User-Defined Distributions and Layouts in Chapel

Philosophy and Framework

Brad Chamberlain, Steve Deitz, David Iten, Sung Choi

Cray Inc.

HotPAR '10

June 15, 2010

What is Chapel?

- A new parallel language being developed by Cray Inc.
- Part of Cray’s entry in DARPA’s HPCS program
- **Overall Goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Provide better portability than current programming models
  - Improve robustness of parallel codes
- Target architectures:
  - multicore desktop machines (and more recently CPU+GPU mixes)
  - clusters of commodity processors
  - Cray architectures
  - systems from other vendors
- A work in progress, developed as open source (BSD license)

Raising the Level of Abstraction

Chapel strives to provide abstractions for specifying **parallelism** and **locality** in a high-level, architecturally-neutral way compared to current programming models

Chapel's Motivating Themes

1) general parallel programming

- *software:* data, task, nested parallelism, concurrency
- *hardware:* inter-machine, inter-node, inter-core, vector, multithreaded

2) *global-view* abstractions

- post-SPMD control flow and data structures

3) *multiresolution* design

- ability to program abstractly or closer to the machine as needed

4) control of locality/affinity

- to support performance and scalability

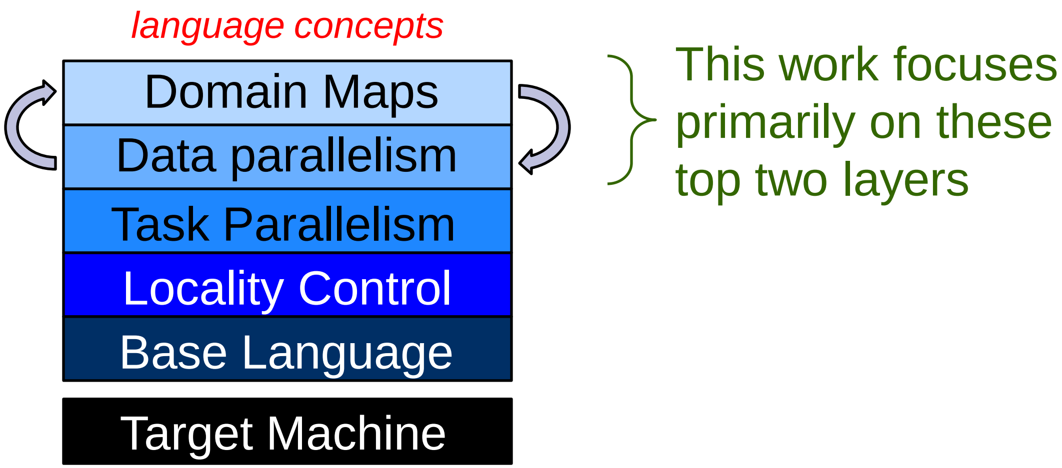
5) reduce gap between mainstream & parallel languages

- to leverage language advances and the emerging workforce

Chapel's Multiresolution Design

Multiresolution Design: Structure the language in layers, permitting it to be used at multiple levels as required/desired

- support high-level features and automation for convenience
- provide the ability to drop down to lower, more manual levels



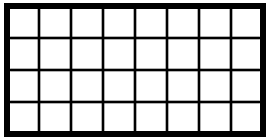
Outline

- ✓ Context
- ➢ Data Parallelism in Chapel
  - domains and arrays
  - *domain maps*
- Domain Map Descriptors
- Sample Use Cases

Data Parallelism: Domains

domain: a first-class index set

```
var m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];
```

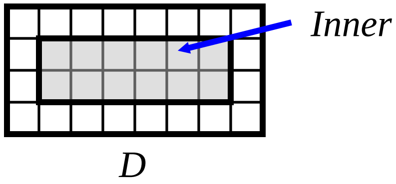


D

Data Parallelism: Domains

domain: a first-class index set

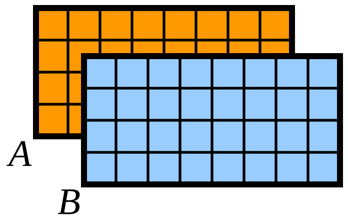
```
var m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];
var Inner: subdomain(D) = [2..m-1, 2..n-1];
```



Domains: Some Uses

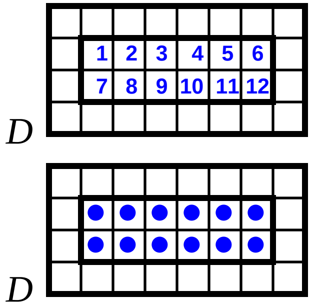
- Declaring arrays:

```
var A, B: [D] real;
```



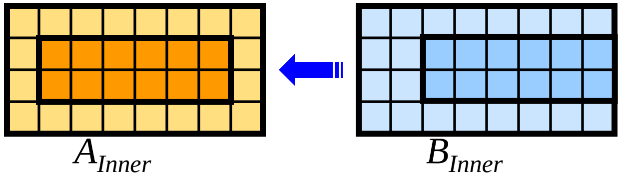
- Iteration (sequential or parallel):

```
    for    ij in Inner { … }
or: forall ij in Inner { … }
or: …
```



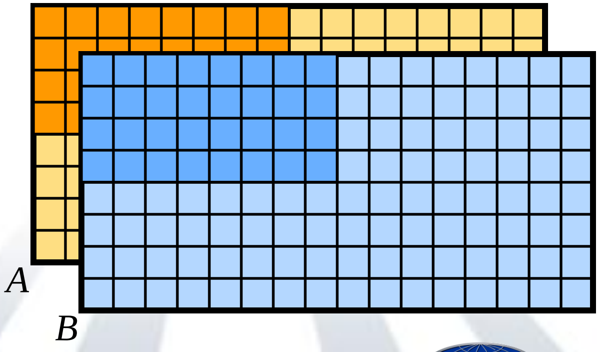
- Array Slicing:

```
A[Inner] = B[Inner+(0,1)];
```



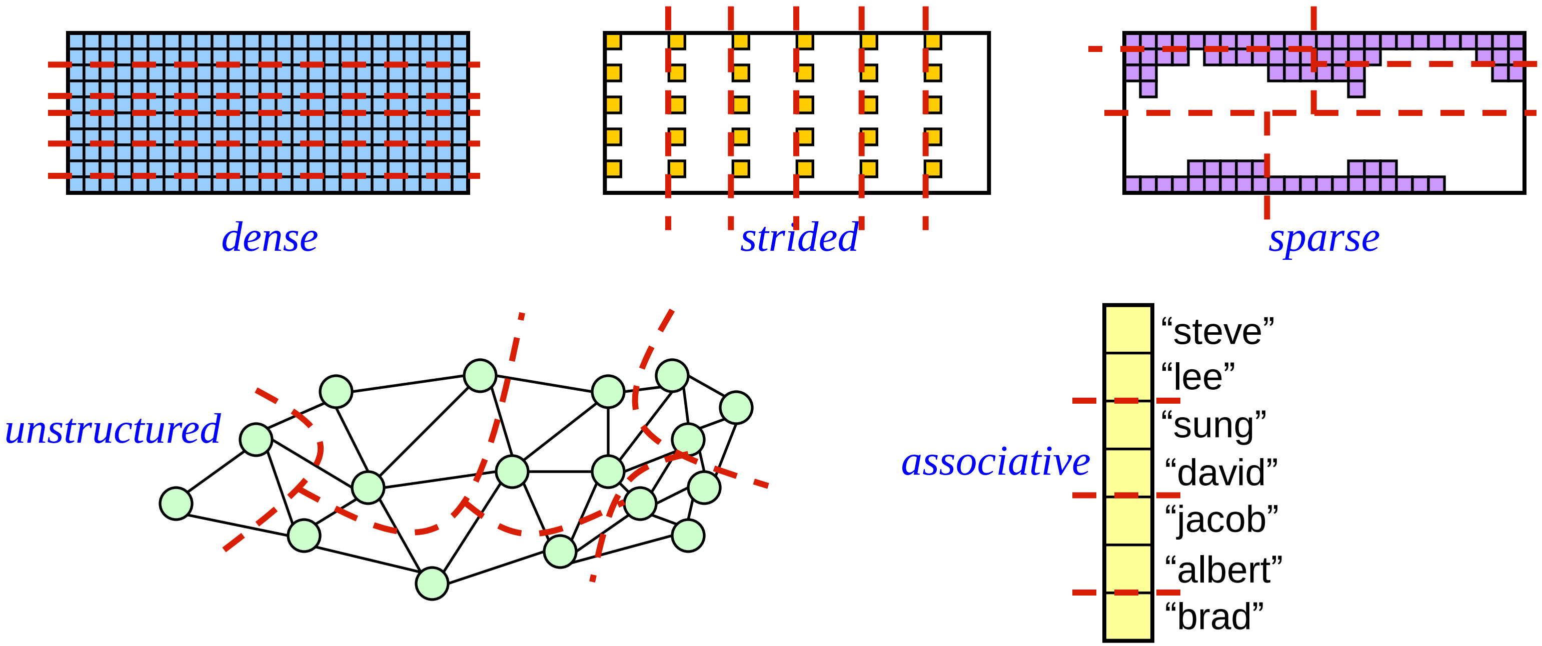
- Array reallocation:

```
D = [1..2*m, 1..2*n];
```



Data Parallelism: Domain/Array Types

Chapel supports several types of domains and arrays…



…all of which support a similar set of data parallel operators:

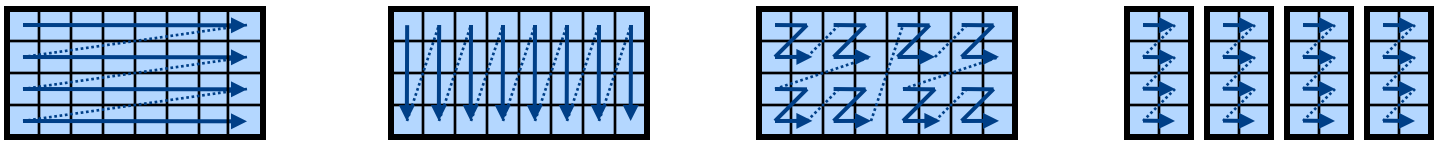
- iteration, slicing, random access, promotion of scalar functions, etc.

…all of which will support distributed memory implementations

Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?

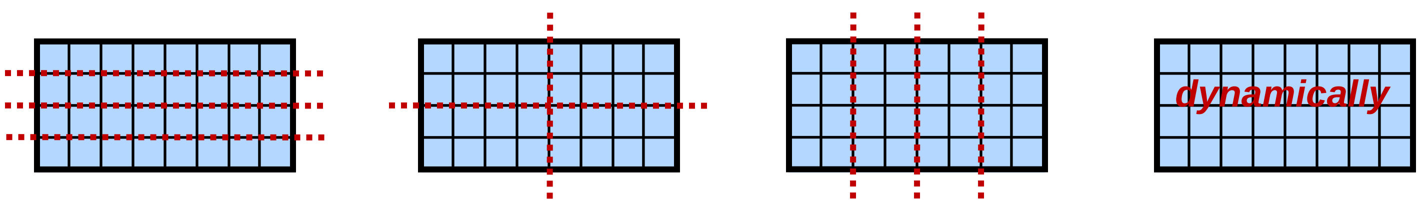
- Are regular arrays laid out in row- or column-major order? Or…?



- What data structure is used to store sparse arrays? (COO, CSR, …?)

Q2: How are data parallel operators implemented?

- How many tasks?
- How is the iteration space divided between the tasks?

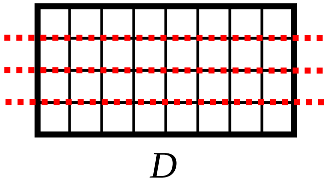


A: Chapel's *domain maps* are designed to give the user full control over such decisions

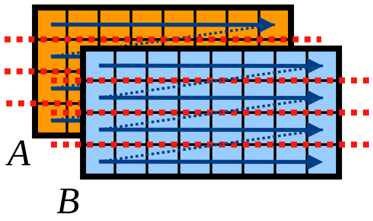
Domain Maps

Any domain can be declared using a domain map

```
var D: domain(2) dmapped RMO(numTasks=here.numCores,
                                parStrategy.rows)
      = [1..m, 1..n];
```




```
var A, B: [D] real;
```



A domain map defines…

…the memory layout of a domain's indices and its arrays' elements

…the implementation of all operations on the domain and arrays

Domain Maps: Layouts and Distributions

Domain Maps fall into two categories:

layouts: target a single shared memory segment

- *e.g.*, a desktop machine or multicore node

distributions: target multiple distinct memory segments

- *e.g.*, a distributed memory cluster or supercomputer

- Most of our work to date has focused on distributions
- Arguably, mainstream parallelism cares more about layouts
  - However, note two crucial trends:
    - as # cores grows, locality will likely be an increasing concern
    - accelerator technologies utilize distinct memory segments

  $\Rightarrow$ mainstream may also care increasingly about distributions

Chapel's Domain Map Strategy

- Chapel provides a library of standard domain maps
  - to support common array implementations effortlessly

- Advanced users can write their own domain maps in Chapel
  - to cope with shortcomings in our standard library

- Chapel's standard layouts and distributions will be written using the same user-defined domain map framework
  - to avoid a performance cliff between "built-in" and user-defined domain maps

- Domain maps should only affect implementation and performance, not semantics
  - to support switching between domain maps effortlessly

Outline

- ✓ Context
- ✓ Data Parallelism in Chapel
- ➤ Domain Map Descriptors
  - Layouts
  - Distributions
- Sample Use Cases

Descriptors for Layouts

Domain Map

Represents: a domain map value

Generic w.r.t.: index type

State: domain map parameters

Size: Θ(1)

Required Interface:

- create new domains

Other Interfaces:

…

Domain

Represents: a domain value

Generic w.r.t.: index type

State: representation of index set

Size: Θ(1) → Θ(*numIndices*)

Required Interface:

- create new arrays
- query size and membership
- serial, parallel, zippered iteration
- domain assignment
- intersections and orderings
- add, remove, clear indices

Other Interfaces:

…

Array

Represents: an array

Generic w.r.t.: index type, element type

State: array elements

Size: Θ(*numIndices*)

Required Interface:

- (re-)allocation of array data
- random access
- serial, parallel, zippered iteration
- slicing, reindexing, rank change
- get/set of sparse "zero" values

Other Interfaces:

…

Descriptor Interfaces

Domain map descriptors support three classes of interfaces:

1. Required Interface

- must be implemented to be a legal layout/distribution

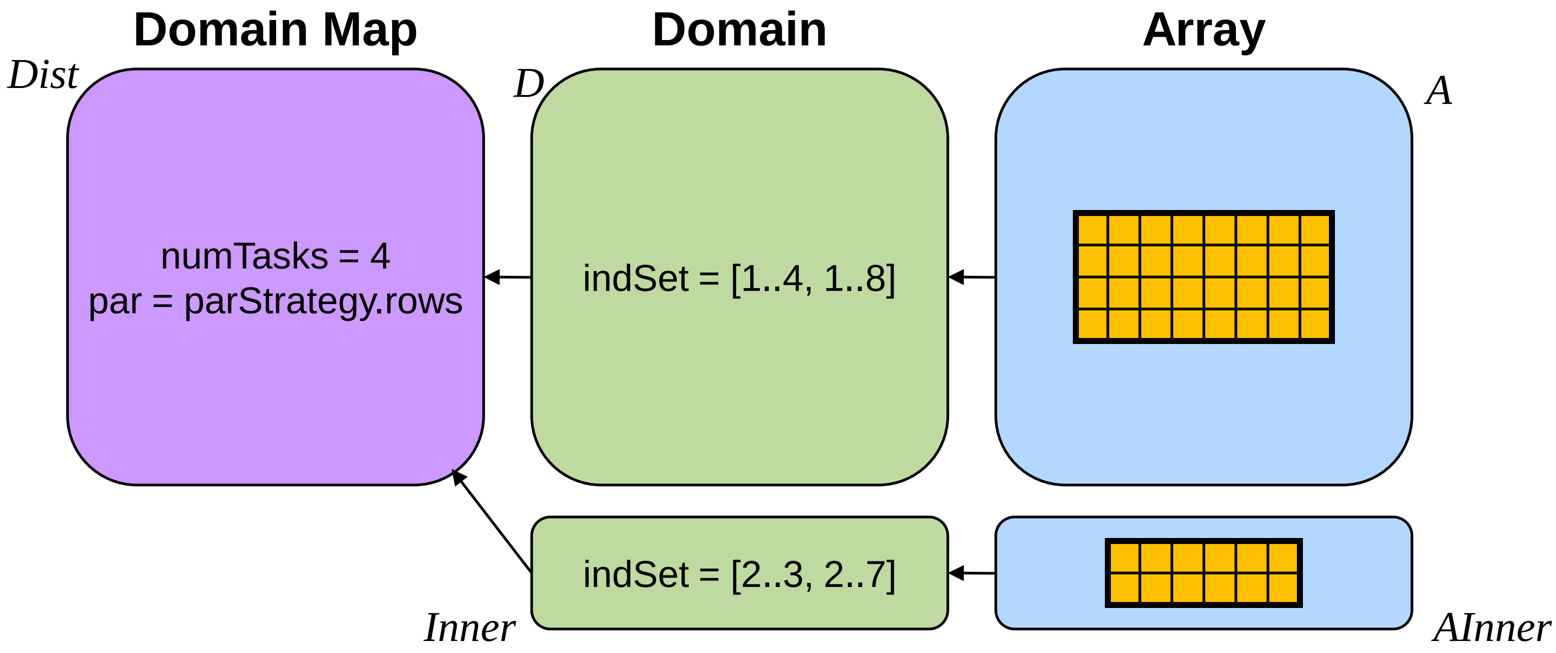
2. Optional Sub-interfaces

- provide optimization opportunities for the compiler when supplied
- *current:*
  - descriptor replication
  - aligned iteration
- *planned:*
  - support for common communication patterns
  - SPMD-ization of data parallel regions

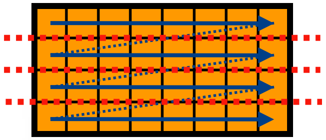
3. User-defined Interfaces

- support additional methods on domain/array values
- intended for the end-user, not the compiler
- by nature, these break the interchangeability of domain maps

Sample Layout Descriptors




```
const Dist = new dmap(new RMO(here.numCores, parStrategy.rows));

const D: domain(2) dmapped Dist = [1..m, 1..n],
      Inner: subdomain(D) = [2..m-1, 2..n-1];

var A: [D] real,
    AInner: [Inner] real;
```



Design Goals

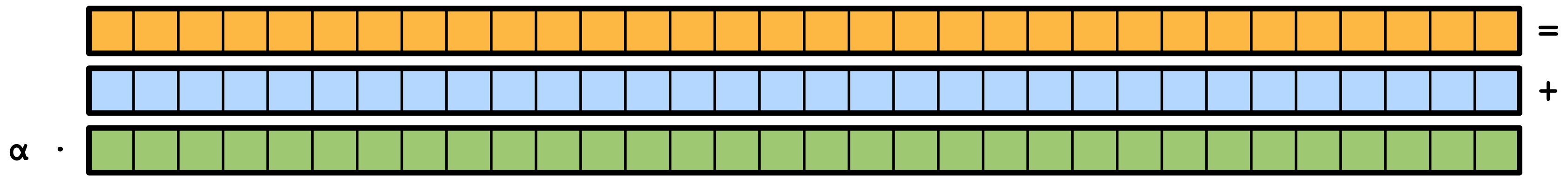
- **For Layouts and Distributions**
  - **Generality:** framework should not impose arbitrary limitations
  - **Functional Interface:** compiler should not care about implementation
  - **Semantically Independent:** domain maps shouldn't affect semantics
  - **Separation of Roles:** parallel experts write; domain experts use
  - **Support Open Libraries:** permit users to share parallel containers
  - **Performance:** should result in good performance, scalability
  - **Known to Compiler:** should support compiler optimizations
  - **Written in Chapel:** using lower-level language concepts:
    - base language, task parallelism, locality features
  - **Transparent Execution Model:** permit user to reason about implementation

- **For Distributions only**
  - **Holistic:** compositions of per-dimension distributions are insufficient
  - **Target Locale Sets:** target arbitrary subsets of compute resources

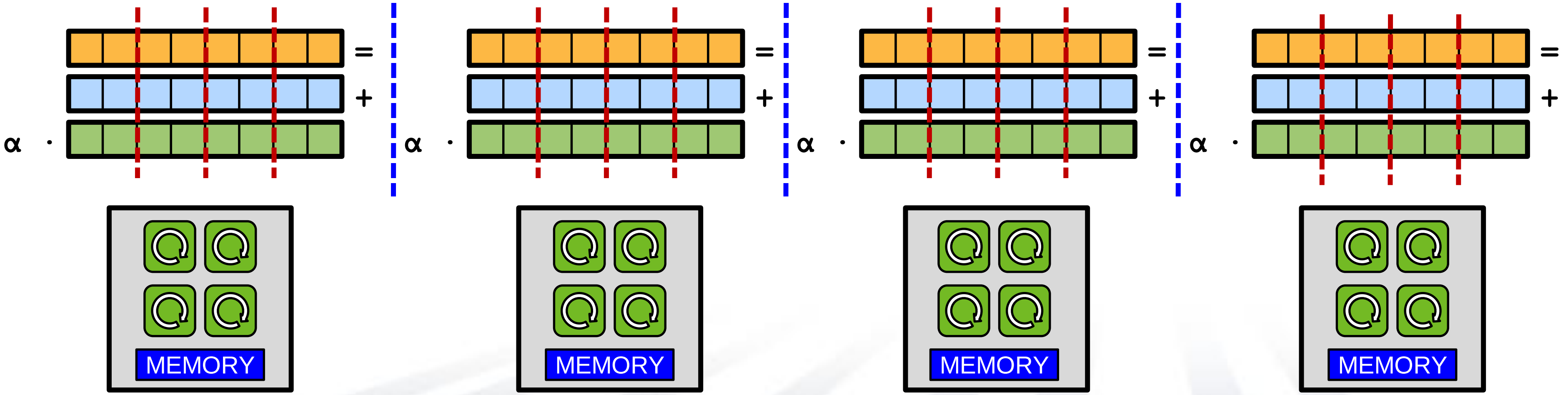
Chapel Distributions

Distributions: "Recipes for parallel, distributed arrays"

- help the compiler map from the computation's global view…

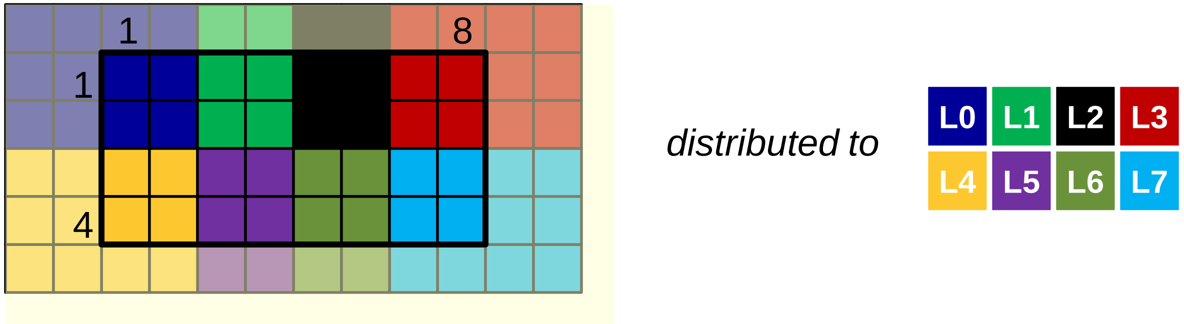


…down to the *fragmented*, per-node/thread implementation

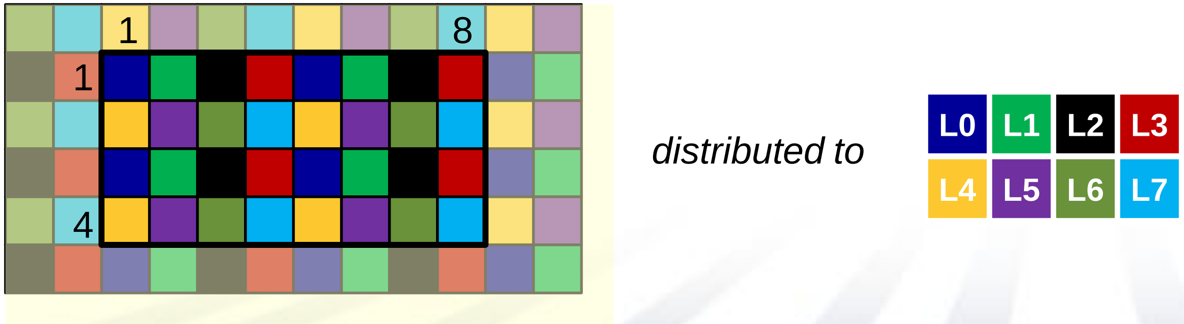


Simple Distributions: Block and Cyclic

```
var Dom: domain(2) dmapped Block(boundingBox=[1..4, 1..8])
         = [1..4, 1..8];
```




```
var Dom: domain(2) dmapped Cyclic(startIdx=(1,1))
         = [1..4, 1..8];
```



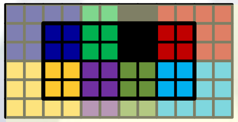
Descriptors for Distributions

	Domain Map	Domain	Array
Global one instance per object (logically)	**Role:** Similar to layout's domain map descriptor	**Role:** Similar to layout's domain descriptor, but no Θ(#indices) storage **Size:** Θ(1)	**Role:** Similar to layout's array descriptor, but data is moved to local descriptors **Size:** Θ(1)
Local one instance per node per object (typically)	**Role:** Stores node-specific domain map parameters	**Role:** Stores node's subset of domain's index set **Size:** Θ(1) → Θ(*#indices / #nodes*)	**Role:** Stores node's subset of array's elements **Size:** Θ(*#indices / #nodes*)

Sample Distribution Descriptors

	Domain Map	Domain	Array
Global one instance per object (logically)	boundingBox = [1..4, 1..8] targetLocales = L0 L1 L2 L3 L4 L5 L6 L7	indexSet = [1..4, 1..8]	--
Local one instance per node per object (typically)	L4 myIndexSpace = [3..max, min..2]	L4 myIndices = [3..4, 1..2]	L4 myElems =

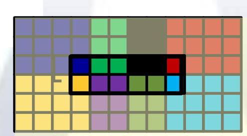
```
var Dom: domain(2) dmapped Block(boundingBox=[1..4, 1..8])
       = [1..4, 1..8];
```



Sample Distribution Descriptors

	Domain Map	Domain	Array
Global one instance per object (logically)	boundingBox = [1..4, 1..8] targetLocales = L0 L1 L2 L3 L4 L5 L6 L7	indexSet = [2..3, 2..7]	--
Local one instance per node per object (typically)	L4 myIndexSpace = [3..max, min..2]	L4 myIndices = [3..3, 2..2]	L4 myElems =

```
var Inner: subdomain(D) = [2..3, 2..7];
```



Implementation Status

- **up and running:**
  - all domains/arrays in Chapel are implemented using this framework
  - **layouts:**
    - parallel layouts for regular domains/arrays
    - serial layouts for irregular domains/arrays (sparse, associative, …)
  - **distributions:** full-featured Block and Cyclic distributions
- **in-progress:**
  - **layouts:** targeting GPU processors (joint work with UIUC)
  - **distributions:** Block-Cyclic, Globally Hashed distributions
- **performance:**
  - reasonable performance & scalability for simple 1D domain/array codes
    - structured communication idioms need more work
  - further tuning required for multidimensional domain/array loops

Next Steps

- **Parallelize layouts for irregular domains/arrays**
- **Complete more distributions**
  - *Regular:* Block-Cyclic, Cut, Recursive Bisection
  - *Irregular:* Block-CSR, Globally Hashed, Graph Partitioned
- **Additional performance improvements**
  - communication aggregation optimizations a la ZPL
  - improved scalar loop idioms
- **Exploration of more advanced domain maps**
  - Dynamically load balanced domain maps
  - Domain maps for resilience
  - Domain maps for *in situ* interoperability
  - Domain maps for out-of-core computation
  - Autotuned domain maps

Related Work

HPF, ZPL, UPC: [Koelbel et al. `96, Snyder `99, El-Ghazawi et al. `05]

- provide global-view arrays for distributed memory systems
- only support a small number of built-in distributions

Vienna Fortran, HPF-2: [Zima et al. `92, HPFF `97]

- support *indirect distributions* that permit the user to specify an arbitrary mapping of array elements to nodes
- O(*n*) space overhead
- no means of controlling details: memory layout, implementation of operations, etc.

A-ZPL: [Deitz `05]

- proposed a taxonomy of distribution types supporting some user specialization
- only a few were ever implemented

Outline

- ✓ Context
- ✓ Data Parallelism in Chapel
- ✓ Domain Map Descriptors
- ➤ Sample Use Cases
  - multicore
  - multi-node
  - CPU+GPU

STREAM Triad (1-locale version)

```
config const m = 1000;
const alpha = 3.0;
```

Default problem size; user can override on executable’s command-line

```
const ProbSpace = [1..m];
```

Domain representing the problem space

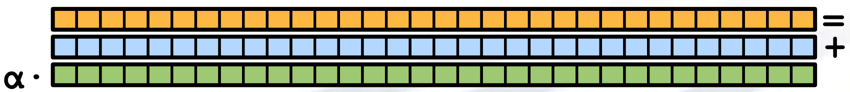
```
var A, B, C: [ProbSpace] real;
```

Three vectors of floating point values

```
B = …;
C = …;
```

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

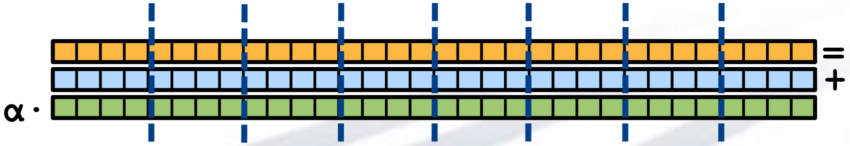
Parallel loop specifying the computation



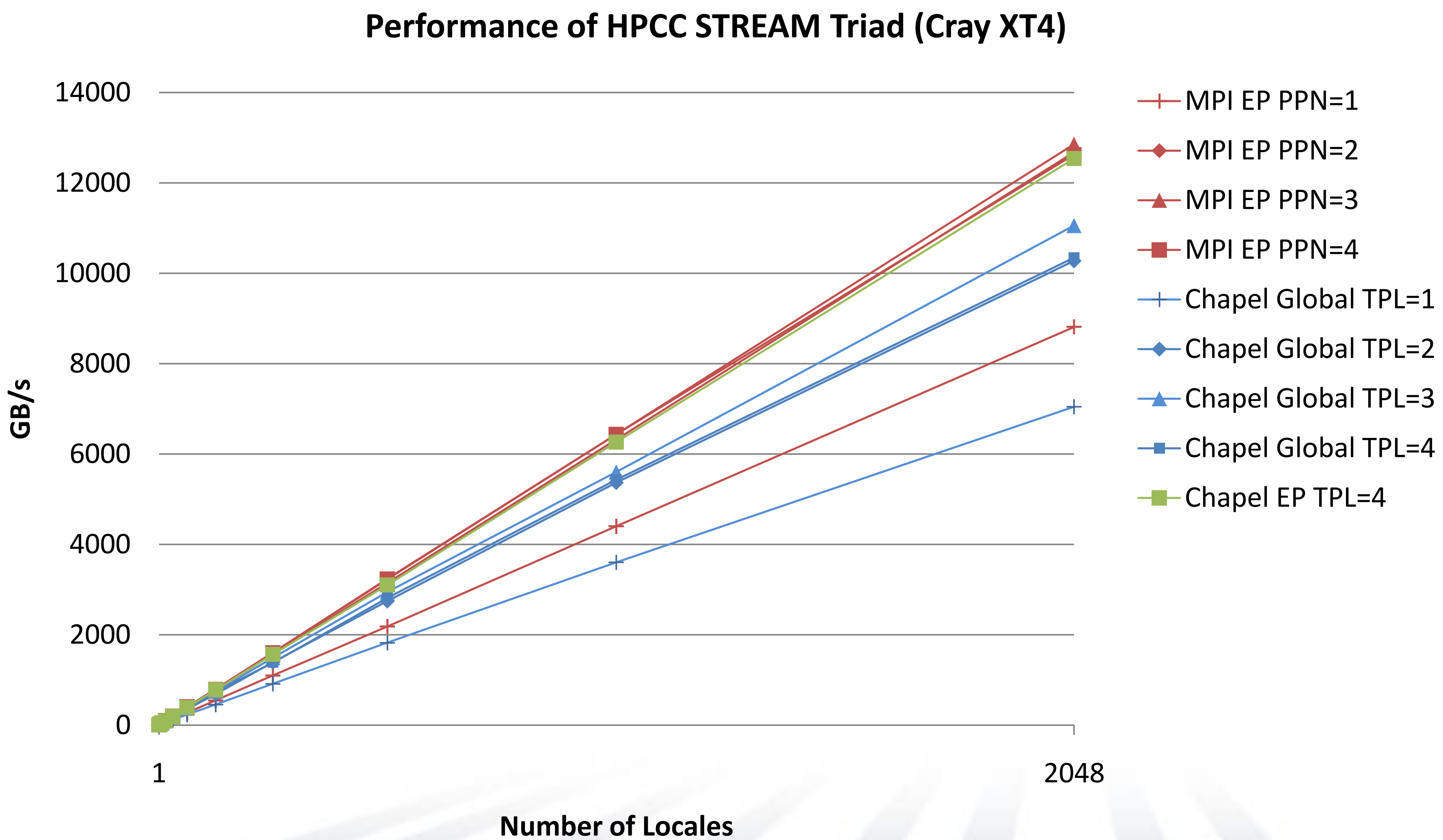
STREAM Triad (multi-locale block version)

```
config const m = 1000;
const alpha = 3.0;

const ProbSpace = [1..m] dmapped Block(boundingBox=[1..m]);


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


B = …;
C = …;



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

add distribution



STREAM Performance: Chapel vs. MPI (2009)

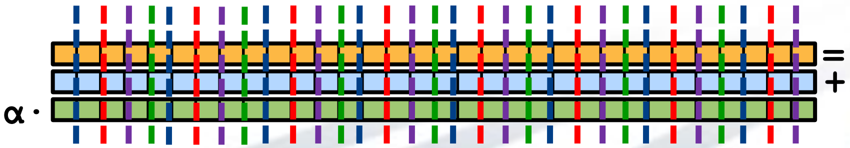


STREAM Triad (multi-locale cyclic version)

```
config const m = 1000;
const alpha = 3.0;

const ProbSpace = [1..m] dmapped Cyclic(startIdx=1);

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

B = …;
C = …;

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

change distribution…

…not computation



STREAM Triad (CPU + GPU version*)

```
config const m = 1000, tpb = 256;
const alpha = 3.0;

const ProbSpace = [1..m];
const GPUProbSpace = ProbSpace dmapped GPULayout(rank=1, tpb);

var hostA, hostB, hostC: [ProbSpace] real;
var gpuA, gpuB, gpuC: [GPUProbSpace] real;

hostB = …;
hostC = …;

gpuB = hostB;
gpuC = hostC;

forall (a,b,c) in (gpuA, gpuB, gpuC) do
  a = b + alpha * c;

hostA = gpuA;
```

Create domains for both host (CPU) and GPU

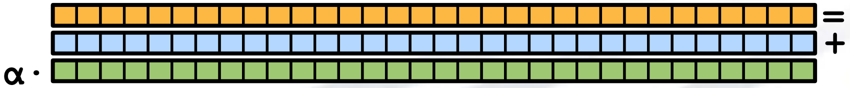
Create vectors on both host (CPU) and GPU

Perform vector initializations on the host

Assignments between host and GPU arrays result in CUDA memcpy

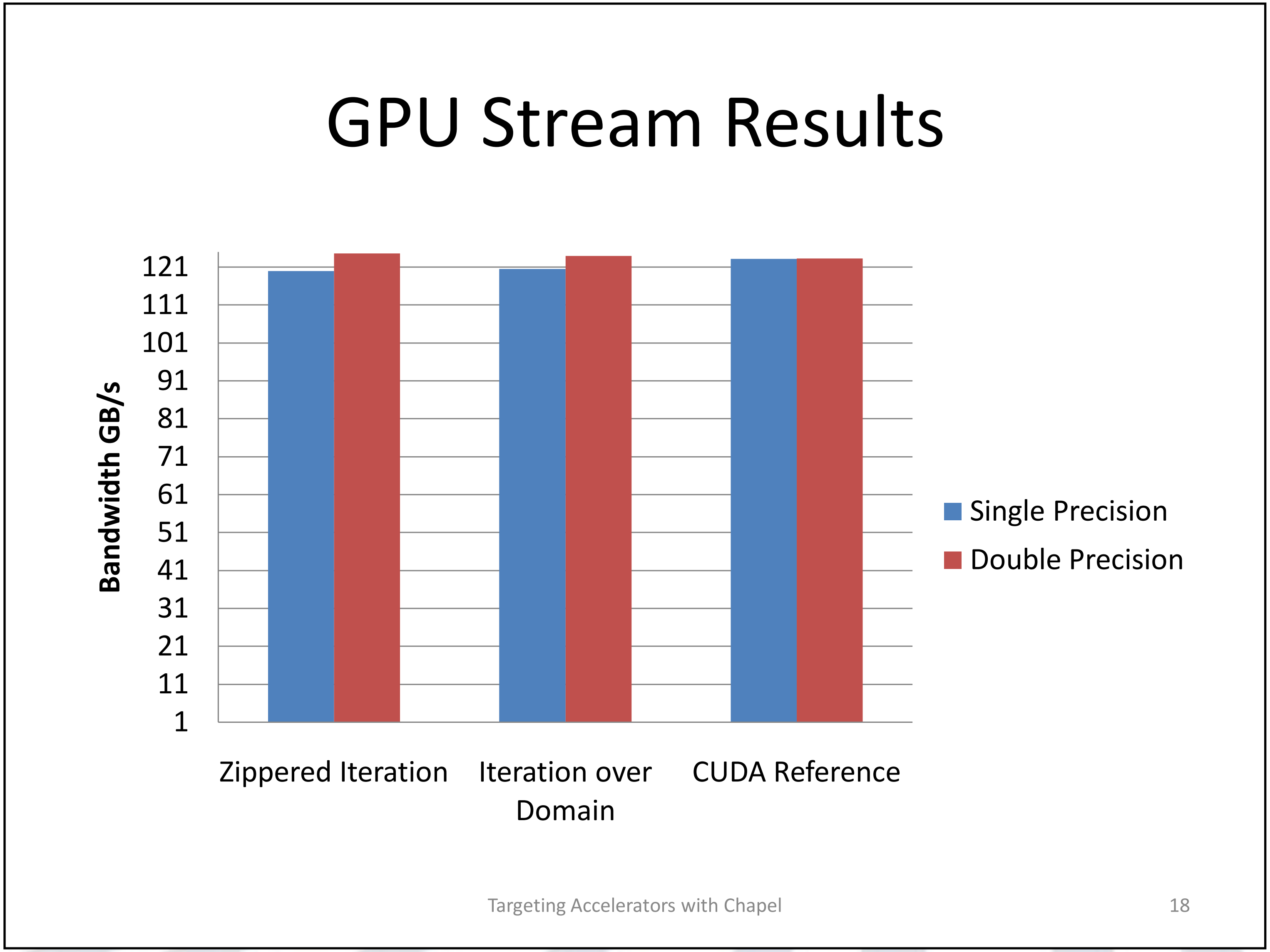
Computation executed by GPU

Copy result back from GPU to host memory



* joint work with Albert Sidelnik, María Garzarán, David Padua, UIUC

Experimental results (NVIDIA GTX 280)



(slide courtesy of Albert Sidelnik)

Since then…

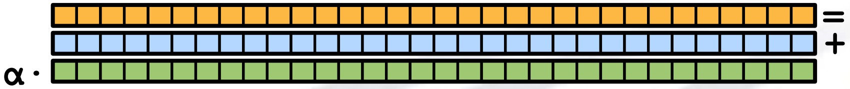
- Albert has studied more interesting GPU patterns in Chapel
  - primarily from the Parboil benchmark suite:
    http://impact.crhc.illinois.edu/parboil.php
  - can achieve competitive performance
  - yet GPU details show up in code more than we'd ideally like

- Next steps for GPU domain maps:
  - repurpose Chapel's locale concept to better suit GPUs/hierarchy
  - reduce user's role in data exchanges
  - and plenty more…

STREAM Triad (notional CPU+GPU version)

```
config const m = 1000, tpb = 256;
const alpha = 3.0;

const ProbSpace = [1..m] dmapped CPUGPULayout(rank=1, tpb);

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

B = …;
C = …;

ProbSpace.changeMode(mode.GPU);

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

ProbSpace.changeMode(mode.CPU);
```

Use single domain map with ability to switch between CPU and GPU modes



Case Study: STREAM (current practice)

CUDA

```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

Case Study: STREAM (current practice)

CUDA

```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d
  cudaMalloc((void**)&d
  cudaMalloc((void**)&d

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlo
  if( N % dimBlock.x !=

  set_array<<<dimGrid,d
  set_array<<<dimGrid,d

  scalar=3.0f;
  STREAM_Triad<<<dimGri
  cudaThreadSynchronize

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_arr
  int idx = threadIdx.x
  if (idx < len) a[idx]
}

__global__ void STREAM_Triad( float *a, float *b, float *c,

  int i
  if (i
}
```

MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
;
, MPI_SUM, 0, comm );

 doIO) {

ams, 3, sizeof(double), 0 );
;
;
;

ate memory (%d).\n", VectorSize );

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for

  return 0;
}
```

Chapel (today)

```
config const m = 1000, tpb = 256;
const alpha = 3.0;

const ProbSpace = [1..m];
const GPUProbSpace = ProbSpace dmapped GPULayout(rank=1, tpb);

var hostA, hostB, hostC: [ProbSpace] real;
var gpuA, gpuB, gpuC: [GPUProbSpace] real;

hostB = …;
hostC = …;

gpuB = hostB;
gpuC = hostC;

forall (a,b,c) in (gpuA, gpuB, gpuC) do
  a = b + alpha * c;

hostA = gpuA;
```

For GPUs, as with supercomputers, it seems crucial to support the specification of **parallelism** and **locality** in an implementation-neutral way

Summary

Domain Maps support high-level data parallel operators on user-defined implementations of parallel arrays

Future work will add optimizations to strengthen our performance argument while also demonstrating advanced applications of domain maps

In the spirit of green conferences…

Would anyone want to share a cab to SFO for a ~6pm flight?

For More Information

chapel_info@cray.com

http://chapel.cray.com

(slides, papers, collaboration possibilities, etc.)

http://sourceforge.net/projects/chapel

(code, mailing lists)

Parallel Programmability and the Chapel Language; Chamberlain, Callahan, Zima; International Journal of High Performance Computing Applications, August 2007, 21(3):291-312.

Questions?