethic Intensity according to the Roofline Model

	Find nearest Cluster	Update cluster information	Total
Iterations	$n * 170$	$n * 34$	-
Memory (Byte)	$8 * 170$	$12 * 34$	1768
Operations (Flop)	$3 * 170$	$1 * 34$	544
Arithmetic Intensity (Flop/Byte)	0,375	0,08	0,31

1.3 kmeans: Maximum Performance

When considering the maximum performance of the given code there are multiple things to consider:

- The kernel does not use SIMD instructions. This immediatly brings down maximum performance by a factor of 4!
- The kernel is not balanced. There is actually only a single relevant multiplication in the inner loop! Which can happen in parallel to the additions.

This means performance is limited by the addition ALU primarily.

In `find_nearest_point` we have two additions and one multiplication per result. This mean on average we will execute an addition every cycle and a

multiplication every second. So we execute 1,5 Instructions per cycle instead of the maximum of 2 in this kernel because of the mul/add imbalance.

We could additionally take into account the smaller loop Listing 2 which is limited purely by addition. However this would bring down flops/cycle only slightly to 1,41 so for simplicity we will use 1.5 for further analysis.

Maximum FP performance under these considerations is:

$$1280GFlops * 0.25(\text{no SIMD}) * 0.75(\text{imbalance}) = 240GFlops$$

.

Which gives us for the demand on the memory subsystem for 80 Threads:

$$\frac{240GFlops}{0,31Flop/Byte} = 774GB/s$$

And 10 Threads:

$$\frac{30GFlops}{0,31Flop/Byte} = 96GB/s$$

However since many of the inputs to the kernel can be cached this is not representative of the actualy demands on the memory system as our measurements show.