# Overcoming the difficulty of large-scale CGH generation on multi-GPU cluster


Takanobu Baba
Utsunomiya University
Utsunomiya, Japan
baba@cc.utsunomiya-u.ac.jp

Shinpei Watanabe
Utsunomiya University
Utsunomiya, Tochigi, Japan
shinpei@virgo.is.utsunomiya-u.ac.jp

Boaz Jessie Jackin
NICT
Koganei, Tokyo, Japan
jackin@nict.go.jp

Takeshi Ohkawa
Utsunomiya University
Utsunomiya, Tochigi, Japan
ohkawa@is.utsunomiya-u.ac.jp

Kanemitsu Ootsu
Utsunomiya University
Utsunomiya, Tochigi, Japan
kim@is.utsunomiya-u.ac.jp

Takashi Yokota
Utsunomiya University
Utsunomiya, Tochigi, Japan
yokota@is.utsunomiya-u.ac.jp

Yoshio Hayasaki
Utsunomiya University
Utsunomiya, Tochigi, Japan
hayasaki@cc.utsunomiya-u.ac.jp

Toyohiko Yatagai
Utsunomiya University
Utsunomiya, Tochigi, Japan
yatagai@cc.utsunomiya-u.ac.jp



## Abstract

The 3D holographic display has long been expected as a future human interface as it does not require users to wear special devices. However, its heavy computation requirement prevents the realization of such displays. A recent study says that objects and holograms with several giga-pixels
should be processed in real time for the realization of high resolution and wide view angle. To this problem, first, we have adapted a conventional FFT algorithm to a GPU cluster environment in order to avoid heavy inter-node communications. Then, we have applied several single-node and multi-node optimization and parallelization techniques. The single-node optimizations include the change of the way of object decomposition, reduction of data transfer between CPU and GPU, kernel integration, stream processing, and utilization of multi-GPU within a node. The multi-node optimizations include distribution methods of object data from host node to the other nodes.

The experimental results show that the intra-node optimizations attain 11.52 times speed-up from the original single node code. Further, multi-node optimizations using 8 nodes, 2 GPUs per node, attain the execution time of 4.28 sec. for generating 1.6 giga-pixel hologram from 3.2 giga-pixel object. It means 237.92 times speed-up of the sequential processing by CPU using a conventional FFT-based algorithm.



***CCS Concepts*** • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Human-centered computing** → *Displays and imagers*; • **Software and its engineering** → *Parallel programming languages*; • **Applied computing** → Physical sciences and engineering;




*GPGPU-11, February 24–28, 2018, Vienna, Austria*



ACM ISBN 978-1-4503-5647-3/18/02…$15.00
https://doi.org/10.1145/3180270.3180273

***Keywords*** computer generated holography, large-scale CGH, GPU cluster

**ACM Reference Format:**
Takanobu Baba, Shinpei Watanabe, Boaz Jessie Jackin, Takeshi Ohkawa, Kanemitsu Ootsu, Takashi Yokota, Yoshio Hayasaki, and Toyohiko Yatagai. 2018. Overcoming the difficulty of large-scale CGH generation on multi-GPU cluster. In *GPGPU-11: General Purpose GPUs, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3180270.3180273


## 1 Introduction

The 3D holographic display has long been expected as a future human interface as it does not require users to wear special devices [9]. In display systems, the computer inputs 3D object data, computes wave propagation from each object point to each hologram point and stores the whole results to a SLM (Spatial Light Modulator). By applying reconstruction light to the SLM, the user can see the reconstructed 3D object as if it exists at the original position. This mechanism is well-known as computer generated holography (CGH).

One of the largest issues for realizing such display systems is their requirement for a huge number of computations. The computation cost of wave propagation from N object points to N hologram points becomes $O(N^2)$ [12]. In order to solve this problem, a wide variety of approaches have been taken. To accelerate the calculation of point-to-point wave propagation, the table-lookup method is used to evaluate trigonometric functions [11], [5], [15]. FFT based methods are also used to reduce the cost to $O(NlogN)$ [17]. However, the size of the objects and holograms these projects treat is still limited to mega-pixels and, for such a large number of pixels, the processing time becomes hundreds to even thousands of seconds [16],[11].

On the other hand, the capability of 14.7 giga pixel object display is required for the human eyes' recognition capability [2]. Further, the necessity of a 4.5 giga pixel SLM is pointed out for realizing the view angle of 20 degree for diagonal 7 inch object [13]. Thus, as the human interface, the giga-pixel order is required to realize high resolution and wide view angle.

If we treat, for example, 4 giga-pixels, each pixel requires 8B as a complex number representation and total 32 GB memory space

is required to store just the object data. We need additional memory space for the output hologram and working space. The global memory size of current GPUs is far behind from these requirements [8]. Thus, if we want to utilize the parallel computation capability of GPU clusters as well as the low computation cost of FFT based methods, we need to decompose the input object to multiple sub-objects and apply distributed FFT algorithms to them. In this case, we encounter a further problem that the distributed FFT algorithms require a lot of node-to-node data transfer for realizing butterfly operations [1].

To this problem, we have been working on a distributed algorithm for generating 2D Fourier holograms [4] and 3D Fresnel holograms [14] on a GPU cluster. Aiming at the ultimate goal of realizing 3D holographic display with high-resolution and wide view angle properties, this research
shows how we resolve the difficulties of large-scale CGH generation on multi-GPU clusters by adapting the FFT-based algorithm to the clusters' environment and applying applicationoriented optimizations under the multicore-CPU and multi-GPU combined heterogeneous architecture.

This paper is organized as follows. Section 2 describes the basic concept of the conventional FFT-based CGH generation and the object decomposition method. Sections 3 and 4 describe the application of optimizations within single node and multiple nodes, respectively. Section 5 shows experimental results using multi-GPU cluster. Section 6 concludes the paper.

## 2 Algorithm adaptation for GPU cluster implementation

Fig.1 shows the process of 3D image reconstruction using conventional FFT-based methods. First, the wave propagation from object planes to hologram plane is computed by using FFT; the computed results are transferred to SLM; and by using laser light the 3D image is reconstructed.

Equations from (1) to (3) express the wave propagation computation from object plane $(x_1, y_1)$ to hologram plane (x, y).

$$H(x,y) = QP(x,y) \times FFT_{N\times N}\left[(O(x_1,y_1) \times QP_1(x_1,y_1))\right] \tag{1}$$

$$QP(x,y) = e^{j\frac{k(x^2+y^2)}{2z}} \tag{2}$$

$$QP_1(x_1,y_1) = e^{j\frac{k(x_1^2+y_1^2)}{2z}} \tag{3}$$

In the expression, $H$ is a hologram, $O$ is the two-dimensional object, $x_1$ and $y_1$ are the object's horizontal and vertical coordinates, $x$ and $y$ represent the hologram ' horizontal and vertical coordinates, $z$ is the distance from the object to the recording plane and $N$ is the number of pixels on one dimension of the object. $k$ is the value of $2\pi$ divided by the wavelength of light ($\lambda$).

Fig. 2 shows the flow of computation. The first three steps realize the computation of Equations (1) through (3). The first step "Calculate $O \times QP_1$" computes $O(x_1, y_1) \times QP_1(x_1, y_1)$. The second step "2D-FFT" realizes $FFT_{N\times N}$. The third step "Multiply QP" realizes multiplication by Equation (2). The final Layer-add module adds the results for multiple layers.

Fig.3 shows the process of 3D image reconstruction using the object decomposition method [4], adapted to GPU clusters. The input object is first decomposed into sub-objects; the three operations of interpolation, FFT, and shift are applied to each sub-object to produce a sub-CGH; and from each sub-CGH the original sub-object is reconstructed at the original position. Interpolation is necessary to keep the original bandwidth and make the object segment
reconstructed successfully. A shift operation is necessary for reconstruction at the original position of the object segment. Without this operation all the object segments are reconstructed at the center. For the theoretical background and detailed explanation of the decomposition method as well as the preliminary results of its application to the Fresnel hologram generation, see [4] and [14], respectively.

As shown in the figure, the sub-CGHs do not need to be summed up to yield the final CGH. Time-sequential reconstruction from sub-CGHs produces the 3D image that can be seen as the original 3D image if the reconstruction is performed at high speed. Thus, compared to conventional FFT-based methods [17], we can save the time for the final addition of sub-CGHs.

Based on these discussions, the flow of the decomposition method can be drawn as Fig.4. In the figure, the interpolation, 1D-FFT and shift operations are applied in the x- and y-directions, sequentially. This process is repeated for different object layers. We can avoid the final addition of sub-CGHs as shown in Fig.3. However, we can not avoid the layer-add by CPU for CGHs of different object layers, as shown in Fig.4.

Notice that we omit the loop structure of the flows for simplicity.

Comparing with the conventional method, the decomposition method requires less communication as the computation processes from sub-objects to sub-CGHs are mutually data-independent. Thus, once the original object is decomposed into sub-objects and they are sent to computing nodes, no node-to-node communications are necessary at all. Notice the final reconstructions are also mutually independent. However, we should be careful about the following two points: the decomposition method increases the amount of computation for interpolation and space-shift; and the sub-CGH generated from sub-object requires the same size of memory space with the whole CGH.

Thus, the key issues of object decomposition method implementation on GPU clusters are, i) if the additional computation cost for the interpolation and shift operations is paid back by less communication cost and omitted final addition, and ii) if the sub-CGH, enlarged within GPU, is appropriately stored back to the host CPU. Our effort of optimizations and parallelization (sections 3 and 4) and the experimental results (section 5) will answer these issues.

## 3 Single-node, multi-GPU optimizations

This section describes the optimizations and parallelization within a single node with multi-GPU.

### 3.1 Changing distributed accesses to contiguous ones (optimization 1)

When we send data from CPU to GPU, the data transfer is performed by DMA. And DMA requires the input data to be on the contiguous address space.

At first, we decompose an input object plane into x- and y-directions like Fig.5 (a). In this case, we need to rearrange the data

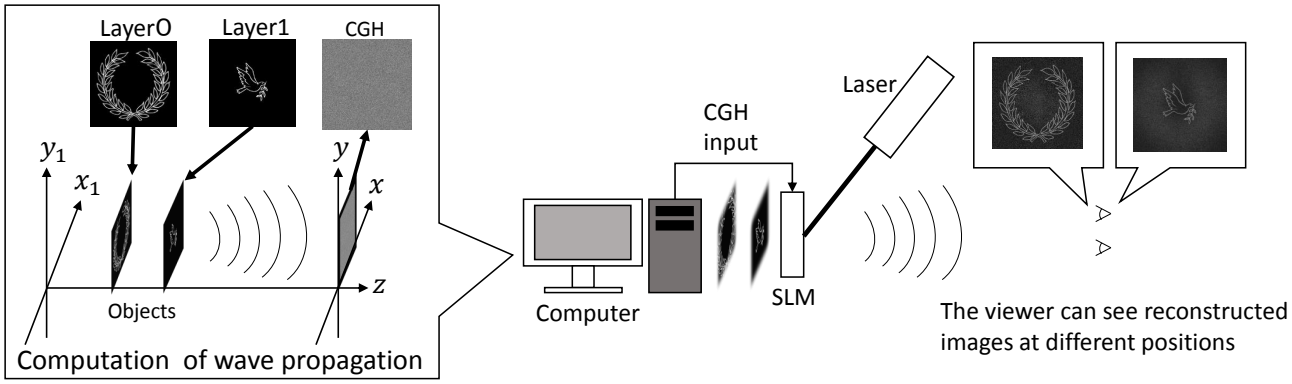


**Figure 1.** FFT-based CGH generation

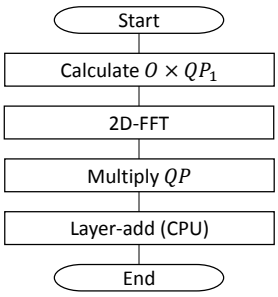


**Figure 2.** Flow of FFT-based CGH generation

so that the selected parts become contiguous on the memory before transferring from CPU to GPU.

We have improved the way of decomposition by changing to y-direction only decomposition, as shown in Fig.5(b). This makes the decomposed areas contiguous in the memory and we can transfer the data without rearrangement.

Fig.6 shows the flow after this optimization. By comparing this flow with the original flow of Fig.4, we can understand that, in the original flow, the combination of the x- and y-direction operations is repeatedly executed. However, in Fig. 5, once the x-direction operation is executed, the resulting data in local memory can be utilized repeatedly by the following y-direction operations. This reduces the number of kernel executions for x-direction operations.

Notice that in the original way of decomposition the number of decomposition should be $2^{2n}$ where $n = 1, 2, \ldots$. However, the change of the way of decomposition changes the number to $2^n$ and thus makes it easier to adjust the sub-object size to the limitation of GPU's memory size.

### 3.2 Reduction of data transfer amount between CPU and GPU (optimization 2)

As the data transfer time between CPU and GPU is added to the computation time, the GPU's fast computation may be impaired by the slow transfer time. Thus, we should be careful about the contents of the transfer.

Our CGH generation program uses cufft, a CUDA library, for the FFT computation [7]. It requires cufftComplex as its input array. The cufftComplex consists of a real part and an imaginary part, and each part requires 4B memory space for storing floating point data. Thus, in our original program, both CPU-to-GPU input transfer and GPU-to-CPU output transfer use an 8B cufftComplex type.

However, the use of this data type is redundant. A pixel of the input object uses just one byte of the real part. This can be replaced with 8-bit unsigned integers. In a similar manner, the output from GPU only uses a 4B real part and can be replaced with a floating point number. Fig.7 shows this change schematically.

### 3.3 Kernel integration and utilization of high-speed memory(optimization 3)

Usually, a GPU has thousands of execution cores for arithmetic and logical operations [3]. In order to utilize their computing power effectively, it is critical to smoothly provide them data to be processed. For this purpose, GPU has a hierarchically structured memory system of register file, local memory, cache memory (usually multi-level), and global memory. The input data to GPU is first sent to the global memory. When a kernel starts its execution, it reads data from the global memory to a higher level memory and tries to keep them at the higher level until the end of the execution. At the end of the execution, the results should be saved to global memory to pass them to the succeeding kernel. Then the succeeding kernel reads the data again into higher level memory to perform its operation. The problem here is the large overhead of this write-back and read-out cost of global memory, the slowest memory in the hierarchy. We cannot overlook that these write-back and read-out operations also lose the locality of references.

In order to solve this problem, we have tried to integrate kernel functions into one kernel function as well as possible. Fig.8 shows the integrated results of the three kernels for y-direction shift, transpose and CGH position calculation, which are separately called in Fig. 6. Before the integration, each kernel reads all the pixels from global memory, applies each computation and stores the results back to global memory. After the integration, the three kernel functions are integrated into one kernel function. This kernel function reads a part of pixels from global memory, applies the three operations, i.e. Shift, Transpose and Multiply QP, to them and stores the partial results back to global memory, repeatedly, until all the pixels are processed. By this integration, we can utilize higher-level memories, such as registers and shared memory, for

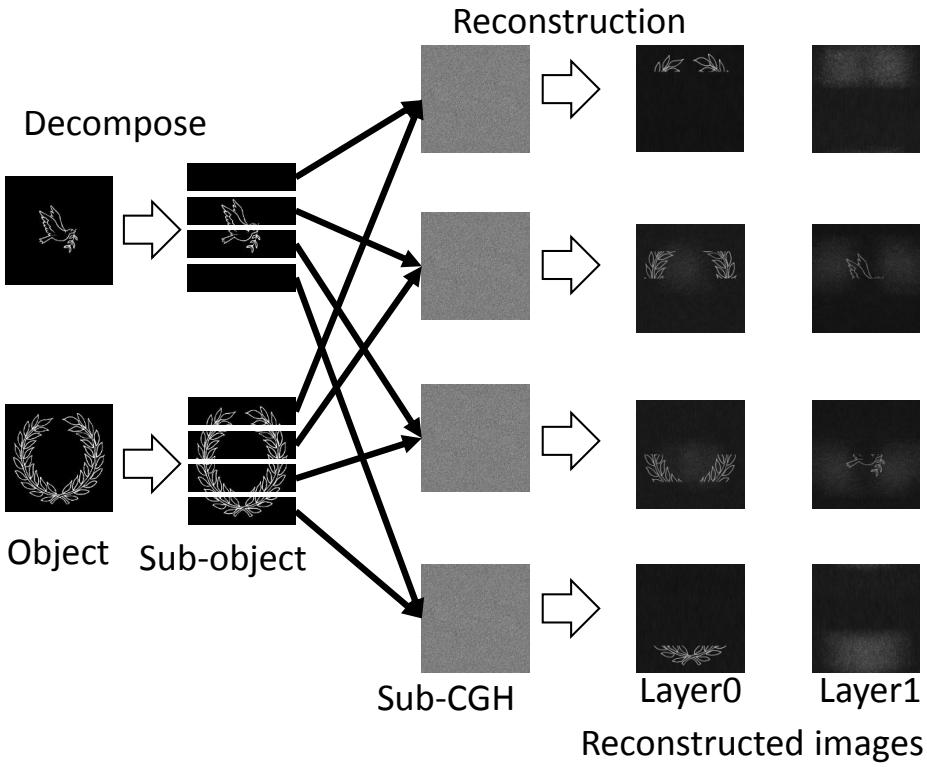


**Figure 3.** CGH generation and its display mechanism of object decomposition method

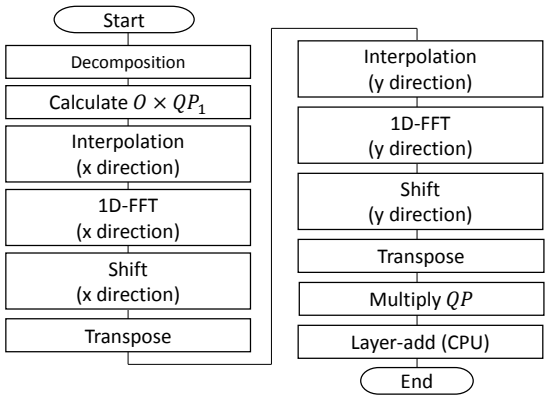


**Figure 4.** Flow of the decomposition method

inter-kernel data passing. The integration also reduces the number of kernel function calls and, thus, reduces the cost for GPU control.

### 3.4 Stream processing of GPU calculations and transfer of the result (optimization 4)

As described in section 2, the sub-CGH, generated from the sub-object, becomes the same size as the final CGH. Thus, we need to send the partial results back to the CPU to save the limited memory space of the GPU. Fortunately, the generation process can be divided into $K$ mutually data-independent operations where $K$ represents the number of decomposition.

We have utilized the stream processing to treat these computation and communication pairs efficiently, as shown in Fig.9. That is, first, the pair of the partial result calculations in GPU, called Sub-CGH calcA and calcB, are executed. Then, the results are transferred from GPU to CPU by Sub-CGH data transfer operation. Finally, the layer-add operation, called Sub-CGH Final-addition, is executed in CPU.

The pipelining of these operations should take into account the following conditions. The first pair calculations should be serial as Sub-CGH calcB inputs the output of calcA. The transfer from GPU to CPU should wait for the completion of Sub-CGH calcB in GPU. Sub-CGH calcA should wait for the completion of Sub-CGH calcB operation of the preceding pipeline as they share the same buffer memory.

### 3.5 Utilization of multi-GPU under the control of multicore CPU (optimization 5)

If multiple GPUs are available in a single node, we may utilize their parallel processing capability. In this case, it is desirable to assign them mutually data-independent processing so that they can execute in parallel.

In the CGH generation, if we assign sub-objects of different layers to the multiple GPUs, the generation process becomes mutually independent. In order to control these parallel operations in multi-GPU, we use multithreading in CPU. One thread controls one GPU, by specifying the GPU using cudaSetDevice[7].

One important issue for utilizing the multiple GPUs is how the CPU receives the results of the computation from them and, if necessary, does some reduction operations on the results. In our

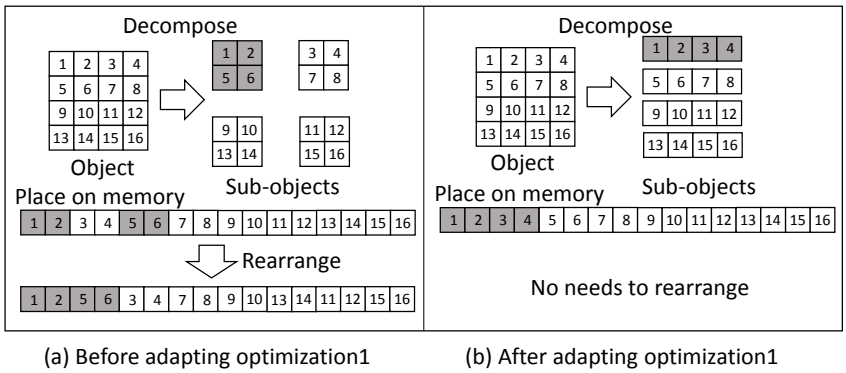


**Figure 5.** Change from two-dimensional decomposition to one-dimensional one

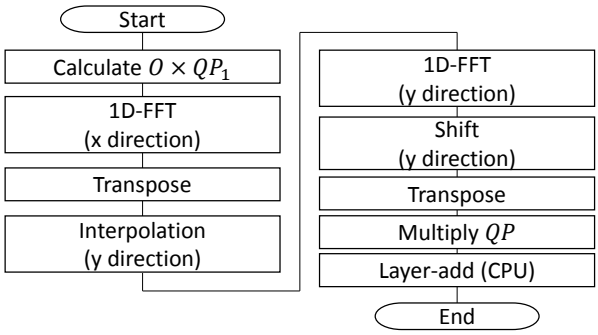


**Figure 6.** Flow after optimization 1

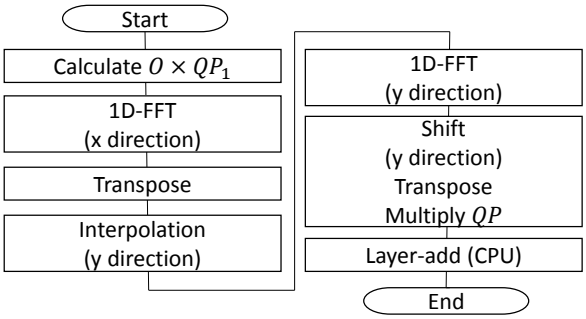


**Figure 8.** Integration of three kernels

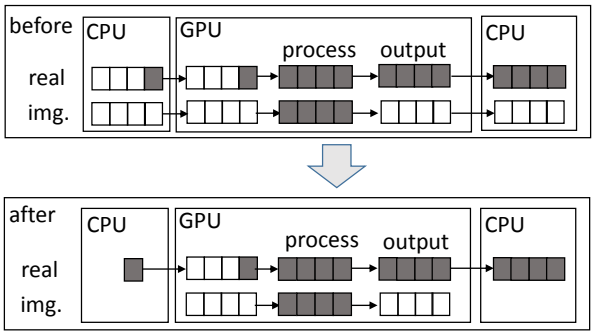


**Figure 7.** Reduction of data size transferred between CPU and GPU

decomposition model, the reduction operation is the layer-add of sub-CGHs generated from different object layers. We considered three methods to treat this issue: (i) to provide disjoint areas for different GPUs, (ii) to use "lock" to prevent the other threads modify the same area, and (iii) to use a synchronization mechanism for avoiding the two threads handle the shared data area. We performed preliminary evaluation of these methods and found that (iii) attains the best performance. Thus, we use this method for our experiments, as described later.

## 4 Multi-node optimizations

The major principle of the decomposition method is to save communication cost by avoiding frequent data transfers, required for 2D-FFT operations. Following this principle, we assign sub-objects with the same x- and y-coordinate values but with different layers to a single node. This assignment avoids unnecessary data transfers during the computation.

One inevitable data transfer operation occurs when the host node distributes sub-objects of different layers to the other nodes. We must consider that the input object is given in a compressed format. The object in the compressed format can not be decomposed and we need to decompress it before decomposition. Thus, there are two possibilities.

The first method is to decompress the input object to bitmap, i.e., one pixel to 8-bit unsigned integer, on the host node, decompose the decompressed object into sub-objects, and transfer each sub-object to an appropriate node. In this method, the necessary memory space for storing the sub-object is determined statically. Further, it should be smaller than the size of the original compressed object. Fig.11 shows how the input object is decompressed and decomposed into sub-objects in the host node, Node 0, and sent to the other nodes for sub-CGH generation. Fig.12 shows the flow of this method. The decompressed and decomposed sub-objects are sent by MPI send and recv pair operations, sequentially.[10]

The second method is that the host node first distributes the compressed whole object to all the other nodes, as shown in Fig.13.

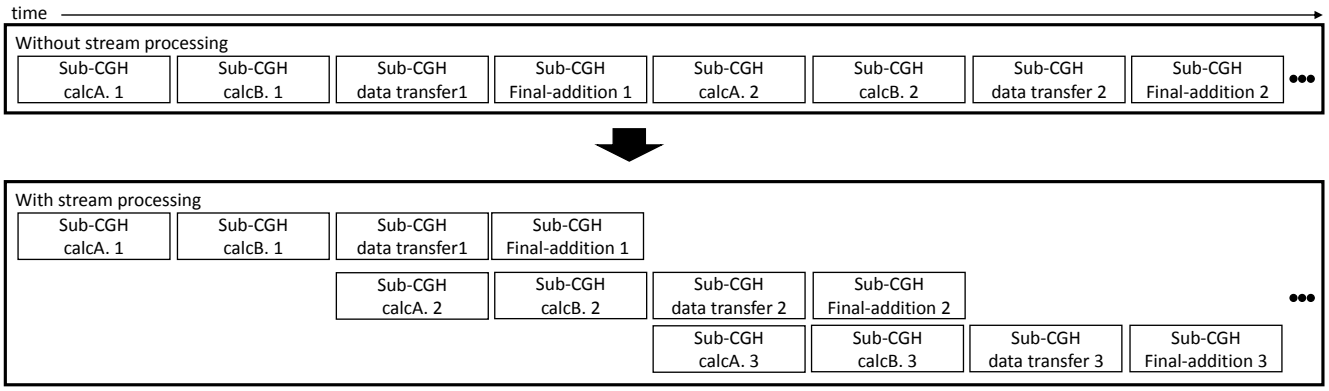


**Figure 9.** Stream Processing between GPU and CPU

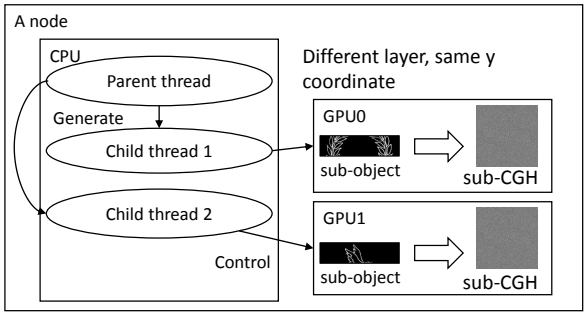


**Figure 10.** Multi-thread controlled multi-GPU

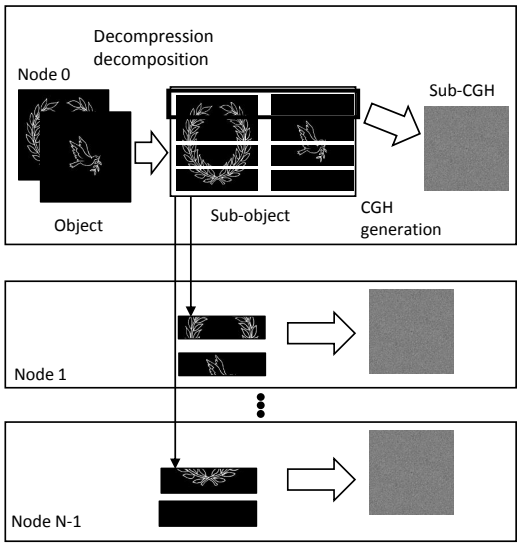


**Figure 11.** Decompress-and-unicast method

Then, each node decompresses the input object and extracts the sub-object to be processed in the node. In this method, the data size of the compressed object is dynamically determined. Thus, the size should be distributed first from the host to the other nodes so that they can allocate necessary memory space for storing input compressed object, as shown in the flow of Fig.14. The MPI broadcast is used to distribute the compressed whole object. Each node needs to perform decompression separately.

In the following section, we will show the performance results of these two methods.

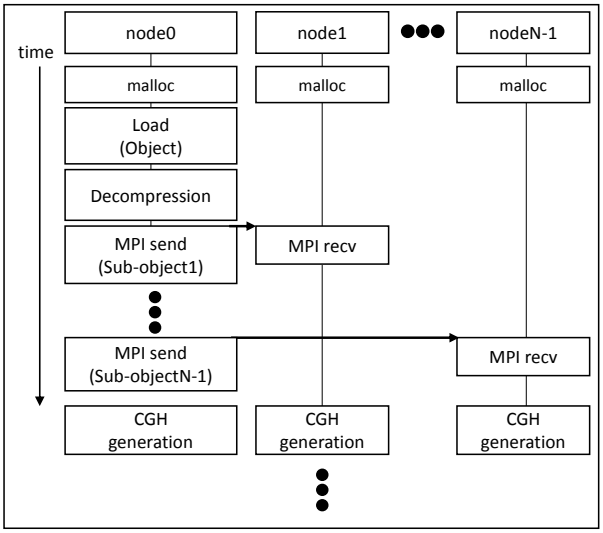


**Figure 12.** Flow of decompress-and-unicast method

## 5 Experimental results

In order to verify the effectiveness of the optimizations, we have performed experiments. The execution environment is summarized in Table 1. Simply speaking, it is 1000BASE-T connected 8 node GPU cluster and each node contains two GPUs. Many similar projects use a high-speed network, such as Infiniband, to treat this kind

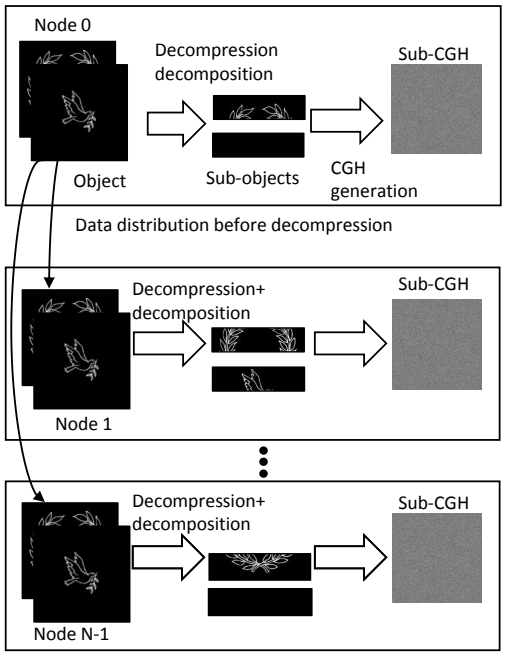


**Figure 13.** Multicast-and-decompress method

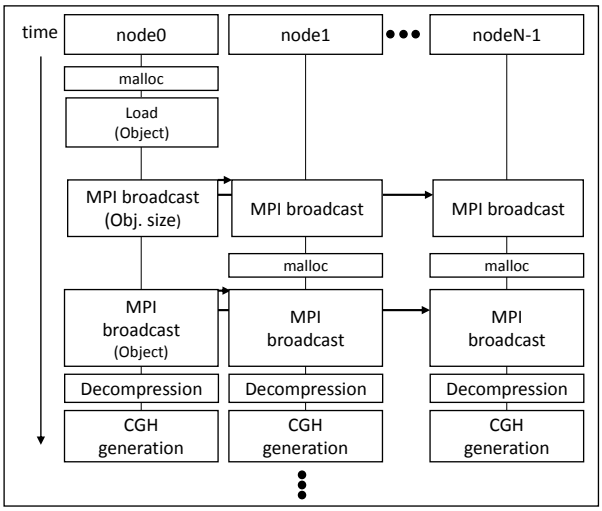


**Figure 14.** Flow of multicast-and-decompress method

of large-scale bandwidth intensive task without locality [1], [6]. Thanks to the use of the decomposition method, we can use a low cost gigabit Ethernet as the data transfer occurs once at the beginning. We use C to describe CPU code, Pthreads to describe multithreading for optimization 5, MPI Ver. 1.10.3 to describe the internode communication, and CUDA version 8.0 to describe the GPU code.

The input object is 2 layers of 40K*40K pixels. Thus, the total size is 3.2 giga pixels. The hologram size is 1.6 giga-pixels.

**Table 1.** Evaluation environment

| | | |
|---|---|---|
| CPU | model number | Core i7 6850K |
| | frequency | 3.60GHz |
| | memory | 64GB |
| | number of cores | 6 |
| GPU (each node has 2GPUs) | model number | GeForce GTX 1080 Ti |
| | frequency | 1.58GHz |
| | memory | 11GB |
| | number of cores | 3584 |
| OS | | CentOS7.3 |
| LAN | | 1000BASE-T |
| CUDA ver. | | 8.0 |
| OpenMPI ver. | | 1.10.3 |

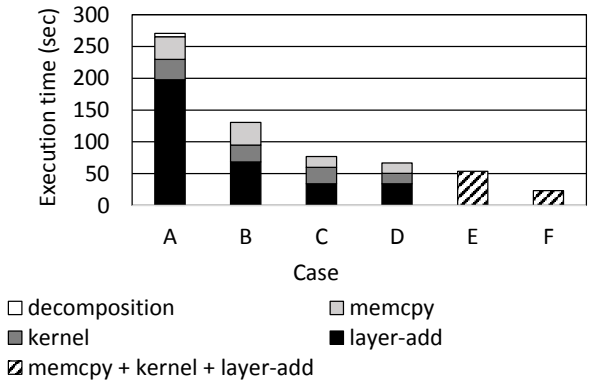


**Figure 15.** Single-node results

Our first experiment is to apply the optimizations 1 through 5 to clarify their effect in a single node. Throughout the experiment the number of decomposition is fixed at 16. Notice the number should be $2^{2n}$ before optimization 1 and we must use the same value consistently. Thus, the host CPU sends total 32 $(16(sub-objects/layer)\times 2(layer))$ sub-objects to the GPU to process all of them.

Fig.15 shows the results where the execution time of each bar is decomposed into the times for decomposition, memory copy (memcpy), kernel, and layer-add. The decomposition and layer-add are done by CPU. Memory copy includes both CPU-to-GPU and GPU-to-CPU transfers. The kernel processes 32 sub-objects and the kernel time includes the kernel-call time.

The bar A represents the starting point of our optimization. Conventional optimizations, such as an appropriate definition of grid and blocks, coalescing access of global memory, and efficient use of local memory, have already been applied.

B shows the effect of optimization 1. The change of the way of decomposition eliminate the decomposition time, shown as a very small white box at the top of bar A. It also reduces the kernel time as the results of $x_1$ direction processing is repetitively utilized as described before. Notice that the effect of contiguous results data also shortens the layer-add time of CPU.

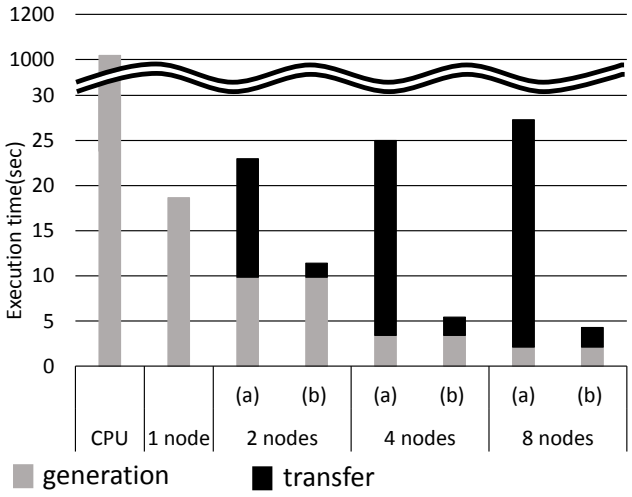


**Figure 16.** Multi-node results

C shows the effect of optimization 2. The reduction of data size reduces the memory copy time. Changing from complex data type to real one reduces the layer-add time of CPU.

D shows the effect of optimization 3. The integration of kernels naturally reduces the kernel time.

As to E and F, we cannot simply decompose the execution time as the optimizations realize parallel operations between computation and communication (optimization 4) and between multiple GPUs (optimization 5).

E shows the effect of stream processing. It is not so much as the GPU-to-CPU data transfer cannot be overlapped with the preceding sub-CGH's layer-add as described before.

F represents the effect of optimization 5. 2 GPUs nearly halves the time for E.

Eventually, F attains 11.52 times speed-up from A.

Fig.16 shows the execution time of CPU and GPU cluster, changing the number of cluster nodes.

The left-most bar represents the sequential processing time by CPU, following the flow of Fig.2, i.e. a conventional FFT-based algorithm. The CPU uses just the CPU part of Table 1. The sequential code is written in C and 2D-FFT is realized by the FFTW 3.3.2 library. We obtain this time as the baseline performance. Thus, we intentionally do not utilize CPU parallelism, such as multicore multithreading and SIMD extensions. The results show 1018.32 seconds of processing time.

For the experiment using the 8 node GPU cluster, we fixed the number of decomposition at 8. The reason is twofold: first, the 8 times decomposition enables us to fit the sub-object and the related working spaces within the global memory size, and second, if we increase the number, the computation cost will increase. Thus, the number of decomposition is determined as a compromise between the global memory size and the computation cost.

For the next bar of 1 node we use one node of the GPU cluster defined in Table 1. From 2 nodes to 8 nodes, we applied the multi-node optimizations described in section 4. The bars for (a) and (b) show the execution time for the decompress-and-unicast method and multicast-and-decompress one, respectively. Each bar is decomposed into CGH generation time and transfer one.

From the results we understand that the multicasting compressed object is much faster than the sequential unicasting of decompressed sub-objects. The right most bar indicates that 8 nodes cluster attains the execution time of 4.28 sec., which means 237.92 times speed-up of the sequential processing on CPU.

## 6 Conclusion

We have described our research results for overcoming the difficulty of large-scale CGH generation on multi-GPU cluster. Our efforts include algorithm adaptation for multi-GPU cluster and both intra- and inter-node optimizations of the code for multicore CPU and multi-GPU combined heterogeneous node architecture.

The intra-node optimizations attain an 11.52 times speed-up from the original single node code. The extreme results, using our 8 nodes 2-GPU architecture, show 4.28 sec. execution time for 3.2 giga-pixel object and 1.6 giga-pixel hologram. This is 237.92 times faster than the sequential processing by CPU using a conventional FFT-based algorithm.

Thus, the major contribution of this paper is to show that we can generate a giga-pixel CGH from a giga-pixel object in a few seconds under the constraints of limited memory size of GPUs. It is enabled by adapting an FFT-based algorithm to GPU cluster environment and by applying application-oriented optimization and parallelization techniques.

Our future plan is to further reduce the extreme processing time of 4.28 sec. to realize our ultimate goal of real time 3D display with high-resolution and wide view angle. For this objective, we need to analyze the current results and seek for possibilities of performance improvement by using higher performance processors and network as well as their code optimization and parallelization.

## Acknowledgment

We would like to thank the reviewers for their helpful comments. This work was supported in part by JSPS KAKENHI Grant Number 17K00265.

## References

[1] Y. Chen, X. Cui, and H. Mei. 2010. Large-Scale FFT on GPU Clusters *(Proc. ACM International Conference on Supercomputing)*. 315–324.
[2] D.G. Curry, G. Martinse, , and D.G. Hopper. 2003. Capability of the human visual system *(Proc. SPIE)*, Vol. 5080.
[3] J. L. Hennessy and D. A. Patterson. 2011. *Computer Architecture 5th Edition A Quantitative Approach* (5th. ed.). Morgan Kaufmann.
[4] B.J. Jackin, H. Miyata, T. Ohkawa, K. Ootsu, T. Yokota, Y. Hayasakiand T. Yatagai, and T. Baba. 2014. Distributed caluculation method for large-pixel-number holograms by decomposition of object and hologram planes. In *Optics Letters*, Vol. 39.
[5] K. Murano, T. Shimobaba, A. Sugiyama, N. Takada, T. Kakue, M. Oikawa, and T. Ito. 2014. Fast computation of computer-generated hologram using Xeon Phi coproceeso. In *Computer Physics Communications 185*. 2742–2757.
[6] H. Niwase, M. Fujiwara, H. Araki, Y. Maeda, H. Nakayama, T. Kakue, T. Shimobaba, T. Ito, and N. Takada. 2015. Fast computation of computer-generated hologram using multi-GPU cluster system for a single spatial light modulator *(Forum on Information Technology)*, Vol. 14. 41–44.
[7] NVIDIA. 2016. CUDA C PROGRAMMING GUIDE NVIDIA.
[8] NVIDIA 2017. Tesla V100. (2017). Retrieved Nov 25, 2017 from https://www.nvidia.com/en-us/data-center/tesla-v100/
[9] L. Onural, F. Yaras., and H. Kang. 2011. Digiral Holographic Three-Dimensional Video Displays *(Proc. IEEE 99)*. 576–589.
[10] Open MPI 2017. Open Source High Performance Computing. (2017). https://www.open-mpi.org/
[11] Y. Pan, X. Xu, and X. Liang. 2013. Fast distributed large-pixel-count hologram computation using a GPU cluster *(Applied Optics)*, Vol. 52.
[12] N. Takada, T. Shimobaba, H. Nakayama, A. Shiraki, N. Okada, M. Oikawa, N. Masuda, and T. Ito. 2012. Fast high-resolution computer-generated hologram

computation using multiple graphics processing unit cluster system *(Applied Optics)*, Vol. 51. 7303–7307.

[13] R.B.A. Tanjung, X. Xu, X. Liang, S. Solanki, F. Farbiz Y. Pan, B. Xu, and T-C. Chong. 2010. Digital holographic three-dimensional display of 50-Mpixel holograms using a two-axis scanning mirror device *(Optical Engineering)*, Vol. 49(2).

[14] S. Watanabe, B.J. Jackin, T. Ohkawa, K. Ootsu, T. Yokota, Y. Hayasaki, T. Yatagai, and T. Baba. 2017. Acceleration of large-scale CGH generation using multi-GPU cluster, *(Proc. Workshop on Advances in Networking and Computing)*.

[15] D. Yang, J. Liu, Y. Zhang, X.Li, and Y. Wang. 2016. The optimizations of CGH generation algorithms based on multiple GPUs for 3D dynamic holographic display *(Proc. of SPIE)*, Vol. 10153.

[16] Y. Zhang, J. Liu, X. Li, and Y. Wang. 2016. Fast processing method to generate gigabyte computer generated holography for three-dimensional dynamic holographic display *(Chinese Optics Letters)*. 030901–1–030901–5.

[17] Y. Zhao, L. Cao, H. Zhang, D. Kong, and G. Jin. 2015. Accurate calculation of Computer-generated holograms using angular-spectrum layer-oriented method. In *Optics Express*, Vol. 23.