

Patus Documentation

Matthias-M. Christen
m.christen@unibas.ch
University of Basel, Switzerland

Contents

Contents	i
1 A PATUS Walkthrough Example	3
1.1 From a Model to a Stencil	3
1.2 Generating The Code	5
1.3 Running and Tuning	6
2 PATUS Usage	11
2.1 Code Generation	11
2.2 Auto-Tuning	13
3 Integrating into User Code	17
4 Alternate Entry Points to PATUS	19
5 More Details on PATUS Stencil Specifications	21
6 Strategies and Hardware Architectures	25
6.1 A Cache Blocking Strategy	25
6.2 Independence of the Stencil	27
6.3 Independence of the Hardware Architecture	29
6.4 Examples of Generated Code	30
7 Current Limitations	33
Bibliography	35
A PATUS Grammars	39
A.1 Stencil DSL Grammar	39
A.2 Strategy DSL Grammar	40
B Stencil Specifications	43

B.1	Basic Differential Operators	43
B.1.1	Laplacian	43
B.1.2	Divergence	43
B.1.3	Gradient	44
B.2	Wave Equation	44
B.3	COSMO	45
B.3.1	Upstream	45
B.3.2	Tricubic Interpolation	45
B.4	Hyperthermia	46
B.5	Anelastic Wave Propagation	46
B.5.1	uxx1	46
B.5.2	xy1	47
B.5.3	xyz1	48
B.5.4	xyzq	50

Copyrights

Cetus

Copyright under the Cetus Artistic License. <http://cetus.ecn.purdue.edu/>

JGAP

Copyright under the GNU Lesser Public License. <http://jgap.sourceforge.net/>

Time measurement

Copyright (c) 2003, 2007-8 Matteo Frigo,

Copyright (c) 2003, 2007-8 Massachusetts Institute of Technology

Chapter 1

A Patus Walkthrough Example

In this section, we give an example, who a PATUS stencil specification can be derived from a mathematical problem description. From this stencil specification, the PATUS code generator generates a C code which implements the compute kernel along with a benchmarking harness that can be used to measure the performance of the generated code and also shows how the kernel is to be called from within a user code.

1.1 From a Model to a Stencil

Consider the classical wave equation on $\Omega = [-1, 1]^3$ with Dirichlet boundary conditions and some initial condition:

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2} - c^2 \Delta u &= 0 && \text{in } \Omega, \\ u &= g && \text{on } \partial\Omega, \\ u|_{t=0} &= f.\end{aligned}\tag{1.1}$$

We use an explicit finite difference method to discretize the equation both in space and time. For the discretization in time we use a second-order scheme with time step δt . For the discretization in space we choose a fourth-order discretization of the Laplacian Δ on the structured uniformly discretized grid Ω_h with step size h . This discretization gives us

$$\frac{u^{(t+\delta t)} - 2u^{(t)} + u^{(t-\delta t)}}{\delta t} - c^2 \Delta_h u^{(t)} = 0,\tag{1.2}$$

where Δ_h is the discretized version of the Laplacian:

$$\begin{aligned} \Delta_h u^{(t)}(x, y, z) = & \frac{-15}{2h^2} u^{(t)}(x, y, z) + \\ & \frac{-1}{12h^2} \left(u^{(t)}(x - 2h, y, z) + u^{(t)}(x, y - 2h, z) + u^{(t)}(x, y, z - 2h) \right) + \\ & \frac{4}{3h^2} \left(u^{(t)}(x - h, y, z) + u^{(t)}(x, y - h, z) + u^{(t)}(x, y, z - h) \right) + \\ & \frac{4}{3h^2} \left(u^{(t)}(x + h, y, z) + u^{(t)}(x, y + h, z) + u^{(t)}(x, y, z + h) \right) + \\ & \frac{-1}{12h^2} \left(u^{(t)}(x + 2h, y, z) + u^{(t)}(x, y + 2h, z) + u^{(t)}(x, y, z + 2h) \right). \end{aligned} \quad (1.3)$$

Substituting Eqn. 1.3 into Eqn. 1.2, solving Eqn. 1.2 for $u^{(t+\delta t)}$, and interpreting u as a grid in space and time with mesh size h and time step δt , we arrive at

$$\begin{aligned} u[x, y, z; t + 1] = & 2u[x, y, z; t] - u[x, y, z; t - 1] + c^2 \frac{\delta t}{h^2} \left(\frac{-15}{2} u[x, y, z; t] + \right. \\ & \frac{-1}{12} (u[x - 2, y, z; t] + u[x, y - 2, z; t] + u[x, y, z - 2; t]) + \\ & \frac{4}{3} (u[x - 1, y, z; t] + u[x, y - 1, z; t] + u[x, y, z - 1; t]) + \\ & \frac{4}{3} (u[x + 1, y, z; t] + u[x, y + 1, z; t] + u[x, y, z + 1; t]) + \\ & \left. \frac{-1}{12} (u[x + 2, y, z; t] + u[x, y + 2, z; t] + u[x, y, z + 2; t]) \right). \end{aligned}$$

To actually solve the discretized equation, we need to specify the mesh size h or, equivalently, the number of grid points N , and the time step δt or, equivalently, the number of time steps t_{\max} . Choosing a concrete number of the number of time steps, we can transform the above equation together with the number of grid points and the number of time steps almost trivially into a PATUS stencil specification:

Example 1.1: A PATUS stencil specification.

The listing below shows the PATUS stencil specification for the classical wave equation discretized by 4th order finite differences in space and by 2nd order finite differences in time. Note that the maximum N in the domain size specification is inclusive.

```
stencil wave
{
    domainsize = (1 .. N, 1 .. N, 1 .. N);
    t_max = 100;

    operation (float grid u, float param c2dt_h2)
    {
        u[x, y, z; t+1] = 2 * u[x, y, z; t] - u[x, y, z; t-1] +
            c2dt_h2 * (
                -15/2 * u[x, y, z; t] +
                4/3 * (
```


Example 1.1: A PATUS stencil specification. (cont.)

```

        u[x+1, y, z; t] + u[x-1, y, z; t] +
        u[x, y+1, z; t] + u[x, y-1, z; t] +
        u[x, y, z+1; t] + u[x, y, z-1; t]
    )
    -1/12 * (
        u[x+2, y, z; t] + u[x-2, y, z; t] +
        u[x, y+2, z; t] + u[x, y-2, z; t] +
        u[x, y, z+2; t] + u[x, y, z-2; t]
    )
);
}

```

1.2 Generating The Code

We feed this stencil specification as an input to PATUS, which will turn it to C code. PATUS expects two other input files: a template defining how the code will be parallelized and how code optimizations will be applied, e.g., how loop tiling/cache blocking is applied. In PATUS lingo, this is called a *Strategy*. The other input is a description of the hardware architecture. It defines which code generation back-end to use (e.g., the OpenMP paradigm for shared memory CPU system, or NVIDIA C for CUDA for NVIDIA GPUs), and how arithmetic operations and data types are mapped to the corresponding vector intrinsics and vector data types, for instance.

Example 1.2: Generating the C code from a stencil specification.

```

java -jar patus.jar codegen --stencil=wave.stc
    --strategy=cacheblocking.stg
    --architecture="arch/architectures.xml,Intel x86_64 SSE"
    --outdir=output

```

The command in Ex. 1.2 will create an implementation for the stencil kernel specified in the file `wave.stc` and a benchmarking harness for that kernel and file it in the directory `output`. The Strategy chosen is a cache blocking strategy defined in the file `cacheblocking.stg`, which comes with the PATUS software. The hardware architecture for which the code will be generated is defined by the identifier `Intel x86_64 SSE`, which can be found in the architecture definition file, `arch/architectures.xml`.

After running the PATUS code generation, the directory `output` will contain the following files:

- `kernel.c` — The implementation of the stencil kernel defined in `wave.stc`.

- `driver.c` — The benchmarking harness invoking the stencil kernel and measuring the time for the stencil call. It allocates and initializes data with arbitrary values and (by default) validates the result returned by the stencil kernel by comparing it to a naïve sequential implementation.
- `timer.c` — Functions related to timing and calculating the performance.
- `patusrt.h` — Header file for the timing functions.
- `cycle.h` — Functions for counting clock cycles to do the timing. (This code has been taken from FFTW [1].)
- `Makefile` — A GNUmake Makefile to build the benchmarking harness.

The benchmarking harness then can be built by typing `make` on the command line. This will compile and link the generated code and produce the benchmarking executable, by default called `bench`.

1.3 Running and Tuning

The benchmark executable requires a number of parameters to run:

Example 1.3: *Starting the benchmark executable.*

```
chrmat@palu1:wave> ./bench
Wrong number of parameters. Syntax:
./bench <N> <cb_x> <cb_y> <cb_z> <chunk> <unroll_p3>
```

N corresponds to the N used in the stencil specification for the definition of the domain size. If additional identifiers would have been used to define the domain size, they would appear as parameters to the executable as well. The `cb_x`, `cb_y`, `cb_z`, and `chunk` arguments come from the Strategy. They specify the sizes of the cache blocks in x , y , and z directions and the number of consecutive cache blocks assigned to one thread. PATUS unrolls the inner-most loop nest containing the stencil evaluation. `_unroll_p3` selects one of the unroll configuration code variants: by default, PATUS creates code variants with the loop nest unrolled once (i.e., no unrolling is done) or twice in each direction. Since the example wave stencil is defined in 3 dimensions, there are $2^3 = 8$ code variants with different unrolling configurations.

In Ex. 1.4, the benchmark executable was run with a domain size of 200^3 grid points and an arbitrary cache block size of $16 \times 16 \times 16$ grid points per block, one block in a packet per thread, and the 0th loop unrolling configuration.

Example 1.4: *Running the benchmark executable with arbitrary parameters.*

```
chrmat@palu1:wave> ./bench 200 16 16 16 1 0
Flops / stencil call: 19
Stencil computations: 40000000
Bytes transferred: 509379840
Total Flops: 760000000
Seconds elapsed: 0.230204
Performance: 3.301418 GFlop/s
Bandwidth utilization: 2.212731 GB/s
506450156.000000
Validation OK.
```

The benchmark executable prints the number of floating point operations per stencil evaluation, the total number of stencil evaluations that were performed for the time measurement, the number of transferred bytes and the total number of floating point operations, the time it took to complete the sweeps and the calculated performance in GFlop/s and bandwidth utilization. The number below that is a representation of the time spent in the compute kernel, on which the auto-tuner bases the search. (It tries to minimize that number.) The string “Validation OK” says that the validation test (against the naïve, sequential implementation) was passed, i.e., the relative errors did not exceed certain bounds.

The idea is that the user chooses the problem size, N , but all the other parameters, which do not change the problem definition, but rather implementation details which affect the performance, are to be chosen by the auto-tuner. In the current state of the software, the user still needs some knowledge how to find the performance-specific parameters (in this case `cb_x`, `cb_y`, `cb_z`, `chunk`, `_unroll_p3`). An idea for future work is to encapsulate this knowledge in the Strategy, which defines the parameters, so that an auto-tuner configuration script can be generated in order to fully automate the auto-tuning process.

We choose $N = 200$. Experience tells us that in cases with relatively small domain size (such as $N = 200$), a good choice for `cb_x` is N . There are several reasons why cutting the domain in the x direction, which is the unit stride direction (i.e. choosing `cb_x` $< N$), is a bad idea. Reading data in a streaming fashion from DRAM is faster than jumping between addresses. Utilizing full cache lines maximizes data locality. And the hardware prefetcher is most effective when the constant-stride data streams are as long as possible. In Ex. 1.5 we let the auto-tuner choose `cb_y`, `cb_z`, `chunk`, and `_unroll_p3` by letting `cb_y` and `cb_z` from 4 to $N = 200$ in increments of 4 (4:4:200), and we want the auto-tuner to try all the powers of 2 between 1 and 16 (1:*2:16!) for the fifth command line argument, `chunk`, and try all the values between 0 and 7 (0:7!) for `_unroll_p3`, by which all the generated loop unrolling variants are referenced. The exclamation mark specifies that the corresponding parameter is searched exhaustively, i.e., the auto-tuner is instructed to visit all of the values in the range specified.

Example 1.5: *Running the PATUS auto-tuner.*

```
java -jar patus.jar autotune ./bench 200 200 4:4:200 4:4:200
1:*2:16! 0:7! "C((\ $1+\ $3-1)/\ $3)*((\ $1+\ $4-1)/\ $4)>=$OMP_NUM_THREADS"
```

The auto-tuner also allows to add constraints, such as the expression in the last argument in the call in Ex. 1.5. Constraints are preceded by a C and can be followed by any comparison expression involving arithmetic. Parameter values are referenced by a number preceded by a dollar sign \$; the numbering starts with 1. In Ex. 1.5, the condition reads $(\$1 + \$3 - 1)/\$3 \cdot (\$1 + \$4 - 1)/\$4 \geq T$. The sub-expression $(\$1 + \$3 - 1)/\$3$ is the number of blocks in the y -direction, using the integer division (which rounds towards zero) to express the missing ceiling function*). Similarly, $(\$1 + \$4 - 1)/\$4$ is the number of blocks in z -direction. Thus, the condition states that the total number of blocks must be greater or equal than the number of threads executing the program (assuming the environment variable `$OMP_NUM_THREADS` is set and controls how many OpenMP threads the program uses. Adding constraints is optional, but they can reduce the number of searches by restricting the search space. In this case, we simply exclude the points in the search space with bad performance, but constraints can be a helpful tool to suppress invalid configurations. For instance, the numbers of threads per block on a GPU must not exceed a particular number lest the program fails. (The numbers are 512 threads per block on older graphics cards, and 1024 threads per block on Fermi GPUs).

Ex. 1.6 shows an excerpt of the auto-tuner output. The program is run for many parameter configurations, and at the end of the search, the auto-tuner displays the parameter configuration and output of the run for which the best performance was achieved.

Example 1.6: *Output of the auto-tuner.*

```
./bench 200 200 4:4:200 4:4:200 1:*2:16! 0:7!
Parameter set { 200 }
Parameter set { 200 }
Parameter set { 4, 8, ...}
Parameter set { 4, 8, ...}
Parameter set , Exhaustive { 1, 2, 4, 8, 16 }
Parameter set , Exhaustive { 0, 1, 2, 3, 4, 5, 6, 7 }
Using optimizer: Powell search method
Executing [./bench, 200, 200, 4, 4, 1, 0]...
...

200 200 36 160 1 3
1.5052755E8
```

*For positive integers a, b and the integer division \div (defined by $a \div b := \lfloor \frac{a}{b} \rfloor$, it holds $\lceil \frac{a}{b} \rceil = (a + b - 1) \div b$.

Example 1.6: *Output of the auto-tuner. (cont.)*

```
Program output of the optimal run:  
Flops / stencil call: 19  
Stencil computations: 40000000  
Bytes transferred: 509379840  
Total Flops: 760000000  
Seconds elapsed: 0.068421  
Performance: 11.107680 GFlop/s  
Bandwidth utilization: 7.444774 GB/s  
150527550.000000  
Validation OK.
```


Chapter 2

Patus Usage

2.1 Code Generation

The PATUS code generator is invoked by the following command:

```
java -jar patus.jar codegen
  --stencil=<Stencil File>  --strategy=<Strategy File>
  --architecture=<Architecture Description File> ,<Hardware Name>
  [--outdir=<Output Directory> ] [--generate=<Target> ]
  [--kernel-file=<Kernel Output File Name> ]
  [--compatibility={ C | Fortran } ]
  [--unroll=<UnrollFactor1> ,<UnrollFactor2> ,... ]
  [--use-native-simd-datatypes={ yes | no } ]
  [--create-validation={ yes | no } ]
  [--validation-tolerance=<Tolerance> ]
  [--debug=<DebugOption1> ,[<DebugOption2> ,[... ,[<DebugOptionN> ] ... ]]
```

`--stencil=<Stencil File>`

Specifies the stencil specification file for which to generate code.

`--strategy=<Strategy File>`

The Strategy file describing the parallelization/optimization strategy.

`--architecture=<Architecture Description File> ,<Hardware Name>`

The architecture description file and the name of the selected architecture (as specified in the name attribute of the `architectureType` element).

`--outdir=<Output Directory>`

The output directory into which the generated files will be written. Optional; if not specified the generated files will be created in the current directory.

`--generate=<Target>`

The target that will be generated. *<Target>* can be one of:

- benchmark
This will generate a full benchmark harness. This is the default setting.
- kernel
This will only generate the kernel file.

`--kernel-file=<Kernel Output File Name>`

Specifies the name of the C source file to which the generated kernel is written. The suffix is appended or replaced from the definition in the hardware architecture description.
Defaults to `kernel.c`.

`--compatibility={C | Fortran}`

Selects whether the generated code has to be compatible with Fortran (Omits the double pointer output type in the kernel declaration; therefore multiple time steps are not supported.) Defaults to C.

`--unroll=<UnrollFactor1>,<UnrollFactor2>,...`

A list of unrolling factors applied to the inner most loop nest containing the stencil computation. The unrolling factors are applied to all the dimensions.

`--use-native-simd-datatypes={yes | no}`

Specifies whether the native SSE data type is to be used in the kernel signature. If set to yes, this also requires that the fields are padded correctly in unit stride direction.
Defaults to no.

`--create-validation={yes | no}`

Specifies whether to create code that will validate the result. If *<Target>* is not benchmark, this option will be ignored.
Defaults to yes.

`--validation-tolerance=<Tolerance>`

Sets the tolerance for the relative error in the validation. This option is only relevant if validation code is generated (`--create-validation=yes`). Defaults to yes.

`--debug=<DebugOption1>,[<DebugOption2>,[...,<DebugOptionN>]...]`

Specifies debug options (as a comma-separated list) that will influence the code generator. Valid debug options (for `<DebugOptioni>`, $i = 1, \dots, N$) are:

- `print-stencil-indices`
This will insert a `printf` statement for every stencil calculation with the index into the grid array at which the result is written.
- `print-validation-errors`
Prints all values if the validation fails. The option is ignored if no validation code is generated.

2.2 Auto-Tuning

The PATUS auto-tuner is invoked on the command line like so:

```
java -jar patus.jar autotune
  <Executable Filename>
  <Param1> <Param2> ... <ParamN>
  [ <Constraint1> <Constraint2> ... <ConstraintM> ]
  [ -m<Method> ]
```

`<Executable Filename>` is the path to the file name of the benchmark executable. The benchmark executable must expose the tunable parameters as command line parameters. The PATUS auto-tuner only generates numerical parameter values. If the benchmark executable requires strings, these must be mapped from numerical values internally in the benchmarking program.

The parameters $ParamI$, $I = 1, \dots, N$, define integer parameter ranges and have the following syntax:

`<StartValue>:[[*]<Step>:]<EndValue>[!]`

or

`<Value1>[,<Value2>[,<Value3>...]][!]`

The first version, when no asterisk $*$ in the $\langle Step \rangle$ is specified, enumerates all values in

$$\{a := \langle StartValue \rangle + k \cdot \langle Step \rangle : k \in \mathbb{N}_0 \text{ and } a \leq \langle EndValue \rangle\}.$$

If no $\langle Step \rangle$ is given, it defaults to 1. If there is an asterisk $*$ in front of the $\langle Step \rangle$, the values enumerated are

$$\{a := \langle StartValue \rangle \cdot \langle Step \rangle^k : k \in \mathbb{N}_0 \text{ and } a \leq \langle EndValue \rangle\}.$$

In the second version, all the comma-separated values $\langle Value1 \rangle$, $\langle Value2 \rangle$, ... are enumerated.

If the optional $!$ is appended to the value range specification, each of the specified values is guaranteed to be used, i.e., an exhaustive search is used for the corresponding parameter.

Example 2.1: *Specifying parameter ranges.*

1 : 10
enumerates all the integer numbers between and including 1 and 10.

2 : 2 : 41
enumerates the integers 2, 4, 6, ..., 38, 40.

1 : *2 : 128
enumerates some powers of 2, namely 1, 2, 4, 8, 16, 32, 64, 128.

Optional constraints can be specified to restrict the parameter values. The syntax for the constraints is

$$C\langle ComparisonExpression \rangle$$

where $\langle ComparisonExpression \rangle$ is an expression which can contain arithmetic operators $+$, $-$, $*$, and $/$, as well as comparison operators $<$, \leq , $=$, \geq , $>$, \neq , and variables $\$1, \dots, \N , which correspond to the parameters $\langle Param1 \rangle, \dots, \langle ParamN \rangle$.

Example 2.2: *Constraints examples.*

C\$2 <= \$1
forces the second parameter to be less or equal to the first.

C(\$2 + \$1 - 1)/\$1 >= 24
forces $\frac{\$2 + \$1 - 1}{\$1} = \left\lceil \frac{\$2}{\$1} \right\rceil \geq 24$.

The PATUS auto-tuner supports a range of search methods. The method can be selected by `-m<Method>` where `<Method>` is one of

- `ch.unibas.cs.hpwc.patus.autotuner.DiRectOptimizer`
DIRECT method
- `ch.unibas.cs.hpwc.patus.autotuner.ExhaustiveSearchOptimizer`
exhaustive search
- `ch.unibas.cs.hpwc.patus.autotuner.GeneralCombinedEliminationOptimizer`
general combined elimination
- `ch.unibas.cs.hpwc.patus.autotuner.GreedyOptimizer`
greedy search
- `ch.unibas.cs.hpwc.patus.autotuner.HookeJeevesOptimizer`
Hooke-Jeeves algorithm
- `ch.unibas.cs.hpwc.patus.autotuner.MetaHeuristicOptimizer`
genetic algorithm
- `ch.unibas.cs.hpwc.patus.autotuner.RandomSearchOptimizer`
draws 500 random samples
- `ch.unibas.cs.hpwc.patus.autotuner.SimplexSearchOptimizer`
simplex search (aka Nelder-Mead method)

These are Java class paths to `IOptimizer` implementations; this allows to extend range of methods easily. Please refer to Chapter ?? for a discussion and comparison of the methods. If no method is specified, the greedy algorithm is used by+- default.

Chapter 3

Integrating into User Code

By default, PATUS creates a C source file named `kernel.c` (The default setting can be overridden with the `--kernel-file` command line option, cf. Appendix 2.) This kernel file contains all the generated code variants of the stencil kernel, a function selecting one of the code variants, and an initialization function, `initialize`, which does the NUMA-aware data initialization and should preferably be called directly after allocating the data (cf. Chapter ??).

Example 3.1: *The generated stencil kernel code for the example wave stencil.*

The generated stencil kernel and the data initialization function have the following signatures:

```
void wave (float** u_0_1_out,
           float* u_0_m1, float* u_0_0, float* u_0_1, float c2dt_h2,
           int N,
           int cb_x, int cb_y, int cb_z, int chunk, int _unroll_p3);

void initialize (
    float* u_0_m1, float* u_0_0, float* u_0_1, float dt_dx_sq,
    int N,
    int cb_x, int cb_y, int cb_z, int chunk);
```

The selector function, which is named exactly as the stencil in the stencil specification (wave in Ex. 3.1), is the function, which should be called in the user code. Its parameters are:

- pointers to the grid array containing the results; these are double pointers and marked with the `_out` suffix;

- input grid arrays, one for each time index required to carry out the stencil computation; e.g., in the wave equation example, three time indices are required: the result time step $t + 1$, which depends on the input time steps t and $t - 1$; the time index is appended to the grid identifier as last suffix; the `m` stands for “minus”;
- any parameters defined in the stencil specification, e.g., `c2dt.h2` in the wave equation example;
- all the variables used to specify the size of the problem domain, only `N` in our example;
- Strategy- and optimization-related parameters, the best values of which were determined by the auto-tuner and can be substituted into the stencil kernel function call in the user code; the strategy used in the example has one cache block size parameter, `(cb_x, cb_y, cb_z)` and a chunk size `chunk`; furthermore, `_unroll_p3` determines the loop unrolling configuration.

The output pointers are required because PATUS rotates the time step grids internally, i.e., the input grids change roles after each spatial sweep. Thus, the user does typically not know which of the grid arrays contains the solution. A pointer to the solution grid is therefore assigned to the output pointer upon exit of the kernel function.

The initialization function can be modified to reflect the initial condition required by the problem. However, the generated loop structure should not be altered. Alternatively, a custom initialization can be done after calling the initialization function generated by PATUS.

Chapter 4

Alternate Entry Points to Patus

As the auto-tuner module is decoupled from the code generating system, it can be used as a stand-alone auto-tuner for other codes besides the ones created by PATUS. If the user has a hand-crafted existing parametrized code for which the best configurations need to be determined, the PATUS auto-tuner is a perfectly valid option to find the best set of parameters. The only requirements are that the tunable parameters must be exposed as command line parameters and that the program must print the timing information to `stdout` as last line of the program output in an arbitrary time measurement unit. Note that the auto-tuner tries to *minimize* this number, i.e., it must be a time measurement rather than a performance number.

Chapter 5

More Details on Patus Stencil Specifications

In the current version, PATUS stencil specifications consist of 3 parts: the size of the domain, a d -dimensional grid, to the points of which the stencil is applied; a specification of the number of time steps (sweeps) that is performed within one stencil kernel call; and, most importantly, the “operation”: the definition of the actual stencil expression(s), which is applied to each point of the grid(s) over which the computation sweeps.

The following listing shows a skeleton of a stencil specification:

```
stencil name
{
    domainsize = (xmin .. xmax, ymin .. ymax, ...);
    t_max = number of sweeps;

    operation (grid argument and parameter declarations)
    {
        stencil expressions
    }
}
```

Sweeps are not programmed explicitly in the stencil specification; the operation only contains a localized, point-wise stencil expression. Currently, PATUS supports Jacobi iterations, which means that the grids which are written to are distinct in memory from the grids that are read. In the future, other types of grid traversals, e.g., red-black Gauss-Seidel sweeps, may be supported. Given the Jacobi iteration as structural principle, there are no loop carried dependencies within the spatial iteration over the grid. Hence, the order in which the grid points are visited and the stencil expressions evaluated does not matter. This fact is actually exploited by the Strategies.

Both the domain size and the number of sweeps can contain symbolic identifiers, in which case these identifiers are added as arguments to the stencil kernel function in the generated code. The dimensionality of the stencil is implicitly defined by the dimensionality of the domain size and the subscripts in the stencil expressions. Currently, it is not possible to mix grids of different dimensionalities. $x_{\min}, x_{\max}, \dots$ can be integer literals, identifiers, or arithmetic expressions. *Number of sweeps* can be either an integer literal or an identifier if the number of time steps has to be controlled from outside the stencil kernel. Both minimum and maximum expressions in domain size specifications are inclusive, i.e., the domain size $(x_{\min}..x_{\max}, \dots)$ has $x_{\max} - x_{\min} + 1$ points in x -direction.

There can be arbitrarily many grid arguments to the stencil operation, depending on the application. For instance, discretizing the divergence operator, which maps vector fields to a scalar function, might be implemented in the 3D case with three grids, which are read, and one grid to which the result is written. Grid arguments (in first line in the following listing) and parameter declarations (second line) have the following form:

```
[const] (float|double) grid grid name [( x'_{\min} .. x'_{\max}, y'_{\min} .. y'_{\max}, ... )] ↵
  [[ number of grids ]]
(float|double) param parameter name [[ array size ]]
```

Both grids and parameters can either be based on single or double precision floating point data types. If a grid is declared to be *const*, it is assumed that the values do not change in time, i.e., the grid is read-only. The optional grid size specification after the grid name defines the size of the array as it is allocated in memory. This is useful if the kernel is to be built into an existing code. If no explicit size of the grid is specified, the size is calculated automatically by PATUS by taking the size of the iteration space *domainsize*, the inner domain, and adding the border layers required so that the stencil can be evaluated on each point of the inner domain. Again, $x'_{\min}, x'_{\max}, \dots$ can be integer literals, identifiers, which can be distinct from the identifiers used for the *domainsize* specification, or arithmetic expressions. Any additional identifiers used to specify the grid size will be added as arguments to the stencil kernel function. Multiple grids can be subsumed in one grid identifier. Instead of using 3 distinct grid identifiers in the above example of the divergence operator, the inputs can be declared as `float grid X[3]` rather than, e.g., `float grid Xx`, `float grid Xy`, `float grid Xz`. In the generated code, an array of grids will be split into distinct grid identifiers, however. The *number of grids* has to be an integer literal. Also, parameters can be both scalar identifiers or arrays.

The body of the operation consists of one or more assignment expression statements. The left hand side of the assignments can be a grid access to which the right hand side is assigned or a temporary scalar variable, which can be used later in the computation. The arithmetic expressions on the right hand side

involve previously calculated scalar variables or references to a grid point.

In the example below some flavors of assigning and referencing points in a grid are shown.

```

— assignment to a grid
u[x,y,z; t+1] = ...           // center point of next time step

— assignment of a value to temporary variables
float tmp1 = ...
double tmp2 = ...

— referencing a “regular” grid u (declared as float grid u)
... = u[x,y,z; t] + ...       // center point
... = u[x+1,y,z; t] + ...     // right neighbor
... = u[x,y,z; t-1] + ...     // center point of previous time step

— referencing an array of grids X (declared as float grid X[3])
... = X[x,y,z; t; 0] + ...    // center point of component 0

— referencing a const grid c (declared as const float grid c)
... = c[x,y,z] + ...          // center point of a const grid

— referencing an array of grids c (decl. as const float grid c[3])
... = c[x,y,z; 0] + ...       // center pt of component 0 of a const grid

```

The center point* is always denoted by $u[x, y, z; \bullet]$, neighboring points by $u[x \pm \delta_x, y \pm \delta_y, z \pm \delta_z; \bullet]$ if u is the identifier of the grid we wish to access and the stencil is defined in 3 dimensions. 2-dimensional grids would only use x and y as spatial references, for arbitrary-dimensional stencils spatial reference identifiers x_0, x_1, \dots are defined. The δ_\bullet must be integer literals, i.e., the neighborhood relationship to the center point must be constant and known at compile time.

Non-constant grids, i.e., grids which change their values in the time dimension and are read from and written to, require an additional index denoting the time step, which is interpreted relatively to the time step of the corresponding left hand side grid. The temporal reference identifier is always t . Note that no references to future time steps can occur on the right hand side, i.e., the temporal indices of the grid references in expressions on the right hand side must be strictly less than the temporal index on the left hand side. (If this was not the case, the method would be implicit, and a linear solver would be required to solve the problem.)

If an identifier has been declared to be an array of grids, an additional index has to be appended after the temporal one, which determines which array element to use. The index has to be an integer literal.

The complete grammar of the stencil specification DSL is given in Appendix A.

*Here, “center” always refers to the current point within a sweep. Sweeps are not explicitly programmed in the stencil specification as detailed above.

Chapter 6

Strategy Examples and Hardware Architecture Considerations

A PATUS Strategy is the means by which we aspire to implement parallelization and bandwidth-optimizing methods such as the ones discussed in Chapter ?? . In practice it mostly is at least cumbersome to adapt an implementation for a toy stencil (e.g., a 3D 7-point constant coefficient Laplacian) to a stencil occurring in an application, since the implementation of the algorithm most likely depends on the shape of the stencil and the grid arrays used, and very probably on the dimensionality of the stencil. Extending a coded template to incorporate a stencil computation for which it was not designed initially is tedious and error prone.

The idea of Strategies is to provide a clean mechanism which separates the implementation of parallelization and bandwidth-optimizing methods from the actual stencil computation. In this way, the implementation of the algorithm can be reused for arbitrary stencils.

6.1 A Cache Blocking Strategy

We start by looking at the implementations of a cache blocking method. The Strategy in Listing 6.1 iterates over all the time steps in the τ loop, and within one time step in blocks v of size cb over the *root domain* u , i.e., the entire domain to which to apply the stencil. Both the root domain and the size of the subdomain v are given as Strategy parameters. The blocks v are executed in parallel by virtue of the `parallel` keyword, which means that the subdomains v are dealt out in a cyclic fashion to the worker threads. The parameter `chunk` to the `schedule` keyword defines how many consecutive blocks one thread is given. Then, the stencil is applied for each point in the subdomain v .

The Strategy argument `cb` has a specifier, `auto`, which means that this parameter will be interfaced with the auto-tuner: it is exposed on the command line of the benchmarking harness so that the auto-tuner can provide values for $cb = (c_1, c_2, \dots, c_d)$, where d is the dimensionality of the stencil, and pick the one for which the best performance is measured.

```
strategy cacheblocked (domain u, auto dim cb, auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max
  {
    // iterate over subdomain
    for subdomain v(cb) in u(:, t) parallel schedule chunk
    {
      // calculate the stencil for each point in the subdomain
      for point p in v(:, t)
        v[p; t+1] = stencil (v[p; t]);
    }
  }
}
```

Listing 6.1: A cache blocking Strategy implementation.

In the Strategy of Listing 6.1, the parameter `cb` controls both the granularity of the parallelization and consequently the load balancing, and the size of the cache blocks. The size of the blocks ultimately limits the number of threads participating in the computation. If the blocks become too large the entire domain will be divided into less subdomains than there are threads available, and the performance will drop. In practice, the auto-tuner will prevent such cases. But the consequence is that a configuration for `cb`, which runs well for a specific number of threads might not perform well for another number of threads.

The Strategy in Listing 6.2 this one block `v` was split into smaller blocks `w`. Here, the idea is that `v` is responsible for the parallelization and load balancing, while the inner subdomain `w` is reserved for cache blocking.

```
strategy cacheblocked2 (domain u,
  auto dim tb, auto dim cb, auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max
  {
    // parallelization
    for subdomain v(tb) in u(:, t) parallel schedule chunk
    {
      // cache blocking
      for subdomain w(cb) in v(:, t)
      {
        for point pt in w(:, t)
          w[pt; t+1] = stencil (w[pt; t]);
      }
    }
  }
}
```

```

    }
  }
}

```

Listing 6.2: Another way of subdividing for cache blocking..

Although this is currently not done yet, restrictions could be inferred for w , limiting the search space the auto-tuner has to explore: Since w is an iterator within the subdomain v , we could restrict cb by $cb_{\bullet} \leq tb_{\bullet}$ (where tb is the size of v).

Also, we could infer a restriction preventing that threads are sitting idle by calculating the number of blocks v : Let (s_1, \dots, s_d) be the size of the root domain u . Then we could require that the following inequality holds:

$$\prod_{i=1}^d \left\lceil \frac{s_i}{tb_i} \right\rceil \geq T,$$

where T is the number of threads.

6.2 Independence of the Stencil

By concept, Strategies are designed to be independent of both the stencil and the concrete hardware platform. The obvious means to achieve independence of the stencil is to not specify the actual calculation in the Strategy, but have a reference to it instead. This is done by the formal `stencil` call, which expects a grid reference as argument (actually, a point within a domain) and assigns the result to another grid reference. Strategies therefore also must have the notion of grids, but interpreted as index spaces rather than actual mappings to values. Each Strategy has one required argument of type `domain`, the *root domain*, which represents the entire index space over which the Strategy is supposed to iterate. In the Strategy, this domain can be subdivided, defining the way in which the actual stencil grids are traversed. This is done using “subdomain iterators,” which are constructs that advance a smaller box, the iterator, within a larger one, its parent domain. The size of a Strategy domain is always the size of the computed output, i.e., not including the border layers required for a stencil computation.

Being independent of the stencil means in particular being independent of the stencil’s dimensionality. The root domain inherits the dimensionality of the stencil. Subdomains, however, need a notion of the dimensionality in order to specify their size. Strategies provide data types `dim` and `codim(n)`, which can be used in Strategy arguments. If the stencil dimensionality is d , a `dim` variable is a d -dimensional vector, and a `codim(n)` variable is a $(d - n)$ -dimensional vector (a vector of co-dimension n).

Again Consider the cache blocking Strategy in Listing 6.1. The size of the subdomain v being iterated over the root domain u is specified by `cb`, an argument to the strategy of type `dim`. This means that the subdomain v will have the same dimensionality as the stencil and the root domain.

The instruments provided by Strategies to deal with the unknown dimensionalities are the following:

- subdomain iterators rather than simple for loops,
- `dim` and `codim(n)` typed variables,
- subdomain and stencil properties: when a Strategy is parsed, both `w.dim` and `stencil.dim` are replaced by the dimensionality of the stencil and of the subdomain w , respectively,
- a subdomain's size property, which is a `dim`-typed variable containing the size of a subdomain (i.e., `w.size` is a d -dimensional vector with its components set to the size of w),
- subscripting `dim` or `codim(n)` type variables by an index vector. An index vector can be either
 - a single integer (e.g., the value of `w.size(1)` is the first component of the size vector of w);
 - a vector of integers (e.g., `w.size(1,2,4)` returns a 3-dimensional vector containing the first, second, and fourth component of the size of w);
 - a range (e.g., `w.size(2 .. 4)` returns a 3-dimensional vector containing the second, third, and fourth component of the size of w , or `w.size(1 .. w.dim-1)` returns a vector containing all the components of the size of w except the last one);
 - ellipses, which fill a vector in a non-greedy fashion so that the vector is of dimension `stencil.dim`:
 - * `(a ...)` is a d -dimensional vector with each component set to a (a must be a compile-time constant);
 - * `(a , b ...)` is a d -dimensional vector with the first component set to a , and all the others to b ;
 - * `(a , b ... c)` is a vector with the first component set to a and the last set to c , and all the components in the middle set to b ;
 - any combinations of the above.

The size of a subdomain might also depend on the structure or the “bounding box” of a stencil. This can be achieved by the `stencil.box` property, which can be used to enlarge a subdomain.

6.3 Independence of the Hardware Architecture

A Strategy's independence of the hardware architecture is given by the notions of general iterators over subdomains and generic data copy operations, and, more importantly, by the liberty of interpreting the parallelism defined in a strategy as a "may parallelism" rather than a "must parallelism".

The underlying model of the hardware is a simple hierarchical model, to some extent inspired by the OpenCL execution model [2]. The execution units are indexed by multi-dimensional indices, similar to OpenCL's "NDRange" index spaces. This guarantees that there is an optimal mapping to architectures that have hardware support for multi-dimensional indexing or have multidimensional indices built into the programming model such as CUDA or OpenCL. We call a level in the hierarchy a *parallelism level*. The dimension of the indices may differ in each parallelism level.

Each parallelism level entity can have its own local memory, which is only visible within and below that parallelism level. We allow the data transfers to the local memories to be either implicit or explicit, i.e., managed by hardware or software. Furthermore, we permit both synchronous and asynchronous data transfers.

According to this model, a shared-memory CPU architecture has one parallelism level with local memory (cache) with implicit data transfer, and a CUDA-capable GPU has two parallelism levels, streaming multiprocessors and streaming processors – or thread blocks and threads, respectively. The thread block level has an explicit transfer local memory, namely the per-multiprocessor shared on-chip memory.

An architecture description together with the PATUS back-end code generators specific for a hardware platform are responsible for the correct mapping to the programming model. In particular, nested parallelism within a Strategy is mapped to subsequent parallelism levels in the model. If a hardware platform has less parallelism levels than given in a Strategy, the parallel Strategy entities will be just mapped onto the last parallelism level of the architecture and executed sequentially.

Domain decomposition and mapping to the hardware is implicitly given by a Strategy. Every subdomain iterator, e.g., "`for subdomain v(size_v) in u(;; t) ...`" decomposes the domain `u` into subdomains of smaller size `size_v`. When, in addition, the `parallel` keyword is used on a subdomain iterator, the loop is assigned to the next parallelism level (if there is one available), and each of the iterator boxes is assigned to an execution unit on that parallelism level. All the loops within the iterator also belong to the same parallelism level (i.e., are executed by the units on that level), until another parallel loop is encountered in the loop nest.

If a parallelism level requests explicit data copies, memory objects are allo-

cated for an iterator box as “late” as possible: since local memories tend to be small, the iterator within the parallelism level with the smallest boxes, i.e., as deep down in the loop nest as possible (such that the loop still belongs to the parallelism level), is selected to be associated with the memory objects. The sizes of the memory objects are derived from the box size of that iterator, and data from the memory objects associated with the parallelism level above are transferred. In the case that the iterator contains a stencil call within a nested loop on the same parallelism level, the iterator immediately above a point iterator “`for point p in v(:, t) ...`” is selected, if there is one, or if there is no such iterator, the iterator containing the stencil call is selected.

The Strategy excerpt

```
for subdomain v(size_v) in u(:, t) parallel
  for subdomain w(size_w) in v(:, t) parallel
    for point p in w(:, t)
      ...
```

and the resulting data decomposition and the ideal hardware mapping are visualized in Fig. 6.1. The upper layer shows the hierarchic domain decomposition of u into v and w . The bottom layer shows an abstraction of the hardware architecture with 2 parallelism levels, work groups and work items, which both can have a local memory. By making the v loop in the Strategy `parallel`, it is assigned to the first parallelism level, labeled “work groups” in the figure (following the OpenCL nomenclature). And by making the nested w loop `parallel`, this, in turn, is assigned to the second parallelism level, the work items within a work group. The points p in w are all assigned to the same work item that “owns” w .

6.4 Examples of Generated Code

To conclude this chapter, we show two examples how a simple strategy iterator is mapped to code, once for a OpenMP-parallelized shared memory CPU system, and once for a CUDA-programmed NVIDIA GPU.

Listing 6.3 shows an example of a generated C code using OpenMP for parallelization. It was generated for a 3D stencil from the parallel Strategy subdomain iterator “`for subdomain v(cb) in u(:, t) parallel ...`”. OpenMP provides one level of one-dimensional indexing (by the thread number, `omp_get_thread_num()`), but the domain to decomposed is 3-dimensional. Thus, the 3-dimensional index range

$$[v_idx_x, v_idx_x_max] \times [v_idx_y, v_idx_y_max] \times [v_idx_z, v_idx_z_max]$$

is calculated based on the thread ID. By incrementing the loop index v_idx by the number of threads, the v blocks are dealt out cyclically.

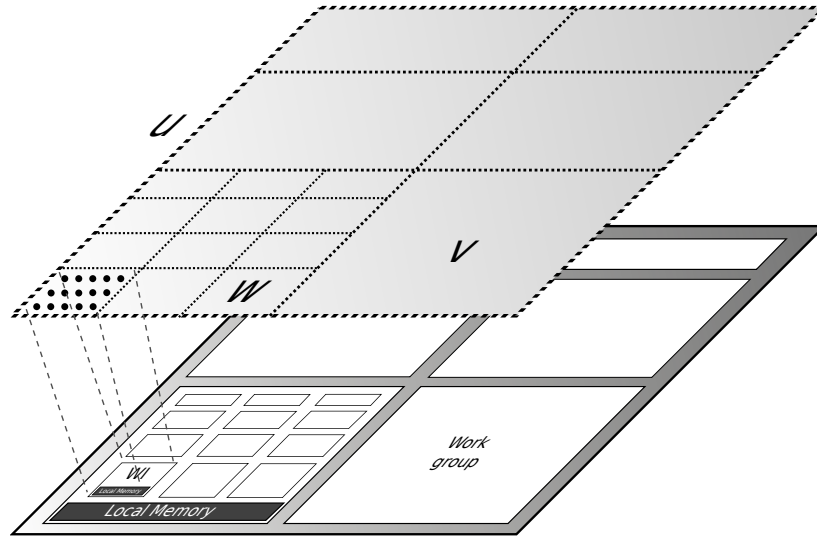


Figure 6.1: Mapping between data and hardware. Both hardware architecture (bottom layer) and data (top layer) are viewed hierarchically: the domain u is subdivided into v and w , the hardware architecture groups parallel execution units on multiple levels together.

```

int dimidx0 = omp_get_thread_num();
int dimsize0 = omp_get_num_threads();
int v_numblocks =
    ((x_max+cb_x-1)/cb_x)*((y_max+cb_y-1)/cb_y)*((z_max+cb_z-1)/cb_z);
for (v_idx=dimidx0; v_idx <= v_numblocks-1; v_idx += dimsize0)
{
    tmp_stride_0z = ((x_max+cb_x-1)/cb_x)*((y_max+cb_y-1)/cb_y);
    v_idx_z = v_idx/tmp_stride_0z;
    tmpidxc0 = v_idx-(v_idx_z*tmp_stride_0z);
    v_idx_y = tmpidxc0/((x_max+cb_x-1)/cb_x);
    tmpidxc0 -= v_idx_y*((x_max+cb_x-1)/cb_x);
    v_idx_x = tmpidxc0;
    v_idx_x = v_idx_x*cb_x+1;
    v_idx_x_max = min(v_idx_x+cb_x, x_max+1);
    v_idx_y = v_idx_y*cb_y+1;
    v_idx_y_max = min(v_idx_y+cb_y, y_max+1);
    v_idx_z = v_idx_z*cb_z+1;
    v_idx_z_max = min(v_idx_z+cb_z, z_max+1);

    // inner loops/computation within [v_idx_x, v_idx_x_max] x ...
}

```

Listing 6.3: C/OpenMP code generated for a 3D stencil from the Strategy iterator “for subdomain $v(cb)$ in u (; t) parallel ...”.

In contrast, CUDA provides 2 levels of indexing, the thread block and the

thread level. Moreover, the indices on the thread block level can be 1 or 2-dimensional (prior to CUDA 4.0) or 1, 2, or 3-dimensional in CUDA 4.0 on a Fermi GPU, and thread indices can be 1, 2, or 3-dimensional. Also, a GPU being a many-core architecture, we assume that there are enough threads to compute the iterates in v completely in parallel. Hence, there is no loop iterating over the domain, but a conditional guarding the “iteration” space instead as shown in Listing 6.4.

```
stride_1 = (blockDim.y+y_max-1)/blockDim.y;
tmp = blockIdx.y;
idx_1_2 = tmp/stride_1;
tmp -= idx_1_2*stride_1;
idx_1_1 = tmp;
v_idx_x = 1+(blockDim.x*blockIdx.x+threadIdx.x)*cbx;
v_idx_x_max = v_idx_x+cbx;
v_idx_y = threadIdx.y+idx_1_1*blockDim.y+1;
v_idx_y_max = v_idx_y+1;
v_idx_z = threadIdx.z+idx_1_2*blockDim.z+1;
v_idx_z_max = v_idx_z+1;

if (((v_idx_x<=x_max)&&(v_idx_y<=y_max))&&(v_idx_z<=z_max)) {
    // inner loops/computation within [v_idx_x, v_idx_x_max] x ...
}
```

Listing 6.4: C for CUDA code generated for a 3D stencil from the Strategy iterator “for subdomain v (cbx, 1 ...) in u (; t) parallel ...”.

Again, the 3-dimensional index range

$$[v_idx_x, v_idx_x_max] \times [v_idx_y, v_idx_y_max] \times [v_idx_z, v_idx_z_max]$$

is calculated before the actual calculation, but now based on the 2-dimensional thread block indices (`blockIdx.x`, `blockIdx.y`) and the 3-dimensional thread indices (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`).

`blockIdx` and `threadIdx` are built-in variables in C for CUDA.

Chapter 7

Current Limitations

In the current state, there are several limitations to the PATUS framework:

- Only shared memory architectures are supported (specifically: shared memory CPU systems and single-GPU setups).
- It is assumed that the evaluation order of the stencils within one spatial sweep is irrelevant. Also, always all points within the domain are traversed per sweep. One grid array is read and another array is written to. Such a grid traversal is called a Jacobi iteration. In particular, this rules out schemes with special traversal rules such as red-black Gauss-Seidel iterations.
- No extra boundary handling is applied. The stencil is applied to every interior grid point, but not to boundary points. I.e., the boundary values are kept constant; this corresponds to Dirichlet boundary conditions. To implement boundary conditions, they could be factored into the stencil operation by means of extra coefficient grids. In the same way, non-rectilinear domain shapes and non-uniform grids can be emulated by providing the shape and/or geometry information encoded as coefficients in additional grids. Alternatively, special $(d - 1)$ -dimensional stencil kernels could be specified for the boundary treatment, which are invoked after the d -dimensional stencil kernel operating on the interior. This approach, however, will invalidate temporal blocking schemes.
- The index calculation assumes that the stencil computation is carried out on a flat grid (or a grid which is homotopic to a flat grid). In particular, currently no spherical or torical geometries are implemented, which require modulo index calculations.

- There is no support for temporally blocked schemes yet.

Bibliography

- [1] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on “Program Generation, Optimization, and Platform Adaptation”. [cited at p. 6]
- [2] Khronos OpenCL Working Group. *The OpenCL Specification*, 8 December 2008. [cited at p. 29]

Appendices

Appendix A

Patu Grammars

A.1 Stencil DSL Grammar

In the following, the EBNF grammar for the PATUS stencil specifications syntax is given. The grayed out identifiers have not yet been specified or implemented and will be added eventually in the future.

```
⟨Stencil⟩ ::= 'stencil' ⟨Identifier⟩ '{' [ ⟨Options⟩ ] ⟨DomainSize⟩ ⟨NumIterations⟩
           ⟨Operation⟩ ⟨Boundary⟩ [ ⟨Filter⟩ ] [ ⟨StoppingCriterion⟩ ] '}'

⟨DomainSize⟩ ::= 'domainsize' '=' ⟨Box⟩ ';'

⟨NumIterations⟩ ::= 't_max' '=' ⟨IntegerExpr⟩ ';'

⟨Operation⟩ ::= 'operation' ⟨Identifier⟩ '(' ⟨ParamList⟩ ')' '{' { ⟨Statement⟩ } '}'

⟨ParamList⟩ ::= { ⟨GridDecl⟩ | ⟨ParamDecl⟩ }

⟨Statement⟩ ::= ⟨LHS⟩ '=' ⟨StencilExpr⟩ ';'

⟨LHS⟩ ::= ⟨StencilNode⟩ | ⟨VarDecl⟩

⟨StencilExpr⟩ ::= ⟨StencilNode⟩ | ⟨Identifier⟩ | ⟨NumberLiteral⟩ | ⟨FunctionCall⟩
               | ( ⟨UnaryOperator⟩ ⟨StencilExpr⟩ ) | ( ⟨StencilExpr⟩ ⟨BinaryOperator⟩
               ⟨StencilExpr⟩ ) | '(' ⟨StencilExpr⟩ ')'

⟨StencilNode⟩ ::= ⟨Identifier⟩ '[' ⟨SpatialCoords⟩ [ ';' ⟨TemporalCoord⟩ ] [ ';'
               ⟨ArrayIndices⟩ ] ']'

⟨SpatialCoords⟩ ::= ( 'x' | 'y' | 'z' | 'u' | 'v' | 'w' | 'x' ⟨IntegerLiteral⟩ ) [ ⟨Offset⟩ ]

⟨TemporalCoord⟩ ::= 't' [ ⟨Offset⟩ ]

⟨ArrayIndices⟩ ::= ⟨IntegerLiteral⟩ { ',' ⟨IntegerLiteral⟩ }

⟨Offset⟩ ::= ⟨UnaryOperator⟩ ⟨IntegerLiteral⟩

⟨FunctionCall⟩ ::= ⟨Identifier⟩ '(' [ ⟨StencilExpr⟩ { ',' ⟨StencilExpr⟩ } ] ')'
```

$\langle \text{IntegerExpr} \rangle ::= \langle \text{Identifier} \rangle \mid \langle \text{IntegerLiteral} \rangle \mid \langle \text{FunctionCall} \rangle \mid ($
 $\quad \langle \text{UnaryOperator} \rangle \langle \text{IntegerExpr} \rangle) \mid (\langle \text{IntegerExpr} \rangle \langle \text{BinaryOperator} \rangle$
 $\quad \langle \text{IntegerExpr} \rangle) \mid '(\langle \text{IntegerExpr} \rangle)'$
 $\langle \text{VarDecl} \rangle ::= \langle \text{Type} \rangle \langle \text{Identifier} \rangle$
 $\langle \text{Box} \rangle ::= '(\langle \text{Range} \rangle \{ ', \langle \text{Range} \rangle)'$
 $\langle \text{Range} \rangle ::= \langle \text{IntegerExpr} \rangle '..' \langle \text{IntegerExpr} \rangle$
 $\langle \text{GridDecl} \rangle ::= [\langle \text{Specifier} \rangle] \langle \text{Type} \rangle \text{'grid'} \langle \text{Identifier} \rangle ['(\langle \text{Box} \rangle)'] [$
 $\quad \langle \text{ArrayDecl} \rangle]$
 $\langle \text{ParamDecl} \rangle ::= \langle \text{Type} \rangle \text{'param'} \langle \text{Identifier} \rangle [\langle \text{ArrayDecl} \rangle]$
 $\langle \text{ArrayDecl} \rangle ::= '[' \langle \text{IntegerLiteral} \rangle \{ ', \langle \text{IntegerLiteral} \rangle \} ']'$
 $\langle \text{Specifier} \rangle ::= \text{'const'}$
 $\langle \text{Type} \rangle ::= \text{'float'} \mid \text{'double'}$
 $\langle \text{UnaryOperator} \rangle ::= \text{'+'} \mid \text{'-'}$
 $\langle \text{BinaryOperator} \rangle ::= \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{'^'}$

A.2 Strategy DSL Grammar

The following EBNF grammar specifies the PATUS Strategy syntax. Again, as the project matures, the specification might change so that yet missing aspects of parallelization and optimization methods can be specified as PATUS Strategies.

$\langle \text{Strategy} \rangle ::= \text{'strategy'} \langle \text{Identifier} \rangle '(\langle \text{ParamList} \rangle)' \langle \text{CompoundStatement} \rangle$
 $\langle \text{ParamList} \rangle ::= \langle \text{SubdomainParam} \rangle \{ ', \langle \text{AutoTunerParam} \rangle \}$
 $\langle \text{SubdomainParam} \rangle ::= \text{'domain'} \langle \text{Identifier} \rangle$
 $\langle \text{AutoTunerParam} \rangle ::= \text{'auto'} \langle \text{AutoTunerDeclSpec} \rangle \langle \text{Identifier} \rangle$
 $\langle \text{AutoTunerDeclSpec} \rangle ::= \text{'int'} \mid \text{'dim'} \mid (\text{'codim'} '(\langle \text{IntegerLiteral} \rangle)')$
 $\langle \text{Statement} \rangle ::= \langle \text{DeclarationStatement} \rangle \mid \langle \text{AssignmentStatement} \rangle \mid$
 $\quad \langle \text{CompoundStatement} \rangle \mid \langle \text{IfStatement} \rangle \mid \langle \text{Loop} \rangle$
 $\langle \text{DeclarationStatement} \rangle ::= \langle \text{DeclSpec} \rangle \langle \text{Identifier} \rangle \text{';'}$
 $\langle \text{AssignmentStatement} \rangle ::= \langle \text{LValue} \rangle \text{'='} \langle \text{Expr} \rangle \text{';'}$
 $\langle \text{CompoundStatement} \rangle ::= \text{'\{'} \{ \langle \text{Statement} \rangle \} \text{'\}'}$
 $\langle \text{IfStatement} \rangle ::= \text{'if'} '(\langle \text{ConditionExpr} \rangle)' \langle \text{Statement} \rangle [\text{'else'} \langle \text{Statement} \rangle]$
 $\langle \text{Loop} \rangle ::= (\langle \text{RangeIterator} \rangle \mid \langle \text{SubdomainIterator} \rangle) [\text{'parallel'} [\langle \text{IntegerLiteral} \rangle$
 $\quad] [\text{'schedule'} \langle \text{IntegerLiteral} \rangle]] \langle \text{Statement} \rangle$
 $\langle \text{RangeIterator} \rangle ::= \text{'for'} \langle \text{Identifier} \rangle \text{'='} \langle \text{Expr} \rangle '..' \langle \text{Expr} \rangle [\text{'by'} \langle \text{Expr} \rangle]$
 $\langle \text{SubdomainIterator} \rangle ::= \text{'for'} \langle \text{SubdomainIteratorDecl} \rangle \text{'in'} \langle \text{Identifier} \rangle '(\langle \text{Range} \rangle$
 $\quad \text{';' } \langle \text{Expr} \rangle)'$

$\langle \text{SubdomainIteratorDecl} \rangle ::= \langle \text{PointDecl} \rangle \mid \langle \text{PlaneDecl} \rangle \mid \langle \text{SubdomainDecl} \rangle$
 $\langle \text{PointDecl} \rangle ::= \text{'point'} \langle \text{Identifier} \rangle$
 $\langle \text{PlaneDecl} \rangle ::= \text{'plane'} \langle \text{Identifier} \rangle$
 $\langle \text{SubdomainDecl} \rangle ::= \text{'subdomain'} \langle \text{Identifier} \rangle \text{'('} \langle \text{Range} \rangle \text{'})'$
 $\langle \text{Range} \rangle ::= \langle \text{Vector} \rangle \{ \langle \text{UnaryOperator} \rangle \langle \text{ScaledBorder} \rangle \}$
 $\langle \text{Vector} \rangle ::= \langle \text{Subvector} \rangle [\text{'...'} [\text{' '}, \langle \text{Subvector} \rangle]]$
 $\langle \text{Subvector} \rangle ::= (\text{' '}, \{ \text{' '}, \langle \text{ScalarList} \rangle \}) \mid \langle \text{DimensionIdentifier} \rangle \mid$
 $\quad \langle \text{DomainSizeExpr} \rangle \mid \langle \text{BracketedVector} \rangle \mid \langle \text{ScalarList} \rangle$
 $\langle \text{ScalarList} \rangle ::= \langle \text{ScalarRange} \rangle \{ \text{' '}, \langle \text{ScalarRange} \rangle \}$
 $\langle \text{DimensionIdentifier} \rangle ::= \langle \text{Expr} \rangle [\text{'('} \langle \text{Vector} \rangle \text{'})'$
 $\langle \text{DomainSizeExpr} \rangle ::= \langle \text{SizeProperty} \rangle [\text{'('} \langle \text{Vector} \rangle \text{'})'$
 $\langle \text{BracketedVector} \rangle ::= \text{'('} \langle \text{Vector} \rangle \{ \text{' '}, \langle \text{Vector} \rangle \} \text{'})'$
 $\langle \text{ScalarRange} \rangle ::= \langle \text{Expr} \rangle [\text{'..'} \langle \text{Expr} \rangle]$
 $\langle \text{SizeProperty} \rangle ::= (\text{'stencil'} \mid \langle \text{Identifier} \rangle) \text{'.'} (\text{'size'} \mid \text{'min'} \mid \text{'max'})$
 $\langle \text{ScaledBorder} \rangle ::= [\langle \text{Expr} \rangle \langle \text{MultiplicativeOperator} \rangle] \langle \text{Border} \rangle [$
 $\quad \langle \text{MultiplicativeOperator} \rangle \langle \text{Expr} \rangle]$
 $\langle \text{Border} \rangle ::= \langle \text{StencilBoxBorder} \rangle \mid \langle \text{LiteralBorder} \rangle$
 $\langle \text{StencilBoxBorder} \rangle ::= \text{'stencil'} \text{'.'} \text{'box'} [\text{'('} \langle \text{Vector} \rangle \text{'})'$
 $\langle \text{LiteralBorder} \rangle ::= \text{'('} \langle \text{Vector} \rangle \text{'})' \text{' '}, \text{'('} \langle \text{Vector} \rangle \text{'})'$
 $\langle \text{LValue} \rangle ::= \langle \text{GridAccess} \rangle \mid \langle \text{Identifier} \rangle$
 $\langle \text{GridAccess} \rangle ::= \langle \text{Identifier} \rangle \text{'['} \langle \text{SpatialIndex} \rangle \text{' '}, \langle \text{Expr} \rangle \{ \text{' '}, \langle \text{Expr} \rangle \} \text{' '}]'$
 $\langle \text{SpatialIndex} \rangle ::= \langle \text{Identifier} \rangle \mid \langle \text{Range} \rangle$
 $\langle \text{Expr} \rangle ::= \langle \text{Identifier} \rangle \mid \langle \text{NumberLiteral} \rangle \mid \langle \text{FunctionCall} \rangle \mid (\langle \text{UnaryOperator} \rangle$
 $\quad \langle \text{Expr} \rangle) \mid (\langle \text{Expr} \rangle \langle \text{BinaryOperator} \rangle \langle \text{Expr} \rangle) \mid \text{'('} \langle \text{Expr} \rangle \text{'})'$
 $\langle \text{FunctionCall} \rangle ::= \langle \text{Identifier} \rangle \text{'('} [\langle \text{Expr} \rangle \{ \text{' '}, \langle \text{Expr} \rangle \}] \text{'})'$
 $\langle \text{ConditionExpr} \rangle ::= \langle \text{ComparisonExpr} \rangle \mid (\langle \text{ConditionExpr} \rangle \langle \text{LogicalOperator} \rangle$
 $\quad \langle \text{ConditionExpr} \rangle)$
 $\langle \text{ComparisonExpr} \rangle ::= \langle \text{Expr} \rangle \langle \text{ComparisonOperator} \rangle \langle \text{Expr} \rangle$
 $\langle \text{UnaryOperator} \rangle ::= \text{'+'} \mid \text{'-'}$
 $\langle \text{MultiplicativeOperator} \rangle ::= \text{'*'}$
 $\langle \text{BinaryOperator} \rangle ::= \text{'+'} \mid \text{'-'} \mid \text{'*'}$
 $\quad \mid \text{'/'}$
 $\quad \mid \text{'\%'}$
 $\langle \text{LogicalOperator} \rangle ::= \text{'||'} \mid \text{'\&\&'}$
 $\langle \text{ComparisonOperator} \rangle ::= \text{'<'} \mid \text{'<='}$
 $\quad \mid \text{'=='}$
 $\quad \mid \text{'>='}$
 $\quad \mid \text{'>'}$
 $\quad \mid \text{'!=}'$

Appendix B

Stencil Specifications

B.1 Basic Differential Operators

B.1.1 Laplacian

```
stencil laplacian
{
  domainsize = (1 .. N, 1 .. N, 1 .. N);
  t_max = 1;

  operation (float grid u, float param alpha, float param beta)
  {
    u[x, y, z; t+1] =
      alpha * u[x, y, z; t] +
      beta * (
        u[x+1, y, z; t] + u[x-1, y, z; t] +
        u[x, y+1, z; t] + u[x, y-1, z; t] +
        u[x, y, z+1; t] + u[x, y, z-1; t]);
  }
}
```

B.1.2 Divergence

```
stencil divergence
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (
    float grid u(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    const float grid ux(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    const float grid uy(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
```

```

const float grid uz(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
float param alpha, float param beta, float param gamma)
{
    u[x, y, z; t] =
        alpha * (ux[x+1, y, z] - ux[x-1, y, z]) +
        beta * (uy[x, y+1, z] - uy[x, y-1, z]) +
        gamma * (uz[x, y, z+1] - uz[x, y, z-1]);
}
}

```

B.1.3 Gradient

```

stencil gradient
{
    domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
    t_max = 1;

    operation (
        const float grid u(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
        float grid ux(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
        float grid uy(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
        float grid uz(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
        float param alpha, float param beta, float param gamma)
    {
        ux[x, y, z; t] = alpha * (u[x+1, y, z] + u[x-1, y, z]);
        uy[x, y, z; t] = beta * (u[x, y+1, z] + u[x, y-1, z]);
        uz[x, y, z; t] = gamma * (u[x, y, z+1] + u[x, y, z-1]);
    }
}

```

B.2 Wave Equation

```

stencil wave
{
    domainsize = (1 .. N, 1 .. N, 1 .. N);
    t_max = 100;

    operation (float grid u, float param c2dt_h2)
    {
        u[x, y, z; t+1] = 2 * u[x, y, z; t] - u[x, y, z; t-1] +
            c2dt_h2 * (
                -15/2 * u[x, y, z; t] +
                4/3 * (
                    u[x+1, y, z; t] + u[x-1, y, z; t] +
                    u[x, y+1, z; t] + u[x, y-1, z; t] +
                    u[x, y, z+1; t] + u[x, y, z-1; t]
                )
                -1/12 * (

```



```

        u[x+2, y, z; t] + u[x-2, y, z; t] +
        u[x, y+2, z; t] + u[x, y-2, z; t] +
        u[x, y, z+2; t] + u[x, y, z-2; t]
    )
    );
}

```

B.3 COSMO

B.3.1 Upstream

```

stencil upstream_5_3d
{
    domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
    t_max = 1;

    operation (double grid u, double param a)
    {
        u[x, y, z; t+1] = a * (
            -2 * (u[x-3, y, z; t] + u[x, y-3, z; t] + u[x, y, z-3; t]) +
            15 * (u[x-2, y, z; t] + u[x, y-2, z; t] + u[x, y, z-2; t]) +
            -60 * (u[x-1, y, z; t] + u[x, y-1, z; t] + u[x, y, z-1; t]) +
            20 * u[x, y, z; t] +
            30 * (u[x+1, y, z; t] + u[x, y+1, z; t] + u[x, y, z+1; t]) +
            -3 * (u[x+2, y, z; t] + u[x, y+2, z; t] + u[x, y, z+2; t]);
    }
}

```

B.3.2 Tricubic Interpolation

```

stencil tricubic_interpolation
{
    domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
    t_max = 1;

    operation (double grid u,
        const double grid a, const double grid b, const double grid c)
    {
        double w1_a = 1.0/6.0*a[x,y,z]*(a[x,y,z]+1.0)*(a[x,y,z]+2.0);
        double w2_a = -0.5*(a[x,y,z]-1.0)*(a[x,y,z]+1.0)*(a[x,y,z]+2.0);
        double w3_a = 0.5*(a[x,y,z]-1.0)*a[x,y,z]*(a[x,y,z]+2.0);
        double w4_a = -1.0/6.0*(a[x,y,z]-1.0)*a[x,y,z]*(a[x,y,z]+1.0);

        double w1_b = 1.0/6.0*b[x,y,z]*(b[x,y,z]+1.0)*(b[x,y,z]+2.0);
        double w2_b = -0.5*(b[x,y,z]-1.0)*(b[x,y,z]+1.0)*(b[x,y,z]+2.0);
        double w3_b = 0.5*(b[x,y,z]-1.0)*b[x,y,z]*(b[x,y,z]+2.0);
        double w4_b = -1.0/6.0*(b[x,y,z]-1.0)*b[x,y,z]*(b[x,y,z]+1.0);
    }
}

```

```

double w1_c = 1.0/6.0*c[x,y,z]*(c[x,y,z]+1.0)*(c[x,y,z]+2.0);
double w2_c = -0.5*(c[x,y,z]-1.0)*(c[x,y,z]+1.0)*(c[x,y,z]+2.0);
double w3_c = 0.5*(c[x,y,z]-1.0)*c[x,y,z]*(c[x,y,z]+2.0);
double w4_c = -1.0/6.0*(c[x,y,z]-1.0)*c[x,y,z]*(c[x,y,z]+1.0);

u[x, y, z; t+1] =
  w1_a * w1_b * w1_c * u[x-1, y-1, z-1; t] +
  w2_a * w1_b * w1_c * u[x, y-1, z-1; t] +
  w3_a * w1_b * w1_c * u[x+1, y-1, z-1; t] +
  w4_a * w1_b * w1_c * u[x+2, y-1, z-1; t] +
  ... // etc. for all 64 combinations of w?_a * w?_b * w?_c
      // and u[x+d1, y+d2, z+d3; t], d1,d2,d3=-1,0,1,2
}
}

```

B.4 Hyperthermia

```

stencil hyperthermia
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (
    float grid T(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    const float grid c(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1)[9])
  {
    T[x, y, z; t+1] =

      // center point
      T[x, y, z; t] * (c[x, y, z; 0] * T[x, y, z; t] + c[x, y, z; 1]) +
      c[x, y, z; 2] +

      // faces
      c[x, y, z; 3] * T[x-1, y, z; t] +
      c[x, y, z; 4] * T[x+1, y, z; t] +
      c[x, y, z; 5] * T[x, y-1, z; t] +
      c[x, y, z; 6] * T[x, y+1, z; t] +
      c[x, y, z; 7] * T[x, y, z-1; t] +
      c[x, y, z; 8] * T[x, y, z+1; t];
  }
}

```

B.5 Anelastic Wave Propagation

B.5.1 uxx1

```

stencil pmcl3d_uxx1
{
    domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
    t_max = 1;

    operation (
        const float grid d1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        float grid u1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        const float grid xx(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        const float grid xy(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        const float grid xz(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        float param dth)
    {
        float c1 = 9./8.;
        float c2 = -1./24.;

        float d = 0.25 *
            (d1[x,y,z] + d1[x,y-1,z] + d1[x,y,z-1] + d1[x,y-1,z-1]);
        u1[x,y,z; t+1] = u1[x,y,z; t] + (dth / d) * (
            c1 * (
                xx[x, y,z] - xx[x-1,y,z] +
                xy[x,y, z] - xy[x,y-1,z] +
                xz[x,y,z ] - xz[x,y,z-1]) +
            c2 * (
                xx[x+1,y,z] - xx[x-2,y,z] +
                xy[x,y+1,z] - xy[x,y-2,z] +
                xz[x,y,z+1] - xz[x,y,z-2])
        );
    }
}

```

B.5.2 xy1

```

stencil pmcl3d_xy1
{
    domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
    t_max = 1;

    operation (
        const float grid mu(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        float grid xy(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        const float grid u1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        const float grid v1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
        float param dth)
    {
        float c1 = 9. / 8.;
        float c2 = -1. / 24.;

        float xmu = 2. / (1. / mu[x, y, z] + 1. / mu[x, y, z-1]);
    }
}

```

```

xy[x, y, z; t+1] = xy[x, y, z; t] + dth * xmu * (
  c1 * (
    u1[x, y+1, z] - u1[x, y, z] +
    v1[x, y, z] - v1[x-1, y, z]
  ) +
  c2 * (
    u1[x, y+2, z] - u1[x, y-1, z] +
    v1[x+1, y, z] - v1[x-2, y, z]
  )
);
}
}

```

B.5.3 xyz1

```

stencil pmcl3d_xyz1
{
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid mu(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid lam(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid u1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid v1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid w1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid xx(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid yy(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid zz(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float param dth)
  {
    float c1 = 9./8.;
    float c2 = -1./24.;

    float b = 8. / (
      1. / lam[x, y, z] + 1. / lam[x+1, y, z] +
      1. / lam[x, y-1, z] + 1. / lam[x+1, y-1, z] +
      1. / lam[x, y, z-1] + 1. / lam[x+1, y, z-1] +
      1. / lam[x, y-1, z-1] + 1. / lam[x+1, y-1, z-1]
    );

    float a = b + 2. * 8. / (
      1. / mu[x, y, z] + 1. / mu[x+1, y, z] +
      1. / mu[x, y-1, z] + 1. / mu[x+1, y-1, z] +
      1. / mu[x, y, z-1] + 1. / mu[x+1, y, z-1] +
      1. / mu[x, y-1, z-1] + 1. / mu[x+1, y-1, z-1]
    );

    // find xx stress

```

```

xx[x, y, z; t+1] = xx[x, y, z; t] + dth * (
  a * (
    c1 * (u1[x+1, y, z] - u1[x, y, z]) +
    c2 * (u1[x+2, y, z] - u1[x-1, y, z])
  ) +
  b * (
    c1 * (
      v1[x, y, z] - v1[x, y-1, z] +
      w1[x, y, z] - w1[x, y, z-1]
    ) +
    c2 * (
      v1[x, y+1, z] - v1[x, y-2, z] +
      w1[x, y, z+1] - w1[x, y, z-2]
    )
  )
);

// find yy stress
yy[x, y, z; t+1] = yy[x, y, z; t] + dth * (
  a * (
    c1 * (v1[x, y, z] - v1[x, y-1, z]) +
    c2 * (v1[x, y+1, z] - v1[x, y-2, z])
  ) +
  b * (
    c1 * (
      u1[x+1, y, z] - u1[x, y, z] +
      w1[x, y, z] - w1[x, y, z-1]
    ) +
    c2 * (
      u1[x+2, y, z] - u1[x-1, y, z] +
      w1[x, y, z+1] - w1[x, y, z-2]
    )
  )
);

// find zz stress
zz[x, y, z; t+1] = zz[x, y, z; t] + dth * (
  a * (
    c1 * (w1[x, y, z] - w1[x, y, z-1]) +
    c2 * (w1[x, y, z+1] - w1[x, y, z-2])
  ) +
  b * (
    c1 * (
      u1[x+1, y, z] - u1[x, y, z] +
      v1[x, y, z] - v1[x, y-1, z]
    ) +
    c2 * (
      u1[x+2, y, z] - u1[x-1, y, z] +
      v1[x, y+1, z] - v1[x, y-2, z]
    )
  )
);

```

```

    );
  }
}

```

B.5.4 xyzq

```

stencil pmcl3d_xyzq
{
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid mu(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid lam(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid r1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid r2(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid r3(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid xx(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid yy(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid zz(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid u1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid v1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid w1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid qp(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid qs(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid tau(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float param dh, float param dt, float param dth,
    float param nz)
  {
    float c1 = 9./8.;
    float c2 = -1./24.;

    float d = 8. / (
      1. / mu[x, y, z] + 1. / mu[x+1, y, z] +
      1. / mu[x, y-1, z] + 1. / mu[x+1, y-1, z] +
      1. / mu[x, y, z-1] + 1. / mu[x+1, y, z-1] +
      1. / mu[x, y-1, z-1] + 1. / mu[x+1, y-1, z-1]
    );

    float a2 = 2 * d;
    float c = a2 + 8. / (
      1. / lam[x, y, z] + 1. / lam[x+1, y, z] +
      1. / lam[x, y-1, z] + 1. / lam[x+1, y-1, z] +
      1. / lam[x, y, z-1] + 1. / lam[x+1, y, z-1] +
      1. / lam[x, y-1, z-1] + 1. / lam[x+1, y-1, z-1]
    );

    float qpa = 0.125 * (
      qp[x, y, z] + qp[x+1, y, z] +
      qp[x, y-1, z] + qp[x+1, y-1, z] +

```

```

    qp[x, y, z-1] + qp[x+1, y, z-1] +
    qp[x, y-1, z-1] + qp[x+1, y-1, z-1]
);

float qsa = 0.125 * (
    qs[x, y, z] + qs[x+1, y, z] +
    qs[x, y-1, z] + qs[x+1, y-1, z] +
    qs[x, y, z-1] + qs[x+1, y, z-1] +
    qs[x, y-1, z-1] + qs[x+1, y-1, z-1]
);

// (we can't handle indirect grid accesses for the time being)
// float tauu = tau[(coords1 * nxt + this.x) % 2] +
//      2 * ((coords2 * nyt + this.y) % 2) +
//      4 * ((nz + 1 - (coords3 * nzt + this.z)) % 2)];

float vxx = c1 * (u1[x+1, y, z] - u1[x, y, z]) +
    c2 * (u1[x+2, y, z] - u1[x-1, y, z]);
float vyy = c1 * (v1[x, y, z] - v1[x, y-1, z]) +
    c2 * (v1[x, y+1, z] - v1[x, y-2, z]);
float vzz = c1 * (w1[x, y, z] - w1[x, y, z-1]) +
    c2 * (w1[x, y, z+1] - w1[x, y, z-2]);

float a1 = -qpa * c * (vxx + vyy + vzz) / (2. * dh);

//float x1 = tauu / dt + 0.5;
float x1 = tau[x, y, z] / dt + 0.5;
float x2 = x1 - 1; // (tauu/dt)-(1./2.)

// normal stress xx, yy and zz
xx[x, y, z; t+1] = xx[x, y, z; t] + dth * (c * vxx + (c - a2) *
    (vyy + vzz)) + dt * r1[x, y, z; t];
yy[x, y, z; t+1] = yy[x, y, z; t] + dth * (c * vyy + (c - a2) *
    (vxx + vzz)) + dt * r2[x, y, z; t];
zz[x, y, z; t+1] = zz[x, y, z; t] + dth * (c * vzz + (c - a2) *
    (vxx + vyy)) + dt * r3[x, y, z; t];

float hdh = -d * qsa / dh;
r1[x, y, z; t+1] = (x2 * r1[x, y, z; t] - hdh*(vyy + vzz) + a1)/x1;
r2[x, y, z; t+1] = (x2 * r2[x, y, z; t] - hdh*(vxx + vzz) + a1)/x1;
r3[x, y, z; t+1] = (x2 * r3[x, y, z; t] - hdh*(vxx + vyy) + a1)/x1;

xx[x, y, z; t+1] = xx[x, y, z; t+1] + dt * r1[x, y, z; t+1];
yy[x, y, z; t+1] = yy[x, y, z; t+1] + dt * r2[x, y, z; t+1];
zz[x, y, z; t+1] = zz[x, y, z; t+1] + dt * r3[x, y, z; t+1];
}
}

```

