Optimization of Dense Stereo Depth Map Calculation

Nicholas and Patrick Geneva

1 CPU Code-base Overview

When looking at the current state-of-the-art implementations of stereo depth-map calculations one of the code packages that shows up is the Library for Efficient Large-scale Stereo Matching (libelas). As a group, we chose this library for both its opensource nature, but also its source code readability. The codebase[1] is based on the work derived in Geiger et al. [1] and boasts fast matching for high resolution images. The team first looked through the original codebase, and strived to understand all parts before transitioning to GPU CUDA implementation.

The main point of this library is its use first computing a set of sparse "support points" that are then used to created a 2D mesh via Dalaunay triangulation [3]. This information is then use to form a maximum a-posteriori (MAP) estimation to compute the final disparities of all points. The disparity of a given pixel is defined as the spacial difference of where that pixel lies between the left and right image. Thus this is a problem of finding points of high interests and finding the respective match in the other image. After this the left and right disparity maps are compared to check for constancy, and run through some post-processing filters to smooth the final output. Ultimately, from the disparity maps the respective depth map can be obtained given the cameras focal length and spacing. However the difficulty lies in the disparity calculation which is what this project is focused on.

2 CPU Benchmark Results

We used the internal timing functions included in the libelas library. This allows for the timing of specific group segments of the code. The timing was preformed on the default library downloaded from the official website. We choose to look at a specific image pair when comparing the GPU performance to the CPU implementation. The timing results for the pure CPU method can be seen below.

[1] All code can be found on our github which is opensourced for others to use. https://github.com/goldbattle/libelas-gpu

Table 1: Time measurement for a calculation of the left and right disparity maps for the urban4_left.pgm and urban4_right.pgm stereo images.

Process Name	**Time (ms)**
Descriptor	12.9
Support Matches	47.4
Delaunay Triangulation	5.5
Disparity Planes	6.4
Grid	5.9
Matching	314.5
L/R Consistency Check	38.5
Remove Small Segments	54.3
Gap Interpolation	10.4
Adaptive Mean	133.9
Total Time	**630.2**

It can be seen that the key areas that the team is going to focus on is the "Support points", "Matching", and "Adaptive Mean". These areas where chosen because of their high computation times relative to the total runtime. The support matches calculate all the support points by using a Sobel filter and finding areas of high interest. The matching section is the actual disparity map calculation after all triangles have been calculated. Finally, the adaptive mean provided an overall smoothing to the entire image, by using bilateral filtering methods.

3 GPU Code-base Overview

For the GPU implementation we have simply extended the C++ class and overloaded the super class' functions. This allows us to use the original implementation while adding specific CUDA implementations to the methods that need it. The team looked at the three methods and first formulated the best way to parallelize each one.

3.1 Support Matches

Support points is one of the first bottleneck of the system. When a set of raw images are revived, they have filter applied (sobel3x3) and this result stored as the "descriptor" of the image. From this key points with high information are found by first summing the pixel value in the image, and ensuring it is above a threshold. If there is enough information, then the disparity of this point is calculated by matching first left to right, and then right to left.

```
1  // For all point candidates in image 1 do
2  for (int32_t u_can=1; u_can<D_can_width; u_can++) {
3    u = u_can*D_candidate_stepsize;
4    for (int32_t v_can=1; v_can<D_can_height; v_can++) {
5      v = v_can*D_candidate_stepsize;
```

```
    // Initialize disparity candidate to invalid
    *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width)) = -1;
    // Find forwards (left to right)
    d = computeMatchingDisparity(u,v,I1_desc,I2_desc,false); //Left image
    // If we have found a disparity in the left, check from the right to left disparity
    if (d>=0) {
      // Find backwards (right to left)
      d2 = computeMatchingDisparity(u-d,v,I1_desc,I2_desc,true);
      // Check our error between the 2 disparity and that its below 2 pixel difference
      if (d2>=0 && abs(d-d2)<=param.lr_threshold) {
        // Save disparity
        *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width)) = d;
      }
  }
}
```

Inside of the *computeMatchingDisparity(...)* function, given a u and v value, and the image descriptors it first calculates the total amount of "information" around the current pixel. This is then checked against a threshold, which ensures that areas of only high interest are used. We will see that this calculation is important on, as this is what brings the sparsity to the problem. If this check is unsuccessful, a invalid valid of -1 is returned, but otherwise a the disparity energy minimization problem is then calculated. This is a loop that runs through the range of valid disparities for this pixel. This loop, runs at least 10 times, for each pixel calculates the difference between the left to right pixel values (right to left if preformed the other way). We define it as follows:

$$E(d) = \beta \parallel \mathbf{f}^{(l)} - \mathbf{f}^{(r)}(d) \parallel_1 \tag{1}$$

where the $E(d)$ is the energy of the difference between the left and right pixels. The full derivation is provided in Geiger et al. [1]. This calculation can become expensive if a lot of pixels are being used as support points. We found that this was normally not the case (less than 200 total for an image) and instead it was the sum calculation that caused a lot of computation time.

Figure 1: Example sobel filter, which is then used to calculate if each pixel has enough information to use as a support point.

When first implementing this on the GPU we first try a "dense" approach where for each pixel it was check if it had enough information and then both its left to right and right to left disparities were calculated. This prove to almost double the total computation time. This is both likely due to how sparse the support points are, and the cost of copying the data to the GPU (which averaged around 20ms). From there, we used an approach similar to the "Matching" section, which worked well in sparse methods. First we check if the support points had enough information, and then sent this smaller subset of the image coordinates to the GPU. This proved to reduce the computation time to around 60ms total, but still did not prove to have an improved performance. This was not taken any further as we focused on improving other areas where the GPU could more benefit the computation speed.

3.2 Matching

The matching method *computeDisparity(...)* takes in the set of support points, triangles, and two arrays that contain the image descriptor information. A prior disparity grid is calculate for the image, and then for each triangle the disparity is calculate for each pixel in the individual triangle. The disparity calculate for each pixel, *computeMatchingDisparity()*, is independent of the rest once the prior disparity map has been calculated.

The main challenge with this function is converting the CPU SSE instruction into normal functions, while also eliminating logic branches. The *computeMatchingDisparity()*, which computes the disparity of a single pixel, has a minimization problem inside of it which allows it to find the best matching disparity.

```
__m128i xmm1 = _mm_load_si128((__m128i*)I1_block_addr);
__m128i xmm2 = _mm_load_si128((__m128i*)(I2_line_addr+16*u_warp));
xmm2 = _mm_sad_epu8(xmm1,xmm2);
val = _mm_extract_epi16(xmm2,0)+_mm_extract_epi16(xmm2,4)+w;
```

This code converted to the following GPU code seen below. We found that using the native CUDA sad had no decrease in performance, when compared to the normal abs and manual subtraction.

```
int32_t val = 0;
for(int j=0; j<16; j++){
  //val += abs((int32_t)(*(I1_block_addr+j))
  //          -(int32_t)(*(I2_line_addr+j+16*u_warp)));
  val = __sad((int)(*(I1_block_addr+j)),(int)(*(I2_line_addr+j+u_warp)),val);
}
val += valid?*(P+abs(d_curr-d_plane)):0;
```

When creating this kernel we launch it as a 1d stored array. Because the data is so sparse, we pre-compute all the needed data for each pixel and then batch process all pixels. Originally we tried creating kernel streams for each triangle, but it was found that only around 30-100 threads would be created in each kernel which caused a lot of unneeded overhead. Batch processing all the needed data at once proved to be far better in performance.

3.3 Adaptive Mean

To refine the disparity map that is calculated previously, an adaptive mean is applied to the image. The specific algorithm used is a bilateral filter that is designed to smooth the image of noise. In essence this algorithm used a predefined window around a given pixel average out the pixel's intensity compared to its neighbors, namely,

$$I^{filter}(x) = \frac{1}{W_p} \sum_{x_i \epsilon \atop \Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|) \tag{2}$$

$$W_p = \sum_{x_i \epsilon \atop \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|) \tag{3}$$

in which I^{filter} is the post filtered pixel intensity, I is the input intensity from the original image, x is a spacial coordinate, Ω is the pixel window, f_r is the range kernel and g_r is the spacial kernel. An example of bilateral filtering can be seen below in Figure 2 courtesy of [4]. Here we can see that the filtered image has significantly less noise. Since each pixel is processed individually, this method is ideal for GPUs.

Figure 2: An example of bilateral filtering with the unprocessed image on the left and the filtered image on the right [4].

In the CPU implementation, a horizontal and then vertical bilateral filter is completed in the *adaptiveMean(...)* method with a window size of 8 pixels. In the original program, similar to the matching method, SSE instructions were used. For example, for the horizontal filter for a given pixel, 7 neighboring pixels in the same row to average. This results in a set of 8 floating point numbers that are processed in two vector sets, as seen below. This is repeated for every pixel for both a horizontal and vertical bilateral filters.

```
//Process first 4 pixels in average
xval     = _mm_load_ps(val);
xweight1 = _mm_sub_ps(xval,_mm_set1_ps(val_curr));
xweight1 = _mm_max_ps(_mm_sub_ps(_mm_setzero_ps(), xweight1), xweight1);
xweight1 = _mm_sub_ps(xconst4,xweight1);
xweight1 = _mm_max_ps(xconst0,xweight1);
xfactor1 = _mm_mul_ps(xval,xweight1);
//Process next 4 pixels in average
xval     = _mm_load_ps(val+4);
xweight2 = _mm_sub_ps(xval,_mm_set1_ps(val_curr));
xweight2 = _mm_max_ps(_mm_sub_ps(_mm_setzero_ps(), xweight2), xweight2);
xweight2 = _mm_sub_ps(xconst4,xweight2);
xweight2 = _mm_max_ps(xconst0,xweight2);
xfactor2 = _mm_mul_ps(xval,xweight2);
```

To convert this to a GPU kernel the SSE vector functions were first converted into standard arithmetic seen below.

```
for(int32_t i=0; i < 8; i++){
    weight_sum0 = 4.0f - fabs(D_shared[ut][vt+(i-4)]-val_curr);
    weight_sum0 = max(0.0f, weight_sum0);
    weight_sum += weight_sum0;
    factor_sum += D_shared[ut][vt+(i-4)]*weight_sum0;
}
```

Given that each pixel in the image needs to be process, each thread processes a single pixel. A thread block consists of a total of 8×8 threads, resulting a small block of 64 pixels being processed in a single block. In the original GPU implementation, just global memory was used to store all memory which resulted in a wall-clock time of 7ms. This is clearly already a significant reduction in computation time compared to the original implementation wall-clock time of 133.9 ms as seen in Table 1.

However since the kernel is constructed to process a small patch of pixels shared memory can be used to boost the programs performance. For each thread block a shared memory array is initialized that is $(8+7) \times (8+7)$ pixels to account for the extra neighboring pixels needed to average. Each thread loads its respective pixel and edge pixels load the outer margin. The use of shared memory shaved off approximately 3ms from previous kernel implementation reducing the wall-clock time to approximately 4ms. A sample of the shared memory initialization and copying can be seen below. Although this method require all threads to sync twice during the kernel, the significant reduction of calls to global memory during the averaging process is much more beneficial.

```
//Allocate Shared memory array with an appropiate margin for the bitlateral filter
//Since we are using 8 pixels with the center pixel being 5,
//we need 4 extra on left and top and 3 extra on right and bottom
__shared__ float D_shared[8+7][8+7];
//Populate shared memory
```

```
if(threadIdx.x == blockDim.x-1){
  D_shared[ut+1][vt] = D[idx+1];
  D_shared[ut+2][vt] = D[idx+2];
  D_shared[ut+3][vt] = D[idx+3];
}
if(threadIdx.x == 0){ ... }
if(threadIdx.y == 0){ ... }
if(threadIdx.y == blockDim.y-1){ ... }

if(D[idx] < 0){
  // zero input disparity maps to -10 (this makes the bilateral
  // weights of all valid disparities to 0 in this region)
  D_shared[ut][vt] = -10;
}else{
  D_shared[ut][vt] = D[idx];
}
__syncthreads();
```

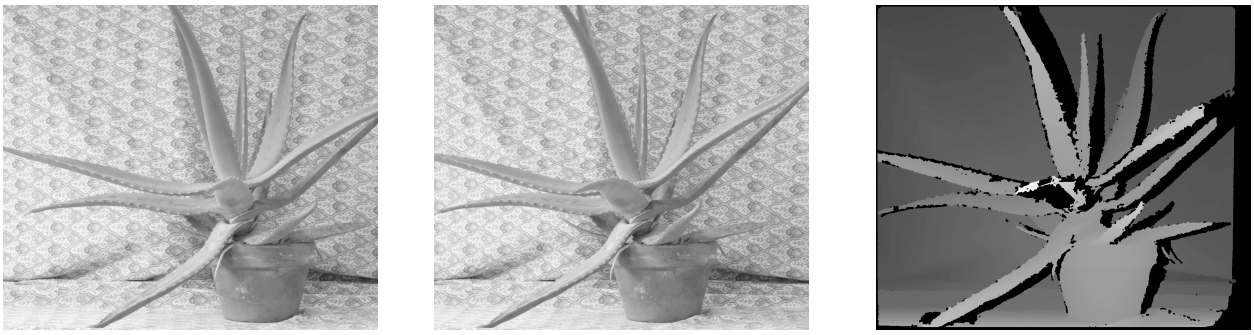
4 Results and Validation

4.1 Results

Below are several example stereo-camera inputs followed by the computed right disparity map calculated by the GPU implementation of the program. All stereo camera data was provided by the Middlebury Stereo Data set [2].

(a) Left stereo input (b) Right stereo input (c) Right disparity map

Figure 3: Cones data set published in 2003.



(a) Left stereo input (b) Right stereo input (c) Right disparity map

Figure 4: Aloe plant data set published in 2006.

(a) Left stereo input (b) Right stereo input (c) Right disparity map

Figure 5: Urban 3 data set published in 2014.

(a) Left stereo input (b) Right stereo input (c) Right disparity map

Figure 6: Urban 4 data set published in 2014.

4.2 MSE Comparison CPU to GPU

To validate the correctness of our program, we wish to simply maintain the same accuracy as the original CPU implementation. To do so, we calculated a simple MSE of the difference between the two corresponding depth maps.

```
// Variable for the total error
double sse = 0;
// Compute the mean squared error between the two images
// http://stackoverflow.com/a/17237076
for(int32_t i=0; i<width; i++) {
  for(int32_t j=0; j<height; j++) {
    sse += pow(imRef(I1, i, j) - imRef(I2, i, j),2);
  }
}
// MSE = sum((frame1-frame2)^2 ) / no. of pixels
double mse = sse / (double)(width*height);
```

Table 2: Calculated MSE values for left and right images comparing the CPU implementation to the GPU implementation.

Image Title	MSE Value
cones_left_disp	0.97456
cones_right_disp	1.01811
urban1_left_disp	1.50266
urban1_right_disp	0.33784
urban4_left_disp	0.48262
urban4_right_disp	0.35474

We found that there was almost no difference between the two implementations. We attribute this to the fact that we are mostly working in the integer numerical range, and thus have minimal issues with floating point algebra. It can be seen in the above table that at max there was around 1.5 pixels difference between the two implementations, which far surpasses the acceptable threshold.

5 GPU Benchmark Results

Table 3: Time measurement for a calculation of the left and right disparity maps for the urban4_left.pgm and urban4_right.pgm stereo images.

Process Name	Time (ms)	Decreased (%)
Descriptor	11.4	11.6
Support Matches	27.7	41.6
Delaunay Triangulation	3.1	43.6
Disparity Planes	4.0	37.5
Grid	2.1	64.4
Matching	**51.4**	83.7
L/R Consistency Check	6.4	83.4
Remove Small Segments	19.8	63.5
Gap Interpolation	4.0	61.5
Adaptive Mean	**3.7**	97.2
Total Time	**133.3**	**78.8**

The final GPU performance used the exact same profile timing method the CPU method used. The results are shown in the table in which running time is shown along with the percent decrease relative to the CPU time. It can be seen that in the methods that where not changed to a GPU implementation also have a decrease in execution time. The team thinks that this is due to improved caching abilities on the CPU due to less data being processed on the CPU side. This can be seen that after methods implemented with the GPU the CPU methods have a larger decrease in run time.

Overall we are very satisfied with the current performance of the GPU accelerated program as the two GPU optimizations implemented resulted in a 78.8% decrease in wall-clock

time for the dense calculation. Although this resulted in a wall-clock time of 133ms, just 33ms of the original goal of 100ms, its believed that further optimization can be achieved on again the matching section of the program along with the support point calculation. With a proper implementation, real time computing speeds should be achievable for the disparity calculation without sub-sampling.

References

[1] A. Geiger, M. Roser, and R. Urtasun. Efficient large-scale stereo matching. In *Asian Conference on Computer Vision (ACCV)*, 2010.

[2] H. Hirschmuller and D. Scharstein. Evaluation of cost functions for stereo matching. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[3] D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980. ISSN 1573-7640. doi: 10.1007/BF00977785. URL `http://dx.doi.org/10.1007/BF00977785`.

[4] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand. A gentle introduction to bilateral filtering and its applications. In *ACM SIGGRAPH 2007 courses*, page 1. ACM, 2007.