THE SCOUT PROGRAMMING LANGUAGE

Language Reference

Los Alamos NATIONAL LABORATORY EST. 1943

Version 1.0 – March 2015

CONTENTS

LIST OF FIGURES

List of Tables

Source Code Listings

PREFACE

scout *verb* [intrans.] – *to explore or examine so as to gather information.*

Welcome

The goal of this manual is to provide you with a quick reference to the syntax, semantics and features of the experimental Scout language. We assume that the reader is an experienced programmer with a basic understanding of parallel computing. Although we do our best to keep the language documentation up to date with the feature set supported by the open-source versions of Scout, readers are encouraged to look over the release notes provided with each version for important details.

About Source Code Listings

The source code listings in this manual uses both font changes and *syntax aware* coloring that helps the reader to identify parts of the language. Table 0.1 provides a key to the font and colors using in the listings.

Support

If you have questions, or encounter problems, please feel free to send an email to the Scout support team via email: scout-support@lanl.gov.

TYPE	COLOR	FONT
KEYWORDS	`keyword`	`keyword`
BUILT-INS	`built-ins`	`built-in`
COMMENTS	*comments*	*comments*
STRINGS	*"string"*	*"string"*

Table 0.1: Source code listing colors and fonts.

Open Source Effort

This version of Scout is an open-source software effort established by Los Alamos National Laboratory's Applied Computer Science Group. The source code is available here:

https://github.com/losalamos/scout.

If you are interested in learning more about the LANL team working on Scout, please visit our LANL web site here:

http://progmodels.lanl.gov/scout.

1 INTRODUCTION

Scout is an experimental programming language that combines sequential and parallel general-purpose constructs with data analysis and visualization-centric features. The language is an extension to the C programming language, but it has also been influenced by many other languages and is fundamentally a higher-level language than C. In this manual we assume the reader is familiar with parallel programming and the basics of C. In this chapter we give a brief introduction to Scout's main features. The following chapters will provide more details about Scout's abstract computational data structures and the parallel, data analysis, visualization and plotting constructs.

1.1 Scout and the C Programming Language

Scout extends the C programming language to support a new set of abstractions and domain-specific constructs. Some of the main domain-specific constructs in Scout are explicit mesh declarations, instantiations and parallel computation over the various components of the mesh (e.g. cells, vertices, edges) or over array elements. Scout builds upon the fundamental types of C to include higher-level abstractions – such as computational meshes and the associated fields stored on a mesh. From this perspective, Scout is a higher-level language than C.

Since Scout is based on the open-source Clang and LLVM Compiler Infrastructure, all of the C language features are available. Scout extends the fundamental data types available in C to include vector types of two, three, and four components. Scout's syntax for vector types follows closely with those used by the OpenGL Shading Language and NVIDIA's C for CUDA. Section 2 below covers these topics in more detail.

1.2 Getting Started

The best way to learn any programming language is to get your hands dirty. The classic hello world program does not need to rewritten for Scout, since Scout fully supports C. Therefore, as an introductory program to Scout, Listing 1.1 illustrates a simple one-dimensional mesh transformation. It uses the `mesh` abstraction, the mesh's associated fields and the parallel `forall` construct.

```
uniform mesh MyMesh{
  cells:
    float a;
    float b;
    float sum;
};

int main(int argc, char** argv){

  MyMesh m[512];

  // initialization of a and b not shown

  forall cells c in m {
    sum = a + b;
  }

  return 0;
}
```

Listing 1.1: Simple mesh example.

1.3 Compiling a Scout Program

Scout's compiler follows the characteristics of traditional (Unix) command line interfaces. In the following example, the simple mesh program from Listing 1.1 is compiled and executed from the command prompt:

```
$ scc simplemesh.sc          compiler produces default executable "a.out"...
$ ./a.out
$ _
```

The previous compilation of the Scout program creates an executable that is targeted to run sequentially on a CPU.

Scout can be enabled to run on an NVIDIA GPU via the `-gpu` option to `scc`.

See the output to `scc -help` or Appendix 4 for more information on the available command line options.

2 DATA TYPES

Those familiar with programming in C should find Scout's syntax and fundamental data types to be very familiar. In this section we quickly review Scout's additional composite data types.

2.1 Vectors

As briefly mentioned in Section 1.1, Scout provides support for two-, three-, and four-component vector types. The "base" type for vectors includes signed and unsigned values of each of the fundamental types (`bool`, `char`, `short`, `int`, `long`, `float`, `double`). The syntax for defining a vector uses these types followed by the number of components in the vector. Listing 2.1 shows some example vector declarations and initializations.

2.2 Meshes

Computational science data structures are frequently based on the concept of a mesh. Scout directly follows this philosophy by introducing mesh-centric data types. Scout supports the `uniform` mesh type.

In addition, the language provides constructs for defining the values stored on the various locations of the mesh (e.g. cell centers, vertices, edges and faces). Figure 2.1 illustrates the layout of a mesh. Figure 2.2 shows the field placement options within a mesh.

```
// Vector declarations
float2  u; // two-component single-precision vector.
int3    v; // three-component integer vector.
double4 w; // four-component double-precision vector.

// Vector declaration and initialization
float3 foo = (float3){1.0f, 2.0f, 3.0f};

// Another way to initialize a vector
float4 bar;
bar.x = 1.0f;
bar.y = 2.0f;
bar.z = 3.0f;
bar.w = 4.0f;
```

Listing 2.1: Vector declarations and initialization.

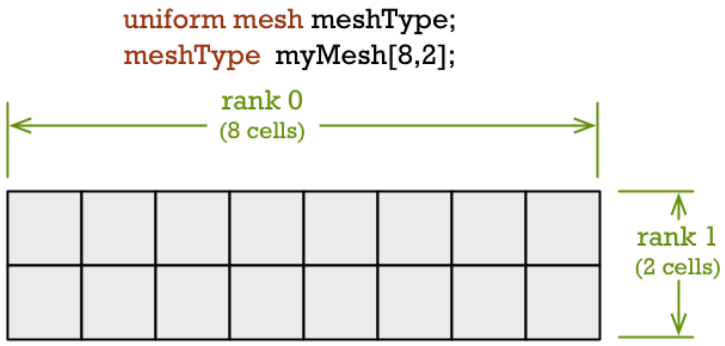

Figure 2.1: Example of a uniform mesh definition.

Here is a code example showing a two-dimensional mesh and a mesh whose dimensions are defined by an input file. Listing 2.2 shows some example mesh declarations. Note that is is not yet implemented to read meshes from input files.

In addition, meshes dimensions can be specified using variables. Listing 2.3 shows this.

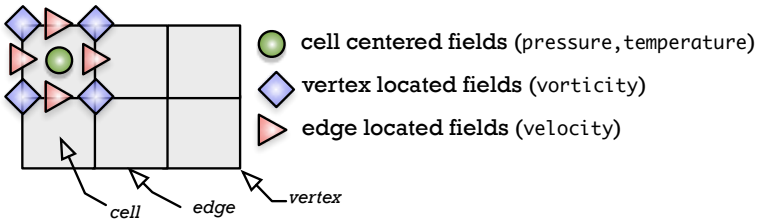

Figure 2.2: Field placement options within a mesh.

```
// Two-dimensional uniform mesh with values stored at cell
// centers and cell verticies.
uniform mesh MeshType {
  cells:    float temperature;
            float presure;
  vertices: float3 vorticity;
  edges:    float  velocity;
};

MeshType myMesh[512,512];
```

Listing 2.2: Mesh declarations.

```
  int dim = 4;

  uniform mesh HeatMeshType{
  cells:
    float t1, t2;
  };

  HeatMeshType heat_mesh[dim];
```

Listing 2.3: Mesh declarations using variable.

3 PARALLEL CONSTRUCTS

3.1 Data-parallel Operations

`forall` Loops for Meshes

Building upon the mesh data types, Scout provides support for parallel computations over the elements/attributes of the mesh (cells, vertices, etc.). The majority of these constructs use an explicit parallel form that is mixed with the main body of the code that is, by definition, executed sequentially.

Listing 3.1 shows an example parallel `forall` construct. In this case we are looping over all the cells of the mesh `myMesh` and setting the values of the cell attributes (`a` and `b`) to 0.0. Note that we can use the specified cell placeholder "`c`" to directly access the attributes of the currently active cell using a C-like structure member access notation. If there are no clashes within scope, the use of the of explicit cell deferencing may be dropped within the body of the loop (e.g. `c.a = 0.0;` can be replaced with `a = 0.0f;`).

Note that the code in the body of a `forall` construct must follow some constraints. A field inside a `forall` construct body may only be read-only or write-only code that does not satisfy this condition is rejected by the compiler. This limitation significantly reduces the complexity of analyzing data dependencies between parallel constructs. Another constraint is that if targeting the GPU, instructions for printing within the `forall` construct are not allowed. Lastly, variables declared outside of the `forall` construct body cannot be assigned to within the `forall` construct body.

```
  uniform mesh MeshType {
    cells: float a, b;
  };
  MeshType myMesh[16];

  forall cells c in myMesh { // 'c' is the active cell.
    c.a = c.b = 0.0f;
}
```

Listing 3.1: A `forall` loop construct.

Additional levels of parallelism can be introduced by nesting parallel constructs. Specifically, the parallel operations over the cells of a mesh can be combined with with a set of operations over the components of each individual cell. Listing 3.2 shows such a nesting.

```
forall cells c in myMesh { // 'c' is the active cell.
  forall vertices v in c { // 'v' is the active vertex.
    c.a += v.a * v.a;
  }
  c.a = sqrt(c.a);
}
```

Listing 3.2: Nested `forall` loop construct over mesh components.

Note that up to two levels of `forall` nesting are supported. The complete listing of all two-level nested combinations is shown in Listing 3.3 .

The `position()` built-in function of a cell is automatically provided to contain the coordinates of the current cell being processed. Listing 3.4 shows how positions can be used.

The `width()`, `height()`, `depth()` and `rank()` built-in functions are provided to enable the user to obtain the width, height, depth or rank of the mesh within a `forall` or outside of one. When using the built-in functions outside of a `forall`, the mesh variable is given as an argument to the built-in function. When used

```
forall edges e in m {
  forall cells c in e {}}

forall edges e in m {
  forall vertices v in e {}}

forall faces f in m {
  forall cells c in f {}}

forall vertices v in m{
  forall cells c in v {}}

forall cells c in m {
  forall edges e in c {}}

forall cells c in m {
  forall faces f in c {}}

forall cells c in m {
 forall vertices v in c {}}
```

Listing 3.3: All supported two-level nested `forall` loop constructs over mesh components.

```
forall cells c in heat_mesh {
  if (c.position().x > 0 && c.position().x < 1023)
    t1 = 0.0f;
  else
    t1 = MAX_TEMP;
}
```

Listing 3.4: Accessing the position of cells within `forall` loop construct.

within a forall, the cell can be given as an argument or no argument need be specified. Listing 3.5 shows how the dimension built-in functions can be used.

Built-in functions can allow neighbors of a cell to be referenced. The `cshift()` (circular shift) built-in function is used to access neighboring cells in the mesh. For now this is implemented for uniform meshes. This is like F90 but we shift index values versus array duplication. The following is a 2D mesh. The `cshift` function takes as arguments the cell field, then the index offset in the x, y and z directions depending on how many dimensions in the mesh. Listing 3.6 shows how the `cshift` function can be used.

```
uniform mesh MyMesh {
 cells:
  float a;
};

MyMesh m[3,4];
assert(width(m) == 3 && "bad width(m)");
assert(height(m) == 4 && "bad height(m)");
assert(depth(m) == 0 && "bad depth(m)");
assert(rank(m) == 2 && "bad rank(m)");

forall cells c in m {
  assert(width() == 3 && "bad width()");
  assert(height() == 4 && "bad height()");
  assert(depth() == 0 && "bad depth()");
  assert(rank() == 2 && "bad rank()");
  assert(width(c) == 3 && "bad width(c)");
  assert(height(c) == 4 && "bad height(c)");
  assert(depth(c) == 0 && "bad depth(c)");
  assert(rank(c) == 2 && "bad rank(c)");
}

}
```

Listing 3.5: Accessing the dimensions of a mesh.

Stencil functions can be defined to access neighbors of a cell. The stencil function `sten()` is shown in Listing 3.7 along with how it can be used.

`forall` Loops for Arrays

The `forall` construct can also be used to do explicitly parallel computations on arrays. The syntax in the `forall` allows the user to specify [*first-element-index*:*array-size*:*step-size*]. An example of a `forall` array construct for a one-dimensional array is shown in Listing 3.8.

Arrays of up to three dimensions can be looped with the `forall` construct. The Listing 3.9 shows an example of a three-dimensional array.

```
forall cells c in heat_mesh {
  if (c.position().x > 0 && c.position().x < heat_mesh.width()-1 &&
      c.position().y > 0 && c.position().y < heat_mesh.height()-1) {

    float d2dx2 = cshift(c.t1, 1, 0) - 2.0f * c.t1
                + cshift(c.t1, -1, 0);
    d2dx2 /= dx * dx;

    float d2dy2 = cshift(c.t1, 0, 1) - 2.0f * c.t1
                + cshift(c.t1, 0, -1);
    d2dy2 /= dy * dy;

    t2 = (alpha * dt * (d2dx2 + d2dy2)) + c.t1;
  }
}
```

Listing 3.6: `forall` loop construct with use of cshift.

```
  stencil int sten(AMethType *c) {
   return cshift(c->a,-1) - 2*c->a + cshift(c->a,1);
  }

  ...

  forall cells c in m {
    c.b = sten(&c);  //
  }

  ...

}
```

Listing 3.7: Nested `forall` loop construct with stencil.

Meshes that have an array field can have nested `forall` constructs over the vertices, cells, edges or faces, and then an inner `forall` over the mesh field array elements. The Listing 3.10 shows an example of this nesting.

Meshes can be passed as a parameter to a function. They are passed as a pointer to a mesh. When used with Scout built-in functions they are dereferenced. The Listing 3.11 shows an example of this.

```
int a[100];

for(size_t i = 0; i < 100; ++i){
  a[i] = i;
}

forall i in [0:100:1]{
  a[i] = i * 2;
}
```

Listing 3.8: `forall` loop construct for one-dimensional array.

```
int a[5][5][5];

forall i,j,k in [0:2:1, 0:3:1, 0:4:1]{
  a[i][j][k] = i*100 + j*10 + k;
}
```

Listing 3.9: `forall` loop construct for three-dimensional array.

```
uniform mesh MyMesh {
 cells:
  float a[4];
};

MyMesh m[8];

forall cells c in m {
  forall i in [:4:] {
    a[i] = position().x + i;
  }
}
```

Listing 3.10: `forall` loop construct over mesh elements and mesh field array elements.

3.2 Queries

The `query` construct can be used to name a selection of mesh fields. The mesh fields are selected using `from`, `select`, and `where`. The resulting query can be used as a target for a `forall` construct. Listing 3.12 illustrates the use of the `query` construct and its supporting constructs.

```
uniform mesh MyMesh {
  cells:
    int val;
};

void func(MyMesh* mp)
{
  assert(rank(*mp) == 2 && "incorrect rank");

  forall cells c in *mp {
    printf("%d %d %d\n", width(), height(), rank());
    assert(width() == 2 && "incorrect width");
    assert(height() == 3 && "incorrect height");
    assert(rank() == 2 && "incorrect rank");

  }
}

int main(int argc, char *argv[])
{
  MyMesh m[2, 3];
  func(&m);

  return 0;
}
```

Listing 3.11: Meshes can be passed as a parameter to a function.

3.3 Tasks

The `task` construct is used to designate a function to be a Legion task. In this scenario, the Legion runtime system will launch the task and schedule asyhronously so that data dependencies between tasks are honored. This functionality is only available for functions that take meshes as arguments. Listing 3.13 illustrates the use of the `task` construct.

Tasks can also call other tasks. Listing 3.14 illustrates how you can create subtasks.

```
uniform mesh MyMesh {
 edges:
  float b;
};

MyMesh m[8];

...

query q =
  from edges e in m
  select e.b where
  e.b > 5.0;

forall edges e in q {
  e.b += 100;
}
```

Listing 3.12: Subsetting using `query` construct.

```
task void MyTask(MyMesh *m) {

  forall cells c in *m {
    a = 0;
    b = 1;
  }

  forall cells c in *m {
    a += b;
  }
}

int main(int argc, char** argv) {
  MyMesh m[512];

  MyTask(&m);
  ...
}
```

Listing 3.13: Specifying tasks using `task` construct.

```
task void MyTask2(MyMesh *m) {
  forall cells c in *m {
    a += b;
  }
}

task void MyTask(MyMesh *m) {

  forall cells c in *m {
    a = 0;
    b = 1;
  }

  MyTask2(m);
}

int main(int argc, char** argv) {
  MyMesh m[512];

  MyTask(&m);
  ...
}
```

Listing 3.14: Specifying subtasks.

4 VISUALIZATION CONSTRUCTS

A key feature of the Scout language is the incorporation of visualization and rendering operations directly within the syntax and semantics of the language as first-class constructs. The `window` is also a Scout construct. Windows are constructed so that images can be displayed on them. Currently 2D rendering is supported. Listing 4.1 shows a simple example.

```
// Example visualization construct for rendering mesh cells.
uniform mesh meshType {
  cells: float a, b;
};

meshType myMesh[16];
window win[1024,1024] {

renderall cells c in myMesh to win
{
    color = rgb(1.0, 0.0, 0.0);
  else
    color = rgb(0.0, 0.0, 1.0);
}
```

Listing 4.1: A `renderall` loop construct.

Note that when meshes are passed as parameters to functions, the `renderall` construct works via an indirection to a mesh pointer as well. Listing 4.2 shows a simple example.

```
void MyFunc(MyMesh *m) {
  window mywin[512,512];

  ...

  renderall vertices v in *m to mywin{
    color = hsva(b/16.0f*360.0f, 1.0, 1.0, 1.0);
  }
```

Listing 4.2: A `renderall` loop construct on mesh parameter.

APPENDIX A: scc COMMAND LINE OPTIONS

The most useful options will probably be `scc -sclegion` and `scc -gpu`. Typing `scc -help` gives the summary of all of the command line options as shown below. There are so many because scc inherits the same options that clang has.

```
OVERVIEW: clang LLVM compiler

USAGE: scc-3 [options] <inputs>

OPTIONS:
  -###                    Print (but do not run) the commands to run for this compilat
  --analyze               Run the static analyzer
  -arcmt-migrate-emit-errors
                          Emit ARC errors even if the migrator can fix them
  -arcmt-migrate-report-output <value>
                          Output path for the plist report
  -cxx-isystem <directory>
                          Add directory to the C++ SYSTEM include search path
  -c                      Only run preprocess, compile, and assemble steps
  -dD                     Print macro definitions in -E mode in addition to normal out
  -debug                  Stop and give PID to attach debugger to
  -dependency-dot <value> Filename to write DOT-formatted header dependencies to
  -dependency-file <value>
                          Filename (or -) to write dependency output to
  -dM                     Print macro definitions in -E mode instead of normal output
  -emit-ast               Emit Clang AST files for source inputs
  -emit-llvm              Use the LLVM representation for assembler and object files
  -E                      Only run the preprocessor
  -faltivec               Enable AltiVec vector initializer syntax
  -fansi-escape-codes     Use ANSI escape codes for diagnostics
  -fapple-kext            Use Apple's kernel extensions ABI
  -fapple-pragma-pack     Enable Apple gcc-compatible #pragma pack handling
  -fblocks                Enable the 'blocks' language feature
```

```
-fborland-extensions    Accept non-standard constructs supported by the Borland compiler
-fbuild-session-file=<file>
                        Use the last modification time of <file> as the build session timestamp
-fbuild-session-timestamp=<time since Epoch in seconds>
                        Time when the current build session started
-fcolor-diagnostics     Use colors in diagnostics
-fcomment-block-commands=<arg>
                        Treat each comma separated argument in <arg> as a documentation comment
-fcoverage-mapping      Generate coverage mapping to enable code coverage analysis
-fcxx-exceptions        Enable C++ exceptions
-fdata-sections         Place each data in its own section (ELF Only)
-fdebug-types-section   Place debug types in their own section (ELF Only)
-fdelayed-template-parsing
                        Parse templated function definitions at the end of the translation unit
-fdiagnostics-parseable-fixits
                        Print fix-its in machine parseable form
-fdiagnostics-print-source-range-info
                        Print source range spans in numeric form
-fdiagnostics-show-note-include-stack
                        Display include stacks for diagnostic notes
-fdiagnostics-show-option
                        Print option name with mappable diagnostics
-fdiagnostics-show-template-tree
                        Print a template comparison tree for differing templates
-fdollars-in-identifiers
                        Allow '$' in identifiers
-femit-all-decls        Emit all declarations, even if unused
-fexceptions            Enable support for exception handling
-ffast-math             Enable the *frontend*'s 'fast-math' mode. This has no effect on optimiz
-ffixed-r9              Reserve the r9 register (ARM only)
-ffp-contract=<value>   Form fused FP ops (e.g. FMAs): fast (everywhere) | on (according to FP
-ffreestanding          Assert that the compilation takes place in a freestanding environment
-ffunction-sections     Place each function in its own section (ELF Only)
-fgnu-keywords          Allow GNU-extension keywords regardless of language standard
-fgnu-runtime           Generate output compatible with the standard GNU Objective-C runtime
-fgnu89-inline          Use the gnu89 inline semantics
-finstrument-functions  Generate calls to instrument function entry and exit
-fintegrated-as         Enable the integrated assembler
-fmath-errno            Require math functions to indicate errors by setting errno
-fmax-type-align=<value>
                        Specify the maximum alignment to enforce on pointers lacking an explic
-fmodule-file=<file>    Load this precompiled module file
-fmodule-map-file=<file>
                        Load this module map file
-fmodule-maps           Read module maps to understand the structure of library headers
-fmodule-name= <name>   Specify the name of the module to build
```

```
-fmodules-cache-path=<directory>
                        Specify the module cache path
-fmodules-decluse       Require declaration of modules used within a module
-fmodules-ignore-macro=<value>
                        Ignore the definition of the given macro when building and l
-fmodules-prune-after=<seconds>
                        Specify the interval (in seconds) after which a module file
-fmodules-prune-interval=<seconds>
                        Specify the interval (in seconds) between attempts to prune
-fmodules-search-all    Search even non-imported modules to resolve references
-fmodules-strict-decluse
                        Like -fmodules-decluse but requires all headers to be in mod
-fmodules-user-build-path <directory>
                        Specify the module user build path
-fmodules-validate-once-per-build-session
                        Don't verify input files for the modules if the module has b
-fmodules-validate-system-headers
                        Validate the system headers that a module depends on when lo
-fmodules               Enable the 'modules' language feature
-fms-compatibility-version=<value>
                        Dot-separated value representing the Microsoft compiler vers
-fms-compatibility      Enable full Microsoft Visual C++ compatibility
-fms-extensions         Accept some non-standard constructs supported by the Microso
-fmsc-version=<value>   Microsoft compiler version number to report in _MSC_VER (0 =
-fno-access-control     Disable C++ access control
-fno-assume-sane-operator-new
                        Don't assume that C++'s global operator new can't alias any
-fno-autolink           Disable generation of linker directives for automatic librar
-fno-builtin-<value>    Disable implicit builtin knowledge of a specific function
-fno-builtin            Disable implicit builtin knowledge of functions
-fno-common             Compile common globals like normal definitions
-fno-constant-cfstrings Disable creation of CodeFoundation-type constant strings
-fno-diagnostics-fixit-info
                        Do not include fixit information in diagnostics
-fno-dollars-in-identifiers
                        Disallow '$' in identifiers
-fno-elide-constructors Disable C++ copy constructor elision
-fno-elide-type         Do not elide types when printing diagnostics
-fno-integrated-as      Disable the integrated assembler
-fno-lax-vector-conversions
                        Disallow implicit conversions between vectors with a differe
-fno-math-builtin       Disable implicit builtin knowledge of math functions
-fno-merge-all-constants
                        Disallow merging of constants
-fno-objc-infer-related-result-type
                        do not infer Objective-C related result type based on method
```

```
-fno-operator-names     Do not treat C++ operator name keywords as synonyms for operators
-fno-reroll-loops       Turn off loop reroller
-fno-rtti               Disable generation of rtti information
-fno-sanitize-blacklist Don't use blacklist file for sanitizers
-fno-sanitize-memory-track-origins
                        Disable origins tracking in MemorySanitizer
-fno-sanitize-recover   Disable sanitizer check recovery
-fno-short-wchar        Force wchar_t to be an unsigned int
-fno-show-column        Do not include column number on diagnostics
-fno-show-source-location
                        Do not include source location information with diagnostics
-fno-signed-char        Char is unsigned
-fno-spell-checking     Disable spell-checking
-fno-stack-protector    Disable the use of stack protectors
-fno-standalone-debug   Limit debug information produced to reduce size of debug binary
-fno-threadsafe-statics Do not emit code to make initialization of local statics thread safe
-fno-unroll-loops       Turn off loop unroller
-fno-use-cxa-atexit     Don't use __cxa_atexit for calling destructors
-fno-use-init-array     Don't use .init_array instead of .ctors
-fobjc-arc-exceptions   Use EH-safe code when synthesizing retains and releases in -fobjc-arc
-fobjc-arc              Synthesize retain and release calls for Objective-C pointers
-fobjc-exceptions       Enable Objective-C exceptions
-fobjc-gc-only          Use GC exclusively for Objective-C related memory management
-fobjc-gc               Enable Objective-C garbage collection
-fobjc-runtime=<value>  Specify the target Objective-C runtime kind and version
-fpack-struct=<value>   Specify the default maximum struct packing alignment
-fpascal-strings        Recognize and construct Pascal-style string literals
-fpcc-struct-return     Override the default ABI to return all structs on the stack
-fprofile-instr-generate
                        Generate instrumented code to collect execution counts
-fprofile-instr-use=<value>
                        Use instrumentation data for profile-guided optimization
-fprofile-sample-use=<value>
                        Enable sample-based profile guided optimizations
-freg-struct-return     Override the default ABI to return small structs in registers
-freroll-loops          Turn on loop reroller
-fsanitize-address-field-padding=<value>
                        Level of field padding for AddressSanitizer
-fsanitize-blacklist=<value>
                        Path to blacklist file for sanitizers
-fsanitize-coverage=<value>
                        Enable coverage instrumentation for Sanitizers
-fsanitize-memory-track-origins=<value>
                        Enable origins tracking in MemorySanitizer
-fsanitize-memory-track-origins
                        Enable origins tracking in MemorySanitizer
```

```
-fsanitize=<check>      Enable runtime instrumentation for bug detection: address (m
-fshort-enums           Allocate to an enum type only as many bytes as it needs for
-fshort-wchar           Force wchar_t to be a short unsigned int
-fshow-overloads=<value>
                        Which overload candidates to show when overload resolution f
-fslp-vectorize-aggressive
                        Enable the BB vectorization passes
-fslp-vectorize         Enable the superword-level parallelism vectorization passes
-fstack-protector-all   Force the usage of stack protectors for all functions
-fstack-protector-strong
                        Use a strong heuristic to apply stack protectors to function
-fstack-protector       Enable stack protectors for functions potentially vulnerable
-fstandalone-debug      Emit full debug info for all types used by the program
-fstrict-enums          Enable optimizations based on the strict definition of an en
-ftrap-function=<value> Issue call to specified function rather than a trap instruct
-ftrapv-handler=<function name>
                        Specify the function to be called on overflow
-ftrapv                 Trap on integer overflow
-funroll-loops          Turn on loop unroller
-fuse-init-array        Use .init_array instead of .ctors
-fvectorize             Enable the loop vectorization passes
-fvisibility-inlines-hidden
                        Give inline C++ member functions default visibility by defau
-fvisibility-ms-compat  Give global types 'default' visibility and global functions
-fvisibility=<value>    Set the default symbol visibility for all global declaration
-fwrapv                 Treat signed integer overflow as two's complement
-fwritable-strings      Store string literals as writable data
-F <value>              Add directory to framework include search path
--gcc-toolchain=<value> Use the gcc toolchain at the given directory
-gdwarf-2               Generate source-level debug information with dwarf version 2
-gdwarf-3               Generate source-level debug information with dwarf version 3
-gdwarf-4               Generate source-level debug information with dwarf version 4
-gline-tables-only      Emit debug line number tables only
-gpu                    Enable NVIDIA gpu mode.
-g                      Generate source-level debug information
-help                   Display available options
-H                      Show header includes and nesting depth
-idirafter <value>      Add directory to AFTER include search path
-iframework <value>     Add directory to SYSTEM framework search path
-imacros <file>         Include macros from file before parsing
-include-pch <file>     Include precompiled header file
-include <file>         Include file before parsing
-index-header-map       Make the next included directory (-I or -F) an indexer heade
-iprefix <dir>          Set the -iwithprefix/-iwithprefixbefore prefix
-iquote <directory>     Add directory to QUOTE include search path
-isysroot <dir>         Set the system root directory (usually /)
```

```
-isystem <directory>    Add directory to SYSTEM include search path
-ivfsoverlay <value>    Overlay the virtual filesystem described by file over the real file sy
-iwithprefixbefore <dir>
                        Set directory to include search path with prefix
-iwithprefix <dir>      Set directory to SYSTEM include search path with prefix
-iwithsysroot <directory>
                        Add directory to SYSTEM include search path, absolute paths are relati
-I <value>              Add directory to include search path
-mabicalls              Enable SVR4-style position-independent code (Mips only)
-mcrc                   Allow use of CRC instructions (ARM only)
-MD                     Write a depfile containing user and system headers
-mfix-cortex-a53-835769 Workaround Cortex-A53 erratum 835769 (AArch64 only)
-mfp32                  Use 32-bit floating point registers (MIPS only)
-mfp64                  Use 64-bit floating point registers (MIPS only)
-MF <file>              Write depfile output from -MMD, -MD, -MM, or -M to <file>
-mgeneral-regs-only     Generate code which only uses the general purpose registers (AArch64 o
-MG                     Add missing headers to depfile
--migrate               Run the migrator
-mllvm <value>          Additional arguments to forward to LLVM's option processing
-mlong-calls            Generate an indirect jump to enable jumps further than 64M
-MMD                    Write a depfile containing user headers
-mms-bitfields          Set the default structure layout to be compatible with the Microsoft c
-mmsa                   Enable MSA ASE (MIPS only)
-MM                     Like -MMD, but also implies -E and writes to stdout by default
-mno-abicalls           Disable SVR4-style position-independent code (Mips only)
-mno-fix-cortex-a53-835769
                        Don't workaround Cortex-A53 erratum 835769 (AArch64 only)
-mno-global-merge       Disable merging of globals
-mno-implicit-float     Don't generate implicit floating point instructions
-mno-long-calls         Restore the default behaviour of not generating long calls
-mno-msa                Disable MSA ASE (MIPS only)
-mno-restrict-it        Allow generation of deprecated IT blocks for ARMv8. It is off by defau
-mno-unaligned-access   Force all memory accesses to be aligned (AArch32/AArch64 only)
-mnocrc                 Disallow use of CRC instructions (ARM only)
-module-dependency-dir <value>
                        Directory to dump module dependencies to
-momit-leaf-frame-pointer
                        Omit frame pointer setup for leaf functions
-MP                     Create phony target for each dependency (other than main file)
-mqdsp6-compat          Enable hexagon-qdsp6 backward compatibility
-MQ <value>             Specify name of main file output to quote in depfile
-mrelax-all             (integrated-as) Relax all machine instructions
-mrestrict-it           Disallow generation of deprecated IT blocks for ARMv8. It is on by def
-mrtd                   Make StdCall calling convention the default
-msoft-float            Use software floating point
-mstack-alignment=<value>
```

```
                        Set the stack alignment
-mstackrealign          Force realign the stack at entry to every function
-mthread-model <value>  The thread model to use, e.g. posix, single (posix by defaul
-MT <value>             Specify name of main file output in depfile
-munaligned-access      Allow memory accesses to be unaligned (AArch32/AArch64 only)
-M                      Like -MD, but also implies -E and writes to stdout by defaul
--no-system-header-prefix=<prefix>
                        Treat all #include paths starting with <prefix> as not inclu
-nobuiltininc           Disable builtin #include directories
-noscstdinc             Disable scout's default standard include directory.
-noscstdlib             Disable scout's automatic link-time inclusion of libraries.
-nostdinc++             Disable standard #include directories for the C++ standard l
-ObjC++                 Treat source input files as Objective-C++ inputs
-objcmt-atomic-property Make migration to 'atomic' properties
-objcmt-migrate-all     Enable migration to modern ObjC
-objcmt-migrate-annotation
                        Enable migration to property and method annotations
-objcmt-migrate-designated-init
                        Enable migration to infer NS_DESIGNATED_INITIALIZER for init
-objcmt-migrate-instancetype
                        Enable migration to infer instancetype for method result typ
-objcmt-migrate-literals
                        Enable migration to modern ObjC literals
-objcmt-migrate-ns-macros
                        Enable migration to NS_ENUM/NS_OPTIONS macros
-objcmt-migrate-property-dot-syntax
                        Enable migration of setter/getter messages to property-dot s
-objcmt-migrate-property
                        Enable migration to modern ObjC property
-objcmt-migrate-protocol-conformance
                        Enable migration to add protocol conformance on classes
-objcmt-migrate-readonly-property
                        Enable migration to modern ObjC readonly property
-objcmt-migrate-readwrite-property
                        Enable migration to modern ObjC readwrite property
-objcmt-migrate-subscripting
                        Enable migration to modern ObjC subscripting
-objcmt-ns-nonatomic-iosonly
                        Enable migration to use NS_NONATOMIC_IOSONLY macro for setti
-objcmt-returns-innerpointer-property
                        Enable migration to annotate property with NS_RETURNS_INNER_
-objcmt-whitelist-dir-path=<value>
                        Only modify files with a filename contained in the provided
-ObjC                   Treat source input files as Objective-C inputs
-o <file>               Write output to <file>
-pg                     Enable mcount instrumentation
```

```
-pipe                    Use pipes between commands, when possible
-print-file-name=<file> Print the full library path of <file>
-print-ivar-layout       Enable Objective-C Ivar layout bitmap print trace
-print-libgcc-file-name Print the library path for "libgcc.a"
-print-prog-name=<name> Print the full program path of <name>
-print-search-dirs       Print the paths used for finding libraries and programs
-pthread                 Support POSIX threads in generated code
-P                       Disable linemarker output in -E mode
-Qunused-arguments       Don't emit warning for unused driver arguments
-relocatable-pch         Whether to build a relocatable precompiled header
-rewrite-legacy-objc     Rewrite Legacy Objective-C source to C++
-rewrite-objc            Rewrite Objective-C source to C++
-Rpass-analysis=<value> Report transformation analysis from optimization passes whose name mat
-Rpass-missed=<value>   Report missed transformations by optimization passes whose name matche
-Rpass=<value>          Report transformations performed by optimization passes whose name mat
-R<remark>              Enable the specified remark
-save-temps             Save intermediate compilation results
-sclegion               Run Scout functions as Legion tasks.
-serialize-diagnostics <value>
                        Serialize compiler diagnostics to a file
-std=<value>            Language standard to compile for
-stdlib=<value>         C++ standard library to use
--system-header-prefix=<prefix>
                        Treat all #include paths starting with <prefix> as including a system h
-S                      Only run preprocess and compilation steps
--target=<value>        Generate code for the given target
-time                   Time individual commands
-traditional-cpp        Enable some traditional CPP emulation
-trigraphs              Process trigraph sequences
-undef                  undef all system defines
--verify-debug-info     Verify the binary representation of debug output
-verify-pch             Load and verify that a pre-compiled header file is not stale
-v                      Show commands to run and use verbose output
-Wa,<arg>               Pass the comma separated arguments in <arg> to the assembler
-Wl,<arg>               Pass the comma separated arguments in <arg> to the linker
-working-directory <value>
                        Resolve file paths relative to the specified directory
-Wp,<arg>               Pass the comma separated arguments in <arg> to the preprocessor
-W<warning>             Enable the specified warning
-w                      Suppress all warnings
-Xanalyzer <arg>        Pass <arg> to the static analyzer
-Xassembler <arg>       Pass <arg> to the assembler
-Xclang <arg>           Pass <arg> to the clang compiler
-Xlinker <arg>          Pass <arg> to the linker
-Xpreprocessor <arg>    Pass <arg> to the preprocessor
-x <language>           Treat subsequent input files as having type <language>
```

-z <arg>	Pass -z <arg> to the linker