DART-CUDA: A PGAS Runtime System for Multi-GPU Systems

Lei Zhou
Department of Computer Science
Ludwig-Maximilians-Universität (LMU) München
Munich, Germany
Email: zhoul@cip.ifi.lmu.de

Karl Fürlinger
Department of Computer Science
Ludwig-Maximilians-Universität (LMU) München
Munich, Germany
Email: Karl.Fuerlinger@nm.ifi.lmu.de

***Abstract*—The Partitioned Global Address Space (PGAS) approach is a promising programming model in high performance parallel computing that combines the advantages of distributed memory systems and shared memory systems. The PGAS model has been used on a variety of hardware platforms in the form of PGAS programming languages like Unified Parallel C (UPC), Chapel and Fortress. However, in spite of the increasing adoption in distributed and shared memory systems, the extension of the PGAS model to accelerator platforms is still not well supported.**

To exploit the immense computational power of multi-GPU systems, this work is concerned with the design and implementation of a Partitioned Global Address Space model for multi-GPU systems. Several issues related to the combination of logically separate GPU memories on multiple graphic cards are addressed. Furthermore, the execution model of modern GPU architectures is studied and a task creation mechanism with load balancing is proposed. Our work is implemented in the context of the DASH project, a C++ template library that realizes PGAS semantics through operator overloading. Experimental results suggest promising performance of the design and its implementation.

***Keywords*-PGAS, Partitioned Global Address Space, Multi-GPU systems, CUDA, Heterogeneous computing**

I. Introduction

Threading and message passing are the two dominant programming models on current parallel systems. Programming with threads on a shared memory system is conceptually simpler than coordinating the explicit sending and receiving of messages. However, the lack of control over data locality can hamper performance and true shared memory systems are limited in their size to a few dozen cores.

PGAS (Partitioned Global Address Space) approaches try to bring the advantages of shared memory style programming to large scale distributed systems. A PGAS language or library uses *put* and *get* operations to local and remote memory locations to provide a programming model that is very similar to programming with threads. Additionally, the locality (affinity to processing elements) of data is made explicit to enable efficient program development.

The efforts of realizing PGAS models have concentrated primarily on multi-core and multi-chip platforms based on message-passing and shared memory techniques [2], while accelerator architectures have rarely been focused on. With the widespread adoption of GPGPU (General Purpose Computing on GPUs) and generally accepted GPU programming models like CUDA and OpenCL gaining popularity, possibilities and demands for extending and unifying CPU memory spaces with GPU memory spaces in GPU-equipped systems arise in the field of high performance computing.

In this paper we explore the extension of the PGAS model to multi-GPU systems. Our work is realized in the context of DASH[1], a C++ template library that implements PGAS semantics through operator overloading. DASH is based on a PGAS runtime system API called DART and we describe the design and implementation of DART for multi-GPU systems and call our implementation DART-CUDA.

The rest of this paper is organized as follows: In Sect. II we provide an overview of DASH and DART, the context for our paper. In Sect. III we describe the design and implementation of DART-CUDA focusing in detail on the memory and execution model for multi-GPU systems. In Sect. IV we evaluate our approach by micro-benchmarks and an application case study. Related work is discussed in Sect. V and we summarize and provide an outlook on future work in Sect. VI.

II. DASH and DART

DASH [5] is a C++ template library that realizes PGAS semantics through operator overloading and provides the programmer with data structures that can be distributed over multiple nodes of a distributed memory system. As DASH does not propose a new language or depend on a custom pre-compiler or compiler it can be integrated more easily in existing applications than stand-alone PGAS languages like UPC [4] or Chapel [1]. DASH uses the concept of iterators to exploit data affinity and extends the two-level (local/remote) locality concept to a more flexible hierarchical organization. DASH is built on top of an intermediate PGAS runtime system interface called DART (the DASH runtime).

DART defines important concepts and programming entities and offers services to DASH. Most importantly, global memory allocation and access operations are realized by DART. The DART API can be implemented atop several underlying programming interfaces. A scalable implementation for distributed memory systems based on MPI-3 RMA

[1] http://www.dash-project.org

(remote memory access) operations is described in [11]. An additional DART implementation for shared memory nodes using System-V shared memory is called DART-SYSV and is the basis for the work described in this paper.

The rest of this section sets the stage by providing an overview of the most important concepts and functionality of DART, the full DART specification is available for download online: http://www.dash-project.org/dart/

The DART API can be categorized into five functional sections (Initialization and Shutdown, Unit and Team Management, Global Memory Allocation, Communication, and Synchronization), the implementation of each section in DART-CUDA is briefly described in Sect. III. The most challenging aspects of porting DART to GPU platforms are related to the memory model and the execution model, these two aspects will be discussed in detail in Sect. III-A and III-B, respectively.

The individual participants in a DASH/DART program are called *units* and DART follows the static SPMD programming model where the number of units is specified at program launch and remains unchanged throughout the entire program run. All calls to DART have to happen between *dart_init* and *dart_exit*. To start a program, either an existing job launcher is reused (e.g., mpirun) or in the case of the shared memory implementation, a custom launcher called `dartrun` is used.

In DASH/DART, units are organized in *teams*. Each program starts with a default team containing all the units in the program called `DART_TEAM_ALL` and based on an existing team, a new team can be formed by selecting a subset of the units of the old team. Teams are the basis for collective communication operations as well as collective memory allocation operations. Teams are organized in DASH in a hierarchy that can be used to exploit the hierarchical organization of a machine or algorithm.

Global memory is allocated on demand with visibility to a certain team using one of two memory allocation mechanisms. A *local-global* allocation is a non-collective call that allocates memory that is visible to all units in the program and a *team-aligned* allocation is a collective operation on a certain team that results in memory that is only accessible by members of that team. All interactions with DART global memory involve global pointers. A *global pointer* is represented as a 128 bit structure where 32 bits are identifying a unit, 16 bits are used to identify a memory segment within that unit, 16 bits are reserved for various flags, and the remaining 64 bits are used as either a virtual address or an offset.

III. Design and Implementation of DART-CUDA

The design for DART-CUDA is based on DART-SYSV. In addition to regular CPU units (those that execute on the CPU and allocate their memory from CPU memory), in DART-CUDA a *GPU unit* is an entity executing on the *CPU* that represents compute and memory capabilities of a *GPU*. I.e., a GPU unit satisfies memory allocation operations by requesting memory on the GPU using CUDA functions such as *cudaMalloc* instead of allocating CPU (host) memory.

There are several options of how we can map between units and GPU devices. Since we are free to create as many GPU units on the host side as we like, we chose a model where one unit corresponds to one particular GPU. There may, however, be several different units mapped to the same GPU. In the following we describe the extensions and adaptations have been implemented to support DART on multiple GPU devices.

Initialization and Shutdown: DART provides *dart_init* and *dart_exit* for initialization and shutdown of the runtime. Applications are started by the launcher `dartrun`, which receives command line parameters that specify the number of GPU units, and creates processes corresponding to unit entities. In DART-CUDA, the launcher further handles the initialization of GPU devices and the runtime process for task services (cf. III-B5). After being initialized by the launcher, GPU units create their own CUDA contexts so that resources of the mapped devices can be used in the subsequent computation.

Units and Teams: DART-CUDA inherits the implementation of unit and team management from DART-SYSV, and extends the support for GPU units. At the startup of a DASH application, the numbers of CPU and GPU units can be specified respectively via command line parameters. GPU and CPU units share a single global address space and can access each other's addresses.

Global Memory Management: Due to the difference between GPU memory and CPU shared memory, the memory management is significantly modified in DART-CUDA to support the integration of device memory on multiple devices. Furthermore, a buddy allocator is added to fulfill memory requests of PGAS memory over multiple GPU devices. The memory model is described in detail in section III-A.

Communication: The one-sided and collective communication functions are modified in DART-CUDA to implement the new memory model. *cudaMemcpy* and *cudaMemcpyAsync* are used to implement GPU memory access, and the collection of *cudaIPC* APIs are called to implement inter-process device memory access that realizes the semantic of global address space.

Synchronization: DART-CUDA reuses the shared memory implementation of basic synchronization mechanisms like *barrier*. For asynchronous PGAS memory access, CUDA functions for stream creation and synchronization (*cudaStreamCreate()*, etc.) are used.

In addition, an execution model was specified and implemented in DART-CUDA. This execution model is described in detail in section III-B.

A. Memory Model

The design of a PGAS memory model includes four challenges. The first one is the integration of multiple logically separate memory spaces, which makes a single global address space possible. The second challenge is the support for global access, especially on the platforms where a shared memory mechanism is absent. The third challenge is the support for common memory allocation methods adopted in PGAS programs like aligned allocation to boost performance of remote

accesses. Finally, the locality of reference has to be expressed in the design. The model should preserve locality properties of variables, while memory allocation should be performed under the consideration of device affinity. The four challenges will be discussed in more detail in the following subsection before describing some implementation details.

1) Challenges: **Unified Address Space:** Typically, a PGAS system achieves a unified global address space by means of shared memory or one-sided messaging. With regard to the GPU architecture and multi-GPU systems, this means that GPU memory spaces on different devices are to be joined, and inter-device access has to be supported. In this new model, each unit corresponds to an entity on the host (CPU) such as a process or thread, and is mapped to one GPU device and occupies the device memory space on the mapped GPU. Standalone address spaces of devices constitute the global shared address space. In other words, the global address space on a multi-GPU system is naturally partitioned by devices, and the space size of each partition is limited by the available physical memory size of respective device due to the lack of a virtual memory mechanism.

Global Memory Access: As introduced in section II, in the DASH/DART programming model all units are capable of accessing the entire shared address space via global pointers. When the global pointer points to an address actually belonging to the address space of another unit, the information helps the runtime to determine the actual address managed by the OS, and the access is performed as if it was local to the accessing unit.

With regard to multi-GPU systems, extra care is required to handle the situation of inter-device access when a global pointer points to memory on a different device. Unit ID and segment ID encoded in the global pointer provide essential information to figure out which device the address actually belongs to. The implementation details are discussed in section III-A2.

Memory Allocation: In the DASH project, two kinds of memory allocation are supported: *local-global* allocation and *team-aligned* allocation. A local-global allocation returns a global pointer to the allocated address in the global space, which is local to the calling unit, and other units can access the address through the pointer.

Team-aligned allocation allocates a memory segment in the local space of each team member with the same size, and the global pointer to the start address of the first unit is returned. Given a global pointer returned by a team-aligned allocation and an offset, global addresses in any team member's address space can be calculated by simple arithmetic. Therefore all units can work on their local memories collectively. Since the memory allocated by the team-aligned allocation spans multiple devices, the address calculation is handled by the runtime to generate the correct device address, which is introduced in section III-A2.

Locality of reference: Each Unit is assigned to one device at initialization, which means that a set of specific units are *local* to the device in terms of memory hierarchy during the life cycle of a DASH program. This affinity information is maintained by the DART-CUDA runtime, according to which calls to the DART allocation APIs can be transformed to physical memory allocation requests to the devices local to the units.

2) Implementation Details: **DART Memory Segment:** In DART-CUDA, the establishment of a partitioned global address space is done by all unit processes at the initialization phase after a DASH program starts. Unit processes are created by the root unit (unit 0). To start with, an array of segment tables is created by the root unit in shared CPU memory. Each segment table is itself an array of DART segment elements and corresponds to one unit. A DART segment represents a section of physical memory allocated in the global memory of the associated device of a unit. The DART segments along with the segment tables form the DART segment hierarchy, as illustrated in Fig. 1.

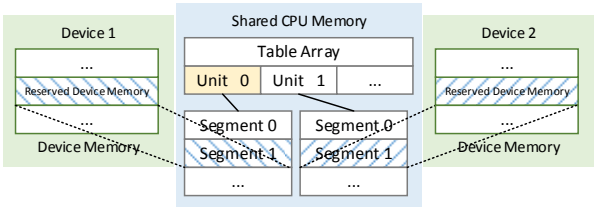


Fig. 1. DART Segment Hierarchy.

A DART segment element records essential information for mapping a segment ID to the actual physical memory region in the device memory, upon which dynamic allocations of the PGAS memory are carried out. Listing 1 shows the complete definition of the data structure.

Having the table array created at the root unit, each unit creates a segment for `DART_TEAM_ALL` as its default segment and inserts it into the respective segment table of the unit. The default segment acts like the heap memory of traditional C programs. All local-global allocations take place in the default segment of the calling unit. The size of the default segment is given as a command line option. Further segments are created and inserted into the segment tables for subsequent team-aligned allocations during the program execution.

The segment structure plays an important role in determining the physical location of an address in the global address space. Given a global pointer, a unit looks up the corresponding segment table and converts the global pointer to the device memory pointer by retrieving the owning unit ID and its DART segment ID encoded in the global pointer.

Local-global allocations are carried out in the default segment (with segment ID 0) of the unit via a per-unit buddy allocator, while a team-aligned allocation results in the creation of a new segment with a unique team ID[2] as segment ID in the segment structure and element subscript in the table

[2]In DASH, team ID does not need to be globally unique. However, if a unit is part of several teams, all these teams will have different team IDs.

at each team member, so that segments of a specified team can be easily addressed by the team ID in the segment table of each team member, and each unit gets the global pointer of its local part by simply setting the unit ID field in the returned head pointer. Thereby, the team-aligned space can be visited as if it is continuous.

```
struct dart_segment
{
  unsigned            state;
  void                *base;
  void                *end;
  int                 dev_ID;
  int                 size;
  int                 shmem_key;
  cudaIpcMemHandle_t  ipc_handle;
  dart_team_t         teamid;
  dart_buddy_t        *buddy;
  unsigned            buddy_block_size;
};
```

Listing 1. Data Structure of DART Segment.

Heterogeneous Memory Support: Based on the segment table structure, the underlying physical memory that can be managed by DART-CUDA is not limited to GPU DRAM. The implementation separates the low-level platform-dependent internal interfaces including segment management and physical memory access from the high-level platform-independent DART interfaces. When a new platform is introduced, its memory space can be integrated by realizing the low-level interfaces and utilized by applications using the DART interfaces. In this version of DART-CUDA, shared CPU memory is supported, which means the units can be mapped to GPU devices or the host. By specifying the numbers of units binding to the devices and the host respectively, a global space combining CPU and GPU memories is then created.

Buddy Memory Allocation: The DART table array and segment table realize the construction of a partitioned global address space, based on which PGAS memory allocations are carried out and managed by the runtime. The DART interfaces currently support local-global and team-aligned allocation in the global space. To overcome the external fragmentation issue caused by the memory pool technique and improve the memory utilization, a buddy memory allocation technique [8] is adopted. In DART-CUDA, the default segment of each unit is managed by a standalone buddy memory system.

Memory Access: In DART-CUDA, the DART global pointer is the only way to access the PGAS memory. As introduced in section II, a global pointer is a 128-bit address representation containing a unit id, a segment id and an offset.

The actual device address corresponding to a global pointer is not visible to DASH programs. Instead, after being returned from the DART memory allocation interfaces, a global pointer to the PGAS memory can be accessed by the DART one-sided interfaces listed in Listing 2, which implement synchronous and asynchronous `get` and `put` operations by specifying the source and destination addresses in the private space of processes. Handles returned by the asynchronous interfaces can be waited and tested using interfaces in Listing 3.

```
/* blocking versions of one-sided communication operations
   */
dart_get_blocking(void *dest, dart_gptr_t ptr,
  size_t nbytes);
dart_put_blocking(dart_gptr_t ptr, void *src,
  size_t nbytes);
/* non-blocking versions returning a handle */
dart_get(void *dest, dart_gptr_t ptr, size_t nbytes,
  dart_handle_t *handle);
dart_put(dart_gptr_t ptr, void *src, size_t nbytes,
  dart_handle_t *handle);
```

Listing 2. DART PGAS Memory Access Interfaces.

```
/* wait and test for the completion of a single handle */
dart_wait(dart_handle_t handle);
dart_test(dart_handle_t handle, int32_t *result);
/* wait and test for the completion of multiple handles */
dart_waitall(dart_handle_t *handle, size_t n);
dart_testall(dart_handle_t *handle, size_t n,
  int32_t *result);
```

Listing 3. DART PGAS Memory Synchronization Interfaces.

Address Translation: To transfer data between the private space of processes and the global space, the actual device address of a given global pointer has to be determined by the runtime. In DART-CUDA, it is done by three steps as illustrated in Fig. 2.

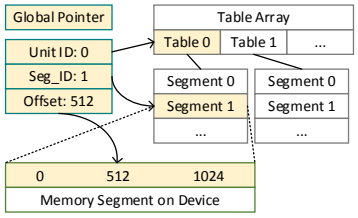


Fig. 2. Translate Global Pointer to Device Pointer.

Firstly, the segment table of the owning unit is identified in the shared table array by the $unit\_id$ in the global pointer. Then, the associated DART segment is located via the $segment\_id$, from which the target CUDA device and the base address of the segment can be retrieved. Lastly, the address offset is added to the base address to obtain the final device address, which is then used to perform get and put operations.

Address Manipulation: Manipulation of the team-aligned global pointer needs to be specially handled by DART-CUDA and an interface for incrementing global pointers has been provided in DART, as shown in Listing 4. The implementation of this function is different from that in DART-SYSV due to the existence of multiple devices and different memory layout.

```
dart_gptr_incaddr(dart_gptr_t *gptr, int offs);
```

Listing 4. DART Address Incrementing Interface.

When handling address increment requests, the runtime first checks the segment field of the given pointer. If the segment field is 0, which means the pointer points to the default segment managing local-global allocations, the offset field of the pointer is directly incremented by the offset.

The DART segment hierarchy provides a way to locate team-aligned addresses to handle address increment for team-aligned allocations. Given a team-aligned pointer, all team

segments can be accessed by visiting the element indexed by encoded team ID in segment tables of ordered team units. To compute a cross-unit address increment with a given offset exceeding the boundary of current unit, the pointer is firstly updated by incrementing the unit field in the team order accordingly. Then the offset field is updated using the remainder of the offset.

Asynchronous Memory Access: Asynchronous memory access is supported in DART-CUDA by means of CUDA streams. CUDA devices with Compute Capability 1.1 and higher are capable of concurrent copy and execution, which means the data transfer can be overlapped with the kernel execution. For each PGAS memory access, the runtime creates a new CUDA stream to execute a data transfer, and the created stream with the corresponding device ID is encoded into a handle, which is returned to the calling unit. The handle can then be waited on or tested via interfaces in Listing 3 by the unit. The host memory involved in the data transfer should be pinned memory allocated via *cudaMallocHost()* for best overlapping opportunities.

Optimization of Remote Access: When accessing a non-local global pointer, a unit firstly locates the DART segment of the owning unit, in which the IPC memory handle representing the memory block of the segment is stored. Then, the accessing unit opens the handle by calling *cudaIpcOpenMemHandle()* for GPU memory or *shmat()* for shared CPU memory to obtain the mapped address in its private process space. The handle is closed in the end as the data copy is finished to ensure that the handle can be opened again in subsequent remote accesses.

To alleviate the overheads from opening and closing handles, each unit maintains a hash table for opened handles. Each hash item contains the handle identifier and the opened pointer of the handle. Thus, repetitive accesses to a remote global pointer will no longer involve multiple handle operations. The overall performance of remote accesses is enhanced and the performance gap between local and remote accesses is hence narrowed down.

B. Execution Model

CUDA adopts a kernel-based programming model to utilize computational and memory resources of GPU devices. Commonly, kernels are called synchronously or asynchronously by CPU codes. However, when there are multiple devices used by PGAS applications, straightforwardly launching kernels by multiple process elements at the same time on one or more GPU devices could be problematic, and could cause performance issues (overheads from CUDA context switching). Extra care is required to manage task creation and management, to facilitate scheduling tasks to appropriate devices and balancing uneven workload distribution so that GPU devices can be utilized in a unified and efficient way. A tasking API called TaskAPI is proposed as an execution model extension to the DASH programming model. It supports both CPU and GPU tasks. Tasks created by TaskAPI are managed by the runtime of DART-CUDA, and can be manipulated by units via task handles returned by TaskAPI. Both synchronous and asynchronous tasks are supported in this implementation.

TaskAPI covers the essential operations on tasks including synchronous and asynchronous task creation, blocking wait on tasks, non-blocking test for task completion and task cancellation. When creating a task via *dart_tapi_task_create()*, the pointer to a CPU function or a CUDA kernel to be executed has to be specified. Global pointers in and out and the size of the input and output data are also required to tell the runtime how to access input and output parameters. Besides the basic tasking functionality, advanced features including load balancing and overlap of data transfer and execution are implemented inside the DART runtime. Notably, only tasks created by TaskAPI are managed by the DART runtime. If the kernel function is launched by units directly rather than by TaskAPI, its execution will not be managed.

TaskAPI complements the original execution model of DART-SYSV with task parallelism and essential tasking functions. However, the peculiarity of the GPU architecture and the GPGPU programming model requires supplementary components to make the new model possible.

1) Task Representation: In DART-CUDA, GPU tasks are implemented on the basis of kernel functions. However, device memory used by a kernel functions is not visible to DASH programs due to the abstraction of the PGAS memory model, and unit processes have no knowledge about which device their data reside in. It is thus not feasible that unit processes interact with CUDA devices and device memories directly. Instead, tasks are created as DART task items and then delegated to the DART-CUDA runtime for scheduling and launching, which requires extra information to enable access to PGAS memory that can be accessed only by DART global pointers, to locate target device and to specify execution configurations of the launching kernel.

2) Overlapping of Data Transfers and Computation: NVIDIA GPU chips of the Kepler Microarchitecture have the capability of 2-way or 3-way concurrency depending on the device model. Copy operations can be overlapped with kernels running on different steams. This characteristic enables the overlapping of data transfers and computing. DART-CUDA takes advantage of this by creating new streams for all kernel execution and reserving dedicated streams on each device for data operations like data transfer, data migration and result write-back. In this way, it is ensured that data operations are always overlapped with running kernels on devices. These streams are stored in the task handle structure so that all stream-related operations can be carried out by the unit process owning the task handle.

3) TaskAPI Process Model: In the shared memory version of DART, a DASH application begins with forking a specified amount of unit processes to establish a SPMD process model. DART-CUDA extends this model by adding one more TaskAPI runtime process to facilitate the proposed tasking functions. The runtime process is forked from the root unit process after all unit processes are ready, which means the address space of the runtime process is identical with that of all others. In this

way, the task function pointers representing kernel functions or CPU functions remain valid in the address space of the runtime process and can be dereferenced correctly. The runtime process also creates a series of functional threads to implement features like scheduling, kernel launching and load balancing. A shared area keeping task information and task queues is allocated in shared CPU memory so that unit processes can enqueue created tasks and communicate with the runtime process.

4) Queuing and Scheduling: To coordinate task execution and achieve better performance in a multi-GPU environment, DART-CUDA implements queuing and scheduling functionality. Started tasks are not launched on devices all at once. Instead, they are enqueued into task queues with different priorities and scheduled by the runtime. Furthermore, the workloads of devices are monitored, and tasks without data affinity can be scheduled to idle devices to improve overall utilization.

In DART-CUDA, FIFO queues are employed to manage started tasks. Fig. 3 shows the design of the queuing system. `Sync Task Queue` and `Async Task Queue` are two main queues for blocking and non-blocking tasks. Each device is associated with a `Dev Queue` for tasks scheduled from the main queues by the runtime. All task items are retrieved via their task handles stored in the task pool.

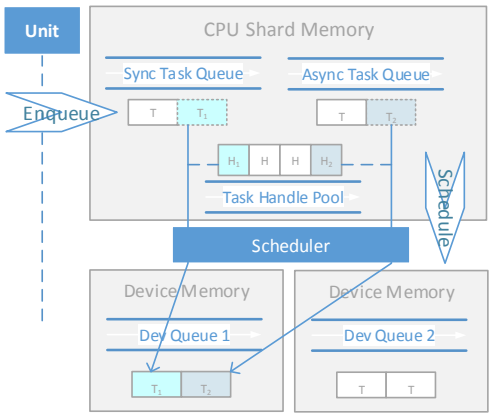


Fig. 3. Queuing and Scheduling in DART-CUDA.

Task items of all started tasks are firstly enqueued into the main queues. The two main task queues reside in a CPU shared memory area and are used to queue synchronous and asynchronous task respectively. Since the main task queues are accessed by multiple unit processes, pthread mutex locks are used to avoid race conditions. When starting a task, the unit process acquires the lock of the main queues and pushes the task item in when the lock is released. Device task queues reside in device-mapped memory allocated via *cudaHostAlloc()*. A daemon scheduler thread created by the TaskAPI runtime process at program startup continuously polls the main queues and schedules tasks to proper device queues. `Sync Task Queue` has a higher priority than `Async Task Queue` by design to ensure lower latency for synchronous tasks.

As shown in Fig. 3, the base pointer of a task handle is changed as soon as a task is scheduled to a device queue, and the task handle pointer of a task item always points to the unique task handle in the task handle pool. Therefore, even if a task that has been started is scheduled to a device queue or is migrated (described in section III-B6), the system is able to cancel or wait for the task execution via streams stored in the unique task handle in the task handle pool.

5) Task Launching: In a normal CUDA program, kernels are launched on-demand with execution configurations specified and target CUDA device set up, and the CUDA runtime is responsible for executing the kernels on the device, whereas in DART-CUDA kernel functions and execution configurations are stored in task items, and the device selection should be handled based on information stored in task items automatically.

Apart from the scheduler thread in charge of dispatching tasks from the main queues to the device queues, multiple kernel launcher threads are created to extract information essential to kernel launch from task items and prepare CUDA devices. Each kernel launcher thread corresponds to one device and takes task items dequeued from the device queue. Data pointers and necessary arguments like execution configurations required for kernel launch are fetched from the task item. Finally, the launcher switches the CUDA runtime to the device specified by the task item and launches the task kernel by dereferencing the kernel function pointer with prepared data pointers and arguments.

To support synchronization of task execution, a new CUDA stream is created prior to launching each task kernel, with which the kernel is executed on the device. The stream is then written into the task handle so that the application can wait for the kernel execution later on.

6) Load Balancing: The aforementioned concept of DASH teams and the programming model derived from it make it easy to exploit computational resources with hierarchical locality, but may lead to imbalanced workload distribution.

To overcome potential workload imbalance, a load balancing thread monitors all device task queues throughout the lifetime of a DASH application. As it detects that the task queue of some device becomes empty, it attempts to steal a task item from the task queue of the busiest device to the idle device with vacant task queue and migrate the task data.

First, temporary task memory is reserved by allocating ad-hoc memory space on the idle device for both input and output data, which is outside the PGAS memory. Afterwards, the input data is copied from the original device to the ad-hoc memory on the idle device. At last, device pointers for input and output data encoded in the task item are accordingly changed. The data copy is asynchronous and carries out on a dedicated data stream, which can be examined by the task handle of the task item. The kernel execution will wait on the data migration process to ensure the validity of the task data.

The task migration involves two steps. Firstly, the task item is dequeued from the device queue of the busy device and pushed into the device queue of the idle device. The second step creates a write-back task on the idle device. When a kernel

launcher performs a write-back task, result data will be written back to the original place after the task completes to make sure the subsequent tasks working on the result can proceed seamlessly.

IV. Evaluation

The implemented PGAS memory component and TaskAPI component for multi-GPU systems have been integrated into the DART-CUDA prototype based on the latest CUDA 6.5 toolkit and DART-SYSV 1.0. A two-GPU test system is constructed with an Intel i5-750 quad-core processor running at 3.3 GHz, 8 GB RAM and two NVidia GTX 750 Ti graphic cards, each of which has 640 CUDA cores and 2 GB on-board DRAM. The devices are connected to the system via one PCIe 2.0 x16 interface with a theoretical bandwidth of 8 GB/s and one 4 GB/s PCIe 1.0 x16 interface. The system runs Linux 3.15 with NVidia Driver 340 and CUDA 6.5 runtime installed.

A. Micro-Benchmarks

1) Methodology: Five micro-benchmarks have been designed and tested to measure the performance and overheads of the DART-CUDA library and CUDA runtime. Different unit configurations are applied to show the performance penalty caused by concurrent operations. In tests with more than two units, the two GPU devices are assigned to units in a round-robin way (1^{st} unit with GPU_1, 2^{nd} unit with GPU_2, 3^{rd} unit with GPU_1, etc.).

2) Results of Memory Allocation Functions: Fig. 4 presents the results of the micro-benchmark of *dart_memalloc()* (a local-global allocation). For each unit configuration, memory areas with increasing sizes with a stepping of 40,000 bytes are allocated at each unit in its default segment ($segment[0]$). The X axis is logarithmic scaled. As can been seen, the time latency remains almost identical for tests with 1 to 4 units at all data sizes, and gets larger for cases with more than 4 units proportional to the unit amount. It can also be concluded that, as the allocating size increases, the time cost decreases constantly and keeps steady in the intervals of every two 2^n numbers for tests with 1 to 4 units.

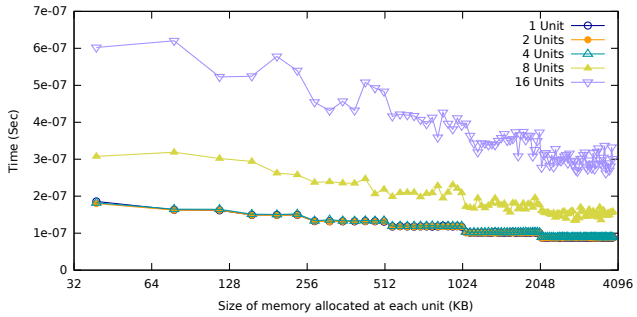


Fig. 4. Latency of Local-Global Allocation.

The trends reflect the characteristic of the buddy memory allocator as expected. If an allocation has a size of power of 2, the allocator always finds a suitable block within $\log_2 n$ steps so long as it is available. The overall performance approaches a constant as the size increases, which explains the stepped lines presented in Fig. 4. Since *dart_memalloc()* does not allocate physical memory directly but performs the allocation algorithm in reserved memory area managed by the buddy allocator, the numerical values of all tests are relatively small with a magnitude of 1×10^{-6}. Lastly, tests with 1 to 4 units in both groups produce the same latencies because each unit has its own buddy allocator in its private process space, and the test system equipped a quad-core CPU, which means up to four processes can be executed with genuine concurrency, and no communication and synchronization is involved during the procedure. For tests with more than 4 units, executions on the CPU are interleaved by multiple processes, which leads to larger latencies on average.

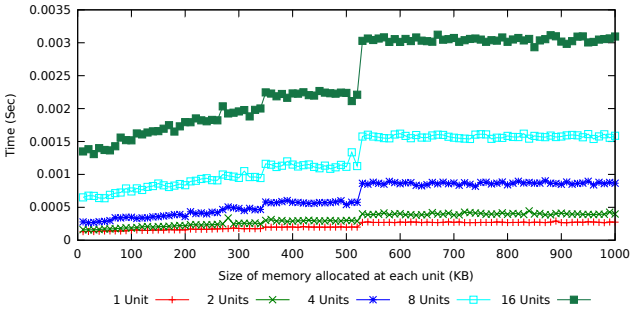
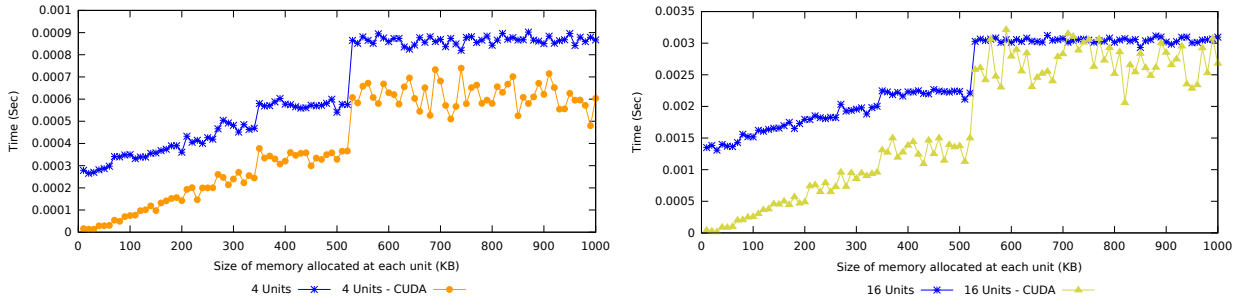


Fig. 5. Latency of Team-Aligned Allocation.

The benchmark result of function *dart_team_memalloc_aligned()* is shown in Fig. 5. In this experiment, all member units call the function collectively and create a team-aligned memory segment. In all unit configurations, the latency increases steadily with a small slope, and has a leap at 530 KB altogether. The latency values are about 1×10^{-3} second, which is about three orders of magnitude greater than those of *dart_memalloc()*. The first explanation is that the function is implemented by invoking CUDA API and allocating GPU DRAM directly, and underlying hardware operations are involved. Secondly, team operations require waiting on locks of synchronized team structures, and cases with more than one unit all suffer performance deterioration regardless of the concurrency provided by the multi-core CPU.

Fig. 6 further explores the performance trends of native CUDA allocation API and compares them with those of the DART interface. As shown in Fig. 6(a), the two libraries provide similar performance trends under a 4-unit configuration, and the overhead introduced by DART-CUDA remains stable, while in a high concurrency scenario with 16 units shown in Fig. 6(b), the native CUDA API outperforms at small sizes and gets close to DART for large size tests. It can also be inferred that the steep increase at 530 KB is due to the CUDA library and runtime since all four lines show the same change.

3) Results of Memory Access Functions: Fig. 7 illustrates the micro-benchmarks of blocking `get` and `put` operations. The test involves two units serially accessing a global pointer



(a) Test with 4 Units

(b) Test with 16 Units

Fig. 6. Comparison of DART Team Aligned Allocation and CUDA Allocation.

allocated at the first unit. Since the pointer is local to the first unit, its access to the pointer is local and the second unit accesses the pointer remotely. The optimized version of remote access is measured here. Slight performance gaps between local accesses and remote accesses can be observed, which are mostly due to the bandwidth difference between intra-device and inter-device communications.

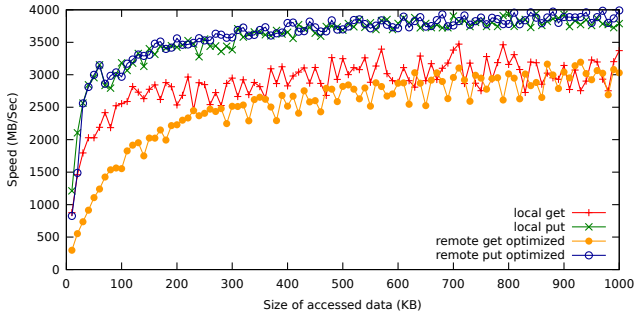


Fig. 7. Performance of Local and Optimized Remote Access.

B. Case Study

Three study cases have been developed to demonstrate the usage of TaskAPI provided by DART-CUDA and to evaluate the programmability of the model compared to the common parallel programming models. In the first example, a conventional MPI-based five-point stencil code is rewritten into a DASH program taking advantage of global memory access and tasking mechanism to show the programmability of the GPU tasking model. It replaces two-sided MPI communication with one-sided PGAS global memory access and delegates decomposed stencil computing tasks to managed GPU devices. The second case further revises the stencil code by distributing stencil computing on both CPU and GPU devices to demonstrate how DART-CUDA can exploit both storage and computing resources of a CPU-GPU system. The last one creates a imbalanced workload scenario and demonstrates the effectiveness of the load balancing mechanism.

1) Results: 100 stencil iterations are carried out for each test to get the average duration of one iteration. The CPU implementation involves four units. For GPU tests two units are created, each one corresponding to one GPU device. The total elapsed time is gathered from each synchronized iteration. Fig. 8 shows the elapsed time of one stencil iteration of GPU based and CPU based implementation on 2-D five point stencil problems with increasing sizes. When the problem scale is smaller than 4000×4000, the CPU implementation prevails since the performance gain from GPUs does not compensate the cost of kernel launch on GPUs and inter-device data transfer. As the problem size increases, the elapsed time of CPU implementation grows exponentially, while the GPU implementation outperforms and the speed ratio grows up to 42 times as shown in Fig. 9.

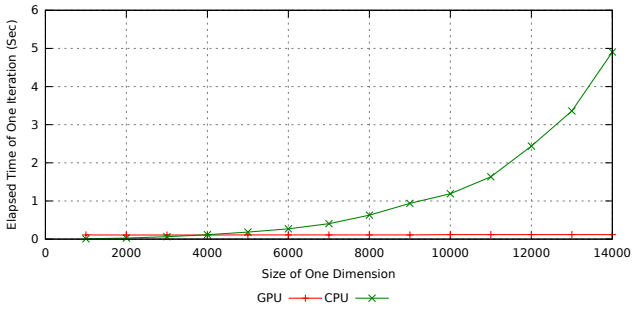


Fig. 8. Performance of 2-D Stencil Code on CPU and GPU.

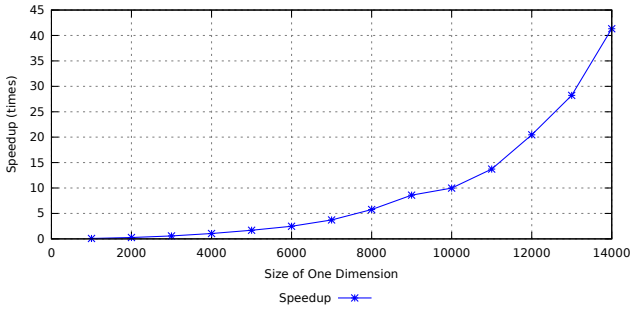


Fig. 9. Speedup between CPU and GPU.

Due to limited memory capacity, problems with size larger than 14000×14000 can not be tested on the system. To obtain the performance trend of the GPU implementation, simulation tests are performed for problems with sizes larger than 14000, in which only the computation is executed and

measured without memory operations and data exchanges. To help understand the composition of the latency, the overhead of kernel launch on the devices is also measured by launching void kernels with execution configuration matching the problem scale. Fig. 10 shows the results of the estimation. The red line represents the time consuming estimation for large problems, indicating that the GPU implementation also has a quadratic growth starting from circa 15000, while the green line stands for the overhead part of the total elapse, which also shows a stepped quadratic growth with small factor. The quadratic growth is not observed in the interval from 0 to 15000 for two reasons. Firstly, the latency of devices to launch a small number of kernels for small problems results in a minimal overhead that dominates the elapsed time, by which the theoretical quadratic growth is weakened to a nearly linear increase. As the problems become larger, the proportion of the overhead gradually reduces and the quadratic part becomes more and more significant. The second reason is that the massive concurrency of GPU devices hides the expected growth for problems too small to utilize all device resources.

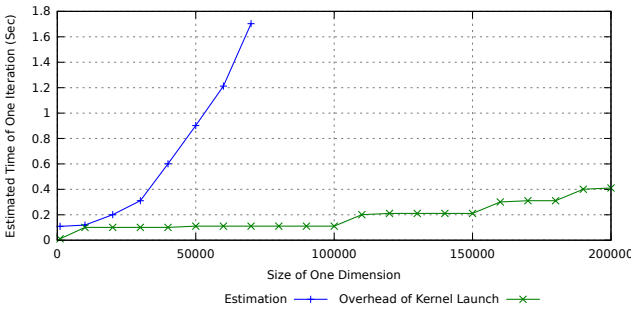


Fig. 10. Estimation of Large Size Problems.

The results of CPU-GPU cooperative approach is illustrated in Fig. 11. By involving four CPU units and two GPU units simultaneously, the overall performance of the CPU-GPU approach is effectively improved for tests of size larger than 13000, comparing to the CPU-only approach. The speedup has a steady increase and approaches 2.15 at size 20000.

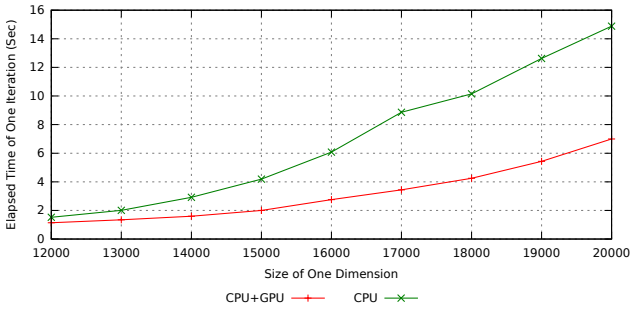


Fig. 11. Performance of CPU-GPU Cooperative 2-D Stencil Code.

Fig. 12 presents the cost for exchanging boundary elements of the two approaches. Due to the bandwidth limit posed by the system bus and PCIe interfaces, memory accesses between CPU and GPU or two GPUs result in 20 to 24 times larger latencies than that of accesses within CPU memory, which grow linearly with the increase of size. Combining the results of Fig. 12 and Fig. 11, the performance benefit is still considerable in spite of the relatively high data transfer cost.

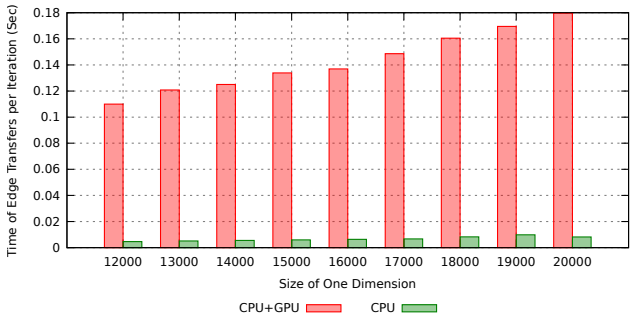


Fig. 12. Cost of Inter-Unit Boundary Exchange.

In the load balancing test, one CPU unit is launched with different granularity of task decomposition (5-way, 10-way and 20-way). Since the two GPU devices are idle at the beginning, the load balancer intermittently migrates the decomposed subtasks and their data from the CPU queue to the queues of GPUs, and writes back results produced by the migrated subtasks. Since the data migration and write-back procedures cause considerable overhead, the overall performance does not benefit from the load balancing for cases smaller than 8000, as shown in Fig. 13. For larger cases, the performance gain from the GPUs grows big enough to countervail the overhead and accelerate the overall computing up to 25%. In addition, the decomposition granularity is an important factor. It can be seen that 5-way decomposition results in the lowest performance loss in tests smaller than 8000, but also the smallest performance boost in large tests, and is outperformed by 10-way and 20-way decomposition as the problem size increases. Lastly, it can be expected that, tasks requiring smaller data and more computation are preferable to be migrated because they lead to lower overhead and higher gains from accelerators like GPUs. Therefore, the strategy of load balancing can be improved by evaluating relevant task information and maximizing the performance gain.

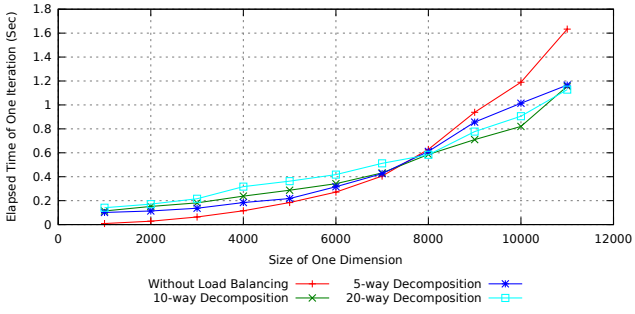


Fig. 13. CPU Stencil Code with GPU Load Balancing via Task Decomposition.

V. Related Work

Recently, there has been some attention focused on implementing PGAS models on GPU platforms. Jacobsen et al. constructed a heterogeneous cluster within which each node equipped with multiple GPUs [7], and developed an application for flow computation employing a mixed MPI-CUDA implementation.

In [10], Zheng et al. gave an overview of the effort extending UPC for GPU computing in a PGAS programming model. The concept of a hybrid partitioned global address space was proposed, in which each thread has only one shared segment either in host memory or in GPU memory. The extension is backward compatible with current UPC and CUDA/OpenCL programs and takes advantage of one-sided communication in UPC. The UPC runtime was extended to manage shared heap on GPU devices. GASNet was accordingly modified to facilitate one-sided communication on GPU by introducing GPU task queue and Active Message.

Similarly, Chen et al. extended UPC to GPU clusters with hierarchical data distribution and augmented the compiler implementation [3]. The execution model of UPC was revised by combining SPMD with fork-join model. The compiling system was implemented with several memory optimizations targeting NVIDIA CUDA like affinity-aware loop tiling transformation and array reuse degree. Unified data management eliminates redundant data transfer and data layout transformation at runtime.

In [9], Lee et al. extended the XcalableMP PGAS Language to GPGPU and supported multi-node GPU clusters. It adopted an OpenMP-like directive based programming model to minimize the need of modification, and the compiler was correspondingly modified to add support for CUDA compiler.

Garland et al. introduced Phalanx in [6], a unified programming model for heterogeneous systems with a prototype implementation supporting CUDA for many-core GPUs, OpenMP for multi-core CPUs and GASNet for clusters. The model assumes distributed hybrid machines with heterogeneous collections of processors and hierarchal memory. A correspondingly optimized task model was designed to support multi-thread tasks with hierarchal organizations on the basis of a PGAS-like memory model. By means of templates and generic functions, the model was realized in the form of a C++ library, which avoids the need for extra compiler supports and meanwhile keeps good interoperability with legacy code.

VI. Conclusion

As a novel programming model and promising solution to massively parallel computing, the PGAS programming model achieves a balance between the scalability, the performance and the programming productivity. This work extends the PGAS concept to the thriving GPGPU platform in terms of two aspects. Firstly, A memory model specialized for multi-GPU systems is proposed. The implementation of the model is compatible with GPU memory and CPU memory, and supports the combined usage of both CPUs and GPUs. Second, a new execution model is proposed to exploit data parallelism of single GPU devices and explore task parallelism of multiple devices. The TaskAPI implements this model and provides a productive approach to creating and performing platform-specific tasks without the need of interacting with the underlying hardware.

The performance of the memory access of the PGAS model is evaluated via several micro-benchmarks. The buddy allocator delivers much lower latency than direct allocation via CUDA APIs. With regards to memory access speed, the implementation shows promising performance and acceptable overheads compared to the performance of CUDA runtime. The study cases present the application of TaskAPI. Comparing to the traditional MPI model, TaskAPI provides better programmability with performance boost from data parallelism. A CPU-GPU cooperative application is explored to attest the feasibility of writing PGAS codes with a multi-platform programming model in a hybrid system. The effectiveness of the load balancing mechanism is also verified in the scenario of imbalanced workload distribution.

Acknowledgment

We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

[1] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, August 2007.

[2] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. ACM.

[3] Li Chen, Lei Liu, Shenglin Tang, Lei Huang, Zheng Jing, Shixiong Xu, Dingfei Zhang, and Baojiang Shou. Unified parallel c for gpu clusters: Language extensions and compiler implementation. pages 151–165, 2011.

[4] UPC Consortium. UPC language specification v1.2. June 2005. Technical Report, Lawrence Berkeley National Laboratory.

[5] Karl Fürlinger, Colin Glass, Jose Gracia, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. DASH: Data structures and algorithms with support for hierarchical locality. In *Euro-Par Workshops*, 2014.

[6] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[7] Dana A Jacobsen, Julien C Thibault, and Inanc Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. 16, 2010.

[8] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[9] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. An extension of XcalableMP PGAS lanaguage for multi-node gpu clusters. pages 429–439, 2012.

[10] Yili Zheng, Costin Iancu, Paul Hargrove, Seung-Jai Min, and Katherine Yelick. Extending unified parallel C for GPU computing. In *SIAM conference on parallel processing for scientific computing*, 2010.

[11] Huan Zhou, Yousri Mhedheb, Kamran Idrees, Colin Glass, Jose Gracia, Karl Fürlinger, and Jie Tao. In *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*, October 2014.