How to write fast code: ORB Algorithm Final Report

Michael Chang, Frank Lee, Minh Truong

December 7, 2019

1 Introduction

1.1 Algorithm

The ORB Algorithm consists of 3 parts: 1) Keypoint Detection, 2) Descriptor Creation, and 3) Feature Matching. We will develop 2 separate kernels for these 3 parts.

1.1.1 Keypoint Detection

The oFAST algorithm compares a pixel (let's call it v) with the 16 pixels that surround it. Pixel v is considered a keypoint if there exists at least 8 consecutive pixels in the circle that are all darker or all lighter than pixel v. To find this contiguous string of pixels, the oFAST kernel two different 32-bit integer masks; first one returns value 1 per pixel if it is brighter than v's brightness plus threshold. Similarly the second mask returns value 1 per pixel if it is darker than v. Afterward, the kernel accumulates the number of 1's in both masks and checks if they are greater than 8 using a hash map. If so, pixel v is considered a keypoint. After identifying the keypoint, the oFAST algorithm needs to calculate the keypoint's intensity centroid angle. This angle is an assigned value of the patch that surround the keypoint, which is necessary to crucial the patch rotation invariant in the subsequent rBRIEF kernel. This is done by calculating `atan2` of the Rosin's x and y moments [1]. As later shown, this angle calculation actually happens during the rBRIEF kernel execution and not oFAST kernel execution.

1.1.2 Descriptor Creation

The rBRIEF algorithm uses oFAST-generated keypoints to perform Gaussian Sampling on a patch of image around each keypoints. Gaussian sampling ensures that the patch is best represented in a binary array form. Using Imagenet Data, a Gaussian sampling pattern is trained. This makes sure that the pattern has a low co-variance and therefore captures more information from the patch. The rBRIEF binary vector output reports a 1 if the 2nd sampling point is brighter than the 1st sampling point, 0 otherwise. The process is repeated n times for a vector length of n. In this implementation, we choose n to be 32, so that 1 binary vector fits exactly into 1 int. As mentioned, the intensity centroid angle is computed in the rBRIEF kernel. With this angle, a rotated version of the sampling points is produced, and subsequently the binary array of the patch is rotation invariant.

1.1.3 Keypoint Matching

The Matching process is absorbed into the rBRIEF kernel because it has a similar thread mapping to that of the rBRIEF process. We will discuss this in more details in section 2. In short, the Matching process takes in test binary vectors produced by rBRIEF and calculate the Hamming distance between these test vectors and the train vectors. The train vectors is a precomputed batch of keypoint descriptor, of which we want to match the test binary vector to. The Matching process outputs a one-to-one mapping between test and train vector based on which test vectors has the lowest Hamming distance to a train vector.

1.2 Targeted Hardware

We choose the Quadro P2000 GPU with the compute capability of 6.1 to develop our oFAST and rBRIEF+Match kernels. The following table is the GPU specification that informs the design of the kernels:

1.3 Pipeline breakdown

As briefly mentioned in Section 1.1.1, the ORB pipeline is broken down into 2 separate kernels: oFAST and rBRIEF+Match. Here, we discuss the specific input, output of each kernels as well as their formats. Section 2 will discuss why such format is chosen. For oFAST, the input is a string of images represented as a 1D array of `uint_8`. The output of oFAST is a `float4`

[1] P. L. Rosin. Measuring corner properties. Computer Vision and Image Understanding, 73(2):291 – 307, 1999. 2

Attribute	Value
Global Memory Bandwidth	140 GB/s
GPU clock	1.37 GHz
Global Memory Size	5.12 GB
Memory Bus	160 bits / 20 bytes
SM Count	8
CUDA cores	1024 (4 warps per SM)
Threads	32 per Warp
Registers	65536 per SM
Shared Memory	49152 per SM
L1 Cache	48 KB per SM
L2 Cache	1280 KB

Table 1: Different Quadro P2000 Specs

array of patches. For each image, 141 patches are created at the output. Each patches has 100 pixels which is represented with 25 `float4`. Figure 1 illustrates how the patches are stored in the oFAST output array. The array stores data for 1

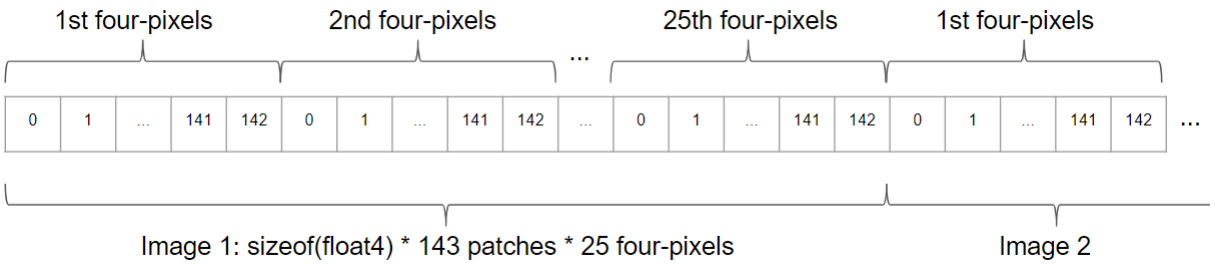


Figure 1: oFAST output Format

image consecutively. Within the subarray of the same image, the first four-pixels of all 141 patches are stored consecutively. A four-pixels is a `float4` object of 4 consecutive pixels within the same patch. Then, the 2^{nd} four-pixels is stored and so on until the 25^{th} four-pixels of all 141 patches. This format is also the rBRIEF kernel input.
On the other hand, the output of the rBRIEF kernel is an `int` array of 128 elements per images. Figure 2 illustrates how the rBRIEF output array should be interpreted. Each element of the rBRIEF output array is mapped to a Train Vector Index.

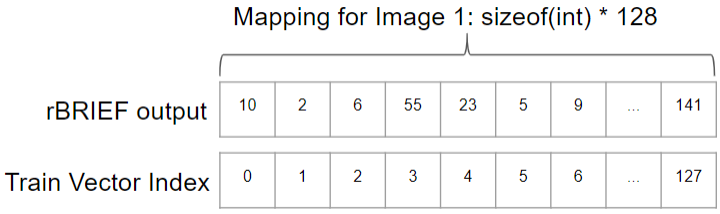


Figure 2: rBRIEF+Match output Format

For example, this means patch 0 of the train vector is mapped to patch 10 of the test vector, 1 maps to 2 , 2 maps to 5 and so on. In reality the Train Vector Index array is not stored explicitly since it is just an array of incremental value from 0 to 127. This rBRIEF output is sufficient to map the same keypoints across 2 different images, as shown in section 5.2

2 Kernel Details

This section contains the psuedo code for the 2 kernels as well as a walkthrough of the code. Any fast code techniques used is also underlined during the walkthrough. A detailed explanation why such technique is used at that specific point is available in section 3.

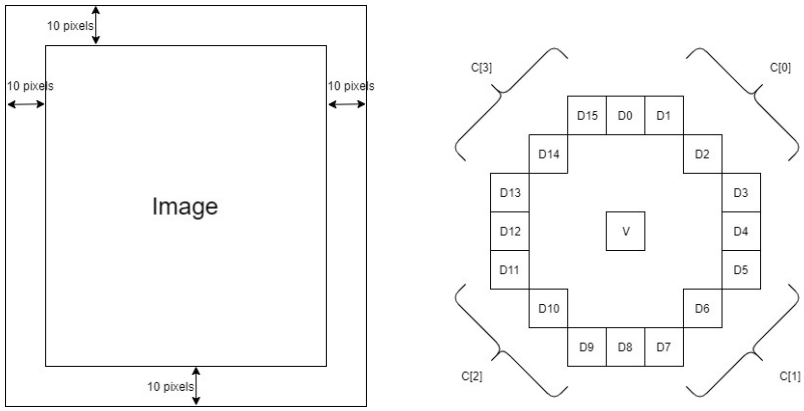
Algorithm 1 The Keypoint Detecting Kernel.

```
1: procedure CALCKEYPOINTS(input, output, threshold)
2:     ▷ Load shared_table as shared memory for all threads to access
3:      extern __shared__ uint8_t shared_table[]
4:     ▷ The (i,j) pair of each thread corresponds to (i + 10,j + 10) coordinates in image
5:      j = threadIdx.x + blockIdx.x * blockDim.x + 10
6:      i = threadIdx.y + blockIdx.y * blockDim.y + 10
7:     ▷ For all the pixels inside the edges - 10
8:      if i and j < rows,cols - 10
9:     ▷ Store the ring values in a local array C[] and v
10:            C[0] = top right coords of the ring
11:            C[1] = bottom right coords of the ring
12:            C[2] = bottom left coords of the ring
13:            C[3] = top left coords of the ring
14:            v = center
15:    ▷ For each pixel, it does 32 comparisons total
16:            for i from 0 to 15
17:    ▷ mask1 only cares about bright values; mask2 only cares about dark values
18:                    if pixel i > v + threshold
19:                            uint16_6 mask1[i] |= 1
20:                    if pixel i > v - threshold
21:                            uint16_6 mask2[i] |= 1
22:                    else
23:                            uint16_6 mask1[i] |= 0
24:                            uint16_6 mask2[i] |= 0
25:            endfor
26:    ▷ Looks up if the mask1 and mask2 combinations exist in shared_table
27:            if (popc(mask1) > 8 and mask1 is in shared_table) or (popc(mask2) > 8 and mask2 is in shared_table)
28:    ▷ ind only increments if the pixel is a keypoint
29:                    ind = atomicInc
30:                    for b from 0 to 99
31:    ▷ Stores 10 by 10 patch centered around the pixel
32:                            output[ind*100+b] = input[cols*(i+4-(b/10))+(j+(-4+(b%10)))]
33:                    endfor
34:            endif
35:     endif
36: end procedure
```

2.1 oFAST

The kernel begins by incorporating shared memory access. The only information that is shared among all the threads is the c_table, which will be used later in the kernel. The number of threads launched is proportional to the number of pixels in an image, except the edges of width 10 pixels. This idea is shown in the Figure 3a. The reason for the threads to be only working on the middle portion is because of the 10 by 10 sized patch around the center pixel that will be used later. If a keypoint is in the corner of the image, we could not have gotten the proper patch as majority of the patch will be outside the image. Another option we could have taken is to use padding. However, we believed that adding more pixels for computation would have taken a bigger toll in timing, and we also believed that the keypoints will tend not to be on the edges or the corners of an image. For each pixel, it creates a ring around the center pixel with the distance 3 pixel. By doing so, we are able to get 16 pixels, which will be stored into an array **C**[ ]. **C**[ ] is an **int** array, meaning it will have 32 bits per element. Because the pixel's value(by value, we mean the brightness of the pixel) is in **uint8_t**, which is only 8 bits, we can store 4 of these values *per* an element of **C**[ ]. How the pixels are stored in the array is shown in the pseudocode. The Figure 3b shows how they are actually allocated. After storing into the array, we must do comparisons of each pixel to the center pixel. If the pixel value is greater than the center pixel plus threshold, it stores bit 1 in **mask1**. Similar to the way **C**[ ] is stored, **mask1** is a **uint16_t**, which can store 16 boolean bits. The same method is done for **mask2** but for pixels darker than center pixel minus threshold. The reason why we chose this method will be discussed later. With the calculated masks, we then use the c_table that was brought into the thread earlier. c_table is list of all the possible 16 bit combination that has eight consecutive 1's. This c_table was provided by opencv's library. To determine if a pixel is a keypoint or not, it first does __popc to make sure there are greater than 8 1's, and uses the table. Finally, if it is a keypoint, we need to output the 10 by 10 patch centered around the keypoint. Because the output is an 1D array, we need to concatenate all the patches back to back. To do this, each keypoint must have its own unique index. We can't use the values **i** and **j** because they are not sequential. We solved this issue by using a CUDA function called **atomicInc**, which awaits all the threads and increment one at a time. By doing so, each keypoint can use this new index **ind** to store into output. This may cause a slowdown as

all threads need to wait for this command. This concludes the keypoint detection kernel.



(a) Typical image (b) Ring around pixel

Figure 3: oFAST figures

2.2 rBRIEF + Match

We start by recognizing that each patch creates 1 binary vector. Moreover, since each patch is 100 floats, while a binary vector is 32 bits (represented by 1 int), this is a reduction operation. This means the transformation from 1 patch to 1 binary vector can be interpreted as a single chain of dependent instruction. Here, we recognize 1 SM as having 32 lanes of pipelines (1 warp). Therefore, we decide to map 1 thread to 1 unique patch. And since there are 4 warps schedule per SM, we can have 1 block processes 128 patches at a time. To ensure uniform access, we will only take 128 keypoints per image, so that 1 image can be processed within a single block and no inter-block communication is needed. This is possible because oFAST kernel output up to 143 keypoints per image, we just need to select the first 128. The reason why oFAST kernel output 143 and not 128 is because the oFAST keypoint threshold. If the threshold is anything lower, we cannot guarantee to get 128 keypoints per image. Although each thread receives a new patch for every new image, it can reuse the Guassian sampling pattern, Therefore, this pattern can be stored in the thread private register. CUDA compute capability 6.1 allows each thread to have up to 256 register, so it is possible to store the entire sampling pattern inside indefinitely. Line 5 to 7 do this. If we decide to have 32 sampling trials, the number of registers needed is 128, since each sampling trial require 2 points, each points has a row and col coordinate. This translate nicely to **int4**, there fore the **pattern** is packed as **int4** in global memory. Line 9 to 11 put the patch in shared memory. We recognize that the data access pattern for each thread to its corresponding patch is not uniform. This is because the access depends on the sampling angle which is data dependent. Having the thread directly accessing the patch from global memory is costly since there is no memory coalescence and the accesses will be serialize. To make matters worse, the patch needs to be accesses twice with non uniform pattern, one to calculate the angle, One to calculate the binary vector. This means there will also be cache conflict in either L1 or L2 after the patch is loaded in from global memory. To combat this, we turn to shared memory. Since there are 32 banks of shared memory, and we know there will only be 32 threads active at any given time, we privatize the banks so that thread 0 has exclusive access to bank 0, and so on. To do this, each thread first call load on the pixel of its patch. The rBRIEF **input** format discussed in section 1.3 allows this load to be coalescence across all warps, since all patches within an image are four-pixel interleaving. For each patch, the thread will skip 31 address every pixel while storing the patch's pixels. This ensure the entire patch for a thread is stored in 1 bank. Then, there will be no bank conflict among active threads and we don't have to pay the play-back cost of non-uniform accessing. Line 8 to 11 illustrates this idea, where the **id** is a unique key that store the patch in a private shared memory. From Table 1 we know that there is 49152 KB of shared memory per SM. With 128 patches per block, this means each patch can have up to 96 pixels, since $96 * 128 * 4 = 49152$. This is possible because oFAST gives us up to 100 keypoints, rBRIEF will just take the first 96. Line 13 to 21 illustrates how the sine and consine of the angle is computed. The **atan2** and **sincos** functions together form the bottleneck of this kernel, since they take the longest amount of time to compute. The nvcc **-use_fast_math** flag already replaces **atan2** and **sincos** with the fastest intrinsic, so no trigonometric manipulation is neccessary to reduce the latency of these operation. We then treat Line 20 and 21 as 1 logical operation. With GPU latency benchmark, this operation takes about 500 ns. This means as soon as a warp hit line 20, there will be a large stall. This automatically trigger a context switching to a new warp. This new warp will execute from line 14 to 20, where it also hit the same stall. The time that this new warp is active is therefore approximately the time it takes to perform the independent multiply-add in line 17 and 18. Knowing that the **fma** instruction takes around 15 ns, and there are only 1 of such function per line, plus there are $96 * 2 = 192$ of such operation, the active time of this new warp should be $192 * 15 = 2880\ ns$. This should be enough to cover the latency of the previous warp bottleneck. The **coordx** and **coordy** are coordinate of the patch required to calculate the moments. Since we know the size of the patch to be 96 (10 by 10 omitting last 4), these coordinates

Algorithm 2 The rBRIEF+Match Kernel.

```
1: procedure RBRIEFMATCH(input, output, pattern, train_vec)
2:     ▷ Perform the following once as setup routine:
3:     id = threadIdx.x
4:     ▷ Load the pattern into register
5:     for i from 0 to 32
6:         reg_pattern[i] = pattern[i]
7:     endfor
8:     ▷ Load patch into thread private bank
9:     for i from 0 to 96
10:        private_patch[id][i] = input[i]
11:    endfor
12:    ▷ Perform the following for each patch assigned to the thread:
13:    ▷ Calculate Intensity Centroid Angle
14:    m01 = 0
15:    m10 = 0
16:    for i from 0 to 96
17:        fma(coordx[i], private_patch[id][i], m01)
18:        fma(coordy[i], private_patch[id][i], m10)
19:    endfor
20:    angle = atan2(m01. m10)
21:    sin, cos = sincos(angle)
22:    ▷ Create binary vector
23:    binVec = 0
24:    for i from 0 to 32
25:        ax = reg_pattern[i].x;
26:        ay = reg_pattern[i].y;
27:        bx = reg_pattern[i].z;
28:        by = reg_pattern[i].w;
29:        Ia = private_patch[id][(cos * ax - sin * ay) + 10 * (sin * ay + cos * ay)]
30:        Ib = private_patch[id][(cos * bx - sin * by) + 10 * (sin * by + cos * by)]
31:        binVec | = (Ia > Ib) << i
32:    endfor
33:    ▷ Match with the train binary vectors
34:    for i from 0 to 128
35:        out = popc(train_vec[i] xor binVec)
36:        out = atomicMin(global_min, out)
37:        if out == global_min
38:            output[i] = id
39:    endfor
40: end procedure
```

can also be precomputed and stored in register, since they are use frequently. In the actual implementation, only 1 set of coordinate needs to be stored since the other set is just a transpose of it. Therefore, each threads need to store additional 96 values in its registers, bringing the total up to $128 + 96 = 224$ registers. Line 24 to 32 describe the creation of the binary vector. At this point, all variables involves are either in private registers or prive shared memory bank, minimizing the cost of data transfer. Line 34 to 38 then describe the Hamming distance calculation. Since private register and shared memory are already used to store the pattern and private patch, we need to store the **train_vec** in L1. This can be achived through preloading the next **train_vec** while Hamming distance is being calculated with the current **train_vec**.

3 Fast Code Techniques

This section highlights the different fast code techniques that were used in the project.
1) **Kernel Fusion**: rBRIEF and Match
2) **Preloading**: training vector
3) **Data Packing**: uses of float4 and int4 for oFAST output, rBRIEF input, sampling pattern
4) **Shared Memory**: private bank for rBRIEF Patch, oFAST hashmap
5) **Loop Unrolling**: during fuse multiply-add for momments calculation, shared arrays load in
6) **Pre-computation**: oFAST hashmap for masks, pixel indexing (e.g. For image[i+3][i+1] and image[i+3][i+2], the value [i+3] can be stored into register and reused)
7) **Data Re-formating**: oFAST output
8) **Memory Coalesce Accessing**: oFAST output, rBRIEF input

10) **Avoiding branch divergence**: Instead of using if/else statements, concatenate comparison bits to create masks that can be checked with pre-computed table

4 Evaluation

4.1 oFAST

From bottleneck latency calculation, the arithmetic throughput of oFAST is:

$$\frac{8 \text{ SM} * 32 \text{ threads/SM} * 100 * 4 \text{ bytes}}{43 * 23 \text{ ns}} \approx 708 \ MB/s$$

For each thread, it does 43 **comparisons**, which cost 23 nanoseconds. There are total of 8 SMs with 32 threads each. This means that at the end of a block of pipeline, 32*8 threads will be completed. Each thread outputs a size-100 patch, with its element being 4 bytes(float). This portion of the code doesn't have any floating point operation, which is why the output is kept at MB/s. Running on 1000 images, the kernel implementation takes around 108 ms. This means that the throughput is:

$$\frac{141*1000 * 100 * 4 \text{ bytes}}{108 * \text{ ms}} \approx 522 \ MB/s$$

Total, 141 keypoints were outputted per image.
Therefore, this portion of the kernel achieved around 73% of the peak throughput. One of the reasons why the performance might be low is because we considered comparison operation as the bottleneck of the kernel. We are certain that it is the slowest operation, but there might be another operation that might have slow latency, which might make the peak equation different.

4.2 rBREIF + Match

From bottleneck latency calculation, the arithmetic throughput of rBRIEF and Match is:

$$\frac{32 \ threads/SM * 4 \ Bytes/Int}{500 \ ns + 128 * 15 \ ns} \approx 53 \ MB/s$$

This is because each thread produces 1 integer after 1 **atan2** + **sincos** and 128 **popc**. Knowing that the Global Memory Bandwidth is 140 GB/s, this algorithm is compute bound. So, peak performance is:

$$\frac{(192 + 2 + 32 + 128) * 32 \ threads/SM}{500 * 10^{-9} + 128 * 15 * 10^{-}9} \approx 4.72 \ GFLOPs$$

Because there is 192 **fma**, 2 trigonometry function, 32 Ia/Ib comparison, and 128 **popc**. Running on 1000 images, the kernel implementation takes around 10 ms.This means for 4 warps per SM, throughput is:

$$\frac{(192 + 2 + 32 + 128) * 32 \ threads/SM * 1000 \ images * 4 \ warps/SM}{10^{-3}} \approx 4.512 \ MFLOPs$$

Therefore, the kernel achieve around 95% that of peak throughput. The blue line in the following figure shows that the average execution time, and subsequently throughput does not change much as the number of workload per block increases. This means the there is little extra cost of data movement, and the latency is determined by the compute capability of the kernel. Interestingly, if we use the **–ftz=true** and **-use_fast_math** flag, the execution time goes down significantly as number of images increase per block. We speculate this is due to the reduction in latency of the bottleneck. This worksout well when we fuse the rBRIEF kernel with Match (yellow line), where there is no significant difference until around 60 images per block. Unfortunately, we cannot have a comparison with the OpenCV GPU ORB baseline due to difficulty of installing it. However, our result definitely meet real-time processing constraints as we can process around 1000 images in a total span of 110 ms.

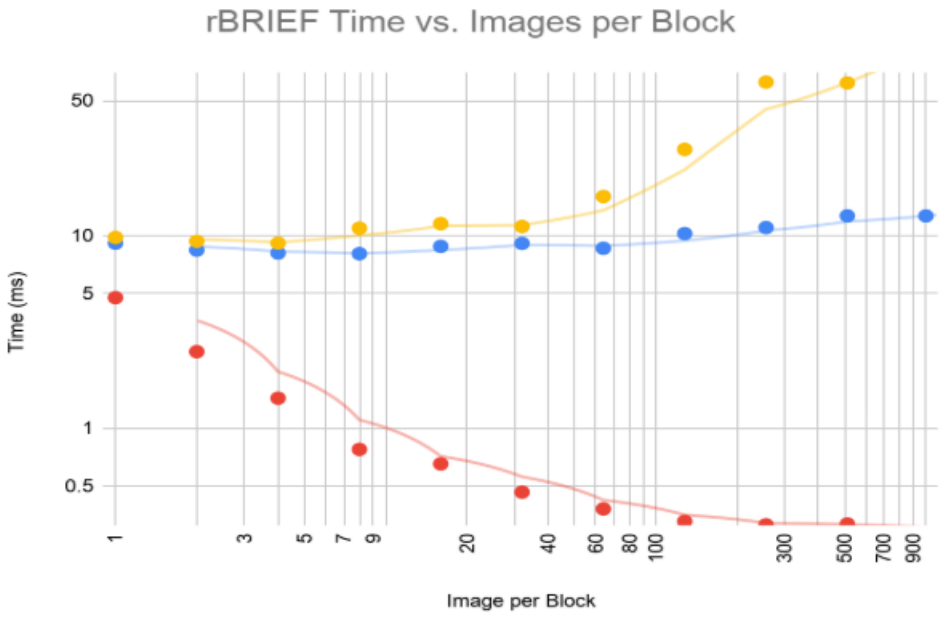


Figure 4: rBRIEF + Match Performance vs. Image per block