OUTLINE

- Update on GPU Support
- Compiler Rework Update
- Attributes
- First-Class Functions and Closure Support
- User How-to Documentation
- Ongoing Library Efforts
- Chapel 2.0 / Standard Library Stabilization
UPDATE ON GPU SUPPORT
BACKGROUND

Overview

- We are developing support for GPU programming from Chapel
  - GPUs are very common, yet challenging to program
  - GPU support is frequently asked about at Chapel presentations
  - it would improve upon Chapel’s “any parallel algorithm on any parallel hardware” theme

Collaborations / External Studies

- early work at UIUC
- partnership with AMD [1] [2] [3]
- recent work from Georgia Tech and ANU, featured at CHIUW 2019, CHIUW 2020 and CHIUW 2021
- meanwhile, user applications have run on GPUs via Chapel interoperability features (e.g., ChOp and CHAMPS)

Releases

- 1.23: Design effort and discussions started
- 1.24: Can use non-user-facing features to generate GPU binaries for Chapel functions and launch them
- 1.25: Can natively generate Chapel functions from order-independent loops and launch them
BACKGROUND
Vision

Memory/Locality Management
• Chapel’s locale model concept supports describing a compute node with a 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

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
  • Kernels are generated for every call inside the loop body
**Memory/Locality Management**

- Chapel’s locale model concept supports describing a compute node with a 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

**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
  - Kernels are generated for every call inside the loop body
BACKGROUND
Sample Computation: Our Goal

Our Goal:

```javascript
on here.getGPU(0) {
    var a = [1, 2, 3, 4, 5];
    forall aElem in a do
        aElem += 5;
}
```
BACKGROUND
Sample Computation: Our Goal

Our Goal:

```javascript
on getGPU(0) {
    var a = [1, 2, 3, 4, 5];
    forall aElem in a do
        aElem += 5;
}
```

This method name is notional—the precise locale model and method for referring to the GPU sublocale are still under discussion.
BACKGROUND
Sample Computation: Status After 1.24

**Chapel**

```chapel
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
    a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [1, 2, 3, 4, 5];
__primitive("gpu kernel launch", funcPtr,
    <grid and block size>,..., c_ptrTo(a), ...);
writeln(a);
```

**Manual CUDA Driver API Calls**

```chapel
extern {
    #define FATBIN_FILE "chpl__gpu.fatbin"
    double createFunction(){
        fatbinBuffer = <read FATBIN_FILE into buffer>
        cuModuleLoadData(&cudaModule, fatbinBuffer);
        cuModuleGetFunction(&function, cudaModule, "add_nums");
    }
}
```

Read fat binary and create a CUDA function
BACKGROUND
Sample Computation: Status After 1.24

Chapel

```
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
  a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [1, 2, 3, 4, 5];
__primitive("gpu kernel launch", funcPtr,
  <grid and block size>,..., c_ptrTo(a), ...);
writeln(a);
```

Manual CUDA Driver API Calls

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
      "add_nums");
  }
}
```

In 1.25: Kernel and launch is created from 'forall' by the compiler
Sample Computation: Status After 1.24

**Chapel**

```chapel
pragma "codegen for GPU"
export proc add_nums(a: c_ptr(real(64))){
    a[0] = a[0]+5;
}

var funcPtr = createFunction();
var a = [1, 2, 3, 4, 5];
__primitive("gpu kernel launch", funcPtr,
    <grid and block size>,
    c_ptrTo(a), ...);
writeln(a);
```

**Manual CUDA Driver API Calls**

```c
extern {
    #define FATBIN_FILE "chpl__gpu.fatbin"

double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
        "add_nums");
}
```

**In 1.25: Kernel and launch is created from 'forall' by the compiler**

**1.25: Function is loaded and launched by the runtime**
**BACKGROUND**
Sample Computation: Status in 1.25

**Our Goal:**

```javascript
on here.getGPU(0) {
  var a = [1, 2, 3, 4, 5];
  forall aElem in a do
    aElem += 5;
}
```

**What works today (1.25):**

```javascript
on here.getChild(1) {
  var a = [1, 2, 3, 4, 5];
  forall aElem in a do
    aElem += 5;
}
```
The current GPU Locale Model

- The ‘gpu’ locale model is used

Diagram:

- Locale
  - Child 0 (CPU)
  - Child 1 (GPU)

One child per device...

...
The current GPU Locale Model

- The ‘gpu’ locale model is used
  
  ```
  var a: [1..n] real;
  on here.getChild(1) {
    var b: [1..n] real;
    ...
  }
  ```

- Allocations on GPU sublocales are done with unified memory
  - Accessible from both host and device
  - Allows initializing class instances allocated on device from host
- May have implications for array data
  - Potentially higher cost per access
  - Future releases will support both unified and device-exclusive memory
**THIS EFFORT**
Potential future locale models

- Currently, there’s no real difference between ‘on Locale[i]’ and ‘on Locale[i].getChild[0]’
- The current approach was adopted from earlier work without much consideration
- So now, we’re brainstorming alternate models

### What we have now (sublocale 0 = CPU)

<table>
<thead>
<tr>
<th>Locale</th>
<th>Child 0 (CPU)</th>
<th>Child 1 (GPU)</th>
<th>Child 2 (GPU)</th>
<th>…</th>
</tr>
</thead>
</table>

**NUMA aware (flat)**

<table>
<thead>
<tr>
<th>Locale</th>
<th>Child 0 (CPU)</th>
<th>Child 1 (CPU)</th>
<th>Child n (GPU)</th>
<th>Child n+1 (GPU)</th>
<th>…</th>
</tr>
</thead>
</table>

### Locale for CPU; sublocales for GPUs

<table>
<thead>
<tr>
<th>Locale (CPU)</th>
<th>Child 0 (GPU)</th>
<th>Child 1 (GPU)</th>
<th>…</th>
</tr>
</thead>
</table>

**NUMA aware (hierarchical)**

<table>
<thead>
<tr>
<th>Locale</th>
<th>Child 0 (CPU)</th>
<th>Child n+1 (GPU)</th>
<th>…</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Child 0 (GPU)</th>
<th>…</th>
<th>Child 0 (GPU)</th>
<th>…</th>
</tr>
</thead>
</table>
**THIS EFFORT**
Creating GPU Kernels from Loops

**User’s loop**

```latex
forall i in 1..n do arr[i] = i*mul
```
Creating GPU Kernels from Loops

**User's loop**

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

```c
for (i=1 ; i<=n ; i++) {  // order-independent loop
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```
Creating GPU Kernels from Loops

User’s loop

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

Conceptual C loop

```
for (i=1 ; i<=n ; i++) {
    // order-independent loop
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

pragma "codegen for GPU"

```
proc kernel(
)
```

Generated GPU Kernel
Creating GPU Kernels from Loops

User’s loop
forall i in 1..n do arr[i] = i*mul

for (i=1 ; i<=n ; i++) {
  // order-independent loop
  var arrData = arr->data;
  ref addrToChange = &arrData[i];
  var newVal = i*mul;
  *addrToChange = newVal;
}

pragma “codegen for GPU”
proc kernel(in startIdx, in endIdx)

The loop’s start and end indices are passed by value
**THIS EFFORT**

Creating GPU Kernels from Loops

---

### User’s loop

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

```plaintext
for (i=1 ; i<=n ; i++) { // order-independent loop
  var arrData = arr->data;
  ref addrToChange = &arrData[i];
  var newVal = i*mul;
  *addrToChange = newVal;
}
```

**pragma “codegen for GPU”**

```plaintext
proc kernel(in startIdx, in endIdx, ref arrArg, in mulArg) {
```

---

### The loop’s start and end indices are passed by value

### Outer variables are passed depending on their type

---

Conceptual C loop

---

Generated GPU Kernel
**THIS EFFORT**
Creating GPU Kernels from Loops

**User’s loop**

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

```
for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

**The loop’s start and end indices are passed by value**

**Outer variables are passed depending on their type**

**Loop body is copied**

**Conceptual C loop**

**Generated GPU Kernel**

```c
pragma "codegen for GPU"
proc kernel(in startIdx, in endIdx, ref arrArg, in mulArg) {
    var arrData = arrArg->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```
Creating GPU Kernels from Loops

User’s loop

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

for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}

pragma “codegen for GPU”
proc kernel(in startIdx, in endIdx, ref arrArg, in mulArg) {
    var arrData = arrArg->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
Creating GPU Kernels from Loops

User's loop

\[
\text{forall } i \text{ in } 1..n \text{ do } \text{arr}[i] = i \times \text{mul}
\]

Conceptual C loop

\[
\text{for } (i=1 \text{ ; } i<=n \text{ ; } i++) \{ \quad \text{// order-independent loop} \\
\text{var } \text{arrData} = \text{arr}->\text{data}; \\
\text{ref } \text{addrToChange} = &\text{arrData}[i]; \\
\text{var } \text{newVal} = i \times \text{mul}; \\
*\text{addrToChange} = \text{newVal}; \\
\}
\]

pragma "codegen for GPU"

\[
\text{proc } \text{kernel} (\text{in startIdx}, \text{in endIdx}, \text{ref arrArg}, \text{in mulArg}) \{ \\
\text{var } \text{index} = \ldots; \quad \text{// calculate and return if >length} \\
\text{var } \text{arrData} = \text{arrArg}->\text{data}; \\
\text{ref } \text{addrToChange} = &\text{arrData}[\text{index}]; \\
\text{var } \text{newVal} = \text{index} \times \text{mulArg}; \\
*\text{addrToChange} = \text{newVal}; \\
\}
\]
Launching GPU Kernels

**User’s loop**

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

```c
for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

**Kernel signature**

```c
proc kernel(in startIdx, in length, ref arrArg, in mulArg);
```
Launching GPU Kernels

User’s loop

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

Conceptual C loop

```c
for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

Generated Kernel Launch

```c
launch_kernel(
);
```

Kernel signature

```c
proc kernel(in startIdx, in length, ref arrArg, in mulArg);
```
Launching GPU Kernels

**THIS EFFORT**

User’s loop

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

Conceptual C loop

```c
for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

Generated Kernel Launch

```c
launch_kernel("kernel",...(n));
```

Kernel signature

```c
proc kernel(in startIdx, in length, ref arrArg, in mulArg);
```

Function name

`launch_kernel("kernel",...(n));`
THIS EFFORT
Launching GPU Kernels

User’s loop
forall i in 1..n do arr[i] = i*mul

for (i=1; i<=n; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}

Generated Kernel Launch

launch_kernel("kernel", n-1, 512, );

Kernel signature
proc kernel(in startIdx, in length,
            ref arrArg, in mulArg);

Function name

Loop length and block size are used for dimension calculation
Launching GPU Kernels

User’s loop

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

for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}

Generated Kernel Launch

launch_kernel(“kernel”, n-1, 512, 1, 0, n, 0, , mul, 0);

Kernel signature

proc kernel(in startIdx, in length, ref arrArg, in mulArg);

Function name

Loop length and block size are used for dimension calculation

Pass-by-value arguments have an accompanying 0

THIS EFFORT
Launching GPU Kernels

User's loop

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

Conceptual C loop

```c
for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

User's loop

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

Generated Kernel Launch

```c
launch_kernel(“kernel”, n-1, 512, 1, 0, n, 0, &arr, 32, mul, 0);
```

Kernel signature

```c
proc kernel(in startIdx, in endIdx, ref arrArg, in mulArg);
```

Function name

Loop length and block size are used for dimension calculation

Pass-by-value arguments have an accompanying 0

Pass-by-offload arguments have an accompanying copy size
Launching GPU Kernels

User’s loop

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

Conceptual C loop

```c
for (i=1 ; i<=n ; i++) {
    var arrData = arr->data;
    ref addrToChange = &arrData[i];
    var newVal = i*mul;
    *addrToChange = newVal;
}
```

Generated Kernel Launch

```c
if executingOnGPUSublocale() then
    launch_kernel("kernel", n-1, 512, 1, 0, n, 0, &arr, 32, mul, 0);
else
    // loop with no change
```

Kernel signature

```c
proc kernel(in startIdx, in length, ref arrArg, in mulArg);
```

Function name

Loop length and block size are used for dimension calculation

Pass-by-value arguments have an accompanying 0

Pass-by-offload arguments have an accompanying copy size

A dynamic check for GPU execution is added
Kernel function

```plaintext
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; }

    // copied loop body
}
```
Translating Loop Indices Into Kernel Indices

**Kernel function**

```plaintext
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; }

    // copied loop body

}  
```

Primitives correspond to CUDA threadIdx, blockIdx, blockDim, and blockDim variables

Primitives lower to calls to corresponding llvm intrinsic functions
(e.g. llvm.nvvm.read.ptx.sreg.ctaid.x)
Translating Loop Indices Into Kernel Indices

**Kernel function**

```haskell
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; }

    // copied loop body
}
```

**Primitives correspond to CUDA threadIdx, blockIdx, blockDim, and gridDim variables**

Primitives lower to calls to corresponding llvm intrinsic functions
(e.g. llvm.nvvm.read.ptx.sreg.ctaid.x)

**Index computation**

Currently we are only targeting 1-dimensional kernels
**THIS EFFORT**
Translating Loop Indices Into Kernel Indices

---

**Kernel function**

```plaintext
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; }

// copied loop body

}
```

---

**Primitives correspond to CUDA threadIdx, blockIdx, blockDim, and gridDim variables**

Primitives lower to calls to corresponding llvm intrinsic functions
(e.g. llvm.nvvm.read.ptx.sreg.ctaid.x)

---

**Index computation**

Currently we are only targeting 1-dimensional kernels

**Check that index is in bounds**

Can occur if length is not evenly divisible by block size
THIS EFFORT
Translating Loop Indices Into Kernel Indices

Kernel function
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; }

// copied loop body
}

Primitives correspond to CUDA threadIdx, blockIdx, blockDim, and gridSize variables

Primitives lower to calls to corresponding llvm intrinsic functions (e.g. llvm.nvvm.read.ptx.sreg.ctaid.x)

Index computation
Currently we are only targeting 1-dimensional kernels

Check that index is in bounds
Can occur if length is not evenly divisible by block size

Loop body is copied
**Putting the Pieces Together**

**User’s loop**

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

**The loop is replaced with:**

```plaintext
if executingOnGPUSublocale() then
    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:**

```plaintext
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;
}
```
Loop Eligibility

- An analysis marks loops as “GPU eligible”
  - We copy and outline eligible loops into a function that can be called on the GPU (i.e., a kernel)
  - Ineligible loops are always executed on the CPU
  - Eligible loops are executed on the GPU if called in a GPU locale
- To be eligible, a loop must be:
  - Order-independent (e.g., ‘forall’ or ‘foreach’)
  - In user code
  - Free of any function calls except those that have been inlined
  - Only use primitives that are “fast” and “local”
    - “fast” means “safe to run in an active message handler”
    - “local” means “doesn’t cause any network communication”
Loop Eligibility

- An analysis marks loops as “GPU eligible”
  - We copy and outline eligible loops into a function that can be called on the GPU (i.e., a kernel)
  - Ineligible loops are always executed on the CPU
  - Eligible loops are executed on the GPU if called in a GPU locale
- To be eligible, a loop must be:
  - Order-independent (e.g., ‘forall’ or ‘foreach’)
  - In user code
  - Free of any function calls except those that have been inlined
  - Only use primitives that are “fast” and “local”
    - “fast” means “safe to run in an active message handler”
    - “local” means “doesn’t cause any network communication”

**Ineligible:**

```plaintext
for i in 1..10 { a[i] += 10; }
forall i in 1..10 { a[i] += foo(); }
```

**Eligible:**

```plaintext
forall i in 1..n { a[i] += i+10; }
foreach i in 1..n { a[i] += i+10; }
```
STREAM

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

• Arrays are allocated in unified memory
• Scalars are allocated on the function stack
  • So, they are on host memory
on here.getChild(1) {
  var a, b, c: [1..n] real;
  const alpha = 2.0;

  forall bElem in b do bElem = 1.0;
  forall cElem in c do cElem = 2.0;
}

- 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
Stream

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

  forall bElem in b do bElem = 1.0;
  forall cElem in c do cElem = 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
An Early Performance Study

![Graph showing throughput vs. number of elements for different methods: forall-based, foreach-based, C+CUDA. The graph indicates that C+CUDA performs better than the other methods as the number of elements increases.](image-url)
**STATUS**

An Early Performance Study

At smaller vector sizes throughput is low

At larger vector sizes efficiency reaches 96%
**An Early Performance Study**

**STATUS**

- 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
**NEXT STEPS**

**Near-Term Goals: Implementation**

- Generate single binary
  - We currently generate a GPU binary and dynamically load the GPU kernel from that binary

- Support reductions

- Allow non-inlined function calls inside loop bodies
  - We do not mark functions called from loops as a GPU kernel
  - Challenge: What if the function has a system call, or ‘halt’?

- Enable communication between device and host
  - The host can access device memory thanks to unified memory, but not vice versa
  - We want to use the existing wide pointers and comm interface to handle this

- Support multiple GPUs in one node
  - The ‘gpu’ locale model creates as many children as number of devices on the node
  - However, the runtime always uses device 0
Near-Term Goals: Design Discussions

• What should the locale model be? (#18529)

• How do we enable block/grid size to be determined by the user?
  • Should there be a new feature for order-independent loops to support that? How can that be portable?

• How can we handle intra-block synchronization?

• Should tasks of a ‘forall’ “know” their IDs?

• Should there be a way to write and launch a GPU kernel explicitly in Chapel?
**NEXT STEPS**

Long-Term Goals

- Support AMD and Intel GPUs
- Support inter-locale communication from GPUs
- Performance optimizations
COMPILER REWORK UPDATE
COMPILER REWORK
OUTLINE

• Introduction and Motivation
• Progress on Untyped AST
• Progress on Resolution
• Summary
PROBLEMS WITH THE CURRENT CHAPEL COMPILER

**Speed**
- The current compiler is generally slow, and extremely so for large programs (~7s to 15 minutes)
- Large programs require complete recompilation whenever a change is made

**Errors**
- For incorrect programs, the compiler frequently displays only some of the errors at a time
- Compilation errors can be hard for users to understand and resolve

**Structure and Program Representation**
- The compiler is structured only for whole-program analysis, preventing separate/incremental compilation
- Unclear how to integrate an interpreter, provide IDE support, or ‘eval’ Chapel snippets
- Compilation passes are highly coupled

**Development**
- The modularity of the compiler implementation needs improvement
- There is a steep learning curve to become familiar with the compiler implementation
COMPILER REWORK DELIVERABLES

Incremental Compilation Frontend
• Only reparse and do type resolution on files that were edited
• Could result in reducing compilation time
• Will still have the whole-program optimization and code-generation back-end

Separate Compilation
• Make most of the whole-program optimization happen per file
• Will need a linking step for optimizations like function inlining that span files
• Should result in significantly faster compilation times

Dynamic Compilation and Evaluation
• Enable Chapel code snippets to be written and run interactively
  • e.g., in Jupyter notebooks

Throughout the effort, working towards improving the learning curve and error messages.
The new front-end will use an untyped AST (uAST) to represent the code.

Began work implementing these changes after the 1.24 release.
**Current Status**
- Can parse and represent about ¾ of the Chapel language in uAST
- Can go from source code from one file to the production compiler AST to working generated code
  - Included library code is still being built with the production compiler
  - Type resolution is well underway including some resolution of generics
PROGRESS ON UNTYPED AST
Around 70 classes have been added to the untyped AST (uAST) class hierarchy
Each uAST class includes comments that generate API documentation
Unlike old AST, prefer one class per feature instead of overloading a single class
  - For example, old AST used CallExpr to represent primitives and function calls
  - In the uAST, these are represented by PrimCall and FnCall

Class hierarchy organizes different classes based on functionality, e.g...
A new parser creates uAST from source code

- ‘--compiler-library-parser’ enables this parser for files named on the command line
- All other modules are currently processed by the old parser

A new pass in the old compiler can translate uAST to the old AST

- Most translation is done using the same builders the production compiler’s parser uses

```
module Mod {
  var x = 8;

  proc f(arg: int) {}
  f(x);
}
```
PROGRESS ON UNTYPED UAST

- All of the “Hello Worlds” in ‘release/examples’ can be compiled with the new front-end

- 9 out of 41 primers in ‘release/examples/primers’ can be compiled with the new front-end

- At present, only the files named on the command line are parsed by the new front-end

- When more primers can pass, the next step is to try compiling internal modules as well
  - This will work out many remaining bugs and edge cases

- The new parser can replace the old one when all tests pass with the new front-end
PROGRESS ON RESOLUTION
Resolution is the process of determining what each symbol refers to and its type

```plaintext
var x: int; // what is the type of 'x'? 
f(x); // what does 'x' refer to? which 'f' function is called?
proc f(x) {
    writeln(x); // which x does this refer to?
}
```

Resolution can be divided into scope resolution and type resolution (see next slides)

New resolution code is implemented as incremental queries
- each query has the same output when given the same input
- queries are memoized—repeated invocations will reuse the computed result
- queries are recomputed when needed when the input changes
- at present, these query results are only saved in memory in a running compiler process
• Scope resolution is the process of determining which symbols a given name can refer to
• The result might be one or many symbols
• It handles details of inner and outer scopes, ‘use’ statements, and ‘import’ statements

For example, the prototype can incrementally determine the information described below:

```plaintext
module Lib {
    var x;
    proc f(arg: int) { } // #1
    proc f(arg: string) { } // #2
}

module Program {
    use Lib;
    x;
    f(...);
    // Scope includes ‘x’ and ‘f’ from ‘Lib’
    // Refers to ‘Lib.x’ above
    // Refers to ‘f’ #1 or #2 above
}
```
TYPE RESOLUTION

- Type resolution is the process of determining the types of all expressions + which functions are called
- Makes use of the result of scope resolution
- Includes instantiating generic functions
- Includes determining values of params

- For example, the prototype can incrementally determine the information below:
  ```
  proc f(arg) {
    return arg;
  }
  ```

  ```
  param p = 1;
  ```

  ```
  var x = f(p);
  ```

  - ‘1’ is an ‘int’ and ‘p’ is ‘1’
  - ‘f’ call invokes an instantiation of ‘f’ with an ‘int’ argument
  - The instantiation returns an ‘int’ and therefore ‘x’ has type ‘int’
SUMMARY

Current Status

- Can parse and represent about ¾ of the Chapel language in uAST
- Can go from source code from one file to the production compiler AST to working generated code
  - Included library code is still being built with the production compiler
  - Type resolution is well underway including some resolution of generics
The Chapel compiler has problems with:
  - speed
  - error reporting
  - structure and program representation
  - its steep learning curve

This rework effort is addressing the speed problems through architectural adjustments:
  - to enable incremental compilation and separate compilation

During development, this rework effort is taking steps to improve the learning curve:
  - more modular design
  - generated API documentation

Next steps are to complete parsing for the full language and then complete the incremental resolver.
**ATTRIBUTES**

**Background**

- Attributes are metadata associated with a particular symbol
  - Attached using the same general syntax to enable easy support of less commonly used features

- Many languages have some form of support
  - Java has annotations
    ```java
    @Override // Annotation that indicates this method overrides one on the parent type
    void myOverrideMethod() { ... }
    ```

  - Rust has attributes
    ```rust
    #[deprecated] // Attribute that indicates function has been deprecated
    pub fn myDepFunc() { ... }
    ```

  - Python has decorators
    ```python
    @memorize // Decorator to store result of previous call to this function to speed up computation (defined by user)
    def fac():
    ```
Chapel's 'pragma' syntax has typically been used for similar purposes

```plaintext
pragma "no doc" // Hide this symbol from documentation output
var x = 17;
```

But pragmas are not intended for users

- Users also can't define their own pragmas since adding one requires modifying the compiler

Need a new feature

- Some pragmas could be converted to this new feature
- Others will need to remain pragmas
ATTRIBUTES

This Effort

• Investigated attribute/annotation syntax in other languages
• Conducted a survey among Chapel implementers about prospective attributes
  • Determined that there are sufficient prospective attribute candidates to warrant implementing this feature

• Agreed on 14 potential attributes, including:
  – opting out of optimizations on a per-symbol basis
  – deprecating a symbol
  – indicating maturity level of code
  – indicating code is stable/unstable
  – controlling warnings within a block
  – turning on ‘--warn-unstable’ for a particular module
  – controlling linter warning levels
  – marking a symbol as “no doc”
  – providing “version changed” info
  – marking a function as a test or a test function as ignored
**ATTRIBUTES**

Next Steps

- Determine syntax to use
  - Likely will involve ‘@’ like other languages

- Determine extent of attribute features to support
  - Should attributes have attributes? Should attributes be macro-like? etc.

- Implement attribute syntax
  - Convert ‘deprecated’ keyword to attribute
  - Implement remaining attributes with broad support

- Continue discussing prospective attributes that were less clear-cut, such as...
  - Indicate if a loop is SIMD (single instruction, multiple data)
  - Indicate a type is/isn’t compatible with copy elision
  - Indicate if type’s deinit method must be called at the end of a block
FIRST-CLASS FUNCTIONS AND CLOSURE SUPPORT
FIRST-CLASS FUNCTIONS AND CLOSURE SUPPORT

Background

• Chapel has prototypical support for first-class functions, anonymous functions, and function types

```chapel
proc foo(x: int) { writeln(x); }

// Capture a named function as a value
var fn1 = foo;
fn1(8);

// Capture an anonymous function as a value
var fn2 = lambda(x: int) { writeln(x); };
fn2(8);

// Construct the type of a function
type fnType = func(int, void);

// These three types are equal!
assert(fn1.type == fn2.type && fn2.type == fnType);
```
FIRST-CLASS FUNCTIONS AND CLOSURE SUPPORT

Closures

- A closure is a first-class function which refers to one or more outer (non-local) variables
  - Can be thought of as referring to a nested function by name

- Chapel does not currently offer support for creating closures
  - The following does not compile today:

```chapel
proc call(fn) { fn(); }

proc main() {
    var x = 0;
    var fn = lambda() { x += 1; };
    call(fn);
    assert(x == 1);
}
```
FIRST CLASS FUNCTIONS AND CLOSURE SUPPORT
Status, Next Steps

Status:
• Recently started developing a prototype to add support for closures
  – not far enough along to make it into 1.25

Next Steps:
• Extend the existing FCF infrastructure to support closures that cannot escape
  – A closure escapes when it outlives its declaration point (e.g., is returned from a procedure)
• Explore extending a function’s type to include its argument/return intents
• Consider adjustments to syntax for function types and anonymous functions
• Investigate optimizations to FCF infrastructure and ways to reduce cost of dynamic dispatch
  – Quietly optimizing to use a function pointer when possible?
  – Experiment with “lightweight” / “interface-backed closures”
USER HOW-TO DOCUMENTATION

Background:
- Have long wanted to create how-to examples illustrating common computational patterns
  - These would be complementary to the existing ‘primers’ examples that introduce specific language concepts

This Effort:
- Created a ‘how-to’ about reading in CSV files
  - In the 1.25 release, can be found in ‘examples/patterns/readmecsv.chpl’

Next Steps:
- Create additional how-to examples, such as:
  - k-mer counting in Chapel (common bioinformatics computation)
  - Calling existing CUDA kernels from within Chapel
  - Using Chapel to orchestrate distributed instances of Python program executions (ensemble computation)
  - Using Chapel in an MPI program for on-node parallelism
- Render the examples as part of Chapel’s online documentation, similar to the primer examples
ONGOING LIBRARY EFFORTS
• Apache Arrow Support
• Apache Parquet Support
• Go-Style Channels
• Socket Library
APACHE ARROW SUPPORT
**APACHE ARROW SUPPORT**

Background

- “Apache Arrow is a language-agnostic software framework” [Wikipedia]
  - Supports columnar storage via “record batches” and tables
  - Inter-process communication and zero-copy memory sharing

- Production implementation is written in C++
  - Also supports a GLib C interface, though it is not feature complete

- Supported to varying degrees by many modern programming languages
  - Rust, Python, Julia, etc.

- Supporting Apache Arrow could...
  - provide a means for Chapel programs to share data with one another
  - enable additional interoperability capabilities in Chapel
  - allow greater in-memory analytics options to Chapel users
APACHE ARROW SUPPORT
This Effort

Goal: add Chapel module that enables the use of Arrow data structures

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>‘foo’</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>‘bar’</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>‘baz’</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>

• Add support using Arrow data structures
• Use C-interoperability to call Apache Arrow GLib library
• Required ~100 lines of C code to create this simple Arrow table
APACHE ARROW SUPPORT

Impact

use Arrow;

var arrowIntArray = new ArrowArray([1,2,3,4]);

var arrowStringArray = new ArrowArray(["foo", "bar", "baz", "bogus value"],
validIndices=[0,1,2]);

var arrowBoolArray = new ArrowArray([true, false, false, true],
invalidIndices=[1]);

Create Arrow Arrays in Chapel

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>f1</td>
<td>f2</td>
</tr>
<tr>
<td>1</td>
<td>‘foo’</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>‘bar’</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>‘baz’</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>true</td>
</tr>
</tbody>
</table>

The Chapel version is much simpler
Impact

The Chapel version is much simpler

```
use Arrow;

var arrowIntArray = new ArrowArray([1,2,3,4]);

var arrowStringArray = new ArrowArray(['foo', 'bar', 'baz', 'bogus_value'],
    validIndices=[0,1,2]);

var arrowBoolArray = new ArrowArray([true, false, false, true],
    invalidIndices=[1]);

var rcbatch = new ArrowRecordBatch("f0", arrowIntArray,
    "f1", arrowStringArray,
    "f2", arrowBoolArray);

var table = new ArrowTable(rcbatch, rcbatch);
```
APACHE ARROW SUPPORT

Status and Next Steps

Status:

• Currently a draft PR while we finalize design decisions (\#18472)
• Can now express those \~100 lines of C code in \~5 lines of Chapel code to create the same Arrow table
• The proposed interface is a superset of the C interface

Next Steps:

• Extend Chapel interface
• Explore how distributed Chapel arrays can be integrated with Arrow
  – Can the Chapel memory allocator be used to allocate Arrow memory?
Background and This Effort

**Background:**

*Apache Parquet:*
- Widely used columnar file format for data analytics supported by Apache Arrow
- Strengths are interoperability, space efficiency, and query efficiency

**This Effort:**

- Added a high-level interface to...
  ...read Parquet file columns into distributed Chapel arrays
  ...write distributed Chapel arrays to Parquet files
- Explored use-cases in Arkouda as an early-adopter of this functionality
Impact

Prototype Chapel Interface

```chapel
use Arrow;
use BlockDist;

var filenames = ['file1.parquet', 'file2.parquet'];
var datasetname = 'first-int-col';
var (sizes, ty) = getArrSizeAndType(filenames, datasetname);
```

Get array size and type from file metadata
**Impact**

**Prototype Chapel Interface**

```chapel
use Arrow;
use BlockDist;

var filenames = ["file1.parquet", "file2.parquet"];  
var datasetname = "first-int-col";
var (sizes, ty) = getArrSizeAndType(filenames, datasetname);

var A = newBlockArr(0..#(+ reduce sizes), ty);
readParquetFiles(A, filenames, sizes, datasetname);
```

- Get array size and type from file metadata
- Read a Parquet column into a Chapel array
**APACHE PARQUET SUPPORT**

**Impact**

**Prototype Chapel Interface**

```chapel
use Arrow;
use BlockDist;

var filenames = ["file1.parquet", "file2.parquet"];
var datasetname = "first-int-col";
var (sizes, ty) = getArrSizeAndType(filenames, datasetname);

var A = newBlockArr(0..#(+ reduce sizes), ty);
readParquetFiles(A, filenames, sizes, datasetname);
writeDistArrayToParquet(A, "dist-parquet");
```

Get array size and type from file metadata

Read a Parquet column into a Chapel array

Write a Chapel array to Parquet files (1 per locale)
**APACHE PARQUET SUPPORT**

Status and Next Steps

**Status:**
- Recently moved from the Arrow GLib C interface to the C++ interface
- This is currently a draft PR as we work on installing Apache Arrow onto our testing machines

<table>
<thead>
<tr>
<th>I/O Type</th>
<th>Parquet</th>
<th>HDF5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>0.65 GiB/sec</td>
<td>2.15 GiB/sec</td>
</tr>
<tr>
<td>Write</td>
<td>0.11 GiB/sec</td>
<td>2.26 GiB/sec</td>
</tr>
</tbody>
</table>

**Next Steps:**
- Improve Parquet read performance
  - Explore parallelizing column reading, interacting with Parquet low level API for batch reading, etc.
- Explore compression and other available Parquet file features
GO-STYLE CHANNELS
GO-STYLE CHANNELS
Background and This Effort

**Background:** Chapel intends to support general parallel programming
- One missing idiom is a message queue like in Go and Rust (known as a ‘channel’ there)

**This Effort:** Implement Go-style channels in Chapel
- Implemented as a Google Summer of Code Project
  - Student: Divye Nayyar
  - Mentors: Michael Ferguson, Aniket Mathur (Chapel GSoC 2020 Alum)

**Status:** Design reviewed; Implementation PR [#18168](#) under review

**Next Steps:**
- Merge the PR
- Add compiler support for blocking ‘select’ statements (see example on next slide)
- Investigate and improve performance
- Enable channels to communicate across locales
// simple send/recv
use Channel;

var chan1 = new channel(int, 5);

begin {
    chan1.send(4);
}
var recv1 : int;
chan1.recv(recv1);
writeln("Received ", recv1);

// proposed select statement syntax
select {
    when var x1 = channel1.recv() {
        writeln("Received: ", x1);
    }
    when channel2.send(x2) {
        writeln("Sent: ", x2);
    }
}

// module currently supports a lower level
// way of expressing this
Background, This Effort

**Background:** TCP and UDP socket programming have not been supported in Chapel

- Could only be done through C interoperability
- Blocking socket calls are a mismatch for qthreads-based user-level tasking

**This Effort:** Implemented a Socket module in Chapel

- Implemented as a Google Summer of Code Project
  - Student: Lakshya Singh
  - Mentors: Ankush Bhardwaj (Chapel GSoC 2020 Alum), Krishna Kumar Dey (Chapel GSoC 2019 Alum), Michael Ferguson
use Socket;

var port:uint(16) = 8812;
var host = "127.0.0.1";
var addr = ipAddr.ipv4(IPv4Localhost, port);

proc server(srvSock: udpSocket) throws {
    writeln("Server recv'ing");
    var got = srvSock.recv(5);
    writeln("Server recv'd: ", got);
}

proc client() throws {
    var clientSock = new udpSocket();
    writeln("Client send'ing");
    var n = clientSock.send(b"hello", addr);
    writeln("Client sent ", n, " bytes");
}

proc main() throws {
    writeln("Creating server socket");
    var srvSock = new udpSocket();
    bind(srvSock, addr);

    cobegin {
        server(srvSock);
        client();
    }
}
SOCKET LIBRARY
Status, Next Steps

**Status:**
- Design has been reviewed and implementation PR [17960](#17960) is under review
- Uses ‘libevent’ to allow useful work in other Chapel tasks while waiting on network activity
- Implementation has some caveats at present:
  - only works with C back-end (e.g. CHPL_TARGET_COMPILER=gnu)
  - only works with CHPL_TASKS=qthreads
  - requires ‘libevent’ header to be installed in ‘/usr/include’

**Next Steps:**
- Complete PR review
- Address caveats listed above
- Use the I/O plugin facility to arrange socket I/O calls to work with libevent
- Study the performance of servers written in Chapel
- Add a helper class to make it easier to implement a server and demonstrate a simple HTTP server
CHAPEL 2.0
Background and This Effort

Background:
• Over the past few years, we have been working toward a forthcoming Chapel 2.0 release
• Intent: stop making backward-breaking changes to core language and library features
  – thereafter, use semantic versioning to reflect if/when such changes are made

This Effort:
• Major language-related changes largely wound down as of 1.24
• Remaining efforts tackled for 1.25 (described in “Language” deck):
  – deprecating old ‘proc’-based operator declarations
  – improving support for resizable arrays and collections
  – addressing feature requests and bugs
• Stretch goals:
  – interfaces
  – first-class functions / closures
• Remaining efforts lie predominantly in stabilizing the standard libraries
STANDARD LIBRARY STABILIZATION: BACKGROUND

- The effort to release Chapel 2.0 is currently focused on standard library stabilization
  - *Stabilization:* The interface is unlikely to change soon; and if so, backward-compatibility will be ensured

- For the Chapel 1.24 release, we started reviewing standard libraries
  - Bi-weekly meetings
  - One team member leads a review discussion on the module interface, scrutinizing...
    - the name of the module itself
    - names of public types, enums, global variables, constants, ...
    - names of public procedures, arguments
    - behaviors / definitions of all public symbols
STANDARD LIBRARY STABILIZATION

This Effort: Our Procedure

- During the Chapel 1.25 release, we kept the same pace for reviews but also started follow-up meetings:
  - We meet every week
    - on even weeks, we review a new module as usual
    - on odd weeks, we follow up on a previously reviewed module
  - The team member who oversaw the initial review leads a follow-up discussion, to...
    - go over remaining open issues
    - find owners for making the changes that are decided but not implemented
  - We also created a sub-team to review the IO module
    - the IO interface is much bigger compared to other modules
    - it is also one of the key modules in the standard library
      - and it has a list of known issues
    - IO sub-team members meet regularly and call full-team meetings when parts of the interface are ready for discussion
**STANDARD LIBRARY STABILIZATION**

This Effort: In Numbers

- As of the Chapel 1.24 release, we had:
  - Reviewed 7 standard libraries

- During this release cycle, we:
  - Reviewed 16 additional standard libraries
  - Re-reviewed 7 standard libraries

- As a result, our current status is:
  - 23 modules reviewed
  - 2 modules stabilized:
    - Path, Builtins
  - 5 modules that are close to being stabilized:
    - CPtr, SysCTypes, Sys, Regex, Time
  - 7 modules that we’ve decided not to stabilize before Chapel 2.0:
    - CommDiagnostics, Memory, BitOps, GMP, DynamicIters, VectorizingIterator, Help
  - 11 modules that still need review:
    - ChapelEnv, Types, Version, SysError, Errors, FileSystem, DateTime, Math, Set, Heap, Random
## STANDARD LIBRARY STABILIZATION

This Effort: Overview

| Builtin | Chapel Env | Heap | List | Map | Set | CommDiag | Memory | FileSystem | IO | Path | Reflection | Types | BigInteger | BitOps | GMP | Math | Random | Barriers | Dynamicities | VectorizingIterator | CPtr | SysCTypes | Spawn | Sys | SysBasic | SysError | DateTime | Help | Regexp | Time | Version | String | Bytes | Ranges | Domains | Arrays | Shared | Owned | Errors |
|---------|------------|------|------|-----|-----|----------|--------|------------|-----|-------|------------|-------|-------------|--------|-----|------|--------|---------|-------------|-------------------|-------|----------|-------|-------|--------|---------|--------|--------|-------|--------|--------|--------|--------|--------|
| 1.24    |            |      |      |     |     |          |        |            |     |       |            |       |             |        |     |      |        |         |              |                   |       |          |       |       |        |         |        |        |       |        |        |        |
| 1.25    | **✓**      |      |      |     |     |          |        |            |     |       | **✓**      |       |             |        |     |      |        |         |              |                   | **✓** | **✓**    |       |       |        |         |        |        |       |        |        |        |        |        |        |

- **Stable**
- **Progress**
- **Review Started**
- **Not 2.0**

* - VectorizingIterator is likely to be deprecated in 1.26—it's functionality is superseded by 'foreach'
STANDARD LIBRARY STABILIZATION

- Path
- List
- Map
- Builtins
- Sys
- CPtr & SysCTypes & SysBasic
- Time
- Range
- Domain
- Array
- Shared & Owned
- String & Bytes
- Regex
- Reflection
- Barriers
- Spawn
- BigInteger
- IO
**PATH MODULE**

**Background:**
- The Path module contains mostly string-based operations on paths
- First reviewed during the 1.24 release cycle

**Actions Taken:**
- Overhauled interactions between this module and the ‘file’ type

**Other Comments:**
- As of 1.25, we consider the Path module to be stable

<table>
<thead>
<tr>
<th>1.24</th>
<th>1.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.getParentName()</td>
<td>N/A – Deprecated</td>
</tr>
<tr>
<td>file.relPath()</td>
<td>relPath(f: file)</td>
</tr>
<tr>
<td>file.realPath()</td>
<td>realPath(f: file)</td>
</tr>
<tr>
<td>file.absPath()</td>
<td>absPath(f: file)</td>
</tr>
</tbody>
</table>
LIST MODULE

Background:
• The List module provides the ‘list’ type, a key data structure
  – Lightweight data structure that supports fast appends and iterations
  –Appending and indexing is O(1)
  –Insertions and removals from the front/middle are O(n)

Open Discussions:
• What are the expectations from this data structure? (#18095)
• How should we control parallel-safety for list? (#18097)
• Reduce the number of ways by which list elements can be modified (#18101)
• Should we deprecate ‘list.sort’? (#18100)
• Retire ‘list.extend’ in favor of new list.append overloads (#18098) – generally supported
• Rename ‘list.indexOf’ as ‘list.find’ and make it not halt on empty lists (#18099) – generally supported
Background:

- Provides a ‘map’ data type, allowing for key/value data storage

Open Discussions:

- Design proposals for serial/parallel/distributed maps ([#18494](#18494))
  - Separation would be useful as a simplification; the current version of Map puts both serial and parallel in a single type
  - Separate types would allow each to be cleaner and more complete
  - However, having a separate type for serial and parallel maps would detract from Chapel’s scalability
  - Create mock proposal to get an idea of how this idea might work

- Deprecate operators ([#18493](#18493))
  - Removing old operator methods (\(=, ==, !=, +, +=, |, |=, &+, &=, -=, ^=, ^=\))
  - Added by default when the module was created
  - Unneeded and unused, will be removed unless requested by users
**BUILTINS MODULE**

**Background:**
- ‘Builtins’ was automatically included and contained these functions:
- It was initially added to close a memory leak
- The name is arguably confusing since it does not include all built-in features

**Actions Taken:**
- Resolved remaining questions about module structure and removed the ‘Builtins’ module
- Decided to move internal type docs to spec ([#18027](https://example.com))
- Created a new automatically-included standard module ‘Errors’ containing:
  - the contents of ‘Builtins’; the contents of ‘ChapelError’; ‘halt’ and ‘warning’ from ‘ChapelIO’
- Decided upon having two flavors of ‘assert’: one disabled with ‘--fast’ and one that is always on

**Open Discussions:**
- How to select each flavor of assert (to be discussed in ‘Errors’ now that they live there—[#18024](https://example.com))
  - Could use ‘assert’ and ‘debugAssert’, where ‘debugAssert’ does nothing with ‘--fast’ (similar to Rust)
  - Could use ‘assert’ and ‘releaseAssert’, where ‘assert’ does nothing with ‘--fast’ (‘assert’ similar to C)
  - Could use an additional argument, e.g. ‘assert(permitOpt=true, ...)’
**SYS MODULE**

**Background:**
- The Sys module defines the operating system interface: types, constants, and functions

**Actions Taken:**
- Agreed to a hierarchical organization for the module:
  ```chapel
def module OS // parent module for all specific OS interfaces
  module POSIX // what is currently in Sys, with some symbol renamings (see below)
  module Linux  // things in Linux but not POSIX
  module MacOS  // things in macOS but not POSIX
  ```
- Agreed that the module should follow the C interface as much as possible, as opposed to being Chapel-tastic
  - other modules may use this as a building block and provide Chapel-tastic interfaces to the same capabilities
- Agreed that the module will initially only define symbols that are currently used or will be soon
  - others will be added as they are needed

**Open Discussions:**
- None; we think we're done with the design of this one! (Implementation tasks remain)
- See [#18448](#18448) for the detailed proposal
**CPTR & SYSCTYPES & SYSBASIC MODULES**

**Background:**
- These modules primarily provide access to C types, like ‘c_int’, ‘c_ptr(c_int)’, etc. for interoperability purposes

**Decisions Made:**
- Agreed to bring all such C types into a single module
- Agreed that ‘[s]size_t’ should become ‘c_[s]size_t’ (#18012)
- Generally agreed that the ‘c_’ prefix is an acceptable departure from typical Chapel style since it’s reflecting C
- Need to stop revealing ‘c_ptr’ to the user as a ‘class’ in the documentation

**Open Discussions (others also exist):**
- What should we name the module that holds these C types? (‘CTypes’, ‘CInterop’, ‘C’, ...?) (#18013)
- Should ‘c_void_ptr’ become simply ‘c_ptr(void)’? Or does it deserve a special identifier? (#18011)
- Questions about ‘c_nil’, ‘is_c_nil’, ‘c_ptrTo()’, and allocation routines (#18014, #18015, #18016, #18017)

**Other Comments:**
- Much of the work here feels like relatively low-hanging fruit
TIME MODULE

Background:
- This is a module from the project’s early days that has felt stale and in need of refreshing

Actions Taken / Decisions Made:
- Added timeSinceEpoch() to ‘DateTime’, necessary to deprecate getCurrentTime() (#17395, #16773, #1442)
- Planning to deprecate ‘Day’, ‘getCurrent*’ features in favor of ‘DateTime’ (#17922)
- Agreed to rename ‘Timer’ to ‘stopwatch’
- Agreed to extend the interface for ‘stopwatch’ to include additional routines (#16395)

Open Discussions (others also exist):
- Where should ‘sleep()’ live? Or should we just retire ‘sleep()’ as a bad practice? (#18467)

Other Comments:
- This effort is largely waiting on a review of the ‘DateTime’ module to establish a unified plan
  - Specifically, we want to avoid deprecating a given symbol multiple times
RANGE MODULE

Background:
- This is a built-in module whose library-like features are being reviewed (methods, functions on ranges)

Actions Taken:
- Updated definition of `.size` and deprecated `identQ` — see “Language Changes” deck for details
- Agreed that ranges should be immutable values (e.g., `myRange.low = 10;` will continue to be illegal)
- Agreed to retain range slicing as intersection and `.contains` for subset queries

Open Discussions (others also exist):
- Redefine `.low` / `.high` for strided ranges to return the aligned low and high bounds? (#17130)
  - generally well-received, but we need to see how impactful it is
- Rename / redefine the ‘enum’ defining a range’s boundedness and the ‘bool’ defining its stridability? (#17131)
  - current names are clunky, old-school, unpopular
- Make ‘range’ a generic type by default (e.g., no default stridability, boundedness, and/or ‘idxType’?) (#18215)

Other Comments:
- This remains a high-priority module for action, due to its ties to the language and heavy use
DOMAIN MODULE

Background:
• This is another built-in module whose library-like features are being reviewed (methods, functions on domains)

Actions Taken:
• Redefining `.size` to return an ‘int’ — see “Language Changes” deck for details
• Agreed to continue disallowing `+` as translation on rectangular domains (e.g., `{0..5, 0..5} + (1, 1) == {1..6, 1..6}`)
• Agreed to replace `isSubsetO`, `isSupersetO` with `containsO` overloads that takes a domain
• Agreed to retain `isEmpty` rather than requiring a `.size == 0` check

Open Discussions (others also exist):
• Are we happy with `.dim(d)` and `.dimsO` as-is? ([#17916](#17916), [#17917](#17917))
• Rename `.dist` query? (which returns the domain map of a domain—its distribution or layout) ([#17908](#17908))
• Should we drop the parenthesis from `.targetLocalesO`? ([#18470](#18470))

Other Comments:
• This remains a high-priority module for action, due to its ties to the language and heavy use
ARRAY MODULE

Background:
- This is another built-in module whose library-like features are being reviewed
  - For this release, we reviewed methods on arrays
    ```
    A.isEmpty(); A.reverse(); ...
    ```

Actions Taken:
- Added ‘dimO’ and ‘dimsO’ methods to arrays
  - Previously, these were only available on domains
  - Languages without domains would put these methods on the array directly
- Reading/writing a multi-dimensional array in Chapel syntax style now produces an error
  ```
  var A:[1..2, 1..2] string = "hi"; writef("%ht\n", A);
  ```
  - Why: there is currently no syntax in Chapel for a multi-dimensional array literal

Open Discussions:
- Should we remove: ‘sortedO’, ‘reverseO’, ‘findO’, and ‘countO’ methods from the array type? (#18089)
Background:
- The Shared and Owned modules implement the ‘shared’ and ‘owned’ class memory management strategies.

Actions Taken:
- Did initial review of Shared and Owned modules.

Next Steps:
- Need clear descriptions for assignment between all combinations and the impact of nilable vs. non-nilable.
- Plan to create documentation table(s) for conversions between differently-managed instances.
- Discuss whether to make borrows explicit with a method call.
- Considering what it would take to implement something like a C++ ‘weak_ptr’ in Chapel.
STRING & BYTES MODULES

Background:
- Modules for manipulating ‘string’ and ‘bytes’ variables
  - ‘bytes’ is like ‘string’ but can store arbitrary data

Open Discussions:
  - Rename ‘needle’ and ‘region’ arguments
- Should the type ‘codepointIndex’ be deprecated? (#18265)
  - Use ‘int’ in most cases or ‘byteIndex’ for maximum performance
- Deprecation of ‘c_str’ method in favor of a common ‘c_ptrTo’ or similar (#18266)
  - Extension of existing issue about C interoperability (#18016)
REGX MODULE

Background:
- The Regexp module provides regular expressions based on Google’s RE2 library

Actions Taken:
- Standard module Regexp renamed to Regex
- Replaced environment variable ‘CHPL_REGEXP’ with ‘CHPL_RE2’
  - In 1.24: ‘CHPL_REGEXP=re2’ or ‘CHPL_REGEXP=none’
  - In 1.25: ‘CHPL_RE2=bundled’ or ‘CHPL_RE2=none’
- Consistent with other environment variables that control third party builds: ‘CHPL_LLVM’, ‘CHPL_HWLOC’, ‘CHPL_GMP’ ...

Open Discussions:
- Should tertiary methods on string/bytes be defined by the Regex module? (#17226)
  - Keep the existing ‘search’, ‘match’, ‘split’, and ‘matches’ methods for strings/bytes or remove them?
- Which method to use for compiling regular expressions? (#17187)
  - Today we use ‘compile(“/a/”)’, should we instead use ‘new regex(“/a/”)’ or ‘regex.compile(“/a/”)’?
- Should Chapel support regex literals? (#17219)
  - Might look like r“alb$” or ‘qr/.../’
Background: Reflection is a module that lets you:
- Query properties about an aggregate type, such as a field’s name or ordinal position
- Get a reference to the field of an instance given a string or ordinal position
- Check if a function or method call could be resolved
- Query properties about the current module (line number, name of current function, etc.)

Open Discussions:
- Should most Reflection module functions be methods instead? (#17984)
- Allow variables as arguments to Reflection module functions (#7650)
- Should Reflection module functions prefixed with ‘get’ drop the ‘get’? (#18006)
- Should the ‘canResolve’ family of functions be renamed? (#17986)
- Should ‘getRoutineName’ be renamed? (#17987)
- Support counting inherited fields with ‘numFields’ (#8736)
**BARRIERS MODULE**

**Background:**
- ‘Barriers’ module provides a general-purpose barrier
  - Initializer only accepts the number of tasks participating with no notion of locality, which limits scalability
- ‘AllLocalesBarriers’ module provides a global singleton barrier between all locales
  - Has good scalability, but only suitable for SPMD-style codes

**Open Discussions:**
- What is the right interface to create a scalable barrier between an arbitrary set of Locales?
  - Should there be different initializers for local and distributed barriers or distinct types?
- Should we move the barriers into a ‘Barrier’ or ‘Collectives’ module?
- Must the number of participants be known at barrier creation time, or can they be registered dynamically?
SPAWN MODULE

Background:

- Provides functionality to create and communicate with subprocesses
  - Send signals, communicate through pipes, wait for termination, etc.

```javascript
use Spawn;

// start a process to run 'ls t*' in the $CWD
var sub = spawn(['ls', 't*']);
sub.wait();
```

Actions Taken:

- Reviewed the Spawn module for changes needed to publicly visible interfaces
- Deprecated 'sublocale.exit_status' in favor of 'sublocale.exitCode'
- Decided that:
  - All CAPS Posix constant names should be renamed with camelCase and moved to 'Posix' module
  - Want to rename the module from 'Spawn' to 'Subprocess'
  - Want to rename remaining publicly visible symbols that have underscores to use camelCase
**BIGINTEGER MODULE**

**Background:**
- Provides ‘bigint’ type for storing very large integers, and many methods and functions that use them

**Actions Taken:**
- Decided module interface should be more independent of its underlying GMP implementation
- Renamed division, ‘powmG’, ‘sizeinbaseG’ methods and ‘Round’ enum to conform with other libraries
  - Also renamed some arguments when adjusting these methods
- Deprecated ‘sizeG’ method - returned “number of limbs”, a GMP implementation detail
- Added documentation for finalized methods and enum

**Open Discussions:**
- There are 27 small library stabilization issues remaining that are likely uncontentious
  - See the list of issues
  - And 9 non-breaking changes
IO MODULE
Background and Actions Taken

Background:
• The IO module handles reading and writing to files, as well as formatted IO
  – ‘write0’, ‘writeln0’ and ‘writef0’ are provided by default, all other IO functions are defined in the IO module
• Implements ‘file’ and ‘channel’ types
• This module is very large, ~7300 lines
• The IO module has quite a few known API design issues (#7954)

Actions Taken:
• Created a sub-team to work towards IO module stabilization
• Examined formatted IO submodule at length
• Developed 5 proposals for improvement (see next slide)
Proposals

- Rename I/O ‘channel’ type to ‘reader’ and ‘writer’ (#18112)
  - because ‘reader’/‘writer’ are more common terminology in other languages
  - and we’d like to have ‘channel’ free for the Go-style channels described earlier in this deck
- Deprecate the I/O style feature (#18501)
  - because it is little-used and has many design problems
- Deprecate binary format strings including endianness specifiers (#18503)
  - to make ‘readf’/‘writef’ more focused on text formatting
  - replace these with individual method calls—e.g., ‘writer.writeBigEndian(1)’
- Add an extensible Encoder/Decoder mechanism (#18499)
- Deprecate ‘j’ and ‘h’ format string specifiers in favor of Encoder/Decoder
  - ‘%jt’ today specifies to read or write a type in JSON format
  - ‘%ht’ today specifies to read or write a type in Chapel syntax
IO MODULE
Open Discussion and Status

Open Discussions:

• Should we prefer ‘read(type t)’ to ‘read(ref value)’? (#18496)
• What should the relationship be between ‘channel’ and Encoder/Decoder? (#18504)
• Should there be more consistency between use of ‘+’ and ‘-’ in format strings? (#18495)
• Meaning of ‘%.6’ varies depending on what type it is applied to (#18497)
• Format string numeric specifiers for complex numbers impact the number differently (#18498)

Next Steps:

• Reach a decision on proposals and on the open discussion items above
• Implement the Encoder/Decoder design
• Review ‘file’ and ‘channel’ interfaces
• Improve the formatted IO documentation
STANDARD LIBRARY STABILIZATION

Next Steps

- Continue with our current process:
  - Start reviewing the remaining 11 modules
  - Revisit modules that were first examined in previous releases
  - Continue resolving issues discussed in reviews, e.g.
    - Design proposal for serial/parallel/distributed maps (#18494)
    - Deprecation of ‘c_str’ method in favor of a common ‘c_ptrTo’ or similar (#18266)

- Determine whether any additional modules can be put into the “stabilize after Chapel 2.0” camp
  - e.g., does ‘Heap’ require stabilization for Chapel 2.0?

- Develop a means of documenting the stability of a module (or language feature)
CHAPEL 2.0
Status and Next Steps

Status:
• language stabilization continues to be in increasingly good shape
• module review process is proceeding apace

Next Steps:
• continue to focus predominantly on the module review process and follow-ups
• while also seeing remaining language-related efforts through to completion
  – and being responsive to new user issues around the language
THANK YOU

https://chapel-lang.org
@ChapelLanguage