OPTIMIZATION SOFTWARE FOR DIGITAL HOLOGRAPHY IN THE ENVIRONMENT OF CUDA

Guzhov V. I., Ilinykh S. P.
NSTU
+7 (912) 925-39-24, vigguzhov@gmail.com

Abstract—The paper deals with the development of software for digital holography in the environment of CUDA is optimized according to the criterion of execution time.

Keywords—digital holographic interferometry, GPGPU, CUDA, algorithms digital holographic interferometry, fast Fourier transforms, discrete Fresnel transform.

INTRODUCTION

This paper discusses the use of GPU for scientific computing in digital holographic interferometry (CGI), which allows to significantly increase the computations liteline performance [1, 2], through the use of graphics processors of NVidia Corporation. This was solved the optimization problem of the following stages of the GRC as discrete Fresnel transform, the suppression of the DC component and the removal of a valid image, the elimination of phase ambiguity, the elimination of speckle noise and other phases. The implementation steps of the algorithm implemented for the graphics card one of the last generations, Quadro 5000, which is optimized to perform operations using real numbers in single precision. The result of parallel implementation of the software (software) is significant (in the 1.5-15 times) decrease in the runtime of the algorithm recovery of digital holograms. The obtained results give the possibility of applying the algorithm for digital holographic interferometry real-time.

PROBLEM STATEMENT

As a result of analytical review and analysis of existing methods of holographic interferometry [7-23], it was found that the most effective they can be solved by using GPGPU technologies-CUDA method of digital holographic interferometry using Fresnel transform. After researching the capabilities of the CUDA Toolkit in version 7.5, as well as hardware architecture of CUDA for parallel implementation, it was decided to use such modules in the CUDA Toolkit, as cuFFT to perform discrete Fourier transforms and cuBLAS to perform basic linear algebra operations. Also used CUDA kernel functions based on the GPGPU.

SOLUTION OF THE PROBLEM

Code of the CUDA kernel is the cyclic shift matrix is presented in listings 1, 2. First, calculate the index of the current item using the available data on the sequence numbers of the block and the thread running this kernel. Then it calculates the indexes of the matrix elements that will be displaced. The final step is to exchange for these items. Cyclic shift is called for half of the elements of the matrix, thus, will be done N/2 exchange elements in parallel, where N is the number of elements in the matrix.

Listing 1. Code CUDA kernel cyclic shift matrix

```
__global__ void MatrixHalfSizeCyrcularShiftKernel(cufftComplex *matrix, int
size) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int source_i = index / size;
    int source_j = index % size;
    int destination_i = (source_i + (size / 2)) % size;
    int destination_j = (source_j + (size / 2)) % size;
    cufftComplex temp = matrix[destination_i * size + destination_j];
    matrix[destination_i * size + destination_j] = matrix[source_i * size +
source_j];
    matrix[source_i * size + source_j] = temp;
}
```

Listing 2. A fragment of C++ code to call the CUDA kernel

```
void MatrixCUDAUtils::MatrixHalfSizeCyrcularShiftInplace(cufftComplex*&
matrix, int size) {
    int blocks_count = ((size * size) / 2) / THREADS_PER_BLOCK;
    MatrixHalfSizeCyrcularShiftKernel<<<blocks_count,
THREADS_PER_BLOCK>>>(matrix, size);
    cudaError_t error_code = cudaGetLastError();
    if (error_code != cudaSuccess) {
        fprintf(stderr, "Failed to launch MatrixHalfSizeCyrcularShiftKernel
kernel (error code %s)!\n", cudaGetErrorString(error_code));
        exit(EXIT_FAILURE);} }
```

Modern GPUs have high computing capabilities. This fact at the initial stage has generated considerable interest to use the GPU for General purpose computing instead of the CPU. Often GP with a large number of computing units the bottleneck is not bandwidth, computing units, and the width of the channel between the memory, CPU and GPU. Due to the large number of arithmetic logic unit (ALU) in the GPU sometimes it is impossible to put this amount of data to be processed, to avoid idle compute units. Thus, to explore the ways in which you can reduce the amount of traffic of memory needed for solving the problem [4].

The CUDA language makes available another kind of memory known as constant memory. From the name it is clear that this type of memory is used for data that is unchanged during the execution of the CUDA kernel.

There is a total of 64 KB constant memory on the GPU. Constant memory is cached, thus the constant appeal to memory is equivalent to reading from memory only if cache misses and otherwise this operation would be equivalent to reading from the constant cache [2].

For all threads in a half warp, reading from cache is as fast as reading from registers if all threads refer to the same address. Access streams to different locations within a half warp are consistent, thus, the complexity of this operation increases linearly with the increase in the number of different addresses that turn threads within the half warp.

In order to declare a variable in constant memory, use the keyword __constant__, as demonstrated in listing 3. Variables in constant memory declared in global scope, thus, the mechanism of Declaration of variables in the constant memory is equivalent to declaring a variable in shared memory.

The function cudaMemcpyToSymbol( ) is a special version of the function cudaMemcpy( ), where is being copied from the device memory to const memory area of GP, an example of using this function in your software presented on listing 4.

Reading from constant memory can be built between the threads of one half of the warp, thus effectively reducing the total number of read operations. Because constant memory is cached, then the serial address to the same address will not increase the traffic between the unit and the device.

Constants, known at the time of compiling a program should be defined using preprocessor macros (listing 5). Using variables in constant memory of the device is justified in the case that it is definitely known that the values will not be changed during run-time kernel. If the number of constants is relatively small, and their values affect the branching when executing the kernel, then the best option could also be the use of templates CUDA cores, whereas the template parameter constants are used [6].

Listing 3. A snippet of the description of the global space

```
#define FACTOR_ARRAY_SIZE 2
__constant__ double device_factor[FACTOR_ARRAY_SIZE];
```

Listing 4. A fragment of C++ code

```
double host_factor[FACTOR_ARRAY_SIZE];
double host_factor[0] = -HOLO_PI * waveLength * z_distance;
double host_factor[1] = -wavenum * z_distance;
cudaMemcpyToSymbol(device_factor, host_factor, sizeof(double) * FACTOR_ARRAY_SIZE);
```

Listing 5. The fragment code CUDA kernel

```
double var_real = cos(device_factor[0] * dist);
double var_img = sin(device_factor[1] * dist);
```

Library NVidia CUDA Fast Fourier Transform (cuFFT) provides a simple interface for computing the fast Fourier transform much faster than the CPU. Using hundreds of processor cores, NVidia GPUs, cuFFT delivers high-performance floating-point calculations to the GPU without the need to develop our own implementation of BFP on GP. Widely used in a variety of applications from computational physics to image processing and signal processing, cuFFT is an effective solution to calculate BFP for complex or real data sets. Library cuFFT uses algorithms based on the known algorithms of Cooley-Tukey and Blustein, so you can be sure that they will be accurate results effective method [3].

Because in your method, you must perform the Fourier transform for a matrix of complex numbers, we will use a two-dimensional Fourier transform library cuFFT, the implementation of which is shown in listing 6.

At the initial stage is determined by the execution plan, you specify the transformation parameters. Next, memory is allocated on the GPU for the matrix, then is done by copying the matrix from the memory to the memory card and executes two-dimensional Fourier transform on the graphics card. At the final stage, the data is copied back into RAM.

Listing 6. Implementation of the discrete Fourier transforms using cuFFT

```
cufftHandle plan2D;
cufftPlan2d(&plan2D, size, size, CUFFT_C2C);
cufftComplex *device_CUDA_data;
checkCudaErrors(cudaMalloc((void**)&device_CUDA_data, sizeof(cufftComplex)
* size * size));
```

```
checkCudaErrors(cudaMemcpy(device_CUDA_data, host_CUDA_data,
sizeof(cufftComplex) * size * size, cudaMemcpyHostToDevice));
cufftExecC2C(plan2D, device_CUDA_data, device_CUDA_data, forward ?
CUFFT_FORWARD : CUFFT_INVERSE);
checkCudaErrors(cudaMemcpy(host_CUDA_data, device_CUDA_data,
sizeof(cufftComplex) * size * size, cudaMemcpyDeviceToHost));
cufftDestroy(plan2D);
checkCudaErrors(cudaFree(device_CUDA_data));
```

Typically, the threads can safely interact with each other only if they exist on one unit. It is technically possible interaction of flows with different blocks, but it is much more difficult, and this feature makes the code prone to a large number of errors in the program.

In this section, you write kernels, which safely and effectively interact with each other to find average value of array of real numbers. CUDA is also easy to compute statistical values such as standard deviation, average, minimum, maximum, etc., using methods similar to the following solution.

Discussed below is the method officially recommended way to perform reduction operations [5]. Because to solve the problem, the threads should share the memory, it is important to form the structure blocks of threads. Variables must divide 8 bytes x 256 threads / block = 2 KB per block. Since PM has only 16 KB of shared (share) memory, thus, it is appropriate to use 2 KB of shared memory. Moreover, the blocks will consist of 256 threads each, giving 256 x 1 x 1 configuration.

The code in listing 7 is a while loop in which each iteration the number of active threads is reduced by half. This reduction is a popular technique because it can be executed in parallel. Initially, the threads block consists of 256 threads, the first time the while loop executes, the flow of 0..127 will compare their average values with threads 128..255. The results will be stored in shared memory for indices 0..127. The next pass through the loop only threads 0..63 will be active, etc. It is necessary to note that, despite the fact that the majority of threads will be idle during the execution of the kernel, search the middle value of the array on the GPU remains much faster than the same operation performed on CPU [2].

You must also pay attention to the fact that the kernel calls the function __syncthreads(). This feature is a barrier for all threads within each block. It may seem that it will slow down the execution due to the fact that some threads will be idle, but it is absolutely necessary to synchronize flows. Without calling this function there is a race.

Listing 7. CUDA to retrieve the average of the array

```
__global__ void getAverageKernel(double *array, double *avg_value){
__shared__ double avg[THREADS_PER_BLOCK];
   int array_index = BLOCKS_PER_GRID_ROW * THREADS_PER_BLOCK * blockIdx.y + 256 * blockIdx.x
+ threadIdx.x;
   avg[threadIdx.x] = array[array_index];
   __syncthreads();
   int nTotalThreads = blockDim.x;
   while(nTotalThreads > 1) {
      int halfPoint = (nTotalThreads >> 1);
      if (threadIdx.x < halfPoint) {
         avg[threadIdx.x] += avg[threadIdx.x + halfPoint];
         avg[threadIdx.x] /= 2;
      }  __syncthreads();  nTotalThreads = (nTotalThreads >> 1);
   }
   if (threadIdx.x == 0) {
      avg_value[BLOCKS_PER_GRID_ROW * blockIdx.y + blockIdx.x] = avg[0];
   }
}
```

CONCLUSION

The article describes the process of optimizing software for digital holography in the environment of CUDA. A parallel version of the algorithm is much faster for medium to large size holograms from 512 x 512 to 4096 x 4096. Compared to the sequential version, the execution time was reduced from eighteen months to fifteen times. The execution time of a parallel algorithm for holograms of small resolution is about a quarter of a second, making possible its application in real-time systems. The developed software was in the Fund of algorithms and programs of SB RAS [24].

ACKNOWLEDGMENT

This work was supported by RFBR (grant № 16-08-00565)

REFERENCES

1. http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/
2. http://docs.nvidia.com/cuda/cuda-c-programming-guide/
3. https://developer.nvidia.com/cuFFT
4. http://cuda-programming.blogspot.ru/2013/01/what-is-constant-memory-in-cuda.html
5. https://www.sharcnet.ca/help/index.php/CUDA_tips_and_tricks
6. http://stackoverflow.com/questions/16119923/using-constants-with-cuda
7. Madrigal R., Acebal P. , Blaya S., Carretero L, Fimia A., Serrano F. GPU-based calculations in digital holography // Holography: Advances and Modern Trends III. – 2013. - Vol. 8776.
8. S. Lee, H. C. Wey, D. K. Nam and D. S. Park. GPU implementation of wave field translation method for fast hologram generation // Optics and Photonics for Information Processing VIII. – 2014. - Vol. 9216.
9. Koki Murano1, Tomoyoshi Shimobaba1, Atsushi Sugiyama1, Naoki Takada1, Takashi Kakue1, Minoru Oikawa1, Tomoyoshi Ito. Fast computation of computer-generated hologram using Xeon Phi coprocessor // Computer Physics Communications. – 2014.
10. N. Múnera Ortiz, C. A. Trujillo and J. García-Sucerquia. Digital holographic interferometry accelerated with GPU: application in mechanical micro-deformation measurement operating at video rate // 8th Iberoamerican Optics Meeting and 11th Latin American Meeting on Optics, Lasers, and Applications. – 2013. - Vol. 8785.
11. Yasuyuki Ichihashi, Ryutaro Oi, Takanori Senoh, Hisayuki Sasaki, KokiWakunami, and Kenji Yamamoto. A real-time 3D system using electronic holography // SPIE. – 2014.
12. Mert Dogar, Hazar A. Ilhan, and Meric Ozcan. Real-time reconstruction of digital holograms with GPU // Practical Holography XXVII. – 2013. - Vol. 8644.
13. R.V. Uskov. O nekotoryh osobennostjah primenenija tehnologii CUDA dlja modelirovanija perenosa izlu-chenija // Vestnik MGTU im. N.Je. Baumana. Ser. “Estestvennye nauki”. 2011. № 3.
14. Guzhov V.I. Ispol'zovanie golograficheskoj i spekl-interferometrii pri kontrole promyshlennyh izdelij// Pjataja mezhdunarodnaja nauchno-tehnicheskaja konferencija "Opticheskie metody issledovanija potokov". - Moskva: MJeI.- 1999.- C.107-108.
15. Cifrovaja golograficheskaja sistema real'nogo vremeni. / Guzhov V.I., Il'inyh S.P., Hajdukov D.S., Kabak E.S. // Sbornik nauchnyh trudov Novosibirskogo gosudarstvennogo tehnicheskogo universiteta, Novosi-birsk , – 2014.-№4(78) – S. 97 – 112.
16. Algoritm rasshifrovki interferencionnyh kartin so sluchajnym fazovym sdvigom. / Guzhov V.I., Il'inyh S.P., Hajdukov D.S. // Sbornik nauchnyh trudov Novosibirskogo gosudarstvennogo tehnicheskogo universiteta, Novosibirsk , – 2014.-№4(78) – S. 79 – 96.
17. Guzhov V.I., Pod#jakov A.E., Solodkin Ju.N., Shtejngol'c Z.I. Vosstanovlenie fazy volnovogo fronta na osnove odnomernogo preobrazovanija Fur'e// Avtometrija.- 1992.- №6.- C.21-24.
18. Cifrovaja golograficheskaja interferometrija real'nogo vremeni dlja jeksperimental'nogo issledovanija naprjazhenno-deformirovannogo sostojanija dinamicheskih ob#ektov. / Guzhov V.I., Il'inyh S.P., Kuznecov R.A., Kabak E.S. // Omskij nauchnyj vestnik, Omsk , – 2015.-№1(137) – S. 158 – 162.
19. Guzhov V.I., Solodkin Ju.N. Analiz tochnosti opredelenija polnoj raznosti faz v celochislennyh in-terferometrah. // Avtometrija. 1992. № 6. S. 24.
20. Guzhov V.I., Il'inyh S.P., Kuznecov R.A., Vagizov A.R. Reshenie problemy fazovoj neodnoznachnosti metodom cifrovoj interferometrii. // Avtometrija. 2013. T. 49. № 2. S. 85-91.
21. Guzhov V.I., Il'inyh S.P., Hajdukov D.S., Kuznecov R.A. Novyj metod kalibrovki fazovyh sdvigov // Nauchnyj vestnik Novosibirskogo gosudarstvennogo tehnicheskogo universiteta. 2013. № 1. S. 185-189.
22. Guzhov V.I., Il'inyh S.P., Ubert A.G. Proekcionnyj metod izmerenija rel'efa ob#ekta. // Nauchnyj vestnik Novosibirskogo gosudarstvennogo tehnicheskogo universiteta. 2012. № 1. S. 23-28.
23. Guzhov V. I. Metody izmerenija 3D profilja ob#ektov. Fazovye metody. : ucheb. posobie / V. I. Guzhov ; Novosib. gos. tehn. un-t. - Novosibirsk : Izd-vo NGTU, 2016. - 83 s.
24. Chislennyj raschjot cifrovoj golograficheskoj interferometrii na platforme CUDA: Svidetel'stvo o registracii programmy v FAP SO RAN / Ryzhov P.S., Il'inyh S.P., Guzhov V.I. (RF). – №PR15011; opubl. 23.07.2015, [elektronnyj resurs] URL: http://fap.sbras.ru/node/4141.