Stencil Domain Specific Language (SDSL) User Guide 0.3.2

Tom Henretty Prashant Rawat Justin Holewinski
Naser Sedaghati Louis-Nöel Pouchet Atanas Rountev
P. Sadayappan

Compiled August 24, 2014

Contents

1 Introduction **1**

2 Installation **3**
2.1 Required Software 3
2.2 Building and Installing 4
2.3 Quick Start 5

3 SDSL Language Reference **7**
3.1 Introduction to the Running Example 7
3.2 Language Features 8
3.2.1 Comments 8
3.2.2 Literal Values 8
3.2.3 Identifiers 9
3.2.4 Arithmetic Expressions 9
3.2.5 Parameters and Constants 9
3.2.6 Grids 9
3.2.7 Grid Data 10
3.2.8 Grid Data References 10
3.2.9 Grid Subsets 10
3.2.10 Point Functions 11
3.2.11 Iterate and Check Every 11
3.2.12 Stencils 12
3.2.13 Reductions 13

4 Embedded SDSL **15**
4.1 Embedding SDSL in C/C++ 15
4.1.1 `#pragma sdsl` 15
4.1.2 Parameter and Array Passing 15
4.1.3 Return Values 16
4.2 Embedding SDSL in MATLAB 16

5 SDSL Compiler Reference **19**
5.1 Common Case Usage . . . 19
5.1.1 SDSL embedded in C/C++ . . . 19
5.1.2 SDSL embedded in MATLAB . . . 20
5.2 Command Line Options . . . 20
5.3 Running the Example Benchmarks . . . 21
5.3.1 CDSC Pipeline . . . 22
5.3.2 General Benchmarks . . . 22

6 SDSL Autotuner Reference **25**
6.1 Common Case Usage . . . 25
6.2 Configuration File Format . . . 26
6.3 Embedded SDSL File Restrictions . . . 27
6.4 Autotuning the Example Benchmarks . . . 27
6.4.1 CDSC Pipeline . . . 27
6.4.2 General Benchmarks . . . 28

7 Detailed Example: Rician Denoise **29**
7.1 C Reference Code . . . 29
7.1.1 Function Signature . . . 29
7.1.2 Arrays . . . 31
7.1.3 Constants and Variables . . . 31
7.1.4 Main Iterative Loop . . . 31
7.1.5 `G` Stencil . . . 32
7.1.6 `U` Stencil . . . 32
7.1.7 Convergence Check . . . 32
7.2 Creating a C/SDSL Skeleton Function . . . 32
7.3 Passing Data from C to SDSL . . . 33
7.3.1 Parameters and Constants . . . 33
7.3.2 Arrays . . . 33
7.4 Implementing `pointfunction`s . . . 34
7.4.1 `approx_g` . . . 34
7.4.2 `update_u` . . . 35
7.5 Defining the Iterate . . . 35
7.5.1 Stencils . . . 36
7.5.2 `reduction` . . . 38
7.5.3 Convergence Check . . . 38
7.6 Generating Code . . . 38
7.7 Autotuning OverTile Parameters for GPU . . . 39
7.7.1 Building the Configuration File . . . 40
7.7.2 Running the Autotuner . . . 40
7.7.3 Examining the Autotuner Results . . . 40

8 Known Limitations **43**
8.1 General 43
8.2 Affine C Backend 43
8.3 OverTile Backend 43
8.4 Nested Split-tiling DLT Backend 44
8.5 Hybrid Split-tiling DLT Backend 44

Bibliography **45**

Chapter 1

Introduction

SDSL (**S**tencil **D**omain **S**pecific **L**anguage) is a domain-specific language for expressing stencil computations. SDSL is loosely based on the RNPL[5] and SNPL[6] languages used for rapid prototyping of partial differential equation solvers, although the resemblance is mostly cosmetic and no code is shared between the projects.

The purpose of SDSL is to provide a programming language that allows for the specification of non-trivial stencil computations in a form that enables the generation of high-performance implementations that can be obtained in a performance-portable manner on multiple platforms.

Currently, SDSL can be translated into codes for CPU and GPU execution. CPU code is generated by number of backends that produce unoptimized and optimized C code with affine structure.

- Unoptimized affine C code is C99 compliant and is meant to be processed further by polyhedral optimization tools such as PoCC [7] and PolyOpt/C [8].
- Optimized CPU code is C99 compliant and uses vector intrinsic functions for x86 and 32-bit ARM CPU. The code is optimized with nested/hybrid split-tiling [3] in conjunction with dimension-lift-and-transpose [2] data layout transformations.
- Optimized GPU code is generated in CUDA C and is performed by the OverTile [4] backend. OverTile generated code can be autotuned using a simple script included with the `sdslc` distribution.

SDSL code can be embedded in C, C++, and MATLAB. Optimized MATLAB code is generated as C functions called via MEX.

The SDSL User Guide is organized into the following chapters:

- Chapter 1: *"Introduction"*
- Chapter 2: *"Installation"*
- Chapter 3: *"SDSL Language Reference"*

- Chapter 4: *"Embedded SDSL"*
- Chapter 5: *"SDSL Compiler Reference"*
- Chapter 6: *"SDSL Autotuner Reference"*
- Chapter 7: *"Example: Rician Denoise"*
- Chapter 8: *"Known Limitations"*

Chapter 2

Installation

2.1 Required Software

The following components are required to build the SDSL compiler:

- Apache Ant
- Bison
- CMake 2.8 or higher
- gcc/g++ 4.4 or higher
- Java JDK 1.6 or higher
  - Must be JDK, not JRE
- LLVM 3.0 or higher
  - Must be built with CMake [1]
- Nvidia CUDA SDK 5.0 or higher
- Python 2.7

The SDSL compiler has been successfully built and run on Fedora 16, Ubuntu 12.04, and RHEL 6.3.

[1] Please note that most Linux distributions including RHEL, Ubuntu, and Fedora **do not** compile LLVM with CMake. This means that LLVM will need to be built from source with CMake. See `http://llvm.org/docs/CMake.html` for details on how to do this.

2.2 Building and Installing

To setup the build environment, please set the following environment variables:

- `JAVA_HOME`: Set to installation directory of Java JDK
  e.g. `export JAVA_HOME=/usr/lib/jvm/java-1.6.0-openjdk`
- `PATH`: Make sure `ant` is available on your `PATH`
- `LLVM_HOME`: Set to installation directory of LLVM
- `CUDA_HOME`: Set to installation directory of CUDA SDK
- `SDSLC_INSTALL_DIR`: Set to directory you wish to install SDSLC

The entire build process is controlled by a CMake script. You can generate the makefiles for the project by creating a build directory and invoking CMake. The following CMake options are recognized:

- `OT_LLVM_BINARY_DIR`
  - The installation path of LLVM
  - Required
- `CUDA_INSTALL_DIR`
  - The installation path of the CUDA SDK
  - Defaults to `/usr/local/cuda`
- `SDSLC_INSTALL_DIR`
  - Path to install the sdslc package at
  - Defaults to `/usr/local`

The following series of commands will build `sdslc` starting from a tar.gz distribution:

```
$ tar xzvf sdslc-0.3.2.tar.gz
$ cd sdslc-0.3.2
$ mkdir build
$ cd build
$ cmake -DOT_LLVM_BINARY_DIR=$LLVM_HOME -DCUDA_INSTALL_DIR=$CUDA_HOME
        -DSDSLC_INSTALL_DIR=/usr/local/sdslc-0.3.2 ..
$ make
$ make install
```

Root or `sudo` access may be required for the `make install` command, depending on the value of `SDSLC_INSTALL_DIR`. The main executable produced is the `$SDSLC_INSTALL_DIR/bin/sdslc` script that wraps the sdslc Java program.

2.3 Quick Start

To quickly build and run SDSL example codes see Section 5.3, *"Running the Example Benchmarks"*, and Section 6.4, *"Autotuning the Example Benchmarks"*.

Chapter 3

SDSL Language Reference

This chapter provides a reference for all major SDSL language features. A simple 2-dimensional Jacobi stencil is used a running example to illustrate the usage of the language features throughout the rest of the chapter.

3.1 Introduction to the Running Example

Figure 3.1 is a complete SDSL program to compute a 2-dimensional, 5 point Jacobi stencil. This section gives an overview of the program; the remainder of this chapter uses this example to illustrate important language features.

```
1  int dim0;
2  int dim1;
3
4  grid g [dim1][dim0];
5
6  float griddata a on g at 0,1;
7
8  pointfunction five_point_avg(p) {
9    float ONE_FIFTH;
10   ONE_FIFTH = 0.2f;
11   [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1] + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
12 }
13
14 iterate 1000 {
15   stencil jacobi_2d {
16     [0][0:dim0-1]      : [1]a[0][0] = [0]a[0][0];
17     [dim1-1][0:dim0-1] : [1]a[0][0] = [0]a[0][0];
18     [0:dim1-1][0]      : [1]a[0][0] = [0]a[0][0];
19     [0:dim1-1][dim0-1] : [1]a[0][0] = [0]a[0][0];
20
21     [1:dim1-2][1:dim0-2] : five_point_avg(a);
22   }
23
24   reduction max_diff max {
25     [0:dim1-1][0:dim0-1] : fabs([1]a[0][0] - [0]a[0][0]);
26   }
27 } check (max_diff < .00001f) every 4 iterations
```

Figure 3.1: Jacobi 2-dimensional, 5 point stencil written in SDSL.

The program begins with a declaration of two `int` parameters, `dim0` and `dim1` on lines 1 and 2. These parameters are used to define a 2-dimensional `grid` of size `dim1`×`dim0` on line

4. Line 6 declares floating point `griddata` in the shape of the `grid` defined on line 4. This data is also declared to exist at two different offsets from the current timestep, `0` and `1`.

Lines 8–12 define a 5 point stencil as a `pointfunction` named `five_pt_avg`. This function takes a parameter `p`, the grid data used to calculate the stencil. `five_pt_avg` averages 5 neighboring points in one timestep of `p` and writes the result to a point in the next timestep of `p`.

Lines 14–27 define an iterative loop with a convergence check. The `iterate` construct on line 14 specifies that the iterative loop should run, at most, 1000 times. Lines 16–21 define a stencil computation over subsets of the problem domain. Lines 16–19 specify the value of the edges of grid data `a` to be the same at both timesteps at which `a` is defined. Line 21 specifies that the point function `five_pt_avg` be applied at every point over the interior of `a`.

Lines 24–26 define a `reduction` to be performed at every point on `a`. The reduction variable is `max_diff`, and it contains the largest difference between a value of an element of `a` at two successive timesteps.

Finally, line 27 defines a convergence condition for the `iterate` and specifies how frequently this check is to be performed. In this program the condition is that the largest difference between successive time values of an element of `a` is less than .00001 and that this condition should be checked every 4 iterations.

3.2 Language Features

The subsections that follow provide detailed information on the various constructs of the SDSL language.

3.2.1 Comments

SDSL programs can be commented with C/C++ style comments. C-style comments begin with the characters `/*` and continue until terminated by the `*/` characters. C++-style comments begin with the characters `//` and continue until terminated by a new line.

3.2.2 Literal Values

Floating point and integer literal values can be explicitly represented in SDSL. These values are used in SDSL expressions, described in Section 3.2.4. Both floating point and integer literals are always interpreted as base 10 numbers.

In the current definition of the language, the syntax and types of these literals correspond to the `int`, `long`, `float`, and `double` literals of the C/C++ language in which SDSL is embedded (see Chapter 4). Future refinements of SDSL may introduce a richer set of literal values (e.g., hex/octal literals).

3.2.3 Identifiers

C-style identifiers are used to name parameters, constants, grids, grid data, point functions, point function parameters, variables in point functions, stencils, and reductions.

The following production defines legal identifiers in SDSL:

```
ID  : (‘a’..‘z’|‘A’..‘Z’|‘_’) (‘a’..‘z’|‘A’..‘Z’|‘0’..‘9’|‘_’)*
```

The first character of an identifier must be a letter or an underscore. Subsequent characters must be letters, numbers, or underscores.

3.2.4 Arithmetic Expressions

Arithmetic expressions contain literals, constants, parameters, grid data references, and function calls. Addition, subtraction, multiplication, division, and modulo operations are permitted. In the current implementation, calls to side-effect-free external functions from C's `math.h` and `stdlib.h` are permitted by default; a more general mechanism for accessing external code from within SDSL code will be designed in the future.

Expression terms may be grouped together using parentheses. Operator associativity and precedence are similar to those in C/C++.

3.2.5 Parameters and Constants

Parameters and constants are the first items declared and defined in a SDSL program. Both parameters and constants can be of type `int`, `long`, `float`, or `double`. A parameter declaration includes the type and identifier of the parameter, while a constant is declared with the type, name, and value. Parameters are bound to values at the time of program execution. Examples of parameter and constant declarations are shown below.

```
int dim0;
int dim1;

float PI = 3.14f;
float CONVERGE_AT = .00001f;
```

3.2.6 Grids

A `grid` defines the physical geometry of a domain over which a stencil is to be computed. Grids are restricted to being rectangular, and can be 1-, 2-, or 3-dimensional.

```
grid g [dim1][dim0];
```

The above SDSL statement defines a grid with the identifier `g`. This grid is 2-dimensional with the first dimension having a size of `dim1` points and the second dimension having a size of `dim0` points.

3.2.7 Grid Data

`griddata` is a concrete instance of a `grid` and has a type of `int`, `long`, `float`, or `double`. Grid data is defined over one or more offsets from the current timestep. Timestep offsets must be consecutive integers.[1]

```
float griddata a on g at 0,1;
```

The above `griddata` declaration states that we are creating a concrete instance of `grid` `g` of type `float` with the identifier `a` defined over timesteps `0` and `1`.

3.2.8 Grid Data References

Grid data is referenced in point functions, iterates, and reductions. A grid data reference consists of a temporal offset, the identifier of the grid data being referenced, and a spatial offset for each dimension of the grid data.

The temporal offset of a grid data reference is the offset with respect to the current timestep. Suppose a point function is currently being executed at time t. A temporal offset of 1 corresponds to grid data at time $t+1$. At the next time iteration $t' = t+1$, a temporal offset of 0 would be used to access the same data accessed with a temporal offset of 1 at time t.

When grid data is referenced spatially, it is always in relation to a point in an n-dimensional grid. All offsets are calculated from this point. Suppose a point function is currently being applied at point i in a one dimensional grid. An offset of 1 corresponds to the point $i+1$.

Below is a simple 2-dimensional grid data references in SDSL representing the current point being processed, 1 timestep in the future.

```
[1]data[0][0]
```

The temporal offset, enclosed by brackets, is first. After the temporal offset is the identifier of the grid data being referenced. Finally, the offsets of the referenced point in each spatial dimension, with each offset enclosed in brackets, are given. Another 2-dimensional reference is shown below. This reference is offset by -1 in the outer spatial dimension.

```
[1]data[-1][0]
```

3.2.9 Grid Subsets

Subsets of a grid can be specified in a `stencil` and a `reduction` so that different computations are performed according to the location of a point on a grid. A subset of a grid is defined in each spatial dimension and is two integer expressions, separated by a colon, enclosed in brackets. The expressions define the start and end coordinates of the subset in the given dimension. An example subset containing all but the edges of a 2-dimensional `dim1`×`dim0` grid is shown below.

[1] A convenient way to view timestep offsets is as another dimension of a grid. A 2-dimensional, `dim1`×`dim0` grid defined on timesteps 0 and 1 can be thought of as a 3-dimensional grid with the third dimension having a size of 2.

`[1:dim1-2][1:dim0-2]`

Grid subsets may also consist of just one expression, in which case the subset exists only at the coordinate obtained by evaluating the expression. The example below illustrates this by defining a subset representing the last column of an `dim1`×`dim0` grid.

`[0:dim1-1][dim0-1]`

More examples of subsets can be seen in Section 3.2.12 and Section 3.2.13.

3.2.10 Point Functions

Point functions contain stencil implementations. A `pointfunction` takes `griddata` as parameters and executes a series of computations using grid data references. When a point function is called, it is at an arbitrary point in a grid. All spatial components of grid data references within the point function are offsets from this point.

A `pointfunction` may compute an arbitrary number of points on an arbitrary number of grid data references. Variables of type `int`, `long`, `float`, and `double` may be declared and used. Constant values and parameters of the SDSL program can be used and should not be included in the parameter list of the point function. Conditional statements are currently not allowed.

```
pointfunction five_point_avg(p) {
 float ONE_FIFTH;
 ONE_FIFTH = 0.2f;
 [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1] + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
}
```

Figure 3.2: Jacobi 2-dimensional, 5 point stencil point function.

Figure 3.2 corresponds to a simple Jacobi stencil that averages the point being processed and four of its neighboring points. It begins with the `pointfunction` keyword followed by the identifier `five_point_avg`. Only one grid data parameter, `p`, is passed. The body of the point function contains the declaration of a floating point variable `ONE_FIFTH` and the subsequent assignment of the value 0.2 to that variable. The last line updates the current point in `p` at timestep 1 with the value of itself and its top, bottom, left, and right neighbors at timestep 0.

3.2.11 Iterate and Check Every

The outer iterative loop of a stencil computation is defined using the `iterate` construct. This construct contains a sequence of stencil and reduction definitions. The defined stencils are executed once every outer loop iteration, and are executed in the order in which the `stencil` definitions appear in the `iterate`. Reductions are performed every n-th iteration as specified by the `check ... every` clause. The defined reductions are also executed in the order in which the `reduction` definitions appear in the `iterate`. The general form of an `iterate` is shown in Figure 3.3.

An iterate defines a maximum number of times that a sequence of stencils and reductions will execute. The only way for an iterate to terminate before the given number of iterations

```
iterate maximum iterations {
  sequence of stencils and / or reductions
} optional termination condition
```

Figure 3.3: General form of the `iterate` construct.

is through the definition of a termination condition (such as a convergence check) via the optional `check ... every` clause following the iterate. The general form of the clause is shown below.

`check (`*conditional expression*`) every` *n* `iterations`

The conditional expression uses the `>, <, <=, >=, ==,` and `!=` operators to form expressions using reduction variables, parameters, constants, floating point literals, and integer literals. This expression evaluates to a boolean value. If the condition evaluates to *true*, the outer iterative loop stops executing. This condition is evaluated at the end of every n-th iteration, as specified in the `every` part of the clause.

Inside of the iterate, stencils and reductions can be specified. These form the bulk of the computational work and are described in the following sections.

3.2.12 Stencils

A stencil is always located inside an iterate and defines an iteration over programmer-defined elements of an abstract n-dimensional grid. This iteration is not guaranteed to occur in any order and can occur in parallel. The dimensionality of the grid is determined by the dimensionality of the grid subsets defined within the stencil. A stencil has the general form shown in Figure 3.4.

```
stencil identifier {
  sequence of (grid subset,expression) pairs
}
```

Figure 3.4: General form of the `stencil` construct.

A stencil begins with the `stencil` keyword followed by an identifier. Inside the body of the stencil are one or more pairs of the form `gs:exp;` where `gs` is a grid subset and `exp` is either an assignment expression or a point function call.

```
stencil jacobi_2d {
  [0][0:dim0-1]      : [1]a[0][0] = [0]a[0][0];
  [dim1-1][0:dim0-1] : [1]a[0][0] = [0]a[0][0];
  [0:dim1-1][0]      : [1]a[0][0] = [0]a[0][0];
  [0:dim1-1][dim0-1] : [1]a[0][0] = [0]a[0][0];

  [1:dim1-2][1:dim0-2] : five_point_avg(a);
}
```

Figure 3.5: Stencil written in SDSL.

An example stencil is given in Figure 3.5. The stencil begins with the `stencil` keyword followed by the identifier `jacobi_2d`. The body of the stencil sets the top, bottom, left, and

right edges of grid data `a` at timestep 1 to their values at timestep 0. Finally, the body of the grid data is computed with the point function `five_point_avg`.

3.2.13 Reductions

A reduction consists of the `reduction` keyword, an identifier that names the reduction, and a reduction operator. Finally, a reduction contains a sequence of grid subsets and an operation that is to be performed on that subset. Reductions cannot contain point function calls.

The results of the computation at each grid point are combined with the reduction operator into a reduction variable. The name of the reduction acts as this reduction variable and can be used in the conditional statement of a `check ... every` clause. Legal reduction operators are '`+`', '`*`','`max`', and '`min`'.

```
reduction max_diff max {
  [0:dim1-1][0:dim0-1] : fabs([1]a[0][0] - [0]a[0][0]);
}
```

Figure 3.6: Reduction written in SDSL.

A simple reduction is shown in Figure 3.6. This reduction uses a reduction variable named `max_diff` to store the largest absolute difference between a value of an element of `a` at timesteps 0 and 1.

Chapter 4

Embedded SDSL

It is possible to embed one or more SDSL code sections in any C/C++ or MATLAB code.

4.1 Embedding SDSL in C/C++

Figure 4.1 shows the running SDSL example embedded in C code. The embedded SDSL code is located between the pragmas at lines 9 and 26. We discuss three aspects of the embedding: SDSL pragmas, parameter / array / grid data passing, and return values.

4.1.1 `#pragma sdsl`

Every embedded section of SDSL begins with `#pragma sdsl begin` and is terminated by `#pragma sdsl end`. Optionally, this pragma may contain a `gpu()` clause specifying parameters used by the OverTile [4] backend for CUDA C code generation.

- `block:<comma separated list of integers>`: A comma separated list of integers specifying the GPU thread block size for generated code. There must be one integer in the list for each spatial dimension of the grid in the SDSL code.
- `tile:<comma separated list of integers>`: A comma separated list of integers specifying the number of elements to be computed per thread. There must be one integer in the list for each spatial dimension of the grid in the SDSL code.
- `time:<integer>`: A single literal integer value specifying the time tile size in the generated code.

4.1.2 Parameter and Array Passing

Both parameters and arrays can be passed from host C/C++ code to embedded SDSL.

- **Parameters**: `float`, `double`, `int`, and `long` parameters may be passed from host code to SDSL code by (1) declaring the parameter in the embedded SDSL code (as

```
#define SIZE (256)

int main() {
  float a[SIZE][SIZE];
  int dim0 = SIZE;
  int dim1 = SIZE;
  float c1 = 0.2f;

  #pragma sdsl begin gpu(block:64,8 tile:1,8 time:10)
    int dim0;
    int dim1;
    float c1;

    grid g[dim1][dim0];
    float griddata a on g at 0,1;

    pointfunction five_pt(x) {
      [1]x[0][0] = c1*([0]x[-1][0] + [0]x[0][-1] + [0]x[0][0] + [0]x[0][1] + [0]x[1][0]);
    }

    iterate 1000 {
      stencil j5p {
        [1:dim1-2][1:dim0-2] : five_pt(a);
      }
    }
  #pragma sdsl end

  return sdsl_return_0;
}
```

Figure 4.1: SDSL code for a 2D Jacobi stencil embedded in C code.

described in Section 3.2.5) and (2) declaring and defining a variable with an identical name and identical type in the host C/C++ code.

- **Arrays**: `float`, `double`, `int`, and `long` arrays may be passed from host code to SDSL code by (1) declaring `griddata` in the SDSL code (as described in Section 3.2.7) and (2) declaring an array with an identical name, identical type, and identical size.

The example in Figure 4.1 passes the parameters `dim0`, `dim1`, and `c1`. The array `a` is also passed.

4.1.3 Return Values

Each section of embedded SDSL bounded by `#pragma sdsl begin` and `#pragma sdsl end` creates an int variable in the host code that contains the exit code of the SDSL program. The exit code will be 0 for successful execution and non-zero for error conditions. Exit code values cannot currently be defined in SDSL code.

The host variable used to hold the exit code is named `sdsl_return_<index>` where `<index>` is the position of the embedded SDSL section in host code relative to other embedded SDSL sections, where the first SDSL section, syntactically, in a source file has `<index> = 0`, the second has `<index> = 1`, etc.

4.2 Embedding SDSL in MATLAB

Many programs for scientific computation and visualization are written in MATLAB, the license for which can be obtained from `http://www.mathworks.com`. MATLAB provides a

simple programming environment, but running MATLAB code is often time-consuming. A general technique to accelerate MATLAB is using the MEX interface to execute optimized native code.

Compute-intensive stencil code in a MATLAB program can be rewritten in SDSL. The resulting MATLAB code can be compiled by `sdslc` to generate a MEX source file containing a C or CUDA implementation of the SDSL functions and necessary MEX glue code. The generated MEX function can be independently optimized for different architectures.

To illustrate the integration of SDSL into MATLAB, a MATLAB code for denoising (`riciandenoise.m`), and its SDSL-integrated equivalent version (`riciandenoise_sdsl.m`) are made available in `SDSLC_INSTALL_DIR/share/sdslc/examples/matlab`. The loop at lines 70–109 in `riciandenoise.m` represents a stencil computation which can be better optimized if represented in SDSL. Since all the arrays used within the loop will be passed as an argument to the MEX function, such arrays must already appear before the definition of the SDSL stencil in the SDSL-integrated MATLAB code.

The MATLAB code in `ricianenoise.m` uses a 2d-array `ulast` to create a copy of the array `u`. In SDSL, however, the array `u` is defined on timesteps 0,1 as:

`double griddata u on gr at 0,1;`

In SDSL, references to `ulast` can be replaced by references to `[0]u[][]`. Hence, there is no explicit declaration for `ulast` in the SDSL code.

While writing the SDSL stencil, the grid dimensions must be declared first (starting with the fastest varying dimension first), followed by the declaration of parameters in the surrounding host program. For example, in `riciandenoise_sdsl.m`, the SDSL stencil begins with the declaration:

```
1    int N2;
2    int N1;
3
4    double dt;
5    ...
6    double gamma;
7
8    grid gr[N1][N2];
9    ...
```

The 2D grid `gr` is defined to be of dimensions `N1` $\times$ `N2`, with `N2` being the fastest varying dimension. Hence, `N2` is declared first, followed by N1. If the grid was 3-dimensional (`N1` $\times$ `N2` $\times$ `N3`), then `N3` would have been declared before `N2`. After declaring all the dimensions, we declare the other scalars in the host program (`dt`, `epsilon`, ..., `gamma`). Then we declare the grid, and lastly the griddata on the grid at different timesteps. SDSL code embedded in MATLAB is defined by the same syntax and semantics as SDSL code embedded in C/C++. See Chapter 3 for a complete description.

Chapter 5

SDSL Compiler Reference

This chapter provides details on the operation of the SDSL source-to-source compiler, located at `$SDSLC_INSTALL_DIR/bin/sdslc`.

5.1 Common Case Usage

5.1.1 SDSL embedded in C/C++

In general, the SDSL compiler will take an input C/C++ file with one or more embedded SDSL sections and convert this to an output C/C++/CUDA file using one of four backends:

- `affine-c`: produces "plain C" code with an affine structure suitable for subsequent polyhedral analyses and transformations
- `overtile`: produces CUDA code that is optimized with overlapped tiling.
- `nest-split-dlt`: produces C code optimized with nested split-tiling and DLT.
- `hyb-split-dlt`: produces C code optimized with hybrid split-tiling and DLT.

Example command line invocations for each backend are shown below:

```
$ sdslc -b affine-c -f example.sdsl.c -o example.affine.c
$ sdslc -b overtile -f example.sdsl.c -o example.overtile.cu
$ sdslc -b nest-split-dlt -f example.sdsl.c -o example.nested.c
$ sdslc -b hyb-split-dlt -f example.sdsl.c -o example.hybrid.c
```

In all examples the same input source file, `example.sdsl.c`, is used. The first invocation produces a C file named `example.affine.c` using the affine C backend; the second produces a CUDA file named `example.overtile.cu` using the OverTile backend; the third produces `example.nest.c` using the nested split-tiling DLT backend; and the fourth produces `example.hybrid.c` using the hybrid split-tiling DLT backend.

The nested and hybrid split-tiled variants of the code use parametric tile sizes that can be tuned at run time for optimum performance on a given platform. By default, SSE2 vector intrinsics are used when generating split-tiled code, although this can be changed with the `-i` flag to `sdslc`. See Section 5.2 for details.

5.1.2 SDSL embedded in MATLAB

SDSL code can also be embedded in MATLAB source and optimized with the `sdslc` compiler. Like C/C++ codes, four backends are available.

- `maffine-c`: produces "plain C" code with an affine structure suitable for subsequent polyhedral analyses and transformations
- `movertile`: produces CUDA code that is optimized with overlapped tiling.
- `mnest-split-dlt`: produces C code optimized with nested split-tiling and DLT.
- `mhyb-split-dlt`: produces C code optimized with hybrid split-tiling and DLT.

Example command line invocations for each backend are shown below:

```
$ sdslc -b maffine-c -f example.sdsl.m -o example.affine.m
$ sdslc -b movertile -f example.sdsl.m -o example.overtile.m
$ sdslc -b mnest-split-dlt -f example.sdsl.m -o example.nested.m
$ sdslc -b mhyb-split-dlt -f example.sdsl.m -o example.hybrid.m
```

Note that the output filename given is the name of the generated MATLAB file name. This file will include one or more MEX calls to a function defined in a file named `sdsl_program_<hexid>.c` in the same directory. The `<hexid>` component of the filename **will not change** when different backends are selected. Practically speaking, this means that any invocation of `sdslc` on MATLAB files, int the same directory, with identical SDSL sections, will result in the `sdsl_program_<hexid>.c` file being overwritten.

5.2 Command Line Options

The SDSL compiler, `sdslc`, is controlled with a number of command line options, listed below. Each option has equivalent short (e.g. `-x`) and long (e.g. `--extern-c`) forms.

- `-b <arg>, --backend <arg>`
  Set backend code generator as '`affine-c`' (default), '`overtile`', '`nest-split-dlt`', '`hyb-split-dlt`', '`maffine-c`', '`movertile`', '`mnest-split-dlt`', or '`mhyb-split-dlt`'.
- `-d, --detailed-timing`
  Time copy in, compute, copy out, and other overhead in generated code. (split-tile backends only)

- `-f <arg>, --in <arg>`
  Embedded SDSL input filename.

- `-g --debug`
  Enable debug statements in generated code.

- `-h, --help`
  Print help message and exit.

- `-i <arg>, --isa <arg>`
  Set vector ISA as '`sse2` (default)', '`sse4`', '`avx`', '`xphi`' (experimental), or '`arm`' (experimental). (split-tile backends only)

- `-l, --legacy-gpu`
  Generate code for NVIDIA GT2xx and Tesla 10xx GPU. (OverTile backend only)

- `-o <arg>, --out <arg>`
  Output filename.

- `-p, --pointer-swap`
  Use higher performance pointer swaps instead of explicit array copies. (split-tile backends only)

- `-s, --sink-reg-loop`
  Sink outermost "regular" intertile loop to innermost intertile loop level (experimental). (split-tile backends only)

- `-v, --verbose`
  Output detailed progress

- `-x, --extern-c`
  Use C linkage for wrapper function. This option will place the `extern` keyword in front of the C function that wraps generated code and should be used when SDSL code is embedded in C++. This option prevents link errors associated with C++ name mangling.

5.3 Running the Example Benchmarks

The SDSL compiler provides a number of example codes located in subdirectories of the `$SDSLC_INSTALL_DIR/share/sdslc/examples` directory.

5.3.1 CDSC Pipeline

3 stages of the CDSC medical imaging pipeline (denoise, registration, and segmentation) are included in the `$SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc` directory. Also included is a program that executes the pipeline. A `Makefile` is provided to build both affine C and CUDA versions of the codes. The following commands will build each pipeline stage and the pipeline application. [1]

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc
$ make denoise
$ make register
$ make segment
$ make pipeline
```

These commands produce affine C++ code, overtile CUDA code, and executables. The affine versions of the pipeline stages and pipeline application can be executed with the following commands:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc
$ ./denoise-dp-affine
$ ./register-dp-affine
$ ./segment-dp-affine
$ ./pipeline-sp-affine
```

Similarly, the overtile versions can be executed with the following commands:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc
$ ./denoise-dp-overtile
$ ./register-dp-overtile
$ ./segment-dp-overtile
$ ./pipeline-sp-overtile
```

The `*-dp-*` codes are double precision; the `*-sp-*` codes are single precision.

5.3.2 General Benchmarks

A number of general stencil kernels with embedded SDSL implementations are included in subdirectories of the `$SDSLC_INSTALL_DIR/share/sdslc/examples/general` directory. All codes have both single and double precision versions and both affine C and overtile versions can be compiled. To build all versions of all benchmarks, execute the following commands:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/general
$ make affine
$ make overtile
```

[1] Users of Geforce GTX2xx or Tesla C10xx GPU must change the file `$SDSLC_INSTALL_DIR/share/sdslc/examples/common.mk` to add the `--legacy-gpu` option to `SDSLC_FLAGS`.

Alternatively, each benchmark can be built individually:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/general/<benchmark>
$ make affine
$ make overtile
```

The benchmark codes may then be executed by executing the following commands:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/general/<benchmark>
# Affine single and double precision
$ ./<benchmark>-sp-affine
$ ./<benchmark>-dp-affine
# Overtile single and double precision
$ ./<benchmark>-sp-overtile
$ ./<benchmark>-dp-overtile
```

Each benchmark computes a reference CPU version followed by the SDSL version. A correctness check is then performed.

Chapter 6

SDSL Autotuner Reference

The SDSL compiler package includes a very basic script for autotuning overtile thread block and tile sizes. This script can be found at `$SDSLC_INSTALL_DIR/bin/autotune-overtile.py`. Ranges for thread block and tile sizes, along with `sdslc` and `nvcc` compiler flags are provided to the script via a configuration file provided by the user. This chapter provides usage instructions for the autotuner along with instructions for building a configuration file and instructions for autotuning the included benchmarks.

6.1 Common Case Usage

The autotune script is invoked via the python interpreter as shown below:

```
$ python $SDSLC_INSTALL_DIR/bin/autotune-overtile.py autotune.conf test.c
```

Two arguments are expected by the script. The first is the name of the configuration file (described in Sec. 6.2) and the second is the name of the embedded SDSL file to be autotuned.

Running the command shown above will generate files with varying thread block and tile size combinations specified in the `gpu` clause of any `#pragma sdsl begin` statements found in `test.c`. Each of these files is translated to CUDA C the OverTile backend of `sdslc`, compiled by `nvcc`. The resulting binary is executed and timed[1], as shown below:

```
$ python $SDSLC_INSTALL_DIR/bin/autotune-overtile.py autotune.conf test.c
Starting autotuning, see 'test.c.autotune.log' for best results...
#pragma sdsl begin gpu(block:4,4,4 tile:1,1,1 time:1)
1.22 sec
#pragma sdsl begin gpu(block:4,4,4 tile:1,1,2 time:1)
1.18 sec
.
.
#pragma sdsl begin gpu(block:1024,1024,1024 tile:8,8,8 time:1)
```

[1]Timing information is generated by the Unix `time` command-line utility

```
INVALID PARAMETERS AND/OR ERROR
.
.
#pragma sdsl begin gpu(block:4,4,4 tile:1,1,1 time:2)
1.06 sec
.
.
Finished autotuning test.c
```

During execution, the SDSL pragma associated with a combination of parameters will be printed to the console, followed by the total time taken to run the input file. There sometimes may be a message stating `INVALID PARAMETERS AND/OR ERROR` in place of timing information. This is common and reflects situations where specified parameters do not match the capabilities of the current GPU or lead to other errors that result in a non-zero return value from the original program.

At any point during autotuning, the fastest code discovered can be found in the files `<base>.autotuned.sdsl.<extension>` (SDSL version) and `<base>.autotuned.cu` (Over-Tile version). In the above example, these files would be `test.autotuned.sdsl.c` and `test.autotuned.cu`.

6.2 Configuration File Format

The autotune script requires a configuration file provided by the user. A configuration file for a 2D SDSL program is shown below:

```
# sdslc flags
-b overtile --legacy-gpu

# nvcc flags
-O3 -arch=sm_20 -Xcompiler -O3 -I ../include -DNOREF

# Space dims
2

# Thread block sizes <start end step>
8 512 8
2 64 2

# Space tile sizes <start end step>
8 256 8
2 64 2

# Time tile sizes <start end step>
```

```
1 50 2
```

Comments start with a '#' and extend to the end of the line. Blank lines are ignored. The first non-comment, non-blank line is the command-line flags passed to `sdslc`. The next line are the command-line flags passed to `nvcc`. After this is the integer number of space dimensions of the grid used in the SDSL code to be autotuned.

The next group of lines are the thread block sizes. There must be one line for each spatial dimension. These are composed of three integers: the thread block size to start autotuning, the thread block size to end autotuning, and the step the thread block size is incremented by. In the example above, the line '`8 512 8`' would run the input file with thread block sizes of $8, 16, 24, ..., 512$ for the first spatial dimension. Space tile sizes are specified in the same way as thread block sizes, with one line per spatial dimension. The final line specifies time tile size.

6.3 Embedded SDSL File Restrictions

The embedded SDSL file passed to the autotuner must, on exit, return integer `0` for success and a non-zero integer value for failure. It is suggested that the return value of embedded SDSL sections (described in Sec. 4.1.3) is used to generate this return value.

If the embedded SDSL file passed to the autotuner contains multiple SDSL sections each one will be tuned using the same configuration file. This means that all sections must operate on a grid of the same dimensionality. Further, all sections will, for any run, have identical thread block and tile sizes. Because of this, we recommend breaking files with multiple SDSL sections into files with single SDSL sections for autotuning.

6.4 Autotuning the Example Benchmarks

Configuration files and `make` targets are provided for all of the benchmarks described in Sec. 5.3. The following two sections describe how to autotune the included example benchmarks.

6.4.1 CDSC Pipeline

The CDSC medical imaging pipeline benchmarks can be autotuned with the following commands.

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc
$ make autotune-denoise
$ make autotune-register
$ make autotune-segment
$ make autotune-pipeline
```

The `make autotune-<benchmark>` command will start the `autotune-overtile.py` script for the specified benchmark. The first 3 benchmarks, `denoise`, `register`, and `segment`, are configured with the `autotune-3d.conf` file in the CDSC examples directory. The last benchmark, `pipeline`, is configured with the `autotune-pipeline.conf` file in the same directory.

6.4.2 General Benchmarks

Each general benchmark can be autotuned with the following commands:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/general/<benchmark>
$ make autotune-sp
$ make autotune-dp
```

All benchmarks are configured with one of the `autotune-1d.conf`, `autotune-2d.conf`, or `autotune-3d.conf` files located in the `$SDSLC_INSTALL_DIR/share/sdslc/examples/general` directory.

Chapter 7

Detailed Example: Rician Denoise

This chapter contains a detailed example of porting a non-trivial stencil, Rician denoising [1], from C to embedded SDSL. Rician denoising is used to remove a particular type of noise commonly found in MRI images and is one step in the CDSC medical imaging pipeline. The complete source code of the example, including the C implementation, SDSL implementation, driver, and correctness check, can be found in the file `$SDSL_INSTALL_DIR/share/sdslc/examples/cdsc/denoise-dp.sdsl.cpp`.

7.1 C Reference Code

A C function implementing Rician denoising is shown in Fig. 7.1. In this section we describe the important features of the code; in subsequent sections we port this C code to an equivalent SDSL version.

7.1.1 Function Signature

The Rician denoise function signature (line 1) is shown below:

```
int rician3d(double *U, double *F, double sigma, double lambda,
             int dim0, int dim1, int dim2)
```

The `rician3d` function returns an integer status and takes the following parameters:

- `double *U`: Pointer to the denoised image (output)
- `double *F` ($0 \leq$ `F[i][j][k]` ≤ 1): Pointer to the original noisy image (input)
- `double sigma` (> 0): Parameter indicating type of Rician noise expected (input)
- `double lambda` (≥ 0): Controls denoising strength, smaller `lambda` implies stronger denoising (input)

```
int rician3d(double *U, double *F, double sigma, double lambda, int dim0, int dim1, int dim2) {
  // U0, G are local arrays
  double *U0 = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  // Constants
  const double DT = 5.0;
  const double EPSILON = 1.0E-20;
  const double sigma2 = sigma*sigma;
  const double gamma = lambda/sigma2;

  // Convergence
  const int max_iter = 50;
  const double TOL = 0.00001;
  bool converged;

  // Macro for 3d indexing
  #define idx(z,y,x) ((x) + (y)*dim0 + (z)*dim0*dim1)
  /* Gradient descent loop */
  for (int t = 0; t < max_iter; ++t) {
    /* Copy U to U0 */
    for (int i = 0; i < dim2; ++i) {
      for (int j = 0; j < dim1; ++j) {
        for (int k = 0; k < dim0; ++k) {
          U0[idx(i,j,k)] = U[idx(i,j,k)];
        }}}
    /* Approximate G = 1/|grad U| */
    for (int i = 1; i < dim2 - 1; ++i) {
      for (int j = 1; j < dim1 - 1; ++j) {
        for (int k = 1; k < dim0 - 1; ++k) {
          G[idx(i,j,k)] = 1.0/sqrt(EPSILON +
          (U0[idx(i,j,k)] - U0[idx(i,j+1,k)])*(U0[idx(i,j,k)] - U0[idx(i,j+1,k)]) +
          (U0[idx(i,j,k)] - U0[idx(i,j-1,k)])*(U0[idx(i,j,k)] - U0[idx(i,j-1,k)]) +
          (U0[idx(i,j,k)] - U0[idx(i,j,k+1)])*(U0[idx(i,j,k)] - U0[idx(i,j,k+1)]) +
          (U0[idx(i,j,k)] - U0[idx(i,j,k-1)])*(U0[idx(i,j,k)] - U0[idx(i,j,k-1)]) +
          (U0[idx(i,j,k)] - U0[idx(i+1,j,k)])*(U0[idx(i,j,k)] - U0[idx(i+1,j,k)]) +
          (U0[idx(i,j,k)] - U0[idx(i-1,j,k)])*(U0[idx(i,j,k)] - U0[idx(i-1,j,k)]));
        }}}
    /* Update U by a semi-implict step */
    converged = true;
    for (int i = 1; i < dim2 - 1; ++i) {
      for (int j = 1; j < dim1 - 1; ++j) {
        for (int k = 1; k < dim0 - 1; ++k) {
          /* Evaluate r = I1(U*F/sigma^2) / I0(U*F/sigma^2)
             with a cubic rational approximation. */
          double r = U0[idx(i,j,k)]*F[idx(i,j,k)]/sigma2;
          r = ( r*(2.38944 + r*(0.950037 + r)) )
          / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
          /* Update U */
          U[idx(i,j,k)] = (U0[idx(i,j,k)] +
                           DT*(U0[idx(i,j+1,k)]*G[idx(i,j+1,k)] +
                               U0[idx(i,j-1,k)]*G[idx(i,j-1,k)] +
                               U0[idx(i,j,k+1)]*G[idx(i,j,k+1)] +
                               U0[idx(i,j,k-1)]*G[idx(i,j,k-1)] +
                               U0[idx(i+1,j,k)]*G[idx(i+1,j,k)] +
                               U0[idx(i-1,j,k)]*G[idx(i-1,j,k)] +
                               gamma*F[idx(i,j,k)]*r))
                          /
                           (1.0 + DT*(G[idx(i,j+1,k)] + G[idx(i,j-1,k)] +
                                      G[idx(i,j,k+1)] + G[idx(i,j,k-1)] +
                                      G[idx(i+1,j,k)] + G[idx(i-1,j,k)] +
                                      gamma));
          if (fabs(U0[idx(i,j,k)] - U[idx(i,j,k)]) > TOL) {
            converged = false;
          }
        }}}
    if (converged) {
      break;
    }
  }
  #undef idx
  free(U0);
  free(G);
  return 0;
}
```

Figure 7.1: Rician denoising, Original C implementation.

- `int dim0`, `int dim1`, `int dim2`: Dimension sizes of `U` and `F` (e.g. `U[dim2][dim1][dim0]`) (input)

We will create a drop-in replacement for this function in C and SDSL.

7.1.2 Arrays

In addition to the arrays passed as parameters, temporary arrays `U0` and `G` are allocated on lines 3 and 4. Array `U0` is used to hold intermediate values of the denoised image `U` in a Jacobi iteration. Array `G` is used to hold the result of a curvature approximation as described in [1]. The output image `U` is initialized to the value of the input image `F` on line 11. All arrays are 1D data structures interpreted as 3D data structures and are accessed with the indexing macro defined on line 22.

7.1.3 Constants and Variables

Lines 11–14 in Fig. 7.1 contain the declarations of a number of constants. `DT`, `EPSILON`, `sigma2`, and `gamma` are used throughout the denoise calculation. The purpose of these variables is beyond the scope of this document; please see [1] for more details.

Lines 17–19 contain declarations and definitions that control how many times the denoise operation will run. `max_iter` is the maximum number of iterations the denoise operation can run. `TOL`, defined on line 18, is used in the convergence check on line 67. Semantically, `TOL` is the maximum absolute difference allowed between `U[i][j][k]` and `U0[i][j][k]` for all $0 <$ `i` $<$ `dim2`, $0 <$ `j` $<$ `dim1`, $0 <$ `k` $<$ `dim0` for the denoise operation to converge. Whether or not the denoising has converged is tracked by the boolean variable `converged`, defined on line 19.

7.1.4 Main Iterative Loop

The main loop nest for Rician denoising runs from lines 24–74 in Fig. 7.1. The outer loop runs, at most, for `max_iter=50` iterations, and has four phases:

- `U` copy (lines 26–30) — Copies `U` to `U0` temp array
- `G` stencil (lines 32–42) — Computes value for `G` using a stencil on `U0`
- `U` stencil (lines 45–70) — Computes value for `U` using a stencil on `U0`, `G`, and `F`
- Convergence check (lines 71–73) — Breaks out of the main loop if `U` converged

The `U` copy phase simply copies the contents of array `U` to `U0` as part of the Jacobi iteration. The `G` stencil, `U` stencil, and convergence check are described in Sec. 7.1.5, Sec. 7.1.6, and Sec. 7.1.7, respectively.

7.1.5 `G` Stencil

The `G` stencil computes values for all elements of the `G` array except the first and last elements in each dimension. Values for array element `G[i][j][k]` are computed with a 7-point 3D stencil on array `U0` using a center element (`U0[i][j][k]`) and its neighbors along each spatial dimension (`U0[i-1][j][k]`, `U0[i+1][j][k]`, `U0[i][j-1][k]`, `U0[i][j+1][k]`, `U0[i][j][k-1]`, and `U0[i][j][k+1]`). While the loop body statement appears to use substantially more than 7 points, close examination reveals that there are six subexpressions of the form $(center - neighbor)^2$ using each of the 7 stencil points multiple times.

7.1.6 `U` Stencil

The `U` stencil computes values for all elements of the `U` array except the first and last elements in each dimension. For each element of `U[i][j][k]` a scalar value `r` is computed using `U0[i][j[k]` and `F[i][j[k]` at lines 50–52. This is followed by an update of `U` at lines 54–66 using a 7-point 3D stencil on `U0`. This stencil is the same "center and neighbors in each dimension" pattern as described in Sec. 7.1.5. The `U` update also uses a 6-point 3D stencil on `G` that is the same shape as above without the center point. After each element of `U` is computed, at lines 67–69 the result is compared to the previous value of `U` (stored in `U0`) to check for convergence.

7.1.7 Convergence Check

At lines 71–73 the `converged` flag, set at lines 44 and 68, is checked. If the absolute difference between successive values of `U` at any given element was greater than `TOL` then the converge flag will be set to false at line 68 and the main loop will continue executing. If, however, absolute difference for *every* computed value of `U` and its predecessor in `U0` was less than `TOL` the `converged` flag is never set to false and the calculation is considered to be converged. Control flow breaks from the main loop at line 72 and proceeds to cleanup.

7.2 Creating a C/SDSL Skeleton Function

We wish to create a drop-in replacement for the Rician denoise C function described in Sec. 7.1. We begin by creating a thin C wrapper function around an empty `#pragma sdsl` as shown in Fig. 7.2. The function from Fig. 7.1 is renamed to `sdsl_rician3d` and its signature

```
1 int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
2                   int dim0, int dim1, int dim2) {
3   #pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
4   #pragma sdsl end
5   return 0;
6 }
```

Figure 7.2: C/SDSL skeleton function

is preserved. Begin and end SDSL pragmas are added, with the begin pragma including the optional `gpu(...)` clause to specify options for OverTile codegen. We arbitrarily set thread block sizes at 4 in each dimension, spatial tile sizes at 4 in each dimension, and time tile size at 1. These parameter values will be autotuned in Sec. 7.7.

7.3 Passing Data from C to SDSL

With the skeleton in place, we need to pass parameters and arrays from C to SDSL.

7.3.1 Parameters and Constants

We begin by declaring SDSL parameters for each of the function parameters as shown in Fig. 7.3. Lines 5–9 contain SDSL parameter declarations binding the declared parameters to identically named C variables.[1] Lines 12–16 declare and define a number of constants in SDSL that are *not* passed in from C.

```
1  int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
2                    int dim0, int dim1, int dim2) {
3    #pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
4    // Parameters
5    int dim0;
6    int dim1;
7    int dim2;
8    double sigma;
9    double lambda;
10
11   // Constants
12   double DT = 5.0;
13   double EPSILON = 1.0E-20;
14   double sigma2 = sigma*sigma;
15   double gamma = lambda/sigma2;
16   double TOL = 0.00001;
17   #pragma sdsl end
18   return 0;
19 }
```

Figure 7.3: Parameters passed from C to SDSL, constants defined in SDSL.

7.3.2 Arrays

The SDSL source with all code to pass the example arrays is shown in Fig. 7.4. Passing arrays from C to SDSL requires both C and SDSL code. In C, on lines 4–5 we declare the temporary array `G` and initialize its contents to 0. Line 8 initializes the contents of `U`, the denoised output image, with the contents of `F`, the noisy input image. Note that `U0` is not declared or defined in C.

In SDSL, line 26 declares a `grid g` with with the same dimension sizes as the C arrays being passed. Lines 28–30 contain SDSL `griddata` declarations for `U`, `G`, and `F`. `U` is defined on 2 timesteps, representing arrays `U0` and `U` from the original C code. `G` and `F` are defined on 1 timestep.

[1] Parameters `dim0`, `dim1`, and `dim2` must be declared first and in that order since they will be used as dimensions of a grid (shown in Fig. 7.4). See Sec.8.3 for more details.

```
1  int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
2                    int dim0, int dim1, int dim2) {
3    // G is a local array
4    double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
5    memset(G, 0, sizeof(double)*dim0*dim1*dim2);
6
7    // Copy F to U
8    memcpy(U, F, dim0*dim1*dim2*sizeof(double));
9
10   #pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
11   // Parameters
12   int dim0;
13   int dim1;
14   int dim2;
15   double sigma;
16   double lambda;
17
18   // Constants
19   double DT = 5.0;
20   double EPSILON = 1.0E-20;
21   double sigma2 = sigma*sigma;
22   double gamma = lambda/sigma2;
23   double TOL = 0.00001;
24
25   // Grid and griddata
26   grid g[dim2][dim1][dim0];
27
28   double griddata U on g at 0,1;
29   double griddata G on g at 0;
30   double griddata F on g at 0;
31   #pragma sdsl end
32   free(G);
33   return 0;
34 }
```

Figure 7.4: Arrays passed from C to SDSL.

7.4 Implementing `pointfunctions`

For Rician denoise we use separate `pointfunction`s to perform the stencil computations that produce elements of `G` and `U`. The next two sections describe the SDSL used to implement these stencil computations.

7.4.1 `approx_g`

The `approx_g` `pointfunction` is shown on lines 32–40 of Fig. 7.5. This `pointfunction` implements the "`G` stencil" described in Sec. 7.1.5. At line 32 the `pointfunction` is declared as taking two `griddata` parameters, `u` and `g`. Lines 33–39 contain a single statement that calculates a point of `g` using neighbor points from `u`.

Note that the `griddata` references in this statement address points via offsets from the current timestep and point, e.g. `[0]u[-1][0][0]` is the value of `u` at the current timestep (`U0` from the original) offset from the current point by -1, 0, and 0 in the three spatial dimensions. Further note that the statement uses the `sqrt()` function from `math.h`, thereby taking advantage of `math.h` function availability in SDSL. Finally, note that the constant `EPSILON` is used without including it as a parameter. This is required; all parameters to a `pointfunction` *must* be `griddata`. All constants and parameters are available within a `pointfunction`.

```
1  int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
2                    int dim0, int dim1, int dim2) {
3    // G is a local array
4    double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
5    memset(G, 0, sizeof(double)*dim0*dim1*dim2);
6
7    // Copy F to U
8    memcpy(U, F, dim0*dim1*dim2*sizeof(double));
9
10   #pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
11   // Parameters
12   int dim0;
13   int dim1;
14   int dim2;
15   double sigma;
16   double lambda;
17
18   // Constants
19   double DT = 5.0;
20   double EPSILON = 1.0E-20;
21   double sigma2 = sigma*sigma;
22   double gamma = lambda/sigma2;
23   double TOL = 0.00001;
24
25   // Grid and griddata
26   grid g[dim2][dim1][dim0];
27
28   double griddata U on g at 0,1;
29   double griddata G on g at 0;
30   double griddata F on g at 0;
31
32   pointfunction approx_g(u,g) {
33     [0]g[0][0][0] = 1.0 / sqrt(EPSILON +
34                           ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) +
35                           ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) +
36                           ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) +
37                           ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) +
38                           ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) +
39                           ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) );
40   }
41   #pragma sdsl end
42 free(G);
43   return 0;
44 }
```

Figure 7.5: Point function to calculate values of `G`.

7.4.2 `update_u`

The SDSL highlighted at lines 42–62 of Fig. 7.6 corresponds to the "`U` stencil" described in Sec. 7.1.6. This `pointfunction` is defined starting at line 42 and takes three `griddata` parameters, `u`, `g`, and `f`. Lines 44–45 compute the local variable `r`, while lines 46–61 are a single statement used to compute the value of the current point of `u` at the next timestep, `[1]u[0][0][0]`.

7.5 Defining the Iterate

Fig. 7.7 shows the complete replacement for the reference Rician denoise code. Lines 64–74 define an `iterate` that implements the main iterative loop control described in Sec. 7.1.4. Line 64 specifies that the iterate runs for *at most* 50 iterations.[2] Subsequent sections describe the stencils, reduction, and convergence check used in the iterate.

[2]The literal integer value '50' is used to specify the number of maximum iterations instead of a constant or parameter, e.g. `max_iter`. This is due to a known limitation with the current SDSL implementation noted in Sec. 8.1.

```
1  int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
2                    int dim0, int dim1, int dim2) {
3    // G is a local array
4    double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
5    memset(G, 0, sizeof(double)*dim0*dim1*dim2);
6
7    // Copy F to U
8    memcpy(U, F, dim0*dim1*dim2*sizeof(double));
9
10   #pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
11   // Parameters
12   int dim0;
13   int dim1;
14   int dim2;
15   double sigma;
16   double lambda;
17
18   // Constants
19   double DT = 5.0;
20   double EPSILON = 1.0E-20;
21   double sigma2 = sigma*sigma;
22   double gamma = lambda/sigma2;
23   double TOL = 0.00001;
24
25   // Grid and griddata
26   grid g[dim2][dim1][dim0];
27
28   double griddata U on g at 0,1;
29   double griddata G on g at 0;
30   double griddata F on g at 0;
31
32   pointfunction approx_g(u,g) {
33     [0]g[0][0][0] = 1.0 / sqrt(EPSILON +
34                           ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) +
35                           ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) +
36                           ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) +
37                           ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) +
38                           ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) +
39                           ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) );
40   }
41
42   pointfunction update_u(u,g,f) {
43     double r = [0]u[0][0][0]*[0]f[0][0][0]/sigma2;
44     r =( r*(2.38944 + r*(0.950037 + r)) )
45          / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
46     [1]u[0][0][0] =   ([0]u[0][0][0] +
47                        DT*([0]u[ 0][ 1][ 0]*[0]g[ 0][ 1][ 0] +
48                            [0]u[ 0][-1][ 0]*[0]g[ 0][-1][ 0] +
49                            [0]u[ 0][ 0][ 1]*[0]g[ 0][ 0][ 1] +
50                            [0]u[ 0][ 0][-1]*[0]g[ 0][ 0][-1] +
51                            [0]u[ 1][ 0][ 0]*[0]g[ 1][ 0][ 0] +
52                            [0]u[-1][ 0][ 0]*[0]g[-1][ 0][ 0] +
53                            gamma*[0]f[0][0][0]*r))
54                       /
55                        (1.0 + DT*[0]g[ 0][ 1][ 0] +
56                                   [0]g[ 0][-1][ 0] +
57                                   [0]g[ 0][ 0][ 1] +
58                                   [0]g[ 0][ 0][-1] +
59                                   [0]g[ 1][ 0][ 0] +
60                                   [0]g[-1][ 0][ 0] +
61                                   gamma);
62   }
63   #pragma sdsl end
64   free(G);
65   return 0;
66 }
```

Figure 7.6: Point function to calculate values of `U`.

7.5.1 Stencils

The `iterate` contains two `stencils`. `stencil gs`, defined on lines 65–67, corresponds to the code described in Sec. 7.1.5; `stencil us`, defined on lines 68–70, corresponds to the loop nest described in Sec. 7.1.6. Each stencil defines a single region over which it is applied, `[1:dim2-2][1:dim1-2][1:dim0-2]`. This region corresponds to the spatial loop bounds of the original `G` and `U` stencils located at lines 32–34 and 45–47 of the original

```
int sdsl_rician3d(double *U, double *F, double sigma, double lambda,
                  int dim0, int dim1, int dim2) {
  // G is a local array
  double *G = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  memset(G, 0, sizeof(double)*dim0*dim1*dim2);

  // Copy F to U
  memcpy(U, F, dim0*dim1*dim2*sizeof(double));

  #pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
  // Parameters
  int dim0;
  int dim1;
  int dim2;
  double sigma;
  double lambda;

  // Constants
  double DT = 5.0;
  double EPSILON = 1.0E-20;
  double sigma2 = sigma*sigma;
  double gamma = lambda/sigma2;
  double TOL = 0.00001;

  // Grid and griddata
  grid g[dim2][dim1][dim0];

  double griddata U on g at 0,1;
  double griddata G on g at 0;
  double griddata F on g at 0;

  pointfunction approx_g(u,g) {
    [0]g[0][0][0] = 1.0 / sqrt(EPSILON +
                           ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][ 1][ 0]) +
                           ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) * ([0]u[0][0][0] - [0]u[ 0][-1][ 0]) +
                           ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][ 1]) +
                           ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) * ([0]u[0][0][0] - [0]u[ 0][ 0][-1]) +
                           ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[ 1][ 0][ 0]) +
                           ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) * ([0]u[0][0][0] - [0]u[-1][ 0][ 0]) );
  }

  pointfunction update_u(u,g,f) {
    double r = [0]u[0][0][0]*[0]f[0][0][0]/sigma2;
    r =( r*(2.38944 + r*(0.950037 + r)) )
          / ( 4.65314 + r*(2.57541 + r*(1.48937 + r)) );
    [1]u[0][0][0] =   ([0]u[0][0][0] +
                       DT*([0]u[ 0][ 1][ 0]*[0]g[ 0][ 1][ 0] +
                           [0]u[ 0][-1][ 0]*[0]g[ 0][-1][ 0] +
                           [0]u[ 0][ 0][ 1]*[0]g[ 0][ 0][ 1] +
                           [0]u[ 0][ 0][-1]*[0]g[ 0][ 0][-1] +
                           [0]u[ 1][ 0][ 0]*[0]g[ 1][ 0][ 0] +
                           [0]u[-1][ 0][ 0]*[0]g[-1][ 0][ 0] +
                           gamma*[0]f[0][0][0]*r))
                     /
                      (1.0 + DT*[0]g[ 0][ 1][ 0] +
                                 [0]g[ 0][-1][ 0] +
                                 [0]g[ 0][ 0][ 1] +
                                 [0]g[ 0][ 0][-1] +
                                 [0]g[ 1][ 0][ 0] +
                                 [0]g[-1][ 0][ 0] +
                                 gamma);
  }

  iterate 50 {
    stencil gs {
      [1:dim2-2][1:dim1-2][1:dim0-2] : approx_g(U,G);
    }
    stencil us {
      [1:dim2-2][1:dim1-2][1:dim0-2] : update_u(U,G,F);
    }
    reduction max_diff max {
      [1:dim2-2][1:dim1-2][1:dim0-2] : fabs([1]U[0][0][0] - [0]U[0][0][0]);
    }
  } check (max_diff < TOL) every 10 iterations
  #pragma sdsl end
  free(G);
  return 0;
}
```

Figure 7.7: Complete SDSL function contains iterate with stencils, reduction, and convergence check.

C implementation in Fig. 7.1. At each point of the defined region, both stencils call the appropriate `pointfunction`. At line 66 `approx_g` is called with `griddata` parameters `U` and `G`. At line 69 `update_u` is called with `griddata` parameters `U`, `G`, and `F`

7.5.2 `reduction`

The `reduction` defined at lines 71–73 of Fig. 7.7 is responsible for computing the absolute difference between successive values of `U` over all points in `[1:dim2-2][1:dim1-2][1:dim0-2]`. Instead of computing these differences per point as the `U` stencil is being executed, as described in Sec. 7.1.6, the `reduction` performs a separate sweep of the region and stores the maximum absolute difference in the reduction variable `max_diff` using the reduction operation `max`. The value of `max_diff` is subsequently used in the convergence check at line 74.

7.5.3 Convergence Check

The convergence check for the SDSL version of Rician denoise is defined at line 74 of Fig. 7.7. It specifies that if the reduction variable `max_diff` is less than the constant `TOL` the `iterate` should terminate. It is specified that this check occurs every 10 iterations.[3] By increasing the number of iterations between convergence checks we enable the `sdslc` compiler to perform aggressive time tiling optimizations. If the exact behavior of the original Denoise implementation was desired, we would modify the convergence check to occur `every 1 iterations`

7.6 Generating Code

The Rician denoise functions shown in Fig. 7.1 and Fig. 7.7 are combined with the driver shown in Fig. 7.8 and accessory functions for timing and correctness checks (not shown) to produce a complete program suitable for compilation and execution. This file can be found at `$SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc/denoise-dp.sdsl.cpp`, along with a Makefile for generating and building the code.

To manually generate affine C code from the SDSL source, the following command would be run:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc/
$ sdslc -b affine-c denoise-dp.sdsl.cpp -o denoise-dp.affine.cpp
```

To manually generate OverTile CUDA code from the SDSL source, the following command would be run:

```
$ cd $SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc/
$ sdslc -b overtile denoise-dp.sdsl.cpp -o denoise-dp.overtile.cu
```

[3]The literal integer value '10' is used to specify the number of iterations between convergence checks instead of a constant or parameter. This is due to a known limitation with the current SDSL implementation noted in Sec. 8.1.

```
int main() {
  // Problem size
  const int dim0 = 128;
  const int dim1 = 128;
  const int dim2 = 128;

  // Timing
  double start, finish;

  // Data arrays
  double *U = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  double *F = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  #ifdef CHECK_REF
  double *Uref = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  double *Fref = (double*)malloc(dim0*dim1*dim2*sizeof(double));
  #endif

  // Populate arrays with random values
  srand(45);
  for (int i = 0; i < dim0*dim1*dim2; ++i) {
    F[i] = (double)rand() / (double)(RAND_MAX);
    #ifdef CHECK_REF
    Fref[i] = F[i];
    #endif
  }

  #ifdef CHECK_REF
  // Reference denoise call
  start = timestamp();
  int ref_ret = rician3d(Uref, Fref, 0.05, 0.065, dim0, dim1, dim2);
  finish = timestamp();
  printf(" REF: %.6f sec\n",finish-start);
  #endif

  // SDSL denoise call
  start = timestamp();
  int sdsl_ret = sdsl_rician3d(U, F, 0.05, 0.065, dim0, dim1, dim2);
  finish = timestamp();
  printf("SDSL: %.6f sec\n",finish-start);

  #ifdef CHECK_REF
  // Check results
  check(Uref,U,dim0,dim1,dim2);
  #endif

  // Cleanup
  free(U);
  free(F);
  #ifdef CHECK_REF
  free(Uref);
  free(Fref);
  #endif
  return sdsl_ret;
}
```

Figure 7.8: Driver code.

7.7 Autotuning OverTile Parameters for GPU

In Sec. 7.2 the OverTile thread block sizes, space tile sizes, and time tile size were arbitrarily set as shown below:

```
#pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)
```

Achieving high performance with OverTile codes requires that these parameters be tuned for the the specific GPU architecture the code is running on. In the following sections we autotune these parameters for the Rician denoise SDSL code developed in Sec. 7.2-Sec. 7.5. We slightly alter the `sdsl_rician3d()` function shown in Fig. 7.7 to return `sdsl_return_0` (instead of `0`) so that CUDA kernel launches with invalid configurations can be detected by the autotuner.

7.7.1 Building the Configuration File

The SDSL autotuner works by systematically varying thread block sizes, space tile sizes, and time tile sizes according to a configuration file. This section explains the autotuner configuration file shown in Fig. 7.9. This file is also included in the SDSLC distribution and can be found at `$SDSLC_INSTALL_DIR/share/sdslc/examples/cdsc/autotune-3d.conf`.

```
1  # sdslc flags
2  -b overtile
3
4  # nvcc flags
5  -O3 -arch=compute_30 -code=sm_30 -Xcompiler -O3 -DSTANDALONE_DRIVER
6
7  # Space dims
8  3
9
10 # Thread block sizes <start end step>
11 4 8 2
12 4 8 2
13 4 8 2
14
15 # Space tile sizes <start end step>
16 1 4 1
17 1 4 1
18 1 4 1
19
20 # Time tile sizes <start end step>
21 1 1 1
```

Figure 7.9: Autotuner configuration file.

Line 2 of the configuration file tells the autotuner to run the OverTile backend for the `sdslc` compiler. Line 4 tells the autotuner to run `nvcc` at optimization level 3 for CUDA devices of compute capability 3.0 and to build the standalone driver code shown in Fig. 7.8. Line 8 gives the number of spatial dimensions we are autotuning. Lines 11–13 tell the autotuner to vary thread block sizes in each dimension from 4 to 8 with a step size of 2. Lines 16–18 tell the autotuner to vary space tile sizes in each dimension from 1 to 4 with a step size of 1. Finally, line 21 tells the autotuner to only use a time tile sizes of 1. This configuration results in $3^3 * 4^3 * 1 = 1728$ separate invocations of the Rician denoise kernel.

7.7.2 Running the Autotuner

Execute the following command, from the CDSC examples directory, to begin autotuning:

```
$ python ../../../../bin/autotune-overtile.py autotune-3d.conf
                                               denoise-dp.sdsl.cpp
```

Running the above command will take a substantial amount of time. On a Tesla K10, for example, this autotune configuration takes approximately 2 hours and 15 minutes to complete.

7.7.3 Examining the Autotuner Results

Running the command in Sec. 7.7.2 continually updates `stdout` with its progress, including the specific pragma values being used for an invocation of the denoise kernel, timing infor-

mation, and error information. Throughout the run, and at the end of the run, three files are available:

- `denoise-dp.sdsl.cpp.autotune.log` – Autotuner log
- `denoise-dp.sdsl.autotuned.sdsl.cpp` – Autotuned SDSL file
- `denoise-dp.sdsl.autotuned.cu` – Autotuned CUDA file

The file `denoise-dp.sdsl.cpp.autotune.log`, shown in Fig. 7.10, contains the fastest runs encountered by the autotuner. As the autotuner is executing, the current run is compared to the fastest run executed to that point. If the current run is faster, the associated pragma and timing information are appended to the log.

```
1  #pragma sdsl begin gpu(block:4,4,4 tile:1,1,1 time:1)
2  2.7
3  #pragma sdsl begin gpu(block:4,4,4 tile:1,1,2 time:1)
4  2.28
5  #pragma sdsl begin gpu(block:4,4,6 tile:1,1,1 time:1)
6  2.07
7  #pragma sdsl begin gpu(block:4,4,6 tile:1,1,2 time:1)
8  1.98
9  #pragma sdsl begin gpu(block:4,4,6 tile:1,2,1 time:1)
10 1.79
11 #pragma sdsl begin gpu(block:4,4,8 tile:1,2,1 time:1)
12 1.58
13 #pragma sdsl begin gpu(block:4,4,8 tile:2,2,1 time:1)
14 1.54
15 #pragma sdsl begin gpu(block:4,6,6 tile:2,1,1 time:1)
16 1.49
17 #pragma sdsl begin gpu(block:4,6,8 tile:2,1,1 time:1)
18 1.41
19 #pragma sdsl begin gpu(block:4,8,6 tile:2,1,1 time:1)
20 1.4
21 #pragma sdsl begin gpu(block:4,8,8 tile:2,1,1 time:1)
22 1.18
23 #pragma sdsl begin gpu(block:6,8,8 tile:1,1,2 time:1)
24 1.15
25 #pragma sdsl begin gpu(block:6,8,8 tile:2,1,1 time:1)
26 1.1
27 #pragma sdsl begin gpu(block:8,4,8 tile:1,2,1 time:1)
28 1.09
29 #pragma sdsl begin gpu(block:8,6,8 tile:1,2,1 time:1)
30 1.05
31 #pragma sdsl begin gpu(block:8,8,6 tile:1,1,2 time:1)
32 1.04
```

Figure 7.10: Contents of `denoise-dp.sdsl.cpp.autotune.log` after running on a Tesla K10. The final entry at lines 31–32 was the fastest run.

The file `denoise-dp.sdsl.autotuned.sdsl.cpp` substitutes the original `#pragma sdsl begin gpu(block:4,4,4 tile:4,4,4 time:1)` directive (line 10 of Fig. 7.7) with the `pragma` corresponding to the fastest run encountered during autotuning (line 31 of Fig. 7.10). Other than this, the code is identical to the source in `denoise-dp.sdsl.sdsl.cpp`.

Finally, the file `denoise-dp.sdsl.autotuned.cu` is the result of running the autotuned SDSL file through `sdslc` with the options specified at line 2 of Fig. 7.9.

Chapter 8

Known Limitations

8.1 General

- Only one `grid` may be defined in an SDSL program
- `griddata` may only be defined on timesteps 0 and 1
- `iterate` and `check ... every` trip counts must be integer literals
- Parameters, constants, grids, griddata, point function parameters, point function variables, and reduction variables all share the same namespace thus identifiers must be unique. For example, you cannot declare a constant `float pi = 3.14f` and in a point function definition declare a local variable `float pi = 3.1415f`
- Xeon Phi backend (`sdslc -b xphi`) is experimental and may produce incorrect results
- ARM backend (`sdslc -b arm`) is experimental and may produce incorrect results
- Sinking the outermost intertile loop (`sdslc -s`) is experimental and may produce incorrect results.

8.2 Affine C Backend

- No known issues.

8.3 OverTile Backend

- Grid sizes must be parameters.
  - Grid size parameter declarations must be the first statements in an SDSL program.

  - Grid size parameters must be declared in order from outermost to innermost, e.g. the grid declaration '`grid g[dim1][dim0]`' must have the '`int dim0; int dim1;`' as the first two SDSL statements.

- `check ... every ... iterations` trip count must be an integer multiple of overtile time tile size as specified in the `gpu` clause of the opening sdsl pragma.
- `iterate` trip count must be an integer multiple of `check ... every ... iterations` trip count.

8.4 Nested Split-tiling DLT Backend

- Inverted tile size in the fastest varying dimension must be greater than or equal to 2, that is, in generated source code `SDSL_INV_TILE_SIZE_0` ≥ 2.
- Inverted tile size in the fastest varying dimension must be greater than the negative of any computed statement alpha offset in the fastest varying dimension. In other words, in generated source code `SDSL_INV_TILE_SIZE_0` $>$ `O_ALPHA_S`n`_0` for all statements `S`n.

  This condition guarantees that all statement instances in the initial and final partial inverted tiles are executed. This is an implementation limitation and could be eliminated by implementing (substantially more complicated) boundary code that allows either upright, inverted, or upright and inverted tiles to cross the DLT boundary.
- For any split-tiled dimension d other than the fastest varying, the following inequality must hold:

  N_d%(`UPR_TILE_SIZE_`d+`INV_TILE_SIZE_`d) $>$ $-$`O_BETA_S`n`_`d

  N_d is the size of the dimension, `O_BETA_S`n`_`d is the calculated beta offset in dimension d of any statement in the program, and tile sizes are `INV_TILE_SIZE_`d and `UPR_TILE_SIZE_DIM_`d.

  This condition guarantees execution of all statement instances in the final tile of a non-fastest varying dimension. This is an implementation limitation and be fixed by peeling the final tile iteration and treating violation of this condition as a special case.
- Repeated calls to the same point function may result in errors. This is a bug.

8.5 Hybrid Split-tiling DLT Backend

- See all items in Section 8.4.

Bibliography

[1] Getreuer, P. riciandenoise: 2d and 3d total variation based Rician denoising. `http://cdsc-image-processing-pipeline.googlecode.com/files/riciandenoise.pdf`.

[2] Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., and Sadayappan, P. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction*, J. Knoop, Ed., vol. 6601 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 225–245.

[3] Henretty, T., Veras, R., Franchetti, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 13–24.

[4] Holewinski, J., Pouchet, L.-N., and Sadayappan, P. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (New York, NY, USA, 2012), ICS '12, ACM, pp. 311–320.

[5] Marsa, R., and Choptuik, M. The RNPL user's guide. `http://laplace.physics.ubc.ca/People/arman/files/RNPL_ref.pdf`.

[6] Marsa, R. L., Wilson, A. G., and Choptuik, M. W. SNPL reference manual. `http://laplace.physics.ubc.ca/People/agwilson/SNPL/snplref2.pdf`.

[7] Pouchet, L.-N. PoCC: the polyhedral compiler collection. `http://pocc.sourceforge.net`.

[8] Pouchet, L.-N. Polyopt/C: A polyhedral optimizer for the ROSE compiler. `http://www.cs.ucla.edu/~pouchet/software/polyopt`.