Exploring Fine-Grained Task-based Execution on Multi-GPU Systems

Long Chen
Qualcomm Incorporated
San Diego, CA 92121
longchen@qualcomm.com

Oreste Villa
Pacific Northwest National Laboratory
Richland, WA 99352
oreste.villa@pnl.gov

Guang R. Gao
University of Delaware
Newark, DE 19716
ggao@capsl.udel.edu

Abstract

Using multi-GPU systems, including GPU clusters, is gaining popularity in scientific computing. However, when using multiple GPUs concurrently, the conventional data parallel GPU programming paradigms, e.g., CUDA, cannot satisfactorily address certain issues, such as load balancing, GPU resource utilization, overlapping fine-grained computation with communication, etc. In this paper, we present a fine-grained task-based execution framework for multi-GPU systems. By scheduling finer-grained tasks than what is supported in the conventional CUDA programming method among multiple GPUs, and allowing concurrent task execution on a single GPU, our framework provides means for solving the above issues and efficiently utilizing multi-GPU systems. Experiments with a molecular dynamics application show that, for non-uniform distributed workload, the solutions based on our framework achieve good load balance, and considerable performance improvement over other solutions based on the standard CUDA programming methodologies.

Keywords: fine-grained, task, GPGPU, multi-GPU, dynamic load balance

1 Introduction

How to efficiently utilize single-GPU systems for general purpose scientific computing has been investigated for many applications. Beyond single-GPU systems, there is a growing interest in exploiting multiple GPUs. The main benefit of using multi-GPU systems is that such systems can provide a much higher performance potential than the single-GPU systems. Further, multi-GPU systems can overcome certain limitations associated with the single-GPU systems, e.g., limited global memory.

Ensuring a good dynmamic balanced load across multiple devices is critical to achieving strong performance on multi-GPU systems. This is particularly true when the target applications exhibit irregular, unbalanced workload, or the computation is to be carried out on heterogeneous platforms, e.g., consisting of both CPUs and GPUs, or a diversity of GPUs of varying capability. A static scheduling approach will not work since it lacks the ability to automatically adapt to the application irregularity and system heterogeneity. One possible approach is to decompose the computation into small chunks. Whenever a GPU is free, it receives a chunk for processing. While this approach potentially can provide better load balancing behavior than the static approach, the overall performance of the program is heavily affected by the granularity of the chunks. Generally speaking, using finer-grained chunks can achieve better load balancing. When the workload in each chunk becomes smaller and smaller, a single chunk may not be able to present enough parallelism to fully utilize the GPU. However, the majority of the conventional NVIDIA CUDA programming methodologies and techniques implies that programmer-defined functions should be executed sequentially on the GPU[1]. Therefore, using these fine-grained chunks could result in the underutilization of the GPUs, and degrade the overall performance. On the other hand, some applications do require a more refined execution behavior on GPU-enabled systems than CUDA. For example, as suggested in [20], a fine-grained GPU execution model would allow fine-grained message-driven applications to overlap the communication with the computation on GPU clusters.

Open Computing Language (OpenCL) [13] supports the both data parallel programming model and task parallel programming model However, its task model is basically established for multi-core CPUs, and does not address the characteristics of GPUs. Moreover, it does not require a particular OpenCL implementation to actually execute multiple tasks in parallel.

Our approach for solving these issues is to allow con-

[1]The latest NVIDIA Fermi architecture supports only 4 concurrent kernels (will increase to 16). Our approach can provide even finer-grained concurrent task execution, and can also be applied to this new architecture to further improve its utilization.

current execution of fine-grained tasks on multi-GPU systems. Specifically, in our approach, the granularity of task execution is finer than what is currently supported in CUDA; the execution of a task only requires a subset of the GPU hardware resources. While some tasks are being processed by part of the GPU resources, CPU can dispatch other homogeneous/heterogeneous tasks to this GPU, and these tasks can be processed by using other part of the GPU resources. All tasks can be processed concurrently and independently, assuming there is no dependence among them. While scheduling fine-grained tasks enables good load balancing among multiple GPUs, concurrent execution of multiple tasks on each single GPU solves the hardware underutilization issue when tasks are small.

In our earlier work [3], we developed a task-queue based load balancing scheme for single-GPU systems, then we replicated the design to enable a rudimentary form of multi-GPU support. While the proposed multi-GPU mechanism was able to reduce multiple kernel calls overhead as well as load balancing the execution of tasks on each individual GPU (better than the CUDA scheduler), it was not able to efficiently balance workload among multiple GPUs. The main reason was that the granularity of tasks was too coarse grained and no effective coordination strategy among multiple GPUs was defined. In this paper, we extend this previous work and make the following contributions:

- We present a framework that coordinates the execution of tasks for multi-GPU systems, where one such system is a compute node equipped with multiple GPUs. This framework features fine task granularities and a *task container hierarchy* such that it maintains good dynamic load balancing while minimizing the coordinating overhead.
- We propose a new GPU level, fine-grained execution scheme that matches the GPU's architectural features. This scheme can achieve better dynamic load balancing and potentially utilize the hardware more efficiently than the one proposed in [3], when a task exposes limited data parallelism.
- We evaluate the solutions based on our framework with a molecular dynamics (MD) application with the NVIDIA CUDA environment[2]. For systems of non-uniform distributions of atoms, our solutions achieve nearly optimal load balancing, and considerable performance improvement over alternative implementations based on the canonical CUDA programming paradigm.

[2]Although our current framework is built with NVIDIA CUDA devices, it can also be implemented with AMD GPUs as well.

The rest of the paper is organized as follows. Section 2 presents a brief overview of the previous works on utilizing multiple GPUs, and exploiting task parallelism on GPUs. Section 3 describes the CUDA architecture and its programming model. Section 4 discusses our fine-grained task-based execution framework for multi-GPU systems. Section 5 evaluates our design with a MD application on a 4-GPU system. Section 6 concludes the paper.

2 Related Work

Using multi-GPU systems, including GPU clusters, is gaining popularity in scientific computing [5, 8, 9, 11, 20, 23]. In general, these works demonstrate that such platforms can be beneficial in terms of performance, power, and price. There are continuing efforts to facilitate programming GPU clusters. The work in [6] employs Global Arrays [17] to simplify the communications among GPUs. A memory consistency model is proposed in [15] to enable a distributed shared memory system, which consists of texture memory across multiple GPUs. Performance modeling of multi-GPU systems and GPU clusters is studied in [21]. Results show that such modeling techniques can be accurate for applications of a deterministic execution manner.

Scheduling task execution on GPU-enabled systems and other heterogeneous platforms has been investigated in a few studies. Merge [14] is a programming framework proposed for heterogeneous multi-core systems. It employs a library-based method to automatically distribute computation across the underlying heterogeneous computing devices. STARPU [1] is another framework for task scheduling on heterogeneous platforms, in which hints, including the performance models of tasks, can be given to guide the scheduling policies. Our work is orthogonal to prior efforts in that our fine-grained approach provides better performance on single-GPU systems[3] and also additional design space for programmers to further improve the performance on multi-GPU systems.

3 CUDA Architecture

In this section we provide a brief introduction of the CUDA architecture and its programming model. More details are available on the CUDA website [18].

In the literature, GPUs and CPUs are usually referred to as the *devices* and the *hosts*, respectively. CUDA devices have one or multiple streaming multiprocessors (SMs), each of which consists of one instruction issue unit, eight scalar processor (SP) cores, two transcendental function units, and on-chip shared memory. For some high-end devices, the SM also has one double-precision floating point unit. CUDA architecture features both on-chip memory

and off-chip memory. The on-chip memory consists of the register file, shared memory, constant cache and texture cache. The off-chip memory consists of the local memory and the global memory. Since there is no ordering guarantee of memory accesses on CUDA architectures, programmers may need to use memory fence instructions to explicitly enforce the ordering, and thus the correctness of the program. The host can only access the global memory of the device. On some devices, part of the host memory can be pinned and mapped into the device's memory space, and both the host and the device can access that memory region using normal memory load and store instructions.

A CUDA program consists of two parts. One part is the portions to be executed on the CUDA device, which are called *kernels*; another part is to be executed on the host, which we call the *host process*. With the current CUDA environment, one host process can communicate with only one device, while one device can be shared by multiple host processes. When a device is shared by multiple host processes, the resources created by a host process, e.g., allocated memory segment, cannot be accessed by other host processes, even they are sharing the same physical device. The device executes one kernel at a time, while subsequent kernels are queued by the CUDA runtime. When launching a kernel, the host process specifies how many threads are required to execute the kernel, and how many *thread blocks* (TB) these threads should be equally divided into. On the device, the CUDA hardware schedules and distributes TBs to SMs with available execution capacity. Each thread is mapped to one SP core, and has its own execution context. Moreover, the SM manages the threads in groups of 32 threads called *warps*, in the sense that all threads in a warp execute one common instruction at a time. Because of this feature, no explicit mechanism is needed for synchronizing threads within a warp. Thread divergences occur when the threads within a warp take different execution paths, and execution of all taken paths will be serialized, which can significantly degrade the performance.

4 A Fine-grained Task-based Execution Framework for Multi-GPU Systems

The multi-GPU systems discussed in this paper can be viewed as illustrated in Figure 1. In the system shown, multiple devices are connected to the host via a PCIe bus. With the current CUDA environment, devices cannot exchange data with each other directly. Instead, data movements across devices have to be done by the host processes.

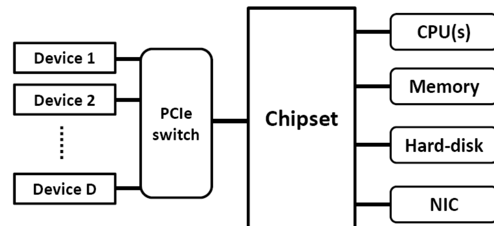


Figure 1: A PCIe connected multi-GPU system

To efficiently utilize such multi-GPU systems, we propose a fine-grained task-based execution framework, which is demonstrated in Figure 2.

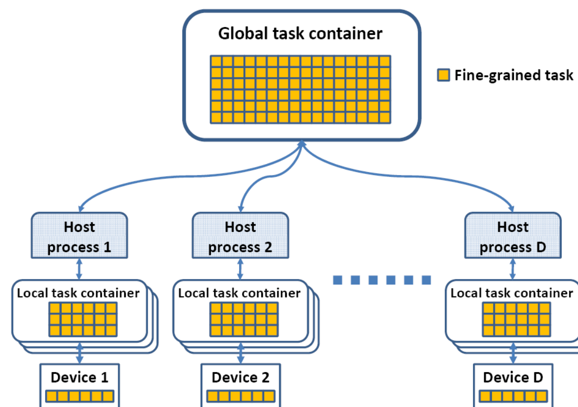


Figure 2: Fine-grained task-based execution framework for multi-GPU systems

In our framework, for each host process-device pair, one or more programmer-created *local task containers* are used to enable the host-device communications when a kernel is running on the device. Such local task containers are only accessible by the corresponding host process and the device. The computation to be carried out is first decomposed into many fine-grained tasks[3], which are kept in a programmer-created *global task container*. All individual host processes can fetch/send tasks from/to this global task container. Once a host process detects that its own local task container has free space and some tasks in the global task container are ready to start, it moves a number of such tasks from the global task container to its own local task container, and informs the device the availability of new tasks. This two-level task container hierarchy enables minization of the access contentions on ei-

[3]Currently the task decomposition and data dependence is explicitly handled at the application level by programmers.

ther level. On each device, a *persistent* kernel is launched at the beginning of the computation. This kernel fetches tasks from the local task container(s), and executes them by groups of threads, which we call *task execution units* (TEUs). Note that the processing of each task is carried out by a single TEU, which can be at a granularity finer than the entire device. Multiple tasks can be fetched and processed by different TEUs (on a same device) concurrently and independently, assuming there is no dependence among them. Moreover, task sending (by the host process) and task fetching (by the TEUs) can happen at the same time. This device-scope fine-grained execution scheme is illustrated in Figure 3.

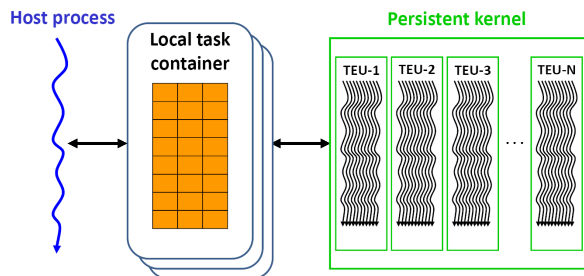


Figure 3: Fine-grained execution scheme on a single device

One of the challenges to implement this device-scope fine-grained execution scheme is to provide a correct and efficient host-device communication mechanism when a kernel is running on the device. As demonstrated in [3], by judiciously utilizing the GPU's architectural features, such as the multiple memory spaces, and the asynchronous concurrent execution, this mechanism can be achieved for host-device communications.

Individual TBs are used as TEUs in [3]; each task is executed by a single TB. We call this the *TB-level* task execution scheme. While this scheme is finer-grained than the normal CUDA scheme (where a function is executed by the entire device), it is not necessarily the optimal granularity of TEUs. For example, if each task only exposes limited data parallelism, which can be handled by a few threads, using the TB-level task execution scheme simply wastes the computation power of other threads in the same TB. Therefore, we propose a *warp-level* task execution scheme, where tasks are fetched and executed by individual warps on the device. Since, on the GPU, the SM creates, manages, schedules, and executes threads in warps, the warp-level task execution scheme perfectly matches this architectural feature, and therefore can potentially utilize the hardware more efficiently than the TB-level scheme, and the normal CUDA programming paradigm. On the other hand, using even finer-grained TEUs, such as individual threads, will not help. Since all threads within a warp share an instruction issue unit, they cannot execute different codes concurrently. In fact, the most efficient execution on GPU is that all threads of a warp take the same execution path [19].

4.1 Dynamic Load Balancing Design

Our design for dynamically balancing workload on multi-GPU systems is based on our fine-grained task-based framework described above. This design follows the basic structure illustrated in Figure 2. Specifically, the work to be processed with a multi-GPU system is decomposed into fine-grained tasks, which are to be executed by individual warps. When a local task container becomes empty, the corresponding host process fills it with certain number of fine-grained tasks retrieved from the global task container. Here we assume that the dependencies among tasks have been taken care of by the host processes, and all tasks in the local task containers can be executed independently by devices. Since all host processes share a host memory space, the orchestrations among them can be accomplished with regular programming methodologies and techniques for shared-memory systems.

5 Experiments and Discussions

In this section, we evaluate the dynamic load balancing solutions based on our fine-grained task-based execution framework on a multi-GPU system, using a molecular dynamics (MD) [7] application, for different workload distributions. We compare our solutions with other techniques based on the standard CUDA programming methodologies. The results show that, for non-uniform distributed workload, our solutions achieve good load balancing across GPUs, with significant performance improvement over other alternative approaches.

5.1 Molecular Dynamics

MD is a simulation method of computing dynamic particle interactions on the molecular or atomic level. The method is based on knowing, at the beginning of the simulation, the mass, position, and velocity of each particle in the system. Each particle interacts with other particles in the system, and such interaction is computed using a distance calculation, followed by a force calculation. When the net force for each particle has been calculated, new positions and velocities are computed through a series of motion estimation equations. The process of net force calculation and position integration repeats for each time step of the simulation. Non-uniform distributions of atoms in

space are found in MD simulations and produce highly irregular computational load [12].

In our experiments, we use synthetic systems of helium atoms, where the force between atoms is calculated using both electrostatic potential and Lennard-Jones potential [7]. The systems are built by following 4 different atom distributions in a 3D space. *Uniform* distribution arranges atoms uniformly distributed in the system. A system built with *Sphere* distribution has a higher density in the center than in periphery. The density decreases from the center to the periphery following a Gaussian curve. *Equal-sized cluster* distribution first partitions the system into clusters of equal number of atoms, where the centers of clusters are randomly generated. Then each cluster is built by following Sphere distribution. *Random-sized cluster* distribution also generates clusters of atoms. Unlike the Equal-size cluster distribution, for each cluster, both the center and the number of atoms in this cluster are randomly generated for the Random-size cluster distribution. Figure 4 shows example systems of these distributions. It is clear that systems built with last three atom distributions have irregular computational load in the space.

The reason for using synthetic systems is two-fold: (1) synthetic systems can isolate the load balancing issue from other complex facts exhibiting in the real life systems, and therefore facilitates the evaluations and analyses of different solutions, (2) it is very difficult to find real life examples where a particular atom distribution is constant as the simulated system size scales up, and therefore it makes very hard to objectively evaluate different solutions with different system sizes.

For each system, the N atom positions are stored in a linear array A. Specifically, the 3D space is first decomposed in boxes of size equal to the *cutoff* radius. Then, atoms in each individual box are stored into the array contiguously. Due to the effect of cutoff radius, the systems built with non-uniform distributions exhibit irregular, unbalanced computation workload for different boxes. Consequently, using this data layout with multi-GPU systems can be challenging, in terms of the load balancing and the absolute performance.

5.2 Implementations

The platform used in our experiments has 1 quad-core AMD Phenom II X4 940 processor and 4 NVIDIA Tesla C1060 GPUs. The system is running 64-bit Ubuntu version 8.10, with NVIDIA driver version 190.10. CUDA Toolkit version 2.3, CUDA SDK version 2.3, and GCC version 4.3.2 were used in the development.

We implement dynamic load balancing solutions based on our fine-grained task-based execution framework for this 4-GPU system, using both warp-level and TB-level schemes. We also implement other load balance techniques based on the conventional CUDA programming method. Note that all solutions use the same device function to perform the force computation, which is based on the atom-decomposition [22] technique. Also, before computation, the array A is already available on devices. In this way, we can ensure that all performance differences are only due to the load balancing mechanisms employed.

Solution STATIC statically divides A into P contiguous regions of equal size, where P is the number of devices used. Each device is responsible for computing forces for atoms within a region. For systems of unbalance workload in the space, although each region can be processed efficiently on a separate device, the overall performance is deemed to be low due to the poor load balance among devices. The objective of having this solution is to use it as a baseline to compare other load-balancing solutions for the multi-GPU systems.

Solution RANDOM randomly permutes the elements in A before the simulation, and/or after every certain amount of time steps in the simulation. After the permutation, the workload is distributed as Solution STATIC; the array A is equally partitioned into P contiguous regions, one for each device. By discarding the locality information of atoms, this solution ensures almost perfect load balance among multiple devices, since now each atom in the array has a (nearly) equal probability to exert a force with all other atoms in the array. This technique is used in parallel implementations of state-of-the-art biological MD programs such as CHARMM [2] and GROMOS [4]. However, when applied to the GPU codes, it introduces the problem of thread divergence inside a warp for simulating systems of non-uniform atom distributions, as now atoms with a lot of force interactions are mixed with atoms with few force interactions.

Solution CHUNKING is a dynamic approach that uses fine-grained workload to dynamically balance load across multiple devices. Specifically, the array A is decomposed into many data chunks of equal atoms. Whenever a host process finds out that the corresponding device is free (on kernel running on the device), it assigns the force computation of atoms within a data chunk to the device, by launching a kernel with the data chunk information. The host process waits until this kernel completes the computation and the device becomes free again, then it launches another kernel with a new data chunk. This solution is designed to take advantage of both good load balancing among multiple GPUs and thread convergence. Since a device only receives a relatively small workload after it finishes the current one, this approach potentially provides better load balancing than Solution STATIC, for

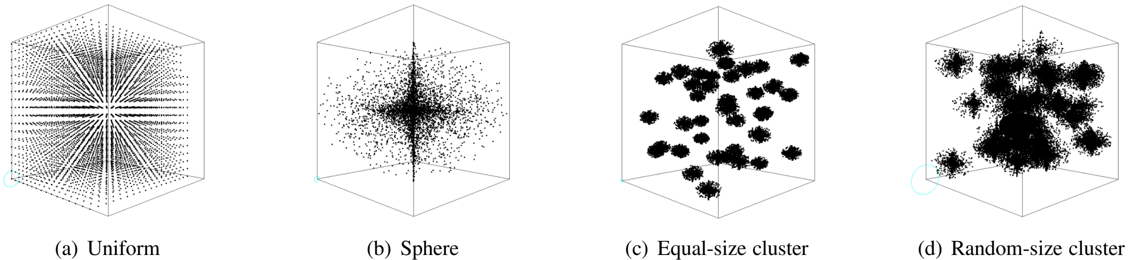

(a) Uniform (b) Sphere (c) Equal-size cluster (d) Random-size cluster

Figure 4: Example synthetic systems of different atom distributions

non-uniform distributed workload. Since the performance of chunking is affected by the chunk size, we use an empirically optimal size of $15,360$ atoms/chunk (120 TBs $\times$ 128 atoms) for this solution, which achieves the best absolute performance among all examined chunk sizes.

Solution WARP-TASK is an approach that employs the dynamic load balancing design (presented in Section 4.1), which is based on our warp-level task-based execution scheme . In this solution, each task is the force evaluation of 32 atoms (stored contiguously in A) with all atoms in the system, and it is to be executed by a single warp. The simulation of each time step is decomposed into tasks, which are kept in the global task container. On each device, two local task containers are used to overlap the host task sending with the device task fetching. Each local task container holds up to 20 tasks. Whenever a task container of a device becomes empty, the corresponding host process tries to fetch as much as 20 tasks from the global task container at a time, and sends them to the device with a single task sending procedure. The kernel is run with 120 TBs, each of 128 threads, i.e., 512 warps on a single device. Note these configuration numbers are determined empirically.

Solution TB-TASK is similar to Solution WARP-TASK, except that now the TB-level task-based execution scheme is used; TEUs are TBs. This approach is also utilized in [3]. To accommodate this granularity change of TEUs, the granularity of each task is accordingly increased to the force evaluation of 128 atoms (stored contiguously in A) with all atoms in the system. The kernel for this solution is also run with 120 TBs, each of 128 threads.

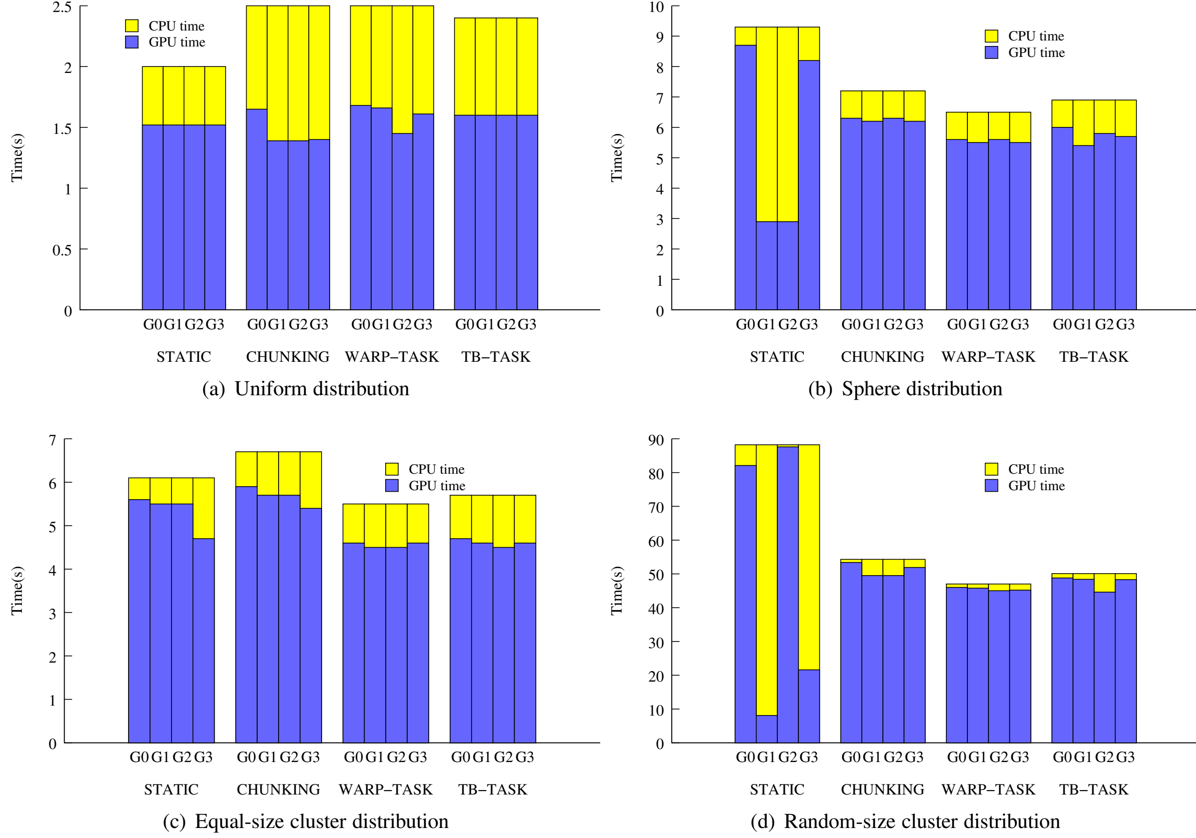
5.3 Results and Discussions

We conduct our experiments on the 4-GPU system described in 5.2; all 4 GPUs are used in the simulations. For each run of the simulation, we start with a system generated by one of the 4 distributions described earlier in Section 5.1, and we use the average runtime in the first 10 time steps as the metric for the absolute performance (the runtime differences among these 10 time steps are trivial). Moreover, we decompose the runtime into *CPU time* and *GPU time*. CPU time denotes the time spent on the corresponding host process for the host-device data transfer, position update, and communication with other host processes. GPU time denotes the time spent on the device for the force computation. In this way, the load balancing behavior is illustrated with the GPU times spent on different devices.

We first investigate how different solutions behave for systems of a particular size, i.e., 256K-atom. Figure 5 shows the average runtime per time step for all solutions (without Solution RANDOM). Particularly, such timing information is presented for each individual GPU (labelled with G0-G3). Solution RANDOM does achieve excellent load balancing among GPUs, however, it is usually much slower than other solutions. Therefore we will only describe its behaviors in text.

From Figure 5, it is clear that, in our experiments, the systems built with non-uniform atom distributions require much more computation time than the system built with the uniform atom distribution. This is because that with our particular Uniform distribution, there are only a few atoms in the space determined by the cutoff radius. However, for other distributions, in average, each atom may interact with up to hundreds of other atoms. Given the force computation is the most expensive part in the MD simulation, the number of force computations involved in various systems causes huge differences among them, in terms of the absolute performance.

For the Uniform distribution, Solution STATIC achieves the best absolute performance and load balancing. It virtually balances the workload among GPUs perfectly with few overhead, while dynamic solutions suffer from the additional overhead due to the runtime scheduling. As we can see from the figure, this additional overhead is quite noticeable when the GPU time is small. However, for non-uniform distributed workload, dynamic solutions show their strengths, in terms of load balancing.



(a) Uniform distribution

(b) Sphere distribution

(c) Equal-size cluster distribution

(d) Random-size cluster distribution

Figure 5: Dynamic load among GPUs for 256K-atom systems

Especially, for Solution WARP-TASK, the difference of GPU time among GPUs is within 3%, while such difference is up to 9% for Solution CHUNKING. Regarding the absolute performance, our fine-grained task-based solutions achieve up to 1.9x speedup over Solution STATIC. Both Solution WARP-TASK and Solution TB-TASK outperform Solution CHUNKING. Particularly, for Solution WARP-TASK, we see improvements of 11%-22% over Solution CHUNKING for different non-uniform distributions. Such improvements are due to the following facts. First, by scheduling the computation in fine granularity, our solution achieves better load balancing among GPUs than Solution CHUNKING. Second, on each individual GPU, Solution CHUNKING makes a kernel call for processing a data chunk. In a kernel execution, because of the unbalanced workload, some TBs may complete their computation earlier than other TBs, and have to wait till the termination of the kernel, which indicates inefficiency. This issue could be alleviated by using larger data chunks. However, our experimental results (not shown) confirm that larger chunks cause serious load imbalance among GPUs, and eventually affect the absolute performance negatively. Actually, the specific chunk size we used in Solution CHUNKING achieves the best performance among all examined chunking sizes. In our Solution WARP-TASK, whenever a warp becomes free, new tasks can be inserted from the host, which can be fetched and executed by this warp, without affecting the execution of other warps on the same device. Therefore, this fine-grained execution scheme achieves better GPU utilization than Solution CHUNKING does. Not shown in the figure, Solution RANDOM also balances the workload among GPUs perfectly. However, because mixed atom data cause serious thread divergence on each device, this solution is the slowest among all solutions, e.g., 6.9x slower than Solution STATIC for the Equal-size cluster distribution.

On the other hand, since Solution WARP-TASK employs the execution scheme that optimally matches the GPU's architectural feature, we expected that it could outperform Solution TB-TASK remarkably, in terms of absolute performance and load balancing, We do see that Solution WARP-TASK achieves better load balancing than

Solution TB-TASK. However, regarding the absolute performance, it only exhibits limited improvements over Solution TB-TASK, i.e., around 5%. A further examination of the force computation function reveals that, due to the specific algorithm used in our MD simulation, Solution WARP-TASK implies much more memory operations (of the same order of magnitude of N, the number of atoms in the system) to the array A, than Solution TB-TASK does. These extra memory operations offset the majority of the benefits of using a warp-level solution.

Figure 6 shows the relative speedup of the average runtime per time step of all solutions (using 4 GPUs) over Solution STATIC, with respect to system sizes. Again, Solution RANDOM is not shown here due to its low performance. For the Uniform distribution, Solution STATIC still achieves the best absolute performance. However, other dynamic solutions reach comparable performance for large system sizes. This is because that the additional runtime scheduling overhead becomes relatively trivial, compared to the GPU time, when the system size increases. For other non-uniform distributions, our fine-grained task-based solutions achieve much better performance than Solution STATIC and Solution CHUNKING when large systems (i.e., 128K-atom and up) are used[4]. For all non-uniform distributions, Solution WARP-TASK constantly outperforms Solution TB-TASK, for system sizes up to 512K-atom. However, the performance improvement becomes less significant when the system size increases. This in fact confirms our previous reasoning on why Solution WARP-TASK only exhibits limited benefits over Solution TB-TASK; when the system becomes large, the execution of those extra memory operations in Solution WARP-TASK will constitute a considerable portion of the overall runtime. In fact, when the system size reaches 1024K-atom, Solution TB-TASK achieves a similar or even better performance than Solution WARP-TASK. Note that the issue of extra memory operations is not directly related to our fine-grained task-based approach, but due to the particular algorithm used in our experiments.

6 Conclusion

This paper proposed a fine-grained task-based execution framework for multi-GPU systems. Based on this framework, we presented a design for dynamically balancing workload on multi-GPU systems. A molecular dynamics application is used to evaluate the effectiveness of our design. Experimental results demonstrate that, for non-uniform atom distributions, our fine-grained task-based solutions achieve good load balancing and absolute performance improvement over other approaches based on the standard CUDA programming methodologies.

There are a number of possible extensions to our current work. In our current design of the GPU task-based execution framework, to ensure the dependencies among tasks, we have to manually schedule the execution of tasks on the CPU side, according to their dependencies. An efficient mechanism to automatically enforce dependencies among tasks will greatly facilitate the design and development of fine-grained data-driven or event-driven applications. Another future work is to extend the current design for GPU clusters, which have been introduced to several scientific sites. In this case, MPI[10], Global Arrays[16], or other alternatives should be integrated into our framework to take care of the distributed memory configuration.

[4]Except for the Equal-size cluster distribution at the size of 128K-atom, where all dynamic solutions are worse than Solution STATIC.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009*, pages 863–874, Delft, Netherlands, 2009.

[2] B. Brooks and H. M. Parallelization of Charmm for MIMD Machines. *CDAN*, 7(16):16–22, 1992.

[3] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single- and Multi-GPU systems. In *IPDPS'10*, Atlanta, GA, USA, 2010.

[4] T. Clark, M. J.A., and S. L.R. Parallel Molecular Dynamics. In *SIAM PP'91*, pages 338–344, March 1991.

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *SC'04*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Z. Fan, F. Qiu, and A. E. Kaufman. Zippy: A framework for computation and visualization on a gpu cluster. *Comput. Graph. Forum*, 27(2):341–350, 2008.

[7] D. Frenkel and B. Smit, editors. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, Inc., Orlando, FL, USA, 1996.

[8] D. Gödeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33(10-11):685 – 699, 2007.

[9] N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using occlusion-switches. In *I3D'03*, pages 103–112, New York, NY, USA, 2003. ACM.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambrdge, MA, Oct. 1994.

[11] D. R. Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming hmmer-search implementatio. In *SC'05*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[12] L. Kalé, M. Bhandarkar, M. Bh, and R. Brunner. Load balancing in parallel molecular dynamics. In *ISSISPP'98*, pages 251–261, 1998.

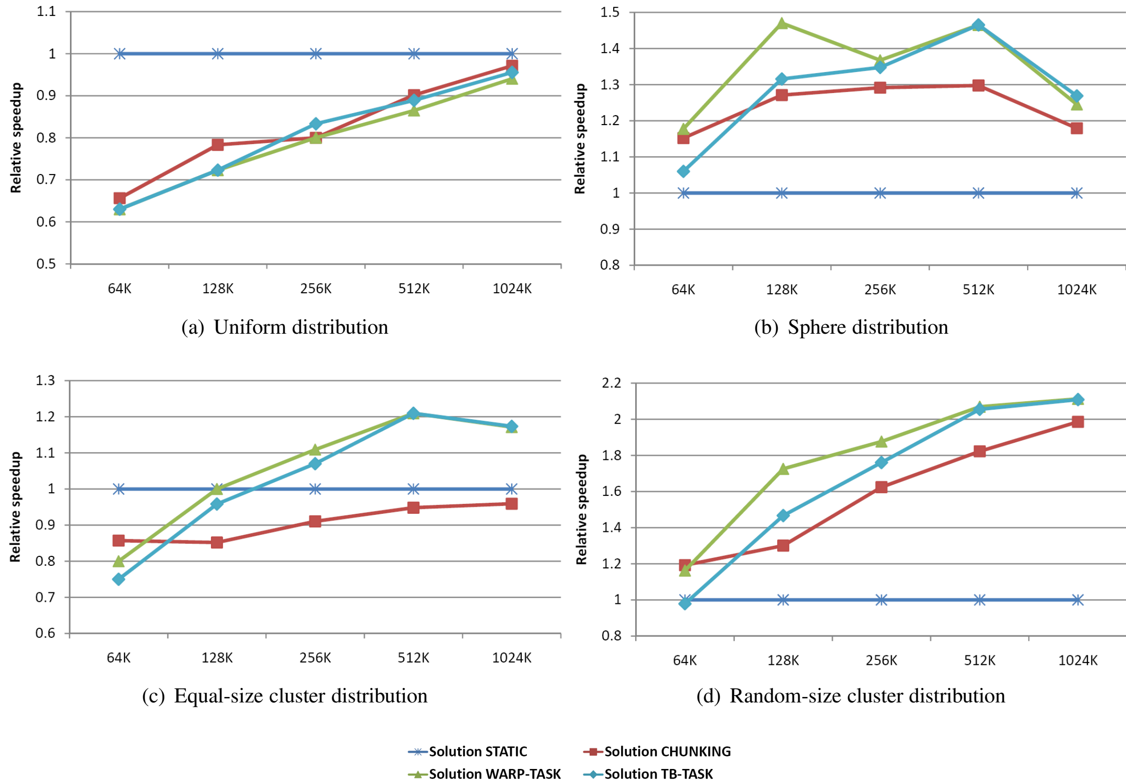


Figure 6: Relative speedup over Solution STATIC versus system sizes

[13] Khronos. OpenCL. `http://www.khronos.org`.

[14] M. D. Linderman, J. D. Collins, H. Wang, and T. H. M. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, 2008.

[15] A. Moerschell and J. D. Owens. Distributed texture memory in a multi-gpu environment. In *GH'06*, pages 31–38, New York, NY, USA, 2006. ACM.

[16] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high performance computers. *The Journal of Supercomputing*, 10:10–197, 1996.

[17] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. HPCA*, 20(2):203–231, 2006.

[18] Nvidia. CUDA. `http://www.nvidia.com`.

[19] Nvidia. NVIDIA CUDA Programming Guide 3.0, 2010.

[20] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *SC'08*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[21] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *IPDPS'09*, pages 1–12, Washington, DC, USA, 2009.

[22] W. Smith. Molecular dynamics on hypercube parallel computers. *CPC*, 62:229–248, 1991.

[23] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Large volume visualization of compressed time-dependent datasets on gpu clusters. *Parallel Comput.*, 31(2):205–219, 2005.