The APGAS Programming Model for Heterogeneous Architectures

David E. Hudak, Ph.D.
Program Director for HPC Engineering
dhudak@osc.edu
Overview

• Heterogeneous architectures and their software challenges

• APGAS introduction
  – PGAS – Partitioned Global Address Space
  – Asynchronous PGAS – X10 and Chapel

• Targeting heterogeneous architectures with Asynchronous PGAS
  – Supporting CUDA in X10
  – Global views for accelerators in Chapel
Heterogeneous Architecture Highlights

• Products from major manufacturers:
  – AMD: *Fusion* architecture
  – nVidia: *Tesla* GPGPU, Denver architecture
  – Intel: *MIC* architecture – Knights Corner

• New technology with many tradeoffs remaining
  – Special-purpose (GPU) vs. General-purpose (MIC)
  – Integration at I/O device level (PCI), socket level (e.g., QPI) or chip-level (e.g., Fusion)

• Notable impact in high-performance computing
  – Three of top five systems in Top 500 use GPUs (November 2011)
  – Future systems announced: ORNL Titan (nVidia GPUs) and TACC Stampede (Intel Knights Corner)
Heterogeneous Cluster Configuration

- Oakley GPU partition
  - Subset of bigger cluster
- 64 nodes linked by QDR
- Each node features dual
  - 6-core Westmere EP
  - 24 GB RAM
  - PCIe Gen 2
  - nVidia M2070
- Each nVidia M2070
  - 448 CUDA cores
  - 6 GB RAM
Heterogeneous Programming Challenges

• Short-term challenges
  – Device-driver style
    • Allocate buffer on device
    • DMA transfer into buffer
    • Call function on device
    • Poll for function completion
    • DMA transfer from buffer
  – Limited accelerator memory
  – Multiple toolchains

• Long-term challenges
  – Express multiple levels of parallelism (task and data parallelism)
  – Synchronize data motion across multiple memories
  – Regularly overlap computation with communication
Heterogeneous Programming Challenges

• HPC node architectures will be increasingly
  – Complicated (e.g., multicore, multilevel caches, RAM and I/O contention, communication offload processing)
  – Heterogenous (e.g, parallelism across nodes, between motherboard and devices (GPUs, IB cards), among CPU cores)

• Achieving high performance requires detailed, system-dependent specification of data placement and data movement
PGAS Background: Global and Local Views

• A parallel program consists of a set of threads (or tasks) and at least one address space.

• A program is said to have a global view if all threads share a single address space (e.g., OpenMP, Cilk Plus)
  – Tough to see when threads share same data
  – Bad data sharing causes race conditions (incorrect answers) and communication overhead (poor performance)

• A program is said to have a local view if the threads have distinct address spaces and pass messages to communicate (e.g., MPI)
  – Message passing code introduces a lot of bookkeeping to applications
  – Threads need individual copies of all data required to do their computations (which can lead to replicated data)
**PGAS Overview**

- **“Partitioned Global View” (or PGAS)**
  - **Global Address Space**: Every task sees entire data set, so no need for replicated data
  - **Partitioned**: Divide global address space so programmer is aware of data sharing among threads

- **Implementations**
  - GA Library from PNNL
  - SHMEM
  - Unified Parallel C (UPC), FORTRAN 2009
  - X10, Chapel

- **Concepts**
  - Memories and structures
  - Partition and mapping
  - Threads and affinity
Memories and Distributions

• Software Memory
  – Distinct logical storage area in a computer program (e.g., heap or stack)
  – For parallel software, we use multiple memories

• In X10, a memory is called a place

• Structure
  – Collection of data created by program execution (arrays, trees, graphs, etc.)

• Partition
  – Division of structure into parts

• Mapping
  – Assignment of structure parts to memories

• In X10, partitioning and mapping information for an array are stored in a distribution
APGAS Languages

• Cray’s Chapel is an instance of the Asynchronous PGAS model implemented in a new programming language

• IBM’s X10 is an instance of the Asynchronous PGAS model implemented in the Java family
  – X10 in HPC: X10 is compiled to C++ (and CUDA)
    • Communication: DMA for GPU, MPI for clusters, LAPI/PAMI for Blue Gene/PERCS (former Blue Waters)

• Both projects provide language constructs for parallelism and locality
  – Parallelism: `async` (X10) and `begin` (Chapel)
  – Locality: `places` (X10) and `locales` (Chapel)
Multiple Activities and Places in X10

• Activities
  – All X10 programs begin with a single activity executing `main` in place 0
  – Create/control activities with `at`, `async`, `finish`, `atomic` (and many others!)

• Places hold activities and objects
  – `class x10.lang.Place`
    • Number of places fixed at launch time, available at `Place.MAX_PLACES`
    • `Place.FIRST_PLACE` is place 0
  – Launch an X10 app with `mpirun`
    • `mpirun -np 4 HelloWholeWorld`
    • Places numbered 0..3
class HelloWholeWorld {
    public static def main(args:Array[String](1)):void {
        for (var i:Int=0; i<Place.MAX_PLACES; i++) {
            val iVal = i;
            async at (Place.places(iVal)) {
                Console.OUT.println("Hello World from place "+here.id);
            }
        }
    }
}

Hello World from place 0
Hello World from place 2
Hello World from place 3
Hello World from place 1

• **at** – place shift
  – Shift current activity to a place to evaluate an expression, then return
  – Copy necessary values from calling place to callee place, discard when done

• **async**
  – start new activity and **don’t wait** for it to complete

• **Note that** async at != at async

• **async** and **at** should be thought of as executing via closure
  – We bundle up the values referenced in its code and create an anonymous function (in at statement, the bundle is copied to the other place!)
  – **Can’t reference external var** in async or at, only val
  – For example, iVal is a val copy of i for use in at. i is a var and would generate an error
Place Objects

- Place objects have a field called `id` that contains the place number
- `here` – Place object always bound to current place

class HelloWholeWorld {
    public static def main(args:Array[String](1)):void {
        for (var i:Int=0; i<Place.MAX_PLACES; i++) {
            val iVal = i;
            async at (Place.places(iVal)) {
                Console.OUT.println("Hello World from place " + here.id);
            }
        }
    }
}

Hello World from place 0
Hello World from place 2
Hello World from place 3
Hello World from place 1
public static def main(args:Array[String](1)):Void {
  val arraySize = 12;
  val R : Region = 1..arraySize;
  show("Dist.makeUnique() ", Dist.makeUnique());
  show("Dist.makeBlock(R) ", Dist.makeBlock(R));
  show("Dist.makeBlock(R)|here", Dist.makeBlock(R)|here);
  val testArray = DistArray.make[Int](Dist.makeBlock(R), ([i]:Point)=>i);
  val localSum = DistArray.make[Int](Dist.makeUnique(), ((Point)=>0));
}

• Distributions map regions to places
• Dist factory methods – makeUnique, makeBlock
  – Cyclic, block-cyclic distributions also supported
• Dist (and range) restrictions using | operator
• DistArray similar to Array instantiation
  – Dist object must be provided in addition to base type and initialization function
• DistArray name is visible at all places
Let's compute the global sum of `testArray`

**Step 1:** sum the subarray at each place
- Every `DistArray` object has a member called `dist`
- Every `dist` object has a method called `places` that returns an Array of `Place` objects
- Create an activity at each place using `async`

**Step 2:** `main` activity at place 0
- Retrieves local sum from each place and adds them together
Targeting CUDA with Asynchronous PGAS

• Two different projects with different goals

• X10 CUDA project
  – Expose low level CUDA fundamentals in as direct a fashion as possible
  – Each node has a local view of its accelerator memories

• Chapel CUDA project at UIUC
  – Provide global view of GPU data structures
  – Retarget code for GPU via distributions (or domain maps, as they are known in Chapel)
Additional Places for Accelerators

Goals

• Re-use existing language concepts wherever possible
• Independent GPU memory space ⇒ new place
• Place count and topology unknown until run-time
• Same X10 code works on different run-time configurations
• Could use same approach for OpenCL, Cell, FPGA, …

A 'place' in APGAS model

http://x10-lang.org/documentation/tutorials
Finding CUDA Places

for (host in Place.places()) at (host) {
    val init = new Array[Float](1000, (i:Int)=>i as Float);
    val recv = new Array[Float](1000);
    for (accel in here.children().values()) {
        val remote = CUDAUtilities.makeRemoteArray[Float](accel, 1000, (Int)=>0.0f);
        val num_blocks = 8, num_threads = 64;

        finish Array.asyncCopy(remote, 0, recv, 0, recv.size);
        Console.OUT.println(recv(42));
    }
}

Discover GPUs
Alloc on GPU
Copy result to host
CUDA Kernel and Implicit Transfer

```scala
for (host in Place.places()) at (host) {
    val init = new Array[Float](1000, (i:Int)=>i as Float);
    val recv = new Array[Float](1000);
    for (accel in here.children().values()) {
        val remote = CUDAUtilities.makeRemoteArray[Float](accel, 1000, (Int)=>0.0f);
        val num_blocks = 8, num_threads = 64;

        finish async at (accel) @CUDA {
            finish for ([block] in 0..(num_blocks-1)) async {
                clocked finish for ([thread] in 0..(num_threads-1)) clocked async {
                    val tid = block*num_threads + thread;
                    val tids = num_blocks*num_threads;
                    for (var i:Int=tid ; i<1000 ; i+=tids) {
                        remote(i) = Math.sqrtf(init(i));
                    }
                }
            }
            // Console.OUT.println(remote(42));
            finish Array.asyncCopy(remote, 0, recv, 0, recv.size);
            Console.OUT.println(recv(42));
        }
    }
}
```

- Discover GPUs
- Alloc on GPU
- Implicit capture and transfer to GPU
- Static type error
- Copy result to host

http://x10-lang.org/documentation/tutorials
Adding Blocking and Threading...

```scala
for (host in Place.places()) at (host) {
    val init = new Array[Float](1000, (i: Int) => i as Float);
    val recv = new Array[Float](1000);
    for (accel in here.children().values()) {
        val remote = CUDAUtilities.makeRemoteArray[Float](accel, 1000, (Int) => 0.0f);
        val num_blocks = 8, num_threads = 64;
        finish async at (accel) @CUDA {
            finish for ([block] in 0..(num_blocks-1)) async {
                clocked finish for ([thread] in 0..(num_threads-1)) clocked async {
                    val tid = block*num_threads + thread;
                    val tids = num_blocks*num_threads;
                    for (var i:Int = tid ; i<1000 ; i+=tids) {
                        remote(i) = Math.sqrtf(init(i));
                    }
                }
            }
            // Console.OUT.println(remote(42));
            finish Array.asyncCopy(remote, 0, recv, 0, recv.size),
            finish Array.OUT.println(recv(42));
        }
    }
}
```

Discover GPUs

Alloc on GPU

GPU code

Implicit capture and transfer to GPU

Static type error

Copy result to host

[http://x10-lang.org/documentation/tutorials](http://x10-lang.org/documentation/tutorials)
HPC CHALLENGE (HPCC) STREAM Triad

```plaintext
config const m = 1000;
const alpha = 3.0;
const ProbSpace = [1..m];
var A, B, C: [ProbSpace] real;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

By default, executes on a multicore

Courtesy Albert Sidelnik
http://chapel.cray.com/papers.html
HPC CHALLENGE (HPCC) STREAM Triad

```
config const m = 1000;
const alpha = 3.0;
const ProbSpace = [1..m] dmapped GPUDist(rank=1);

var A, B, C: [ProbSpace] real;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

Distribution to target a GPU

Courtesy Albert Sidelnik
http://chapel.cray.com/papers.html
HPC CHALLENGE (HPCC) STREAM Triad

config const m = 1000;
const alpha = 3.0;
const ProbSpace = [1..m] dmapped GPUDist(rank=1);

var A, B, C: [ProbSpace] real;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;

Distribution to target a GPU

Arrays are declared on the GPU device
HPC CHALLENGE (HPCC) STREAM Triad

```chapel
config const m = 1000;

const alpha = 3.0;

const ProbSpace = [1..m] dmapped GPUDist(rank=1);

var A, B, C: [ProbSpace] real;

forall (a, b, c) in (A, B, C) do
    a = b + alpha * c;
```

Distribution to target a GPU

Arrays are declared on the GPU device

No changes required to the computation for other architectures

Courtesy Albert Sidelnik
http://chapel.cray.com/papers.html
HPC CHALLENGE (HPCC) STREAM Triad

```plaintext
config const m = 1000;
const alpha = 3.0;
const ProbSpace = [1..m] dmapped GPUDist(rank=1);

var A, B, C: [ProbSpace] real;

forall (a,b,c) in (A,B,C) do
    a = b + alpha * c;
```

Distribution to target a GPU

Arrays are declared on the GPU device

No changes required to the computation for other architectures

No need for explicit transfers of data between host and device (e.g. cudaMemcpy(...))
Conclusion

• Heterogenous architectures new and in flux
  – Parallelism and locality are issues that will remain

• APGAS model provides higher-level abstraction
  – Dealing with multiple memory spaces
  – Automating data allocation movement
  – Performing data-parallel operations

• Programming heterogenous architectures will never be as easy (see blocking and threading)