18645 How To Write Fast Code Project - K-Means and Semi-Global Matching

Soham Kelkar (AndrewID: sohamsak)
Bhargav Ghanekar (Andrew ID: bghaneka)
Nishant Gurunath (Andrew ID: ngurunat)

December 2019

Abstract

Autonomous driving systems are generating interest these days, and there are a number of challenges faced in the domain, including computational problems. In this project, we focus on two aspects that can be useful in the field - Scene segmentaion and Depth estimation from stereo cameras. We focus on the classic K-Means algorithm for clustering and segmentation, and on the Semi-Global Matching algorithm for stereo correspondence matching, and attempt to design fast kernels for the same.

1 K-Means

1.1 Algorithm

K Means is a clustering algorithm which is an unsupervised machine learning algorithm. It consists of two major steps -

- Cluster Assignment: Involves distance computation between each point and each cluster and each point is assigned to the closest cluster according to this distance.
- Mean Computation: The mean is computed for each cluster which is assigned as the cluster center for the next iteration

The bottleneck of the algorithm is distance computation. Consider there are N input vectors of dimension d. There are K final clusters. The time complexity of the naive algorithm is O(NKd).

The step-by-step algorithm with their complexities is:

1. Initialize k cluster centers - O(1)
2. Calculate distance of each point from all clusters - O(dNk)
3. Assign each point to the cluster with minimum distance - O(Nk)
4. Update the cluster center to the mean of the points of its cluster - O(N + k)
5. Repeat steps 2 to 4 until Max Iterations - O(MAX_ITERS)

Total number of Instructions = (d x N x k) + (N x k) + (N + k)) x MAX_ITERS

1.2 Data Specifications

The distance computation formula is

$$\sum_{i=0}^{d}(X_i^n - Y_i^k)$$

X^n is the n^{th} input data point and Y^k is the k^{th} cluster. This is a Subtract followed by a FMA. This computation is dependent across dimensions and independent for each point and cluster. Thus the algorithm can be parallelized for distance computation for each point with each cluster.

1.3 Architecture Specifications

- Hostname: ece031.ece.local.cmu.edu
- CPU Model and Manufacturer:
  - Vendor Id : GenuineIntel
  - CPU family: 6
  - model : 79
  - model name: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.4GHz
- Base and Maximum Frequencies: 2.4 GHz, 3.3 GHz
- Number of Processors: 56
- Number of Cores: 14
- Number of threads: 28
- Number of Caches: 1 (35 MB Intel® Smart Cache) (3 Levels)
- Cache Size:
  - Level 1 cache size : 14 x 32 KB 8-way set associative instruction caches, 14 x 32 KB 8-way set associative data caches
  - Level 2 cache size : 14 x 256 KB 8-way set associative caches
  - Level 3 cache size : 35 MB 20-way set associative shared cache
  - Cache Size: 35 MB
- Ref: Intel, CPU World

1.4 Theoretical Peak

The computational bottleneck for this algorithm comes from the presence of only two SIMD FMA functional units on an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz. Hence, the maximum performance that can be achieved is 2(functional units) x 4 (SIMD) x 2 (FMA) FLOPS/cycle. The peak theoretical performance is 16 FLOPS/cycle. However, as we discuss later, our performance is not compute bound but latency bound.

1.5 Initial Kernel Design

- We are using an Intel(R) Xeon(R) CPU E5-2680 v4 where have 2 SIMD Functional units having a latency of 5. We thus need 10 independent operations to fill the pipeline. Since we are performing operations for each point and cluster in parallel so $NK \geq 40$. For $K = 3$, $N \geq 14$
- We have to perform a subtract followed by a FMA which are dependent. So we decided to perform all the subtracts first then followed by all FMA.

- Since the operation across the data point/cluster dimensions are dependent as we need to accumulate the distance for each dimension. Our most outer loop iterates over these dimensions and the two inner (iteration independent) loops iterate over cluster centers and data points.
- We have 16 SIMD registers and since we have floating point input, we can fit 4 floating points in one register. We have to store the input data, cluster centers and the output of Subtract and FMA operations. Our clusters are fixed to be 3. So

$$\begin{aligned} \frac{NK}{4} + \frac{NK}{4} + \frac{N}{4} + \frac{K}{4} &\leq 16 \\ \therefore 2NK + N + K &\leq 64 \\ \therefore 6N + N + 3 &\leq 64 \\ \therefore 7N &\leq 61 \\ \therefore N &\leq 8 \end{aligned}$$

- As we can observe, the peak performance of the kernel is limited by the availability of the number of registers.

So our initial kernel design is register bound and consists of 8 input datapoints and all 3 clusters. Thus creating a 8 x 3 sub-matrix. With the baseline implementation with limited registers we only get 6 independent chains instead of 10 (as required for the peak performance). We utilize the registers for each iteration over dimension in the following way:

- 8 Data Points: 2 SIMD Registers
- 3 Cluster Centers: 1 SIMD Register
- Broadcast Register of cluster center: 1 SIMD Register
- Subtraction Result: N*K/4 = 6 SIMD Registers
- Distance Result (Accumulates): N*K/4 = 6 SIMD Registers
- Total = 16 SIMD Registers

1.6 Designing Better Kernels

To design faster kernels than the one described above we can play with following parameters:

- Increasing kernel size to have more data points/ clusters so that we have more independent operations while using Load-Store to utilize the registers (Data points, Cluster Centers, Subtraction, Distance)
- Get rid of the repeated Broadcast operation over cluster centers instead load them into separate registers and do all the broadcast only at the beginning. However, this requires 2 additional registers than available, therefore incurs some Load-Store operations.
- Reduce the size of the kernel further to get rid of the repeated broadcast operations altogether.

1.7 Results

Here are the results that we obtain after optimizing between kernel size, broadcast operations and load-store operations.

Method	Flops	% of Peak
Baseline	8.43	52.7
Initial Kernel (1.5)	9.19	57.4
Load & Store for SUB and Output Registers (1.6.1)	9.03	56.4
Smaller Kernel (1.6.3)	8.32	52
Load & Store for BCAST Registers (1.6.2)	9.6	60

Table 1: Results using different Techniques

2 Semi-Global Matching

2.1 Stereo correspondence problem and disparity estimation

The problem of estimating depth of a scene from a pair of images taken from slightly different views is of integral interest in the sub-field of stereo vision. On high level, given two images I_l and I_r, to extract depth,

- Find correspondence between pixels of I_l and I_r (which pixel maps to which pixel)
- Compute the difference between the coordinates of two corresponding pixels (called disparity)
- Estimate depth from the disparity map (Depth has a reciprocal relation with disparity values)

Thus, in essence, given two images I_l, I_r, the goal is to estimate disparity map $d(i, j)$, which states that pixel (i, j) from I_l corresponds to pixel $(i - d(i, j), j)$ from I_r. Fig. 1 illustrates the same.

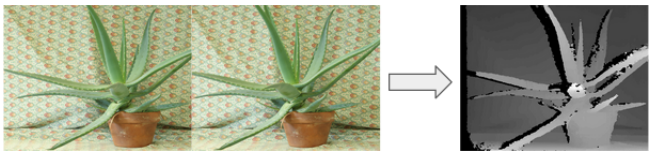

Figure 1: Disparity map from a stereo image pair

2.2 Overview of SGM algorithm

The SGM algorithm [6] is an algorithm used for finding correspondences between two stereo image pairs. Given a point **p1** in the left (base) image, the goal is to estimate disparity d, which is nothing but

$$d = |p1 - p2|$$

, where **p2** is the corresponding point in the right (match) image.

We use the baseline code from [4], which uses CUDA programming to execute the program on a GPU. The pseudo-code for the SGM algorithm is -

- Load images, and resize images s.t. their widths are equal to the specified width. Then the images are converted to grayscale.
- For every pixel **p** in the base image, and for every candidate disparity value between 0,1,..d_{max}, compute the cost C(**p**,d) which is given as

$$C(p, d) = ||I_L(p_x, p_y) - I_R(p_x - d, p_y)||$$

- For 8 directions (corresponding to top, left, right, bottom, bottom-left, etc.), solve the following dynamic programming problem:

  $L_r(p, d) = C(p, d) + min(L_r(p-1, d), L_r(p-1, d-1) + P_1, L_r(p-1, d+1) + P_1, min_i(L_r(p-1, i) + P_2))$

  where r corresponds to one of the 8 directions.

- Perform a weighted sum of $L_r(p,d)$ for the entire image, to get a disparity estimate.
- Solve the same Dynamic Programming problem after interchanging base and match images to get another disparity estimate
- Use the two disparity estimates to find out occlusions (Parts of the scene that are not visible in both images simultaneously). If the two disparity estimates of a pixel differ by an amount greater than the tolerance, the disparity is set to zero (the pixel is considered occluded)
- Median filtering is performed at the end to smoothen out the disparity image

2.3 Architecture specifications

We run the algorithm on the local CMU ECE Cluster, in which each machine has an NVIDIA Quadro P2000 GPU. Some of the salient features [4] of this GPU are -

- 1024 CUDA Cores
- 8 SMs (Streaming Multi-processors)
- 4 (32-wide)warps per SM
- L1 Cache = 48KB per SM
- L2 Cache = 1280KB
- Clock frequency = 1076MHz (base), 1480MHz (boost)

2.4 Theoretical analysis of code

In essence, the following 4 equations are the main computation terms for the SGM algorithm

$$
\begin{aligned}
C(p_x,p_y,d) &= |I_b(p_x,p_y) - I_m(p-x-d,p_y)| \\
E(p_x,p_y,d) &= C(p_x,p_y,d) + \\
&\qquad \min\left(E(p_x-1,p_y,d), E(p_x-1,p_y,d\pm 1)+P_1, \min_i\left(E(p_x-1,p_y,i)+P_2\right)\right) \\
S(p_x,p_y,d) &= \sum_{r=1}^{8} w_r E_r(p_x,p_y,d) \\
D(p_x,p_y) &= \text{argmin}_d S(p_x,p_y,d)
\end{aligned}
$$

Let an image be of size HxW. The first line constitutes $2HW$ fl-ops. The second line constitutes $16HWD$ fl-ops (8 directions, 2D fl-ops per pixel). The third line is an FMA operation, taking $16HWD$ fl-ops. The fourth line is a comparison operation, taking HWD fl-ops. All these computations are done twice (exhanging base and match images). Hence, the number of floating point operations required is

$$\sim 70HWD$$

where (H,W) is the size of the images, and D is the number of disparity values

The theoretical peak in terms of GFLOPS (Giga-floating operations per second) for float data type for this GPU is 47.36 GFLOPS [4]. We compare the results of our kernel to this value.

2.5 Baseline code details and benchmarking

We obtained the baseline code from [4]. The way the code is run on the GPU is as follows -

- Each direction is put on a different thread block (grid). This makes sense, since the code for each direction is slightly different.
- Within a thread block, each thread corresponds to a certain disparity value. The energy matrix for that disparity value is computed by using 3 for loops (2 loops for looping over space, one for performing comparison operations). Note that the threads need to be synced up in the outermost for loop.

2.6 Designing better kernels

Since each thread runs for loops that have independent iterations, atleast within the inner 2 for loops, our kernel design involved creating more threads, so as to run parts of the for loop on different threads. For example with a 2x interleaving, for each disparity value, there are now 2 threads, one running on odd indices, the other on even indices. Likewise, this interleaving was increased to use up all the functional units (baseline code was using only half of the functional units present on 1 SM), and also to fill up the pipeline. This was done for optimizing the second equation shown in the Theoretical analysis section.

Furthermore, the third and fourth equations are also embarassingly parallel for different pixel and disparity values. These were put on different threads to run in parallel and to fill up the FMA pipeline. The baseline version had created 32x32 threads. But since FMA takes 6 cycles, and there were 1024 cores, what was needed were more threads. Running on 64x64 threads was better, and gave incremental benefits.

2.7 Results

We ran the codes on 13 different images taken from [5], and averaged the results. We used `rdtsc` to compute the number of cycles taken, converted that to time taken, and compared with the time taken in case it was the theoretical peak performance.

2.7.1 Interleaving parallel threads

Interleaving factor	GFLOPS (Max. 47)	% of theoretical peak
1x (baseline)	0.40	0.85
2	0.73	1.55
4	1.26	2.69
8	2.03	4.32
16	2.81	5.99
17	5.55	11.80
18	5.52	11.74

Table 2: Results of interleaving threads

2.7.2 FMA pipeline filling

	GFLOPS (Max. 47)	% of theoretical peak
No change to baseline (32x32 threads)	5.55	11.80
64x64 threads	13.91	29.59

Table 3: Results of FMA pipeline filling

3 Conclusions and Discussion

K-Means is a latency bound algorithm and does we cannot fill the pipeline without interleaving load and stores between independent operations. Thus we cannot achieve theoretical peak but we can get close to it by balancing the load-stores and the FMA operations. Also, after understanding the data dependency of the algorithm, we realized that the cluster registers are not longer required after their iteration and hence can be reused. The baseline code of Stero Matching algorithm had a clear struture in terms of hardware. For example computation of each direction was on one block and computation of one disparity value is on one thread. Thus, it becomes important to know the in-depth hardware specs of the GPU which includes the number of blocks, hardware threads, Warps, SMs etc. We also realized that adding threads more than the maximum warp capacity leads to better performance as it enables filling the pipeline. We thus could achieve 32x better performance than the baseline. The link to the codes can be found at Github.

References

[1] Jianbo Shi and J. Malik, "Normalized cuts and image segmentation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 8, pp. 888-905, Aug. 2000

[2] Stella X. Yu, Jianbo Shi, Multiclass spectral clustering, 2003

[3] "Quadro-Powered All-In-One Workstations" (PDF). Nvidia. Archived (PDF) from the original on 2013-06-24. Retrieved 2015-12-11.

[4] https://github.com/rmahieu/SemiGlobalMatching

[5] http://vision.middlebury.edu/stereo/

[6] Hirschmuller, Heiko. "Stereo processing by semiglobal matching and mutual information." IEEE Transactions on pattern analysis and machine intelligence 30.2 (2007): 328-341.