Group 6: Video Encoding on Tegra Xavier using the CUDA framework

Skjalg Gustav Eriksen
University of Oslo
skjale@ifi.uio.no

ABSTRACT

In this paper we document our optimization of the Codec63 encoder. We will optimize the encoder with CUDA to off-load Computation to the Graphics Processing Unit (GPU) to accelerate Codec63.

1 INTRODUCTION

This paper is the home exam 2 for group 6 in IN5050 at UiO. The assignment is to use the CUDA framework and off-load Computation to the GPU to optimize Codec63. In this assignment we will use CUDA Toolkit v10 and the GPUs on the Nvidia Jetson AGX Xavier boards.

During the optimization process we will keep the `-pg` cflag on so we can see the changes to the gprof profile and document bottlenecks that may arise. As for the the functions offloaded to the GPU we will use nvprof as they will not show up in gprof.

1.1 Encoder profile

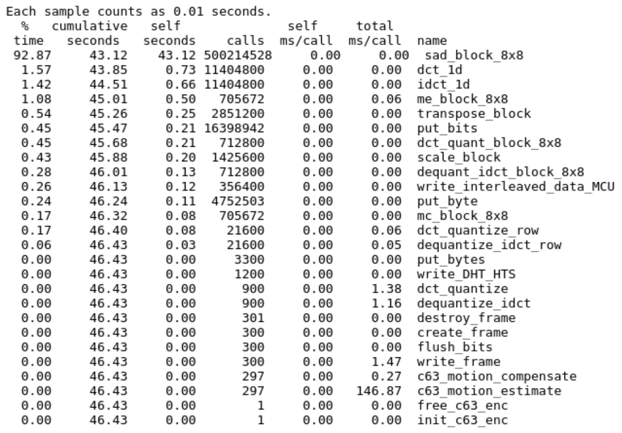
In figure 1 we have the encode profile of the precode. As it was in exam 1 the functions `sad_block_8x8(), me_block_8x8(), dct_1d(), idct_1d(), transpose_block()` are the major bottlenecks in the code.

As the assignment text suggest we should focus our effort on motion estimate which is responsible for `sad_block_8x8()` and `me_block_8x8()`, and also DCT/iDCT that have `dct_1d(), idct_1d()` and `transpose_block()`.

1.2 Sections

The remainder of this paper is organized in eight sections. Section 2 we document the optimizations for the motion estimate and motion compensation functions and discuss the results. Section 3 we discuss the optimization done on DCT/iDCT. Section 4 we explore the Dynamic parallelism. Section 5 we look at pinned Memory allocation. Section 6 we explore using FP16 half types and intrinsic from the CUDA math API. Section 5 we explore the memory allocation options in CUDA. section 7 we summarize the results of our optimizations. Section 8 concludes the paper.

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 92.87     43.12    43.12 500214528     0.00     0.00  sad_block_8x8
  1.57     43.85     0.73 11404800     0.00     0.00  dct_1d
  1.42     44.51     0.66 11404800     0.00     0.00  idct_1d
  1.08     45.01     0.50   705672     0.00     0.06  me_block_8x8
  0.54     45.26     0.25  2851200     0.00     0.00  transpose_block
  0.45     45.47     0.21 16398942     0.00     0.00  put_bits
  0.45     45.68     0.21   712800     0.00     0.00  dct_quant_block_8x8
  0.43     45.88     0.20  1425600     0.00     0.00  scale_block
  0.28     46.01     0.13   712800     0.00     0.00  dequant_idct_block_8x8
  0.26     46.13     0.12   356400     0.00     0.00  write_interleaved_data_MCU
  0.24     46.24     0.11  4752503     0.00     0.00  put_byte
  0.17     46.32     0.08   705672     0.00     0.00  mc_block_8x8
  0.17     46.40     0.08    21600     0.00     0.06  dct_quantize_row
  0.06     46.43     0.03    21600     0.00     0.05  dequantize_idct_row
  0.00     46.43     0.00     3300     0.00     0.00  put_bytes
  0.00     46.43     0.00     1200     0.00     0.00  write_DHT_HTS
  0.00     46.43     0.00      900     0.00     1.38  dct_quantize
  0.00     46.43     0.00      900     0.00     1.16  dequantize_idct
  0.00     46.43     0.00      301     0.00     0.00  destroy_frame
  0.00     46.43     0.00      300     0.00     0.00  create_frame
  0.00     46.43     0.00      300     0.00     0.00  flush_bits
  0.00     46.43     0.00      300     0.00     1.47  write_frame
  0.00     46.43     0.00      297     0.00     0.27  c63_motion_compensate
  0.00     46.43     0.00      297     0.00   146.87  c63_motion_estimate
  0.00     46.43     0.00        1     0.00     0.00  free_c63_enc
  0.00     46.43     0.00        1     0.00     0.00  init_c63_enc
```

Figure 1: initial profile with gprof

2 MOTION ESTIMATE AND COMPENSATE

2.1 memory allocation using cudaMallocManaged

When we started optimizing the first thing we did was change all the memory allocation in the precode to `cudaMallocManaged()`. Using `cudaMallocManaged()` we can allocate variables to managed memory that both the GPU and the CPU can access. This is possible in our case because Nvidia Jetson AGX Xavier boards have a unified memory, which means they share their memory address space [1].

With `cudaMallocManaged()` the code ran a bit slower `0m41,266s` to `0m49,330s` real time. This is likely because memory allocation to managed data that is accessible to both the CPU and GPU is more expensive to do. running the code with nvprof we see that the CUDA API spends 55% of its time in `cudaMallocManaged()` and 45% in `cudaFree()`.

2.2 sad_block_8x8

In our first attempt to use CUDA we went to `sad_block_8x8()` which is the heaviest load in the code and we tried optimizing it locally with CUDA running the calculations in parallel. We declared a function `sad_block()` with the CUDA keyword `__global__`, this makes a function that can be used to make a CUDA kernel. We used this function to configure a CUDA kernel with 8 block and 8 threads, `sad_block<<<8,8>>>` in `sad_block_8x8()`. With this we would do all the calculations in parallel and store each result in a array at index `index = blockIdx.x * blockDim.x + threadIdx.x`. In CUDA `blockIdx`, `blockDim` and `threadIdx` are built-in variables in the CUDA kernel that store the kernels indexes during run-time. after calling the CUDA kernel we would use `cudaDeviceSynchronize()` to make sure all the calculations were finished and then sum the array.

This did not work as the program was unbelievably slow, and We are not sure if it even would run to completion.

2.3 c63_motion_estimate

Since optimizing `sad_block_8x8()` in section 2.2 did not have a desired effect we thought that maybe there's a big cost to starting many small CUDA kernels so we moved to the top of the motion estimate call graph to `c63_motion_estimate`. In `c63_motion_estimate` we have two for-loops, one that loops through mb_rows and one for mb_cols which is the image divided into 8 by 8 blocks. looking more into the CUDA programming guide [2] the difference between blocks and threads is that the threads inside a block can talk to each other, but there's no communication between the normal CUDA blocks. With this in mind we decided make `me_block_8x8` into a `__global__` function and configure its block-size to the dimensions the for-loops go through. This replaces the for-loops and instead of passing the for-loop indexes, inside of `me_block_8x8` we set them to the blocks block index as shown in figure 3. the implementation of the new `c63_motion_estimate` using the CUDA framework is shown in figure 2 with the for-loops commented out.

This worked a lot better and was a significant speed up from `0m49,330s` to `0m15,692s` real time. Looking at the gprof profile there's not a lot different from the initial profile in figure 1 with the exception of sad_block_8x8 and me_block_8x8 functions are missing. As for nprof we see some more interesting changes, we see that the me_block_8x8 runs as a GPU activity and takes up 100% of GPU time, and the `cudaDeviceSynchronize()` takes up 91% of the time of CUDA API calls.

2.4 me_block_8x8 with threads for sad_block_8x8

We now have a working CUDA kernel for me_block_8x8 that we want multi-threaded. Our first thought was to optimize the sad_block_8x8. To achieve this we changed the kernel configuration to have 8x8 threads. In order for the threads inside a block to communicate with each other you need to use the CUDA keyword `__shared__` to initialize a variable. For our case we want a 8x8 array that all the threads can access. Since we want to remove the for-loops in sad_block_8x8 we moved the SAD calculation part into me_block_8x8. Then we replaced the loop indexes u, v with the CUDA built-in variables `threadIdx.x, threadIdx.y`. To make sure all the threads gets a chance to finish we also needed to add `__synchthreads()` after the the SAD calculation as shown in figure 4. Since the array have a value for each thread to write to we don't need to worry about the threads overwriting each other. we then summed all shared array in thread with index x=0 and y=0. This gave another speed up from `0m15,692s` to `0m13,286s` real time.

in the [2] we read about way of utilizing the threads to get the sum of the shared array using multiple threads called reduction. The idea being as long as the threads don't work on the same variables at the same time we can have them sum different parts of the array at the same time. Our shared array has a size of 64 variables and 64 corresponding threads. so we used half of the threads to sum their neighbor. then we used a quarter of the threads to sum the nearest neighbor sum and so on. we did this until we had gathered the sum of the shared array into the first value in the array.

this gave us a small speed up from `0m13,286s` to `0m13,187s` real time.

2.5 me_block_8x8 with threads for search range

When examining our solution looking for ways to speed it up more we noticed for-loops that are used to loop through the search range times 2. the search range is hard coded to be 16, but is cut in half for U and V component of the image. optimizing This loop would be a lot bigger than the 64 loop iterations we optimized for sad_block_8x8. So we reverted the changes we did in the last section to work on multi-threading these loops.

With similar approach we made a shared array with 32x32 values, since the shared variables have to be declared with constants we declared it with 32x32 since that was the biggest number of iterations the for-loops would get. Now instead of having each thread calculate a SAD value, each thread will instead call sad_block_8x8. To do this we needed to make sad_block_8x8 into a device function that can be called in the GPU with the CUDA keyword `__device__`. The thread indexes would now be used to feed a different block into the sad_block_8x8 function as shown in figure 5. We also tried implement reduction but it quickly became complicated when we needed to find the xy-indexes to the smallest SAD value and we got some broken motion vectors we did not find the solution for. When looking at the output from pred dumping the motion vectors we got the results in figure 6. Examining the image we see a lot of different colors but foreman is still clearly there, since those colors in YUV video formats are in the U and V components we suspect the issue lies in an error working with the indexes when the search range is 16x16 and not 32x32 as they are in the Y component.

Despite our challenges this was a significant speed up, `0m13,187s` to `0m10,889s` real time.

2.6 motion compensation

We can now apply what we have learnt from optimizing c63_motion_estimate to optimize c63_motion_compensate in the same way. We make mc_block_8x8 into a global function that can be configured as a kernel and we use same block dimensions as we did in motion estimate. In mc_block_8x8 the loops always have 64 for-loop iterations so we use 64 threads to do them in parallel.

This gives a slight speed down from `0m10,889s` to `0m11,085s` real time. Since moving data between the CPU and GPU has a small cost we tried merging the c63_motion_estimate and c63_motion_compensate functions to mitigate this. This did however not have any effect on the time but we decided to keep it like this as it off-loads c63_motion_compensate to GPU.

3 DCT AND IDCT

When optimizing DCT and IDCT We applied the same lessons we had learnt and went to the top of the call graph. We made the functions dct_quantize and dequantize_idct into global functions. They each had a loop that looped through the height of the image in increments of 8, so we configured the kernel to have the height of the image divided by 8 amount of threads. To make it run we

```
void c63_motion_estimate(struct c63_common *cm)
{
  /* Compare this frame with previous reconstructed frame
     */
  //int mb_x, mb_y;
  dim3 Y_dim(cm->mb_rows, cm->mb_cols);
  dim3 UV_dim(cm->mb_rows / 2, cm->mb_cols / 2);

  /* Luma */
  /*for (mb_y = 0; mb_y < cm->mb_rows; ++mb_y)
  {
    for (mb_x = 0; mb_x < cm->mb_cols; ++mb_x)
    {*/
  me_block_8x8 <<<Y_dim, 1>>>(cm,  cm->curframe->orig->Y,
      cm->refframe->recons->Y, Y_COMPONENT);
      cudaDeviceSynchronize();
    //}
  //}

  /* Chroma */
  //for (mb_y = 0; mb_y < cm->mb_rows / 2; ++mb_y)
  //{
    //for (mb_x = 0; mb_x < cm->mb_cols / 2; ++mb_x)
    //{
  me_block_8x8<<<UV_dim, 1>>>(cm,  cm->curframe->orig->U,
      cm->refframe->recons->U, U_COMPONENT);
      cudaDeviceSynchronize();
  me_block_8x8<<<UV_dim, 1>>>(cm,  cm->curframe->orig->V,
      cm->refframe->recons->V, V_COMPONENT);
      cudaDeviceSynchronize();
    //}
  //}
}
```

Figure 2: c63_motion_estimate with CUDA

```
  int mb_x = blockIdx.y;
  int mb_y = blockIdx.x;
```

Figure 3: mb_x and mb_y inside CUDA kernel

```
    __shared__ int s[8][8];
    // let each thread calculat their calculate SAD value
     , this replaces sad_block_8x8
     s[index_x][index_y] =
            abs(block2[threadIdx.x * w + threadIdx.y]
                - block1[threadIdx.x * w + threadIdx.y]);
    __syncthreads();
```

Figure 4: calculating SAD values in parallel

```
  int x = left+threadIdx.y, y = top+threadIdx.x;

  // let each thread calculate a SAD block with in bounds
     the bottom & right
  if(y < bottom && x < right)  {
    sad_block_8x8(orig + my*w+mx, ref + y*w+x, w, &s[
     index_x][index_y]);
  }
  // synch all threads to wait for all sad_block_8x8s to
    finish
  __syncthreads();
```

Figure 5: calculating sad_block_8x8 in parallel

also had to make all the other functions used by DCT and iDCT into device functions that are accessible to the GPU.

This gave a good improvement, going from `0m11,085s` to `0m8,881s` real time. Since there is a kernel call for each component Y, U and V we thought they could should rather each run in their own block in

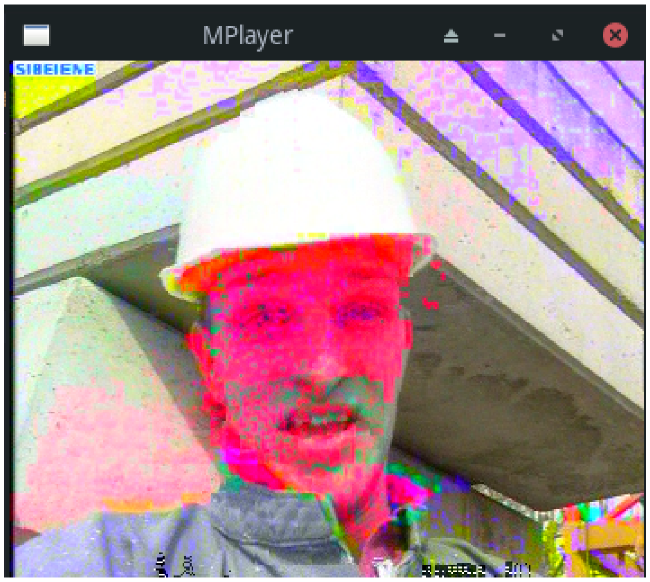


Figure 6: broken motion vectors with reduction in me_block_8x8

the same kernel. Having Y, U and V in separate blocks gave a small speed up `0m8,881s` to `0m8,001s`. Since moving data between the CPU and GPU has a small cost we tried merging the dct_quantize and dequantize_idct functions. This gave a pretty big speed up compared to when we did the same in motion estimate/compensate, going from `0m8,001s` to `0m7,079s`.

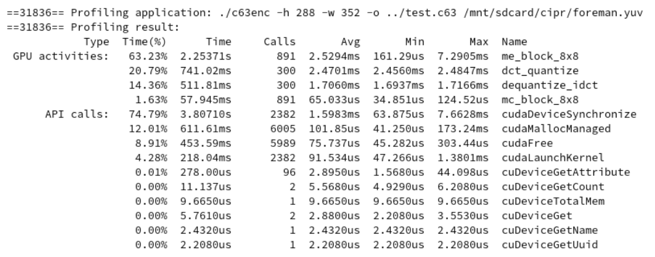
3.1 nvprof profile

Now after having off-loaded all the heavy functions into the GPU we can compare them in the nvprof profiling results shown in figure 7. compared to the neon intrinsic optimization in exam 1 motion estimation takes up 63% of the time in the GPU while in exam 1 we finished with motion estimation spending 85% of CPU time. A interesting observation is that `cudaDeviceSynchronize()` is starting to become a real bottleneck in the CUDA api calls.

4 DYNAMIC PARALLELISM

4.1 motion estimate and compensate

So far we have optimized c63_motion_estimate by making me_block_8x8 into a `__global__` function and launching it as a kernel, but so far we have learnt that the higher in the call graph we optimize from the better. When examining c63_motion_estimate we noticed that the computation on the Y, U and V component don't interact with each other. This means they can be in their separate blocks. We then got the idea to make c63_motion_estimate into a global function that have 3 blocks that launch their separate kernel of me_block_8x8 in parallel. A motivation for this is that we would then be able to reduce the bottleneck `cudaDeviceSynchronize()` is becoming. In the programming guide [2] we find that this is called Dynamic Parallelism. Separating Y, U and V for motion estimate and compensate into their own blocks gave the following speed up from `0m7,079s` to `0m6,980s`.



```
==31836== Profiling application: ./c63enc -h 288 -w 352 -o ../test.c63 /mnt/sdcard/cipr/foreman.yuv
==31836== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   63.23%  2.25371s       891  2.5294ms  161.29us  7.2905ms  me_block_8x8
                   20.79%  741.02ms       300  2.4701ms  2.4560ms  2.4847ms  dct_quantize
                   14.36%  511.81ms       300  1.7060ms  1.6937ms  1.7166ms  dequantize_idct
                    1.63%  57.945ms       891  65.033us  34.851us  124.52us  mc_block_8x8
      API calls:   74.79%  3.80710s      2382  1.5983ms  63.875us  7.6628ms  cudaDeviceSynchronize
                   12.01%  611.61ms      6005  101.85us  41.250us  173.24ms  cudaMallocManaged
                    8.91%  453.59ms      5989  75.737us  45.282us  303.44us  cudaFree
                    4.28%  218.04ms      2382  91.534us  47.266us  1.3801ms  cudaLaunchKernel
                    0.01%  278.00us        96  2.8950us  1.5680us  44.098us  cuDeviceGetAttribute
                    0.00%  11.137us         2  5.5680us  4.9290us  6.2080us  cuDeviceGetCount
                    0.00%  9.6650us         1  9.6650us  9.6650us  9.6650us  cuDeviceTotalMem
                    0.00%  5.7610us         2  2.8800us  2.2080us  3.5530us  cuDeviceGet
                    0.00%  2.4320us         1  2.4320us  2.4320us  2.4320us  cuDeviceGetName
                    0.00%  2.2080us         1  2.2080us  2.2080us  2.2080us  cuDeviceGetUuid
```

Figure 7: nvpof results after moving motion estimate, motion compensate, DCT and iDCT to the GPU

Dynamic parallelism breaks the nvprof so we will not be able to profile the kernels anymore while using this but we expect to have reduced time spent in `cudaDeviceSynchronize()`.

4.2 DCT and iDCT

Using the same thought, we tried to go down in the call graph for DCT and iDCT and turn dct_quantize_row and dequantize_idct_row into global functions but this slowed the program down. This is probably due to the same issue we ran into at the very beginning that launching a lot of smaller CUDA kernels is not a good idea. While for the motion estimation and compensation it only dynamically launches 3 new kernels in each Y, U and V block, the DCT and iDCT would launch a kernel for each thread it has.

4.3 Streams

We also tried using streams to replace the dynamic parallelism. Normally when you launch a kernel they are launched to the default stream 0. In CUDA stream 0 all the kernels will run sequentially. However you can make and specify different streams, this way the kernels can run in parallel in almost the same way we do it with dynamic parallelism. After testing it out we found Streams to be a little bit slower.

5 CUDA MEMORY ALLOCATION

So far we have used managed memory to allocate memory in our code. We are now interested in comparing managed memory with pinned memory. By exchanging `cudaMallocManaged()` with `cudaMallocHost()` we can allocate pinned memory on the host (CPU), then we can the variable with `cudaMemcpy()` to copy the variable over to the device device (GPU) and back again. We can also now remove all the `cudaDeviceSynchronize()` since `cudaMemcpy()` will wait for the kernels to finish working on the memory before copying it back to the CPU. This gave a slight speed up from `0m6,980s` to `0m6,281s`. But upon investigating the nvprof for the memory we noticed the bottleneck has moved from `cudaDeviceSynchronize()` to `cudaMemcpy()`.

6 NVIDIA CUDA MATH API

When looking into using mixed precision (FP16) with cuda i found the NVIDIA CUDA Math API [3]. Here was the documentation for half and half2 types, all you had to do was include `cuda_fp16.h`. Figure 8 shows the implementation of the different ways of calculating SAD values using the math api. Using half types was slower than doing it the normal way going from `0m6,281s` to `0m7,580s`. Looking throuhg the api there are various intrinsics listed, the ones catching our attention was the SIMD and Integer Intrinsics. When testing out the SIMD intrinsic __vsadu4 that calculates SAD value we found it to be slower than the normal way of doing it going from `0m6,281s` to `0m6,979s`. The last intrinsic we tried was the integer intrinsic `__sad()` which we found to have just about the same performance than the normal one so we decided to keep it.

```
*result = 0;
#pragma unroll
for (v = 0; v < 8; ++v)
{
  #pragma unroll
  for (u = 0; u < 8; ++u)
  {
      // normal SAD calculation
      *result +=
          abs(block2[v*stride+u] - block1[v*stride+u]);

      // Integer Intrinsics
      *result = __sad(block2[v*stride+u],
                      block1[v*stride+u], *result);

      // SIMD Intrinsics
      *result += __vsadu4(block2[v*stride+u],
                          block1[v*stride+u]);

      // half types
      half a = __uint2half_ru(block2[v*stride+u]);
      half b = __uint2half_ru(block1[v*stride+u]);
      *result += __half2int_ru( __hsub(a, b) );
  }
}
```

Figure 8: testing NVIDIA CUDA Math API in sad_block_8x8

7 EVALUATION

In section 2 we used CUDA to parallelize me_block_8x8 for motion estimation and mc_block_8x8 for motion compensation, this allowed the GPU calculate the heavy estimation calculations in parallel. We did lose some time by off-loading mc_block_8x8 to the GPU but we believe this is won back later in the optimization process when moving data between the CPU and GPU is more of a bottleneck. After optimizing motion estimation and compensation we went from `0m41,266s` to `0m11,085s`.

In section 3 we used CUDA to parallelize the dct_quantize and dequantize_idct functions and then we put each component Y, U and V in their own block so they could run in parallel. Lastly we merged together the two functions to reduce movement between CPU and GPU. After optimizing DCT and iDCT we went from `0m11,085s` to `0m7,079s`

In Section 4 we looked into dynamic parallelism and used it to compute motion vectors for Y, U and V in their own blocks by making the merged motion estimation/compensation function into a kernel and launching it with 3 blocks. This resulted in a small speed up from `0m7,079s` to `0m6,980s`.

In Section 5 we looked into using pinned memory instead of managed memory by replacing `cudaMallocManaged()` with `cudaMallocHost()`. this gave a small speed up `0m6,980s` to `0m6,281s`.

Finally in section 6 we looked into the CUDA math api and ended up using a integer intrinsic with the same performance to what we already had.

8 CONCLUSION

In this assignment we have learnt how to replace loops with CUDA kernels that runs in parallel which gives a big performance boost. We have discovered that the higher in the call graph we parallelize the better. We learnt about using reduction to make use of the threads we have to do calculations. We have learnt that in some cases Dynamic parallelism can give a slight boost. We also tried using the CUDA math API which ran slower or as fast. other lessons we picked up on the way is that `cudaDeviceSynchronize()` slows down the code and should be avoided when possible.

REFERENCES

[1] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. (2017). [Online; accessed 14-april-2021].

[2] nvidia. 2018. CUDA C Programming Guide. https://docs.nvidia.com/cuda/archive/10.0/cuda-c-programming-guide/index.html. (2018). [Online; accessed 14-april-2021].

[3] nvidia. 2018. NVIDIA CUDA Math API. https://docs.nvidia.com/cuda/archive/10.0/cuda-math-api/index.html. (2018). [Online; accessed 14-april-2021].