Springer

KSME

Journal of Mechanical Science and Technology 29 (7) (2015) 2631~2637
www.springerlink.com/content/1738-494x(Print)/1976-3824(Online)
DOI 10.1007/s12206-015-0508-0

Optimization of operating and assembling mass properties of solid elements on heterogeneous platforms using OpenCL framework †

Ji-Hyun Jung and Dae-Sung Bae*

Department of Mechanical Engineering, Hanyang University, Ansan, Gyeonggi-do, 425-791, Korea

(Manuscript Received December 4, 2014; Revised March 18, 2015; Accepted April 28, 2015)

Abstract

Model sizes have increased significantly in the fields of engineering and scientific computation. Some additional computing devices such as GPU, accelerators and co-processors have been applied to improve the computation performance. This paper presents several strategies to optimize the computation performance. The first strategy is to combine a computation unit with multiple of 4-tetrahedrons to support AVX vectorization. The second strategy is to utilize a GPU device. Several techniques are proposed to reduce the time for data exchange between host and GPU memory spaces. The proposed techniques are implemented by using OpenCL framework. The mass property of many solid finite elements is calculated and its computation performances on various computation platforms are compared. Numerical experiments showed that computation performance has improved 26.47 times on CPU and 6.95 on GPU, compared to the version without using the proposed techniques.

Keywords: GPU; Heterogeneous platform; Mass properties of solid elements; OpenCL framework

1. Introduction

As modern large-scale science and engineering problems increase exponentially, it is getting harder to conduct a large-scale computation, depending on the existing CPU device due to power consumption and heat generation problem. As an example, the fastest super computer registered on the TOP 500 in November 2013 is Tianhe-2 from China with 16000 computer nodes, each comprising two CPU processors and three accelerator ships, counting a total of 3120000 cores, theoretical peak performance of 54.9 petaflops [1]. If the system constructs the same performance supercomputer using only CPU, it will roughly need 8 times more CPU and 3.5 times more power consumption. For this reason, in terms of large-scale science and engineering computing, it is essential to choose additional computing devices like GPU, Accelerator, or Co-processor along with CPU. Some of researchers noticed that GPUs are an ideal solution for a data-parallel computation and they are attempted to utilize the device to analyze a combination of a multi-body system and particles [2, 3].

It is necessary to calculate the exact mass properties (Mass, mass center, mass moment of inertia) when modeling of flexible body in MBD (Multi-body dynamics) CAE software [4]. When the mass properties are implemented on CPU naively, it takes a few minutes to calculate for several million elements. One way to reduce the computing time is to utilize various computing devices.

In this study, we apply OpenCL (Open computing language) framework to support only CPU as well as other computing devices [5-8]. Sec. 2 presents a naive OpenCL version which is able to execute on various devices. Any optimization has not been applied. Sec. 3 presents several general optimizations that have an effect on both CPU and GPU devices. Secs. 4 and 5 present special optimizations for each device. Sec. 6 explains the numerical performance results. Sec. 7 concludes the paper and presents a future work.

2. Mass properties

The mass, mass center and moment of inertia which is called mass properties must be calculated for the finite element calculation process. A body is modeled by many finite solid elements. The size of elements must be small enough to calculate accurate mass properties of a complex geometry body. As the size of elements is smaller, the number of elements becomes larger, which requires more computation time.

In this study, the research investigated six kinds of solid elements such as SOLID4, SOLID6, SOLID8, SOLID10, SOLID15, and SOLID20. Fig. 1 illustrates the relations of a

*Corresponding author. Tel.: +82 2 2135 7097, Fax.: +82 2 2135 7099
E-mail address: dsbae@hanyang.ac.kr

† This paper was presented at the Joint Conference of the 3rd IMSD and the 7th ACMD, Busan, Korea, June, 2014. Recommended by Guest Editor Sung-Soo Kim and Jin Hwan Choi

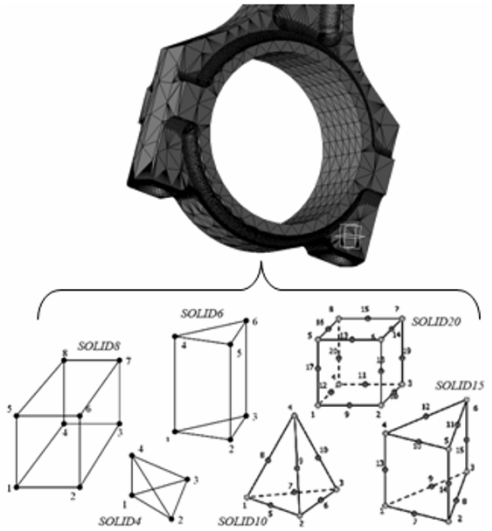

Fig. 1. Flexible body and various solid elements constituting it.

complex geometry body and well-known six elements.

2.1 Mass properties of tetrahedron

All types of elements can be decomposed into tetrahedrons, e.g. 6 tetrahedrons composes a SOLID8 element. Therefore, if exact mass properties of tetrahedrons can be calculated, the mass properties of all elements can be obtained. The mass and mass center of an element can be calculated as follows.

$$m_e = \frac{1}{6}\left|\{(\mathbf{p}_2-\mathbf{p}_1)\times(\mathbf{p}_3-\mathbf{p}_1)\}\cdot(\mathbf{p}_4-\mathbf{p}_1)\right|\cdot\mu \tag{1}$$

$$\mathbf{p}_e = \frac{1}{4}(\mathbf{p}_1+\mathbf{p}_2+\mathbf{p}_3+\mathbf{p}_4) \tag{2}$$

where, $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ and $\mathbf{p}_4$ are positions of tetrahedron points, $\mathbf{p}_e$ is a center of the 4 points and μ is its density.

Especially, the mass moment of inertia tensor of the 3-D tetrahedron is introduced with the explicit formula [9]. The mass moment of inertia $\mathbf{J}_e$ of the tetrahedron in a domain of $\mathbf{D}$ with respect to the reference frame of x, y and z centered at Q is defined as following equations.

$$\mathbf{J}_e = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{yx} & J_{yy} & -J_{yz} \\ -J_{zx} & -J_{zy} & J_{zz} \end{bmatrix} \tag{3}$$

where,
$$J_{xx}=\int_D \mu(y^2+z^2)dD \quad J_{xy}=\int_D \mu\, xy\, dD$$
$$J_{yy}=\int_D \mu(x^2+z^2)dD \quad J_{yz}=\int_D \mu\, yz\, dD$$
$$J_{zz}=\int_D \mu(x^2+y^2)dD \quad J_{zx}=\int_D \mu\, zx\, dD\,.$$

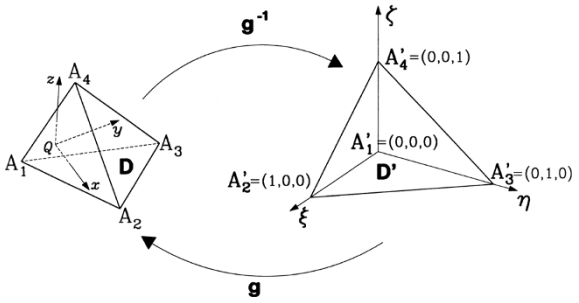

Fig. 2. Axes transformation.

As shown in Fig. 2, the coordinate transformation of $\mathbf{g}$ is used to calculate the above integrals more easily.

Transformation of $\mathbf{g}^{-1}$ is used to transform the tetrahedron in the domain of $\mathbf{D}$ into a normalized tetrahedron in the domain of $\mathbf{D}'$. Generic function of $f(x, y, z)$ can be written as:

$$\begin{aligned}\mathbf{J}_e &= \int_D f(x,y,z)dD \\ &= \int_{D'} f\left[x(\xi,\eta,\zeta),\, y(\xi,\eta,\zeta),\, z(\xi,\eta,\zeta)\right]\left|DET(J)\right| dD.\end{aligned} \tag{4}$$

Since $\mathbf{D}'$ is normal with respect to the (ξ,η)-plane and the projection of $\mathbf{D}'$ on plane (ξ,η) is normal with respect to the ξ-axis, one has:

$$\mathbf{J}_e = \left|DET(J)\right|\cdot\int_0^1 d\xi\int_0^{1-\xi} d\eta\int_0^{1-\xi-\eta} f\left[x,\, y,\, z\right]d\zeta\,. \tag{5}$$

For example, J_{xx} can be obtained by Eq. (6).

$$J_{xx} = \mu\cdot\left|DET(J)\right|\cdot\int_0^1 d\xi\int_0^{1-\xi} d\eta\int_0^{1-\xi-\eta}\left[y^2+z^2\right]d\zeta\,. \tag{6}$$

2.2 Assembling mass properties

After getting the mass properties of tetrahedrons, they are assembled to obtain the mass properties of the element and body through two steps. The mass properties of all tetrahedrons which belong to the element are assembled to get the mass properties of element. The mass properties of all elements are then assembled to get those of body. The assembly of mass properties is written as following equations and this process is shown in Fig. 3 briefly.

$$M = \sum_{k=1}^{n} m_k \tag{7}$$

$$\mathbf{P} = \left(\sum_{k=1}^{n} m_k\cdot\mathbf{p}_k\right)/M \tag{8}$$

$$J_{ij} = \sum_{k=1}^{n}\left[\,\mathbf{J}_{k,ij} + m_k(\mathbf{d}_k{}^T\mathbf{d}_k\cdot\delta_{ij} - \mathbf{d}_{k,i}\mathbf{d}_{k,j})\right] \tag{9}$$

where, δ_{ij} is a kronecker delta.

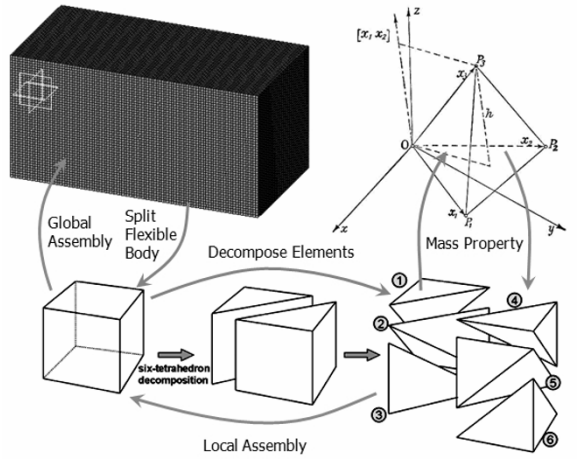

Fig. 3. Decomposing and assembling mass properties of SOLID8 elements.

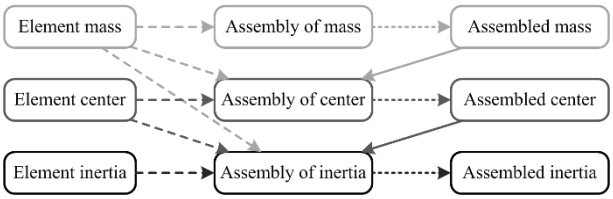

Fig. 4. Data dependencies while assembling mass properties.

As shown in Fig. 4, the data dependencies among mass properties are during the assembly process. This will make a problem in the reduction process and this paper proposes how to avoid the problem.

2.3 Basic implementation

As shown in Fig. 5, the mass properties are implemented by using naive version of OpenCL.

3. General optimization

We replaced power function and division operation with a multiply operation to reduce computation time without changing algorithms. It is desirable to replace power functions with explicit multiplications for integer powers of real number to improve performance. The divide operation invokes a hazard since it does not support pipelining. To minimize this hazard, repeated divide operation proceeds only once and saves the inverted value. Then, the divide operations change the multiply operations using the value. An applying general optimization is depicted in Fig. 6.

4. Optimization for multi-core CPU device

In this section, we will describe the implementation and optimization process for modern multi-core CPU device. Only 10 years ago, computing powers improvement of CPUs had depended on the clock speed but it caused power consumption and heat generation to increase explosively. To solve this

```
////////////// BEGIN OpenCL HOST PART //////////////
// Set kernel input arguments
clSetKernelArg(kernel_calc_solid, 0, sizeof(int), (void*)
   &count_assem);
clSetKernelArg(kernel_calc_solid, 1, sizeof(cl_mem), (void*)
   &memobj_prop.memobj_conn_tetra);
clSetKernelArg(kernel_calc_solid, 2, sizeof(cl_mem), (void*)
   &memobj_prop.memobj_density);
clSetKernelArg(kernel_calc_solid, 3, sizeof(cl_mem), (void*)
   &memobj_prop.memobj_pos_solid);

// Set kernel output arguments
clSetKernelArg(kernel_calc_solid, 4, sizeof(cl_mem), (void*)
   &memobj_prop.memobj_mass);
clSetKernelArg(kernel_calc_solid, 5, sizeof(cl_mem), (void*)
   &memobj_prop.memobj_center);
clSetKernelArg(kernel_calc_solid, 6, sizeof(cl_mem), (void*)
   &memobj_prop.memobj_inertia);

// Execute kernel
clEnqueueNDRangeKernel(cmd_queue, kernel_calc_solid, 1, NULL,
   &global_size, &local_size, ...);

// Read mass properties of elements
clEnqueueReadBuffer(cmd_queue, memobj_mass, CL_TRUE, NULL,
   size_mass, pmass, ...);
clEnqueueReadBuffer(cmd_queue, memobj_center, CL_TRUE,
   NULL, size_center, pcenter, ...);
clEnqueueReadBuffer(cmd_queue, memobj_inertia, CL_TRUE,
   NULL, size_inertia, pinertia, ...);

// Assemble mass properties of all elements
assemble_mass_props_host(count_solid, pmass_e, pcenter_e,
   pinertia_e, pmass, pcenter, pinertia);
////////////// END OpenCL HOST PART //////////////

////////////// BEGIN OpenCL DEVICE PART //////////////
// Get thread variables
size_t global_id = get_global_id(0);
density = pdensity[global_id];

// Get mass properties of tetrahedron
for (i=0; i<COUNT_TETRA_SOLIDX; i++) {
   conn_tetra = pconn_tetra[i];
   get_tetra_mass_props(&density,
      &pos_ele[ conn_tetra.x ], &pos_ele[ conn_tetra.y ],
      &pos_ele[ conn_tetra.z ], &pos_ele[ conn_tetra.w ],
      &mass_t [i], &center_t[i], &inertia_t[i]);
}

// Assemble mass properties of tetrahedrons
assemble_mass_props_device(COUNT_TETRA_SOLIDX,
   mass_t, center_t, inertia_t, &mass, &center, &inertia);
////////////// END OpenCL DEVICE PART //////////////
```

Fig. 5. Basic implementation of mass properties.

```
// Before replacing pow functions and a division operation with multiple operation
pinertia_tetra_iter[0] += pow(A1.p[1], 2.0);
...
pinertia_tetra_iter[0] /= 60.0;

// After replacing pow functions and a division operation with multiple operation
#define INV_60 0.016666666666666666666666666666666666666666666666
...
pinertia_tetra_iter[0] += A1.p[1] * A1.p[1];
...
pinertia_tetra_iter[0] *= INV_60;
```

Fig. 6. Part of applying an optimal code.

problem, CPU manufacturers typically have integrated the cores onto a single integrated circuit die. However, due to synchronizing data among the CPU cores, the number of it is not able to increase continuously. Recently, a way of improving computing powers by increasing in SIMD register size has been in the spotlight.

4.1 Avoiding needless data copy

OpenCL Framework requires the data copy between host memory and device memory in order to support various heterogeneous platforms. However, CPU Device is host device and the source and the destination of the data to copy is the same. Therefore, the explicit data copy process can be skipped.

4.2 Vectorization for CPU device

The CPU device used in this study supports a 256-bit intrinsic set extension called Intel AVX (Advanced vector extensions) [10]. It is able to process 4-double precision elements simultaneously. In order to fully utilize the register size, data process unit of the program should be modified properly. Each solid element consists of different numbers of tetrahedrons and most of them are not a multiple of 4.

There are two methods to consider an AVX Intrinsic, one is to set a process unit to 4-elements and the other is to rearrange 4-tetrahedrons with several adjacent elements. Our experiments have shown that the former method the increase on register pressure often deteriorates performance [11]. Hence, we choose the latter method, it binds some adjacent elements to implement efficient vectorization and we named it "grouping". For example, a SOLID6 element consists of 3 tetrahedrons. If an AVX vectorization is implemented without grouping, it losses 25% computing powers because only three double precisions of register size are filled. If we set adjacent 4 SOLID6 elements and compute 4-tetrahedral units belonging to the elements, there is no loss of a computing power. Fig. 7 illustrates an example how to grouping on SOLID6 elements

Table 1. Grouping properties in accordance with element types.

	No. of tetra.	AVX loss ratio	No. of grouping
SOLID4	1	75%	4
SOLID6	3	25%	4
SOLID8	6	25%	2
SOLID10	8	0%	1
SOLID15	15	6.25%	4
SOLID20	22	8.33%	2

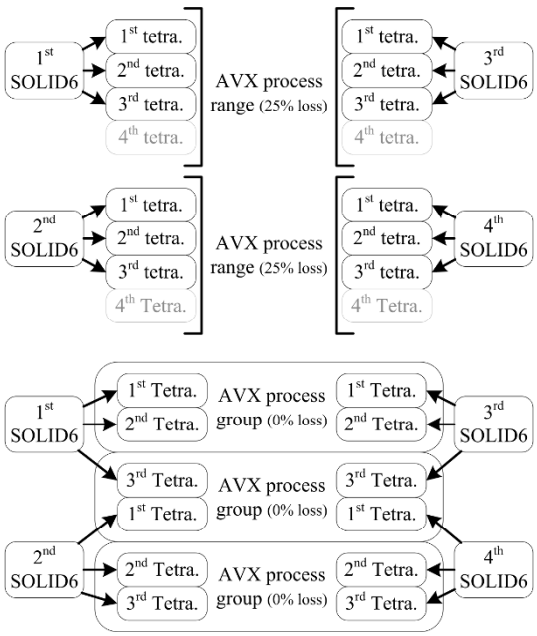

Fig. 7. Grouping of SOLID6 elements for AVX vectorization.

for AVX vectorization and the number of tetrahedrons consists of each element, computing power loss when applying AVX vectorization without grouping and grouping count to overcome the loss are shown in Table 1.

4.3 Reduction process

A reduction process is to reduce a quantity of mass properties to a particular unit for each element. Although it is designed for a GPU device to minimize transmission data from Device to Host memory space, there is also a performance improvement in a CPU device.

This effect comes from cache reuse during recursive reduction operation; hence it minimizes a cache miss and a request data from physical memory. More information on the reduction process will be explained in the next section.

5. Optimization for many-core GPU device

In this section, we will explain the implementation and optimization for modern many-core GPU device. Originally, the GPU's advanced capabilities were primarily used for 3D game

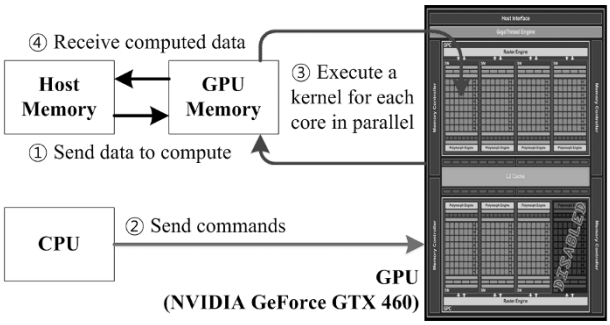

Fig. 8. Overall computing process using GPU device.

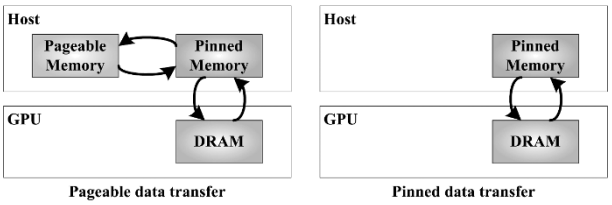

Fig. 9. Effects of pinned memory.

rendering. But now those capabilities are being utilized more broadly to accelerate computational workloads. Structurally, a CPU is composed of an only few cores with lots of cache memory that can handle a few software threads at a time. In contrast, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. Therefore, a GPU device is suitable for a data parallelism.

Computation using GPU device requires 4 steps. (1) After copying data from host memory to GPU memory, (2) CPU instructs the process to GPU and (3) GPU executes parallel in each core. (4) Copying the result from GPU memory to CPU memory is the last step. This process is illustrated in Fig. 8. In this whole process, the part with the biggest bottleneck is a data transfer process between host and GPU memory. Thus, minimizing the time consumption on this process is a core of GPU computation.

5.1 Pinned memory

The role of CPU device is to manage the whole process of host program and the role of GPU device is to assist a part of host program. Since CPU and GPU do not share memory space, it is necessary to exchange data for them to perform. CPU device is responsible for the data transmission. Generally, Host operation system needs a large virtual address space in order to guarantee a bigger space than physical memory. This is available when the disk replaces physical memory.

When the Host needs the saved data in this space, it reads the data on the disk and transfers it into physical memory. This is called non-locked memory. The non-locked memory needs to access every single page of the non-locked memory, copy it into pinned buffer and pass it to the direct memory access (DMA). However, with today's memories, the use of virtual

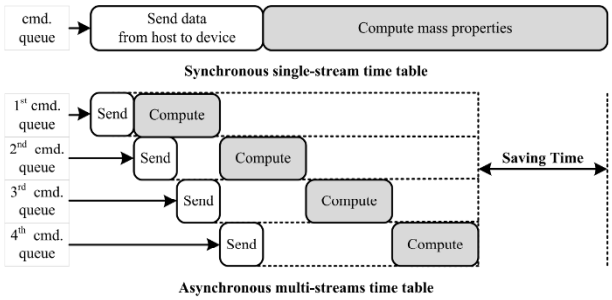

Fig. 10. Time swapping using asynchronous multi-streams.

memory is no longer necessary for many applications which will fit within the host memory space. In those cases, it is more convenient to use page-locked (Pinned) memory which enables a DMA on the GPU to request transfers to and from the host memory without the involvement of the CPU. In other words, locked memory is stored in the Host physical memory, so the GPU can fetch it without the help of the host [12].

5.2 Asynchronous multi-streams

The previous optimization item is able to raise data transfer speed, but the time itself cannot be hidden. However, computing and data transfer time can be overlapped if a partitioned data is transferred and computed asynchronously. This is similar to CUDA stream [13], and inspired by it we realized the function so that it can be suitable for OpenCL. For example, let's assume input data is quartered and transferring and computing commands are issued in each command queue. The first transfer time cannot be hidden but the others can. It will be the most effective when the transfer time is almost the same as the computing. The effect of asynchronous multi-streams is depicted in Fig. 10.

5.3 Reduction of the result data

If the second optimization is to hide the transfer time, this time, we tried to reduce a quantity of transferring data itself.

Although it is impossible to reduce an input data, it is possible to minimize the transferring output data significantly if we use the feature that is able to assemble 2 or more mass properties. This is called reduction process.

A general reduction operation overwrites original values, but with mass properties, original values should be taken into account. Mass properties should be reduces recursively by assigning data space additionally. The process is described in Fig. 11.

6. Results

A Table 2 shows hardware and software specification to perform benchmark in this study.

Based on above hardware and software specifications, we conducted a test regarding all cases. The application goes

Table 2. System description used for this study.

CPU	Intel Core i7-3770K / 3.5GHz / 4-Cores / HT, TB OFF
RAM	Samsung DDR3 PC3-12800 8GB x 4 (32GB)
GPU	NVIDIA GeForce GTX 460 / 1GB / PCI Express 2.0
OS	Windows 8.1 Enterprise K x64
Compiler	Intel C++ Compiler XE 14.0
Intel OpenCL	CPU only Runtime Package 2013
Nvidia OpenCL	NVIDIA Graphic Drive 334.89

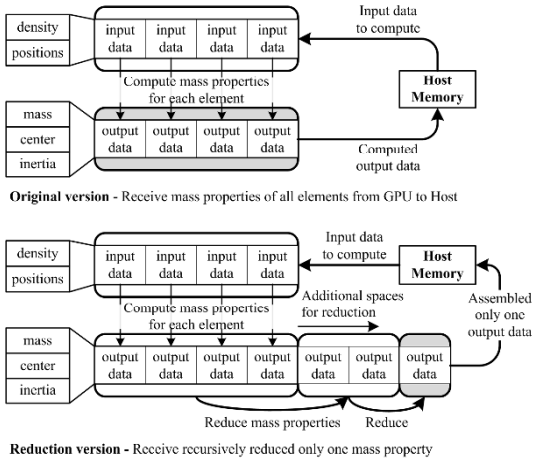

Fig. 11. Reduction of the mass properties to minimize the output data.

through the following time steps during its run: (1) Initializing OpenCL objects (Platform, device, context, program, kernel) (2) Reading the input data (Positions and connectivity of nodes, densities of elements) (3) Allocating memory object of compute device (4) Copying the input data from host to device memory space (5) Running the kernel on the device (6) Copying data back from device to host (7) Assembling the retuned data using CPUs (8) De-allocating all OpenCL objects and output the results. We are interested in only substantive computing time. Therefore, only four steps from (4) to (7) are used to measure the time and the other steps are excluded. We avoided the power function on all performance tests except for a pure Naive version since its effect is too obvious.

The test condition is to measure time of about 40 million (40000000) solid elements with the same position but different densities for each element. The result is arranged in Tables 3 and 4. According to the result, the more complicated solid type is, the better speed up ratio can be obtained. At the naive version, GPU device is much faster than CPU device. However, this trend is reversed after applying all optimizations. Effective optimization items are different for each device.

On CPU device, no copy and AVX items contribute to a performance improvement. This represents that it needs a vectorization with a proper vector data type and its hint attribute like "_attribute__ ((vec_type_hint(double4)))" to disable implicit kernel vectorization of OpenCL compiler [12].

On GPU device, pinned memory, stream and reduction optimizations are more effective. It recalls that minimizing the time consumption to exchange data between Host and GPU memory is an important factor of GPU computation. The time decrease of second order elements by Stream is more obvious than linear order elements, which comes from overlapping the time between computing and transferring an input data. On the other hand, the effect of reduction is nearly similar regardless of element types. The reason is that the number of all elements for the test is equal.

Table 3. Optimization effect on multi-core CPU device.

Time (sec)	SOLID4	SOLID6	SOLID8
	SOLID10	SOLID15	SOLID20
Naïve	9.504	28.960	60.115
	65.202	130.192	209.335
No pow + naive	4.619	6.704	10.449
	13.378	19.807	25.851
No pow + no division	4.727	6.384	10.273
	11.983	19.312	22.565
No pow + no copy	2.259	2.944	5.580
	7.228	11.001	16.252
No pow + reduction	3.573	5.200	8.910
	11.144	18.498	24.039
No pow + AVX	3.210	5.818	7.957
	11.279	17.709	20.267
All optimizations	*0.373*	*1.085*	*2.085*
	2.791	*5.058*	*7.431*
Speedup over naive	**25.48x**	**26.69x**	**28.83x**
	23.36x	**25.74x**	**28.71x**

Table 4. Optimization effect on many-core GPU device.

Time (sec)	SOLID4	SOLID6	SOLID8
	SOLID10	SOLID15	SOLID20
Naïve	6.789	14.374	25.035
	32.435	57.640	81.728
No pow + naive	3.962	5.477	7.800
	9.151	14.200	18.258
No pow + no division	3.918	5.442	7.613
	8.649	13.220	16.891
No pow + pinned	2.259	4.994	6.789
	8.105	12.382	16.807
No pow + stream	3.181	3.972	4.982
	5.519	8.056	10.352
No pow + reduction	2.694	5.200	6.549
	7.592	12.398	17.068
All optimizations	*1.946*	*3.104*	*4.503*
	4.058	*5.967*	*7.887*
Speedup over naive	**3.49x**	**4.63x**	**5.56x**
	7.99x	**9.66x**	**10.36x**

7. Conclusions and future work

The purpose of this study is to optimize the computing time

of mass properties with various heterogeneous devices. We were able to achieve a significant effect using these optimization strategies.

(1) Five optimization strategies are attempted for CPU and GPU. No power function, no divide operation, no copy, reduction process and AVX vectorization are attempted for the CPU device. And no power function, no divide operation, pinned memory, multi-streams and reduction data are tried for the GPU device.

(2) The most effective strategies are to adjust a computation unit to fit a SIMD register size and to avoid unnecessary copy of data on latest CPUs.

(3) To minimize and hide a time of data transfer between host and device memory space is the most effective strategies on GPUs.

(4) We obtained speed up factors of over 26.47 on CPU and 6.95 on GPU as an average of the latest results for six element types against the naive version for each computing device.

(5) When data complexity is higher, the computing time of the GPU device is similar with that of the CPU. It shows that GPUs could be alternative computing device on CPUs.

Based on this successful optimization, we are planning to proceed to compute a work not only among the same devices but also among different devices simultaneously. The total computing time depends on the slowest device. It will be resolved by considering the characteristics of each device and load-balancing.

Acknowledgment

This work was supported by VirtualMotion, Inc, Korea.

References

[1] *TOP500 : http://www.top500.org/.*

[2] H. Y. Jung, C. W. Jun and J. H. Sohn. GPU-based collision analysis between a multi-body system and numerous particles, *Journal of Mechanical Science and Technology*, 27 (4) (2013) 973-980.

[3] C. W. Jun and J. H. Sohn, Numerical efficiency of CUDA based parallel programming for dynamic analysis of multi-body systems with multi-joints and multi-force elements, *Journal of Mechanical Science and Technology*, 27 (12) (2013) 3565-3570.

[4] *DAFUL 4.2 User's Manual*, Virtual Motion, Inc. (2013).

[5] *OpenCL : http://www.khronos.org/opencl/.*

[6] A. Munshi, B. Gaster, T. G. Mattson and D. Ginsburg, *OpenCL programming guide,* Pearson Education (2011).

[7] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry and D. Schaa, *Heterogeneous Computing with OpenCL*, Newnes (2011).

[8] *OpenCL : http://www.khronos.org/opencl/.*

[9] F. Tonon, Explicit exact formulas for the 3-D tetrahedron inertia tensor in terms of its vertex coordinates, *Journal of Mathematics and Statistics*, 1 (1) (2005) 8-11.

[10] *Intel OpenCL SDK v1.1*, Intel (2013).

[11] R. Karrenberg and S. Hack, Improving Perfor-mance of OpenCL on CPUs, *CC'12 Proceedings of the 21st international conference on Compiler Construction*, Springer Berlin Heidelberg (2012) 1-20.

[12] *http://en.wikipedia.org/wiki/CUDA_Pinned_memory*, Wikipedia.

[13] *CUDA Programming Guide*, NVIDIA (2011).

[14] J. Shen, J. Fang, H. Sips and A. L. Varbanescu, Performance gaps between OpenMP and OpenCL for multi-core CPUs, *Parallel Processing Workshops (ICPPW), 2012 41st International Conference*, IEEE (2012) 116-125.

Ji-Hyun Jung received the B.S. and M.S. in Mechanical Engineering from Hanyang University in 2007 and 2010, respectively. He is now enrolled in the Doctorial course in Mechanical Engineering at Hanyang University. His current research interests are heterogeneous computing with GPU and co-processor and improvement of mechanical software.

Dae-Sung Bae received the M.S. and Ph.D. from the university of Iowa in 1983 and 1986, respectively. His currect research interests are meshfree method and parallel processing in the field of mechanical and structural dynamics.