Multi-GPU Acceleration of Optical Flow Computation in Visual Functional Simulation

Junichi Ohmura, Akira Egashira, Shunji Satoh, Takefumi Miyoshi, Hidetsugu Irie and Tsutomu Yoshinaga
Graduate School of Information Systems, University of Electro-Communications,
1-5-1 Chofugaoka, Chofu-shi 182-8585, Tokyo, Japan
Email: {ohmura, egashira}@comp.is.uec.ac.jp, {shun, miyoshi, irie, yosinaga}@is.uec.ac.jp

***Abstract*—Numerical simulation for visual processing of the human brain is one of time-consuming applications. This paper shows acceleration techniques for a simulation program of the visual processing. We parallelize convolution calculations, which are core operations, which the simulation program requests, on a GPU-accelerated PC cluster. Our implementation includes three improvement points. Firstly, we consider efficient data mapping onto global and shared memories[1] of the GPU. Secondly, multiple convolutions for the same input data are computed by each node's GPU, referred to as package execution. Finally, an input 2-dimensional image is divided into regions and convolutions for these regions are executed in parallel utilizing MPI (Message Passing Interface). Our experimental results show a linear speedup up to 12 nodes in the PC cluster for the convolution program. We also show the effects of the package execution and reduced communication on NVIDIA tesla C1060 and C2070, respectively.[2]**

I. Introduction

Human visual system has high competence for visual processing. The flexibility of the visual system sometimes exceeds an engineering pattern recognition algorithm. Understanding mechanism detail of the visual system enables to apply that to various beneficial application, such as avoiding accidents in robot visions. Whereas neurons of the visual system also have an instability property. One typical example is optical illusion. A lot of efforts have been done to analyze visual processing of humans.

Blue Brain Project[2] focuses on detailed neural system simulation. To describe neurons mathematically, this project adopts a compartment model. The model enables fine-grained simulation of visual neurons. However, it does not suit for macroscopic analysis of human visual system. Instead of the model, a linear model is suitable for the purpose to understand the visual system. The linear model is more coarse-grained and focuses on only input and output for the neurons. Actually various visual simulations apply the linear model[3]–[5].

Although the linear model is simple, it requires long simulation time in order to treat a large number of these models. In fact, a lot of human visual system researchers simplify these models and confine these models functions for saving the simulation time. We believe that parallel processing solves the problem. Therefore, our primary task is accelerating convolution, which is a fundamental calculation in the visual simulation, Our implementation accelerate convolution by utilizes parallel processing.

One of the typical accelerating techniques for the convolution is FFT(Fast Fourier Transform). FFT is applicable for a convolution between a target (input) and a kernel (fourier basis) under a condition that the kernel is unique temporally and spatially. However, the convolution kernels are different for each neuron in the linear model simulation, hence FFT is not applicable. To solve the problem, we parallelized the convolutions by utilizing MPI (Message Passing Interface) on a GPU-accelerated PC cluster in a previous studies[6].

This paper reports the implementation and improvements of high speed convolutions for movie data inputs. Our experimental results show good speedups for a program to calculate optical flow up to 16 nodes in the GPU-accelerated PC cluster. Our acceleration techniques have three folds. Firstly, in order to access large amount of data efficiently, we utilize shared memories to read target data which are more often used during the calculations compared with kernel data. Because of the low reuse ratio of the kernel data, we read them directly from a global memory and utilize a mean for storing multiple kernel data which enable an efficient use of wide memory bandwidth of GPUs. Secondly, multiple convolutions for the same input data are performed in a single device code (CUDA kernel program[1]) to minimize overheads for global memory accesses[3]. A reduction operation for the multiple convolutions is also performed in the device code to reduce communication between GPU and host CPU. Finally, an input 2-dimensional image is divided into regions and convolutions for the regions are executed in parallel by utilizing MPI (Message Passing Interface).

The rest of the paper is organized as this follows; Section II explains a basic mathematical model for the visual simulation. Section III describes our parallel implementation. Section IV and V show evaluation environments and the results, respectively. And section VI concludes this paper.

II. Basic mathematical model

For a linear model simulation of human visual functions, we adopt a mathematical model for the visual function. The

[1]Shared memory is a fast memory region on GPU. A location of the shared memory is significantly close to GPU's processing unit, like cache. This is a key of efficient data communication[1].

[3]Global memory is a memory region on GPU. This data transfer speed is slower than shared memory. Meanwhile, this data capacity is much larger than that of the shared memory.

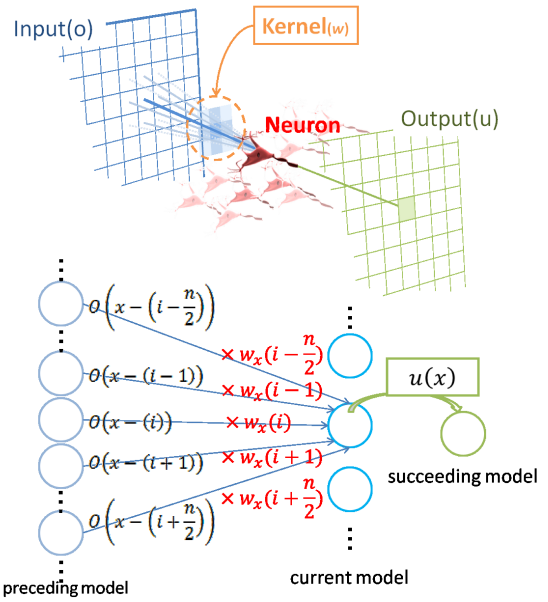

Fig. 1. A basic model of visual functions in the linear model.

mathematical model works the visual function as a neuron on human visual system. Fig. 1 shows a basic mathematical model for the linear model. Each model's function is realized by enormous mathematical models. Generally, each model receives input signals from preceding models, and transmits output signals to succeeding models.

As in Fig. 1, each model receives *n+1* inputs, $o_i : -n/2 \leq i \leq n/2$. The input signals are weighted by synapse load value (henceforth kernel). The output u from the model is calculated by the convolution with o_i (input) and w_i (kernel).

- n: A range of signal that each model can be received.
- w: Weight coefficient of synapse load value(kernel).
- o: Input to succeeding models(Output from the preceding models).
- u: Output to the succeeding models.

These linear model in Fig. 1 are based on equation (1).

$$u(x) = \sum_{i=-\frac{n}{2}}^{\frac{n}{2}} w_x(i)o(x-i) \tag{1}$$

When we extend the relation between w and o to calculate u in 2D space with taking account of temporal variation, u can be represented in equation (2).

$$u(x,y,t) = \sum_{\xi=-\frac{n}{2}}^{\frac{n}{2}} \sum_{\eta=-\frac{m}{2}}^{\frac{m}{2}} \sum_{\tau=0}^{l} w_{x,y}(\xi,\eta,\tau)o(x-\xi, y-\eta, t-\tau) \tag{2}$$

n, m, l: spatio temporal range(Kernel size)

In engineering image processing such as an edge detection, small kernel size, less than ten in each dimension, is usually enough to perform applications. On the other side, in the visual neuron simulation, each dimension size of a 3D kernel w is typically larger than ten. In addition, each neuron, whether relates to the same visual function or not, has different properties. In other words, kernel values for models differ from position to position, hence spatially different kernel $w_{x,y}$ is needed. Moreover, the kernel could be varied temporally, resulting the 3D kernel.

TABLE I
TOTAL SIZE OF ALL KERNEL DATA FOR VARYING FRAME SIZE AND KERNEL SIZE (NON-SEPARABLE KERNEL).

	$15 \times 15 \times 15$	$20 \times 20 \times 20$	$25 \times 25 \times 25$
320×240	0.97GB	2.29GB	4.47GB
512×384	2.47GB	5.86GB	11.44GB
640×480	3.86GB	9.16GB	17.88GB

TABLE II
TOTAL SIZE OF ALL KERNEL DATA FOR VARYING SPACE SIZE AND KERNEL SIZE (SEPARABLE KERNEL).

	$15 \times 15 \times 15$	$20 \times 20 \times 20$	$25 \times 25 \times 25$
320×240	13.2MB	17.6MB	22.0MB
512×384	33.8MB	45.0MB	56.3MB
640×480	52.7MB	70.3MB	87.9MB

The visual system simulation is expressed by multi-stage convolutions that require large-scale calculations. Additionally, the mathematical model needs an arithmetic operation per each spatio temporal coordinate in order to adjust and reduct these convolution outputs. This model calculates convolutions with different kernels for an input, and then summarizes the outputs from models in the kernel size into one.

Although such a summation is a simple operation, that often requires communication tasks and affects the system performance. Therefore, we also need to tune the system from the aspect of communication timing and data volume.

A. Mathematical model for Optical flow

Optical flow is the apparent motion vector of objects in a visual scene. A motion, relates between an eye and a visual scene, causes the motion vector. Human visual simulation often uses the motion vector. Similarly, our numerical simulation uses the motion vector.

Optical flow is represented by the motion vector which is calculated by Lucas-Kanade method. This numerical simulation uses input data, called $I(x,y,t)$. That is a brightness datum for the coordinate x, y and a time in a frame t. This simulation calculates three differentials of $I(x,y,t)$; $\frac{\partial I}{\partial x}$, $\frac{\partial I}{\partial y}$ and $\frac{\partial I}{\partial t}$. Using these differentials, calculates the optical flow with the follow equation (3) in the Lucas-Kanade method.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i w_i I_x(q_i)^2 & \sum_i w_i I_x(q_i) I_y(q_i) \\ \sum_i w_i I_x(q_i) I_y(q_i) & \sum_i w_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i w_i I_x(q_i) I_t(q_i) \\ -\sum_i w_i I_y(q_i) I_t(q_i) \end{bmatrix} \tag{3}$$

q_i: pixcel values in the movie data

Here, w_i is a weighting factor, which is calculated by the distance from the central pixel of a frame[7]. This weighting factor contributes more satisfactory simulation results.

On this study, we adopt the following TABLE I and II as our simulation subjects.

III. PARALLEL IMPLEMENTATION

A. Supporting multiple kernels

In our simulation, a communication between CPU and GPU happens per $frame$, followed by the optical flow computation on a GPU. This procedure is similar to an ordinary graphics processing on GPU. Input data are accessed via a shared memory, which are frequently used in the calculation with

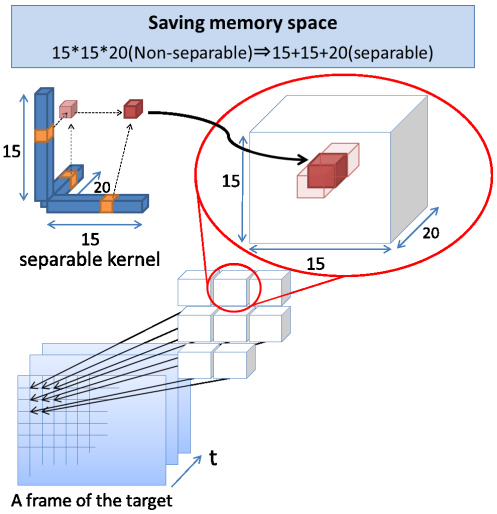

Fig. 2. Multiple separable kernel for each convolution.

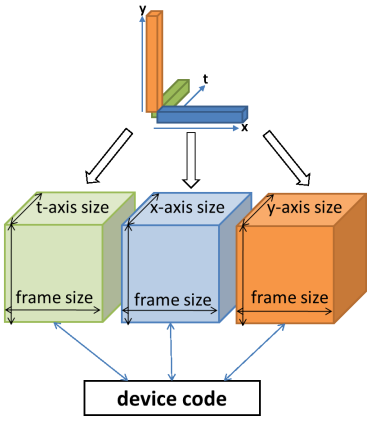

Fig. 3. A method to store and access to multiple kernels (separable).

spatially different coordinates. While, kernel data are read directly from a global memory because of their low reuse ratios. Nevertheless, our implementation can not efficiently use wide memory bandwidth of GPUs, so we utilize special means for storing kernel data.

As mentioned above, our target convolutions require 3D kernels, which are different spatially and temporally, hence these kernel data size becomes huge. Fortunately, we can reduce the data size by utilizing a separable kernel, which can be calculated with x, y and t coordinates. Fig. 2 illustrates how a 3D kernel is divided into three single dimensional arrays and kernel data for 3D coordinates are calculated by multiplications with three values of the 1D arrays. And Table I and II show the sizes of non-separable and separable kernels correspond to each image(frames) resolution. The size of each dimension varies from 15 to 25, respectively. As shown in Fig. 2 and Table I and II, the separable kernel requires only 15+15+20 memory spaces, which is much smaller than the non-separable kernel. In spite of additional calculations for the separable kernel, the reduction effect of these data size is more efficient than that. A device code on a GPU access kernel data and flame coordinates. The kernel data specifies an index of the separable kernel x, y or t. Such a data method

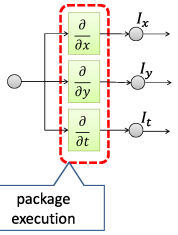

Fig. 4. A package execution of three convolutions.

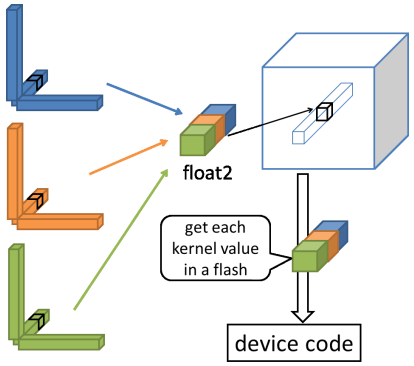

Fig. 5. Multiple kernel data access for a package execution of multiple convolutions.

is shown in Fig.3.

B. Supporting package execution for multiple convolutions

This package execution reduces the number of data accesses to the target (input data). As shown in Fig. 4, we pack multiple convolutions for the same input in a program on a single node. To confirm a relationship between the number of packed convolutions and the execution time, we conduct a experience. This experience executes convolutions on two GPU, which differ from each other in the number of these convolutions, and measures each execution time. Section V details this experience results.

Additionally, we use a data type of float2 which is defined in the NVIDIA CUDA Toolkit, as shown in Fig. 5. This data type enables to access two float values at a time.

Actually, float3 and float4 have been also defined by NVIDIA CUDA Toolkit. Unfortunately, the use of float3 degrades the calculation performance substantially. Also float4 can not shorten the execution time on C1060. float2 enables to shorten the execution time on C1060. Consequently our implementation executes a float2 data access two times instead of executing a float4 data access one time. All of our experiences results are results by a float2 data access.

C. Reducing communication volume

One of the main problem of our implementation is a communication task. Our implementation has a number of communication tasks, this degrades our system performance. To recover the disadvantage, we improve the communication part in our implementation. Targets of our improvement are a reduction operation, which our simulation system requires frequently, and inter node data exchanges.

Before an output operation, our simulation system executes a reduction operation. The reduction operation reduces three

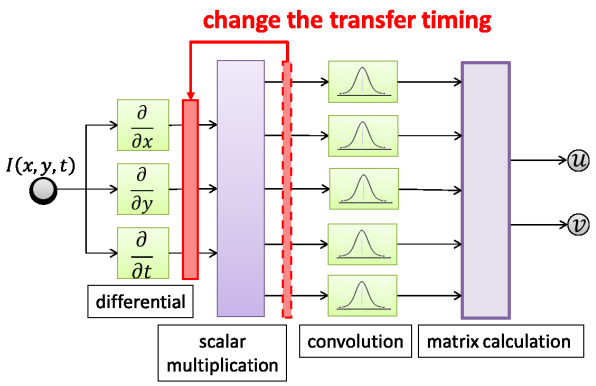

Fig. 6. Reducing communication volume in the optical flow computation.

output data to two output data per spatio temporal coordinate. The operation reduces communication data size on output and can be expected to reduce the communication time. The right side of Fig. 6, called matrix calculation, illustrates the reduction operation of the optical flow computation.

The another improvement to reduce communication data size is changing the timing of a data exchange, which normally required just before convolutions in our simulation system. These convolutions require values; I_x^2, I_y^2, I_xI_y, I_xI_t and I_yI_t. Because each nodes is assigned to a divided 2D image(flame) region, the calculation for boundary region requires data exchange these values with neighboring nodes. Our improvement moves the data exchange timing to immediately after differentials on each node. The variable of the data exchange are I_x, I_y and I_t. Our improvement reduces the amount of the data exchange to $\frac{3}{5}$, Fig. 6 illustrates our improvement.

D. Parallel computing on a GPU-accelerated PC cluster

For parallel computation on a GPU-accelerated PC cluster, we divide each frame into regions between the cluster's nodes, as shown in Fig. 7. This spatial data division reduces the kernel and target size per node, hence we can increase applicable target size. Since this spatial data division does not have temporal dependency in the target(input), our simulation system can support stream inputs.

The input and output data are assigned and reducted by a special node, called root node. It is high load for root node to assign and reduct these data all alone. In order to reduce root node load, as the below list shows, our implementation divides this assignment and reduction into 2 stage.

- The assignment part
  1) Root node assigns divided data to four nodes.
  2) These four nodes assigns each three node, which these four nodes are in charge of they.
- The reduction part
  1) These four nodes are received output data from these three nodes, and integrate the output data.
  2) The output data from these four nodes is integrated by root node.

Because dividing to four nodes is more effective for a concentration of communication tasks compared to dividing to 16 nodes at a time. Similarly, a whole procedure from inputs till output computational results is as follows;

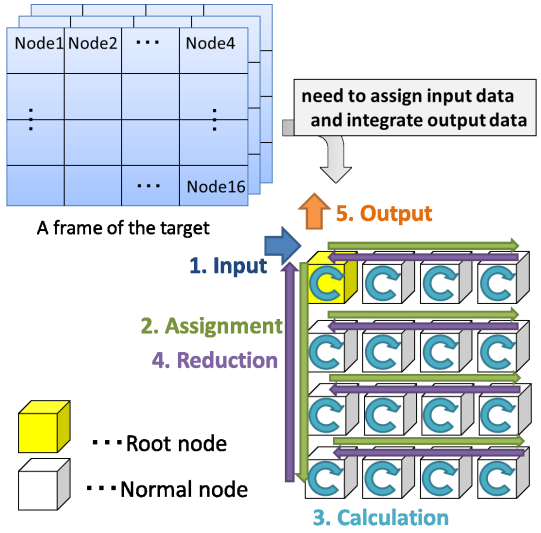

Fig. 7. Distributed processing on the multiple nodes.

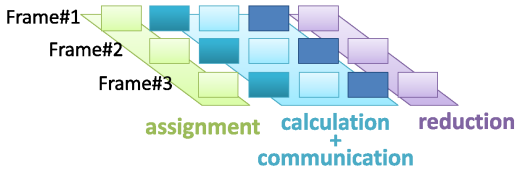

Fig. 8. An image of pipeline execution

Step 1 Root node reads input frames.
Step 2 Root node divides the frame data into regions and distribute them by inter-node communication.
Step 3 All nodes calculate on GPU.
Step 4 All nodes exchange intermediate data between GPU among neighboring nodes.
Step 5 All nodes compute remaining task on GPU.
Step 6 All nodes reduct results from each nodes.
Step 7 Root node outputs the results.

We execute one process per node, and the process invokes three threads. The first thread executes Step 1, and the second one executes from steps 2 to 6 in a pipeline fashion. Finally, the third one executes step 7. We explain the detailed pipeline process in the next section.

E. Pipeline processing

Input frames are processed by the five-stage tasks, from 2 to 6 stage, one by one in a pipeline fashion, as mentioned in the previous section.

Fig. 8 illustrates the pipeline processing behavior. Only one thread per process (or node) is responsible to this pipeline. These five-stage tasks for different frames are conducted in a certain iteration, simultaneously. Each communication stage tasks (2, 4 and 6 stage) perform MPI asynchronous communication (MPI_Isend, MPI_Irecv and MPI_Wait) and each computation stage tasks (3 and 5 stage) are performed by GPU asynchronously. Therefore, this pipeline process can cover data transfer latency by the communication task if the computation task is large enough.

Kernel data are not changed during the process, hence they

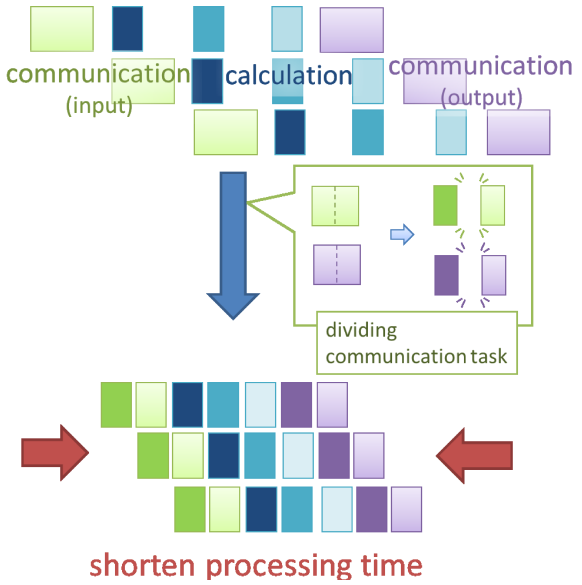

Fig. 9. An image of pipeline execution (with divided communication tasks).

are allocated in the global memory of GPU by an initialization task.

The communication data size is proportional to the input data size. Similarly, the calculation time is proportional to the input data size and the calculation time per one spatio temporal coordinate. The kernel size is $15\times15\times20$ in this study, and two operations (multiplication and addition) are required per spatio temporal coordinate. The total number of floating point operations are more than 40,000 per spatio temporal coordinate. More detailed consideration in these operations is mentioned in next page, Section IV. We estimate the computation and communication times per frame. The computation time is about 10^{-8} seconds with a 1 TFlops GPU, and that about the communication time is 4×10^{-9} seconds with double links of gigabit Ethernet. By this estimation, we can see that the computation time is shorter than the communication time. However, it does not consider any overheads. The computation performance may be also degraded by memory latency, and the communication time may incur messaging software overhead. Actually, our preliminary experiment showed communication latency was longer than the computation time, so we adjusted the pipeline process.

To prevent the communication overheads affect the overall performance, we divide each communication task into two smaller ones, one of them is for launching and the other for waiting a completion of a certain communication task. This is depicted in Fig 9.

IV. Evaluation

TABLE III and IV show hardware and software specification of nodes in our experiments. The first experiment considers effect of the package execution of multiple convolutions. This experiment is done on a single node PC in order to eliminate effect of inter-node communication. We use one of two kinds of GPU architectures, NVIDIA C1060 or C2070. This experience result is mentioned in Section V-A.

The second experiment executes parallel execution of the optical flow computation on a 16-node PC cluster. The node shown in TABLE III is used, and 16 nodes are connected with two Gigabit Ethernet links. Although this experiment adopts relatively slow interconnects, as above-mentioned, we carefully adjusted the pipeline process including computation and communication as we described in section III, so we can expect speedup by multiple node computation. Obtained speedups are discussed in the next section. For the inter-node communication, we use OpenMPI 1.4.1. This experience result is mentioned in Section V-B.

TABLE III
The hardware and software specification about the PC equipped with NVIDIA C1060.

CPU	Intel Xeon Quad-Core CPU W3520
Clock speed	2.67 GHz
memory	6 GB
GPU	NVIDIA C1060 (GT200 architecture)
Clock speed	1.296 GHz
Number of Streaming Processor	240
Peak performance	933 GFLOPS
Memory	4 GB
Memory bandwidth	102 GB/sec
Graphics bus	PCI Express x16 Generation 2.0
OS	CentOS 5.3
C Compiler	Intel C compiler 11.1
CUDA	CUDA Toolkit 3.2

TABLE IV
The hardware and software specification about the PC equipped with NVIDIA C2070.

CPU	Intel Xeon Quad-Core CPU W5667 x 2
Clock speed	3.07 GHz
Memory	24 GB
GPU	NVIDIA C2070 (Fermi architecture)
Number of CUDA Core	448
Clock speed	1.15 GHz
Peak performance	1.03 TFLOPS
Memory	6 GB
Memory bandwidth	144 GB/sec
Graphics bus	PCI Express x16 Generation 2.0
OS	Fedora 11
C Compiler	Intel C compiler 11.1
CUDA	CUDA Toolkit 4.0

The third experiment confirms performance of our simulation system. The experiment measures throughput for three size of input frames; 320×240, 512×384 and 640×480. This experience result is mentioned in Section V-C.

The last experiment confirms effect of dividing communication tasks. The experiment measures throughput for three size of input frames like the third experiment. However that case except dividing communication task, which is mentioned Section III-E, in order to confirm effect dividing communication task. This experience result is mentioned in Section V-D.

And for all experiments, the kernel size is $15\times15\times20$ as a moderate size for our linear model simulation of visual neurons. There are 41,400 of floating point operations per spatio temporal coordinate. The operation includes multiplication and addition with the kernel and target data. The amount of the operations are calculated by the equation (4).

$$\{\underbrace{(15\times15\times20)}_{\substack{\text{convolutions}\\\text{per each dimension}}}+\underbrace{(15\times20)+(15\times15\times20)}_{\text{calculations for kernel}}\}\times\underbrace{3}_{\text{dimension}}$$
$$=41,400 \qquad (4)$$

We use single precision operations in our simulation, since

they guarantee enough accuracy and ten times faster than double precision operations.

All of our experiment uses movie data as input data, which are popular in Internet. The frame size of these movie data are 320×240, 512×384, and 640×480. The vertical and horizontal size of those are multiples of 16, that are also popular for the movie data.

V. Results

A. Effect of the package executions for multiple convolutions

We conduct a experiment that executes package execution for multiple convolutions. Expected and measured execution times for the package execution are depicted in Fig. 10. $expected\ time$ in this graph is a estimation based on the execution time when calculating only one convolution. Expected time of more than two convolutions is estimated from the product of the number of convolutions and the execution time when calculating only one convolution.

In Fig. 10, we can notice that these execution times are always shorter than $expected\ time$. Although the peak performance of C2070 is higher than that of C1060, C1060 exceeds C2070 on the package execution of one or two convolutions. convolutions is one or two. It is inferred that data access overheads of C2070 are more than those of C1060, hence those of C2070 seem to lead these results.

To the contrary, C2070 exceeds C1060 on the package execution of three or four convolutions. It is might be said that this execution increases the amount of calculations and memory access, except for these memory access overheads, and weakens the influence of these overheads. In addition, C2070 shows the higher computational power and memory bandwidth. These reasons cause C2070 to exceed C1060 on that of three or four convolutions.

B. Effect of reducing communication volume

This section examines effect of reducing output data volume. In our simulation, outputs from convolutions are written into off-chip memory of the GPU. Then, they are transferred to main memory of the host via the PCI Express x16 bus. We can expect shortening the data transfer time by reducing the output data size.

We compare two cases on two GPU architectures; C1060 and C2070. The first one computes three convolutions then outputs three results. The second one computes three convolutions followed by 12 floating point operations per spatio temporal coordinate to reduce the output data size into $\frac{2}{3}$. Fig. 11 shows the results. Here, the first one is denoted as 3con. And the second one is denoted as 3con.+calc.

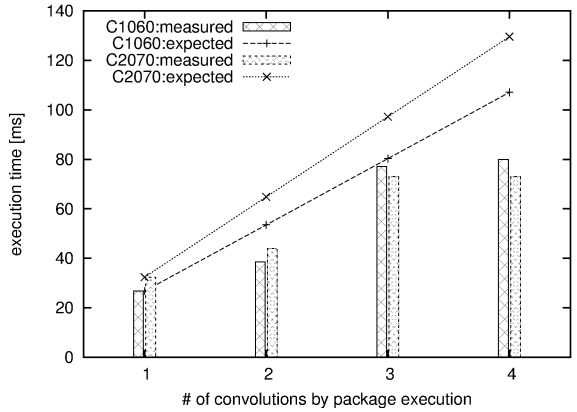

Fig. 10. Performance of the package execution (C1060 and C2070).

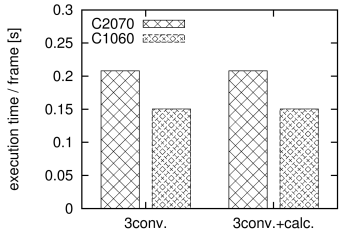

Fig. 11. Effect of reducing communication volume(1 node).

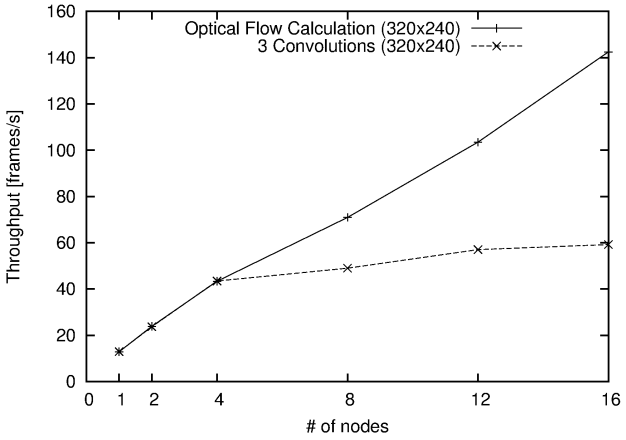

Fig. 12. A performance comparison of calculation for optical flow and 3 convolutions.

The execution time on both C2070 and C1060 is shortened by reducing the output data size, but the reduced time is limited considerably. Because the output data size is relatively small compared to the input sizes of the kernel and target. On the other hand, in a case of multi-node parallel computing, we recognize a performance improvement by effects of reducing communication volume. Fig. 12 shows a performance comparison of multi-node calculations between the optical flow and 3 convolutions.

From these results, we have recognized that reducting communication volume is profitable. This is happened because inter node communication as well as data reallocation is required for execution on multiple nodes. Hence, reducing the time of communication tasks is important. However, this execution is not overlapped with other computation nor communication tasks. This may degrade the overall performance, so overlapped execution is one of our future work.

C. Experimental results on the GPU-accelerated PC cluster

Fig. 13 shows the speedups of the optical flow computation when we increase the number of nodes in the GPU-accelerated PC cluster. Three frame sizes, 320×240, 512×384, and 640×480, are used.

For the smallest frame size(320×240), the throughput reaches 140 FPS. We obtained almost linear speedup up to 16 nodes for the smallest frame size. For other flame sizes, the throughput reaches 40 FPS when the flame size is

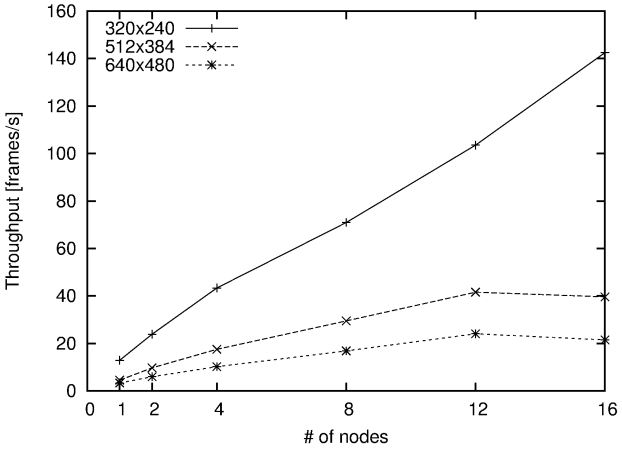

Fig. 13. Throughput for three size of input frames.

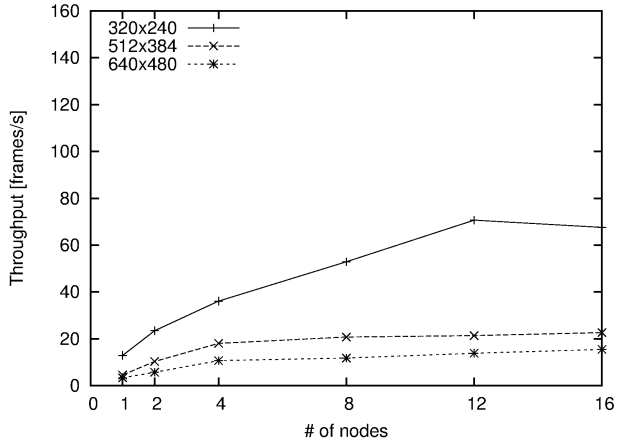

Fig. 14. Throughput for three size of input frames without dividing communication tasks.

512×384, and reaches 20 FPS when that is 640×480. But, throughputs for larger frame sizes are saturated at 12 nodes for the larger sizes (512×384, 640×480). In cases on 16 nodes, communication sometimes can not catch up with calculations on GPU because of messaging overhead in MPI. Even so, in the case of 512×384, 40 FPS, which is enough to simulate using an movie file with ordinary frame rate, is achieved. We also attained 20FPS performance for a movie with 640×480 frame sizes. An idea for further improvement is tuning task timing among multiple nodes utilizing OpenMP.

The reason of the saturation is that the communication task can not be covered with the calculate task. The amount of the communication task is much larger than that of the calculation task when the flame size is 512×318 or 640×480. And dividing an input data between each node reduces the calculation task mainly. Because the data dividing can not reduce overheads by the communication task. These two reason prevent from covering the communication task with the calculation task.

D. Effect of dividing communication tasks

As we described in Section III-E, firstly we designed the five-stage pipeline process, then redesigned it in seven-stage pipeline by dividing communication tasks into smaller ones. Figure 14 shows the results of original five-stage pipeline for the same experiments with figure 13. It is clear that new seven-stage pipeline shows good scalability, thanks for reducing communication overhead.

VI. Conclusion

This paper reports implementation of a program for visual neuron simulation using a GPU-accelerated PC cluster. For the optical flow computation using the linear model of visual neurons, we have confirmed speedups up to 12 nodes for all three sizes of input frames we have examined. Especially for the smallest frame size (320×240), we obtained almost linear speedup up to 16 nodes, and over 140FPS. These results are derived not only from efficient data allocation in GPUs, but also designing pipeline process for input frames with taking account of valance between the computation and communication.

In this study, we used separable kernel to reduce data size required to store on the GPU. Supporting visual functions which need to deal with the non-separable kernels is our future work. Further improvements include timing tuning among thread on a node and latency hiding for the inter-node communication.

Finally, we also plan to simulate larger and more complex neural networks to help investigating realistic visual functions of humans.

Acknowledgments

This research is supported in part by JSPS Grants-in-Aid for Scientific Research (C) No.22500042.

References

[1] NVIDIA, “Cuda toolkit,” http://developer.nvidia.com/cuda-toolkit-sdk.

[2] H. Markram, “The blue brain project,” *Neuroscience*, vol. 7, pp. 153–160, 2006.

[3] Z. Li, “A neural model of contour integration in the primary visual cortex,” *Neural Computation*, vol. 10, pp. 903–940, 1998.

[4] I. Motoyoshi and F. A. A. Kingdom, “Differential roles of contrast polarity reveal two streams of second-order visual processing,” *Vision Research*, vol. 47, pp. 2047–2054, 2007.

[5] S. Satoh and S. Usui, “Computational theory and applications of a filling-in process at the blind spot,” *Neural Networks*, vol. 21, pp. 1261–1271, 2008.

[6] T. Saitou, S. Satoh, J. Ohmura, T. Miyoshi, H. Irie, and T. Yoshinaga, “Parallel numerical simulation for the linear model of visual neurons with mpi,” *Information Processing Society of Japan (IPSJ)*, vol. 2011-HPC-129, no. 4, pp. 1–8, mar 2011, (in Japanese).

[7] L. B.D and K. T, “An iterative image registration technique with an application to stereo vision,” *Proceedings of the Seventh International Joint Conference on Artificial Intelligence(IJCAI-81)*, pp. 674–679, 1981.