PATUS Quickstart

PATUS v.0.1

Matthias-M. Christen
University of Lugano, Switzerland
matthias.christen@usi.ch

November 7, 2012

1 Introduction

PATUS is a code generation and auto-tuning tool for the class of stencil computations.

Stencil computations are constituent computational building blocks in many scientific and engineering codes. These codes typically achieve a low fraction of peak performance. Computational domains that involve stencils include medical and life science applications, petroleum reservoir simulations, weather and climate modeling, and physics simulations such as fluid dynamics or quantum chromodynamics. Stencil codes may perform tens of thousands of iterations over the spatial domain in order to resolve the time-dependent solution accurately; hence, may require significant core hours on supercomputers. Thus, any performance improvement may lead to a significant reduction in time to solution.

Despite the apparent simplicity of stencil computations and the fact that their computation structure maps well to current hardware architectures, meticulous architecture-specific tuning is still required to elicit a platform's full computational power due to the fact that microarchitectures have grown increasingly complex. Manual architecture-specific tuning requires a significant effort: Not only does it require a deeper understanding of the architecture, but it is also both a time consuming and error-prone process. Although the performance gain might justify the effort, the code usually becomes non-portable and hard to maintain.

The goal of PATUS is to accept an intuitive stencil specification and turn it into architecture-specific, optimized, high-performance code. The stencil specification is formulated in a small domain-specific language, which is explained in section 3.2.

Currently, PATUS supports shared-memory CPU architectures and, as a proof of concept, CUDA-programmable GPUs.

1.1 Copyrights

Cetus. A Source-to-Source Compiler Infrastructure for C Programs. Copyright under the Cetus Artistic License.
http://cetus.ecn.purdue.edu/

JGAP. Java Genetic Algorithms Package. Copyright under the GNU Lesser General Public License.
http://jgap.sourceforge.net/

Time measurement. Adapted from the time measurement in FFTW. Copyright (c) 2003, 2007-8 Matteo Frigo. Copyright (c) 2003, 2007-8 Massachusetts Institute of Technology.

2 Installation

2.1 Prerequisites

PATUS works both on Linux and on Microsoft Windows. It should also work on Mac, but this has not been tested. PATUS assumes that a 64-bit operating system is installed.

There is currently no IDE integration, so the tool has to be started on the command line. However, in the `etc` directory of the PATUS distribution, you can find syntax files for `vi` and `gedit`.

Prior to using PATUS, the following software needs to be installed on your computer:

- **Java 7 or newer.** If you have an older version, please download the JRE 7 or the JDK 7 from http://www.oracle.com/technetwork/java/javase/downloads/index.html and follow the respective installation instructions.
- **Maxima.** Download from http://maxima.sourceforge.net/ and follow the respective installation instructions. For Linux distributions, installation packages are available. E.g., under Ubuntu, type
  ```
  sudo apt-get install maxima
  ```
  in a shell to install the software.
- **gcc.** To compile the code generated by PATUS, you will need a recent version of the GNU C compiler.
  On Linux, if gcc is installed the default version usually should be fine. On Windows, we recommend installing TDM-GCC, which features an easy install of the GCC toolset and MinGW-w64.
  Note that the more "sophisticated" generated code versions require a 64-bit operating system.
- **Make.** To build the generated benchmarking harness and starting the auto-tuner, PATUS generates Makefiles.
  On Linux, if build tools are installed, `make` will be available. On Windows, we recommend installing Make for Windows.

2.2 Installation

2.2.1 Linux

Unzip the `patus-0.1.zip` file in a directory of your choice.

```
unzip patus-0.1.zip
```

Change into the newly created directory.

```
cd Patus
```

Execute the shell script that will set the environment variable PATUS_HOME and modify your PATH to include the Patus directory.
In bash and sh:

```
source util/patusvars.sh
```

In csh and tcsh:

```
source util/patusvars.csh
```

These scripts assume that they are sourced from within the Patus directory.

Now you can run PATUS by typing

```
patus
```

on the command line.

2.2.2 Microsoft Windows

Microsoft Windows is not supported by PATUS as well as Linux, but the `bin` directory of PATUS contains a batch file which can be used to start PATUS.

Unzip the `patus-0.1.zip` file in a directory of your choice.

PATUS can be started by starting

```
patus.bat
```

on the command line.

3 Using PATUS

3.1 A Walkthrough Example

In PATUS, the user specifies the actual stencil computation. Thus, if the tool is to be used to solve differential equations using a finite difference-based discretization method, the discretization needs to be done prior to the implementation in PATUS. In the following, we show briefly how this can be done by means of a simple example.

3.1.1 From a Model to a Stencil Specification

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

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} - c^2 \Delta u &= 0 \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega, \\ u(x, y, z)|_{t=0} &= \sin(2\pi x)\sin(2\pi y)\sin(2\pi z). \end{aligned} \tag{1}$$

Using an explicit finite difference method to discretize the equation both in space and time by means of a fourth-order discretization of the Laplacian

Δ over an equidistant spatial grid with stepsize h and a second-order scheme with time step δt in time, we obtain

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

where Δ_h is the discretized version of the Laplacian:

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

Substituting Eqn. 3 into Eqn. 2, solving Eqn. 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

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

This can now be turned into a PATUS stencil specification almost trivially:

```
stencil wave(
  float grid U(0..x_max-1, 0..y_max-1, 0..z_max-1),
  float param fMin = -1,
  float param fDX = 2 / (x_max-3),
  float param fDT_DX_sq = 0.25)
{
  // do one timestep within the stencil kernel
  iterate while t < 1;

  // define the region on which the stencil is evaluated
  domainsize = (2..x_max-3, 2..y_max-3, 2..z_max-3);

  // define the initial condition (how the data is initialized
  // before the computation)
  // note that 0 <= x < x_max, etc.
  initial {
    U[x,y,z; -1] = sinf(2*π*((x-1)*fDX+fMin)) *
      sinf(2*π*((y-1)*fDX+fMin)) * sinf(2*π*((z-1)*fDX+fMin));
    U[x, y, z; -1 :
      x==0 || y==0 || z==0 || x==x_max-1 || y==y_max-1 || z==z_max-1 ] = 0;
    U[x,y,z; 0] = U[x,y,z; -1];
    U[x,y,z; 1] = 0;
  }

  // define the actual stencil computation
  operation {
    float c1 = 2 - 15/2 * fDT_DX_sq;
    float c2 = 4/3 * fDT_DX_sq;
    float c3 = -1/12 * fDT_DX_sq;

    U[x,y,z; t+1] = c1 * U[x,y,z; t] - U[x,y,z; t-1] +
      c2 * (
```

```
        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]
      ) +
      c3 * (
        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]
      );
  }
}
```

From this stencil specification, PATUS will generate

- a C source code file implementing the stencil computation and
- source files from which a benchmarking harness can be built.

The benchmarking executable is then used by the auto-tuner, which determines a set of architecture-specific parameters, for which the stencil achieves the best performance.

3.1.2 The Specification Explained

- The **stencil** specification defines a stencil "wave," which operates on a grid (called "U") of size [0, x_max-1] $\times$ [0, y_max-1] $\times$ [0,z_max-1]. Note that the size parameters do not have to be defined in the stencil specification. Instead, they will appear as arguments to the generated stencil kernel function, and, in the benchmarking harness, as command line arguments.

  The remaining arguments, "fMin," "fDX," and "fDT_DX_sq," are used for initializing the grid and performing the stencil computation. These parameters will also appear as arguments to the generated C function implementing the computation.

  Optionally, parameters can be initialized with default values (as shown in the listing above). This affects only the benchmarking harness and will fix the values which are passed to the generated stencil kernel.

- The **iterate while** statement defines the number of timesteps to be performed within one stencil kernel call. In the example, one timestep per kernel invocation will be performed. (Timesteps are counted from 0.) The statement can be omitted. Then, the number of timesteps defaults to 1. In the future we will also allow (reduction-based) convergence criteria (e.g., "iterate as long as the residual is larger than some ε").

- The **domainsize** defines the iteration space. While the total size of the grid extends from 0 to $*$_max-1, the stencil is only applied to the points between 2 and $*$_max-3 ($* \in \{x, y, z\}$), i.e., only to the *interior* grid points.

- The **initial** block defines how the grid is initialized, or more mathematically: it defines the initial condition of the (discretized) PDE. The "sinf" function is actually a C function (single precision sine) which will be called (this is the behavior if there is a function which is not known to PATUS). Note that PATUS also defines the π literal with the obvious meaning.

The second statement of the initialization sets all the grid points for which x, y, z are 0 or ∗_max-1 (the condition after colon) to zero. I.e., you can use the set builder notation (with any logical and comparison operators you know from C/C++) to select certain grid points and initialize them.
initial blocks are not mandatory; if no **initial** is provided, PATUS will create an initialization routine anyway, initializing the data with arbitrary values. (This is to ensure the correct data placement on NUMA machines.)

- The **operation**, finally, defines the actual stencil computation. It can contain definitions of constants (as in the listing) or temporary values, and it can also contain more than one stencil expression.

- Optionally, the stencil specification can also contain a **boundaries** block, in which special treatment of boundary regions can be specified. Essentially, within the **boundaries** block, special stencils are defined which are applied to boundary regions. See section 3.2.

3.1.3 Building the Benchmarking Harness

Once the stencil specification is written, PATUS can be run to transform it into C code. Assume the stencil specification was saved in the file `examples/stencils/wave-1.stc`. (This very stencil specification is actually there.) In your shell (in Linux), type

```
cd examples/stencils
patus wave-1.stc
```

or, in Microsoft Windows, on the command line type

```
cd examples\stencils
..\..\bin\patus.bat wave-1.stc
```

This will generate the C code implementing the Wave stencil, using the default architecture, and putting the generated files in the `out` directory. (You can change the output directly by adding the `--outdir=<your-output-dir>` to the PATUS command line.) Change into the output directory,

```
cd out
```

and type

```
make
```

to build the benchmarking harness, which will be used for the auto-tuning process, and can also be used for simple simulations. When the build completes successfully, there is an executable file, `bench` (or `bench.exe` on Windows) in the directory.

If you try to start the benchmarking executable, you will see that it expects some command line arguments:

```
$ ./bench
Wrong number of parameters.  Syntax:
./bench <x_max> <y_max> <z_max> <cb_x> <cb_y> <cb_z> <chunk> <_unroll_p3>
```

The ∗_max correspond to the unspecified domain size variables in the stencil specification. All the other arguments come from the code generator, and it is the auto-tuner's task to find the best values for them.

3.1.4 Auto-Tuning

The Makefile (which was also used to build the benchmarking harness) defines a "tune" target, which starts the auto-tuner. "tune" expects the domain size variables to be specified:

```
make tune x_max=64 y_max=64 z_max=64
```

Type the above command in the shell. The auto-tuner will run the benchmarking executable a number of times, varying the values of the arguments to determine, and it will terminate with a message like

```
Optimal parameter configuration found:
64 64 64 62 20 8 2 0

Timing information for the optimal run:
0.9047935972598529

Program output of the optimal run:
Flops / stencil call:  16
Stencil computations:  1116000
Bytes transferred:     15728640
Total Flops:           17856000
Seconds elapsed:       0.000423
Performance:           42.193517 GFlop/s
Bandwidth utilization: 37.166590 GB/s
1352692.000000
Validation OK.
```

Thus, according to the auto-tuner, on the machine the benchmark was run and for x_max=y_max=z_max=64, the best parameter combination is

```
cb_x=62, cb_y=20, cb_z=8, chunk=2, _unroll_p3=0.
```

PATUS has certain requirements for the grid sizes in the unit stride dimension. If the requirements are not met when you run the executable or the auto-tuner, you may see an error message like

```
Non-native, aligned SIMD type mode requires that (x_max+2) is divisible
by 4 [(x_max+2) = 102].
```

If this is the case, adjust the grid size (here, e.g., set `x_max` to 102 instead of 100) and run again.

3.1.5 Simple Visualization

You can run the benchmarking executable with the arguments provided by the auto-tuner,

```
./bench 64 64 64 62 20 8 2 0
```

If you add the flag `-o` to the command line,

```
./bench 64 64 64 62 20 8 2 0 -o
```

the program will take snapshots of the data grids before and after the execution of the stencil kernel and write them to text files. If you have gnuplot installed on your system, you can run

```
make plot
```

which will use gnuplot to create image files from the data text files.

3.2 Writing Your Own Stencil Specifications

A stencil specification has the following form:

```
stencil stencil-name ( stencil-arguments ) {
    iteration-space
    ( number-of-timesteps )
    operation
    ( boundaries )
    ( initial )
}
```

The order of the iteration space, number of timesteps, operation, boundaries, and initial specifications is not relevant.

3.2.1 Stencil Arguments

Arguments to the stencil can be either **grid**s or **param**s. Two datatypes are supported: **float** (for single precision floating point numbers) and **double** (double precision floating point numbers). Grids are multi-dimensional arrays of data which can be read and written to. Params are application-specific read-only data used in the computation. They can be scalars or fixed-size arrays.

A **grid** can be declared as follows:

```
( const ) ( float | double ) grid grid-name (( lbnd1 .. ubnd1 , lbnd2 .. ubnd2 ...))
( [ albnd1 (.. aubnd1) , albnd2 (.. aubnd2) ...] )
```

All the upper bounds are inclusive.

Examples:

```
float grid A
const double grid B
float grid C(-1 .. size_x)
const double grid D(0..sx, 1..sy, -2..sz+2)[3]
float grid E(min_x..max_x, min_y..max_y)[0..1, -1..3]
```

The grid `A` is declared as a single precision grid with no explicit size declaration. With the size declaration the size of the grid in memory can be defined, which is distinct from the iteration domain, i.e., the specification of the domain to which the calculation is applied. In this case, the grid's size is implied from the **domainsize** in the stencil specification: the grid is assumed to be the smallest grid such that all the grid accesses in the stencil computation are valid.

For the grid `B` also no explicit size is declared. **const** means that the grid is read-only.

Grid `C` is declared as a one-dimensional grid indexable from -1 to size_x. In the actual stencil computation, not all of these points need to be written to (or read), but no bounds checking is performed. If the computation violates the bounds, the compiled code probably will raise segmentation faults.

`D` is declared as an array of 3 three-dimensional grids, `E` as a two-dimensional array of two-dimensional grids. The examples also show that the lower and upper bounds can be arbitrary arithmetic expressions containing variables. Any variables appearing in size declarations will become arguments

to the generated stencil function. The array indices of `E` can be 0 or 1 (first component) and between (and including) -1 and 3 (second component).

A `param` can be declared as follows:

```
( float | double ) param param-name ( [ albnd1 (.. aubnd1) , albnd2 (.. aubnd2) ...] ) (= values )
```

Examples:

```
float param a
double param b[2]
float param c = 1.23
double param d[2..5, 3] = {{0.1,0.2,0.3,0.4}, {1.1,2.2,3.3,4.4}, {5,6,7,8}}
```

params can be simple scalar values or, as **grid**s, (multi-dimensional) arrays of fixed size. Furthermore, default values can be provided. If the **param** is an array, the default values are grouped by curly braces (cf. the initialization of `d`).

3.2.2 Iteration Space

The iteration space, i.e., the domain of the grids on which the stencil computation is executed, is defined by

```
domainsize = ( lbnd1 .. ubnd1 , lbnd2 .. ubnd2 ...);
```

Again, the upper bounds are inclusive. The domain size definition also defines the dimensionality of the stencil.

3.2.3 Number of Timesteps

The number of timesteps to be performed within one call to the generated stencil kernel function is defined by

```
iterate while condition ;
```

Currently, the *condition* must have the form "`t` < *num-timesteps*" where *num-timesteps* is an integer literal. The statement is optional. If it is not specified, by default, one timestep will be performed (this corresponds to "`iterate while t < 1`").

In the future, reduction-based stopping criteria will also be supported, which, e.g., can check for convergence.

3.2.4 The "operation"

In the **operation** block, the actual stencil computation is specified which is executed on the inner points of the grid. An operation can contain one or more statements within curly braces. Statements can be either computations of temporary values or assignments to read-and-write grids defined in the stencil's parameter list. Any of arithmetic operators (`+`, `-`, `*`, `/`, `^`) can be used (for addition, subtraction, multiplication, division, and exponentiation).

Grids are indexed as follows:

```
grid-name [ x ( ( +|- ) offsetx ) , y ( ( +|- ) offsety ) , ...
( ; t ( ( +|- ) time-offset ) )
( ; idx1 , idx2 ...) ]
```

i.e., by spatial coordinates first, then by the temporal index (which must not be present if the grid was declared as `const`), and then by array indices if the grid was declared to be an array.

The offsets to the spatial identifiers (`x`, `y`, `z`, `u`, `v`, `w`, `x`*i* for some integer *i*) and to the spatial identifier `t` have to be compile-time constants. The array indices also have to be compile-time constants (e.g., integer literals).

Examples for accessing the grids defined above:

```
A[x,y+1,z-2; t+2]    // assuming that the stencil dimensionality is 3
B[x+1,y,z]           // B is const => no time component
C[x+1; t-1]
D[x,y,z; 0]          // D is const => no time component, but an array index
E[x,y,z; t+1; 0,-1]  // time component and two-dimensional array index
```

Furthermore, temporary variables (scalars or arrays) can be declared and assigned expressions. Assigning values to arrays works as described for stencil arguments.

The stencil **operation** can also contain reductions (sums or products), which are resolved at compile time. This feature is syntactic sugar which can save some typing. The syntax is

{ idxvar$_1$*=lbnd*$_1$*..ubnd*$_1$ *, idxvar*$_2$*=lbnd*$_2$*..ubnd*$_2$ *... (: constraints) }*
(`sum` | `product` *) (* *expression* *)*

The set defines the index space for the sum or the product, and the expression within the sum or the product the expression, possibly depending on index variables, which is summed up or multiplied for all index variable values. Optionally, separated by a colon, in the index set definition, constraints can be specified, which are logical expressions separated by commas.

The following example shows an **operation** which uses both a temporary array and a sum:

```
operation {
  float c[0..2] = {
    2 - 15/2 * fDT_DX_sq,
    4/3 * fDT_DX_sq,
    -1/12 * fDT_DX_sq
  };

  U[x,y,z; t+1] = c[0]*U[x,y,z; t] - U[x,y,z; t-1] + (
    { i=-1..1, j=-1..1, k=-1..1, r=1..2 : i^2+j^2+k^2==1 } sum(
      c[r] * U[x+r*i,y+r*j,z+r*k; t]
    )
  );
}
```

Here, a temporary array `c` is declared and initialized, and in the second part a compile time reduction is used to sum up the neighboring points of the center point: The three index variables `i`, `j`, `k` take the values -1, 0, and 1, but all combinations are filtered except if $i^2+j^2+k^2=1$. `r` takes the values 1 and 2. Note that the index of `c` and the spatial offsets in the grid `U` within the `sum` are compile-time constants (the variables appear only in the sum's index set).

3.2.5 Boundary Conditions

The **boundaries** block can be used to specify special stencils which are to be evaluated in boundary regions. A boundary point is characterized by

having one or more spatial coordinates in the grid access fixed to discrete values. For instance:

```
boundaries {
  U[0,y,z;t+1] = 0;
  U[x,y,z; t+1 : 0<=x<=1] = 1;
  U[x,0,z;t+1] = U[x,y_max-1,z;t];
}
```

In first statement of the example, all the points with 0 x-coordinates will be set to zero. In the second statement all the points for which the x-coordinate is either 0 or 1 will be set to 1 (the colon notation is the same as mentioned previously for compile-time reductions). "`0<=x<=1`" is a shorthand for "`0<=x && x<=1`". The third statement shows that the right-hand side does not have to be a constant, but can be in fact any arithmetic expression.

If the boundary block is omitted, no boundary handling is implemented in the generated stencil function.

3.2.6 Initial Condition

In the **initial** condition specification, the temporal coordinates of grids are fixed (as opposed to the spatial ones in the boundary specification). All the grids read in the `operation` should be initialized with 0 substituted for `t`. `initial` is optional; if it is not provided, PATUS creates a default initialization with arbitrary values.

Again, as in the **boundary** block, the set-builder notation can be used to select a number of grid points to initialize. E.g., in the last line in the example below, all the points within a cylinder around the origin with radius 30 parallel to the z-axis are selected. No error is raised if points are selected that are not contained in the actual grid. The first statement shows that points can be initialized depending on their grid coordinates.

```
initial {
  U[x,y,z; -1] = sinf(2*π*((x-1)*fDX+fMin)) *
    sinf(2*π*((y-1)*fDX+fMin)) * sinf(2*π*((z-1)*fDX+fMin));
  U[x, y, z; 0] = U[x, y, z; -1];
  U[x, y, z; 1] = 0;

  U[x,y,z;-1 : 10<=x<=20, 5<=y<=10] = 0;
  U[x,y,z; 0 : x^2+y^2 <= 900] = 0;
}
```

3.3 Integrating Into Your Own Application

In the generated code (usually the file implementing the stencils in C is called `kernel.c`), search for the function with the same name as the name of your stencil to determine the signature. This is the function that should be called from within your application code.

See the directory `examples/applications` for examples how to integrate PATUS-generated code into user code. The examples also contain Makefiles that show how the process of generating and automatically tuning stencil kernels and use them to build the final application executable can be facilitated. The examples are described briefly below.

4 Examples

In the **examples** directory some example stencil specifications can be found.

The **stencil** subdirectory contains single stencil specifications, for which code can be generated using the provided Makefile. The Makefile builds the benchmarking harnesses at the same time.

In the **applications** subdirectory, two example applications can be found, which use PATUS to generate the computate kernels. For both applications, Makefiles are provided. Typing

```
make
```

in the shell will build and run the simulations. In the Makefiles, remove the "-mavx" flag from the compiler command line if your CPU does not support AVX.

- **wave** is the simple 3D wave equation solver from 3.1. The file `wave.c` contains both the application code and the embedded PATUS stencil specification, showing how both parts can be used in a single source file.
  A simple visualization (using gnuplot) is provided.
  ```
  make run
  ```
  both starts the program and the visualization.

- **whispering-gallery** is a 2D nano-photonics simulation developed by Max Nolte, who has kindly agreed to publish his source code. The simulation uses two stencils (defined in the external files `fdtdE2.stc` and `fdtdH2.stc`) to calculate Maxwell's equation using the FDTD method, and a third stencil to integrate the energy density (`integrate.stc`).
  Typing
  ```
  make
  ```
  in the shell will first build the simulation and then start it. When the computation completes, the result is visualized with gnuplot and displayed.

5 Current Limitations

In the current release, there are the following limitations to PATUS:

- Only shared memory architectures are supported directly (specifically: shared memory CPU systems and single-GPU setups). However, if you have MPI code that handles communication and synchronization for the distributed case, PATUS-generated code can be used as a replacement for the per-node stencil computation code.

- 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 (Jacobi iteration). In particular, this rules out schemes with special traversal rules such as red-black Gauss-Seidel iterations.

- Boundary handling is not yet optimized in the generated code. Expect performance drops if a boundary specification is included in the stencil specification.

- There is no support yet for temporally blocked schemes.
- Limitation for Fortran interoperability: only one timestep per stencil kernel is supported.
- Limitation for (CUDA-capable) GPUs: the generated code does not do any significant performance optimizations yet. Also, the stencil kernel can only perform one timestep (since CUDA does not allow global synchronization from within a kernel).

Acknowledgments

PATUS has been developed within the High Performance and High Productivity Computing initiative of the Swiss National Supercomputer Center (CSCS).

We also want to acknowledge Max Nolte for testing and using PATUS for his nano-optics simulation project and for kindly agreeing to publish his code.