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

\[ A = \alpha \times B + C \]
HPC CHALLENGE (HPCC) STREAM Triad

\[
\text{config const } m = 1000;
\]

\[
\text{const alpha } = 3.0;
\]

\[
\text{const ProbSpace } = [1..m];
\]

\[
\text{var A, B, C: [ProbSpace] real;}
\]
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
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
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

```haskell
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

```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 need for explicit transfers of data between host and device (e.g. cudaMemcpy(...))

No changes required to the computation for other architectures

Distribution to target a GPU

Arrays are declared on the GPU device

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

No changes required to the computation for other architectures
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];
}

#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif
static int VectorSize;
static double *a, *b, *c;
int HPCC_StartStream(HPCC_Params *params) {
    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 (doIO) {
        fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
        fclose( outFile );
    }
    return 1;
}
#else
#pragma omp parallel for
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }
#else
#pragma omp parallel for
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];
#endif
HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
return 0;

CUDA

MPI + OpenMP

illinois.edu

10
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)

The graph shows the time (in milliseconds) for different methods: CUDA, Chapel with Explicit Transfer, and Chapel with Implicit Transfer. The y-axis represents the time, ranging from 0 to 200 milliseconds. The data transfer and compute times are indicated by different bars for each method.
MRI

MRI-FHD

MRI-Q

Time (ms)

CUDA
Chapel w/ Explicit Transfer
Chapel w/ Implicit Transfer

CUDA
Chapel w/ Explicit Transfer
Chapel w/ Implicit Transfer
Rys Polynomial Equation Solver (RPES)

![Graph showing time (ms) for CUDA, Chapel with explicit transfer, and Chapel with implicit transfer.](image-url)
# Code Size Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Lines (CUDA)</th>
<th># Lines (Chapel)</th>
<th>% difference</th>
<th># of Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>186</td>
<td>154</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>MRI-FHD</td>
<td>285</td>
<td>145</td>
<td>49</td>
<td>2</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>250</td>
<td>125</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>RPES</td>
<td>633</td>
<td>504</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>TPACF</td>
<td>329</td>
<td>209</td>
<td>36</td>
<td>1</td>
</tr>
</tbody>
</table>
Hierarchical Locales

Global Memory

Locale 1: Memory

Locale N: Memory
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

```
CBB1
A(1) = ...;
A(3) = ...;
if (A(1) > 3) {
    CBB2
    A(2) = ...;
   forall i in N {
        CBB3
        for j in M {
            A(i) = f(i,j);
        }
        A(i) += ...;
    }
} else {
    CBB4
   forall i in N{
        B(i) = ...;
    }
    B(3) = ...;
}
writeLn(A);
```