Targeting GPUs and Other Hierarchical Architectures in Chapel

Albert Sidelnik
University of Illinois at Urbana-Champaign

Acknowledgment:
Pan-American Advanced Studies Institute (PASI) program of NSF

Goals of this work

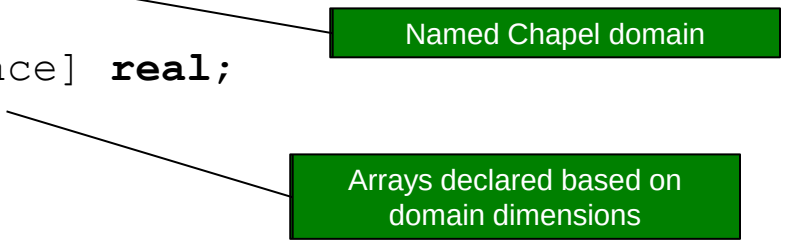
- Make programming GPUs easier
- Easy method to program clusters of GPUS

Motivating example : SAXPY

A = alpha * B + C

HPC CHALLENGE (HPCC) STREAM Triad

```
config const m = 1000;

const alpha = 3.0;

const ProbSpace = [1..m];

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



HPC CHALLENGE (HPCC) STREAM Triad

```
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

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

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

```
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

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

No changes required to the computation for other architectures

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

STREAM Triad (current practice)

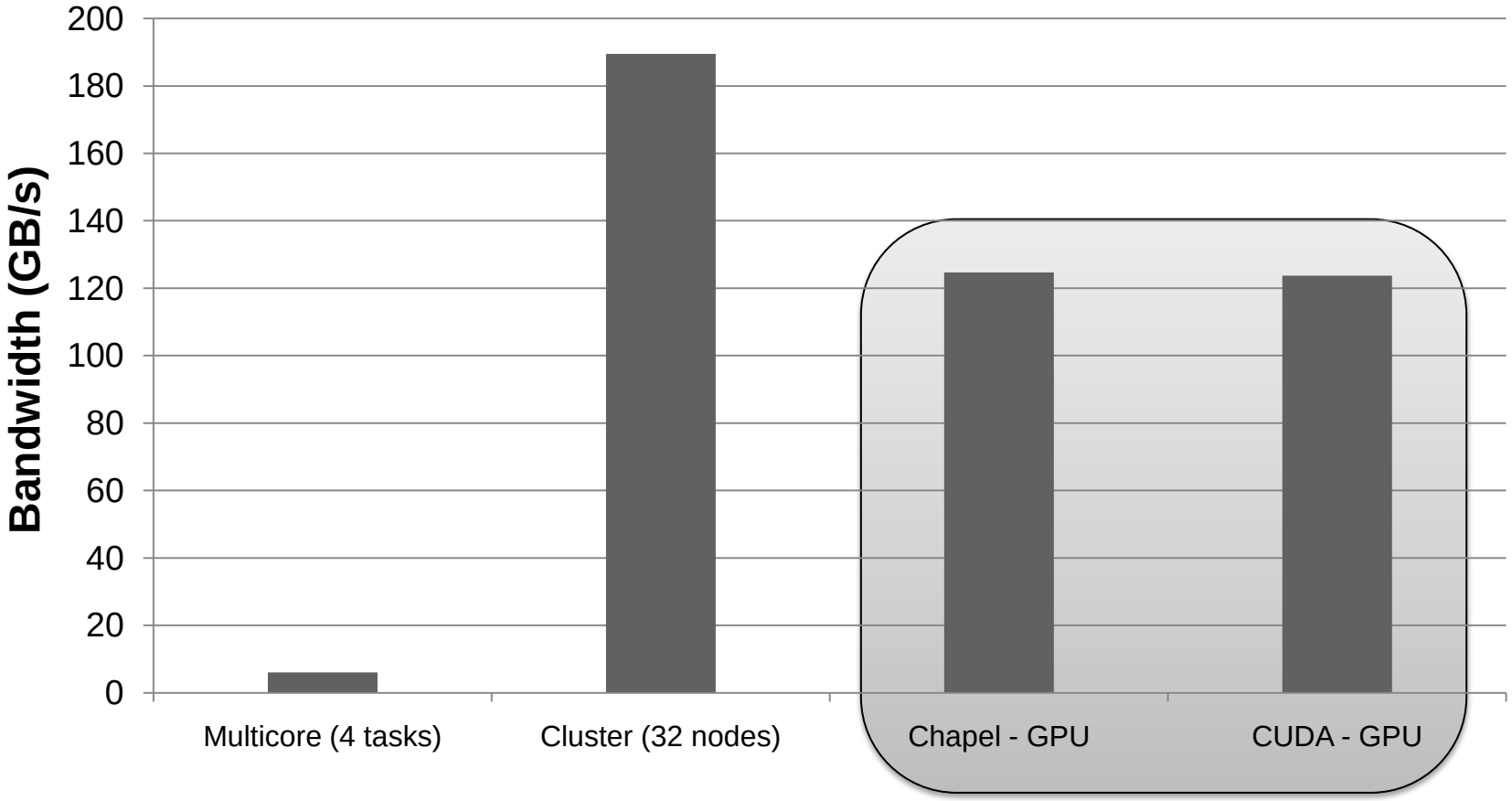
CUDA

```
#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;
}
```

Performance of STREAM Triad
Multicore vs. Cluster vs. GPU

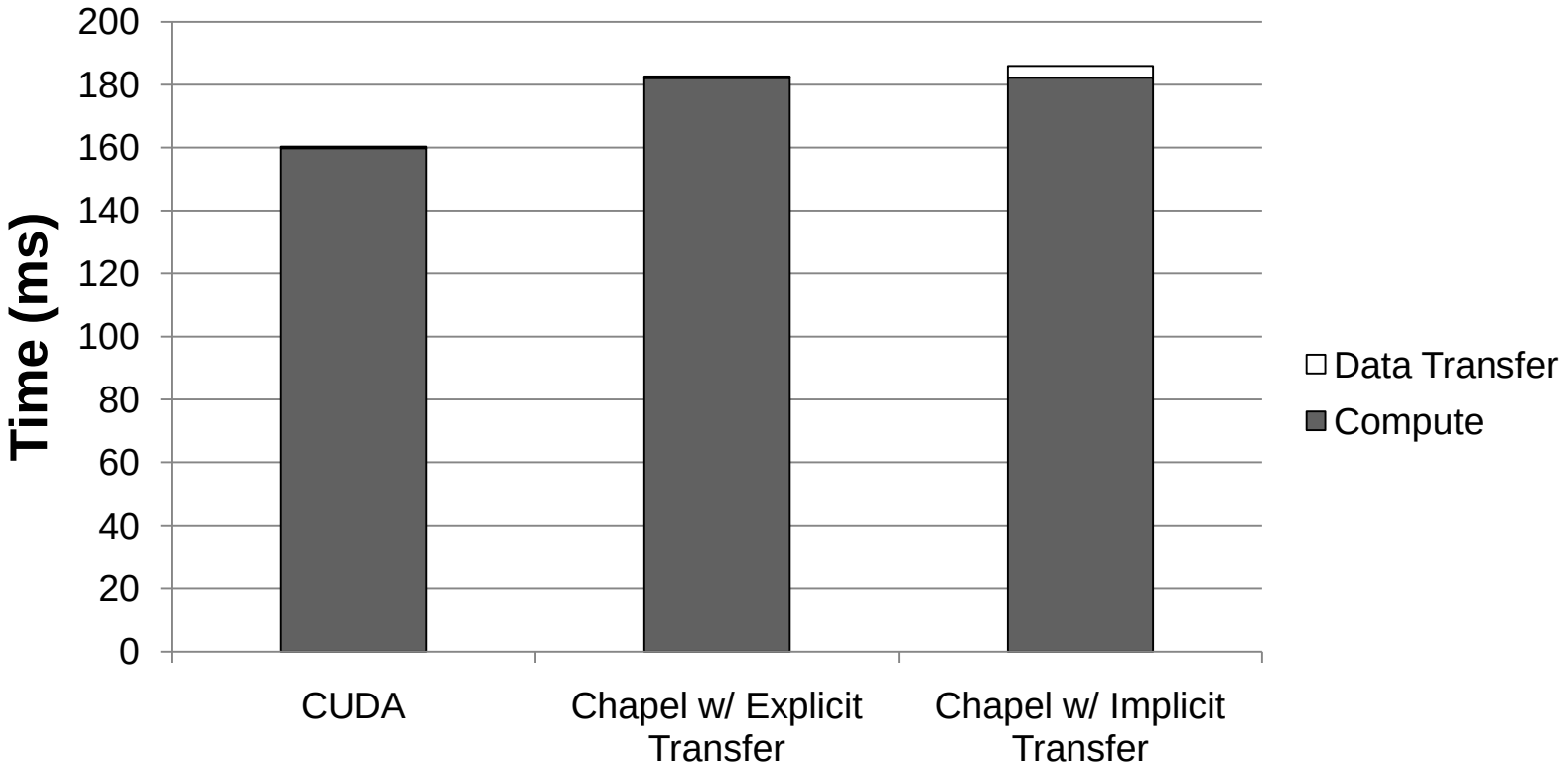


For STREAM, the Chapel and CUDA implementations match performance

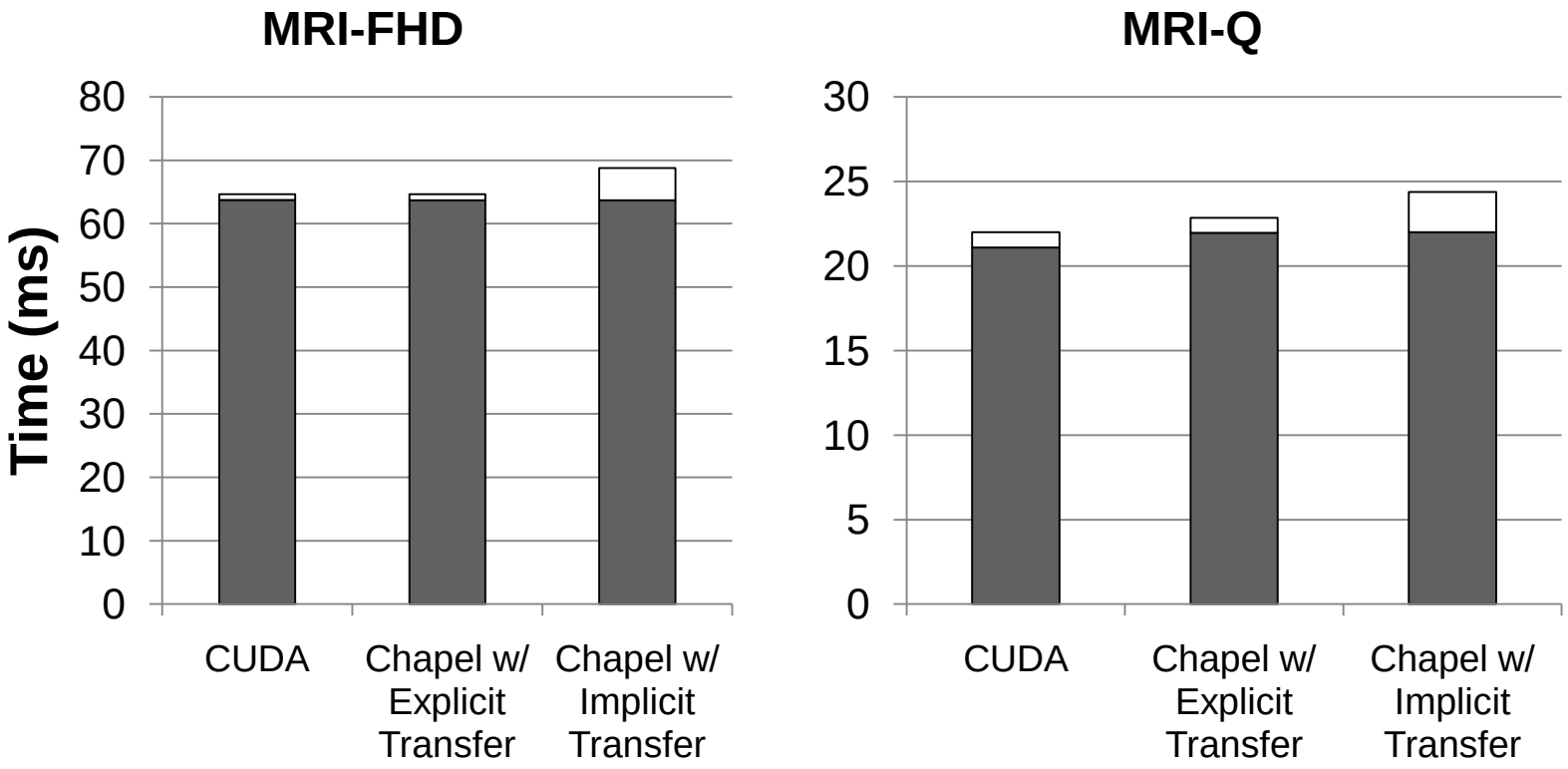
Experimental Results

- Parboil Benchmark Suite
  - Hand-tuned CUDA Benchmarks
- Ported to Chapel
  1. Implicit Data Transfer
  2. Explicit Data Transfer
- Compare to CUDA
  1. Performance
  2. Productivity “gains”

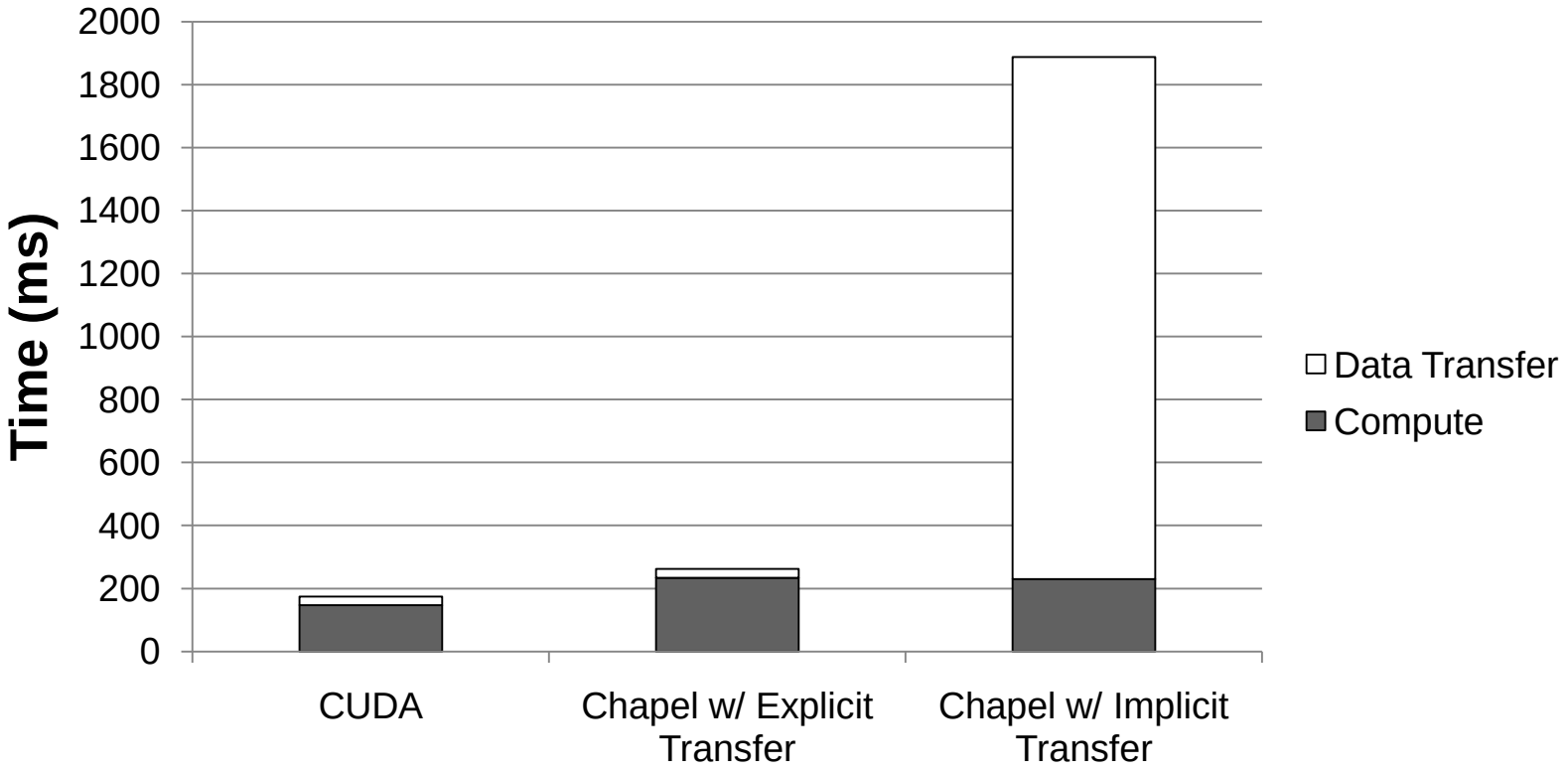
Coulombic Potential (CP)



MRI


MRI-FHD
Time (ms)
80
70
60
50
40
30
20
10
0
CUDA
Chapel w/ Explicit Transfer
Chapel w/ Implicit Transfer
MRI-Q
30
25
20
15
10
5
0
CUDA
Chapel w/ Explicit Transfer
Chapel w/ Implicit Transfer

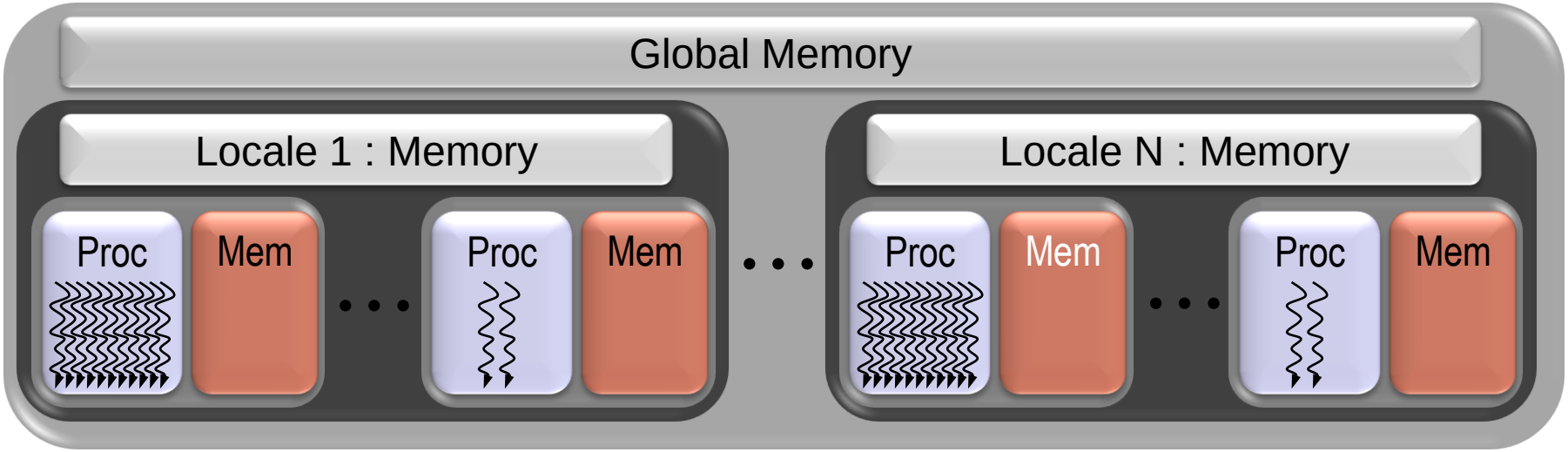
Rys Polynomial Equation Solver (RPES)



Code Size Comparison

Benchmark	# Lines (CUDA)	# Lines (Chapel)	% difference	# of Kernels
CP	186	154	17	1
MRI-FHD	285	145	49	2
MRI-Q	250	125	50	2
RPES	633	504	16	2
TPACF	329	209	36	1

Hierarchical Locales

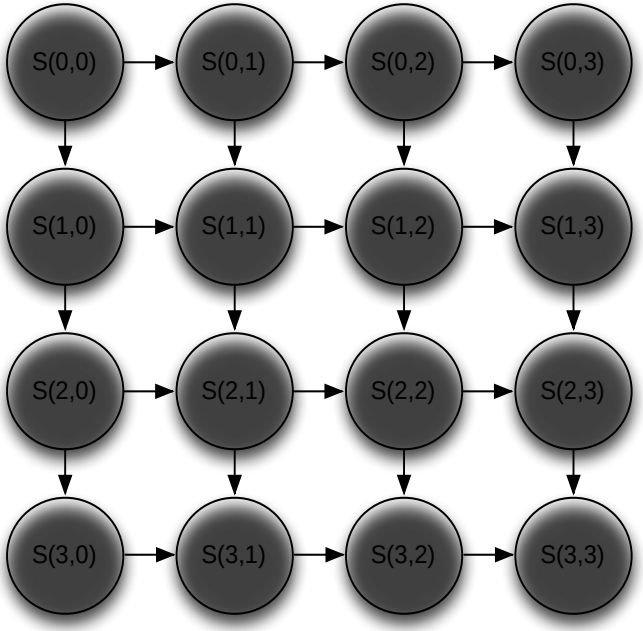


Programming Hierarchical Locales

- Three main components:
  1. **Hierarchical Machine Model**
     - Method to define the *machine* (locale hierarchy)
  2. **Task Execution Model**
     - *Synchronous* (`forall` loops, whole-array operations)
     - *Asynchronous* (`on`, `begin`, `cobegin`, `coforall`, etc.
  3. **Data Model**
     - Distribution of data across (and within) locales

In progress: Generalizing Parallel Loops

- Method to perform *wavefront* & *pipeline* computations
- Programmer provides annotated dependence information
- Transformations performed to parallelize loop



In progress: Compiling for Data Flow