MaxGenFD
Tutorial

Version 2014.1

MAXELER
Technologies
MAXIMUM PERFORMANCE COMPUTING

Contents

Preface

Purpose of this document

This tutorial is designed to show the reader how to use all of the main features of MaxGenFD in real-world seismic applications.

This document can be read independently, though some terminology and concepts used here are introduced in the Dataflow Programming with MaxCompiler tutorial.

Document Conventions

When important concepts are introduced for the first time, they will appear in **bold**. *Italics* are used for emphasis. Directories and commands are displayed in `typewriter` font. Variable and function names are displayed in `typewriter` font.

MaxJ methods and classes are shown using the following format:

FDVar *convolve(**FDVar** toConvolve, ConvolveAxes axes, **Stencil** stencil)*

C function prototypes are similar:

int *maxlib_run(maxlib_context context);*

Actual MaxJ usage is shown without italics:

FDVar laplacian = convolve(p, ConvolveAxes.XYZ, stencil);

C usage is similarly without italics:

maxlib_stream_earthmodel_from_lmem(maxlib, vel_array);

Sections of code taken from the source of the examples appear with a border and line numbers:

```
package ex1_simple;

import com.maxeler.maxgen.fd.ConvolveAxes;
import com.maxeler.maxgen.fd.FDKernel;
import com.maxeler.maxgen.fd.FDKernelParameters;
import com.maxeler.maxgen.fd.FDVar;
import com.maxeler.maxgen.fd.stencils.Stencil;

public class SimpleFDKernel extends FDKernel {

    private final int stencilSize = 13;
    private final Stencil stencil = fixedStencil(-stencilSize/2, stencilSize/2, new double[][] {
            { -0.00003006253006249946f, 0.0005194805194800871f, -0.004464285714282842f, 0.02645502645501459f,
                    -0.133928571428538f, 0.857142857142776f, -1.491388888888878f, 0.857142857142776f, -0.133928571428538f,
                    0.02645502645501459f, -0.004464285714282842f, 0.0005194805194800871f, -0.00003006253006249946f }
    }, 8.0);
```

1 Introduction to MaxGenFD

MaxCompiler provides a general-purpose programming system for Maxeler hardware accelerators. **MaxGen** introduces an application-domain-specific layer to enable rapid development and deployment of accelerators.

1.1 MaxGen

MaxGen is implemented as a layer on top of MaxCompiler, so the user retains the full programming power of MaxCompiler, while pre-defined constructs and libraries implement the details of the domain-specific application. MaxGen allows users to very quickly write accelerated applications with a minimum of effort, with automatic generation of a highly-optimized, parallel, multi-DFE implementation.

1.2 MaxGenFD

A number of important seismic applications such as Forward Modeling, Reverse Time Migration and Waveform Inversion have 3D finite difference at their computational core: **MaxGenFD** addresses these algorithm domains. This document focuses on such seismic applications, though there are many other applications of finite difference methods and therefore MaxGenFD.

MaxGenFD handles the complexities facing any finite difference implementation such as managing very large data sets, boundary conditions and domain decomposition across multiple compute elements with halo exchange. In addition, the compiler minimizes the need for the geoscience programmer to perform DFE-specific optimizations such as customizing data-types and generating optimized stencil descriptions.

The diagram in *Figure 1* shows the interaction of MaxGenFD and MaxCompiler when creating finite difference designs.

MaxGenFD has a MaxCompiler element that provides a domain-specific MaxCompiler interface producing both a Manager and a Kernel. On the software side, MaxGenFD provides a MaxGenFD **MaxLib**, which is a domain-specific library abstracting the software-accelerator communications provided by SLiC.

1.3 MaxGenFD Programming Model

In a typical software finite difference application, the code loops through many timesteps of a core kernel that performs wave propagation. Broadly, the computation in the kernel can be broken down into derivative calculations (convolving one or more of the input wavefields with a stencil operator), wave equation (computing the next wavefield state based on the input wavefields and the derivatives), and boundary conditions. Convolution with the finite difference stencil usually dominates both operation count and computation time.

Figure 2 shows the architecture of a MaxGenFD-accelerated application compared to a software-only implementation. In the pure software implementation, the CPU executes the kernel computation and the compute node's main memory contains the wavefield and earth model data. When using MaxGenFD, the core kernel of code, including the critical convolutions, has been reimplemented as a specialized finite difference Kernel (**FDKernel**) and is now running on the DFE. The FDKernel behaves as a function with some number of wavefield inputs, some number of earth model inputs, and some number of wavefield outputs. Wavefields and earth models are now stored (possibly in compressed form) in the LMem memory on a Maxeler DFE. In addition, the FDKernel can have some number of

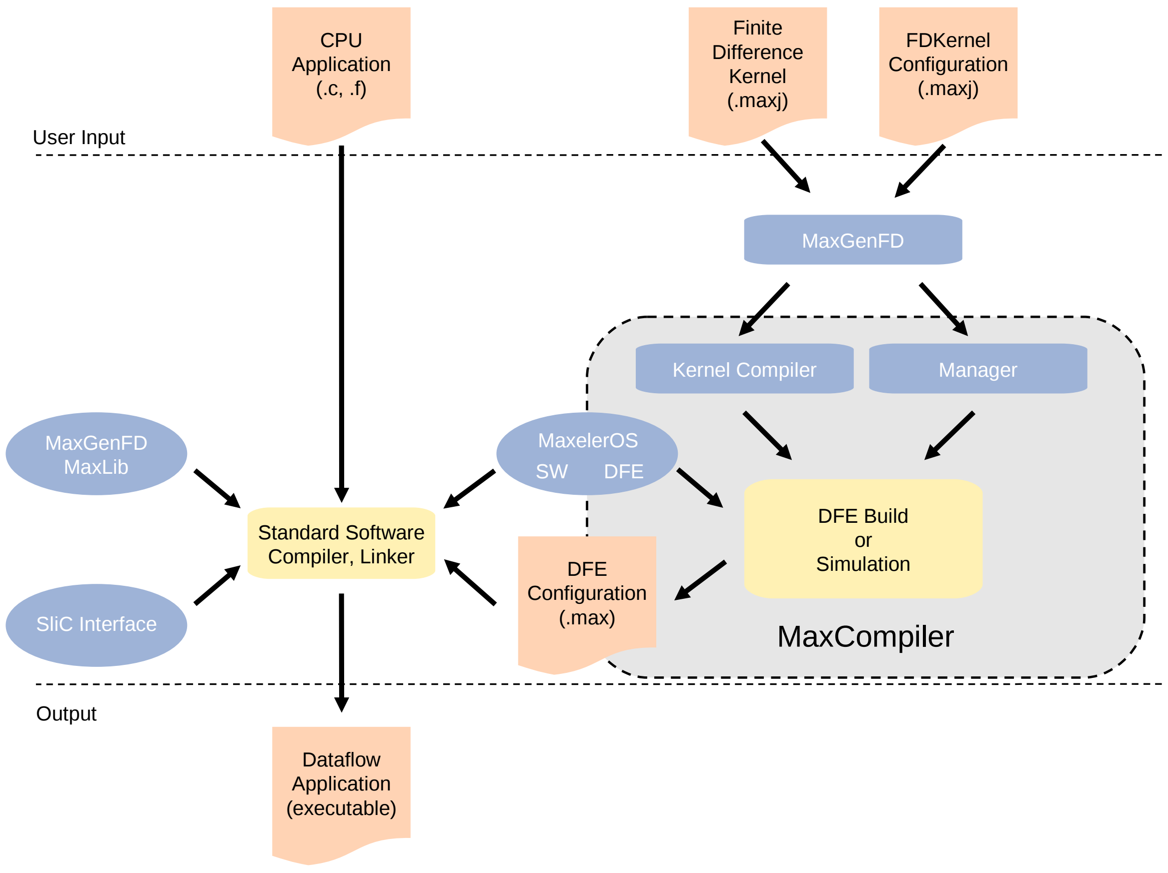


Figure 1: Layered architecture of MaxGenFD on top of MaxCompiler.

host inputs and outputs that can be used to receive stimulus data from the CPU and to return output data to the CPU during the execution of a timestep.

To accelerate an application using MaxGenFD, the user splits the application into three components:

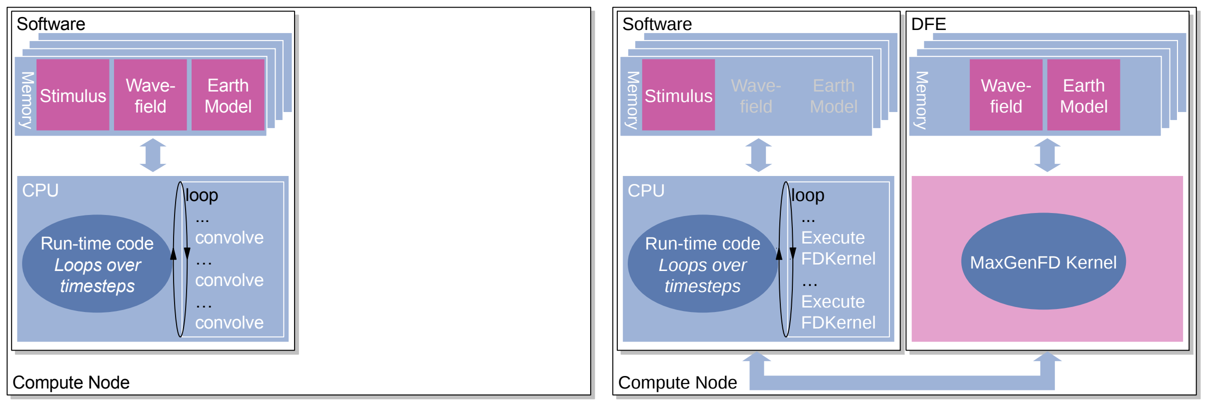
- FDKernel (written in MaxJ)
- FDKernel configuration (written in MaxJ)
- Runtime (written in C, C++ or FORTRAN), which includes the parts of the application code that will remain in software.

1.4 FDKernel

The FDKernel is a mathematical description of a specialized function that, given appropriate input data, performs some part of a finite difference calculation (usually a timestep, or part of a timestep).

1.5 FDKernel Configuration

The FDKernel configuration sets the performance options for the FDKernel, such as types, compression and number of pipelines, at DFE configuration compile time. This allows a single mathematical



(a) Software implementation *(b)* Accelerated implementation

Figure 2: Software and accelerated finite difference implementations. In the accelerated implementation, the earth model and wavefields are stored in the LMem of the DFE(s).

FDKernel description to be used to generate multiple DFE configurations with different performance characteristics.

1.6 Runtime

The runtime integrates the accelerated kernel into a complete software application. The runtime code drives the FDKernel and decides what input data to provide to the FDKernel at any point in time. The runtime can also set scalar input values for the FDKernel to configure specific functionality, for example to set different boundary conditions for different surfaces of the problem domain.

By partitioning the application in this way, functionality that is fixed (e.g. the shape of the finite difference stencil) is hard-coded into the DFE configuration and highly optimized, however parts that need to change on a timestep-to-timestep basis (e.g. which input wavefields should be read) are configured in software.

1.7 MaxGenFD capabilities

MaxGenFD kernels provide automatic support for a variety of constructs that you may need when accelerating finite difference applications.

1.7.1 Optimized stencil generation

As well as coding the finite difference stencil operators directly (using *, +, etc operators), you can opt to describe your finite difference stencil in a more abstract form and then apply it to input streams as a single operation. This enables MaxGenFD to generate highly optimized hardware for the stencil operator.

You can also easily describe different stencils to be used at the edges of the domain to provide “roll-on/roll-off” conditions: MaxGenFD will automatically apply these stencils at the appropriate points at edges of the domain.

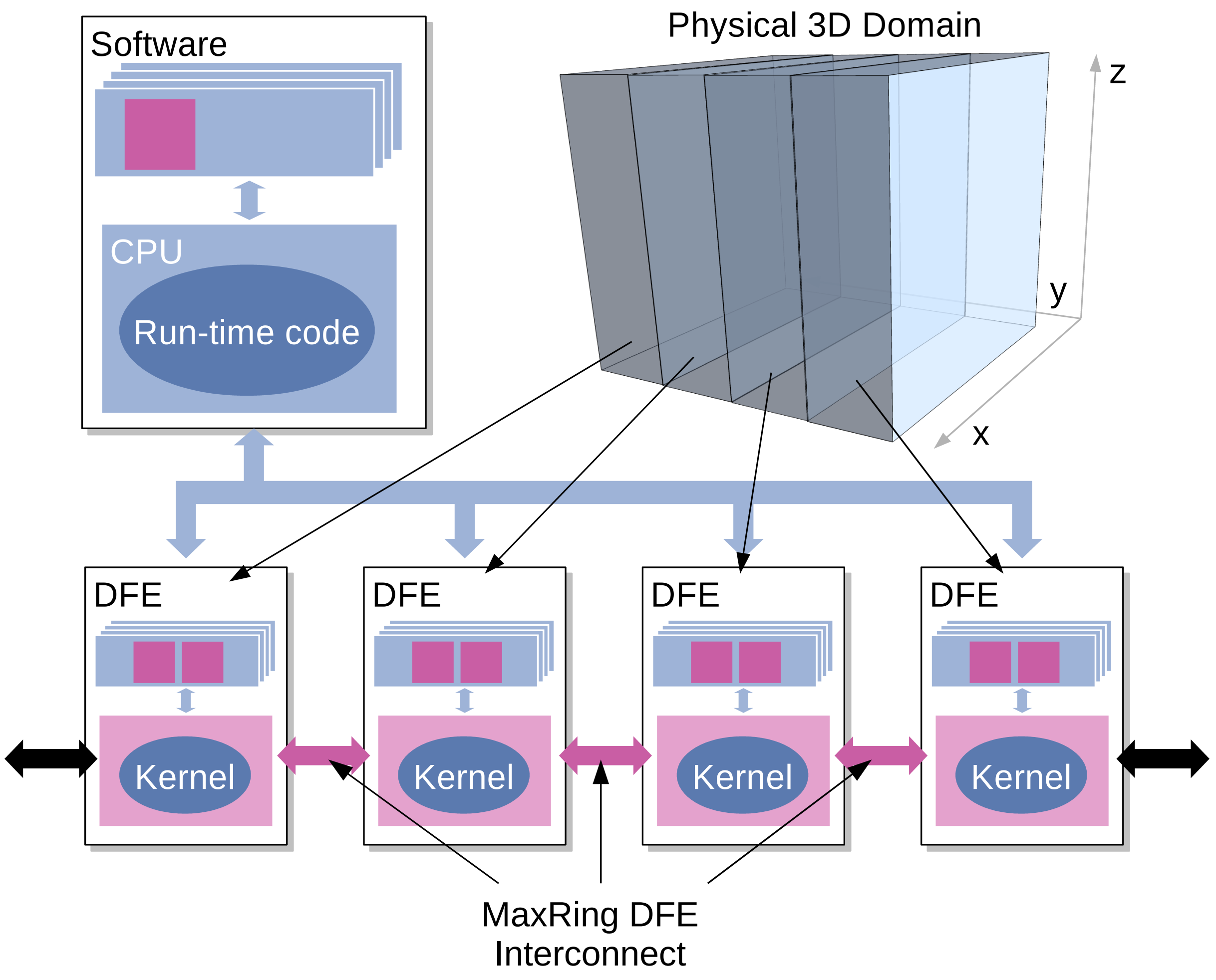


Figure 3: Accelerated finite difference with 3D domain decomposed over multiple DFEs connected via a MaxRing interconnect.

1.7.2 Parallelism

FDKernel descriptions are inherently **multi-pipe**, providing both parallelism within a single computational pipeline and the instantiation of multiple computational pipelines running in parallel. The FDKernel description is the same regardless of the level of parallelization: you can specify the number of computational pipelines that are created during the DFE build process.

1.7.3 Domain Decomposition

A large domain can be decomposed into multiple blocks running either on a single DFE or across multiple MaxCards. The MaxCards can communicate via direct MaxRing or the CPU. *Figure 3* shows a 3D domain decomposed into four blocks, with each block mapped to a different DFE.

The halos for each of the blocks of the domain are automatically exchanged between the blocks over the MaxRing or through the CPU , giving maximum computational throughput in each of the blocks with no added complexity for the user.

1.7.4 Debugging

DFE simulation allows you to simulate FDKernels quickly to verify their functionality and to tune the results. Using a simulated system, the same host software code can be run against small problem sizes for many timesteps purely in simulation (e.g. 64x64x64 domains) before building a real DFE configuration.

Once the behavior has been verified, then the design can be compiled into a DFE configuration and run with a large domain. On a DFE, debugging facilities are still available in the form of numeric exceptions (detecting overflow, underflow, divide by zero and invalid operations) and MaxDebug, a graphical tool for debugging data flow scheduling in the complete accelerated system.

2 Getting Started

In this section, we show all the development tasks needed to create a MaxGenFD application, following a simple example as we go (Example 1). We use floating point in this example for simplicity: we will revisit this example in *section 7* to add fixed-point optimizations.

2.1 MaxIDE

This tutorial assumes that you have setup MaxIDE and imported the MaxGenFD tutorial examples. For more information on how to setup MaxIDE see section 4 of the MaxCompiler tutorial. The MaxCompiler tutorial can be found in the docs folder inside the directory that MaxCompiler was installed to. The `MAXCOMPILERDIR` environment variable points to MaxCompiler's installation directory.

2.2 A Basic Example

To introduce MaxGenFD, we consider an example of an Acoustic Forward Modeling code. We model an isotropic medium with variable velocity, described by the equation:

$$\frac{\partial^2 p}{\partial t^2} = v^2 \nabla^2 p + s(t)$$

We use an absorbing "sponge" region at the edges of the domain to reduce the effect of reflections. A finite difference implementation of this equation can be described in pseudo-code as in *Listing 1*, where `X`, `Y` and `Z` are the dimensions of the domain, `prev`, `curr` and `next` are the previous, current and next wavefields, `source` is a source wavelet and `vv` is a velocity-squared earth model.

Listing 1: Pseudo-code for an Acoustic Forward Modeling implementation.

```
for timestep = 1 to timestep_max:
  for i = 0 to X*Y*Z-1:
    laplacian := convolve(curr[i], stencil)
    next[i] := 2 * curr[i] - prev[i] + vv[i] * laplacian
    next[i] := next[i] + source[i]
  apply_boundary_sponge(curr, next)
  swap_buffer_pointers(prev, curr, next)
```

2.3 The FDKernel

In our example, the timestep is implemented as an FDKernel using MaxGenFD. *Figure 4* shows how the FDKernel executes each timestep. The current wavefield (p_t) is read from the DFE's LMem memory into the FDKernel and convolved. The wave equation uses the results of the convolution, the current and previous wavefields (p_t and p_{t-1}) and the earth model (read from the DFE's LMem). The source wavelet (streamed from the CPU) is then added for the current timestep (s_t). The result (p_{t+1}) is written back to the DFE's LMem.

Although this diagram has been shown with three wavefield buffers for simplicity (previous, current and next), it is preferable to use only two buffers, writing the results from a timestep directly into the previous wavefield buffer: we use this more efficient implementation in our example.

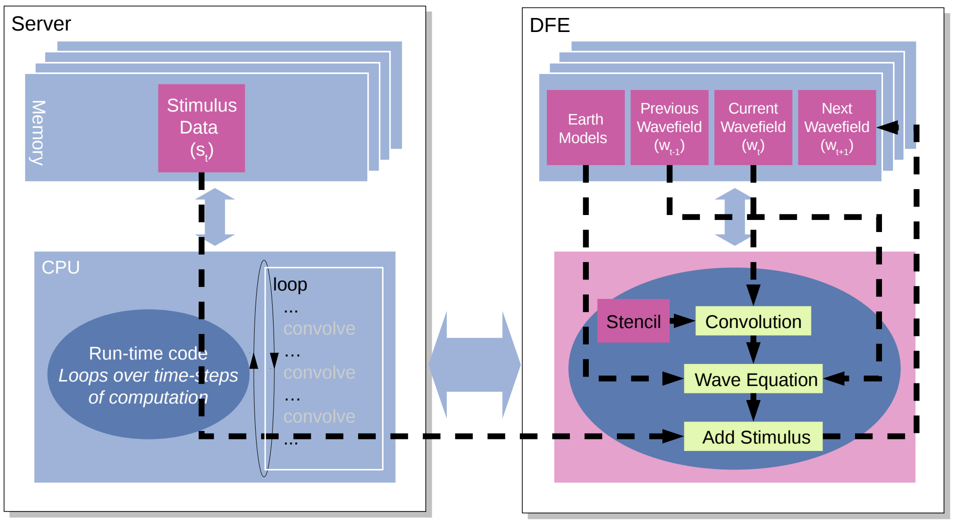


Figure 4: Acoustic Forward Modeling example Kernel showing inputs and outputs of the convolution and wave equation on each time step.

The source code for our FDKernel is shown in *Listing 2*. `SimpleFDKernel` is a MaxJ class that extends the MaxGenFD base class `FDKernel`.

Stencils are created as linear stencils that can then be applied in one or more dimensions. If the stencil contains symmetry, then this will automatically be optimized to reduce the number of multiplication operations that need to be performed on the DFE.

In our Acoustic Forward Modeling example, we declare a 13-element stencil, starting at location −6 and ending at +6 relative to the central point (0). We expect the result from our convolution to be no larger than 8 times the input, so we pass this as the final argument to the stencil creation method to enable MaxGenFD to optimize the arithmetic and data types:

```
private final int stencilSize = 13;
private final Stencil stencil = fixedStencil(
    -stencilSize / 2,
    stencilSize / 2,
    new double[] { -0.000030062530062499 46f, 0.0005194805194800871f,
            -0.004464285714282842f, 0.02645502645501459f,
            -0.133928571428538f, 0.857142857142776f, -1.491388888888878f,
            0.857142857142776f, -0.133928571428538f, 0.02645502645501459f,
            -0.004464285714282842f, 0.0005194805194800871f,
            -0.00003006253006249946f },
    8.0);
```

MaxGenFD introduces the stream reference `FDVar`, which is automatically parallel and allows the FDKernel to specify the processing algorithm without being type-specific: the actual Kernel type for the stream is set in a separate configuration object.

MaxGenFD leverages the streaming computation model of MaxCompiler, so all of our inputs to and outputs from the Kernel are streams. We do not specify where streams should get their data explicitly: we declare our streams using input and output types (e.g. wavefield, earth model, host input), which allows MaxGenFD to connect the streams appropriately.

The inputs for the wavefields are read from input streams `curr_w` for the current wavefield and

Listing 2: Acoustic Forward Modeling Kernel (SimpleFDKernel.maxj).

```
/**
 * Document: MaxGenFD Tutorial (maxgenfd-tutorial.pdf)
 * Example: 1      Name: Simple FD
 * MaxFile name: SimpleFD
 * Summary:
 *      An FDkernel that will apply our stencil .
 */
package simplefd;

import com.maxeler.maxgen.fd.ConvolveAxes;
import com.maxeler.maxgen.fd.FDKernel;
import com.maxeler.maxgen.fd.FDKernelParameters;
import com.maxeler.maxgen.fd.FDVar;
import com.maxeler.maxgen.fd.stencils.Stencil;

class SimpleFDKernel extends FDKernel {
    private final int stencilSize = 13;
    private final Stencil stencil = fixedStencil (
        -stencilSize / 2,
        stencilSize / 2,
        new double[] { -0.00003006253006249946f, 0.0005194805194800871f,
                -0.004464285714282842f, 0.02645502645501459f,
                -0.133928571428538f, 0.857142857142776f, -1.491388888888878f,
                0.857142857142776f, -0.133928571428538f, 0.02645502645501459f,
                -0.004464285714282842f, 0.0005194805194800871f,
                -0.00003006253006249946f },
        8.0);


    SimpleFDKernel(FDKernelParameters parameters) {
        super(parameters);
        FDVar curr = io.waveFieldInput("currW", 1.0, stencilSize / 2); // Current wavefield

        FDVar prev = io.waveFieldInput("prevW", 1.0, 0); // Previous wavefield
        FDVar dvv = io.earthModelInput("dvv", 1 / 4.0, 0); // Earth model
        FDVar source = io.hostInput("source", 1.0, 0); // Stimulus data
        FDVar sponge = boundaries.sponge(50); // Sponge

        prev = prev * sponge; // Sponge previous wavefield

        FDVar laplacian = convolve(curr, ConvolveAxes.XYZ, stencil);
        FDVar result = curr * 2 - prev + dvv * laplacian + source;

        result = result * sponge; // Sponge result

        io.hostOutput("receiver", result); // Receiver output to host
        io.waveFieldOutput("nextW", result); // Wavefield output
    }
}
```

`prev_w` for the previous wavefield:

```
32          FDVar curr = io.waveFieldInput("currW", 1.0, stencilSize / 2); // Current wavefield
33
34          FDVar prev = io.waveFieldInput("prevW", 1.0, 0); // Previous wavefield
```

The two input wavefields are specified with a string name of the input (`"curr_w"` and `"prev_w"`), relative magnitude (`1.0`, used for optimization purposes, see *subsubsection 7.8.2*) and halo size (`stencilSize/2` and `0`, see *subsection 3.4* for more detail) as their arguments.

The earth model is read from an input stream that comes from the local LMem on the DFE:

```
        FDVar dvv = io.earthModelInput("dvv", 1 / 4.0, 0); // Earth model
```

The earth model input is declared with a string name (`"dvv"`), the absolute maximum value in the earth model (`1/4.0`, used for optimization, see *subsubsection 7.8.1*) and halo size (`0`).

The state of the source wavelet for the current timestep is streamed over from the CPU:

```
        FDVar source = io.hostInput("source", 1.0, 0); // Stimulus data
```

The source wavelet also has a magnitude relative to the other inputs (`1.0`, see *subsubsection 7.8.2* for more detail on relative magnitudes) and halo size specified (`0`).

A stream of sponge coefficients is created using MaxGenFD's boundary conditions library. A 50-pt absorbing "sponge" region is generated by the call to `boundaries.sponge` as a multiplicand that is multiplied with the two wavefields. The values for the sponge and the edges of the domain to which it should be applied are set in the host software code.

```
        FDVar sponge = boundaries.sponge(50); // Sponge

        prev = prev * sponge; // Sponge previous wavefield
```

Because the FDKernel description is a streaming program, all points in the problem domain are multiplied by the sponge value, even if they are in the interior of the domain, but the boundary library will set the sponge coefficient to 1.0 when outside the absorbing region.

Notice that the way the sponge is applied is subtly different to the software pseudo-code (*Listing 1*) for two reasons:

- Performing the sponging before the wave equation update requires writing out only one wavefield to LMem for `result` each execution step instead of two output wavefields, one for `curr` and one for `prev`, using half as much of the available memory bandwidth.
- Applying the sponge to `result` before writing out to the LMem in the current execution step rather than sponging `curr` in the next execution step means that the two sponging operations take place much closer together in the schedule for the FDKernel. Sponging `curr` and `next` before the wave equation means that more buffering is necessary as the two uses of the sponge are either side of the convolution operation, which has significant buffering associated with it.

At this point, all of the input streams are set up, so we can implement the calculation.

In our Acoustic Forward Modeling example, we apply our linear stencil to all three dimensions of the current wavefield to perform a Laplacian transformation. This is done with a single call to the `convolve` method:

```
        FDVar laplacian = convolve(curr, ConvolveAxes.XYZ, stencil);
```

`curr` is the wavefield to convolve, `ConvolveAxes.XYZ` indicates that we want to perform the convolution in all three axes and `stencil` is the stencil to use for the convolution.

The output from the convolution, `laplacian`, is used in the update to the wave equation and the source wavelet is added:

```
        FDVar result = curr * 2 - prev + dvv * laplacian + source;
```

The resultant wavefield has the sponge applied to it in this execution step:

```
        result = result * sponge; // Sponge result
```

We can read back some or all of the data using an output back to the CPU:

```
        io.hostOutput("receiver", result); // Receiver output to host
```

Exactly how much data, from where in the domain it should be taken and how often we want to read it can be specified at run time in the CPU code.

Finally, the resultant wavefield for this timestep is written to an output stream which writes to `"next_w"`:

```
        io.waveFieldOutput("nextW", result); // Wavefield output
```

2.4 Configuring the Kernel

With no specification of data types within the FDKernel in MaxGenFD, all optimization is performed via a configuration object, shown in *Listing 3*.

The configuration object should be fully set up at the beginning of the build process and then passed to the Manager. Attempting to change the configuration object from within the FDKernel will result in an error from MaxGenFD.

We create the configuration object with engine parameters which store user configuration settings and the basic type that we will use in the FDKernel, in this case floating point:

```
        // Use floating point
        FDConfig config = new FDConfig(engineParameters, DefaultType.FLOAT);
```

We set the number of parallel processing pipelines we want to use, in our case 1:

```
        config.setParallelPipelines(1);
```

The number of pipelines we can build into a single DFE depends mainly on the size of the stencil and the data types in use. In this design we are using floating point which requires more resources per pipeline than fixed point. We use just one pipeline here to accelerate the building of DFE configurations.

In order to make best use of the available memory bandwidth, we enable compression on wavefields and our earth model:

```
        config.setWavefieldStorageType(StorageType.float8_24);
        config.setEarthModelStorageType("dvv", StorageType.compressedTable(10));
```

We use 16-bit compression for our wavefields (see *subsubsection 7.2.2*) and table compression for our earth model (see *subsubsection 7.2.1*).

We can set the types for each stage of the computation. For our simple implementation of Acoustic Forward Modeling, we use single-precision floating point throughout the FDKernel by declaring a

Listing 3: Acoustic Forward Modeling configuration object (SimpleFDConfig.maxj).

```
/**
 * Document: MaxGenFD Tutorial (maxgenfd-tutorial.pdf)
 * Example: 1      Name: Simple FD
 * MaxFile name: SimpleFD
 * Summary:
 *      Configuration settings for the model.
 */

package simplefd;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEFloat;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFETypeFactory;
import com.maxeler.maxgen.fd.ComputeType;
import com.maxeler.maxgen.fd.FDConfig;
import com.maxeler.maxgen.fd.FDConfig.DefaultType;
import com.maxeler.maxgen.fd.StorageType;

class SimpleFDConfig {
    static FDConfig config(EngineParameters engineParameters) {
        // Use floating point
        FDConfig config = new FDConfig(engineParameters, DefaultType.FLOAT);

        config.setParallelPipelines(1);

        config.setWavefieldStorageType(StorageType.float8_24);
        config.setEarthModelStorageType("dvv", StorageType.compressedTable(10));

        // Set our floating point type for all calculations
        config.setWavefieldComputeType(ComputeType.float8_24);
        // IEEE single-precision floating point
        DFEFloat t = DFETypeFactory.dfeFloat(8, 24);
        config.setCoefficientType(t);
        config.setEarthModelComputeType("dvv", t);

        return config;
    }
}
```

MaxCompiler floating point type and passing it to methods that set type options on the FDKernel:

```
        // Set our floating point type for all calculations
        config.setWavefieldComputeType(ComputeType.float8_24);
        // IEEE single-precision floating point
        DFEFloat t = DFETypeFactory.dfeFloat(8, 24);
        config.setCoefficientType(t);
        config.setEarthModelComputeType("dvv", t);
```

2.5 Configuring a Manager

After designing the Kernel, we need to configure a Manager that will connect our Kernel to the outside world and build our design for either a DFE (real or simulated).

`SimpleFDManager.maxj` presents the MaxJ code for the Manager that will build the DFE configuration implementing our simple Kernel. We specify a `main()` method to run the build process, in which we pass a `FDConfig` to a `FDManager` on which we call the `build()` method:

```
    public static void main(String[] args) {

        EngineParameters params = new EngineParameters(args);
        FDConfig config = SimpleFDConfig.config(params);

        FDManager m = new FDManager(config);
        SimpleFDKernel k = new SimpleFDKernel(m.makeKernelParameters());

        m.setKernel(k);

        m.build();
    }
}
```

The constructor of the FDManager takes the engine parameters as a parameter These specify, among other things, the DFE model to target during the build. Do not modify these directly. Instead edit the RunRule you are using in MaxIDE. MaxIDE will pass the necessary parameters to `EngineParameters` automatically.

To start a build right-click on one of the Run Rules and select Build from the menu. Be aware that the DFE Run Rule will take much longer to build than the Simulation Run Rule.

Listing 4 shows example console output from building the DFE Run Rule. The result of the build process is the file `Simple.max` in the `results` sub-directory of the build path (line 42 in *Listing 4*).

Listing 4: Simplified DFE Build Output (SimpleFDHWBuilderOutput.txt).

```
Fri 18:21: Build location: /disk/builds/SimpleFD_VECTIS_DFE
Fri 18:21: Detailed build log available in "_build.log"
Fri 18:21: MaxGenFD version: 2014.1
Fri 18:21: Instantiating kernel "FDKernel"
Fri 18:21:
Fri 18:21: Compiling kernel "FDKernel"
Fri 18:22: Generating input files (VHDL, netlists, CoreGen)
Fri 18:26: Running back-end build (12 phases)
Fri 18:26: (1/12) - Prepare MaxFile Data (GenerateMaxFileDataFile)
Fri 18:26: (2/12) - Synthesize DFE Modules (XST)
Fri 18:34: (3/12) - Link DFE Modules (NGCBuild)
Fri 18:36: (4/12) - Prepare for Resource Analysis (EDIF2MxruBuildPass)
Fri 18:37: (5/12) - Generate Preliminary Annotated Source Code (Prel..
Fri 18:37: (6/12) - Report Resource Usage (XilinxPreliminaryResource..
Fri 18:37:
Fri 18:37: PRELIMINARY RESOURCE USAGE
Fri 18:37: Logic utilization:  111165 / 297600 (37.35%)
Fri 18:37:  LUTs:               65557 / 297600 (22.03%)
Fri 18:37:  Primary FFs:        90716 / 297600 (30.48%)
Fri 18:37: Multipliers (25x18):    25 / 2016  (1.24%)
Fri 18:37:  DSP blocks:            25 / 2016  (1.24%)
Fri 18:37: Block memory (BRAM18): 523 / 2128 (24.58%)
Fri 18:37:
Fri 18:37: (7/12) - Prepare for Placement (NGDBuild)
Fri 18:41: (8/12) - Place and Route DFE (XilinxMPPR)
Fri 18:41: Executing MPPR with 1 cost tables and 1 threads.
Fri 18:41: MPPR: Starting 1 cost table
Fri 19:32: MPPR: Cost table 1 met timing with score 0 (best score 0)
Fri 19:32: (9/12) - Prepare for Resource Analysis (XDLBuild)
Fri 19:34: (10/12) - Generate Resource Report (XilinxResourceUsageBu..
Fri 19:34: (11/12) - Generate Annotated Source Code (XilinxResourceA..
Fri 19:34: (12/12) - Generate MaxFile (GenerateMaxFileXilinx)
Fri 19:45:
Fri 19:45: FINAL RESOURCE USAGE
Fri 19:45: Logic utilization:   90310 / 297600 (30.35%)
Fri 19:45:  LUTs:               63904 / 297600 (21.47%)
Fri 19:45:  Primary FFs:        74847 / 297600 (25.15%)
Fri 19:45:  Secondary FFs:      14003 / 297600 (4.71%)
Fri 19:45: Multipliers (25x18):    25 / 2016  (1.24%)
Fri 19:45:  DSP blocks:            25 / 2016  (1.24%)
Fri 19:45: Block memory (BRAM18): 524 / 2128 (24.62%)
Fri 19:45:
Fri 19:45: MaxFile: /disk/builds/SimpleFD_VECTIS_DFE/results/SimpleFD.max (
    MD5Sum: 221551c1414e39c49e7356b7641143fd)
```

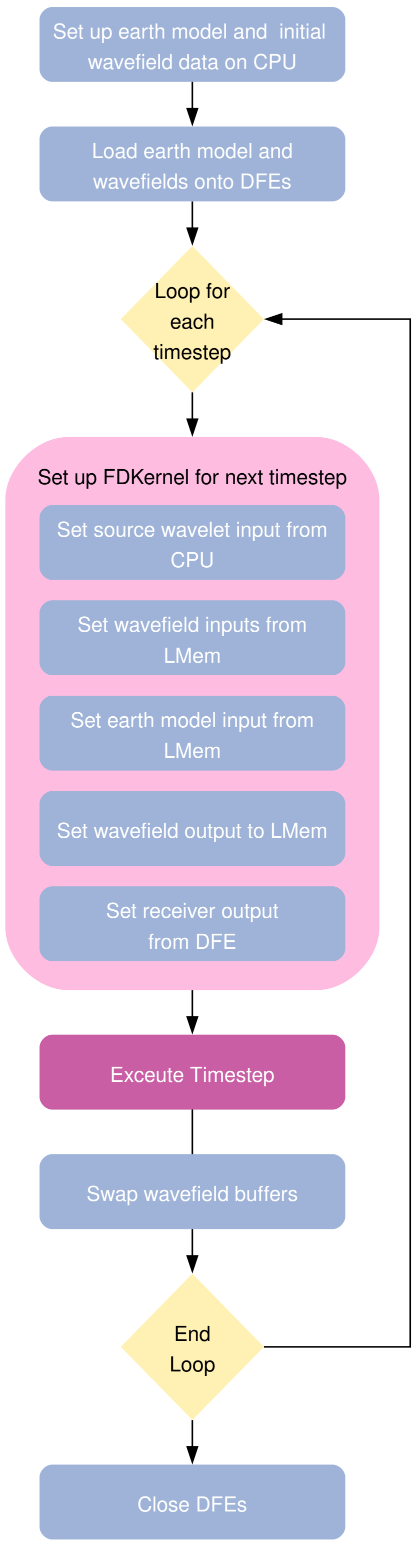

Set up earth model and initial wavefield data on CPU
Load earth model and wavefields onto DFEs
Loop for each timestep
Set up FDKernel for next timestep
Set source wavelet input from CPU
Set wavefield inputs from LMem
Set earth model input from LMem
Set wavefield output to LMem
Set receiver output from DFE
Exceute Timestep
Swap wavefield buffers
End Loop
Close DFEs

Figure 5: CPU code execution flow for the Acoustic Forward Modeling Example.

2.6 Integrating with the Software

Figure 5 shows the flow of execution of the CPU code for the Acoustic Forward Modeling Example. Firstly, the earth model and initial wavefields are set up on the DFE. These are then transferred to the DFE.

For each timestep, the CPU sets up the computation it would like the DFE to perform. The software selects whether the source wavelet should be added and generates its current state, which wavefields should be streamed to the inputs of the FDKernel and which earth model to stream in. It also sets up where the output stream should be written and whether or not to stream out the receiver plane.

Once the set-up is complete, the CPU instructs the FDKernel to run the computation. The source wavelet is streamed from the host for each timestep if required. The specified wavefield inputs and earth models are streamed into the FDKernel from the LMem. The output from the calculation is streamed back into the LMem and the receiver is read back if required. At the end of each timestep, the wavefield buffers are swapped.

The host code has been split into two sections for convenient display in this document.

2.6.1 Setting up the Accelerator Cards

The first section, shown in *Listing 5*, declares constants and variables that are used throughout the design and sets up the accelerator cards.

We first declare the size of the domain we are using:

```
const int nx = maxlib_is_simulation () ? 66 : 676;
const int ny = maxlib_is_simulation () ? 66 : 676;
const int nz = maxlib_is_simulation () ? 66 : 240;
int n = nx * ny * nz;
```

We use a smaller domain when using the simulated system (see *subsection 2.7*) as executing the FDKernel in the simulator is much slower than executing in real hardware.

We declare the boundary sponge to be 50-points wide, with an active area of all 50 points in real hardware but only 10-points in simulation when we have a much smaller domain:

```
const int spongeWidth = 50; // Width of the sponge in points
const int spongeActiveWidth = maxlib_is_simulation() ? 10 : 50; // Width of the sponge in use
// Bit mask for the sides of the domain to sponge (all sides except z=0)
const int spongeMask = MAXLIB_BOUNDARY_ALL ^ MAXLIB_BOUNDARY_Z0;
```

We also declare our pointers for storing data in memory:

```
float *dvv; // Velocity earth model
float *zeroWavefield; // Empty wavefield to initialize simulation
float *receiver; // Receiver
float source[3 * 3 * 3]; // Source stimulus
float *sponge; // Sponge
```

The first step in setting up the accelerator cards is to initialize the MaxLib with the dimensions of the 3D domain:

```
maxlib_context maxlib = maxlib_open(nx, ny, nz); // Initialise maxlib with dimensions of domain
maxlib_print_status (maxlib);
```

We also print out the properties of the MaxLib so that we can check that everything is as we expect. This gives the output:

```
MaxGenFD simulation run status
==============================
       Version: 2014.1
       Data flow engine: Vectis (MAX3424A)
       Device topology: [X 0 X]
       Performance: 1 pipe @ 100 MHz
       Computation type: floating-point
       Problem size: 66 x 66 x 66 (f x m x s)
       Block size: 96 x 96 (f x m)
       Tile size: 8 x 6 (f x m)
       Wavefield tracking support: no
       IDFE swap support: no
       MaxRing swap support: no
       PCIe swap support: no
       ECC enabled: no
       Global parameters:
              MAX_DEVICES = 8
```

- `Version`: The version of MaxGenFD that is loaded.
- `Data flow engine`: The DFE model in use.
- `Device topology`: This shows the halo exchange topology of the cards that MaxGenFD has opened.

  This contains one or more devices. Each device is of the format `[? N ?]` where `N` is the device number and `?` represents the type of link to the next card.

  `?` can be one of:

  - `X`: No link
  - `I`: IDFE link (MAX2 only)
  - `R`: MaxRing link
  - `P`: CPU (PCI Express) link

  For example:

  - `[X 0 I] [I 1 P] [P 2 I] [I 3 X]`: Two MAX2 cards connected by a CPU link.
  - `[X 0 I] [I 1 R] [R 2 I] [I 3 X]`: Two MAX2 cards connected by a MaxRing link.
- `Perfomance`: Configuration parameters relevant to the speed of computation performed by the DFE.
- `Computation type`: Either fixed-point or floating point.
- `Problem size`: The dimensions of the domain.
- `Block size`: The size of the block that the DFE will process (see *subsection 8.3*).
- `Tile size`: The size of the tile used by MaxGenFD (see *subsubsection 9.2.6*).

- `Wavefield tracking support`: Whether this `.max` file supports the wavefield tracking optimization or not.
- `IDFE swap support`, `MaxRing swap support`, `PCIe swap support`: Whether these halo swap features are enabled in the `.max` file.
- `ECC enabled`: Whether ECC error checking is enabled on the data into and out of LMem.
- `Global parameters`
  - `MAX_DEVICES`: The maximum number of DFEs that MaxGenFD will open.

Note that all of the functions that you call in the MaxGenFD MaxLib are automatically applied across all of the DFEs in the system.

We next set up our earth model using a function that we have defined elsewhere. This is the same function as we would use for a software implementation of the application. The earth model is compressed into a new block of memory according to the compression model that was specified in the `FDConfig` object:

```
dvv = create_2layer_earth_model_vel(nx, ny, nz); // Set up earth model
preprocess_earth_model_dvv(dvv, nx, ny, nz, deltaDistance, deltaTime);

maxlib_earthmodel em = maxlib_earthmodel_create_in_memory(nx, ny, nz);

 printf ("Compressing earth model\n");
maxlib_earthmodel_set_data(em, "dvv", dvv);
free(dvv); // We don't need the original earth model any more
```

We call functions to allocate LMem on the DFEs to hold the wavefields and earth model:

```
 // Allocate memory on the MaxCard for the wavefields and earth model
maxlib_lmem_array dfeCurrArray = maxlib_lmem_alloc_wavefield(maxlib);
maxlib_lmem_array dfePrevArray = maxlib_lmem_alloc_wavefield(maxlib);
maxlib_lmem_array dfeEarthmodels = maxlib_lmem_alloc_earthmodel(maxlib);
```

The contents for these arrays are then copied over from the host to the DFEs. The appropriate sections of the 3D volume will be copied to the LMem of each DFE opened.

```
 printf ("Loading curr wavefield to DFE\n");
maxlib_lmem_load_wavefield(maxlib, dfeCurrArray, zeroWavefield);
 printf ("Loading prev wavefield to DFE\n");
maxlib_lmem_load_wavefield(maxlib, dfePrevArray, zeroWavefield);
 printf ("Loading earth model to DFE\n");
maxlib_lmem_load_earthmodel(maxlib, dfeEarthmodels, em);
maxlib_earthmodel_release(em);
```

We create values for our sponge array using a function defined elsewhere. Again, this is the same function as we would use for defining the sponge boundary condition in a software implementation. We set the number of active points in the sponge in our function: the remaining points will be set to 1.0 so that no sponging will be performed to this area of the wavefield.

We need to tell the DFE, using a bit mask (sponge_mask), to which sides of the 3D domain it should apply the sponge (covered in *subsection 6.1*). The last step of setting up the sponge is to copy the values to the DFEs:

```
 maxlib_set_scalar_flag (maxlib, "absorb", spongeMask); // Set bit mask
sponge = create_simple_sponge(spongeWidth, spongeActiveWidth); // Create sponge
 printf ("Loading sponge to DFE\n");
for (int i = 0; i < spongeWidth; i++) // Set the sponge on the MaxCard
    maxlib_set_mapped_memory_f(maxlib, "sponge", i, sponge[i]);
```

The final set-up step is to allocate memory on the host for the receiver which will have data written into it from the DFEs.

```
receiver = malloc(ny * nz * sizeof(float)); // Allocate memory for receiver plane
```

2.6.2 Executing the Timesteps

With all the data now set up on both the CPU and the DFEs, we can start the computation. We iterate through each of the timesteps, injecting the source if necessary, setting the input and output wavefield locations and reading out the receiver data when required. The code for this section is shown in *Listing 6*

The source wavelet is created for each timestep using another function that we have defined elsewhere. Again, this function is the same as we would use for a software implementation of the application. The source wavelet array is copied across to the DFEs. As the source wavelet is only a small volume in the whole 3D domain, the source is specified in its own small 3D volume and transferred to the DFEs with its position within the larger domain. This minimizes the amount of data that must be transferred to the DFEs each timestep.

```
    if (time <= 2.0f / fpeak)
        gen_source(source, time, fpeak); // Generate source wavelet

    float* data = (time <= 2.0f / fpeak) ? source : NULL;
    maxlib_stream_region_from_host(
        maxlib,
        "source",
        data,
        sourceX -1,
        sourceY -1,
        sourceZ -1,
        sourceX + 2,
        sourceY + 2,
        sourceZ + 2);
```

In our case the center of the source wavelet is the mid-point in a 3x3x3 array, which we use to anti-alias the source into a 3D cube, so we pass the start and end points of the source wavelet volume such that the center of the cube is at the point *(source_x, source_y, source_z)*.

Each timestep, we must specify the wavefields and earth model that should be streamed from LMem into the FDKernel and where the output wavefield should be written to:

```
    // Set up inputs and output arrays for this timestep
    maxlib_stream_from_lmem(maxlib, "currW", dfeCurrArray);
    maxlib_stream_from_lmem(maxlib, "prevW", dfePrevArray);
    maxlib_stream_earthmodel_from_lmem(maxlib, dfeEarthmodels);
    maxlib_stream_to_lmem(maxlib, "nextW", dfePrevArray);
```

Every few timesteps, we want read back the current status of the receiver section of the wavefield to see the progress of the simulation. In our case, we are reading a vertical plane from the middle of the domain which slices through the center of the source stimulus, so we specify the plane each timestep:

```
        data = ( iteration  % iterationsPerFrame == 0) ? receiver : NULL;
        maxlib_stream_region_to_host(
            maxlib,
            "receiver",
            data,
            sourceX,
            0,
            0,
            sourceX + 1,
            ny,
            nz);
```

Calling `maxlib_run` executes the timestep, which will return once all of the computation and data transfer to and from the DFEs has completed:

```
        // Run timestep
        maxlib_run(maxlib);
```

The final task at the end of each timestep is to swap our wavefield buffer pointers such that the output wavefield becomes the input to the next timestep:

```
        // Swap wavefield buffers
        maxlib_lmem_array tmpArray = dfeCurrArray;
        dfeCurrArray = dfePrevArray;
        dfePrevArray = tmpArray;
```

This simply swaps the pointers to the data buffers in the LMem on the DFE: no data copying is performed by either the CPU or the DFE.

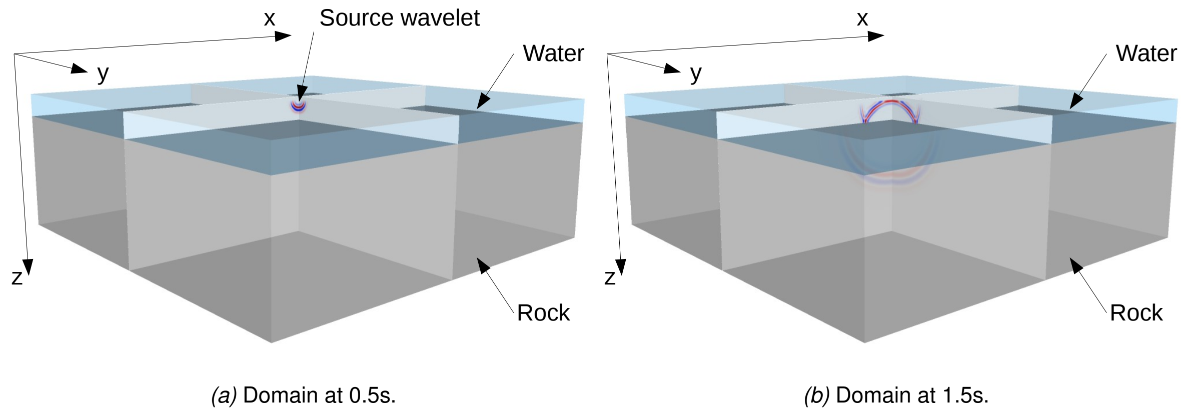
2.7 Running the Simulator

The same CPU code is used to communicate with both a real DFE and a simulated DFE. The simulated system runs as a separate process that the host code communicates with as if it were a real DFE. The simulated system has the advantages of much faster build times and greater visibility into the design via watch nodes for debugging.

To build and run the FDKernel for simulation, right-click on the Simulation Run Rule and select Run.

2.8 Output Wavefields

The simple earth model used has two layers: the first with a velocity of 1500 m/s to a depth of 800 meters (40 points) and the second with a velocity of 4482 m/s for the rest of the volume. *Figure 6* shows 2 slices of the wavefield output overlaid onto a 3D representation of the two-layer earth model.



(a) Domain at 0.5s. *(b)* Domain at 1.5s.

Figure 6: 3D domain showing receiver slices and earth model boundary.

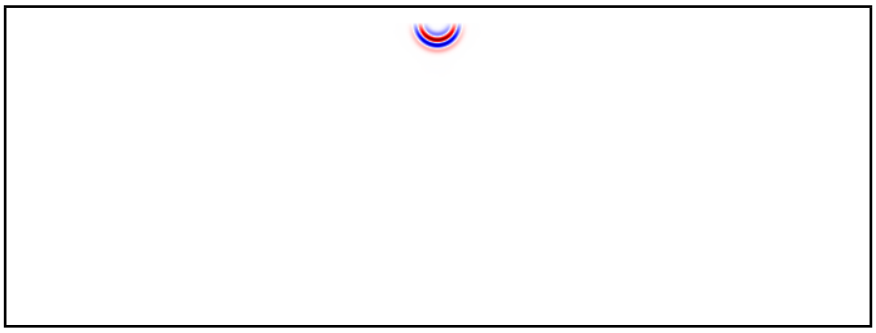
Figure 7 shows images of the output of the example for the receiver plane at four different timesteps.

Listing 5: Acoustic Forward Modeling Host Code - set up (SimpleFDCpuCode.c).

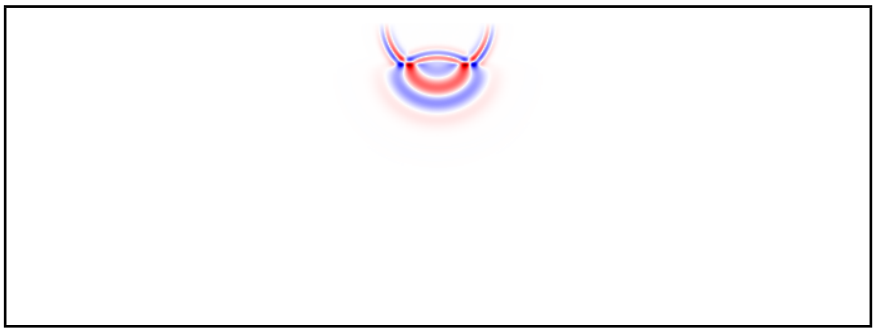
```
/**
 * Document: MaxGenFD Tutorial (maxgenfd-tutorial.pdf)
 * Example: 1      Name: Simple FD
 * MaxFile name: SimpleFD
 * Summary:
 *      Configures settings for the sponge and number of iterations. Creates
 *      some initial data which is loaded to LMem. The simulation is then
 *      computed for a fixed number of iterations.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "maxlibfd.h"
#include " file_utils .h"
#include "data_gen.h"

int main()
{

    // printf ( "IS SIM: %d", SimpleFD_IS_SIMULATION);
    const int nx = maxlib_is_simulation () ? 66 : 676;
    const int ny = maxlib_is_simulation () ? 66 : 676;
    const int nz = maxlib_is_simulation () ? 66 : 240;
    int n = nx * ny * nz;
    const int sourceX = nx / 2, sourceY = ny / 2, sourceZ = 7; // Source coordinates

    const int spongeWidth = 50; // Width of the sponge in points
    const int spongeActiveWidth = maxlib_is_simulation() ? 10 : 50; // Width of the sponge in use
    // Bit mask for the sides of the domain to sponge (all sides except z=0)
    const int spongeMask = MAXLIB_BOUNDARY_ALL ^ MAXLIB_BOUNDARY_Z0;

    const int numIterations = maxlib_is_simulation () ? 128 : 2000;
    const int iterationsPerFrame = maxlib_is_simulation () ? 1 : 50;
    const float deltaDistance = 20.0f; // Delta distance in metres
    const float deltaTime = 0.002f; // Delta time in seconds
    const float fpeak = 5; // Peak frequency of source wavelet

    float *dvv; // Velocity earth model
    float *zeroWavefield; // Empty wavefield to initialize simulation
    float *receiver; // Receiver
    float source[3 * 3 * 3]; // Source stimulus
    float *sponge; // Sponge

    const char *outputFilename = "output";

    maxlib_context maxlib = maxlib_open(nx, ny, nz); // Initialise maxlib with dimensions of domain
    maxlib_print_status (maxlib);

    dvv = create_2layer_earth_model_vel(nx, ny, nz); // Set up earth model
    preprocess_earth_model_dvv(dvv, nx, ny, nz, deltaDistance, deltaTime);

    maxlib_earthmodel em = maxlib_earthmodel_create_in_memory(nx, ny, nz);

    printf ("Compressing earth model\n");
    maxlib_earthmodel_set_data(em, "dvv", dvv);
    free(dvv); // We don't need the original earth model any more

    // Allocate memory on the MaxCard for the wavefields and earth model
    maxlib_lmem_array dfeCurrArray = maxlib_lmem_alloc_wavefield(maxlib);
    maxlib_lmem_array dfePrevArray = maxlib_lmem_alloc_wavefield(maxlib);
    maxlib_lmem_array dfeEarthmodels = maxlib_lmem_alloc_earthmodel(maxlib);

    zeroWavefield = malloc(n * sizeof(float));
```

Listing 6: Acoustic Forward Modeling Host Code - execution of timesteps (SimpleFDCpuCode.c).

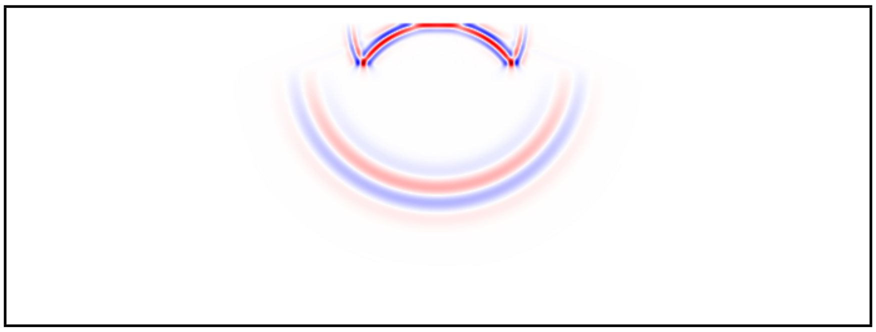
```
        zeroWavefield[i] = 0.0f;

    printf ("Loading curr wavefield to DFE\n");
    maxlib_lmem_load_wavefield(maxlib, dfeCurrArray, zeroWavefield);
    printf ("Loading prev wavefield to DFE\n");
    maxlib_lmem_load_wavefield(maxlib, dfePrevArray, zeroWavefield);
    printf ("Loading earth model to DFE\n");
    maxlib_lmem_load_earthmodel(maxlib, dfeEarthmodels, em);
    maxlib_earthmodel_release(em);

    maxlib_set_scalar_flag (maxlib, "absorb", spongeMask); // Set bit mask
    sponge = create_simple_sponge(spongeWidth, spongeActiveWidth); // Create sponge
    printf ("Loading sponge to DFE\n");
    for (int i = 0; i < spongeWidth; i++) // Set the sponge on the MaxCard
        maxlib_set_mapped_memory_f(maxlib, "sponge", i, sponge[i]);

    receiver = malloc(ny * nz * sizeof(float)); // Allocate memory for receiver plane
    FILE* output = fopen(outputFilename, "w"); // Output file for the receiver
    // Create a header file for the output file
    if (create_header_file(outputFilename, nz, ny, 1, "Y", "Z", "Time"))
        exit(1);

    // Execute timesteps
    for (int iteration = 0; iteration < numIterations; iteration++) {
        printf (" Iteration %d / %d\n", iteration, numIterations);

        float time = iteration * deltaTime;

        if (time <= 2.0f / fpeak)
            gen_source(source, time, fpeak); // Generate source wavelet

        float* data = (time <= 2.0f / fpeak) ? source : NULL;
        maxlib_stream_region_from_host(
            maxlib,
            "source",
            data,
            sourceX -1,
            sourceY -1,
            sourceZ -1,
            sourceX + 2,
            sourceY + 2,
            sourceZ + 2);

        // Set up inputs and output arrays for this timestep
        maxlib_stream_from_lmem(maxlib, "currW", dfeCurrArray);
        maxlib_stream_from_lmem(maxlib, "prevW", dfePrevArray);
        maxlib_stream_earthmodel_from_lmem(maxlib, dfeEarthmodels);
        maxlib_stream_to_lmem(maxlib, "nextW", dfePrevArray);

        // Read back the receiver every iterationsPerFrame iterations
        data = ( iteration % iterationsPerFrame == 0) ? receiver : NULL;
        maxlib_stream_region_to_host(
            maxlib,
            "receiver",
            data,
            sourceX,
            0,
            0,
            sourceX + 1,
            ny,
            nz);

        // Run timestep
        maxlib_run(maxlib);

        // Write out the receiver data to the output file
        if ( iteration % iterationsPerFrame == 0) {
```



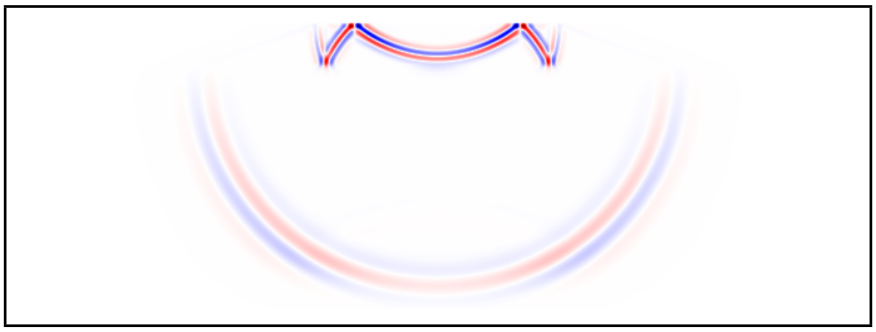
(a) t=0.5s



(b) t=1.0s



(c) t=1.5s



(d) t=2.0s

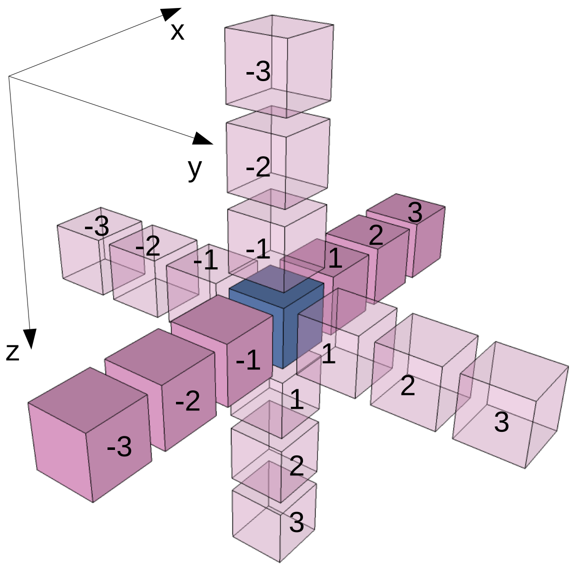
Figure 7: Output from using MaxGenFD for Acoustic Forward Modeling through a simple earth model.

3 Convolution

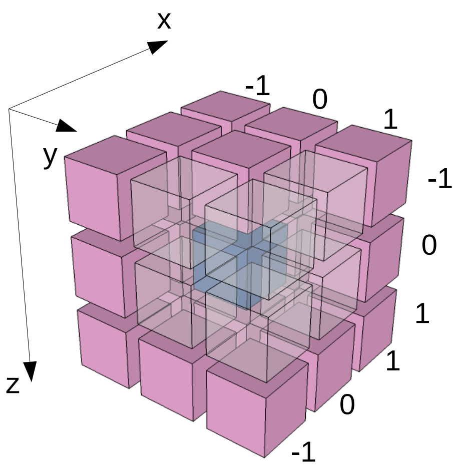
The core of any MaxGenFD application is the convolution operation. In a software implementation, the convolution is where most of the CPU time is spent in a finite difference application. The convolution operation is also where the bulk of the resources of a MaxGenFD design are used. MaxGenFD allows the convolution operation to be defined very succinctly and is one of the key areas where MaxGenFD raises the level of abstraction over both a MaxCompiler implementation and most software implementations.

3.1 Stencils

MaxGenFD supports three types of stencil: fixed 1D stencils, variable 1D stencils and cube stencils.



(a) 7-point 1D stencil applied in three dimensions to implement a star stencil around the central point (in blue).



(b) 3x3x3 cube stencil around the central point (in blue).

Figure 8: 1D and cube stencils.

3.1.1 Fixed 1D Stencils

Fixed 1D stencils are stencils with coefficients fixed at build-time that can be applied in one or more dimensions. If the stencil contains symmetry, then this will automatically be optimized to reduce the number of multiplication operations that need to be performed on the DFE.

A fixed 1D stencil is created using the `fixedStencil` method on the `FDKernel` and is stored in a `Stencil` object:

Stencil *fixedStencil (**int** min, **int** max, **double[]** coeffs, **double** outputScale)*

The first two integer arguments, `min` and `max`, specify the start and end positions of the stencil relative to the central point at position `0`. The third argument is the coefficient values themselves, specified as an array of double-precision floating-point values.

`outputScale` is a scale factor that describes the expected approximate maximum relative magnitude of the output values from this stencil in terms of the input values. This should be a positive value but can be greater than or less than 1. For example, a scale of 1.0 means that the output number range is similar to the input number range, 2.0 means the output is twice as large, 0.5 means it is half as large. This is used by MaxGenFD to optimize the data types and arithmetic used in the FDKernel (only applies to fixed-point FDKernels: see *subsection 7.8* for more detail on fixed-point scaling).

If you are uncertain what scale factor would be appropriate to use, specifying 0.0 will cause MaxGenFD to use a safe conservative value.

In our Acoustic Forward Modeling example, we declare a 13-element array, starting at location −6 and ending at +6 relative to the central point (0). We expect the result of our convolution operation to be no larger than 8 times the input:

```
private final int stencilSize = 13;
private final Stencil stencil = fixedStencil (
    -stencilSize / 2,
    stencilSize / 2,
    new double[] { -0.00003006253006249946f, 0.0005194805194800871f,
            -0.004464285714282842f, 0.02645502645501459f,
            -0.133928571428538f, 0.857142857142776f, -1.49138888888878f,
            0.857142857142776f, -0.133928571428538f, 0.02645502645501459f,
            -0.004464285714282842f, 0.0005194805194800871f,
            -0.00003006253006249946f },
    8.0);
```

3.1.2 Variable 1D Stencils

A variable 1D stencil is applied in the same way as a fixed 1D stencil, but in this case the coefficients can be changed at run-time. This can be used, for example, to have different coefficients at different points in the domain.

The declaration of a variable 1D stencil is similar to a fixed 1D stencil:

```
Stencil variableStencil (int min, int max, double outputScale, FDVar... coefficients)
```

The coefficients are made up of an `FDVar` stream for each coefficient, where the number of streams is equal to the size of the stencil (though of course the same stream may be supplied, possibly negated, to different coefficients). These can be passed in as an array of `FDVars`. These can be fed from, for example, a mapped ROM or scalar inputs, or can be calculated in the FDKernel itself.

MaxGenFD cannot apply the same level of optimization to variable stencils as to fixed stencils, so fixed stencils are recommended wherever possible.

3.1.3 Cube Stencils

A cube stencil provides a coefficient for every point in a cubic volume (fixed at build-time). The range of the stencil is specified by the start and end positions of the stencil relative to the central point at position 0 for each dimension:

```
Stencil cubeStencil(int minX, int maxX, int minY, int maxY, int minZ, int maxZ, double[][][] coeffs, double outputScale)
```

The coefficients are provided via a 3D Java array, where the dimensions are in the order ([z][y][x]), as shown in the example in *Listing 7*.

Listing 7: Definition of a cube stencil.

```
double [][][] coeffs = new double[][][]
{
    {
        { 1, 2, 3 }, // z0 y0
        { 4, 5, 6 }, // z0 y1
        { 7, 8, 9 }  // z0 y2
    },
    {
        { 10, 11, 12 }, // z1 y0
        { 13, 14, 15 }, // z1 y1
        { 16, 17, 18 }  // z1 y2
    },
    {
        { 19, 20, 21 }, // z2 y0
        { 22, 23, 24 }, // z2 y1
        { 25, 26, 27 }  // z2 y2
    },
};
Stencil myStencil = cubeStencil(-1, 1, -1, 1, -1, 1, coeffs, 27.0);
```

3.2 Performing the Convolution

The convolution is achieved using a single call to the `convolve` method, which takes a stream to convolve (`toConvolve`), the axes about which to perform the convolution (`axes`) and the stencil as arguments:

```
FDVar convolve(FDVar toConvolve, ConvolveAxes axes, Stencil stencil)
```

The `axes` argument is an item from the enumerated type `ConvolveAxes`, which is used to specify that the stencil should be applied to one (`X`, `Y` or `Z`), two (`XY`) or three axes (`XYZ`).

Note that a cube stencil must be applied to all three dimensions simultaneously using `ConvolveAxes.XYZ`.

The output of the convolution is another stream of data.

In our Acoustic Forward Modeling example, we apply our stencil in all three dimensions to give our result `laplacian` (in blue):

```
41        FDVar laplacian = convolve(curr, ConvolveAxes.XYZ, stencil);
```

The output of the convolution is then used in the update of the wave equation:

```
        FDVar result = curr * 2 - prev + dvv * laplacian + source;
```

The call to the `convolve` method is mathematically equivalent to the hand-written MaxCompiler code shown in *Listing 8*.

Listing 8: Manual implementation of convolution of 1D coefficients in 3 dimensions.

```
// Stream for our convolution output
FDVar laplacian = constant.fdvar(0.0);

// Convolve in x
for (int dx = -6; dx <= 6; dx++)
    laplacian += stream.offset(curr, dx, 0, 0) * coeffs[dx+6];

// Convolve in y
for (int dy = -6; dy <= 6; dy++)
    laplacian += stream.offset(curr, 0, dy, 0) * coeffs[dy+6];

// Convolve in z
for (int dz = -6; dz <= 6; dz++)
    laplacian += stream.offset(curr, 0, 0, dz) * coeffs[dz+6];

// Check we are not in the boundary where we don't have enough data
FDVar is_edge = domain.getX() > 5 & domain.getY() > 5 & domain.getZ() > 5
& domain.getXD() > 5 & domain.getXD() > 5 & domain.getZD() > 5;

// Output the convolution result if we are not in the boundary, otherwise 0
laplacian = is_edge ? laplacian : constant.fdvar(0.0);
```

In *Listing 8*, we make use of `domain.getX`, `domain.getY` and `domain.getZ` to give us `FDVar` streams for the distance of the current point in the wavefield from the start of the domain in each dimension and `domain.getXD`, `domain.getYD` and `domain.getZD` for the distance from the end of the domain. `domain.getNX`, `domain.getNY` and `domain.getNZ` can be used to get the dimensions of the domain as set by the host software.

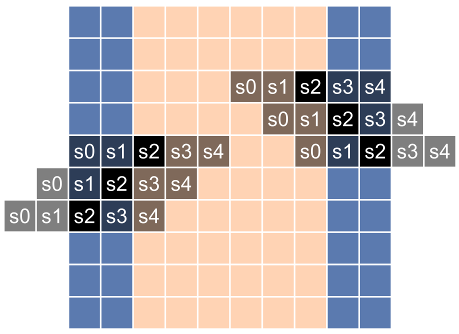
The `convolve` method, however, automatically performs a number of optimizations that are tedious to perform by hand.

If you intend to apply a stencil in three dimensions, it is more efficient to do so on all 3 dimensions in a single call to `convolve`, rather than three successive 1-D convolutions, because the stencil can be optimized across the dimensions. This can reduce the number of multiplies in a convolution by at least a factor of three.

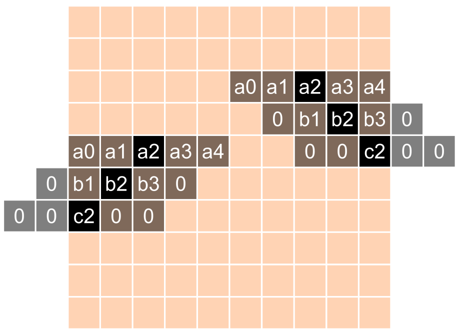
3.3 Roll-on/Roll-off

With an n-point stencil, a layer of $floor(n/2)$ points around each edge of the domain cannot be convolved because there is insufficient wavefield data for one or more axes of the stencil. This is shown in 2D in *Figure 9a*. For each of these points, the convolution will automatically output 0.

To make better use of the available data, MaxGenFD allows the declaration of an array stencils to provide "roll-on" and "roll-off" at the edges of the domain. This allows modeling up to the edge of the domain by switching to smaller and smaller stencils as we approach the edge of the domain. *Figure 9b* shows the application of multiple stencils to provide roll-on/roll-off.



(a) A single stencil.



(b) Symmetrical roll-on/roll-off stencils.

Figure 9: Stencils applied to a 2D domain in 1 dimension. Inputs for which no valid output can be calculated are in blue.

Roll-on/roll-off is not available for cube stencils.

Automatic roll-on/roll-off is not available for variable 1D stencils as you must implement the coefficients yourself.

3.3.1 Symmetric Roll-on/Roll-off

A variant of `fixedStencil` takes a two-dimensional $floor(n/2) + 1$ by n array of coefficients as its `coeffs` argument:

```
Stencil fixedStencil(int min, int max, double[][] coeffs, double outputScale)
```

Each element in the first dimension of `coeffs` is a stencil, where `coeffs[0]` is the stencil for use when not in the roll-on/roll-off area and `coeffs[1]` to `coeffs[floor(n/2)]` are the roll-on/roll-off stencils as the wavefield approaches the boundary. For each stencil in the array, MaxGenFD will check that there are the correct number of zeros at each side of the supplied array and produce an error if not.

An example declaration of the multiple stencils for roll-on/roll-off shown in *Figure 9b* is shown below:

```
double[][] coeffs = new double[][]
{
    { a2,  a1,  a0,  a1,  a2},
    {0.0,  b1,  b0,  b1, 0.0},
    {0.0, 0.0,  c0, 0.0, 0.0}
};
Stencil stencil = fixedStencil (-2, 2, coeffs, 1.0);
```

The choice of whether to use roll-on/roll-off is a trade-off between the value of fully utilizing the points in the domain versus the computational complexity of working out the roll-on roll-off conditions.

3.3.2 Asymmetric Roll-on/Roll-off

A further 1D fixed stencil variant, `asymmetricStencil`, creates a 1D fixed stencil with asymmetric roll-on/roll-off:

```
Stencil asymmetricStencil(int min, int max, double[][] coeffsRollOn, double[] coeffsInterior , double[][] coeffsRollOff, double
        outputScale)
```

`coeffsInterior` is an array of coefficients specifying the stencil when not in the roll-on/roll-off area. `coeffsRollOn` is a two dimensional array specifying stencils to apply in the roll-on condition (when the coordinate is close to zero) and `coeffsRollOff` specifies the stencils for the roll-off condition (when the coordinate is close to the maximum).

The stencil range is specified by the `min` and `max` indexes that should be accessed relative to the current position. For example, consider this 4-point asymmetric stencil with a range from -2 to +1:

```
double rollOn [][]  = { { 0.0, a, b,  c  }, { 0.0, 0.0, b, c } };
double rollOff [][] = { {   d, e, f, 0.0 } };
double interior []  =   {   p, q, r,   s };

Stencil s = asymmetricStencil(-2, 1, rollOn, interior , rollOff , 0.0);
```

Because there are 2 points of roll-on and only 1 of roll-off, the roll-on array is larger than the roll-off array. Over a problem size of 6 points ($x = 0$ to $x = 5$), this stencil will take the following values:

	Stencil Index			
x	-2	-1	0	+1
0	0.0	0.0	b	c
1	0.0	a	b	c
2	p	q	r	s
3	p	q	r	s
4	p	q	r	s
5	d	e	f	0.0

3.4 Halos

When the domain is decomposed into multiple blocks, a convolution operation on one block will require inputs from one or more neighboring blocks for points on the boundary of the current block. The number of points that are required from adjacent blocks is the **halo** of the operator.

Figure 10a shows the halo for a 1D stencil applied in two dimensions and *Figure 10b* shows a cube stencil applied to a 16 by 16 2D slice from a larger domain. The area labeled “external halo” shows the points that are required from the adjacent block for the convolution performed in this block and the area labeled “internal halo” shows the points that are required from this block by adjacent blocks.

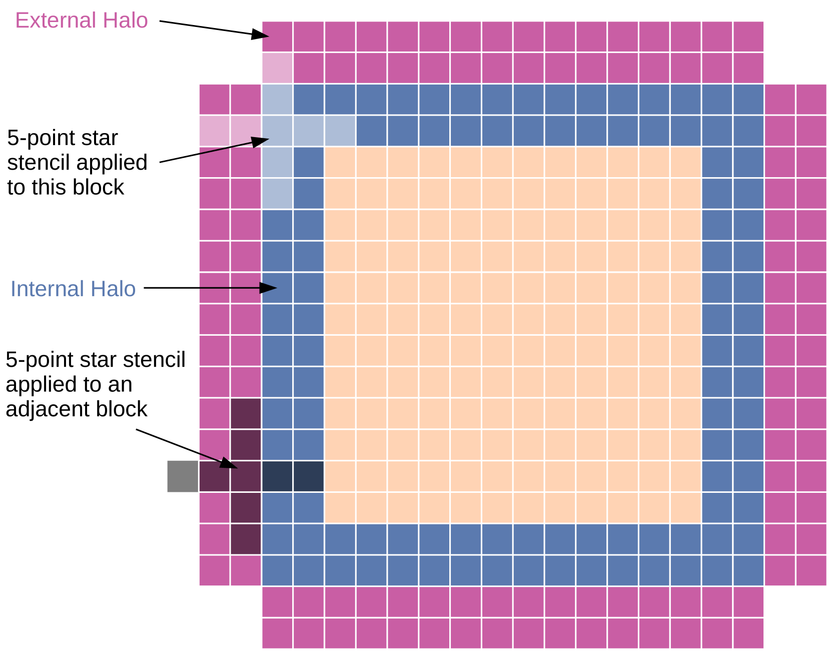
For a cube stencil, points from all of the immediately neighboring blocks are required, including diagonally neighboring blocks. For a convolution with a star stencil, the points from the diagonally neighboring blocks are not required, though in MaxGenFD these points are in fact exchanged to decrease the complexity, and thus increase performance, of the hardware.

When a domain is decomposed across multiple DFEs in a system, halos must be exchanged between DFEs at the boundaries of adjacent sub-domains. MaxGenFD automatically divides the domain between the available DFEs in the system and exchanges the halos between these engines: the only interaction required of the user is defining the halo size in the FDKernel.

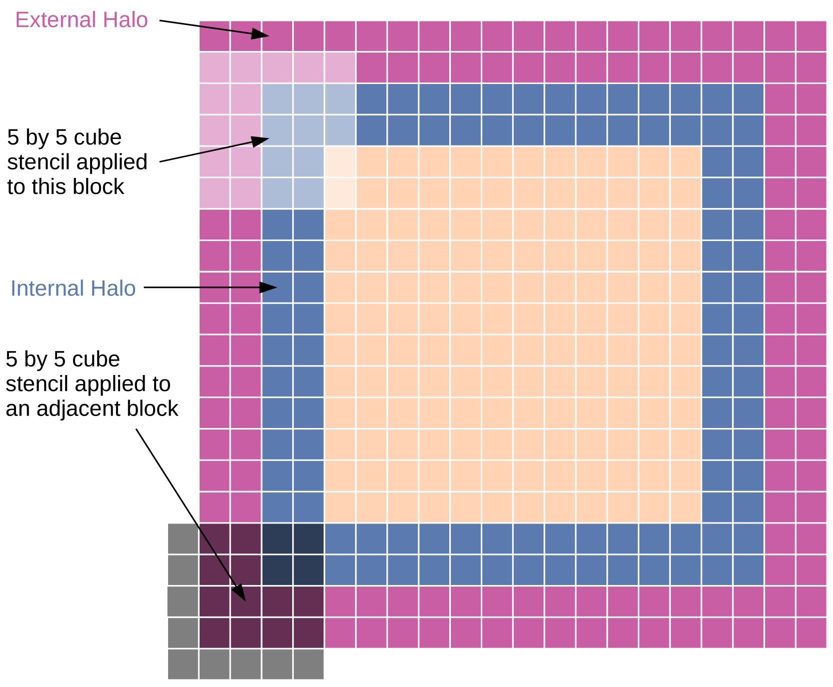
An FDKernel processes the sub-domain allocated to it by the CPU in multiple blocks, so the halo size is also important within a single DFE for the FDKernel to correctly read in external halos when processing each block.

Every convolution applied to a stream *in the same axis* in an FDKernel increases the halo required for the stream by the operator halo. For example, an 11-point operator convolved once gives a 5-point halo; chaining another convolution in the same FDKernel gives a 10-point halo.

Note that declaring a halo size that is too small can lead to strange errors that are difficult to debug.


External Halo
5-point star stencil applied to this block
Internal Halo
5-point star stencil applied to an adjacent block

(a) 1D stencil applied in two dimensions.


External Halo
5 by 5 cube stencil applied to this block
Internal Halo
5 by 5 cube stencil applied to an adjacent block

(b) Cube stencil.

Figure 10: A 16 by 16 2D slice from a larger domain showing the internal and external halos.

4 Earth Models

MaxGenFD allows the user to specify multiple earth model parameters, representing different properties of the domain, for example velocity and density. Each earth model parameter has a value for every point in the domain.

4.1 Stream Inputs

An earth model parameter input is declared as an `FDVar` stream input in the FDKernel:

```
FDVar io.earthModelInput(String name, double max, int halo)
```

The first argument is the name for the input stream that will be used in the host software. `max` is the maximum value that the earth model parameter contains (this only affects fixed-point FDKernels). The `halo` argument defines the number of points from the earth model parameter that will be required from beyond the edge of the sub-domain allocated to this device when the domain is decomposed over multiple devices (this need only be non-zero if the earth model parameter forms part of an input to a convolution operation).

The earth model parameter can be used as a normal stream:

```
35        FDVar dvv = io.earthModelInput("dvv", 1 / 4.0, 0); // Earth model
```

```
42        FDVar result = curr * 2 - prev + dvv * laplacian + source;
```

4.2 Setting Earth Model Types

In the FDConfig object for the FDKernel, the storage type for the earth model parameter needs to be set. This is set with a `StorageType` object (see *section 7* for more information on storage types).

```
void setEarthModelStorageType(String name, StorageType type)
```

If a compressed storage type is used, then the computation type for the earth model must be set in order for MaxGenFD to know how to expand the compressed values correctly:

```
void setEarthModelComputeType(String name, HWFloat type)
```

4.3 Setting the Earth Model from the Host

The earth model values are initially loaded in host memory and then transferred to the accelerator card. Earth models are managed on the host via a `maxlib_earthmodel` instance.

An instance of this structure can be created using `maxlib_earthmodel_create`:

```
maxlib_earthmodel maxlib_earthmodel_create_in_memory(int nx, int ny, int nz);
```

section 9 deals with more complex management of earth models for large domains.

The earth model parameters must first be loaded into host memory as an array of floating-point values. This array of data can then be added to the `maxlib_earthmodel` instance using `maxlib_earthmodel_set_data`:

```
maxlib_earthmodel_set_data(maxlib_earthmodel em, const char *field_name, const float *data);
```

This call will compress the earth model data to the required format for the earth model parameter.

For example, to load the earth model data for the velocity earth model parameter called `dvv` in the FDKernel for Example 1, we create a floating point array in memory (`dvv`) and then set this as the data for the single earth model parameter `"dvv"`:

```
dvv = create_2layer_earth_model_vel(nx, ny, nz); // Set up earth model
preprocess_earth_model_dvv(dvv, nx, ny, nz, deltaDistance, deltaTime);

maxlib_earthmodel em = maxlib_earthmodel_create_in_memory(nx, ny, nz);

 printf ("Compressing earth model\n");
maxlib_earthmodel_set_data(em, "dvv", dvv);
free(dvv); // We don't need the original earth model any more
```

If we have two earth models, one for velocity and one for density, then we would write something like this:

```
float *vel = malloc(nx*ny*nz * sizeof(float));
float *den = malloc(nx*ny*nz * sizeof(float));
load_earth_model_vel(nx, ny, nz, vel);
load_earth_model_den(nx, ny, nz, den);

maxlib_earthmodel em = maxlib_earthmodel_create_in_memory(nx, ny, nz);
maxlib_earthmodel_set_data(em, "dvv", vel);
maxlib_earthmodel_set_data(em, "den", den);
```

The storage space in LMem for the earth model must be allocated on the DFE using a `maxlib_lmem_array` handle, as in our example:

```
maxlib_lmem_array dfeEarthmodels = maxlib_lmem_alloc_earthmodel(maxlib);
```

Using this handle, earth model data can then be loaded into the DFE. This is done with a call to `maxlib_lmem_load_earthmodel`, for example:

```
 printf ("Loading earth model to DFE\n");
maxlib_lmem_load_earthmodel(maxlib, dfeEarthmodels, em);
```

The earth model is no longer required in the host memory, so we can release it:

```
maxlib_earthmodel_release(em);
```

At this point, all the data is now setup on the card. For each timestep, we need to instruct the FDKernel to stream the earth model parameters into the input streams. This is done using a call to maxlib_stream_earthmodel_from_lmem, for example:

```
        maxlib_stream_earthmodel_from_lmem(maxlib, dfeEarthmodels);
        maxlib_stream_to_lmem(maxlib, "nextW", dfePrevArray);

        // Read back the receiver every iterationsPerFrame iterations
        data = ( iteration  % iterationsPerFrame == 0) ? receiver : NULL;
        maxlib_stream_region_to_host(
            maxlib,
            "receiver",
            data,
            sourceX,
            0,
            0,
            sourceX + 1,
            ny,
            nz);

        // Run timestep
        maxlib_run(maxlib);

        // Write out the receiver data to the output file
        if ( iteration  % iterationsPerFrame == 0) {
            fwrite (receiver, sizeof(float), ny * nz, output);
            fflush (output);
        }

        // Swap wavefield buffers
        maxlib_lmem_array tmpArray = dfeCurrArray;
        dfeCurrArray = dfePrevArray;
        dfePrevArray = tmpArray;
    }

    // Clean up
    fclose(output);
    maxlib_close(maxlib);
    return 0;
}
```

4.4 Derived Earth Model Parameters

An earth model parameter may be used in one or more mathematical functions in the FDKernel. Storing the earth model parameter in memory and then calculating the mathematical functions in the DFE in each pipeline may use a significant number of resources.

MaxGenFD provides expressions for common mathematical functions on earth model parameters that can be used in the FDKernel. These expressions, rather than building logic in the Kernel to calculate these functions, build a look-up table to hold pre-calculated values. These values are calculated by the CPU automatically from the floating-point values supplied at run-time and loaded into the DFE. We refer to these look-up tables containing the pre-calculated values as “derived” earth model parameters.

The expressions are specified in the FDKernel using objects of class FDExpr. We first declare a primitive earth model parameter, which is the earth model parameter from which the derived model will be calculated, using the method makeEarthModelPrimitive:

```
FDExpr makeEarthModelPrimitive(String name)
```

A primitive earth model parameter will never actually be transferred over to the DFE: only its derivatives will be transferred.

FDExpr objects can be manipulated via the overloaded operators +, -, * and / and via a library of common transcendental functions (see *Table 1*).

We can specify our derivatives using an extended version of the `earthModelInput` method with an extra argument for the mathematical expression:

***FDVar** earthModelInput(**String** name, **double** max, **int** halo, **FDExpr** value)*

✱ Note that derived earth model parameters only save resources with table-compression enabled.

Function	Description
`sin(FDExpr x)`	Sine of `x`
`sinh(FDExpr x)`	Hyperbolic sine of `x`
`asin(FDExpr x)`	Arc sine of `x`
`cos(FDExpr x)`	Cosine of `x`
`cosh(FDExpr x)`	Hyperbolic cosine of `x`
`acos(FDExpr x)`	Arc cosine of `x`
`tan(FDExpr x)`	Tangent of `x`
`tanh(FDExpr x)`	Hyperbolic tangent of `x`
`atan(FDExpr x)`	Arc tangent of `x`

`atan2(FDExpr x, FDExpr y)`
`atan2(double x, FDExpr y)`
`atan2(FDExpr x, double y)`
	Arc tangent of `y/x`, using the signs of the arguments to compute the quadrant of the return value

`ceil(FDExpr x)`	Returns the smallest integer not less than `x`
`floor(FDExpr x)`	Returns the largest integer not greater than `x`
`log(FDExpr x)`	Natural logarithm (to base e) of `x`
`log10(FDExpr x)`	Common logarithm (to base 10) of `x`
`sqrt(FDExpr x)`	Square root of `x`

`pow(double x, FDExpr y)`
`pow(FDExpr x, double y)`
`pow(FDExpr x, FDExpr y)`
	`x` to the power `y`

`mod(FDExpr x, FDExpr y)`	Returns the remainder of `x/y`
`exp(FDExpr x)`	Returns e raised to the power `x`
`sqrt(FDExpr x)`	Square root of `x`

Table 1: Transcendental functions available on `FDExpr` objects.

To see how derived earth model parameters are used, consider the example of “tilt”, where we have an earth model parameter *theta* (θ), to which we apply three different functions, $\sin(2\theta)$, $\sin^2\theta$ and $\cos^2\theta$.

To do this, we first create the earth model primitive as “theta”:

```
FDExpr theta = io.makeEarthModelPrimitive("theta");
```

We then create the three functions of theta as earth model inputs using `io.earthModelInput` with an `FDExpr` argument:

```
FDVar sin_2_theta = io.earthModelInput("sin_2_theta",  1.0,  0,  FDExpr.sin(2*theta));
FDVar sin_sq_theta = io.earthModelInput("sin_sq_theta",  1.0,  0,  FDExpr.sin(theta)*FDExpr.sin(theta));
FDVar cos_sq_theta = io.earthModelInput("cos_sq_theta", 1.0,  0,  FDExpr.cos(theta)*FDExpr.cos(theta));
```

The `FDVars` can be used as normal in the FDKernel, just as any other earth model parameter.

If we are using table compression on our earth model (see *subsubsection 7.2.1* for more details on table compression of earth models), we only need to specify the storage type for the earth model parameter primitive. For example:

```
config.setEarthModelStorageType("theta", StorageType.compressedTable(5));
```

In this table-compressed case, the same index into the table will be used for all of the derived earth model parameters, reducing both the LMem storage requirement and the number of independent streams.

We specify the earth model compute types for each of the derived earth model parameters in our configuration object:

```
HWFloat t = HWTypeFactory.hwFloat(8, 24);
config.setEarthModelComputeType("sin_2_theta", t);
config.setEarthModelComputeType("sin_sq_theta", t);
config.setEarthModelComputeType("cos_sq_theta", t);
```

Each of the derived earth model parameters can have a different compute type, if required, in fixed-point FDKernels:

```
config.setEarthModelComputeType("sin_2_theta", 22);    // 22-bit  fixed-point  data
config.setEarthModelComputeType("sin_sq_theta", 18);   // 18-bit  fixed-point  data
config.setEarthModelComputeType("cos_sq_theta", 18);   // 18-bit  fixed-point  data
```

5 Host Inputs and Outputs

This section introduces host inputs and outputs to the FDKernel, which provide efficient methods for injecting and reading back data. These host inputs and outputs do not need to be set every timestep: they can be enabled only when required. For example, a source wavelet may only be injected for a short period of time at the beginning of a simulation and a receiver plane may only be read every few timesteps.

We could send a complete wavefield of data every timestep with our wavelet at the center, but this would send a vast amount of redundant data. Reading back the complete wavefield is also an inefficient way of inspecting the data, especially as we are most likely to look only at a few slices of the wavefield.

5.1 Injecting Data into the FDKernel

A host input differs from a wavefield input in two important ways:

- It is injected directly into the Kernel during a timestep, rather than being copied into LMem to be streamed into the Kernel during the timestep.
- It is possible to inject a sub-volume of the domain, ensuring that copying the data to the card over the PCI Express bus does not slow down the processing of the timestep.

We specify a host input in the FDKernel source using the method `io.hostInput`:

***FDVar io**.hostInput(**String** name, **double** relMax, **int** halo)*

The argument `relMax` defines the maximum value that the input contains relative to other inputs. This is discussed in detail in *section 7*: for now we will just use `1.0`, as `relMax` has no effect for floating point numbers.

The `halo` argument defines the size of the halo for the input. A halo is required wherever an input is used as part of an input to a convolution as the FDKernel will require the input for the points surrounding the current point to generate the all of the inputs to the convolution.

A host input sub-volume forms a cuboid. The bounds of this cuboid are specified in the host software by providing a start coordinate (inclusive) and end coordinate (exclusive) of the sub-volume. These start and end values are the same as the values that we would use in a software loop to iterate through the sub-volume.

Figure 11 shows a host input sub-volume of a larger domain with the start and end coordinates of the sub-volume.

The function to stream in this cuboid specifies these coordinates:

***void** maxlib_stream_region_from_host (maxlib_context context, **const char** ∗input, **const void** ∗data, **int** x_start , **int** y_start , **int** z_start , **int** x_end, **int** y_end, **int** z_end)*

Supplying a `NULL` pointer as the `data` argument disables the host input for the current timestep.

✷ A host input can only have one input region from the host per timestep.

The FDKernel will automatically set the output of the input stream to `0` when the current position in the 3D domain being calculated in the FDKernel is outside the specified sub-volume for the input: this allows the stream to be added to another stream without having to check whether the input is currently

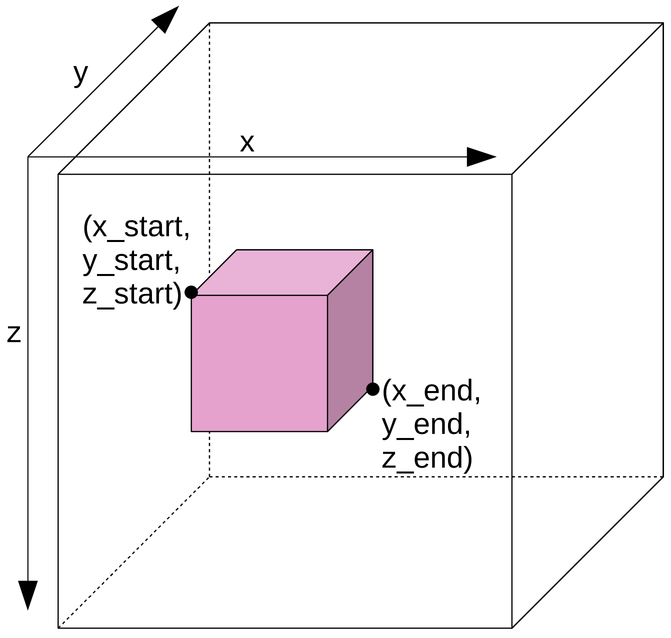


Figure 11: Sub-volume of the domain for a host input showing start and end coordinates.

inside the sub-volume. If the input stream is not to be added to another stream, for example if it is to replace a region of a wavefield input (see *subsection 5.3*), then there is a method to detect whether a stream is currently within its active region:

***FDVar** isHostInputActive(**String** name)*

`name` is the name of the host input stream. This method returns a Boolean stream which will be set to `1` if the current point is within the sub-volume and `0` if not.

In our Acoustic Forward Modeling example, we create a host input in the FDKernel called `source` with no halo (as our source will be applied after the convolution). We provide a relative maximum of `1.0`, but this does not apply to our floating-point FDKernel (see *section 7* for more details on this optimization).

```
36        FDVar source = io.hostInput("source", 1.0, 0); // Stimulus data
```

In the corresponding host code, we generate the source wavelet, a cube with a side of 3 points, for each timestep and then stream it into the FDKernel:

```
        if (time <= 2.0f / fpeak)
            gen_source(source, time, fpeak); // Generate source wavelet

        float* data = (time <= 2.0f / fpeak) ? source : NULL;
        maxlib_stream_region_from_host(
            maxlib,
            "source",
            data,
            sourceX -1,
            sourceY -1,
            sourceZ -1,
            sourceX + 2,
            sourceY + 2,
            sourceZ + 2);
```

In our case the center of the source wavelet is the mid-point in our 3x3x3 array, so we pass the start and end points of the source wavelet volume such that the center of the cube is at the point *(source_x, source_y, source_z)*.

A `NULL` pointer is used to disable the host input once the source wavelet is complete.

5.2 Reading Data from the FDKernel

We can also read back sub-volumes of the wavefields from the FDKernel: this is much more efficient, especially in large domains, than streaming out the whole wavefield to the host and then only reading the volumes of interest.

The output is declared in the FDKernel with a name to be referenced in the host software and the output stream to which to connect the host output:

```
void io.hostOutput(String name, FDVar out)
```

The location and dimensions of the volume to be read are set in the host software in the same manner as a host input, with start and end coordinates:

```
maxlib_stream_region_to_host (maxlib_context context, const char *output, void *data, int x_start, int y_start, int z_start, int
        x_end, int y_end, int z_end)
```

Supplying a `NULL` pointer as the `data` argument disables the host output.

We can read out any cuboid that we like: the commonest shape to read out is a plane, which is a cuboid with one dimension of only 1 point.

Figure 12 shows the start and end coordinates for an output plane.

In our Acoustic Forward Modeling example, we create a host output to read back the output of the wave equation from the `result` stream:

```
        io.hostOutput("receiver", result); // Receiver output to host
```

In the host software, we specify the receiver to be a plane with a constant value in the x-dimension,

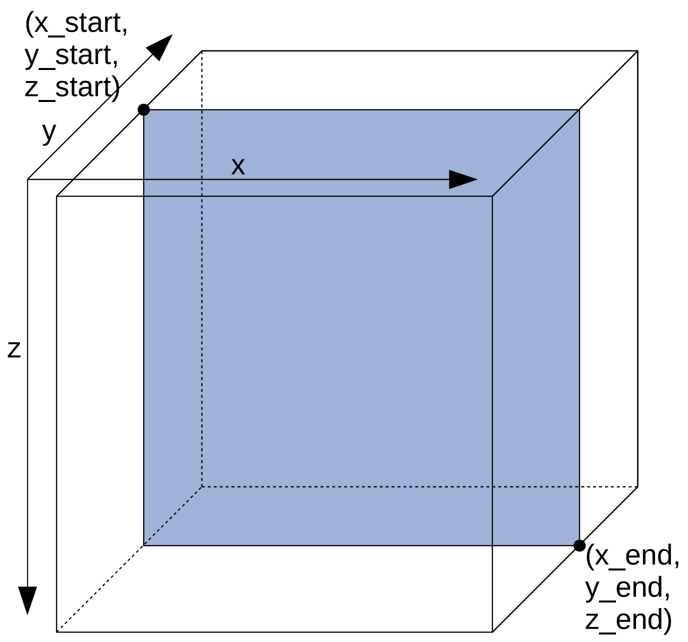


Figure 12: Sub-volume of the domain for a host output showing start and end coordinates.

slicing through the center of the source wavelet:

```
    data = ( iteration  % iterationsPerFrame == 0) ? receiver :  NULL;
    maxlib_stream_region_to_host(
        maxlib,
        "receiver",
        data,
        sourceX,
        0,
        0,
        sourceX + 1,
        ny,
        nz);
```

A `NULL` pointer is used to disable the host output so that we only read the receiver plane every few timesteps.

5.3 Hollow Cube Host IO

For some purposes, such as injecting and reading back boundaries for an RTM simulation, a cuboid host input or output is inadequate. For such purposes, hollow cubes can be streamed in and out of the FDKernel.

A host input or output declared in the FDKernel can be used to transfer either a solid or hollow cube: this is specified by the host code each execution step.

A host input or output can stream only one region, hollow or solid, per execution step.

A hollow cube is specified by an inner and outer cube. The inner and outer cubes are defined using two `maxlib_regions`, a C `struct` that specifies start and end coordinates for a cube:

```
typedef struct {
    int x_start, x_end;
    int y_start, y_end;
    int z_start, z_end;
} maxlib_region;
```

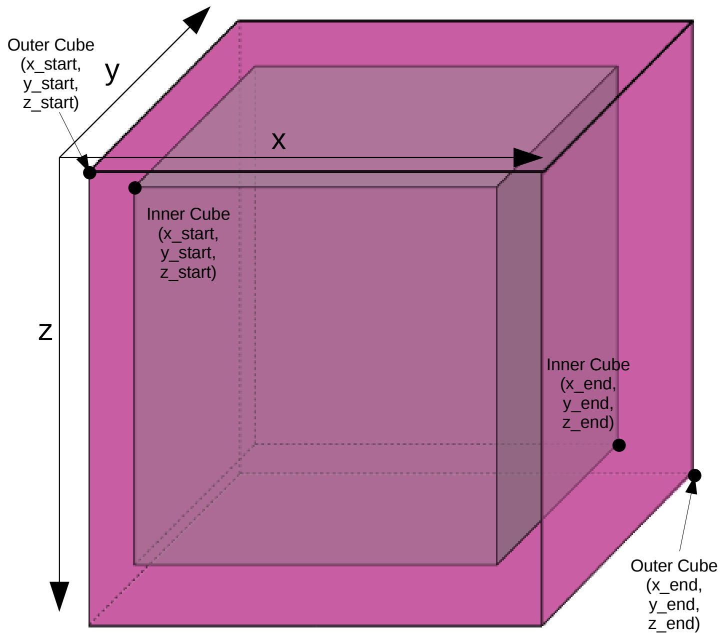

Outer Cube (x_start, y_start, z_start)
y
x
z
Inner Cube (x_start, y_start, z_start)
Inner Cube (x_end, y_end, z_end)
Outer Cube (x_end, y_end, z_end)

Figure 13: Hollow cube for a host input showing start and end coordinates for the inner and outer cubes.

Figure 13 shows a hollow cube with the start and end coordinates of the inner and outer cubes.

To determine the correct amount of memory to allocate to store a hollow cube, we can call `maxlib_hollow_cube_size` with the inner and outer cube regions, which will return the number of bytes to allocate:

```
size_t maxlib_hollow_cube_size(maxlib_context context, const char* input_name, const maxlib_region inner_cube, const maxlib_region
        outer_cube);
```

The two regions are then used to stream a host input or output in a similar manner to solid cuboid inputs and outputs:

```
void maxlib_stream_hollow_cube_from_host(maxlib_context context, const char* input_name, const void* data, const maxlib_region
        inner_cube, const maxlib_region outer_cube);

void maxlib_stream_hollow_cube_to_host(maxlib_context context, const char* output_name, void* data, const maxlib_region
        inner_cube, const maxlib_region outer_cube);
```

The `data` argument is a void pointer to the data for the hollow cube input or output. The layout in memory for this data is an optimally packed format for MaxGenFD to use internally.

By default, the data is streamed as single-precision floating-point values to and from the FPGA and automatically converted into the representation used within the FDKernel. Utility functions are provided in the `maxlib` to read from and write to a hollow cube using floating-point values:

```
float maxlib_get_hollow_cube_value(void* hollow_cube_data, maxlib_region inner_cube, maxlib_region outer_cube, int x, int y, int z);

void maxlib_set_hollow_cube_value(void* hollow_cube_data, maxlib_region inner_cube, maxlib_region outer_cube, int x, int y, int z,
        float value);
```

Streaming the hollow cube out in the internal representation (possibly storing it) and then streaming it back into the FDKernel again, for example in an RTM implementation, is often more efficient than converting it into floating-point data and back again. The automatic floating-point conversion can be disabled for a host input or output in the configuration object using the method `setHostDataFormat()`:

```
void setHostDataFormat(String name, HostDataFormat format)
```

`format` can be one of `HostDataFormat.FLOATING_POINT` or `HostDataFormat.INTERNAL`.

5.4 Scalar inputs

A scalar input to an FDKernel is used to transfer single operands between the host application and the FDKernel. These can be used either to control the computation or as values in the computation itself. When used for computation purposes, the value will be appropriately scaled for additions and subtractions, but not scaled for multiplications and divisions.

A scalar input is declared in the FDKernel using one of:

```
FDVar io.scalarInput(String name, double absMax, double relMax)
```

This takes both an absolute maximum value and a value relative to the other inputs to the FDKernel.

The type of the input is specified in the configuration object, using the appropriate method for floating or fixed point data:

```
void setScalarInputType(String name, HWFloat type)
void setScalarInputType(String name, int numBits)
```

For passing flags to control computation in the FDKernel, a scalar flag input is also available:

```
HWVar io.scalarFlagInput(String name)
```

To set the scalar input from the host code, there are two functions provided, one for integer flag values and one for floating point values:

```
void maxlib_set_scalar_flag  (maxlib_context context,  const char *name, uint64_t value)
void maxlib_set_scalar  (maxlib_context context,  const char *name, double value)
```

5.4.1 Scalar Input Example

As an example, we will modify our existing Acoustic Forward Modeling code from Example 1. Our new implementation is Example 2. In our original Example 1, we have a scaled velocity squared ($(\frac{v \times dt}{dd})^2$) as our earth model. We can modify this to pass in a value for dt^2/dd^2 to the FDKernel as a scalar input to be able to vary the delta time and delta distance for our simulation. We also now need to pass in the square of the velocity as our earth model.

Our earth model input is now velocity squared (`vel_sq`):

```
        FDVar vel_sq = io.earthModelInput("velSq", 10000 * 10000, 0); // Earth model
```

We specify our scalar input in the FDKernel:

```
        FDVar dttd = io.scalarInput("dttd", 5e-9, 1.0);
```

In the configuration object, we specify the type of our scalar input to be our single-precision floating-point type:

```
        // IEEE single-precision floating point
        DFEFloat t = DFETypeFactory.dfeFloat(8, 24);
```

```
        config.setScalarInputType("dttd", t);
```

In our host code, we load in the square of the velocity as our earth model:

```
velSq = create_2layer_earth_model_vel(nx, ny, nz); // Set up earth model
// Calculate the square of the earth model
for (size_t i = 0; i < n; i++) {
    velSq[i] = velSq[i] * velSq[i];
}

maxlib_earthmodel em = maxlib_earthmodel_create_in_memory(nx, ny, nz);

printf("Compressing earth model\n");
maxlib_earthmodel_set_data(em, "velSq", velSq);
```

Now we can vary delta distance and delta time for our simulation by passing in dt_2/d_2 via our scalar input:

```
    maxlib_set_scalar(maxlib, "dttd", deltaTime * deltaTime
        / (deltaDistance * deltaDistance));
```

A scalar input must be set before every timestep.

6 Boundaries

The simulation domain cannot extend infinitely, so we are left with discrete boundaries to our finite domain. These boundaries will affect the simulation results by giving unwanted reflections that would not occur in the the real world. We therefore need to compensate for these conditions somehow at the boundaries of the domain.

MaxGenFD provides a library of boundary functions that can be easily configured and applied to the edges of the domain. By default, an FDKernel does not have any boundary conditions enabled, so the behavior of the boundary depends on the initial state of the wavefields. In our Acoustic Forward Modeling example, we initialize our wavefields to zero, which means that the boundary behaves like a *Dirichlet* boundary i.e. a perfect reflector.

6.1 Boundary Masks

The sides of the domain that should be processed by a sponge or boundary conditions are set in the host code via bit masks. Each item in the first column of *Table 2* is the name of a `#define` in `maxlib.h`. These can be ORed together to select multiple sides.

`#define`	Value	Surface
MAXLIB_BOUNDARY_X0	0x1	x=0
MAXLIB_BOUNDARY_XN	0x2	x=nx
MAXLIB_BOUNDARY_Y0	0x4	y=0
MAXLIB_BOUNDARY_YN	0x8	y=ny
MAXLIB_BOUNDARY_Z0	0x10	z=0
MAXLIB_BOUNDARY_ZN	0x20	z=nz
MAXLIB_BOUNDARY_ALL	0x3f	All surfaces

Table 2: #defines representing domain boundaries for boundary condition and sponge bit-masks.

The masks are set in the host code via scalar inputs.

6.2 Simple Sponge

A sponge applies a simple damping coefficient to waves once they enter a damping region (typically 20-60 points) at the edge of the domain. This is an implementation of the sponge described by Charles Cerjan et al.

Figure 14 shows a comparison of the Acoustic Forward Modeling example with and without sponging as a wave approaches the edges of the domain.

The sponge is an `FDVar` stream created using a method on the `boundaries` property of the FDKernel:

FDVar *boundaries.sponge(**int** spongeSize)*

The size of the sponge is the number of points from the edge of the domain to which it will be applied, in all three dimensions.

The values that make up the sponge are coefficients that will be multiplied with a wavefield. The sponge is applied to the *entire* wavefield, with the value for the sponge being set to `1.0` when the current point in the domain is not in the boundary area.

In our example, we sponge the previous wavefield before the wave equation:

```
        FDVar sponge = boundaries.sponge(50); // Sponge

        prev = prev * sponge; // Sponge previous wavefield
```

And sponge the result afterward, before it is written out to DRAM:

```
        result = result * sponge; // Sponge result
```

The boundary mask is set in the host code via a scalar flag input with the name `‘‘sponge’’`. In our Acoustic Forward Modeling example, we set the bit mask to apply the sponge to all sides apart from the top of the domain as there is a real reflective surface there: the surface of the water.

```
    // Bit mask for the sides of the domain to sponge (all sides except z=0)
    const int spongeMask = MAXLIB_BOUNDARY_ALL ^ MAXLIB_BOUNDARY_Z0;
```

```
    maxlib_set_scalar_flag (maxlib, "absorb", spongeMask); // Set bit mask
```

✱ Current wavefield and previous wavefield must be sponged when using the simple sponge.

Smaller sponge regions than the size specified in the FDKernel can be configured at run-time by setting the sponge coefficients nearer to `spongeSize-1` to `1.0`.

The coefficients for the sponge are set in the host code via writes to a mapped ROM called `"sponge"`, as in our Acoustic Forward Modeling example:

```
    sponge = create_simple_sponge(spongeWidth, spongeActiveWidth); // Create sponge
     printf ("Loading sponge to DFE\n");
    for (int i = 0; i < spongeWidth; i++) // Set the sponge on the MaxCard
        maxlib_set_mapped_memory_f(maxlib, "sponge", i, sponge[i]);
```

This ROM must always be fully populated, and the maximum value for a sponge coefficient is `1.0`. The sponge coefficients are applied with index `0` as the point closest to the edge, and index `spongeSize-1` as the point nearest the interior.

The `‘‘sponge’’` mapped ROM is actually made up of three ROMs, one for each dimension, which are all written to with the same data when “sponge” is written to. For more specific control, you can set different sponge conditions at different boundaries using different coefficients in X, Y and Z by setting the three mapped ROMs called `‘‘boundaryspongeX’’`, `‘‘boundaryspongeY’’` and `‘‘boundaryspongeZ’’`. All three tables need to be set with the full set of `spongeSize` coefficients.

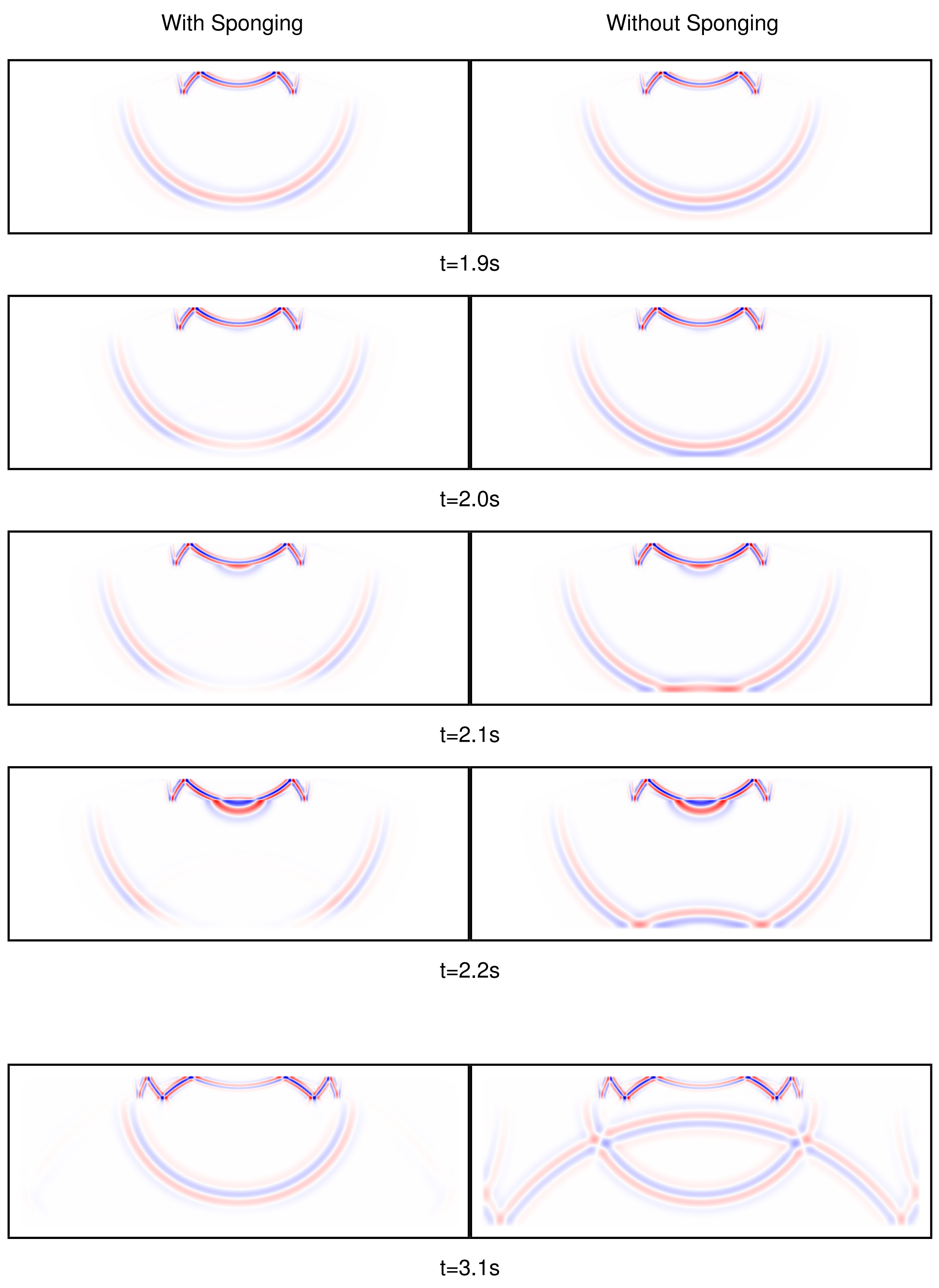


Figure 14: Output from MaxGenFD comparing the same simulation with (left) and without (right) the simple sponge being applied.

7 Optimizing Data Types

We can use different data sizes at each stage within the FDKernel to minimize memory space utilization, make the most efficient use of available memory bandwidth and minimize design area, while maximizing data precision. The data types are all set in the configuration object for the FDKernel.

To extract the best performance out of a DFE, Maxeler recommends the use of fixed-point data types in an FDKernel. Fixed-point calculations generally take far fewer resources and can be run at much higher clock rates. Equivalent or better precision than floating-point precision can be achieved using fixed-point data types.

In this section, we take our existing floating-point Acoustic Forward Modeling implementation and convert it to make use of fixed point.

7.1 Specifying Types in the FDKernel

We explicitly declare whether an FDKernel will use fixed-point or floating-point data when constructing the `FDConfig` object. In our Acoustic Forward Modeling example, we specify a floating-point FDKernel:

```
        // Use floating point
        FDConfig config = new FDConfig(engineParameters, DefaultType.FLOAT);
```

In MaxGenFD, we specify two different sets of types: storage types and compute types.

Figure 15 shows our Acoustic Forward Modeling example FDKernel with the storage and compute types annotated for the different data sources and streams. The arrows at the input and output of streams to the DFE show conversion and compression being implemented automatically.

7.1.1 Storage Types

Storage types specify how wavefield and earth model data should be stored in LMem on the DFE. Storage types can be compressed types.

The storage types available are limited to a subset of the computation types to ensure efficient use of the LMem space and bandwidth by packing into 16, 24 or 32 bit locations. All wavefields use the same storage type, whereas each earth model can use a different storage type depending on the precision to which the different earth model parameters are known:

```
void setWavefieldStorageType(StorageType type)
void setEarthModelStorageType(String name, StorageType type)
```

The available storage types are summarized in *Table 3*.

7.1.2 Compute Types

Compute types specify the types of the data streams in the FDKernel (and coefficient storage as this is internal to the DFE). The wavefield compute type is used for all wavefield inputs to the FDKernel (after decompression or conversion from the storage type) and host inputs. Different compute types can also be set for coefficients (stencils, sponge coefficients), scalar inputs and earth models.

Any operation within the FDKernel, except for the special case of the convolution operation, will produce a result of an intermediate type. In a floating-point FDKernel, this is the same as the wavefield compute type. In a fixed-point FDKernel, this is a type with the same width as the wavefield compute type but scaled to hold a larger maximum value.

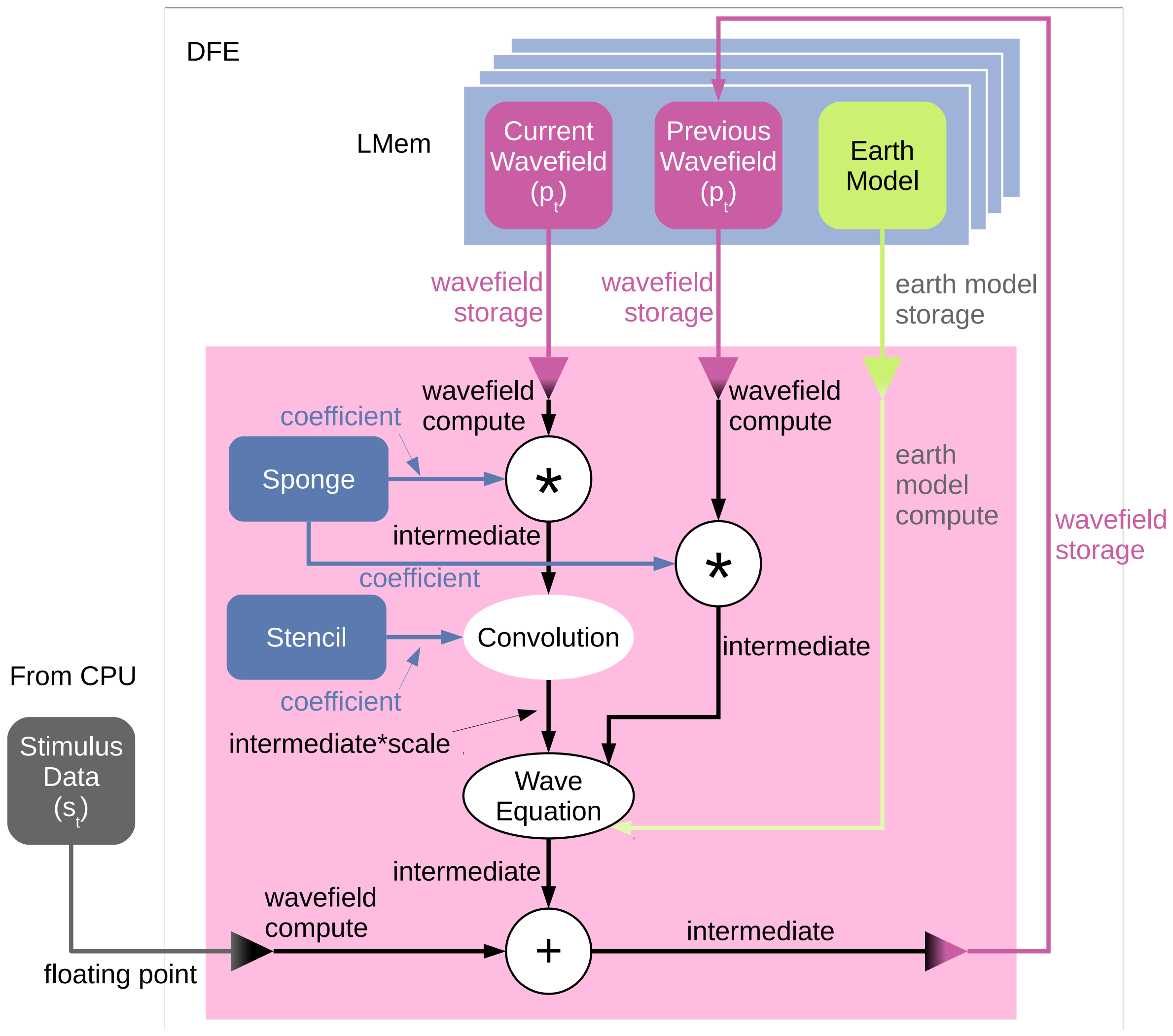


Figure 15: Acoustic Forward Modeling example FDKernel showing the data storage and compute types at different stages of the computation.

A convolution operation produces an output type that is its input type scaled by the scale factor specified in the stencil (see *subsubsection 7.8.3*).

In the case of compressed storage types, the type to which data should be expanded is specified by the wavefield compute type and earth model compute type.

The available methods each have two versions, one specific to floating point and one specific to fixed point:

***void** setWavefieldComputeType(ComputeType type)*

***void** setEarthModelComputeType(**String** name, **HWFloat** type)*
***void** setEarthModelComputeType(**String** name, **int** numBits)*

***void** setScalarInputType(**String** name, **HWFloat** type)*
***void** setScalarInputType(**String** name, **int** numBits)*

For table compression on earth models, the data type stored in the look-up table will be the earth model compute type.

Coefficients and earth models retain their specified compute type on input to calculations: they are

Storage Type	Description
`StorageType.float6_10`	16-bit floating-point type with 6-bit exponent and 10-bit mantissa
`StorageType.float7_17`	24-bit floating-point type with 7-bit exponent and 17-bit mantissa
`StorageType.float8_24`	32-bit floating-point type with 8-bit exponent and 24-bit mantissa
`StorageType.fixed16`	16-bit fixed-point type
`StorageType.fixed24`	24-bit fixed-point type
`StorageType.fixed32`	32-bit fixed-point type
`StorageType.compressed16`	16-bit compressed type (floating-point or fixed-point depending on default type). Compressed floating-point types are not supported on MAX4.
`StorageType.compressedTable (int numBits)`	`numBits`-bit table-compressed type quantized to $2^{numBits}$ values (floating-point or fixed-point depending on default type)

Table 3: Available storage types for earth models and wavefields.

not expanded to the wavefield compute type before computation takes place, which allows more optimal implementation of arithmetic operations.

The convolution operation is a special case for a fixed-point FDKernel as precision can be set differently for each stage of the operation (see *subsection 7.7*).

7.2 Compression

Wavefields and earth models have a value for every point in the domain, which can lead to very large memory requirements for storing the data on the accelerator cards. With a number of earth models and wavefields, this can lead to difficulty in packing all this data into the available storage.

To optimize use of memory space and bandwidth, MaxGenFD can add compression and decompression function blocks for earth models and wavefields at the beginning and end of the FDKernel computation pipeline. On-chip compression and decompression have no run-time cost, provided there is sufficient silicon area available, and earth models and wavefields can be automatically compressed and decompressed on the fly, which increases the amount of memory bandwidth available by the compression factor. MaxGenFD provides a number of in-built compression schemes that have been verified for use in seismic modeling and migration applications.

Compression of wavefields and earth models is enabled in the configuration object for the FDKernel.

7.2.1 Earth Model Compression

Earth model compression is enabled per earth model by setting the storage type:

***void** setEarthModelStorageType(**String** name, **StorageType** type)*

Earth model compression is covered in more detail in *section 9*.

7.2.2 Wavefield Compression

Wavefield compression is set for all wavefields via one method call to `setWavefieldStorageType`.

__void__ setWavefieldStorageType(__StorageType__ type)

Wavefields are automatically decompressed after reading from the LMem and results are automatically compressed before writing back to the LMem. Wavefields written from the host to the LMem of the card are compressed automatically by the DFE before being written into the LMem.

The recommended compression scheme for wavefields is **16-bit compression**. This scheme compresses fixed-point and floating-point types to an average 16-bits of storage per datum with minimum precision loss. The compression scheme is block-based, so loses less precision than a simple truncation of the original data.

The storage type to use is `StorageType.compressed16`. Wavefields will be expanded to the wavefield compute type in the DFE when they are read from the LMem.

7.3 Floating-Point FDKernels

MaxGenFD allows the use of the configurable floating-point types available in MaxCompiler. Creating a floating-point FDKernel is a useful way to quickly get an FDKernel written and functionally debugged before optimizing with fixed-point types.

For all floating-point operations, the output type of the operation is the same as the input type of the operation, so in *Figure 15*, the intermediate type in a floating-point implementation is the same as the wavefield compute type.

See the MaxCompiler Tutorial for more details on the allowable combinations of exponent and mantissa for floating-point types.

When we specify a floating-point FDKernel, the FDKernel defaults to a floating-point data type with a mantissa of 17 bits and an exponent of 7 bits (`HWTypeFactory.hwFloat(7, 17)`) for the wavefield compute and storage types.

In our Acoustic Forward Modeling example, we can change the code to support compressed storage types by setting the following lines in SimpleFDConfig.maxj.

```
config.setWavefieldStorageType(StorageType.compressed16);
config.setEarthModelStorageType("dvv", StorageType.compressedTable(10));
```

In our Acoustic Forward Modeling example, we declare an `HWFloat` type of `HWTypeFactory.hwFloat(8, 24)` and specify this type as the wavefield type, coefficient type and compute type for our earth model:

```
        // Set our floating point type for all calculations
        config.setWavefieldComputeType(ComputeType.float8_24);
        // IEEE single-precision floating point
        DFEFloat t = DFETypeFactory.dfeFloat(8, 24);
        config.setCoefficientType(t);
        config.setEarthModelComputeType("dvv", t);
```

MaxGenFD only supports arithmetic operations on two floating-point streams where the exponent and mantissa of the two streams are the same. It is therefore recommended to use the same floating-point type throughout an FDKernel.

7.4 Fixed-Point FDKernels

In our Acoustic Forward Modeling Example 1, we used single-precision floating point for all of our data (except for the compression on storage of wavefields and earth models). To get the best performance out of an FDKernel, however, we need to convert our design to use fixed-point data, which is done in Example 3. The configuration object for this is shown in *Listing 9*.

Conversion from floating point to fixed point for data transferred to the card, and transformation back again on transfer of data back from the card, is performed automatically. The earth model is converted to fixed point on the host before being transferred to the accelerator card. For wavefields and host inputs, all conversion to and from fixed point is performed on the accelerator card.

In MaxGenFD, fixed-point data types are specified by the total number of bits for the data word. MaxGenFD infers the position of the binary point in two ways:

- Statically within the FDKernel, from the absolute magnitude, relative magnitudes and scale factors specified in the FDKernel (see *subsection 7.8*). This allows the maximum precision to be kept for the number of bits in the word for each operation within the FDKernel.
- Dynamically between executions of the FDKernel, from the magnitude of the data in host inputs (e.g. source stimulus) and wavefields for each timestep. The scale factor of the wavefields stored in LMem is also tracked over time. This allows the maximum utilization over time of the number of bits available by moving the binary point as the magnitude of the host inputs and wavefields changes.

Note that if you wish to debug the FDKernel using MaxCompiler watches, the data at any point in the FDKernel may be scaled, so care must be taken in inferring any absolute value for the data.

Listing 9: Fixed point FDConfig object for our Acoustic Forward Modeling example (FixedPointFDKernel.maxj).

```
/**
 * Document: MaxGenFD Tutorial (maxgenfd-tutorial.pdf)
 * Example: 3      Name: Fixedpoint FD
 * MaxFile name: FixedPointFD
 * Summary:
 *      Configuration settings for the model running with 8 pipes
 *      at a frequency of 100Mhz.
 */

package fixedpointfd;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxgen.fd.ComputeType;
import com.maxeler.maxgen.fd.FDConfig;
import com.maxeler.maxgen.fd.FDConfig.DefaultType;
import com.maxeler.maxgen.fd.StorageType;

class FixedPointFDConfig {
    static FDConfig config(EngineParameters engineParameters) {
        // Use fixed point
        FDConfig config = new FDConfig(engineParameters, DefaultType.FIXED);

        config.setMaxBlockSize(192, 144);

        config.setClockFrequency(100); // This can go up to 150Mhz but requires
                                       // a long build time

        config.setParallelPipelines(8);

        // Enable compression for wavefields and earth model
        config.setWavefieldStorageType(StorageType.compressed16);
        config.setEarthModelStorageType("dvv", StorageType.compressedTable(10));

        // Set our fixed point type for all calculations
        config.setWavefieldComputeType(ComputeType.fix24);
        config.setCoefficientType(18);
        config.setEarthModelComputeType("dvv", 18);
        config.setConvolveTypes(25, 48);

        return config;
    }
}
```

Fixed-point and floating-point types cannot be mixed in a single FDKernel, unless explicitly implemented using MaxCompiler features.

In our Acoustic Forward Modeling example, we now set the wavefield compute type to 24-bit fixed point, the coefficient type to 18-bit fixed-point and the earth model compute type to 18-bit fixed-point:

```
        // Set our fixed point type for all calculations
        config.setWavefieldComputeType(ComputeType.fix24);
        config.setCoefficientType(18);
        config.setEarthModelComputeType("dvv", 18);
        config.setConvolveTypes(25, 48);
```

7.5 Intermediate Type

Intermediate results from computations in the FDKernel will automatically be given an intermediate type by MaxGenFD that is estimated to prevent overflow and minimize underflow. Setting the appropriate maximum values for inputs and scale factors for stencils will help optimize these intermediate types.

If a significant number of operations are carried out in a chain, some precision loss may occur because types propagate conservatively and thus numbers may grow more bits on the most significant bit than are required to hold results that practically occur. This can be addressed one of two methods:

- Manually applying casts at specified points to change the type to something more efficient at representing the actual values
- Switching some parts of the computation into an alternative type propagation mode using **guard bits**.

7.5.1 Guard Bits Mode

This mode can be enabled and disabled around a region of computation in an FDKernel by calling `pushGuardedOpModes()` and `popGuardedOpModes()`:

```
void pushGuardedOpModes(int guard_bits);
void popGuardedOpModes();
```

The number of guard bits `guard_bits` represents the number of bits by which the intermediate type is able to represent numbers larger than the wavefield compute type, for example 3 guard bits means that the computation will be able to represent values 8 times larger than the wavefield compute type. By fixing the number of guard bits around a region of the computation where large growth in values is not expected, the precision of the computation can be increased.

7.6 Coefficient Types

The important factor in determining the size for coefficient types is the precision to which the coefficients are known. As the data is scaled automatically according to the magnitude of the coefficients, only the number of bits needs to be specified.

```
void setCoefficientType(HWFloat type)
void setCoefficientType(int  numBits)
```

Where a coefficient is used in a calculation, such as a convolution, sponge or wave equation, the input type of the coefficient to the calculation is the same as the coefficient type: the data is not transformed to the wavefield compute type before computation.

7.7 Convolution Types

The convolution operation has an extra option available for specifying the maximum fixed point width at two different stages of computation:

- Stage 1 of the convolution is the initial adds/subtracts of symmetrical/antisymmetrical terms and the input to the coefficient multiplier.
- Stage 2 is the output from the multiplier and the accumulation of the convolution output.

The default values are 25 for the first stage and 48 for the second stage. These sizes map directly onto the sizes supported by the dedicated DSP blocks in the DFE.

The values are set using the method `setConvolveTypes`:

```
void setConvolveTypes(int limit_stage1bits, int limit_stage2bits )
```

7.8 Fixed-Point Scaling

The use of fixed-point types can be further optimized by supplying information about the data itself. This meta-data is supplied in the FDKernel.

There are three types of meta-data that can be provided: absolute maximum values for inputs, relative magnitudes of inputs and scale factors for stencils. These three pieces of meta-data are used to move the position of the binary point in the fixed-point word at different stages of the computation in the FDKernel.

Note that setting any of these scaling options incorrectly may lead to data overflowing the fixed-point word, leading to incorrect results.

7.8.1 Absolute Maximum Values

Scalar inputs and earth model inputs can have absolute maximum values specified. This is the maximum value that will written into these streams. The fixed point word will be scaled to fit the maximum value with as much precision remaining in the lower bits as possible.

```
HWVar io.scalarInput(String name, double absMax, double relMax)
FDVar io.earthModelInput(String name, double max, int halo)
FDVar io.earthModelInput(String name, double max, int halo, FDExpr value)
```

In our Acoustic Forward Modeling example, we specify an absolute maximum of $1/4$ for the velocity earth model derivative:

```
36        FDVar dvv = io.earthModelInput("dvv", 1 / 4.0, 0); // Earth model
```

7.8.2 Relative Magnitudes

For host inputs (for example our source wavelet) and wavefield inputs, the size relative to each other can be set using the `relMax` argument.

If one input is set to have a `relMax` of `1.0` and another is to have a `relMax` of `2.0`, this means that the second input contains values up to a maximum of the twice the size of the first input.

```
HWVar io.scalarInput(String name, double absMax, double relMax)
FDVar io.hostInput(String name, double relMax, int halo)
FDVar io.waveFieldInput(String name, double relMax, int halo)
```

For example, in the following code, wavefield input b contains values up to twice the size of host input a and wavefield input c contains values up the half the size of host input a:

```
FDVar a = io.hostInput("a", 1.0, 0);
FDVar b = io.waveFieldInput("b", 2.0, 6);
FDVar c = io.waveFieldInput("c", 0.5, 6);
```

Relative magnitudes are commonly used where a timestep is split over multiple executions of an FDKernel.

Scalar inputs have an absolute maximum that is used for multiplication and division operations and a relative maximum that is used for addition and subtraction operations.

7.8.3 Convolution Output Scaling

Stencils have an argument `outputScale` which specifies the size the output of a convolution with this stencil relative to the input. For example, an `outputScale` of `3.0` means that the output is up to three times the size of the input of a convolution performed with this stencil.

```
Stencil fixedStencil(int min, int max, double[] coeffs, double outputScale)
Stencil cubeStencil(int minX, int maxX, int minY, int maxY, int minZ, int maxZ, double[][][] coeffs, double outputScale)
Stencil variableStencil(int min, int max, double outputScale, FDVar... coefficients)
```

If you do not have enough data to choose a scale factor, or are concerned about overflow errors, you can specify a scale of `0.0` to have MaxGenFD automatically choose a conservative value that may not provide maximum precision but will guarantee prevention of overflow.

In our Acoustic Forward Modeling example, we specify a scale factor of `8.0` for the stencil:

```
    private final int stencilSize = 13;
    private final Stencil stencil = fixedStencil(
        -stencilSize / 2,
        stencilSize / 2,
        new double[] { -0.000030062530062499946f, 0.0005194805194800871f,
                -0.004464285714282842f, 0.02645502645501459f,
                -0.133928571428538f, 0.857142857142776f, -1.491388888888878f,
                0.857142857142776f, -0.133928571428538f, 0.02645502645501459f,
                -0.004464285714282842f, 0.0005194805194800871f,
                -0.000030062530062499946f },
        8.0);
```

The worst-case scale factor (which cannot be exceeded regardless of the input values) can be calculated as the sum of the absolute values of the coefficients in the stencil. A scale factor for more aggressive optimization can be calculated by recording the maximum input array and corresponding output value of the convolution for a number of timesteps of the simulation with some real data.

Using a more precise scale factor can give greater precision through the FDKernel, but care must be taken to verify that this scale factor is correct for the input used in a simulation.

7.8.4 Additional Scale Factor

It is also possible to explicitly set an additional scale factor for the FDKernel in the host code:

```
void maxlib_set_scale_factor (maxlib_context context, float scale_factor)
```

This additional scale factor is applied *in addition* to the automatic dynamic scaling between timesteps and the scaling within the FDKernel. This can be useful for debugging an application if overflow or underflow is suspected as the cause for errors.

> ✷ It is not recommended to set the additional scale factor in a production application as it makes inefficient use of the fixed-point data word in the FDKernel.

8 Performance Optimization

The performance of an FDKernel can be further improved by adjusting various parameters and enabling features via the configuration object.

8.1 Multiple Processing Pipelines

The definition of an FDKernel does not specify the number of parallel pipelines, or *pipes*, in a design: this is specified in the configuration object. A key method for increasing the performance of a MaxGenFD implementation is to increase the number of pipes.

The number of pipes is set using the method `setParallelPipelines`:

```
void setParallelPipelines (int numPipes)
```

Each pipe generates another copy of the arithmetic logic and some of the buffering logic, so this has an impact on the size of the design. There is therefore a maximum number of a given FDKernel pipe that can be implemented in a DFE. The reports generated by MaxGenFD can be useful for estimating how many pipes may fit in the device, but a full build is required to verify that the design will fit and meet timing constraints.

In Example 3, we can run 8 parallel pipelines with our fixed-point implementation of the Acoustic Forward Modeling code:

```
        config.setParallelPipelines(8);
```

Note that an implementation with many pipes may take a long time to build, so using a single (or small number) of pipes is recommended for debugging purposes.

8.2 Clock Frequency

The clock frequency for the FDKernel can be set in the configuration object:

```
void setClockFrequency(int mhz)
```

Whether the design will meet the constraints for this clock frequency depends on a number of factors, including the data types in use and the size of the design. Determining the maximum clock frequency of a design is an iterative process involving setting a target clock speed, building the design to see if the constraints were met and then either increasing or lowering the clock speed accordingly.

In Example 3, we set the clock speed to 100 MHz:

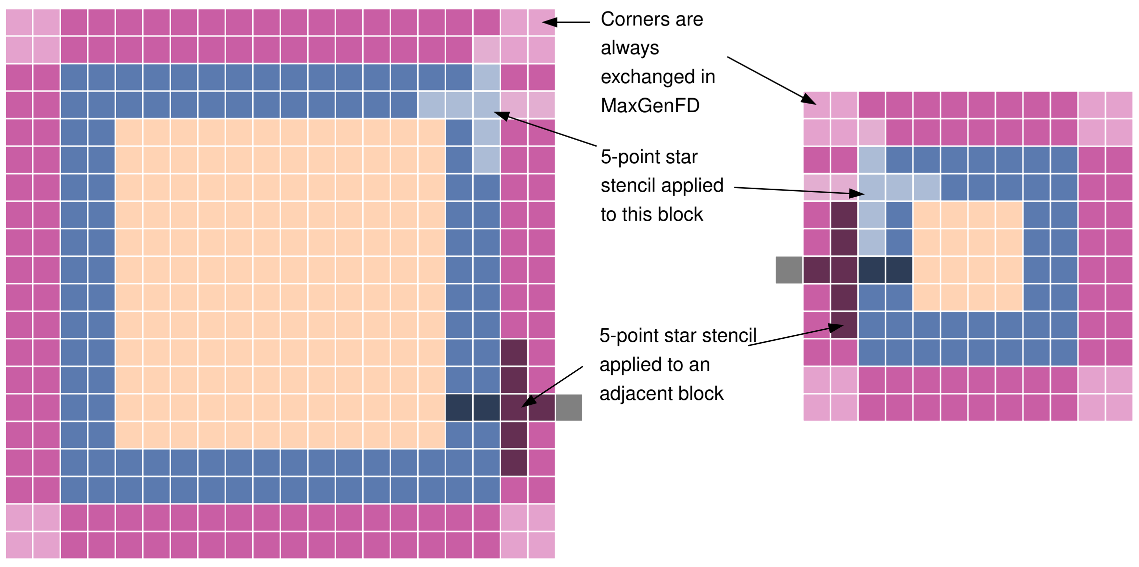
```
        config.setClockFrequency(100); // This can go up to 150Mhz but requires
                                       // a long build time
```

Note that setting a high clock speed may result in a long build time, so using a lower clock speed is recommended for debugging purposes.

8.3 Block Size

An FDKernel operates on blocks of data at a time from the sub-domain that is allocated to it. The size of block on which the FDKernel operates affects the number of memory components within the FPGA that the FDKernel uses.

Better performance is gained from the FDKernel if it can use more memory components. If the block size is small, then a relatively large amount of processing time for the block is spent in the halo area, which slows down the process. The efficiency of the MaxGenFD implementation is directly proportional to the number of points not in the halo area. *Figure 16* shows shows the efficiency with two different block sizes with a 5-point stencil.


Corners are always exchanged in MaxGenFD
5-point star stencil applied to this block
5-point star stencil applied to an adjacent block

(a) 20x20 Block Size: 64% efficiency (b) 12x12 Block Size: 44.5% efficiency

Figure 16: Efficiency for different block sizes.

Note that the corner points are always exchanged in MaxGenFD, even for star stencils, for efficiency reasons.

The general equation for the efficiency for a block, for an n-point stencil, a stencil overlap of s and a block size of X by Y, is:

$$\frac{(X-2s)\times(Y-2s)\times 100}{X\times Y}$$

The stencil overlap for a symmetric n-point stencil is $floor(n/2)$, or the largest integer less than or equal to $n/2$.

MaxGenFD uses a default maximum size of block (96x96) which is conservative to ensure that there is sufficient block RAM available, but this size can be tweaked.

The method used to do this is `setMaxBlockSize` which takes two arguments, `blockWidth` which applies to the width in the *fast* dimension and `blockHeight` which applies to the height in the *medium* dimension:

```
void setMaxBlockSize(int blockWidth, int blockHeight)
```

These numbers are suggestions to MaxGenFD rather than absolute values, as the underlying block size is constrained by rules that change depending on data types, device and even board model. This means that the process of determining the optimum values for maximizing performance of an FDKernel on a given device is an iterative one that involves setting some values and seeing the block RAM usage reported after the build process is complete.

In Example 3, we set a more efficient block size for our Acoustic Forward Modeling code:

```
        config.setMaxBlockSize(192, 144);
```

With the stencil size of 13 and the block size of 192 by 144, we have an efficiency of 86% instead of 77% with the default block size of 96 by 96.

8.4 Interleaving wavefields

Where wavefields are always read from and written to at the same time, which commonly occurs when there are two wavefields representing the state for a timestep in an anisotropic mode, more efficient use of device resources can be made by interleaving the wavefields. This reduces two fully independent streams stored in two separate blocks in the LMem into two streams that share much of their control logic with the data stored interleaved in one block of LMem twice the size.

Interleaved wavefields are grouped together in the configuration object with a call to `setStoreWavefieldsInterleaved` with a string argument for the group name and then a list of string names for the wavefields to interleave:

```
void setStoreWavefieldsInterleaved(String groupName, String... names)
```

For example, if we were to create two wavefields called `"wave_a"` and `"wave_b"`, we would interleave them in a group called `"mywaves"` like so:

```
FDConfig config = new FDConfig(DefaultType.FIXED);
config.setStoreWavefieldsInterleaved("mywaves", "wave_a", "wave_b");
```

In the host code, storage on the accelerator card is allocated for all of the interleaved wavefields in a group via `maxlib_lmem_alloc_packed_wavefield`, which takes an additional argument of the number of wavefields that will be packed into that group:

```
maxlib_lmem_array maxlib_lmem_alloc_packed_wavefield(maxlib_context context, int n_packed)
```

Wavefields are transferred to the card using the same function as for normal wavefields, `maxlib_lmem_load_array`, except that a list of pointers to wavefield data is passed in. For example, to load our two interleaved wavefields `"wave_a"` and `"wave_b"`, we would write:

```
float *wave_a;
float *wave_b;

maxlib_lmem_array mywaves_array = maxlib_lmem_alloc_packed_wavefield(maxlib, 2);
maxlib_lmem_load_wavefield(maxlib, mywaves_array, wave_a, wave_b);
```

Finally, to stream the interleaved wavefields in a timestep, we reference the group name in the host rather than the individual streams, for example:

```
maxlib_stream_from_lmem(maxlib, "mywaves", mywaves_array);
```

✷ Interleaving wavefields uses the same amount of space in the LMem.

8.5 Multi-Chip Implementations

MaxGenFD will automatically decompose a domain into sub-domains and distribute these sub-domains to a number of accelerator cards in a system. MaxGenFD will automatically split the domain over all the devices that it finds and transfer halo data over MaxRing (a direct high-speed cable connection between Maxeler accelerator cards) or PCI Express between execution steps (see *subsection 3.4* for more detail on halos). MaxGenFD will favor MaxRing transfers over PCI Express as this method offers higher performance. In order for MaxGenFD to perform this distribution, the FDKernel must have MaxRing, PCI Express Swap or both enabled.

MaxRing halo swap is enabled for all halos through a single method call in the configuration object:

void *setMaxRingEnabled(**boolean** enabled)*

MaxRing swap is disabled by default.

PCI Express Swap is enabled on a per output basis. Every output that feeds an input in the next execution step must have PCI Express Swap enabled:

void *setPCIeSwapOutput(**String** outputName)*

MaxGenFD will automatically transfer the halo data when all of the correct outputs have PCI Express Swap enabled.

PCI Express Swap is disabled by default on all output streams.

MaxGenFD will automatically try to use up to 8 devices, but the upper limit can be set lower than this in the host code at run-time, using a call to `maxlib_set_global_parameter` to set `"MAX_DEVICES"`, for example:

```
maxlib_set_global_parameter("MAX_DEVICES", 4);
```

✷ `"MAX_DEVICES"` must be set before `maxlib_open` is called.

The first device ID to use can be set via another parameter, `"FIRST_DEVICE"`, for example to ignore the first two devices (which we may want to reserve for another application, for example) we would add:

```
maxlib_set_global_parameter("FIRST_DEVICE", 2);
```

9 Advanced Earth Model Usage

In real-world applications, earth models may be very large. This leads to two problems:

- Fitting the required earth model region into the LMem on the DFE.
- Loading and manipulating an earth model that may be much larger than the available memory on the host.

MaxGenFD provides earth model compression to solve the first issue and an earth model API for the CPU code to solve the second.

9.1 Enabling Earth Model Compression

Earth model compression is enabled per earth model by setting the storage type:

void *setEarthModelStorageType(**String** name, **StorageType** type)*

The recommended compression scheme for earth models is **table compression**, in which the earth model is quantized to a number of values. A lookup table is created and each point in the earth model is assigned the value of the table index of the entry that is closest to its real value.

You can create a table-compressed storage type by specifying the size of the lookup table:

StorageType StorageType*.compressedTable(**int** numBits)*

The number of entries in the table is $2^{numBits}$, so 10 bits will produce a table with 1024 entries. This means that the word size for each compressed value will be 10 bits, rather than the size of the original data type. For example, the compression ratio for a 32-bit fixed point type to a 1024-element (10-bit) table-compressed storage type is around 3:1.

The actual type of the data in the table is set to any floating-point or fixed-point type by setting the earth model compute type:

void *setEarthModelComputeType(**String** name, **HWFloat** type)*
void *setEarthModelComputeType(**String** name, **int** numBits)*

This is the type of the data in the lookup table and is the type of the stream that we get back via `io.earthModelInput`.

The earth models are initially compressed on the host before writing to the LMem on the DFE; the models are automatically decompressed when they are streamed out of the LMem.

You should choose the table size based on the precision of the parameter that you want to represent, for example velocity needs to be represented quite precisely (8-10 bits) whereas some other quantities, for example anisotropy parameters, are known much less accurately and may only require 2-6 bits.

9.2 Handling Earth Models in the Host Code

The MaxGenFD earth model API allows you to:

- Work with a subregion of a large earth model
- Reduce execution time by saving a compressed earth model for reuse across multiple shots.

9.2.1 Loading an Earth Model

There are two methods of working with earth models:

- In memory, if you have enough on the local machine.
- On disk, if you have a large earth model that is too big for the memory in the local machine.

Earth models are managed in the host code using a `maxlib_earthmodel` instance.
A `maxlib_earthmodel` instance can be either in memory or file-backed, depending on how it is created or loaded.

An earth model is created in memory by calling the following function with the required dimensions:

```
maxlib_earthmodel maxlib_earthmodel_create_in_memory(int nx, int ny, int nz);
```

A new file-backed earth model can be created using the following function, where `filename` is the name of the file to be created:

```
maxlib_earthmodel maxlib_earthmodel_create_file_backed(int nx, int ny,  int  nz,  const char *filename);
```

9.2.2 Loading an existing Earth Model

An existing MaxGenFD earth model that has been saved to file can be loaded using:

```
maxlib_earthmodel maxlib_earthmodel_file_load(const char *filename);
```

Loading an existing earth model from file creates a file-backed instance of `maxlib_earthmodel` which uses this file.

9.2.3 Closing an Earth Model

When an earth model is no longer needed, it should be closed using `maxlib_earthmodel_release`:

```
void maxlib_earthmodel_release(maxlib_earthmodel em);
```

`maxlib_earthmodel_release` checks the `maxlib_earthmodel` instance `em` to see if any other `maxlib_earthmodel` instances reference the same memory (if the earth model is stored in memory) or file (if the earth model is file-backed).

For an earth model in memory, if `em` is the only instance that references that memory, the memory is released and `em` is deleted. Otherwise `em` is deleted without releasing the memory.

For a file-backed earth model, if `em` is the only instance that references that file, all data is flushed to the file, the file is closed and `em` is deleted. Otherwise `em` is deleted without closing the file.

9.2.4 Saving an Earth Model

If the `maxlib_earthmodel` instance that you wish to save is file-backed, then calling `maxlib_earthmodel_release` ensures that all data is flushed to the file and closes it.

Alternatively, to save an earth model stored in memory or to create a copy of a file-backed earth model, you can call `maxlib_earthmodel_file_save`:

```
void maxlib_earthmodel_file_save(maxlib_earthmodel em, const char *filename);
```

✷ If the earth model is incomplete, then MaxGenFD will report an error when you try to save it.

9.2.5 Using Part of an Earth Model

To load the subregion of an earth model, there are functions for extracting a subregion from an `maxlib_earthmodel` instance.

Regions are defined using the `maxlib_region` C `struct`, which defines the start (*inclusive*) and end (*exclusive*) coordinates for a 3D region:

```
typedef struct {
    int x_start, x_end;
    int y_start, y_end;
    int z_start, z_end;
} maxlib_region;
```

A `maxlib_region` can be created using a function:

```
maxlib_region maxlib_region_create(int x_start, int y_start, int z_start, int x_end, int y_end, int z_end);
```

`maxlib_earthmodel_get_subregion` returns a new `maxlib_earthmodel` instance that is a reference to the existing region in the `em` argument:

```
maxlib_earthmodel maxlib_earthmodel_get_subregion(maxlib_earthmodel em, maxlib_region sub_region);
```

For example, in the following code `sub_em` is a file-backed instance of `maxlib_earthmodel` that is backed by the same file as `big_em` (`"myEarthModelFile"`) and refers to the subregion `sub_region` within that file:

```
maxlib_earthmodel big_em = maxlib_earthmodel_file_load("myEarthModelFile");
maxlib_region sub_region = maxlib_region_create(0, 32, 0, 32, 0, 32);
maxlib_earthmodel sub_em = maxlib_earthmodel_get_subregion(big_em, sub_region);
```

An alternative function returns a new `maxlib_earthmodel` instance that has a copy in memory of the region in the `em` argument:

```
maxlib_earthmodel maxlib_earthmodel_copy_subregion_into_memory(maxlib_earthmodel em, maxlib_region sub_region);
```

For example, in the following code `sub_em` is an instance of `maxlib_earthmodel` that has a copy in memory of the subregion `sub_region` from the file `"myEarthModelFile"`:

```
maxlib_earthmodel big_em = maxlib_earthmodel_file_load("myEarthModelFile");
maxlib_region sub_region = maxlib_region_create(0, 32, 0, 32, 0, 32);
maxlib_earthmodel sub_em = maxlib_earthmodel_copy_subregion_into_memory(big_em, sub_region);
```

9.2.6 Tile Size

It is not possible to work with arbitrary subregions of an earth model: you can only select subregions that align to **tile** boundaries. A tile is a small region of data within the domain that is treated as a single unit by MaxGenFD. You can only access subregions that are aligned with tiles i.e. `nx%tile_size = 0`.

You can determine the tile size in each dimension by calling `maxlib_earthmodel_get_tile_size`:

```
void maxlib_earthmodel_get_tile_size(maxlib_earthmodel em, int* tile_size_x , int* tile_size_y , int* tile_size_z );
```

9.3 Setting Data in an Earth Model

To either set uncompressed earth model data or use the default compression scheme, a single function call for each parameter can be used. To use an alternative or custom compression scheme for table-compressed earth model parameters, the table contents must be calculated and then the table values and indices must be set.

9.3.1 Using the Default Compression Mode

Data is set for each parameter in an earth model using single-precision floating-point data: this data is automatically converted to the storage type specified in the `FDConfig` object for that earth model parameter. If the earth model storage type is specified as table compressed, then the data will be compressed using the default `maxlib_table_compress_uniform` compression scheme (see *subsection 9.4*).

The data can be set either from memory or from file:

```
void maxlib_earthmodel_set_data(maxlib_earthmodel em, const char *param_name, const float *data);
void maxlib_earthmodel_set_data_from_file(maxlib_earthmodel em, const char *param_name, const char *filename);
```

The `param_name` argument is the name of the parameter in the earth model to set.

9.4 Specifying a Table Compression Scheme

Three compression schemes are implemented in MaxGenFD:

- Uniform (default)
- Fast Uniform (faster than default uniform, but less accurate)
- Adaptive (slowest but most accurate)

These three schemes are implemented by function calls to perform the compression:

```
void maxlib_table_compress_uniform(const float *data, size_t data_len, float *table_values, size_t table_len, uint16_t *table_indices
        );
void maxlib_table_compress_uniform_fast(const float *data, size_t data_len, float *table_values, size_t table_len, uint16_t *
        table_indices);
void maxlib_table_compress_adaptive(const float *data, size_t data_len, float *table_values, size_t table_len, uint16_t *
        table_indices);
```

All of these functions have the same arguments:

- `data` is an array of floating-point data representing the earth model parameter. This data must be at least `data_len` elements long.
- `data_len` is the number of elements in the `data` array. This should be the number of points in the earth model.
- `table_values` is an array that will contain the lookup table values. `table_values must` be at least `table_len` elements long.
- `table_len` is the number of elements in the lookup table.
- `table_indices` is an array of integers that will contain the table indices representing the compressed earth model parameter. `table_indices` must be at least `data_len` elements long.

The `maxlib_earthmodel_get_table_size` function is provided, which returns the number of elements in the lookup table for a given earth model parameter:

```
size_t maxlib_earthmodel_get_table_size(maxlib_earthmodel em, const char *param_name);
```

9.4.1 Loading Compressed Data into an Earth Model

The contents of the lookup table and the indices to store in DRAM to represent the compressed earth model parameter are written separately into the earth model.

`maxlib_earthmodel_set_table_values` sets the values in the lookup table for the named parameter:

```
void maxlib_earthmodel_set_table_values(maxlib_earthmodel em, const char *param_name, const float *table_values);
```

The floating point values in `table_values` are automatically converted into the earth model compute type specified in the `FDConfig` object.

`maxlib_earthmodel_set_table_indices` sets the indices into the table that represent the compressed earth model parameter in DRAM:

```
void maxlib_earthmodel_set_table_indices(maxlib_earthmodel em, const char *param_name, const uint16_t *table_indices);
```

9.4.2 Implementing Your Own Table Compression Scheme

You can implement your own table compression scheme that suits your application's requirements.

Your table compression scheme must produce an array of table values to fill the lookup table and an array of indices that will replace the values for the earth model parameter in LMem.

The table values array must be of single-precision floating-point data and have the correct number of elements for the lookup table for the target earth model parameter. The size of the lookup table for each earth model parameter can be retrieved using `maxlib_earthmodel_get_table_size`:

```
size_t maxlib_earthmodel_get_table_size(maxlib_earthmodel em, const char *param_name);
```

The array of indices must be of `uint16_t` data and have the same number of elements as the target `maxlib_earthmodel` instance. The value of each element in the array must be greater than zero and less than the size of the lookup table.

Implement your own function and then call set `maxlib_earthmodel_set_table_values` and `maxlib_earthmodel_set_table_indices` to set the data in the earth model.