Los Alamos
NATIONAL LABORATORY
EST. 1943

Exploring the Construction of a Domain-Aware Toolchain for High-Performance Computing

November 17, 2014

WOLFHPC Workshop

LA-UR-14-28752

Authors: Patrick McCormick,
Christine Sweeney (presenter), Nick Moss,
Dean Prichard, Samuel K. Gutierrez,
Kei Davis, Jamaludin Mohd-Yusof

Los Alamos National Laboratory
Funding by Office of Advanced Scientific Computing Research,
Office of Science, Program Manager, Lucy Nowell

Scout Project

- Scout domain-specific language via conservative extensions to C/C++ (compiled, not source-to-source)
- Supports mesh-based applications, *in situ* visualization and data and task parallelism.
- Includes a domain-aware debugging tool
- Targets GPU (CUDA or OpenCL)
- Targets Legion Runtime/Programming Model (http://legion.stanford.edu)

Talk Overview

- Motivation and design decisions
- Domain-specific language constructs
- Compiler implementation and debugger
- Evaluation
- Conclusion and future work

Motivation and Vision for Scout Domain-Aware Toolchain

- Enable scientists to productively develop mesh-based HPC applications via language and toolchain infrastructure
- Enable scientific applications to be portable to different and future large-scale computer architectures with little or no modification.
- Focus on toolchain, not so much language details, a specific scientific domain or performance at the moment.

Design Decisions for Scout

DSL versus general purpose library?

- **DSL** provides natural way to express science via domain-specific notations

Embedded versus extensions versus standalone DSL?

- Domain-centric conservative **extensions** to C/C++

Compiled versus source-to-source?

- **Compiled** can preserve domain-awareness
- Enables finer-grained control over performance optimizations

Scout Domain-Specific Data Types

```
uniform mesh MyUniformMesh {
  // Define the fields stored on the mesh.
  cells       : float pressure, temperature;
  vertices    : float3 vorticity;
  edges       : float3 velocity;
};

// Declare a two-dimensional uniform mesh with 3 cells
// along the x-axis and 2 cells along the y-axis
MyUniformMesh umesh[3,2];
```

- Mesh is first-class concrete data type
- Unlike C/C++, developer should not assume any details about memory layout of mesh structure.
- Mesh can be passed as an argument to a function

Data Parallel Scout DSL Constructs

```
// For all cells 'c' of the mesh 'umesh'
forall cells c in umesh {
        ...
  forall vertices v in c { // 'v' -> active vertex
        // vertex values are read-only, cell values
        // are read/write-able
    c.temperature = ... v.velocity ...;
  }
}
```

- Mesh elements may only be accessed via mesh-centric constructs.
- No assumptions should be made about order of execution.
- Built-ins for position, width, height, depth, cshift

Task Parallel Scout DSL Constructs

```
task void MyTask(MyMesh &m) {
  // body of task...
}

...

MyTask(m); // Invoke the task on the mesh
```

- A "task function" must operate on a mesh instance passed as a parameter
- Task must not modify global variables otherwise will not compile…

Visualization Scout DSL Constructs

```
extern const float MAX_TEMPERATURE;
...
// Create a 512 x 512 window for displaying mesh elements
window win[512, 512];
...
// Render the cells to the window.  ‘color’
// must be assigned to within the loop body.
// This assigns a color to the ‘active’ cell.
renderall cells c in umesh to win {
        float norm_temp = c.temperature / MAX_TEMPERATURE;
        // Use the HSV (hue, saturation, value) colorspace
        // to assign a color from blue (cold) to red (hot)
        // for the cell.
        color = hsv(240.0 – 240.0 * norm_temp, 1.0, 1.0);
}
```

- Data-parallel model for doing *in situ* visualization of mesh topology.

Visualization Demo

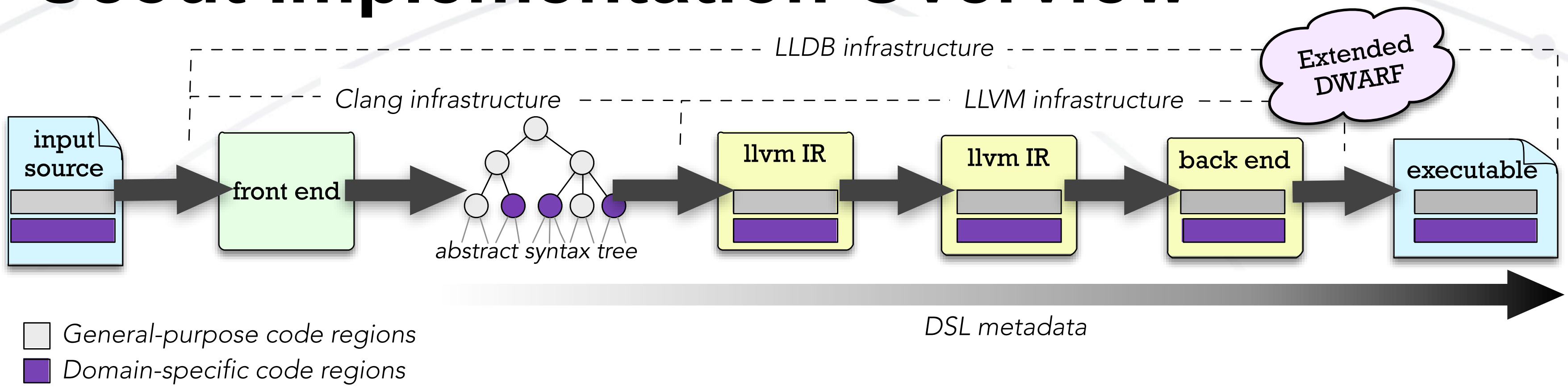
```
// Heat Transfer Example
// Time steps loop.
for(unsigned int t = 0; t < NTIME_STEPS; ++t) {

  forall cells c in heat_mesh {
    if (position().x > 0
        && position().x < width()-1
        && position().y > 0
        && position().y < 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;
    }
  }
  forall cells c in heat_mesh {
    t1 = t2;
  }
  renderall cells c in heat_mesh to render_win {
    float norm_t1 = t1 / MAX_TEMP;
    float hue = 240.0f - 240.0f * norm_t1;
    color = hsv(hue, 1.0f, 1.0f);
  }
}
```

LLVM Compiler Infrastructure

LLVM Project - modular and reusable compiler and toolchain technologies. Subprojects:

- **LLVM Core** - source- and target-independent optimizer plus code-generation for CPU and GPU targets.
- **LLVM Intermediate Representation** (IR) -language and architecture independent representation of source code
- **Clang** – C/C++ front-end and platform for building source-level tools.
- **LLDB** - native debugger built on Clang and LLVM libraries.
- See http://llvm.org

Scout Implementation Overview



- **Front End (Clang)** is modified to recognize Scout syntax and semantics (rules).
- **Abstract Syntax Tree (AST**) is modified to store Scout's own unique nodes.
- **Intermediate Representation (IR**) is generated to support Scout's data types and statements.
- **Metadata** maintains domain-specific information throughout compilation and into debugging (DWARF data structures).

Metadata in LLVM

- Metadata is additional data that can be stored on LLVM IR and gets used by debugging
- Scout uses metadata to store:
  - mesh fields
  - GPU kernel indicators
  - task indicators.

Metadata

```
uniform mesh MyUniformMesh {
  // Define fields stored on
  // the mesh.
  cells: float temperature;
  vertices: float3 velocity;
  edges: float3 flux;
};

// define a two-dimensional
// uniform mesh with 3 cells
// along the x-axis and 2 cells
// along the y-axis
MyUniformMesh umesh[3,2];
```

```
%MyUniformMesh = type {
  float*,         ; temperature
  <3 x float>*,  ; velocity
  <3 x float>*   ; flux
...

; mesh metadata
!scout.meshmd = !{!0}  // one mesh entry.

!0=metadata
  !{metadata !"MyUniformMesh", ; 1st entry
    metadata !"uniform", i32 2,; mesh kind/rank
    metadata !"cells",
    metadata !1,          ;cell fields at !1
    metadata !"vertices",
    metadata !2,          ;vertex fields at !2
    metadata !"edges",
    metadata !3           ;edge fields at !3
   }
; cell fields
!1=metadata !{metadata !"float",
              metadata !"temperature"}
; vertex fields
!2=metadata !{metadata !"float3",
              metadata !"velocity"}
; edge fields
!3=metadata !{metadata !"float3",
              metadata !"flux"}
```

IR and Code Generation for GPU

- Lower `forall` body and use hardware-independent loop variables
- Create function out of `forall` to represent GPU kernel (flag it via metadata)
- Via an LLVM pass, transform thread index values
- For NVIDIA (CUDA), generate in-lined character string version of kernel in NIVIDIA PTX.
- For AMD (OpenCL runtime) create an Executable and Linking Format (ELF) version of the kernel.

IR Generation for Legion Runtime

- Legion Runtime provides single programming model for target, insulates from data layout, movement and hardware
- Generate LLVM IR that calls simplified C-based Legion runtime interface
- Express meshes as Legion logical regions and task functions as Legion tasks
- Distinguish task functions from non-task functions via metadata
- Initialize Legion and register tasks – transform main()

Debugging with LLDB

- During compilation with debug flag, Clang generates IR metadata which then gets converted to DWARF data structures.
- DWARF Debugging Information Entry (DIE) data structure is used for each function or variable; many DIEs form a tree-like structure representing the program.
- When the user enters an expression into the debugger, LLDB uses DWARF information to reconstruct Clang AST, lower to IR and execute.

Enabling Domain-Aware Debugging

- Extend DWARF DIE tags and attributes
- Extend LLDB and Clang to reconstruct domain-specific AST nodes from the mesh DWARF information
- Leverage LLDB's use of clang to JIT expressions containing Scout constructs in the debugger
  - LLDB recreates the Clang AST (which includes nodes for Scout constructs)
  - AST gets lowered to IR as usual

Debug Session

```
145  // Time step loop.
146  for(unsigned int t = 0; t < NTIME_STEPS; ++t) {
147
148    forall cells c in heat_mesh {
149      // compute h_next
…        // … code omitted
156    }
157
158    forall cells c in heat_mesh {
159      h_now = h_next;
160    }
161   }
162  }
163  return 0;
```

```
(lldb) b heat4.sc:162
(lldb) expr  { window render_win[512,512];
 renderall cells c in heat_mesh to render_win{
   float norm_h = h_now / MAX_TEMP;
   float hue = 240.0f - 240.0f * norm_h;
   color = hsva(hue, 1.0f, mask_now, 1.0f);
  }}
(lldb) c
```

/Users/nickm/sc/heat4.sc

scratch | heat4.sc

```
forall cells c in heat_mesh{
  h_now = 0.0f;
  h_next = 0.0f;
  mask_now = 1.0;

  if(position().y == 0 || position().y == (height() - 1)){
    h_now = MAX_TEMP;
    h_next = MAX_TEMP;
    mask_now = 0.0;
  }

  for(int i = 0; i < N_BODIES; i++){
    float r2 = (position().x - c_x[i])*(position().x - c_x[i]) +
      (position().y - c_y[i])*(position().y - c_y[i]);

    if(r2 < r2cyl){
      if(SOLUBLE){
        mask_now = r2/r2cyl;
      }
      else{
        mask_now = 0.0;
      }

      h_now = MAX_TEMP;
      h_next = MAX_TEMP;
    }
  }
}

const float dx    = 10.0f / MESH_DIM;
const float dy    = 10.0f / MESH_DIM;
const float alpha = 0.00001f;
const float dt    = 0.5f * (dx * dx+ dy * dy)/4.0f/alpha;

for(unsigned int t = 0; t < NTIME_STEPS; t += 30){
  for(unsigned int t2 = 0; t2 < 30; ++t2){
    forall cells c in heat_mesh {
      float ddx =
        0.5*(cshift(c.h_now, 1, 0) - cshift(c.h_now, -1, 0))/dx;

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

      d2dx2 /= dx * dx;

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

      d2dy2 /= dy * dy;

      h_next =
        mask_now*dt*(alpha * (d2dx2 + d2dy2) -
                     mask_now*u*ddx) + c.h_now;
    }

    forall cells c in heat_mesh{
      h_now = h_next;
    }
  }
}

return 0;
}
```

-:--- heat4.sc   Bot (164,0)   (C++/l Abbrev)

sc

```
zoot:sc nickm$
```

Evaluation of Scout

Challenges:

- Significant investment for development
- Acceptance and adoption of DSLs

Benefits:

- Produces mesh-based programs with far fewer lines of code
- Significantly simplified a complex runtime interface
- Familiar programming language and toolchain

Conclusions

- Scout is a solid and extensible basis for further exploration.
- Initial results show that Scout's approach can improve productivity and simplify programming.
- Chances for adoption and acceptance are higher with familiar toolchain implementation.

Future Work

- Distributed debugging and debugging on heterogeneous architectures
- Preserve domain context within runtime for debugging tasks that use Legion data model
- Extend task keyword for data parallelism via data decomposition of the mesh

Thank you!

Questions?

Legion Terminology

Via an API,

- Define *logical data regions*
- Register *tasks* — C/C++ functions that operate on data regions, may invoke other tasks

For each task definition specify

- What parts of logical data regions the task may access
- Potentially fine-grained information about type of access —read, write, read/write, etc.

Tasks *launched* (enqueued) serially, or mapped over multiple (sub-)regions