Research on SPH Parallel Acceleration Strategies for Multi-GPU Platform

Lei Hu, Xukun Shen, and Xiang Long

Beihang University, Beijing 100191, PRC
hu_lei_1989@163.com,
{xkshen,long}@buaa.edu.cn

Abstract. This paper proposes an acceleration strategy for SPH on single-node multi-GPU platform. First the acceleration strategy for SPH on single-GPU is studied in conjunction with the characteristics of architecture. Then the changing pattern of SPH's computation time has been discussed. Based on the fact that the changing pattern is rather slow, using a simple dynamic load balancing algorithm an acceptable load balance is obtained on multi-GPU. Finally, an almost linear speedup is achieved on multi-GPU by further optimizing dynamic load balancing algorithm and communication strategy among multiple GPUs

Keywords: SPH, multi-GPU, dynamic load balancing, communication optimization.

1 Introduction

Smoothed Particle Hydrodynamics(SPH) is a mesh-free Lagrangian method based on particles. Due to its self-adaptivity, SPH can handle the problem of large deformation in simulation in a natural way and is free of several flaws that a grid-based method usually has, when they are used independently to deal with the same problems. In recent years, SPH has made much headway; its accuracy, stability and adaptability have already met the requirements of a great variety of engineering applications. However, the huge computation cost is a bottleneck to its use in many real-time situations. Therefore, nowadays great efforts have been made to explore its acceleration strategies.

In this paper, an appropriate neighbour list algorithm for SPH is selected according to the architectural characteristics of GPU. Then two optimization methods are proposed to solve code divergence problem and reduce potential neighbour particles. After observing the behavior of SPH computing process, a simple yet acceptable dynamic load balancing algorithm is given. Finally, optimization strategies of inter-GPU communication are illustrated.

2 SPH Method

SPH, developed by Monaghan[1] and Lucy[2] initially for astrophysical problems, has been studied and extended extensively; it has been applied to solving the

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 104–118, 2013.
© Springer-Verlag Berlin Heidelberg 2013

problem of dynamic response of material strength and hydrodynamics with large deformations. The objects to be simulated are divided into a set of discrete elements called particles, which can move freely in space. The physical quantities of particles are updated according to their neighbours in each time step. Two particles are called neighbours only if the distance between them is less than a special value, which is called smooth radius. The physical quantity A of a particle is given by

$$A_s(r_i) = \sum_j m_j \frac{A_j}{\rho_j} W(r_i - r_j, h) \tag{1}$$

where m_j is the mass of neighbour particle j; A_j is the physical quantity of neighbour particle j; ρ_j is the density of neighbour particle j; W is a kernel function governing the contribution of neighbour particle j according to the distance between particles i and j, and the smooth length h. As particles move freely in a scenario without spatial relationship, we need to search for their neighbours in each time step.

3 Related Work

Currently, most researches on the acceleration for SPH focus on two aspects: how to speed up neighbour search and how to utilize the computation power of parallel architecture.

SPH implements the interaction between particles, and so how to create the neighbour list is one of the key points to improve SPH's performance. To speed up neighbour search, the simulation space is divided into cubical cells, called uniform grid. The size of a cell is usually equal to smooth radius. Before neighbour search, each particle is assigned to only one cell according to its center point to create a particle list of each cell. To find the neighbours of a certain particle, $27(3*3*3)$ cells need to be searched(including the cell itself resident in and 26 adjacent cells). Denote by c the number of particles residing in each cell, then the complexity of this method is $O(cN)$ where N is the total number of particles in the simulation. Dominguez et al.[3] have compared the time cost and memory consumption between 4 gridding algorithms to create the particle list of each cell. Taking time cost and memory consumption both into consideration, the algorithm in which particles are sorted according to the cells performs the best. In this method, each cell's particle list only needs to store the start position of sorted particles in the particle array. By comparing the start position of continuous cells, we can find all particles in each cell. Dominguez et al. also compare time cost and memory consumptions between VL(Verlet List) algorithm and CLL(Cell Linked List) algorithm which are both used for the creation of each particle's neighbour list. The main difference between these two algorithm lies in the fact that VL keeps neighbour list and reuses it in several time steps while CLL does not.

Fleissner et al.[4] use CPU cluster to accelerate SPH. They select ORB (orthogonal recursive bisection) domain decomposition algorithm to split simulation

space from the perspective of optimizing communication between nodes. Each CPU is assigned a subspace. Dynamic load balance is achieved by moving the boundary of subspace with a PI controller. The main advantage of this method is the decreased amount of communication between CPUs, but it has a drawback that the communication pattern gets more sophisticated.

With the rapid development of GPU technology in recent years, the performance and programmability of GPU have been promoted significantly. GPU, which is used in computer graphics traditionally, has been extended to high performance computing field. GPU has higher floating-point performance and better power-efficiency than CPU. Many computation-intensive applications have migrated from CPU to GPU and a speedup of two orders of magnitudes has been achieved. Because of its natural parallelism, SPH is remarkably suitable for parallel architectures such as GPU. In fact, even before the advent of specific languages like CUDA and OpenCL, researchers had started to use GPU to speed up SPH with graphic API. Amada et al.[5] implement SPH method partially on GPU. They create neighbour list of each particle on CPU, and then transport it to GPU to calculate force between particles. Harada et al.[6] implement SPH on GPU entirely. After CUDA, Herault et al.[7] implement SPH with CUDA for the first time. The neighbour search algorithm they use is same with that of Simon Green's[8].

As it is hard to meet the speed requirement of real-time simulation in a scale of millions of particles on a single GPU, it is extremely necessary to utilize multi-GPU in single computing node or even GPU cluster. Rustico et al.[10] and Dominguez et al.[11] have proposed multi-GPU SPH implementation independently. They both use the one-dimensional decomposition to divide simulation space into subspaces, whose number is equal to that of GPUs. The boundary of subspace is aligned at the boundary of cell. Dynamic load balance is achieved by passing the outmost cell slices of a subspace, whose corresponding GPU is overloaded, to others. The difference between Rustico et al.'s and Dominguez et al.'s dynamic load balancing algorithm is only in implementing details. As for communication strategy, both implementations divide subspace further into two boundary areas and an inner area, and cover the overhead of data exchange cost by exchanging boundary particles' data in parallel with the computation of inner area. Dominguez et al. further point out that the main factor influencing the performance of GPU is synchronization between GPUs.

4 Accelerating SPH On Single-GPU

First an appropriate neighbour list algorithm for GPU is chosen from existing ones in terms of time cost and memory consumptions. Then SPH's code divergence problem in traditional CUDA implementation is analyzed in conjunction with characteristics of SIMT architecture of CUDA-enabled GPU. Finally, Smaller Cell optimization is presented from the viewpoint of reducing the quantity of potential neighbour particles in the neighbour search.

4.1 Choosing Appropriate Neighbour List Algorithm

For SPH in which all particles have the same smooth radius, there exists two popular methods to create neighbour list, named CLL(Cell Linked List) and VL(Verlet List), respectively. Both algorithms use the same method mentioned in Section 3 to create a particle list of each cell. The difference between the two algorithms is whether to keep the neighbour list or not. Over CLL algorithm, the main advantage of VL algorithm is that it keeps neighbour list in several time steps and reduces the time cost in neighbour search. In contrast, VL algorithm requires much more memory because of storing the neighbour list. According to Dominguez et al., under the condition of searching for neighbours only once in each time step, VL algorithm is 6% faster than CLL algorithm while its memory consumption is 30 times larger. As GPU has a relatively small size of memory(GTX480 has about 1.5 GB), VL algorithm brings minor performance promotion along with excessive memory consumption, thus limiting the simulation scale fatally. Rustico et al. keep neighbour list in several time steps in their multi-GPU SPH implementation and find that each particle needs nearly 1KB memory, and that a scale of only 1.8 million particles can be simulated on a GTX480 graphic card. Considering storage and performance comprehensively, CLL algorithm is more suitable for GPU.

4.2 Code Divergence Optimization

In SIMT architecture of CUDA-enabled GPU, threads are scheduled and executed in warp. A warp is composed by 32 threads which execute same instruction at the same time in parallel. Due to conditional control flow instructions, sometimes the threads in a same warp may need to execute instructions in different code paths. In that case, the threads of a warp will execute instructions in each path serially, thereby bringing huge negative influence on performance. In traditional implementation, a thread only computes one particle's force. Each thread traverses all 27 cells with a triple loop and traverses all particles in each cell in the innermost loop(shown in figure 1).

After traversing all particles in cell, as all threads in a same warp have to execute the same instruction, those threads must be synchronized implicitly before traversing the next cell. As the relationship between CUDA threads and particles is one-to-one and the number of particles in most cells is not equal to the number of threads in a warp, different threads in a same warp may process particles in different cells. So each thread in a warp is likely to traverse cells with different particle quantities at the same time, causing imbalance workload between two synchronization points. Obviously, reducing the frequency of synchronization can lower down the negative influence of imbalance workload on performance. The frequency of synchronization is equal to the number of cells to be searched. In CLL algorithm, particles are reordered according to the cells and cells following the order of Z, Y and X axis. The particles in continuous cells in Z axis will be continuous in particle array. As each cell only stores the range(start index and end index in our implementation) of particles contained in

```
for( x=-1; x<2; x++ )
{
    for( y=-1; y<2; y++ )
    {
        for( z=-1; z<2; z++ )
        {
            read neighbour cell k's info which reside in direction (x,y,z)
            for( each particle m in neighbour cell k )
            {
                read position of m and check if m is a neighbour
                if( m is a neighbour )
                {
                    force calculation
                }
            }
            (implicit synchronization exists in CUDA)
        }
    }
}
```

Fig. 1. Code snippets of neighbour search in traditional SPH implementation

itself in particle array, the range of 3 continuous cells in Z axis can be replaced by a larger cuboid with the same range. As a result, to find the neighbours of each particle, only 9 nearby cuboids should be searched and branch instructions reduce in number, consequently greatly improving the negative influence of code divergence, and meanwhile reducing global memory access of cell information. This optimization is called Cell Merging.

4.3 Reduce Potential Neighbours

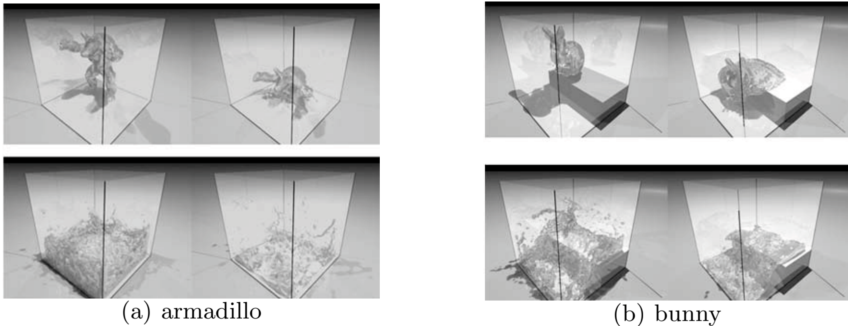
The size of cell determines the volume of the neighbour search space. When the cell size is equal to smooth radius h, each particle has to traverse 27 cells with a volume of $27h^3$. However, for each given particle i, it is only affected by particles in a sphere of radius h whose centre is i, and volume is $4\pi * h^3/3$. Smaller cell size can make the volume of neighbour search space of each particle get much closer to the volume of sphere to reduce potential neighbour particles. Supposing cell size is h/n, in order to ensure that all neighbor particles can be found for a given particle, the volume being searched should completely cover the volume of the sphere with radius h. So we have to search n cells with cell size h/n on both sides of the cell where the particle stays in each dimension of neighbour search space. Thus the side length of the cubic searched is $2*n*(1/n)h+1*(1/n)h = (2+1/n)h$ and the volume of neighbour search space is reduced to $(2 + 1/n)^3h^3$. However two drawbacks occur when n is too large. First, the total memory consumption of cells increases rapidly with the decreasing cell size:

$$mem_n = n^3 * mem_1 \tag{2}$$

where mem_n represents the total memory consumption of cells when the cell size is reduced to h/n. Second, larger n makes neighbour search more sophisticated and increases code divergence and global memory access of cell information. As mentioned in section 4.2, more neighbour cells which need to search to find neighbour particles lead to more implicit synchronization points. When n becomes larger, the amount of neighbour cells need to be searched increases rapidly, thus leading to the increasing of implicit synchronization points and code divergence. Besides, more neighbour cells means more global memory access, as the cell information is resident in the GPU's global memory. Taking all this into consideration synthetically, n=2 is the best choice in our implementation and the volume of the neighbour search space shrinks to $15.625h^3$. Smaller Cell is the name we give to this optimization.

4.4 Speedup on Single-GPU

For a better understanding of the experiment results shown in this paper, a brief description of our experimental environment is given first. The hardware platform is a dual quad-core Intel Xeon processor E5520 (2.27GHz, 8MB cache) Server with 4 GTX480 GPU cards, and each GTX480 has 480 CUDA cores with 1.5GB global memory. The operating system is Ubuntu 11.04 x86_64, CUDA runtime 4.1. Each experiment includes 1000 computing time steps for a certain scenario. Two scenarios, one of which has different scales, are used as testing cases in this paper, named armadillo and bunny respectively(shown in figure 2).



(a) armadillo (b) bunny

Fig. 2. Snapshot of scenario armadillo(a) and bunny(b) in 1st(upper-left), 300th(upper-right), 500th(lower-left), 1000th(lower-right) time step

Figure 3 shows the performance improvements with the two optimizations mentioned above. Only the time cost of updating particles' physical quantity is compared. As Smaller Cell will significantly increase code divergence(When n=2 and only Smaller Cell is applied, the time cost of updating particles' physical quantity increases by about 80%), it should be applied with Cell Merging optimization to get the best performance. In that case, each cuboid covers 5 continuous cells in Z axis.

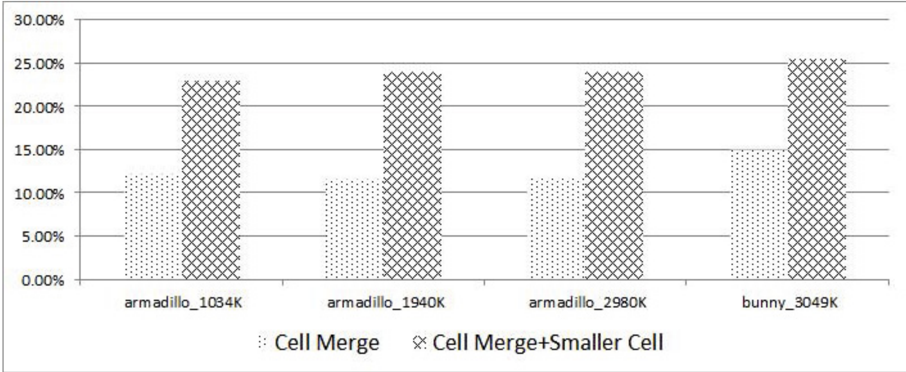


Fig. 3. Speedup on single-GPU with two optimization methods

5 Multi-GPU

In this section, we extend SPH implementation from single-GPU to multi-GPU. In multi-GPU implementation, the Cell Merging optimization is applied to accelerate the execution. Different from prior work on multi-GPU SPH, the feature of SPH which can be utilized to design a simple yet acceptable dynamic load balancing algorithm is mainly discussed instead of proposing a dynamic load balancing algorithm directly. Communication optimizations focus on additional steps introduced in multi-GPU SPH. For a better understanding of the problems faced in multi-GPU SPH, a brief description of the basic frame of multi-GPU SPH based on CLL algorithm is given first.

5.1 Basic Design

In our design, the same domain decomposition algorithm described in [9,11] is used to divide particles among multiple GPUs. Simulation space is divided into subspaces along X-axis whose number is same as the number of GPUs. The interface of subspaces is aligned at cells' boundary. The smallest unit of division is n cell slices, as each cell has a cell size of h/m in Y-Z plane. For convenience, we assume that the size of cell is equal to smooth radius below (the method is similar when cell size is different). For the reason that particle's physical quantity is influenced by all its neighbour particles, each GPU contains not only the particles in corresponding subspace, but also those near the interface within a distance of smooth radius but on the neighbour GPU's side. These particles are called ghost particles below. The ghost particles exist in both neighbour GPUs simultaneously but are updated by only one of them. After any physical quantity of particles is updated, each GPU needs to exchange the new physical quantity of ghost particles.

Multi-GPU algorithm should include 4 main steps as follows:

1. Create Neighbour List
2. Dynamic Load Balancing

```
if (load imbalance)
{
    divide the simulation space anew
    each GPU exchanges boundary particles
}
```

3. Update Particles' Physical Quantity

```
3.1 force calculation
3.2 integration in time
```

4. Particle Migration

```
4.1 gather particles needed by other GPU, called migrating
    particles.
4.2 exchange migrating particles
```

Step 1 and step 3 are the basic steps of single-GPU SPH. The reason why we put Dynamic Load Balancing step after Create Neighbour List is that CLL algorithm can make particles in same cell slice continuous in particle array, thereby simplifying the data exchange after the subspaces are repartitioned.

5.2 Dynamic Load Balancing

The key idea to obtain dynamic load balance between multiple GPUs is moving boundaries of each subspace to change the number of particles on each GPU. The key point of a good dynamic load balancing algorithm is how and when to move the boundaries. Before a new dynamic load balancing algorithm is proposed, the feature of SPH which can be utilized to design a simple yet acceptable dynamic load balancing algorithm is discussed first.

To ensure the accuracy of simulation in SPH, the moving length of any particle is usually set not longer than the smooth radius h in each time step, and consequently the spatial distribution of particles will not change a lot in two contiguous time steps. The relative stability of distribution keeps the two main factors that influence performance—the quantity of potential neighbours and neighbour particles—all stay stable. So in two continuous time steps, computation time changes slightly. We have used the scenario of armadillo with a scale of 2980K particles and scenario of bunny with 3049K to test changing range of each time step's time cost of updating particles' physical quantity in comparison with the previous one in the first 1000 time steps. Figure 4 shows the result.

From 300 to 480 time steps, the fluid in armadillo is compressed because of the initial collapse and then it becomes sparser gradually. So not only the number of potential neighbours but also the number of neighbours change severely; as a

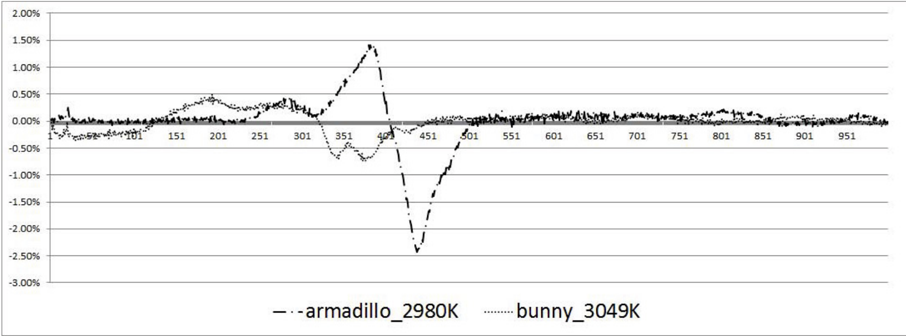


Fig. 4. Comparison of each time steps' time cost(the time cost of updating particles' physical quantity) with previous one tested with scenario armadillo and bunny

consequence, the time cost of updating particles' physical quantity also changes substantially. However, even in that case, the changing range remains within 3%. For bunny scenario that is affected less by collapse, its changing range stays within 1% all along. Because of the slow changing pattern of SPH's computation time, simple dynamic load balancing algorithm described below can get satisfying results:

```
for( each pair of neighbour GPUs A and B )
{
      if( time_A > time_B )
      {
          A gives one cell slice to B
      }else{
          B gives one cell slice to A
      }
}
```

Figure 5 shows a comparison of real wall time(longest time cost of updating particles' physical quantity among multiple GPUs) with an ideal one(average time cost of all GPUs), demonstrating that simple dynamic load balancing algorithm has acceptable effect. Figure 6 shows the speedup of multi-GPU in the same situation. As particle scale goes up, the effect of the simple dynamic load balancing algorithm improves.

The main disadvantage of simple dynamic load balancing algorithm is that it is hyper-sensitive to load imbalance among GPUs. As the whole boundary cell slice is the smallest unit of data exchanging, the alternation between the overload and underload on a GPU in continuous time steps is unnecessarily frequent, which leads to an unnecessary communication overhead in the step of Dynamic Load Balancing. There are two ways to reduce the frequency. One is to balance workload every k time steps. The other is to give a threshold to dynamic load balancing algorithm. Simulation space is repartitioned if and

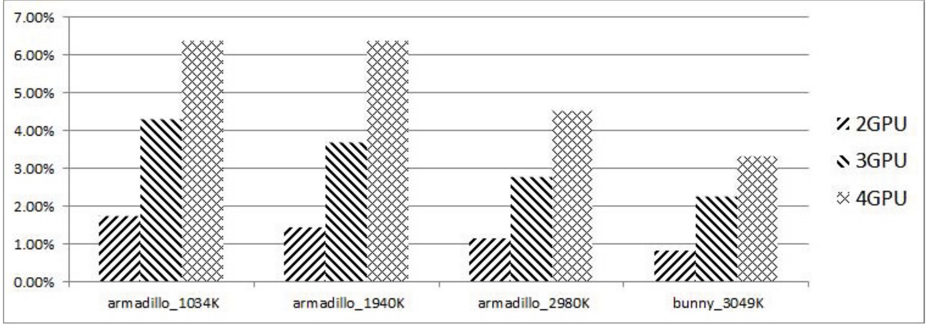


Fig. 5. Comparison of real wall time with ideal one when simple dynamic load balancing algorithm is applied

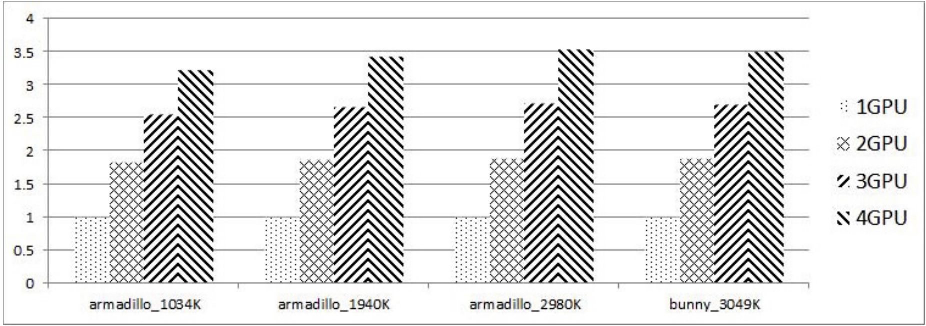


Fig. 6. Speedup with simple dynamic load balancing algorithm

only if the difference of computation overhead between two neighbour GPUs is bigger than this threshold. Dynamic load balancing algorithm should discover the imbalance among GPUs timely and gives response. So the second way is our choice.

$$new_time_diff_{left} = fabs(t_1 * nc_{left1} - t_2 * nc_{left2}) \quad (3)$$

$$new_time_diff_{right} = fabs(t_1 * nc_{right1} - t_2 * nc_{right2}) \quad (4)$$

$$nc = n_{new}/n_{original} \quad (5)$$

$$threshold = min\{new_time_diff_{left}, new_time_diff_{right}\} \quad (6)$$

where t_1 and t_2 represent the time GPU1 and GPU2(GPU1 and GPU2 are a certain pair of neighbour GPUs, GPU1 is in the left of GPU2) used to update particles' physical quantity respectively; nc stands for the changing rate of the number of particles updated by GPU after boundary moves in left or right direction; nc_{left} indicates the boundary moves to the left and nc_{right} on the opposite side; n_{new} stands for the number of particles in GPU's subspace after

boundary moves; $n_{original}$ stands for the number of particles in GPU's subspace before boundary moves. Figure 7 shows the speedup after adding the threshold. Figure 8 exhibits a comparison of real wall time with ideal one.

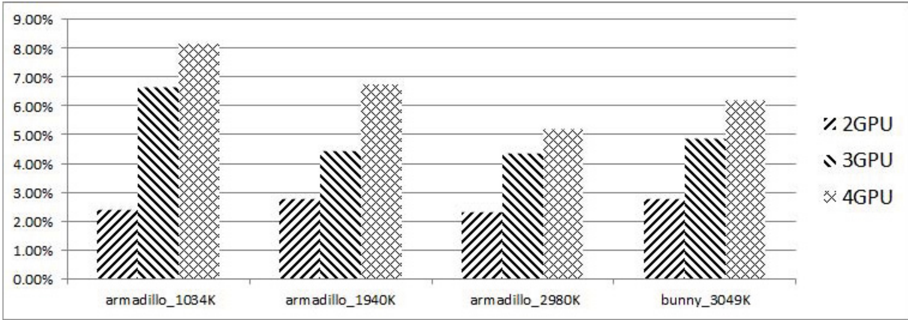


Fig. 7. Speedup of optimized dynamic load balancing algorithm compared to the simple one

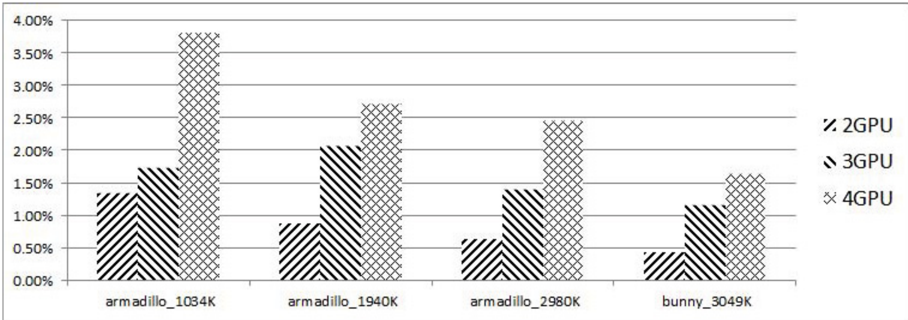


Fig. 8. Comparison of real wall time with ideal one. Optimized dynamic load balancing algorithm is applied.

Figure 9 demonstrates the ideal speedup(the performance of multi-GPU is only affected by dynamic load balancing algorithm without any other negative influences such as communication overhead) and the real speedup. Ideal speedup is calculated as (on homogeneous multi-GPU platform)

$$speedup_{ideal_n} = n * walltime_{ideal_n}/walltime_{real_n} \tag{7}$$

where $speedup_{ideal_n}$ represents ideal speedup when the number of GPU is equal to n, n represents the number of GPUs used in simulation, and walltime represents the longest time cost of updating particles' physical quantity in each time step. The real speedup is achieved without any optimization in the Dynamic Load Balancing and Particle Migration steps. Both steps are relative with communication among multiple GPUs.

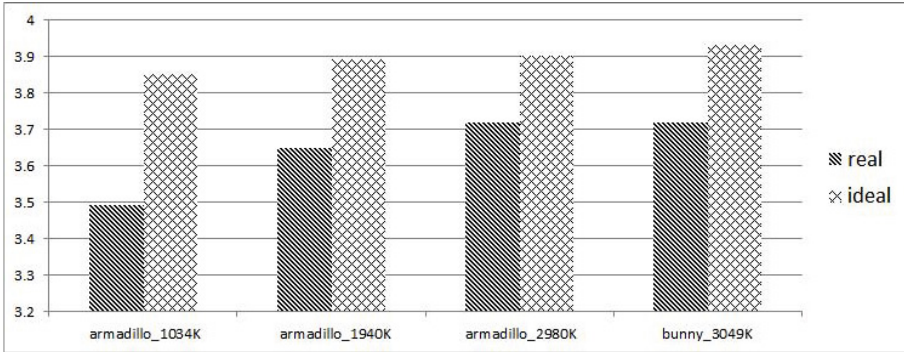


Fig. 9. Real speedup and ideal speedup of 4GPUs. Tested with: scenario armadillo with 1034K, 1940K, 2980K particles and bunny with 3049K particles.

5.3 Communication Optimization

An often used method in CUDA applications to hide communication overhead is to parallelize computation and communication. Communication among GPUs goes in the following steps:

1. Particle Migration

After sub-step integration in time in the step Updating Particles' Physical Quantity, particles may migrate from one subspace to another subspace. At the end of each time step, GPU needs to identify and exchange those particles(called migrating particle below), so there exists data transfer overhead. As the distribution of migrating particles in memory space is irregular, GPU needs extra computation to gather those particles into continuous memory space before sending them to neighbour GPU.

2. Updating Particles' Physical Quantity

As described before, it is necessary to exchange the information of boundary particles among GPUs.

3. Dynamic Load Balancing

Each GPU needs to wait for new space division and exchanges boundary cell slices according to new division of simulation space. As a result, there exist synchronization overhead and data transfer overhead.

The same method described in Rustico for covering the overhead is used in the Updating Particles' Physical Quantity step. We focus on the optimizations in Particle Migration and Dynamic Load Balancing steps. Here we give a brief description of how to hide communication overhead in those two steps.

The first sub-step of Particle Migration is gathering migrating particles in irregular distribution into continuous memory space. One way is to use a compress function to gather migrating particles, but non-negligible computation cost has to be added. For SPH, the time of updating particles' physical quantity is more than 10 times longer than the time of data exchange of boundary particles in general cases. The other way is to send migrating particles to neighbour

GPU(s) without gathering them. Migrating particles reside in two cell slices in the vicinity of GPU's boundary because the migration distance is shorter than smooth radius (the size of cell is smooth radius). Instead of gathering migrating particles, we send all particles in the two cell slices(called potential migrating particles below) to neighbour GPU(s), but it will unfortunately increase the cost of data exchange. If we can hide the communication overhead, the second way is clearly a better choice. To hide the time cost of exchanging potential migrating particles, the subspace of each GPU is divided into two boundary areas(consist of two cell slices) and an inner area. First, the kernel which is used to update boundary particles' physical quantity is launched. Then, the update of inner area works in parallel with the exchange of potential migrating particles. In this way, the time cost of exchange of potential migrating particles is hidden. In the next time step, when neighbour GPU(s) creates neighbour list with CLL algorithm, it is feasible to extract those particles that are not needed from others at a cost of a slight increase in the overhead only by giving them a sufficiently large cell value.

The space repartition in the Load Balancing step can be delayed to the time GPU starts to update particles' physical quantity. In this way, it can be parallelized with Updating Particles' Physical Quantity to hide the synchronization overhead. The data exchange of boundary cell slices after space repartition can be done together with particle migration at the end of each time step. As a result, all overheads in Dynamic Load Balancing step can be hidden through parallelization with Updating Particle's Physical Quantity.

6 Final Result

Figure 10 shows the final speedup of multi-GPU SPH implementation via communication optimization using optimized dynamic load balancing algorithm. Corresponding to the speedup presented in figure 6, the performance of multi-GPU increases by about 10%. Multi-GPU's speedups all exhibit the trend of linear acceleration when different numbers of GPUs simulate millions scale scenario.

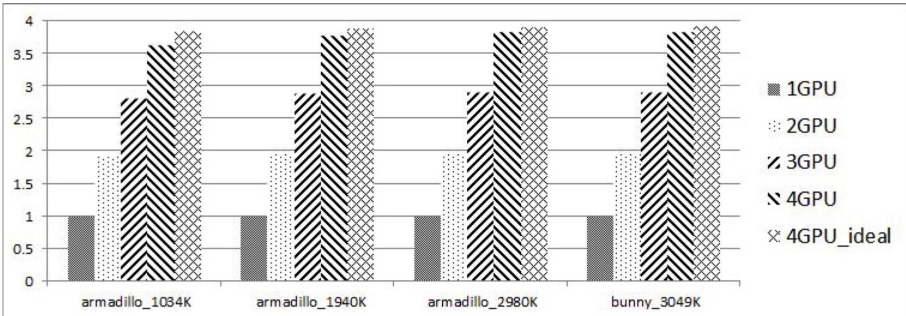


Fig. 10. The final speed up

7 Conclusions

An acceleration strategy for SPH method on single-node multi-GPU platform has been proposed in this paper. For single-GPU, we first choose an appropriate neighbour search algorithm CLL combined with architectural characteristics. Subsequently, two optimizations are made. To solve code divergence problem we merge continuous cells into a huge cell to reduce synchronization point in traditional implementation. By decreasing the cell size, less potential particles are searched in neighbour search. For multi-GPU, we focus on the changing patterns of SPH's computational time. Simple dynamic load balancing algorithm works well because the computational time of each time step changes slowly compared to previous time step. By further optimizing dynamic load balancing algorithm and the communication strategy among GPUs, a nearly linear speedup is achieved in different scenarios with a scale of millions particles.

8 Future Work

We will study the specific acceleration strategy for SPH on GPU cluster. The main difference between GPU cluster and single-node multi-GPU is that the bandwidth among nodes is far narrower than PCI-E 2.0(infiniband has not been taken into consideration yet), which may have significant effect on SPH's performance on GPU cluster. Therefore, its communication strategy may be different with what we addressed in this paper. This deserves further research.

References

1. Gingold, R.A., Monaghan, J.J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. Mon. Not. R. Astron. Soc. 181, 375–389 (1977)
2. Lucy, L.B.: A numerical approach to the testing of the fission hypothesis. Astron. J. 82, 1013–1024 (1977)
3. Dominguez, J.M., Crespo, A.J.C., et al.: Neighbour lists in smoothed particle hydrodynamics. International Journal for Numerical Methods in Fluids 67(12), 2026–2042 (2011)
4. Fleissner, F., Eberhard, P.: Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. International Journal for Numerical Methods in Engineering 74(4), 531–553 (2011)
5. Amada, T., Imura, M., et al.: Partilce-based fluid simulation on GPU. In: ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH (2004)
6. Harada, T., Koshizuka, S., et al.: Smoothed particle hydrodynamics on GPUs. In: Proceedings of Computer Graphics International (2007)
7. Herault, A., Bilotta, G., et al.: SPH on GPU with CUDA. Journal of Hydraulic Research 48(1, suppl. 1) (2010)
8. Simon Green: Particle Simulation using CUDA, http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/nv_particles.pdf

9. Rustico, E., Bilotta, G., et al.: Smoothed particle hydrodynamics simulations on multi-GPU systems. In: 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012, February 15-17 (2012)
10. Rustico, E., Bilotta, G., et al.: A journey from single-GPU to optimized multi-GPU SPH with CUDA. In: 7th SPHERIC Workshop (2012)
11. Dominguez, J.M., Crespo, A.J.C., et al.: New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. Computer Physics Communications (2013)