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

Lei Zhou,
Karl Fürlinger

presented by
Matthias Maiterth
Ludwig-Maximilians-Universität München (LMU)
Munich Network Management Team (MNM)
Institute of Computer Science
DASH is a data-structure oriented C++ template library that realizes the PGAS (Partitioned Global Address Space) model.

```cpp
dash::Array<int> a(1000);
da[23]=412;
cout << a[42] << endl;
```

- Array `a` can be stored in the memory of several nodes
- `a[i]` transparently refers to local memory or to remote memory via operator overloading

Not a new language to learn
- Can be integrated with existing (MPI) applications

Support for hierarchical locality
- Team hierarchies and locality iterators

Not a new language to learn
- Can be integrated with existing (MPI) applications

Support for hierarchical locality
- Team hierarchies and locality iterators
DASH – Overview and Project Partners

- **Funding:**
  - DFG priority programme: “Software for Exascale Computing” (SPPEXA)

- **Project Partners:**
  - LMU Munich (K. Fürlinger)
  - HLRS Stuttgart (J. Gracia)
  - TU Dresden (A. Knüpfer)
  - KIT Karlsruhe (J. Tao)
  - CEO DE Beijing (L. Wang, associated)
DART: The DASH Runtime Interface

- The DART API:
  - Plain-C based interface
  - Follows the SPMD execution model
  - Defines **Units** and **Teams**
  - Defines a **global memory** abstraction
  - Provides a **global pointer**
  - Defines one-sided access operations (puts and gets)
  - Provides collective and pair-wise synchronization mechanisms
Memory Allocation and Access

- **Symmetric and team-aligned allocation**
  - The same memory is allocated at each unit and each member of the team can easily compute the address of any location in any unit’s part of the allocation

- **Local global allocation**
  - Globally accessible, no alignment guarantees, tied to DARTTEAM_ALL
Three DART Implementations

- **DART-MPI**
  - Uses MPI-3 RMA
  - Scalable runtime

- **DART-SYSV** shared-memory based implementation
  - For shared-memory nodes only
  - Proof of concept & testing of DASH

- **DART-CUDA** extends DART-SYSV with support for accelerators
  - Research vehicle for the next iteration of the DART interface (execution model)

<table>
<thead>
<tr>
<th>DART API</th>
<th>DASH C++ Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>DART-MPI</td>
<td></td>
</tr>
<tr>
<td>DART-SYSV</td>
<td></td>
</tr>
<tr>
<td>DART-CUDA</td>
<td></td>
</tr>
</tbody>
</table>

Today
DART interfaces provide abstractions of the PGAS model

- **Unit/Team**
  - `dart_myid();`
  - `dart_size();`
  - `dart_team_create();`
  - `dart_team_destroy();`
  - `dart_team_size();`
  - ...

- **Global Pointer**
  - `dart_gptr_getaddr();`
  - `dart_gptr_setaddr();`
  - `dart_gptr_incaddr();`
  - `dart_gptr_setunit();`
  - ...

- **Memory allocation**
  - `dart_memalloc();`
  - `dart_memfree();`
  - `dart_team_memalloc_aligned();`
  - `dart_team_memfree();`
  - ...

- **Memory Access**
  - `Dart_get_blocking();`
  - `dart_put_blocking();`
  - `dart_get();`
  - `dart_put();`
  - ...

- **DART-CUDA** implements the interfaces for multi-GPU platforms.
Design of PGAS Memory Model on the GPU

- **Unified Address Space**
  - Units share a global address space.
  - The global space is partitioned by units.
  - Each unit is assigned to one device (CPU or GPU).
  - Team aligned allocation can span multiple devices.
  - Uses Segments.

- **DART Segment**
  - basic allocation unit.
  - corresponds to a physical memory segment
  - contains device info, base pointer and IPC handle for remote access.
Implementation of PGAS Memory Model

- **Local-Global Allocation**
  - is performed on the default segment. (preallocated on initialisation)
  - The default segment is managed by a buddy allocator per unit.
  - *Offset* represents the memory location.

- **Team-Aligned Allocation**
  - Creates a new DART segment at each team member.
  - *Segment ID* represents the memory location.
  - *Segment ID* is unique in terms of multiple sub-team membership
Implementation of PGAS Memory Model

- **Address Translation Procedure** *(necessary Overhead for Memory Access)*

1. Retrieves the *unit ID* and accesses its segment table.
2. Accesses the target DART segment by *segment ID*.
3. Local access: base pointer of the **default** segment + offset
   Remote access: base pointer of the **target** segment + offset

- **Optimization**
  - Units cache IPC handle of remote accesses.
  - A hash table is maintained at each unit.
  - One IPC operation per DART segment required.
Design of DART-CUDA Task Model

- Task Model needed to keep GPU-devices busy
  - Mapping of units to Device (CPU/GPU)
  - Task Model supports task scheduling for CPU and GPU.
  - Units can schedule tasks (i.e. CUDA-Kernels)

- Functional Overview:
  - Two Schedulers for asyn/syn queue.
  - One Load Balancer
  - One kernel launcher per device
Evaluation

Environment

- A two-GPU test system
- Intel i5-750 quad-core processor running at 3.3 GHz
- 8 GB RAM
- Two NVidia GTX 750 Ti GPUs,
  - 640 CUDA cores and 2 GB on-board DRAM
  - one 8 GB/s PCIe 2.0 x16 interface;
  - 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.

Tests and Measurements:

- Local-Global and Team-Aligned Memory Allocation
- Local and remote Memory Access
- Simple Stencil Code
Shows the effectiveness of the buddy allocation. (preallocated)

Function call: dart_memalloc()
Team-aligned Allocation calls CUDA memory functions.

- Synchronization necessary
- Function call: `dart_team_memalloc_aligned()`
CUDA vs. DART-CUDA overhead – the 530KB leap

- Top graph: Comparison of time (Sec) for different sizes of memory allocated at each unit (KB) for 4 units. CUDA (blue) and DART-CUDA (orange) are compared.
- Bottom graph: Comparison of time (Sec) for different sizes of memory allocated at each unit (KB) for 16 units. CUDA (blue) and DART-CUDA (orange) are compared.
Performance-Gap minimized but still there.
Case Study – Simple Stencil Code

Out of Device Memory with larger Dimensions ➔ CPU+GPU test
Case Study – Simple Stencil Code

- Cost of transfer boundary elements between units.
- Global Accesses require CUDA memory transfers.
- Load balancing via Task Decomposition possible
- Optimal over-decomposition depends on Problem.
DART: Final v1.0 spec
- Available online: http://www.dash-project.org/dart/
- DART can be the foundation for other PGAS approaches
- Next iteration:
  - General execution model
  - Integration of DART-CUDA and DART-MPI

DASH
- DART-MPI + DASH release in the works (array and matrix)
- Next iteration: dynamic data structures

Thank you for your attention!