Hewlett Packard Enterprise

TARGETING GPUS USING CHAPEL'S LOCALITY AND PARALLELISM FEATURES

Engin Kayraklioglu, Andy Stone, David Iten
Sarah Nguyen, Michael Ferguson, Michelle Strout

CHIUW 2022 - June 10th

GPU SUPPORT
Background

- We have been working on adding native GPU support to Chapel
  - One of the most sought after features among users
  - Earlier collaborations with academia and industry influenced the design

Vision

- Order-independent, vectorizable Chapel loops can execute on GPU
- ‘on’ statements can be used to choose between GPU and CPU for execution and allocations

AN OVERVIEW WITH IDIOMS

SINGLE GPU STREAM

```
config const n = 1_000_000;

on here.gpus[0] {
  var A: [0..<n] int;
  foreach a in A do
    a += 1;
}
```

‘on’ statement controls the execution/allocation policy

‘A’ will be allocated on the Unified Virtual Memory

‘foreach’ will turn into a GPU kernel

Sidebar

- ‘forall’ and loop can also be used here
- Similarly, promoted expressions are also turned into GPU kernels

```
on here.gpus[0] {
  var A: [0..<n] int;
  A += 1; // this will launch as a kernel
}
```

MULTI-GPU STREAM

```
config const n = 1_000_000;

coforall gpu in here.gpus do on gpu {
  var A: [0..<n] int;
  foreach a in A do
    a += 1;
}
```

Multiple GPUs within one node can be used with a 'coforall … on' idiom
(nothing specific for GPU, really)

DATA OFFLOAD

```
config const n = 1_000_000;
var A: [0..<n] int;

// populate A on the host

on here.gpus[0] {
  var AonGPU = A;
  foreach a in AonGPU do
    a += 1;
}
```

'A' will be allocated on the host as usual

This will be a bulk copy from regular host memory to UVM

CPU-GPU OVERLAP

Note: There may be better idioms for expressing the same operation in the future

```
var A: [0..<n] int;

// assign half the work to CPU, the rest to GPUs. Assume divisibility
const numGPUs = here.getChildCount()-1;
const cpuSize = n/2;
const gpuSize = (n/2)/numGPUs;

cobegin {
  A[0..<cpuSize] += 1;

  coforall gpuID in 0..#numGPUs do on here.gpus[gpuID] {
    const myShare = cpuSize+gpuSize*(gpuID-1)..#gpuSize;

    var AonThisGPU = A[myShare];
    AonThisGPU += 1;
    A[myShare] = AonThisGPU;
  }
}
```

Compute 'gpuSize' and 'cpuSize' based on the decomposition

Two concurrent tasks

CPU works on its part

GPUs work on their part and copy the result back

MULTILOCALE/MULTIGPU STREAM

```
// Assume A, B, C local arrays on Locales[0]
coforall (l,lid) in zip(Locales, LocaleSpace) do on l {
  const locChunk = ...;
  var Al: [locChunk] int;
  var Bl = B[locChunk], Cl = C[locChunk];

  const numGPUs = here.gpus.size;
  coforall (g,gid) in zip(here.gpus, here.gpus.domain) do on g {
    const gpuChunk = ...;
    var Ag: [gpuChunk] int;
    var Bg = Bl[gpuChunk], Cg = Cl[gpuChunk];

    Ag = Bg + alpha * Cg;

    Al[gpuChunk] = Ag;
  }
  A[locChunk] = Al; }
```

Create local array

Copy a chunk of the remote array

Create device array

Copy a chunk of the host array

Copy device chunk to host

Copy local chunk to remote


NUTS ‘N BOLTS

LOOP OUTLINING

```
on here.gpus[0] {
  var A: [0..<n] int;
  foreach i in A.domain do
    A[i] += 1;
}
```

Kernel launch vs Host loop is chosen based on an execution-time check

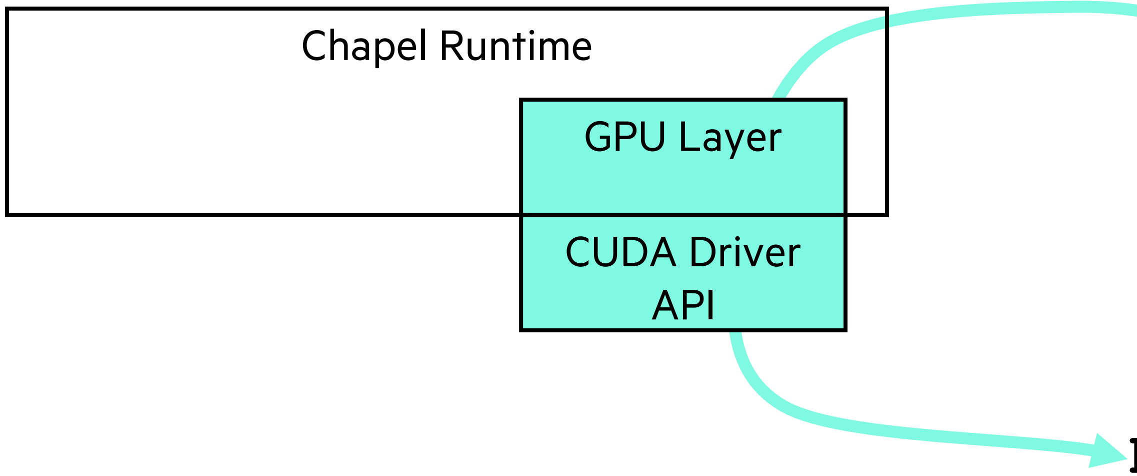
```
on here.gpus[0] {
  var A: [0..<n] int;
  if runningOnGPUSubLoc() then
    launch("kernel1", ...);
  else
    foreach i in A.domain do
      A[i] += 1;
}
```

Loop body is transformed into the function's body

Compiler generates the function stub and index computation

```
proc kernel1(A, ...) {
  // compute based on A.domain's bounds and
  // CUDA thread coords
  const i = ...;

  A[i] += 1;
}
```

KERNEL LAUNCH

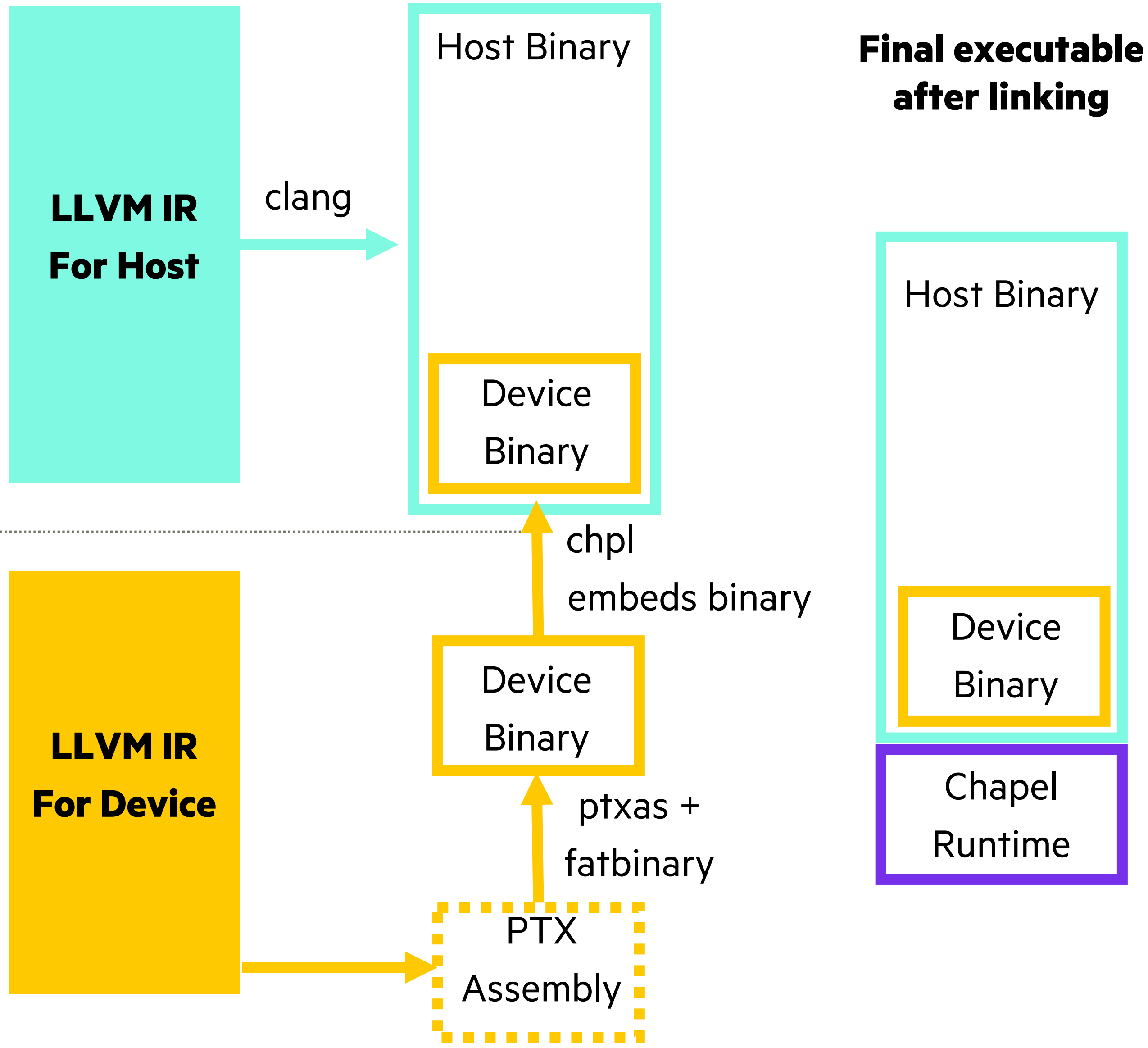



```
on here.gpus[0] {
  var A: [0..<n] int;
  if runningOnGPUSubLoc() then
    launch("kernel1", ...);
  else
    foreach i in A.domain do
      A[i] += 1;
}

proc kernel1(A, ...) {
  // compute based on A.domain's bounds and
  // CUDA thread coords
  const i = ...;

  A[i] += 1;
}
```

CHAPEL'S GPU COMPILATION TRAJECTORY

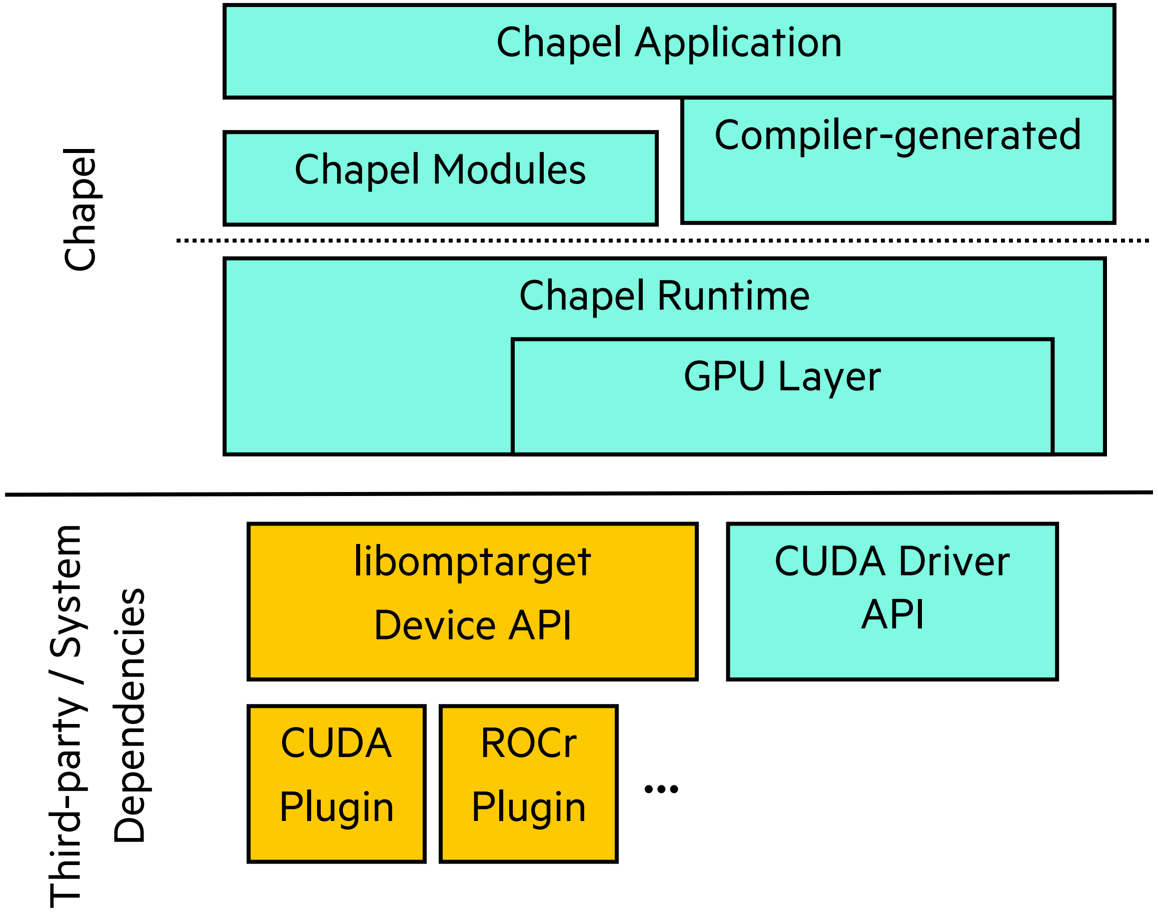
```
on here.gpus[0] {
  var A: [0..<n] int;
  if runningOnGPUSubLoc() then
    launch("kernel1", ...);
  else
    foreach i in A.domain do
      A[i] += 1;
}
```

```
proc kernel1(A, ...) {
  // compute based on A.domain's bounds
  // and CUDA thread coords
  const i = ...;

  A[i] += 1;
}
```



WHAT’S NEXT?

VENDOR PORTABILITY



- Current GPU layer is a wrapper around CUDA Driver API
- Vendor portability is a key goal

Ongoing Work

- We are investigating using libomptarget's Device API
- Used by clang to implement OpenMP target clause
  - Comes with plugins for different architectures

GPU-DRIVEN COMMUNICATION

- Currently, GPU support can be used with multilocale execution
  - However, GPU kernels have implied ‘local’ blocks in them
  - In other words, communication initiated from/to GPU is not allowed

- We are working on supporting GPU-driven communication
  - Still in the very early phases of assessing what’s needed
  - NVSHMEM is a good example for us at a high-level
    - But it uses InfiniBand Verbs API
  - We are planning to investigate GASNet EX memory kinds as the first step
  - Further down the road: support ugni and ofi communication layers

DISTRIBUTED ARRAY SUPPORT

Note: Work in progress. This snippet does not work as of 1.26

```
use BlockDist;

var Dom = {1..n} dmapped Block({1..n}, targetLocales=here.gpus);
var Arr: [Dom] int;

Arr = 5;
```

’Arr’s data will be allocated on GPU(s)

Promoted expression will run on GPU(s)

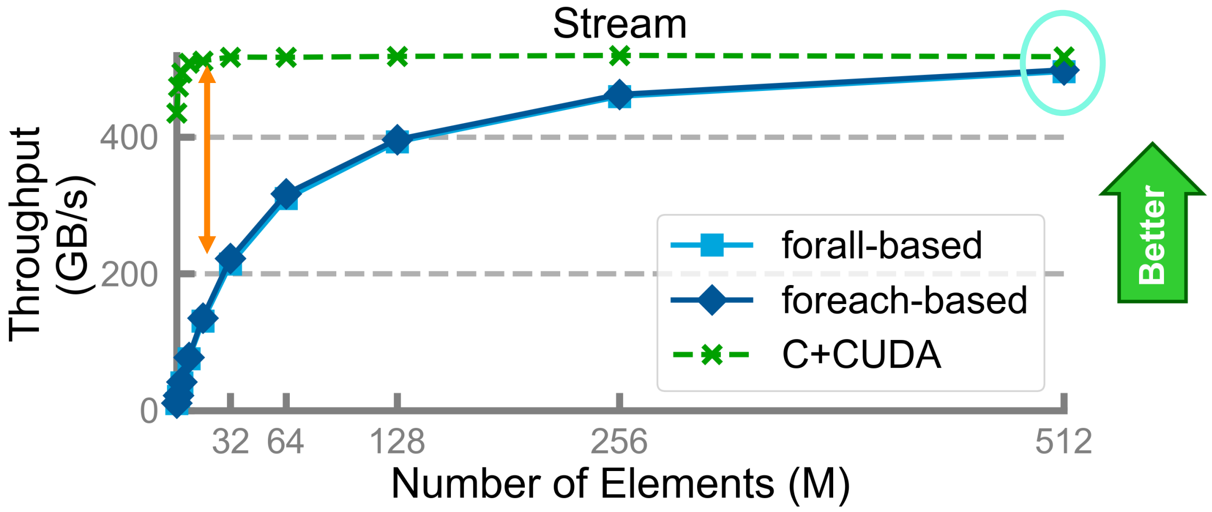
DESIGN DISCUSSIONS

- New features for forall loops:
  - Supporting queries for task/vector lane IDs
    - Not limited to GPU context
    - Can enable powerful SPMD-like programming idioms, too
  - Supporting shared memory allocations
  - Supporting block synchronization

- Designed features will be extended to foreach loops as well

- See https://github.com/chapel-lang/chapel/issues/16405

PERFORMANCE TUNING



At smaller vector sizes throughput is low

At larger vector sizes efficiency reaches 96%

Observations

- Can perform comparably to hand-written code
- Gets close to 100% efficiency with large datasets
- ‘foreach’ is slightly faster than ‘forall’

Potential Sources of Overhead

- Unified memory vs. device memory
- Dynamic allocations per kernel launch
- Dynamic kernel load

Future Work for Performance

- Understand the performance with small vectors
- Profile the remaining costs
- Study other benchmarks

SUMMARY

- Chapel's GPU support will rely on existing semantics as much as possible
  - Intuitive GPU programming for Chapel programmers

- GPU support is still under development

- Chapel's language constructs for parallelism and locality suit GPU programming well

THANK YOU

engin@hpe.com