Hewlett Packard
Enterprise

Chapel 2.1 / 2.2 Release Notes: Implementation, Tools, and Portability

Chapel Team

June 27, 2024 / September 26, 2024

Outline

New Incremental Resolver

Incremental Resolver
Background and This Effort

Background:

- *Dyno* is an ongoing effort to address problems with the Chapel compiler
- Focused on improving:
  - Speed
  - Error Messages
  - Compiler architecture and program representation
  - Compiler development
- This effort led to the development of the compiler frontend library
- A recent focus is creating a new incremental type and call resolver for the compiler frontend library
- This new resolver can be used from Visual Studio Code as an experimental feature

This Effort:

- Significantly improved the incremental resolver
- Improved the stability of using resolver-based features in Visual Studio Code

New Incremental Resolver
Status and Next Steps

Status:

- Can now resolve “Hello World”
  - Not as trivial as it sounds due to the amount of internal and standard module code involved
- Can now resolve about 65% of the examples from the language specification

Next Steps:

- Complete the new incremental resolver
- Use the incremental resolver in the production compiler
- Continue working towards separate and incremental compilation

Language Tooling

Language Tooling
Background and This Effort

Background:

- Several editor support tools have been available to developers since 2.0:
  - The Chapel Language Server (CLS) provides go-to-definition, hover information, and much more
  - The 'chplcheck' linter detects common stylistic problems in user code
  - A Chapel extension integrates these language tools with Visual Studio Code
  - The tools are powered by the *Dyno* compiler frontend library and 'chapel-py', a Python interface to it

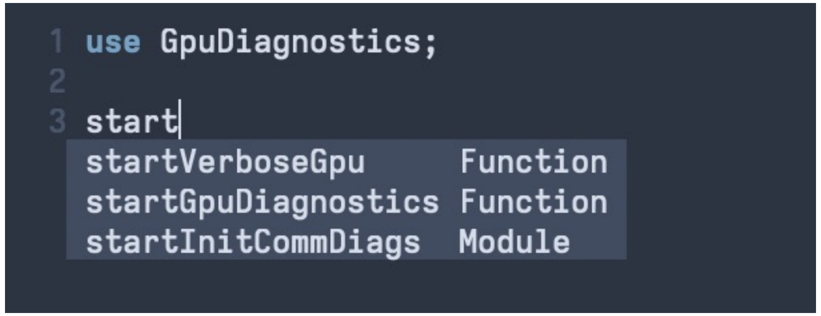
This Effort:

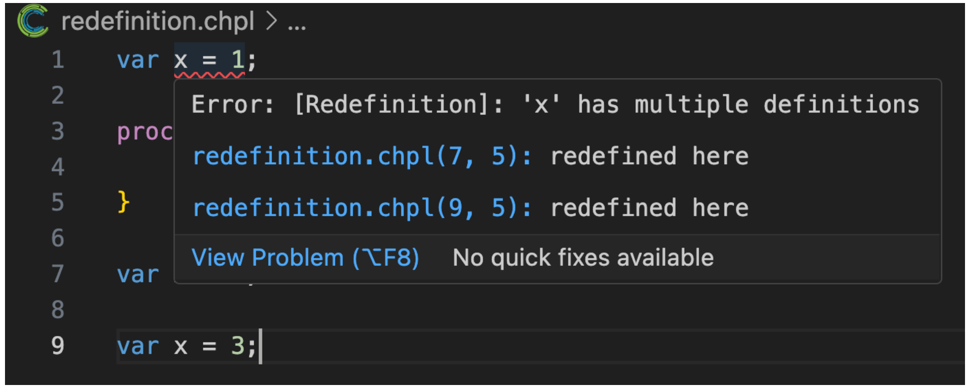
- Extended CLS with additional functionality, including autocompletion
- Improved the ergonomics of 'chplcheck', including optionally bundling it with CLS
- Added additional rules for 'chplcheck'
- Made it easier to build and install Python-based tools

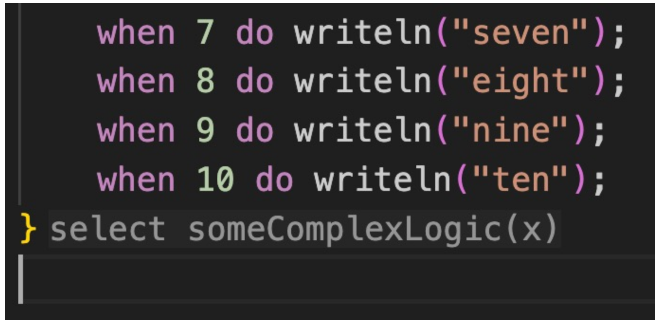
Language Tooling

This Effort: CLS Improvements

- Added support for autocompletion from globally visible scopes
  - Includes ‘use’d modules and keeps track of renaming and transitive uses
  - Skips explicitly undocumented symbols and built-in functions unless in internal module code
- Started displaying extended error messages as part of editor errors
  - The detailed error messages can provide additional locations and suggestions
- Added more inlay types
  - End markers for ‘select’ and ‘when’
  - Function argument name for complex literals







Language Tooling

This Effort: ‘chplcheck’ linter

Ergonomics:

- ‘chplcheck’ can now be optionally executed as part of CLS
  - Improves support on systems where only one language server can run at a time
- Various Python API improvements to make it easier to write custom rules that can be auto-fixed
  - Built-in rules benefit from this, getting auto-fixes

New Linter Rules:

- Unneeded pattern matching in declarations (‘(_, _)’ should be ‘_’ since both components are ignored)
- Unnecessary control flow parentheses (‘if (expr)’ should be ‘if expr’)
- Complex literal ordering (‘2i + 1’ should be ‘1 + 2i’)
- Incorrect indentation (pictured)

```
proc f3() {
writeln("hi");        Lint: rule [IncorrectIndentation] violated
}

proc f4() {
  writeln("hi");
    writeln("hi");        Lint: rule [IncorrectIndentation] violated
}
```

Language Tooling

This Effort: Building and Packaging Tooling

Packaging:

- ‘chpl-language-server’ and ‘chplcheck’ are now bundled in Homebrew installations

Build Improvements:

- Recently, ‘chapel-py’ was also used as part of Arkouda’s message registration revamp
  - This required building it on more systems and in more situations
- ‘chapel-py’ can now be built without ‘CHPL_HOME’ set
- Python-based tools issue better errors for version mismatches between ‘chapel-py’ and system Python
  - ‘chapel-py’ uses the CPython API and must be rebuilt against updated Python interpreters
  - Previously, errors caused by not rebuilding were very difficult to interpret
- Python bindings now use the same C/C++ compiler as Chapel to build, avoiding portability issues

Language Tooling
Next Steps

- Continue to expand CLS and ‘chplcheck’ rules and functionality
  - For ‘chplcheck’, a good approach seems to be implementing rules based on user experiences
  - For CLS, most additional functionality will come with the *Dyno* resolver coming online
  - Improving the location information captured by the parser can greatly help with auto-fixes and error messages

- Bring editor integration to more users
  - Publishing VSCode extension to OpenVSX can make it usable in other derived editors like VSCodium

- Improve graphical execution and debugging in VSCode extension

GPU-Based Reductions

GPU-Based Reductions

Background

- The GPU module provided preliminary support for GPU-based reductions using subroutines, e.g.:

```
proc gpuSumReduce(arr: []);
```

- This approach helped us address user requests quickly, however:
  - Chapel supports reductions at the language-level with 'reduce' expressions and 'reduce' intents
  - Standalone functions are significantly more limited in capability:
    - Arbitrary expressions like A+B cannot be reduced with gpuSumReduce , which only supports single arrays
    - TeaLeaf could be implemented more efficiently with 'reduce' intents

GPU-Based Reductions
This Effort

- Introduced GPU support for most common 'reduce' operations

- The compiler and the runtime use CUB (NVIDIA) and hipCUB (AMD) to implement a 2-level reduction
  - **Level 1:** among threads within block; this happens inside the compiler-generated kernel
  - **Level 2:** among blocks across grid; this happens right after the kernel finishes executing

GPU-Based Reductions

Status and Next Steps

Status: '+', 'min' and 'max' reduce expressions and intents are now supported on GPU:

```
on here.gpus[0] {
  var Arr: [1..n] real = 1;     // create an array of 'n' reals, and set elements to 1 on GPU
  writeln(+ reduce Arr);        // compute the sum of the elements on GPU (this will launch a kernel)

  var sum: real;
  forall elem in Arr with (+ reduce sum) {    // this will launch a GPU kernel
    elem = sin(elem);       // compute each element of the array
    sum += elem;            // also, sum their values within the same kernel
  }
  writeln(sum);
}
```

Next steps: Support user-defined reductions to cover other reduction kinds

GPU Eligibility Attributes

GPU Attribute Improvements

Background

- Chapel's GPU support makes use of attributes applied to loops to control their execution
- The '@assertOnGpu' attribute is used to ensure code executes as a GPU kernel

```
@assertOnGpu
foreach i in 1..128 do ineligibleFunction();    // compilation error: call makes loop unable to run on GPU
```

- At compile-time, this attribute can report why code is not GPU-eligible
- At execution-time, the attribute halts if the code isn't executing on the GPU despite its eligibility

- No way to assert that code is eligible for the GPU without requiring it to run there
  - Causes problems for code intended to run on both GPU and CPU

GPU Attribute Improvements
This Effort, Impact, and Next Steps

This Effort:
- Added a new ‘@gpu.assertEligible’ attribute which is like ‘@assertOnGpu’, but without a runtime check
- Ensured GPU attributes do not cause errors when not using the GPU locale model

Impact:
- Patterns of writing GPU-and-CPU code can be greatly simplified
- The same code can be used for both CPU and GPU, while still ensuring it remains GPU-eligible

```
// Before
if onGpu then
  @assertOnGpu foreach i in 1..128 { … }
else
  foreach i in 1..128 { … }
```

```
// After
@gpu.assertEligible foreach i in 1..128 { … }
```

Next Steps:
- Consider deprecating ‘@assertOnGpu’ in favor of ‘@gpu.assertEligible’

GPU Attributes on Variables

GPU Attributes on Variables
Background

- To apply GPU attributes to other GPU-eligible constructs, attributes can be attached to variables
- For example, the attribute below applies to the loop expression:

```
@assertOnGpu
var A = foreach i in 1..128 do i;
```

- This worked for explicit loop expressions but not for promoted expressions
- The implementation had some difficulty with multiple loop expressions in a single variable

GPU Attributes on Variables
This Effort

- Allowed applying GPU attributes to promoted expressions in variable declarations
- Allowed multiple expression-level GPU loops in a single variable declaration
- Everything in the variable declaration is checked for GPU-eligibility

- The following program now works as expected:

```
@assertOnGpu
@gpu.blockSize(128)
var B = (foreach i in 1..256 do i*i) +   // the foreach loop expression is checked for eligibility
        (2 * A + 10);                    // these promoted function calls are also checked
```

Other GPU Updates

Other GPU Updates

- Added support for ROCm 6
  - Requires using the bundled LLVM to work around upstream LLVM issues

- Added GPU support to co-locales
  - GPUs are partitioned evenly among co-locales
  - Each co-locale uses the GPU(s) closest to it

```
> ./hello -nl 1x4      # on an 8-GPU node, assigns 2 GPUs to each co-locale
```

- Fixed behavior for 'ref' intent when a scalar is modified in a loop when it is executed as a kernel, e.g.:

```
var x = 1;
on here.gpus[0] do
  foreach i in 0..0 with (ref x) do x = 100;
writeln(x);  // Now prints 100; in 2.1 printed 1
```

AWS Portability Improvements

AWS Portability Improvements
Background

- Running on AWS (Amazon Web Services) with EFA (Elastic Fabric Adapter) had some limitations:
  - Limited to 96 GB of memory on each node
  - Programs using non-blocking operations could occasionally hang
  - Required building Chapel from source

- The 96 GB memory limit comes from a fundamental hardware limitation of EFA:
  - The number of memory pages that can registered per process is limited
  - However, more than 96 GB can be allocated using Transparent Hugepages (THP)

AWS Portability Improvements
This Effort and Impact

This Effort:

- Added Transparent Hugepages (THP) support to register more memory
  - The Chapel runtime provides an explicit hint to the OS to use hugepages
  - THP support can be used with other networks that use libfabric
- Fixed hangs with non-blocking operations
  - Caused by incorrect injection logic for EFA
  - Impacted the scale at which larger applications could run, especially with atomics
- Expanded package support for AWS
  - Getting Chapel installed on a cloud cluster only takes 2 commands

Impact:

- More than 96GB can now be registered
- Using Chapel at scale on AWS is easier to do and more robust
- Large, memory intensive applications can be used more freely with EFA

Other Tools and Implementation Improvements

Other Tools and Implementation Improvements

For a more complete list of tool and implementation changes and improvements in the 2.1 and 2.2 releases, refer to the following sections in the CHANGES.md file:

- GPU Computing
- Tool Improvements
- Documentation Improvements for Tools
- Configuration / Build Changes
- Portability / Platform-specific Improvements
- Compiler Improvements
- Compiler Flags
- Bug Fixes for GPU Computing
- Bug Fixes for Tools
- Bug Fixes for Build Issues
- Bug Fixes for the Runtime

Thank you

https://chapel-lang.org
@ChapelLanguage