Adaptive Task Size Control on High Level Programming for GPU/CPU Work Sharing

Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault, Olivier Aumage

Adaptive Task Size Control on High Level Programming for GPU/CPU Work Sharing

Tetsuya Odajima[1], Taisuke Boku[1,2], Mitsuhisa Sato[1,2],
Toshihiro Hanawa[2], Yuetsu Kodama[1,2],
Raymond Namyst[3], Samuel Thibault[3], and Olivier Aumage[3]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba
[2] Center for Computational Sciences, University of Tsukuba
[3] University of Bordeaux - LaBRI - INRIA Bordeaux Sud-Ouest

Abstract. On the work sharing among GPUs and CPU cores on GPU equipped clusters, it is a critical issue to keep load balance among these heterogeneous computing resources. We have been developing a runtime system for this problem on PGAS language named XcalableMP-dev/StarPU [1]. Through the development, we found the necessity of adaptive load balancing for GPU/CPU work sharing to achieve the best performance for various application codes.
In this paper, we enhance our language system XcalableMP-dev/StarPU to add a new feature which can control the task size to be assigned to these heterogeneous resources dynamically during application execution. As a result of performance evaluation on several benchmarks, we confirmed the proposed feature correctly works and the performance with heterogeneous work sharing provides up to about 40% higher performance than GPU-only utilization even for relatively small size of problems.

1 Introduction

While GPU clusters with high performance GPUs provide cost effective HPC environment, still there is a serious problem on programming for users who are forced to describe complicated codes with mixed paradigm on parallel processing and GPU computing. Recent programming codes on a modern PC cluster commonly described in a hybrid manner to combine MPI and OpenMP to exploit the parallelism of resources effectively. In addition on GPU clusters, the programmers additionally have to describe GPU manipulation. As a result, large scale parallel GPU programming on GPU clusters becomes the toughest work on parallel processing which easily causes numerous coding errors and reduces code productivity.

We have been developing a language named XcalableMP (hereinafter called XMP for short)[2], which is a directive-based PGAS (Partitioned Global Address Space) language for parallel systems with distributed memory architecture. In addition to the original XMP specification, we also proposed an extension of XMP for accelerating device programming environments such as CUDA or

OpenCL, named XcalableMP-dev [3] (hereinafter called XMP-dev for short), which employs the concept of XMP by supporting a feature to off-load the computation of a specified section (loop) to the target accelerating devices.

In our previous work [1], we utilized both GPU and CPU resources on each node for work sharing of the loop execution within the context of the XMP-dev language. For this purpose, we apply StarPU [4] for sub-task management and scheduling where a loop execution is divided into a number of sub-tasks for multiple GPUs and CPU cores on each computation node. Based on this concept, we implemented XcalableMP-dev/StarPU (hereinafter called XMP-dev/StarPU for short) which enables the loop-level work sharing among CPU cores and GPU on each computation node while the framework of XMP. In some cases, we confirmed that this new feature improves the performance of GPU clusters with additional power by CPU cores rather than using GPU only, with very simple and easy programming for high productivity. In many cases, however, the performance gain with GPU/CPU work sharing is not enough as estimated. The basic problem is how to decide the task size to be assigned to CPU cores and GPUs which has different characteristics on the performance.

In this paper, to solve this problem, we propose a new framework to control the task size to be dispatched to heterogeneous devices (CPU cores and GPUs) individually and dynamically. In this method, we can keep the execution time on each device as almost the same by dispatching different size of tasks to them even with a limited but moderate number of tasks according to the problem size.

2 XcalableMP, XcalableMP-dev and StarPU

2.1 Overview of XcalableMP (XMP)

XMP [5] is a PGAS language for describing large-scale scientific code on parallel systems with distributed memory architecture. For simplification for easy understanding, XMP is a directive-based parallelizing language with grammar similar to that of OpenMP. And its concept came from HPF [6] for global array distribution and work sharing on loop construction. It is possible to parallelize the target code with just a few changes on the original serial code, thus the programming effort can be significantly reduced compared with many additional lines in MPI programming.

2.2 XcalableMP-dev (XMP-dev)

In addition to XMP directives (please refer [5] in detail) which enable data/task parallelization between nodes in a distributed memory system, XMP-dev directives (please refer [3] in detail) enable further data/task parallelization between one or more acceleration devices on each node under the concept of off-loading the computation to them.

Figure 1 shows an example of XMP-dev code. When a code is written in XMP-dev, XMP directives describe the distribution of data among. On each

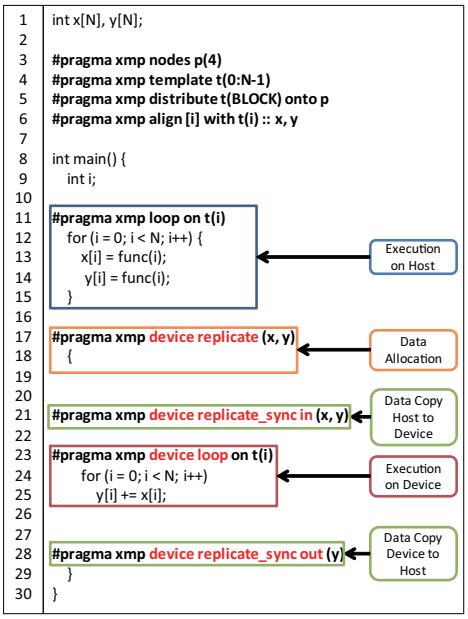


Fig. 1. An example code segment of XMP-dev

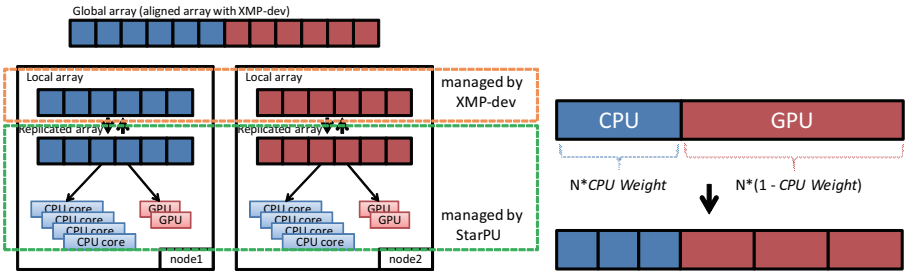
node, with XMP-dev directives (starting with "#pragma xmp device"), the user can specify the data to be allocated on GPU device, data movement between CPU and GPU, and computation offloading to GPU an shown in Figure 1. This is a large advantage for highly productive coding compared with complicated orthogonal programming with MPI and CUDA mixture, and an incremental code enhancement is possible for users as like as OpenMP.

2.3 StarPU

In this subsection, we briefly describe the concept and features of StarPU. For more details of the StarPU system, please refer [4].

StarPU is a run-time system that allocates and dispatches a collection of computations as a task to any computation resource and schedules the task execution dynamically. The target computation resources include multicore CPUs and GPUs, where each task is dispatched to a core of the multicore CPUs or to GPU device(s).

Although StarPU manages the task execution on both CPU cores and GPUs simultaneously, it is a critical issue how to decide the task size for GPUs and CPU cores to achieve high performance. The performance of recent CPUs has been increased to several hundreds of GFLOPS. However, there is still a big difference in the factor of ten between the performance of single CPU core and GPU. When there is a large number of tasks generated from a large scale data,



(a) Execution flow of XMP-dev/StarPU

(b) Image of array division with different amount for GPU and CPU

Fig. 2. Implementation of XMP-dev/StarPU

tens or more times of tasks can be allocated to GPU device while CPU cores are processing small number of them. However, it is impossible to keep the load of them when the number of tasks is limited for a small size problem, and it causes a serious situation where even the additional CPU cores become the bottleneck in total execution time.

StarPU is equipped with a feature to anneal the task size automatically when the task size (the data size associated with the task) is tuned step-by-step during the code execution in the iteration of time step simulation. However, this feature works only for cases with simple iterations of code, and it is difficult to apply this feature to a complicated execution phase of general codes. It is especially difficult when a computation node is equipped with multiple GPU devices and the performance gap between the CPU part and the GPU part is really large.

3 Dynamic Load Balancing on XMP-dev/StarPU

Our previous implementation without dynamic load balancing feature among GPUs and CPU cores are described in [1] in detail. In this section, we describe the essence of the work to understand our new feature, then introduce a dynamic load balancing feature on XMP-dev/StarPU.

3.1 Previous Implementation Strategy and Its Performance

To implement the feature of work sharing among GPUs and CPU cores for loop execution, we modified the XMP-dev compiler and the run-time system to utilize StarPU as the task scheduler and execution engine. Since the original XMP-dev/CUDA compiler is ready for data distribution and parallel execution management of basic XMP features for GPU clusters, we run StarPU on each node in the single node mode for simplicity. The XMP-dev/CUDA compiler generates a code with CUDA functions on each node, and the data distribution and synchronization among multiple nodes are performed by MPI.

Figure 2(a) shows the execution flow of XMP-dev/StarPU. The basic strategy for implementation of XMP-dev/StarPU is as follows.

- XMP-dev compiler aligns the distributed Global array to Local array for each node.
- Runtime system replicates Local array as another one named Replicated array. Local array is for communication with MPI in the context of XMP, and Replicated array is a target of task assignment and management by StarPU.
- Replicated array is allocated to StarPU's data pool and divided to a number of tasks which has the same size. Then, tasks are allocated to each device by the StarPU scheduler.
- The programmer explicitly describes the synchronization on data between Local array and Replicated array accordingly in XMP-dev syntax.

In [1], we observed that the relative performance gain by XMP-dev/StarPU to XMP-dev/CUDA is very low in many cases, at a performance with just around 45% of original XMP-dev/CUDA. This is because the task size is always constant where Replicated array on the node is always divided equally over all tasks. But the performance gap among GPU and CPU core is so large. Therefore, much larger number of tasks should be assigned to GPU than CPU core to keep the execution time balance among all computation resources. In many cases, however, there is a limit of the number of total tasks because too small size of tasks cannot be effectively executed by GPU to hide the task invocation cost including data movement between CPU and GPU. As a result, we cannot create an appropriate number of tasks to keep a good load balance between these heterogeneous resources and the efficiency of GPU execution, in a moderate size of problem. Thus, we concluded the essential problem of low performance in this work is the task size control on GPUs and CPU cores.

3.2 Improvement of XMP-dev/StarPU with Dynamic Load Balancing Feature

Previous XMP-dev/StarPU divides Replicated arrays with fixed task size whereas the performance gain by additional CPU cores to GPUs is not enough due to performance imbalance on different type of resources. We found the problem in [1] and performed a preliminary study to observe the performance behavior when the task size to be assigned CPU cores and GPUs differ to keep a good load balance. According to this study, we propose and implement a new feature for dynamic load balancing in this paper.

The key of task size control is the performance gap between CPU and GPU. In heterogeneous hybrid work sharing, CPU and GPU execution time for each task should be close or in the same order at least. On the other hand, it is required to allocate a large size of task to GPU since it has to tolerate a data transfer overhead and have a large degree of parallelism to utilize a number of cores inside the device. The best solution for this problem is to assign well

balanced task size for all GPU and CPU cores in each of Replicated array to be processed in a loop.

As the answer to this problem, we introduce a parameter to decide the balance of working set size for GPU and CPU on loop work sharing, named "CPU Weight". To simplify the control of load balance on resources, we decided to apply this value to divide a Replicated array into two parts which are processed by GPUs and CPU cores before StarPU run-time system makes further subdivision of each part of array. Figure 2(b) shows the image of dividing a Replicated array where the blue part is a relatively small portion for CPU cores while the red part is a large portion for GPUs, for the total number of array elements N. This array corresponds to the Replicated array in Figure 2(a). These two parts are further divided by StarPU to smaller size of tasks. By making the data structure appropriately to bind the task data portion and computing resource in StarPU framework, we can control the task dispatch for sub-divided data portion in each side (blue or red) so as to be correctly assigned to CPU core or GPU, respectively. In Figure 2(b) for example, StarPU divides each array into three parts. In this way, we can allocate different size of tasks to heterogeneous resources to keep the load balance controlled by CPU Weight.

Since it is difficult to set the value of CPU Weight automatically by the system, we design the system where this value is decided and set by the user explicitly in the program code. It is described by new pragma "reset_weight" as "#pragma xmp device reset_weight (*cpu_weight*)". Here, *cpu_weight* provides the CPU Weight to be applied after this point until it is reset again. The user can reset CPU Weight anywhere before entering a loop construction. There are several use cases of this feature. A user may decide to set the CPU Weight statically according to his knowledge on program behavior, or he can adaptively apply it based on the program execution behavior with any hint. Since the CPU Weight is applied to divide a Replicated array linearly into just two parts for GPU and CPU as shown in Figure 2(b), it is still difficult to find the best value of it to keep perfect load balance. One of the effective ways is to find it based on dynamic profiling of execution time by GPU and CPU. Actually, it is possible to anneal the CPU Weight during multiple time steps for most of simulation codes with time development scheme.

There is an idea to imply such an adaptive optimization of CPU Weight into the run-time system to hide it from user's view, however, we think it is very sensitive parameter to role the entire load balance and it is better to pass its control to users for various applications. For example, the user can even keep several set of CPU Weight according to different loop body, and switch them to describe the "reset_weight" pragma before entering to the different loop.

4 Performance Evaluation

We use a massively GPU cluster HA-PACS in University of Tsukuba for performance evaluation. The node specification is shown below. The CPU is Intel Xeon E5-2670 2.6 GHz (8 cores $*$ 2 sockets) with Sandy Bridge architecture, and

the GPU is NVIDIA Tesla M2090 (4 GPUs/node) with Fermi architecture. All nodes are connected with dual rails of InfiniBand QDR x4 by Fat-Tree topology with full-bisection bandwidth. In this evaluation, we use just two nodes because our purpose is to observe the behavior of adaptive load balance on each node for parallel execution. Although each node has 16 CPU cores per node, the management of each GPU consumes one thread on a CPU core in StarPU. So the number of CPU cores is "16 − (Number of used GPUs)" per node.

We evaluate two benchmark codes: N-Body and MM (Matrix-Matrix Multiplication). Since the program runs on just two nodes, the communication time on MPI which is automatically generated by XMP-dev/StarPU compiler is negligible (under 1%) for both benchmarks. Thus, we evaluate the computation time on each node including data transfer overhead between CPU and GPUs, without caring MPI communication overhead.

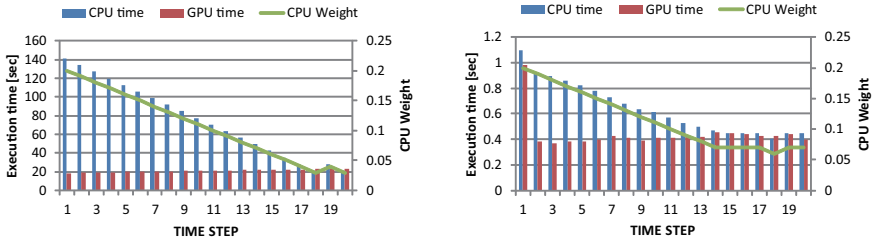
The purpose of this evaluation is to confirm the function of dynamic load balancing based on CPU Weight works correctly and to show the potential of any dynamic load balancing algorithm by the user. Therefore, we introduce a very simple annealing algorithm to modify CPU Weight dynamically in the time step development of the simulation. In all case of evaluation, CPU Weight is changed according to the computation time of each CPU core and GPU for the outer-most loop iteration. For N-Body, the outer-most loop corresponds to the time step of physics simulation. For MM benchmark, we assume that some size of matrix-matrix multiplication such as DGEMM routine in BLAS (Goto-blas [7] for CPU kernel and MAGMA blas [8] for GPU) is executed repeatedly for larger computation. Therefore, the same size of matrix-matrix multiplication is repeated in MM and the computation time of one outer-most iteration is examined.

Based on rough estimation on sustained performance of up to four GPUs and 12 CPU cores in each node, we set the initial value of CPU Weight as 0.2. The policy to decide CPU Weight in the next time step is as follows.

$ratio = T_{CPU}$ / $(T_{CPU} + T_{GPU})$;
if $(ratio > 0.5)$ cpu_weight -= 0.01;
else cpu_weight += 0.01;

Here, T_{CPU} and T_{GPU} are the task execution time by CPU and GPU in last task execution, respectively. Those values can be directly extracted by our API function. This policy shows a very simple annealing algorithm to modify CPU Weight in step by step manner. Of course, the user can describe more sophisticated algorithm in any style.

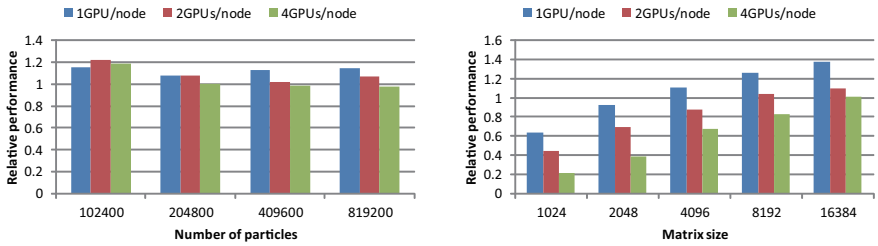
Figures 3(a) and 3(b) show the time development of execution time (both on GPU and CPU) and CPU Weight on N-Body and MM, respectively. In both figures, blue and red bars show the execution time of one task on GPU and CPU core, respectively, and green line shows the CPU Weight applied on that time step. According to the CPU Weight adjustment policy, it is decreased constantly until the execution time of tasks on both types of resources becomes nearly equal in step by step, and finally CPU Weight keeps almost constant when they are



(a) N-Body at $N = 819200$ (b) MM at $8192 * 8192$

Fig. 3. Transitions of CPU Weight

balanced. In this way, we can achieve the best balance with adaptive control of load balance among GPUs and CPU cores where the user just provides a simple notation of CPU Weight control as well as high-level PGAS programming style without any effort on describing MPI or CUDA code.



(a) N-Body (b) MM

Fig. 4. Execution time of last time step and CPU Weight

Finally, we show the total performance gain by GPU/CPU work sharing driven by our new system compared with the performance by GPU-only. Figures 4(a) and 4(b) show the relative performance to the execution with GPU-only for various cases, on N-Body and MM benchmarks, respectively. In N-Body, GPU/CPU work sharing constantly improves the performance although the ratio of gain differs in the cases. The largest gain with up to 20% is achieved for the smallest size of problem. Basically, the performance gain is reduced when the number of GPUs increases where the usable CPU cores are decreased as mentioned before. N-body benchmark is suitable for GPU computing where the sustained performance gap between GPU and CPU is large and the effect of additional CPU cores is relatively small for larger problem size. On the other hand for MM benchmark, the performance gain by GPU/CPU work sharing increases according to the problem size where the maximum gain with approximately 40% is achieved for the largest problem size using single GPU. In MM, the task ex-

ecution time for the matrix size is shorter than N-Body, and the overhead of fine task control cannot be covered by the performance of additional CPU cores. When enough size of problem is provided, the contribution of CPU cores is great to raise the total performance of computation node.

Since current implementation of XMP-dev/StarPU compiler can handle just single dimension decomposition for multi-dimension arrays, the memory utilization and data movement efficiency is low. We think that this is one of the major factors for performance limitation. We are now developing multi-dimensional array decomposition to achieve higher performance in our system.

5 Related Works

There are several compilers for GPU accelerators, for example PGI Accelerator Compile [8] and HMPP Workbench [9]. These compilers provide a directive-based language for some accelerator including GPU. HMPP Workbench use CUDA or OpenCL for backend compiler. So, it is possible to program a hybrid work sharing with GPU and CPU, by inserting some directive in user code. However, the user has to describe additional MPI code with this complicated work sharing because the compiler just cares on a single node computation. Our XMP-dev/StarPU provides a sophisticated PGAS model programming for distributed memory system.

Also, there is a work [10] that applies StarPU to BLAS (Basic Liner Algebra Subprograms) library designed for NVIDIA GPU. This work performs work sharing with GPU and CPU on library level. In the performance aspect, this work can obtain about four times performance enhancement compared with only single GPU at Cholesky decomposition by using Intel Nehalem X5560 6 cores and 3 NVIDIA FX5800. In the XMP-dev/StarPU framework, users can write a work sharing program code more flexibly, as for the problem can be written in loop distribution basically, not just with a limited function of such a library-based approach. We need to compare the performance of our approach with native implementation of MAGMA in the same class of libraries.

As the original work by StarPU research team, the load balance issue was studied[11]. From user's view point, it is difficult to describe a code directly with StarPU for GPU/CPU work sharing. Our approach is based on high-level PGAS language (XMP-dev) using StarPU as underlying supporting system. We have not modified StarPU itself to apply to our system, and our system can work as user friendly interface for StarPU feature.

6 Concluding Remarks

In this paper, we proposed and implemented a programming environment to enable GPU/CPU work sharing on our PGAS language XMP-dev compiler and run-time system to be applied to multi-GPU equipped PC clusters. The proposed system allows users to keep the load balance among GPUs and CPU cores on each node for adaptive tuning of work sharing performance. We confirmed the

effectiveness of this approach through several HPC benchmarks and achieved up to 40% of performance gain compared with original GPU-only solution. To exploit this performance gain utilizing full computation resources on each node, the user just has to add several key directives to his original serial code to describe data distribution, loop distribution, GPU/CPU data movement and synchronization, without complicated combined paradigm of MPI and CUDA.

Our future work includes applying this system for wider variety of applications and larger systems, and also improves the performance based on multi-dimensional array dividing on global array handling.

Acknowledgment

This work is partially supported by a JST-ANR Joint Project entitled "Framework and Programming for Post Petascale Computing (FP3C)" and JST/CREST program entitled "Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era", in the research area of "Development of System Software Technologies for post-Peta Scale High Performance Computing".

References

1. T. Odajima, T. Boku, T. Hanawa, J. Lee, and M. Sato. GPU/CPU Work Sharing with Parallel Language XcalableMP-dev for Parallelized Accelerated Computing. In *Sixth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, pages 97–106, Sept. 2012.
2. XcalableMP. `http://www.xcalablemp.org/`.
3. J. Lee, T. MinhTuan, T. Odajima, T. Boku, and M. Sato. An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters. In *HeteroPar '2011 (with EuroPar 2011)*, pages 429–439, 2011.
4. StarPU. `http://runtime.bordeaux.inria.fr/StarPU/`.
5. J. Lee and M. Sato. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, pages 413–420, Sep. 2010.
6. High Performance Fortran Version 2.0. `http://www.hpfpc.org/jahpf/spec/hpf-v20-j10.pdf`.
7. Texas Advanced Computing Center - GotoBlas2. `http://www.tacc.utexas.edu/tacc-projects/gotoblas2`.
8. PGI Accelerator Compiler. `http://www.softek.co.jp/SPG/Pgi/Accel/index.html`.
9. HMPP Workbench. `http://www.caps-entreprise.com/hmpp.html`.
10. E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2, Sep. 2010.
11. C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. In *Concurrency Computat.: Pract. Exper*, Mar. 2010.