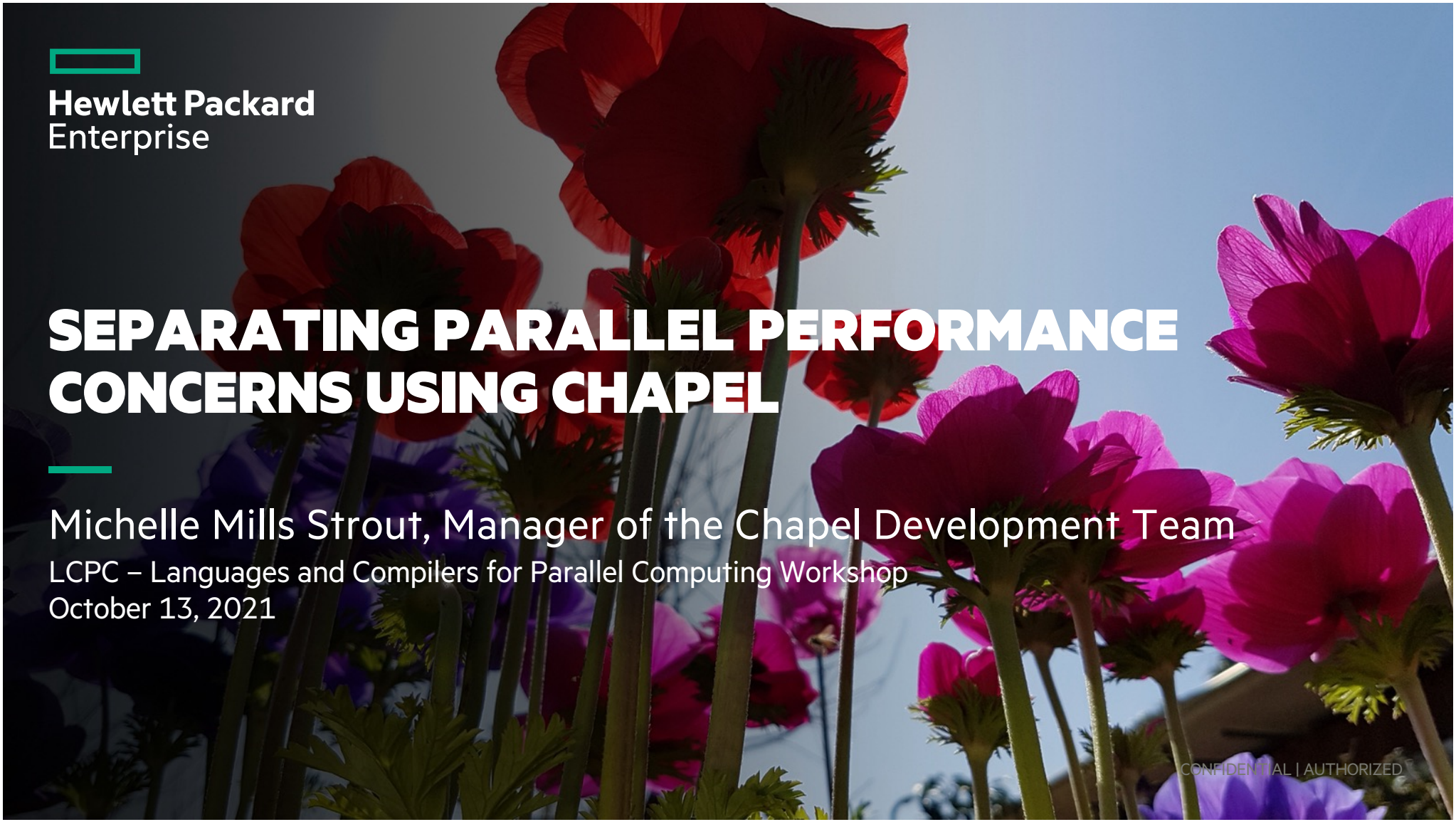
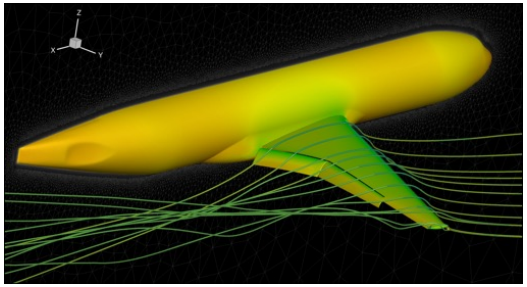
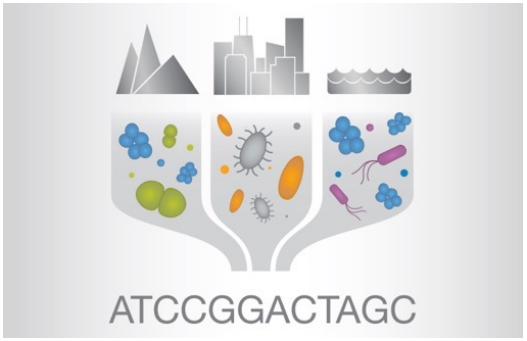

Hewlett Packard
Enterprise
SEPARATING PARALLEL PERFORMANCE CONCERNS USING CHAPEL
Michelle Mills Strout, Manager of the Chapel Development Team
LCPC – Languages and Compilers for Parallel Computing Workshop
October 13, 2021
CONFIDENTIAL | AUTHORIZED

CHAPEL DEVELOPMENT TEAM

HIGH PERFORMANCE COMPUTING





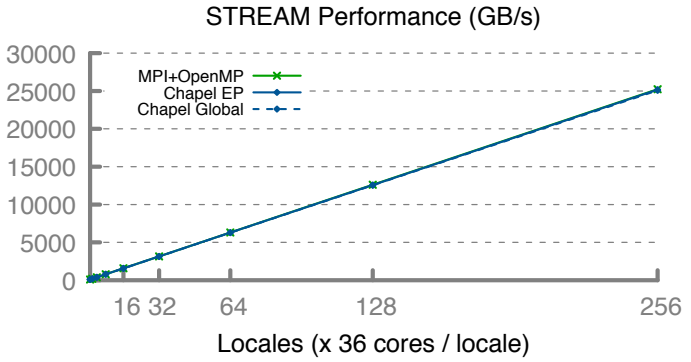
- Simulations
- Analysis of massive datasets
- Ever-changing machines
- Challenging for programmer productivity

CHAPEL BENCHMARKS TEND TO BE CONCISE, CLEAR, AND PERFORMANT

STREAM TRIAD: C + MPI + OPENMP

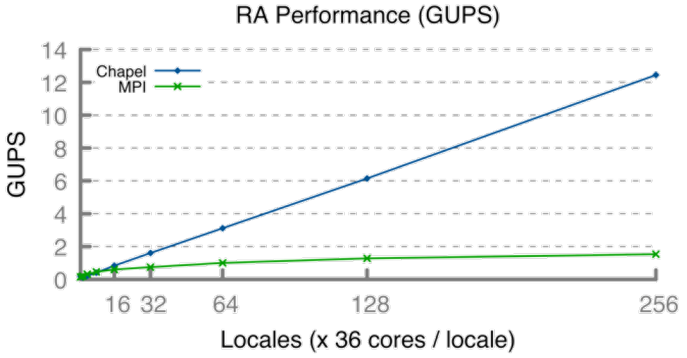
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  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;
}
```

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;
const Dom = {1..m} dmapped …;
var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```



HPCC RA: MPI KERNEL

```
…
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
…
```



TAKEAWAY: CAN SEPARATE PERFORMANCE CONCERNS WITH CHAPEL

STREAM Triad: $\vec{A} = \vec{B} + \alpha\,\vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

```
use BlockDist
config const m = 1000, alpha = 3.0;
const Domain = {1..m}
               dmapped Block({1..m});

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

B = 2.0;
C = 1.0;

forall i in Domain do
    A[i] = B[i] + alpha * C[i];
```

***Goal**: Let programmers **control** performance concerns in a **separate** stack of abstractions.*

***Talk**: Example abstractions in Chapel, including for GPUs*

HPC PROGRAMMERS WANT CONTROL

STREAM Triad: $\vec{A} = \vec{B} + \alpha \vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

HPC programmers obtain control by coding at a low-level of detail

Let's provide separate **What** and **How** constructs and abstractions

***Goal**: Let programmers **control** performance concerns in a **separate** stack of abstractions.*

SEPARATING INTO WHAT AND HOW CONCERNS

STREAM Triad: $\vec{A} = \vec{B} + \alpha \vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

- **What**
  - STREAM Triad computation

SEPARATING INTO WHAT AND HOW CONCERNS

STREAM Triad: $\vec{A} = \vec{B} + \alpha \vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

- **What**
  - STREAM Triad computation
- **How**
  - Data organization across nodes in a parallel machine

SEPARATING INTO WHAT AND HOW CONCERNS

STREAM Triad: $\vec{A} = \vec{B} + \alpha \vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

- **What**
  - STREAM Triad computation
- **How**
  - Data organization across nodes in a parallel machine
  - Schedule: process per node (MPI, message passing interface)

SEPARATING INTO WHAT AND HOW CONCERNS

STREAM Triad: $\vec{A} = \vec{B} + \alpha \vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

- **What**
  - STREAM Triad computation
- **How**
  - Data organization across nodes in a parallel machine
  - Schedule: process per node (MPI, message passing interface)
  - Schedule: threads per process (OpenMP)

JUST HIDE DETAILS IN WHAT CODE FUNCTIONS

STREAM Triad: $\vec{A} = \vec{B} + \alpha \vec{C}$

```
#include <hpcc.h>
#include <omp.h>
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 );
  // deleted error malloc error checking code
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;
  #pragma omp parallel for
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
  // deleted deallocation code
  return 0;
}
```

```
Vector*
STREAM_Triad(Vector *B, double alpha,
  Vector* C) {
        . . .
        . . .
        . . .
        . . .
}

Vector* A = STREAM_Triad(B, 2.0, C);
```

Doesn't work! Composition? New Machine?

JUST RAISE THE LEVEL OF ABSTRACTION

If we just raise the abstraction level in the **What** stack high-enough, the compiler or library can effectively map computations to a variety of hardware.

Problem: Programmer loses control

Need: Programmer control at multiple levels of a **How** stack, **multiresolution**

CHAPEL: MULTIRESOLUTION PROGRAMMING EXAMPLE

```
use BlockDist

config const m = 1000,
             alpha = 3.0;

const Domain = {1..m}
             dmapped Block({1..m});

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

B = 2.0;
C = 1.0;

forall i in Domain do
    A[i] = B[i] + alpha * C[i];
```

- Domain mapping construct “dmapped …” semantics like HPF
- However, “Block” is implemented with user-facing constructs
- **Users can write their own domain mappings**
- Provides multiresolution programming

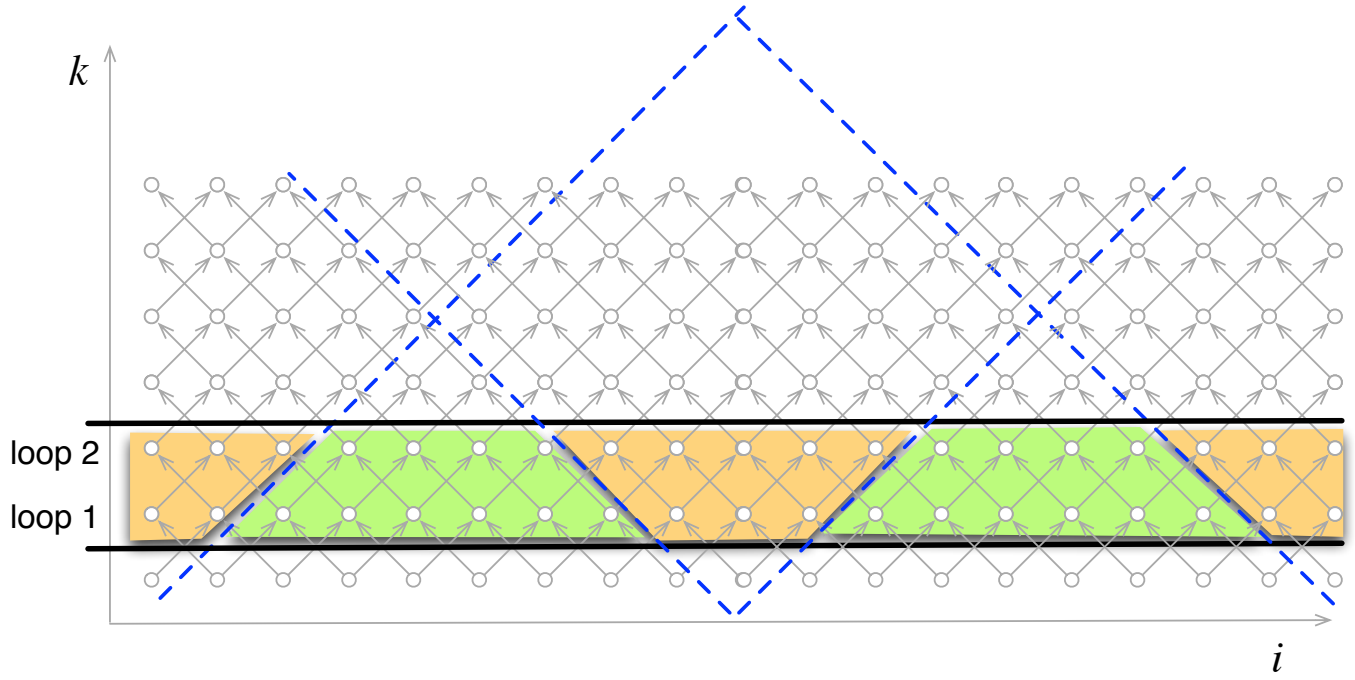
CHAPEL ITERATORS PROVIDE MULTIRESOLUTION SCHEDULE CONTROL

```
use BlockDist

config const m = 1000,
             alpha = 3.0;

const Domain = {1..m}
               dmapped Block({1..m});

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

B = 2.0;
C = 1.0;

forall i in Domain do
  A[i] = B[i] + alpha * C[i];
```

- Programmer can see details of default iterator
- Programmer can write own domains mappings and iterators

```
// Default iterator for a Block domain
iter these ( ... ) {

 coforall loc in Domain.targetLocales {
   on loc {
     const numTasks = computeNumTasks();
     const myInds = Domain.localSubdomain(loc);
     coforall tid in 0..<numTasks {
     const myChunk = chunk(myInds, tid, numTasks);
     for i in myChunk do
       yield i;
     }
   }
}
```

MORE COMPLEX SCHEDULE: DIAMOND SLAB TILING



MAKING SCHEDULES AVAILABLE IN LIBRARIES

Diamond slab tiling written in C

```
int Li=0, Ui=N, Lj=0, Uj=N;

    for(int ts=0; ts<T ; ts+=subset_s)
     for (int c0 = -2; c0<=0; c0+=1)
       for (int c1 = 0; c1 <= (Uj+tau-3)/(tau-3) ; c1+=1)
        for (int x = (-Ui-tau+2)/(tau-3); x<=0 ; x += 1){
          int c2 = x-c1; //skew
          // loops for time steps within a slab
          // (slices within slabs)
          for (int c3 = 1; c3<=subset_s; c3 += 1)

            for (int c4 = max(max(max(-tau * c1 - tau *
c2 + 2 * c3 - (2*tau-2), -Uj - tau * c2 + c3 - (tau-2)),
tau * c0 - tau * c1 - tau * c2 - c3), Li); c4 <=
min(min(min(tau * c0 - tau * c1 - tau * c2 - c3 + (tau-
1), -tau * c1 - tau * c2 + 2 * c3), -Lj - tau * c2 + c3),
Ui - 1); c4 += 1)

              for (int c5 = max(max(tau * c1 - c3, Lj), -
tau * c2 + c3 - c4 - (tau-1)); c5 <= min(min(Uj - 1, -tau
* c2 + c3 - c4), tau * c1 - c3 + (tau-1)); c5 += 1)

                computation(c3, c4, c5);
        }
```

Diamond slab tiling made available as a Chapel iterator

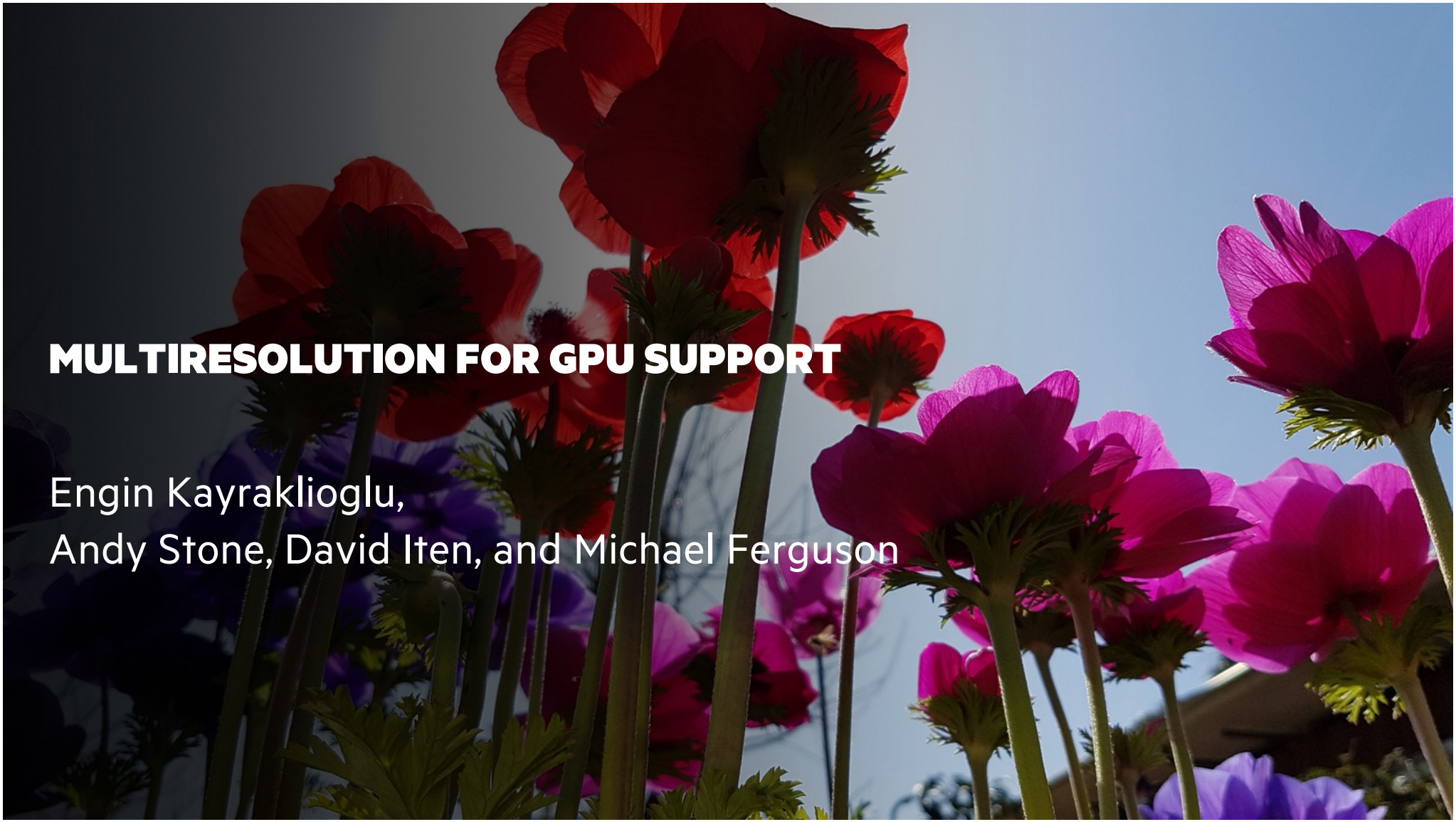
We want to transform our original schedule:

```
for t in timeRange do
    forall (x,y) in spaceDomain do
        computation( t, x, y );
```

into a faster schedule:

```
forall (t,x,y)
in diamondTileIterator(…) do
    computation( t, x, y );
```

Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. "Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators." In the Proceedings of the 29th International Conference on Supercomputing (ICS), June 2015.



MULTIRESOLUTION FOR GPU SUPPORT

Engin Kayraklioglu,
Andy Stone, David Iten, and Michael Ferguson

PLANS FOR GPU SUPPORT

Vision

Memory/Locality Management

- Chapel's locale model concept supports describing a compute node with GPU naturally
  - The execution and memory allocations can be moved to GPU sublocales
- Arrays can be declared inside 'on' statements to allocate them on GPU memory
- Or distributed arrays that target GPU sublocales can be created

Done in 1.25

Next step

Execution

- Chapel's order-independent loops (i.e., 'forall' and 'foreach') can be transformed into GPU kernels
  - If such a loop is encountered while executing on a GPU sublocale, the corresponding kernel is launched
  - GPU code is generated for every call inside the loop body

Done in 1.25

Next step

Other Possible Multiresolution Features

- Specifying the grid and block organization for the GPU version of the computation
- Queries about node architecture including number and kind of GPUs to guide iteration and data org

CHAPEL 1.25 EFFORT

Putting the Pieces Together

User's loop

```
forall i in 1..n do arr[i] = i*mul;
```

The loop is replaced with:

```
if executingOnGPUSublocale()
  launch_kernel("kernel", n-1, 512, 1, 0,
                n, 0, &arr, 32, mul, 0)
else
  for (i=1 ; i<=n ; i++) {
    var arrData = arr->data
    ref addrToChange = &arrData[i]
    var newVal = i*mul
    *addrToChange = newVal
  }
```

Generated GPU kernel looks like:

```
pragma "codegen for GPU"
proc kernel(in startIdx, in endIdx,
            ref arrArg, in mulArg) {
  var blockIdxX  = __primitive('gpu blockIdx x')
  var blockDimX  = __primitive('gpu blockDim x')
  var threadIdxX = __primitive('gpu threadIdx x')

  var t0 = blockIdxX * blockDimX
  var t1 = t0 + threadIdxX
  var index = t1 + startIdx

  var chpl_is_oob = index > endIdx
  if (chpl_is_oob) { return; }

  var arrData = arrArg->data
  ref addrToChange = &arrData[index]
  var newVal = myIdx*mulArg
  *addrToChange = newVal
}
```

STATUS
Stream

```
on here.getChild(1) {
  var a, b, c: [1..n] real;
  const alpha = 2.0;

  b = 1.0;
  c = 2.0;

  forall aElem, bElem, cElem in zip(a, b, c) do
    aElem = bElem + alpha * cElem;
  // or
  forall i in a.domain do
    a[i] = b[i] + alpha * c[i];

}
```

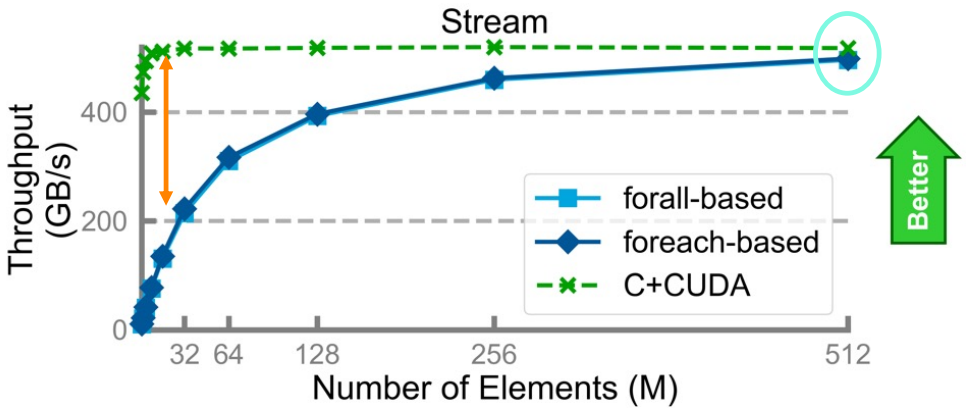
- Arrays are allocated in unified memory
- Scalars are allocated on the function stack
  - So, they are on host memory

Promotion (e.g., 'b = 1.0') still executes on host

These foralls will execute on GPU

STATUS OF GPU PERFORMANCE FOR CHAPEL

An Early Performance Study



At smaller vector sizes throughput is low

At larger vector sizes efficiency reaches 96%

Takeaways

- No major performance-related issue in the prototype
- Gets close to 100% efficiency with large datasets
- 'foreach' is slightly faster than 'forall'

Potential Sources of Overhead

- I/O for loading the GPU kernel for each launch
- Unified memory vs device memory
- Kernel argument allocations

Prospects

- Generating single binary will remove the I/O cost
- Profile the remaining costs
- Implement other benchmarks

SUMMARY AND THANK YOU!

michelle.strout@hpe.com

- **Goal**: *Let programmers **control** performance concerns in a **separate** stack of abstractions*
- Chapel's multiresolution control enables the separation of parallel performance concerns
  - Domain mappings
  - Iterators
- Multiresolution control in for GPU support in Chapel 1.25
- The Chapel team is hiring! (https://chapel-lang.org/jobs.html)

```
use BlockDist
config const m = 1000, alpha = 3.0;
const Domain = {1..m}
                dmapped Block({1..m});

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

B = 2.0;
C = 1.0;

forall i in Domain do
    A[i] = B[i] + alpha * C[i];
```


THANK YOU
michelle.strout@hpe.com