Targeting GPUs and Other Hierarchical Architectures in Chapel

Albert Sidelnik
University of Illinois at Urbana-Champaign
Goals of this work

• Leverage Chapel to generate efficient data parallel code to execute on GPUs
• Provide the ability to support execution on any hierarchical parallel architecture
• Generalize parallel loops for efficiency and portability across many parallel architectures and runtimes
A = alpha * B + C

MOTIVATING EXAMPLE:
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

```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
HPC CHALLENGE (HPCC) STREAM Triad

```plaintext
cfg 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
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)

```c
#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];
}
```

CUDA

```c
#include <hpcc.h>
#include <omp.h>

static int VectorSize;
static double *a, *b, *c;
int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize(params, 3, sizeof(double), 0);
    a = HPCC_MALLOC(double, VectorSize);
    b = HPCC_MALLOC(double, VectorSize);
    c = HPCC_MALLOC(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;
    }
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }
    scalar = 3.0;
    for (j=0; j<VectorSize; j++) {
        a[j] = b[j]+scalar*c[j];
        HPCC_free(c);
        HPCC_free(b);
        HPCC_free(a);
        return 0;
    }
}
```
For STREAM, the Chapel and CUDA implementations match performance.
Parboil Benchmarks

MRI-FHD
- CUDA
- Chapel w/ Explicit Transfer
- Chapel w/ Implicit Transfer

MRI-Q
- CUDA
- Chapel w/ Explicit Transfer
- Chapel w/ Implicit Transfer

Two Point Angular Correlation Function
- CUDA
- Chapel w/ Explicit Transfer
- Chapel w/ Implicit Transfer

Coulombic Potential
- CUDA
- Chapel w/ Explicit Transfer
- Chapel w/ Implicit Transfer
Hierarchical Locales

- **Today**: Support for locality is horizontal across nodes
  - Lacking support for vertical locality (within a node)

- **Challenge**: How to map a computation onto a deeply hierarchical parallel architecture
Programming Hierarchical Locales

• Three main components:
  1. Hierarchical Machine Model
     • Method to define the *machine* (locale hierarchy)
  2. Task Execution Model onto locale (or sublocale)
     • 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 as simple loops
- Programmer provides annotated dependence information
- Compiler performs necessary transformations to parallelize the loop
In progress: Compiling for Data Flow

• Transformation to convert original Chapel program into a task graph
  • Nodes represent an atomic unit of computation
  • Edges express dependences between tasks
• Compiler maps the graph onto a data flow runtime and architecture
Any questions: asiden2 at illinois.edu
Thanks!