

---

# HiPerC Documentation

*Release 0*

**Trevor Keller**

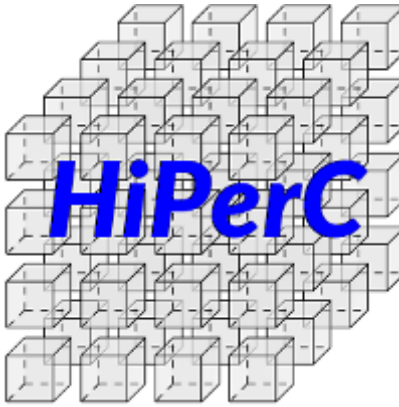
**Sep 20, 2023**



## CONTENTS:

<b>1</b>	<b>High Performance Computing Strategies for Boundary Value Problems</b>	<b>3</b>
1.1	Accelerator Languages	3
1.2	Basic Algorithm	4
1.2.1	Source Code Documentation	5
1.3	Running the Demonstration Programs	5
1.3.1	What to Expect	5
1.4	Reusing the Demonstration Code	7
1.5	Completed Examples	7
1.5.1	Diffusion Equation	7
1.5.2	Spinodal Decomposition	7
1.6	Contributions and Contact	7
1.7	Disclaimer	7
<b>2</b>	<b>API Reference</b>	<b>9</b>
2.1	common-diffusion	9
2.1.1	boundaries.h	9
2.1.2	mesh.h	9
2.1.3	numerics.h	10
2.1.4	output.h	12
2.1.5	timer.h	12
2.1.6	type.h	12
2.2	gpu-cuda-diffusion	13
2.2.1	cuda_kernels.cuh	13
2.3	gpu-opencl-diffusion	14
2.3.1	opencl_data.h	14
2.3.2	opencl_kernels.h	16
<b>3</b>	<b>CPU Specifics</b>	<b>17</b>
3.1	cpu-analytic-diffusion	17
3.1.1	analytic_main.c	17
3.2	cpu-serial-diffusion	17
3.2.1	serial_boundaries.c	17
3.2.2	serial_discretization.c	17
3.3	cpu-openmp-diffusion	18
3.3.1	openmp_boundaries.c	18
3.3.2	openmp_discretization.c	18
3.4	cpu-tbb-diffusion	19
3.4.1	tbb_boundaries.cpp	19
3.4.2	tbb_discretization.cpp	19

<b>4 GPU Specifics</b>	<b>21</b>
4.1 gpu-cuda-diffusion . . . . .	21
4.1.1 cuda_boundaries.cu . . . . .	21
4.1.2 cuda_discretization.cu . . . . .	21
4.2 gpu-openacc-diffusion . . . . .	23
4.2.1 openacc_boundaries.c . . . . .	23
4.2.2 openacc_discretization.c . . . . .	23
4.3 gpu-opencl-diffusion . . . . .	24
4.3.1 opencl_boundaries.c . . . . .	24
4.3.2 opencl_discretization.c . . . . .	24
<b>5 Terms of Use</b>	<b>25</b>
<b>6 Looking for something specific?</b>	<b>27</b>
<b>Index</b>	<b>29</b>





---

# HIGH PERFORMANCE COMPUTING STRATEGIES FOR BOUNDARY VALUE PROBLEMS

---



Ever wonder if a GPU would make your code faster? Fast enough to justify the expense to your manager, adviser, or funding agency? This project can help answer your questions!

The example codes in this repository implement the same basic algorithm using whichever of the mainstream accelerator programming methods apply. Running the code on different parallel hardware configurations — CPU threading, GPU offloading, and CPU coprocessing — provides a benchmark of these tools using common computational materials science workloads. Comparing performance against the serial baseline will help you make informed decisions about which development pathways are appropriate for your scientific computing projects. Note that the examples do not depend on a particular simulation framework: dependencies are kept minimal, and the C functions are kept as simple as possible to enhance readability for study and reusability in other codes. The goal here is to learn how to use accelerators for materials science simulations, not to enhance or promote any particular software package.

## 1.1 Accelerator Languages

There are six mainstream approaches to shared-memory parallel programming, with varying coding complexity and hardware dependencies:

### POSIX threads

MIMD-capable threading for multi-core CPU architectures. Challenging to properly implement, but with ample opportunity to tune performance. Provided by all compilers and compatible with any hardware configuration.

### OpenMP

Loop-level parallelism for multi-core CPU architectures. Simple to implement for SIMD programs, but with little opportunity for performance tuning. Implementation simply requires prefixing target loops with `#pragma` directives. Provided by all compilers and compatible with any hardware configuration.

### Threading Building Blocks

Loop-level parallelism for multi-core CPU architectures using C++. Similar to OpenMP, but requires a wrapper around parallel regions that is more complicated than a one-line `#pragma`. This provides more direct opportunities for performance tuning. Available as an open-source library.

### OpenACC

Loop-level massive parallelism for GPU architectures. Like OpenMP, implementation requires prefixing target code with `#pragma` directives, with little opportunity for performance tuning. Provided in compilers from Cray, PGI, and GNU; depends on a compatible graphics card, drivers, and CUDA or OpenCL library installation.

### CUDA

General-purpose massive parallelism for GPU architectures. Like POSIX threading but for GPUs, provides

low-level capabilities and ample opportunity for performance tuning. Requires a purpose-built compiler (nvcc, gpucc), libraries, and a compatible graphics card or accelerator.

Generically speaking, [OpenMP](#) and [OpenACC](#) provide low barriers for entry into acceleration; [CUDA](#) and [OpenCL](#) require high investments for hardware and compilers, but offer the greatest capabilities for performance and optimization of a specific application. CUDA hardware can be emulated on the CPU using the [MCUDA](#) framework. Proof-of-concept trials on GPU and KNL hardware can be run on Amazon's [EC2](#), Rescale's [ScaleX](#), and equivalent HPC cloud computing platforms. Most of the current generation of research supercomputers contain GPU or KNL accelerator hardware, including Argonne National Labs' [Bebop](#), NERSC [Cori](#), TACC [Stampede2](#), and [XSEDE](#).

## 1.2 Basic Algorithm

Diffusion and phase-field problems depend extensively on the divergence of gradients, *e.g.*

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

When  $D$  is constant, this simplifies to

$$\frac{\partial c}{\partial t} = D \nabla^2 c$$

This equation can be discretized, *e.g.* in 1D:

$$\frac{\Delta c}{\Delta t} \approx D \left[ \frac{c_+ - 2c_o + c_-}{(\Delta x)^2} \right]$$

This discretization is a special case of [convolution](#), wherein a constant kernel of weighting coefficients is applied to an input dataset to produce a transformed output.

1D Laplacian		
1	-2	1

2D Laplacian		
5-point stencil		
0	1	0
1	-4	1
0	1	0

2D Laplacian		
9-point stencil*		
1	4	1
4	-20	4
1	4	1

\* This canonical 9-point (3×3) stencil uses first- and second-nearest neighbors. There is a 9-point (4×4) form that uses first- and third-nearest neighbors, which is also implemented in the source code; it is less efficient than the canonical form.

In addition, computing values for the next timestep given values from the previous timestep and the Laplacian values is a vector-add operation. Accelerators and coprocessors are well-suited to this type of computation. Therefore, to demonstrate the use of this hardware in materials science applications, these examples flow according to the following pseudocode:



```

Start
  Allocate arrays using CPU
  Apply initial conditions to grid marked "old" using CPU
  While elapsed time is less than final time
  Do
    Apply boundary conditions using CPU
    Compute Laplacian using "old" values using accelerator
    Solve for "new" values using "old" and Laplacian values using accelerator
    Increment elapsed time by one timestep
    If elapsed time is an even increment of a specified interval
    Then
      Write an image file to disk
    Endif
  Done
  Write final values to disk in comma-separated value format
  Free arrays
Finish

```

### 1.2.1 Source Code Documentation

You are encouraged to browse the source for this project to see how it works. This project is documented using [Doxygen](#), which can help guide you through the source code layout and intent. This guide is included as [hiperc\\_guide.pdf](#). To build the documentation yourself, with [Doxygen](#), [LaTeX](#), and [Make](#) installed, `cd` into `doc` and run `make`. Then browse the source code to your heart's content.

## 1.3 Running the Demonstration Programs

This repository has a flat structure. Code common to each problem type are lumped together, *e.g.* in `common-diffusion`. The remaining implementation folders have three-part names: `architecture-threading-model`. To compile code for your setup of interest, `cd` into its directory and run `make` (note that this will not work in the `common` folders). If the executable builds, *i.e.* `make` returns without errors, you can `make run` to execute the program and gather timing data. If you wish to attempt building or running all the example codes, execute `make` or `make run` from this top-level directory: it will recursively call the corresponding `make` command in every sub-directory.

### 1.3.1 What to Expect

As the solver marches along, an indicator will display the start time, progress, and runtime in your terminal, *e.g.*

```
Fri Aug 18 21:05:47 2017 [••••••••••••••••••••] 0h: 7m:15s
```

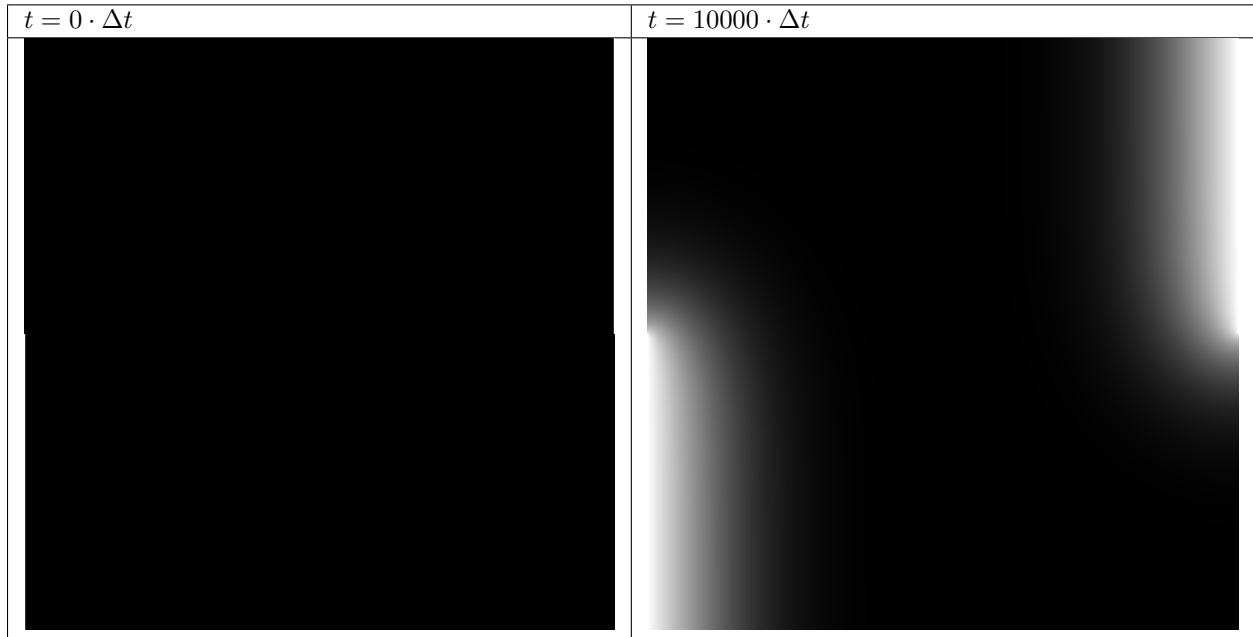
If the progress bar is not moving, or to check that the machine is working hard, use a hardware monitoring tool. Here is a brief, definitely not comprehensive list of options:

- **CPU**: any system monitor provided by your operating system will work. Look for CPU utilization greater than 100%, but moderate memory consumption. On GNU/Linux systems, [htop](#) provides a rich interface to running processes in the terminal, which is helpful if you're running remotely.
- **GPU**: use a GPU monitor designed for your hardware. Some options include [nvidia-smi](#), [radeontop](#), and [intel\\_gpu\\_top](#).
- **KNL**: the same monitor used for the CPU should also report load on the Knights Landing processor.

As it runs, the code will write a series of PNG image files (`diffusion.00?0000.png`) in the same directory as the running executable resides; at the end, it will write the final values to `diffusion.0100000.csv`. It will also write a summary file, `runlog.csv`, containing the following columns:

- **iter**: number of completed iterations
- **sim\_time**: elapsed simulation time (with  $\Delta t = 1$ , the first two columns are equal)
- **wrss**: weighted sum-of-squares residual between the numerical values and analytical solution
- **conv\_time**: cumulative real time spent computing the Laplacian (convolution)
- **step\_time**: cumulative real time spent updating the composition (time-stepping)
- **IO\_time**: cumulative real time spent writing PNG files
- **soln\_time**: cumulative real time spent computing the analytical solution
- **run\_time**: elapsed real time

At timestep 10,000 the expected `wrss=0.002895` (0.2%) using the 5-point stencil; the rendered initial and final images should look like these (grayscale, 0 is black and 1 is white):



The boundary conditions are fixed values of 1 along the lower-left half and upper-right half walls, no flux everywhere else, with an initial value of 0 everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

If your compiler returns warnings or errors, your simulation results do not look like this, or if `wrss` at  $t = 10000 \cdot \Delta t$  is greater than 0.5% or so, something may be wrong with the installation, hardware, or implementation. Please [file an issue](#) and share what happened. You probably found a bug!

Note that a flat field of zeros at  $t = 10000 \cdot \Delta t$ , about as wrong an answer as possible, gives `wrss=0.07493` (7.5%) relative to the analytical solution. Small differences in `wrss` indicate large errors.

## 1.4 Reusing the Demonstration Code

The flat file structure is intended to make it easy for you to extract code for modification and reuse in your research code. To do so, copy the three-part folder corresponding to your setup of interest, *e.g.* `gpu-cuda-diffusion`, to another location (outside this repository). Then copy the contents of the common folder it depends upon, *e.g.* `common-diffusion`, into the new folder location. Finally, edit the `Makefile` within the new folder to remove references to the old common folder. This should centralize everything you need to remix and get started in the new folder.

## 1.5 Completed Examples

### 1.5.1 Diffusion Equation

- CPU - analytical - serial - OpenMP - TBB
- GPU - CUDA - OpenAcc - OpenCL

### 1.5.2 Spinodal Decomposition

- CPU - OpenMP
- GPU - CUDA

## 1.6 Contributions and Contact

Forks of this git repository are encouraged, and pull requests providing patches or implementations are more than welcome. Please review the [Contributing Guidelines](#). Questions, concerns, and feedback regarding this source code should be addressed to the community on [Gitter](#), or directly to the developer ([Trevor Keller](#)).

## 1.7 Disclaimer

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the [National Institute of Standards and Technology](#), nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.



## API REFERENCE

### 2.1 common-diffusion

#### 2.1.1 boundaries.h

Declaration of boundary condition function prototypes.

##### Functions

void **apply\_initial\_conditions**(*fp\_t* \*\*conc\_old, const int nx, const int ny, const int nm)

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

void **apply\_boundary\_conditions**(*fp\_t* \*\*conc\_old, const int nx, const int ny, const int nm)

Set fixed value ( $c_{hi}$ ) along left and bottom, zero-flux elsewhere.

#### 2.1.2 mesh.h

Declaration of mesh function prototypes for diffusion benchmarks.

##### Functions

void **make\_arrays**(*fp\_t* \*\*\*conc\_old, *fp\_t* \*\*\*conc\_new, *fp\_t* \*\*\*conc\_lap, *fp\_t* \*\*\*mask\_lap, const int nx, const int ny, const int nm)

Allocate 2D arrays to store scalar composition values.

Arrays are allocated as 1D arrays, then 2D pointer arrays are mapped over the top. This facilitates use of either 1D or 2D data access, depending on whether the task is spatially dependent or not.

void **free\_arrays**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_new, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*mask\_lap)

Free dynamically allocated memory.

void **swap\_pointers**(*fp\_t* \*\*\*conc\_old, *fp\_t* \*\*\*conc\_new)

Swap pointers to 2D arrays.

Rather than copy data from *fp\_t* \*\* *conc\_old* into *fp\_t* \*\* *conc\_new*, an expensive operation, simply trade the top-most pointers. New becomes old, old becomes new, with no data lost and in almost no time.

void **swap\_pointers\_1D**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_new)

Swap pointers to data underlying 1D arrays.

Rather than copy data from *fp\_t* \* *conc\_old*[0] into *fp\_t* \* *conc\_new*[0], an expensive operation, simply trade the top-most pointers. New becomes old, old becomes new, with no data lost and in almost no time.

### 2.1.3 numerics.h

Declaration of Laplacian operator and analytical solution functions.

#### Defines

**MAX\_MASK\_W**

Maximum width of the convolution mask (Laplacian stencil) array.

**MAX\_MASK\_H**

Maximum height of the convolution mask (Laplacian stencil) array.

#### Functions

void **set\_mask**(const *fp\_t* dx, const *fp\_t* dy, const int code, *fp\_t* \*\*mask\_lap, const int nm)

Specify which stencil (mask) to use for the Laplacian (convolution)

The mask corresponding to the numerical code will be applied. The suggested encoding is mask width as the ones digit and value count as the tens digit, *e.g.* 53 specifies *five\_point\_Laplacian\_stencil()*, while 93 specifies *nine\_point\_Laplacian\_stencil()*.

To add your own mask (stencil), add a case to this function with your chosen numerical encoding, then specify that code in the input parameters file (params.txt by default). Note that, for a Laplacian stencil, the sum of the coefficients must equal zero and *nm* must be an odd integer.

If your stencil is larger than  $5 \times 5$ , you must increase the values defined by *MAX\_MASK\_W* and *MAX\_MASK\_H*.

void **five\_point\_Laplacian\_stencil**(const *fp\_t* dx, const *fp\_t* dy, *fp\_t* \*\*mask\_lap, const int nm)

Write 5-point Laplacian stencil into convolution mask.

$3 \times 3$  mask, 5 values, truncation error  $\mathcal{O}(\Delta x^2)$

void **nine\_point\_Laplacian\_stencil**(const *fp\_t* dx, const *fp\_t* dy, *fp\_t* \*\*mask\_lap, const int nm)

Write 9-point Laplacian stencil into convolution mask.

$3 \times 3$  mask, 9 values, truncation error  $\mathcal{O}(\Delta x^4)$

void **slow\_nine\_point\_Laplacian\_stencil**(const *fp\_t* dx, const *fp\_t* dy, *fp\_t* \*\*mask\_lap, const int nm)

Write 9-point Laplacian stencil into convolution mask.

$5 \times 5$  mask, 9 values, truncation error  $\mathcal{O}(\Delta x^4)$

Provided for testing and demonstration of scalability, only: as the name indicates, this 9-point stencil is computationally more expensive than the  $3 \times 3$  version. If your code requires  $\mathcal{O}(\Delta x^4)$  accuracy, please use `nine_point_Laplacian_stencil()`.

void **compute\_convolution**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*mask\_lap, const int nx, const int ny, const int nm)

Perform the convolution of the mask matrix with the composition matrix.

If the convolution mask is the Laplacian stencil, the convolution evaluates the discrete Laplacian of the composition field. Other masks are possible, for example the Sobel filters for edge detection. This function is general purpose: as long as the dimensions *nx*, *ny*, and *nm* are properly specified, the convolution will be correctly computed.

void **update\_composition**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*conc\_new, const int nx, const int ny, const int nm, const *fp\_t* D, const *fp\_t* dt)

Update composition field using explicit Euler discretization (forward-time centered space)

*fp\_t* **euclidean\_distance**(const *fp\_t* ax, const *fp\_t* ay, const *fp\_t* bx, const *fp\_t* by)

Compute Euclidean distance between two points, *a* and *b*.

*fp\_t* **manhattan\_distance**(const *fp\_t* ax, const *fp\_t* ay, const *fp\_t* bx, const *fp\_t* by)

Compute Manhattan distance between two points, *a* and *b*.

*fp\_t* **distance\_point\_to\_segment**(const *fp\_t* ax, const *fp\_t* ay, const *fp\_t* bx, const *fp\_t* by, const *fp\_t* px, const *fp\_t* py)

Compute minimum distance from point *p* to a line segment bounded by points *a* and *b*.

This function computes the projection of *p* onto *ab*, limiting the projected range to [0, 1] to handle projections that fall outside of *ab*. Implemented after Grumdrig on Stackoverflow, <https://stackoverflow.com/a/1501725>.

void **analytical\_value**(const *fp\_t* x, const *fp\_t* t, const *fp\_t* D, *fp\_t* \*c)

Analytical solution of the diffusion equation for a carburizing process.

For 1D diffusion through a semi-infinite domain with initial and far-field composition  $c_\infty$  and boundary value  $c(x=0, t) = c_0$  with constant diffusivity *D*, the solution to Fick's second law is

$$c(x, t) = c_0 - (c_0 - c_\infty) \operatorname{erf} \left( \frac{x}{\sqrt{4Dt}} \right)$$

which reduces, when  $c_\infty = 0$ , to

$$c(x, t) = c_0 \left[ 1 - \operatorname{erf} \left( \frac{x}{\sqrt{4Dt}} \right) \right].$$

void **check\_solution**(*fp\_t* \*\*conc\_new, *fp\_t* \*\*conc\_lap, const int nx, const int ny, const *fp\_t* dx, const *fp\_t* dy, const int nm, const *fp\_t* elapsed, const *fp\_t* D, *fp\_t* \*rss)

Compare numerical and analytical solutions of the diffusion equation.

Overwrites *conc\_lap*, into which the point-wise RSS is written. Normalized RSS is then computed as the sum of the point-wise values.

#### Returns

Residual sum of squares (RSS), normalized to the domain size.

## 2.1.4 output.h

Declaration of output function prototypes for diffusion benchmarks.

### Functions

void **param\_parser**(int argc, char \*argv[], int \*bx, int \*by, int \*checks, int \*code, *fp\_t* \*D, *fp\_t* \*dx, *fp\_t* \*dy, *fp\_t* \*linStab, int \*nm, int \*nx, int \*ny, int \*steps)

Read parameters from file specified on the command line.

void **print\_progress**(const int step, const int steps)

Prints timestamps and a 20-point progress bar to stdout.

Call inside the timestepping loop, near the top, e.g.

```
for (int step=0; step<steps; step++) {  
    print_progress(step, steps);  
    take_a_step();  
    elapsed += dt;  
}
```

void **write\_csv**(*fp\_t* \*\*conc, const int nx, const int ny, const *fp\_t* dx, const *fp\_t* dy, const int step)

Writes scalar composition field to diffusion.?????.csv.

void **write\_png**(*fp\_t* \*\*conc, const int nx, const int ny, const int step)

Writes scalar composition field to diffusion.?????.png.

## 2.1.5 timer.h

Declaration of timer function prototypes for diffusion benchmarks.

### Functions

void **StartTimer**()

Set CPU frequency and begin timing.

double **GetTimer**()

Return elapsed time in seconds.

## 2.1.6 type.h

Definition of scalar data type and Doxygen diffusion group.



## Typedefs

typedef double **fp\_t**

Specify the basic data type to achieve the desired accuracy in floating-point arithmetic: float for single-precision, double for double-precision. This choice propagates throughout the code, and may significantly affect runtime on GPU hardware.

struct **Stopwatch**

*#include <type.h>* Container for timing data

## Public Members

*fp\_t* **conv**

Cumulative time executing *compute\_convolution()*

*fp\_t* **step**

Cumulative time executing *solve\_diffusion\_equation()*

*fp\_t* **file**

Cumulative time executing *write\_csv()* and *write\_png()*

*fp\_t* **soln**

Cumulative time executing *check\_solution()*

## 2.2 gpu-cuda-diffusion

### 2.2.1 cuda\_kernels.cuh

Declaration of functions to execute on the GPU (CUDA kernels)

## Functions

void **boundary\_kernel**(*fp\_t* \*conc, const int nx, const int ny, const int nm)

Boundary condition kernel for execution on the GPU.

This function accesses 1D data rather than the 2D array representation of the scalar composition field

Boundary condition kernel for execution on the GPU.

Boundary condition kernel for execution on the GPU

This function accesses 1D data rather than the 2D array representation of the scalar composition field

void **convolution\_kernel**(*fp\_t* \*conc\_old, *fp\_t* \*conc\_lap, const int nx, const int ny, const int nm)

Tiled convolution algorithm for execution on the GPU.

This function accesses 1D data rather than the 2D array representation of the scalar composition field, mapping into 2D tiles on the GPU with halo cells before computing the convolution.

Note:

- The source matrix (*conc\_old*) and destination matrix (*conc\_lap*) must be identical in size
- One CUDA core operates on one array index: there is no nested loop over matrix elements
- The halo ( $nm/2$  perimeter cells) in *conc\_lap* are unallocated garbage
- The same cells in *conc\_old* are boundary values, and contribute to the convolution
- *conc\_tile* is the shared tile of input data, accessible by all threads in this block

void **diffusion\_kernel**(*fp\_t* \*conc\_old, *fp\_t* \*conc\_new, *fp\_t* \*conc\_lap, const int nx, const int ny, const int nm, const *fp\_t* D, const *fp\_t* dt)

Vector addition algorithm for execution on the GPU.

This function accesses 1D data rather than the 2D array representation of the scalar composition field. Memory allocation, data transfer, and array release are handled in `cuda_init()`, with arrays on the host and device managed through `CudaData`, which is a struct passed by reference into the function. In this way, device kernels can be called in isolation without incurring the cost of data transfers and with reduced risk of memory leaks.

## Variables

*fp\_t* **d\_mask**[5 \* 5]

Convolution mask array on the GPU, allocated in protected memory.

## 2.3 gpu-openc1-diffusion

### 2.3.1 openc1\_data.h

Declaration of OpenCL data container.

## Functions

void **report\_error**(cl\_int error, const char \*message)

Report error code when status is not CL\_SUCCESS.

Refer to <https://streamhpc.com/blog/2013-04-28/openc1-error-codes/> for help interpreting error codes.

void **build\_program**(const char \*filename, cl\_context \*context, cl\_device\_id \*gpu, cl\_program \*program, cl\_int \*status)

Build kernel program from text input.

Source follows the OpenCL Programming Book, <https://www.fixstars.com/en/openc1/book/OpenCLProgrammingBook/calling-the-kernel/>

void **init\_openc1**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*mask\_lap, const int nx, const int ny, const int nm, struct *OpenCLData* \*dev)

Initialize OpenCL device memory before marching.

void **device\_boundaries**(struct *OpenCLData* \*dev, const int flip, const int nx, const int ny, const int nm, const int bx, const int by)

Apply boundary conditions on OpenCL device.

void **device\_convolution**(struct *OpenCLData* \*dev, const int flip, const int nx, const int ny, const int nm, const int bx, const int by)

Compute convolution on OpenCL device.

void **device\_diffusion**(struct *OpenCLData* \*dev, const int flip, const int nx, const int ny, const int nm, const int bx, const int by, const *fp\_t* D, const *fp\_t* dt)

Solve diffusion equation on OpenCL device.

void **read\_out\_result**(struct *OpenCLData* \*dev, const int flip, *fp\_t* \*\*conc\_new, const int nx, const int ny)

Copy data out of OpenCL device.

void **free\_opencl**(struct *OpenCLData* \*dev)

Free OpenCL device memory after marching.

struct **OpenCLData**

*#include <opencl\_data.h>* Container for GPU array pointers and parameters.

From the [OpenCL v1.2 spec](#):

- A *Context* is the environment within which the kernels execute and the domain in which synchronization and memory management is defined. The context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects.
- A *Program Object* encapsulates the following information:
  - A reference to an associated context.
  - A program source or binary.
  - The latest successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log.
  - The number of kernel objects currently attached.
- A *Kernel Object* encapsulates a specific `__kernel` function declared in a program and the argument values to be used when executing this `__kernel` function.

## Public Members

cl\_context **context**

OpenCL interface to the GPU, hardware and software

cl\_mem **conc\_old**

Copy of old composition field on the GPU

cl\_mem **conc\_new**

Copy of new composition field on the GPU

cl\_mem **conc\_lap**

Copy of Laplacian field on the GPU

cl\_mem **mask**

Copy of Laplacian mask on the GPU

cl\_program **boundary\_program**

Boundary program source for JIT compilation on the GPU

cl\_program **convolution\_program**

Convolution program source for JIT compilation on the GPU

cl\_program **diffusion\_program**

Timestepping program source for JIT compilation on the GPU

cl\_kernel **boundary\_kernel**

Boundary program executable for the GPU

cl\_kernel **convolution\_kernel**

Convolution program executable for the GPU

cl\_kernel **diffusion\_kernel**

Timestepping program executable for the GPU

cl\_command\_queue **commandQueue**

Queue for submitting OpenCL jobs to the GPU

### 2.3.2 opengl\_kernels.h

<b>Warning:</b> doxygenfile: Cannot find file “opengl_kernels.h”
--

#### Looking for something specific?

- [genindex](#)
- [search](#)

## CPU SPECIFICS

### 3.1 cpu-analytic-diffusion

#### 3.1.1 analytic\_main.c

<b>Warning:</b> doxygenfile: Cannot find file “analytic_main.c”
---

### 3.2 cpu-serial-diffusion

#### 3.2.1 serial\_boundaries.c

Implementation of boundary condition functions without threading.

##### Functions

void **apply\_initial\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

void **apply\_boundary\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Set fixed value ( $c_{hi}$ ) along left and bottom, zero-flux elsewhere.

#### 3.2.2 serial\_discretization.c

Implementation of boundary condition functions without threading.

## Functions

void **compute\_convolution**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*mask\_lap, const int nx, const int ny, const int nm)

Perform the convolution of the mask matrix with the composition matrix.

If the convolution mask is the Laplacian stencil, the convolution evaluates the discrete Laplacian of the composition field. Other masks are possible, for example the Sobel filters for edge detection. This function is general purpose: as long as the dimensions *nx*, *ny*, and *nm* are properly specified, the convolution will be correctly computed.

void **update\_composition**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*conc\_new, const int nx, const int ny, const int nm, const *fp\_t* D, const *fp\_t* dt)

Update composition field using explicit Euler discretization (forward-time centered space)

## 3.3 cpu-openmp-diffusion

### 3.3.1 openmp\_boundaries.c

Implementation of boundary condition functions with OpenMP threading.

## Functions

void **apply\_initial\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

void **apply\_boundary\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Set fixed value ( $c_{hi}$ ) along left and bottom, zero-flux elsewhere.

### 3.3.2 openmp\_discretization.c

Implementation of boundary condition functions with OpenMP threading.

## Functions

void **compute\_convolution**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*mask\_lap, const int nx, const int ny, const int nm)

Perform the convolution of the mask matrix with the composition matrix.

If the convolution mask is the Laplacian stencil, the convolution evaluates the discrete Laplacian of the composition field. Other masks are possible, for example the Sobel filters for edge detection. This function is general purpose: as long as the dimensions *nx*, *ny*, and *nm* are properly specified, the convolution will be correctly computed.

```
void update_composition(fp_t **conc_old, fp_t **conc_lap, fp_t **conc_new, const int nx, const int ny, const int nm, const fp_t D, const fp_t dt)
```

Update composition field using explicit Euler discretization (forward-time centered space)

## 3.4 cpu-tbb-diffusion

### 3.4.1 tbb\_boundaries.cpp

Implementation of boundary condition functions with TBB threading.

#### Functions

```
void apply_initial_conditions(fp_t **conc, const int nx, const int ny, const int nm)
```

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

```
void apply_boundary_conditions(fp_t **conc, const int nx, const int ny, const int nm)
```

Set fixed value ( $c_{hi}$ ) along left and bottom, zero-flux elsewhere.

### 3.4.2 tbb\_discretization.cpp

Implementation of boundary condition functions with TBB threading.

#### Functions

```
void compute_convolution(fp_t **conc_old, fp_t **conc_lap, fp_t **mask_lap, const int nx, const int ny, const int nm)
```

Perform the convolution of the mask matrix with the composition matrix.

If the convolution mask is the Laplacian stencil, the convolution evaluates the discrete Laplacian of the composition field. Other masks are possible, for example the Sobel filters for edge detection. This function is general purpose: as long as the dimensions  $nx$ ,  $ny$ , and  $nm$  are properly specified, the convolution will be correctly computed.

```
void update_composition(fp_t **conc_old, fp_t **conc_lap, fp_t **conc_new, const int nx, const int ny, const int nm, const fp_t D, const fp_t dt)
```

Update composition field using explicit Euler discretization (forward-time centered space)

```
void check_solution_lambda(fp_t **conc_new, fp_t **conc_lap, const int nx, const int ny, const fp_t dx, const fp_t dy, const int nm, const fp_t elapsed, const fp_t D, fp_t *rss)
```

### Looking for something specific?

- [genindex](#)
- [search](#)



## GPU SPECIFICS

### 4.1 gpu-cuda-diffusion

#### 4.1.1 cuda\_boundaries.cu

Implementation of boundary condition functions with OpenMP threading.

##### Functions

void **apply\_initial\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

void **boundary\_kernel**(*fp\_t* \*d\_conc, const int nx, const int ny, const int nm)

Enable double-precision floats.

Boundary condition kernel for execution on the GPU.

Boundary condition kernel for execution on the GPU

This function accesses 1D data rather than the 2D array representation of the scalar composition field

#### 4.1.2 cuda\_discretization.cu

Implementation of boundary condition functions with CUDA acceleration.

##### Functions

void **convolution\_kernel**(*fp\_t* \*d\_conc\_old, *fp\_t* \*d\_conc\_lap, const int nx, const int ny, const int nm)

Tiled convolution algorithm for execution on the GPU.

This function accesses 1D data rather than the 2D array representation of the scalar composition field, mapping into 2D tiles on the GPU with halo cells before computing the convolution.

Note:

- The source matrix (*conc\_old*) and destination matrix (*conc\_lap*) must be identical in size

- One CUDA core operates on one array index: there is no nested loop over matrix elements
- The halo ( $nm/2$  perimeter cells) in *conc\_lap* are unallocated garbage
- The same cells in *conc\_old* are boundary values, and contribute to the convolution
- *conc\_tile* is the shared tile of input data, accessible by all threads in this block

void **diffusion\_kernel**(*fp\_t* \*d\_conc\_old, *fp\_t* \*d\_conc\_new, *fp\_t* \*d\_conc\_lap, const int nx, const int ny, const int nm, const *fp\_t* D, const *fp\_t* dt)

Vector addition algorithm for execution on the GPU.

This function accesses 1D data rather than the 2D array representation of the scalar composition field. Memory allocation, data transfer, and array release are handled in `cuda_init()`, with arrays on the host and device managed through `CudaData`, which is a struct passed by reference into the function. In this way, device kernels can be called in isolation without incurring the cost of data transfers and with reduced risk of memory leaks.

void **device\_boundaries**(*fp\_t* \*conc, const int nx, const int ny, const int nm, const int bx, const int by)

Apply boundary conditions on device.

void **device\_convolution**(*fp\_t* \*conc\_old, *fp\_t* \*conc\_lap, const int nx, const int ny, const int nm, const int bx, const int by)

Compute convolution on device.

void **device\_composition**(*fp\_t* \*conc\_old, *fp\_t* \*conc\_new, *fp\_t* \*conc\_lap, const int nx, const int ny, const int nm, const int bx, const int by, const *fp\_t* D, const *fp\_t* dt)

Step diffusion equation on device.

void **read\_out\_result**(*fp\_t* \*\*conc, *fp\_t* \*d\_conc, const int nx, const int ny)

Read data from device.

void **compute\_convolution**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*mask\_lap, const int bx, const int by, const int nm, const int nx, const int ny)

Reference showing how to invoke the convolution kernel.

A stand-alone function like this incurs the cost of host-to-device data transfer each time it is called: it is a teaching tool, not reusable code. It is the basis for `cuda_diffusion_solver()`, which achieves much better performance by bundling CUDA kernels together and intelligently managing data transfers between the host (CPU) and device (GPU).

void **cuda\_diffusion\_solver**(struct `CudaData` \*dev, *fp\_t* \*\*conc\_new, const int bx, const int by, const int nm, const int nx, const int ny, const *fp\_t* D, const *fp\_t* dt, struct *Stopwatch* \*sw)

Reference optimized code for solving the diffusion equation.

Solve diffusion equation on the GPU.

Compare `cuda_diffusion_solver()`: it accomplishes the same result, but without the memory allocation, data transfer, and array release. These are handled in `cuda_init()`, with arrays on the host and device managed through `CudaData`, which is a struct passed by reference into the function. In this way, device kernels can be called in isolation without incurring the cost of data transfers and with reduced risk of memory leaks.

## Variables

*fp\_t* **d\_mask**[5 \* 5]

Convolution mask array on the GPU, allocated in protected memory.

## 4.2 gpu-openacc-diffusion

### 4.2.1 openacc\_boundaries.c

Implementation of boundary condition functions with OpenMP threading.

#### Functions

void **apply\_initial\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

void **boundary\_kernel** (*fp\_t* \*\*\_\_restrict\_\_ conc, const int nx, const int ny, const int nm)

void **apply\_boundary\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Set fixed value ( $c_{hi}$ ) along left and bottom, zero-flux elsewhere.

### 4.2.2 openacc\_discretization.c

Implementation of boundary condition functions with OpenACC threading.

#### Functions

void **convolution\_kernel**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_lap, *fp\_t* \*\*mask\_lap, const int nx, const int ny, const int nm)

Tiled convolution algorithm for execution on the GPU.

void **diffusion\_kernel**(*fp\_t* \*\*conc\_old, *fp\_t* \*\*conc\_new, *fp\_t* \*\*conc\_lap, const int nx, const int ny, const int nm, const *fp\_t* D, const *fp\_t* dt)

Vector addition algorithm for execution on the GPU.

## 4.3 gpu-openc1-diffusion

### 4.3.1 openc1\_boundaries.c

Implementation of boundary condition functions with OpenCL acceleration.

#### Functions

void **apply\_initial\_conditions**(*fp\_t* \*\*conc, const int nx, const int ny, const int nm)

Initialize flat composition field with fixed boundary conditions.

The boundary conditions are fixed values of  $c_{hi}$  along the lower-left half and upper-right half walls, no flux everywhere else, with an initial values of  $c_{lo}$  everywhere. These conditions represent a carburizing process, with partial exposure (rather than the entire left and right walls) to produce an inhomogeneous workload and highlight numerical errors at the boundaries.

### 4.3.2 openc1\_discretization.c

Implementation of boundary condition functions with OpenCL acceleration.

#### Functions

void **device\_boundaries**(struct *OpenCLData* \*dev, const int flip, const int nx, const int ny, const int nm, const int bx, const int by)

Apply boundary conditions on OpenCL device.

void **device\_convolution**(struct *OpenCLData* \*dev, const int flip, const int nx, const int ny, const int nm, const int bx, const int by)

Compute convolution on OpenCL device.

void **device\_diffusion**(struct *OpenCLData* \*dev, const int flip, const int nx, const int ny, const int nm, const int bx, const int by, const *fp\_t* D, const *fp\_t* dt)

Solve diffusion equation on OpenCL device.

void **read\_out\_result**(struct *OpenCLData* \*dev, const int flip, *fp\_t* \*\*conc, const int nx, const int ny)

Copy data out of OpenCL device.

#### Looking for something specific?

- `genindex`
- `search`

## **TERMS OF USE**

NIST-developed software is provided by NIST as a public service. You may use, copy, and distribute copies of the software in any medium, provided that you keep intact this entire notice. You may improve, modify, and create derivative works of the software or any portion of the software, and you may copy and distribute such modifications or works. Modified works should carry a notice stating that you changed the software and should note the date and nature of any such change. Please explicitly acknowledge the National Institute of Standards and Technology as the source of the software.

NIST-developed software is expressly provided “AS IS.” NIST MAKES NO WARRANTY OF ANY KIND, EXPRESS, IMPLIED, IN FACT, OR ARISING BY OPERATION OF LAW, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND DATA ACCURACY. NIST NEITHER REPRESENTS NOR WARRANTS THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT ANY DEFECTS WILL BE CORRECTED. NIST DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE SOFTWARE OR THE RESULTS THEREOF, INCLUDING BUT NOT LIMITED TO THE CORRECTNESS, ACCURACY, RELIABILITY, OR USEFULNESS OF THE SOFTWARE.

You are solely responsible for determining the appropriateness of using and distributing the software and you assume all risks associated with its use, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and the unavailability or interruption of operation. This software is not intended to be used in any situation where a failure could cause risk of injury or damage to property. The software developed by NIST employees is not subject to copyright protection within the United States.



## LOOKING FOR SOMETHING SPECIFIC?

- `genindex`
- `search`





## A

analytical\_value (C++ function), 11  
 apply\_boundary\_conditions (C++ function), 9, 17–19, 23  
 apply\_initial\_conditions (C++ function), 9, 17–19, 21, 23, 24

## B

boundary\_kernel (C++ function), 13, 21  
 build\_program (C++ function), 14

## C

check\_solution (C++ function), 11  
 check\_solution\_lambda (C++ function), 19  
 compute\_convolution (C++ function), 11, 18, 19, 22  
 convolution\_kernel (C++ function), 13, 21, 23  
 cuda\_diffusion\_solver (C++ function), 22

## D

d\_mask (C++ member), 14, 23  
 device\_boundaries (C++ function), 14, 22, 24  
 device\_composition (C++ function), 22  
 device\_convolution (C++ function), 14, 22, 24  
 device\_diffusion (C++ function), 15, 24  
 diffusion\_kernel (C++ function), 14, 22, 23  
 distance\_point\_to\_segment (C++ function), 11

## E

euclidean\_distance (C++ function), 11

## F

five\_point\_Laplacian\_stencil (C++ function), 10  
 fp\_t (C++ type), 13  
 free\_arrays (C++ function), 9  
 free\_opengl (C++ function), 15

## G

GetTimer (C++ function), 12

## I

init\_opengl (C++ function), 14

## M

make\_arrays (C++ function), 9  
 manhattan\_distance (C++ function), 11  
 MAX\_MASK\_H (C macro), 10  
 MAX\_MASK\_W (C macro), 10

## N

nine\_point\_Laplacian\_stencil (C++ function), 10

## O

OpenCLData (C++ struct), 15  
 OpenCLData::boundary\_kernel (C++ member), 16  
 OpenCLData::boundary\_program (C++ member), 15  
 OpenCLData::commandQueue (C++ member), 16  
 OpenCLData::conc\_lap (C++ member), 15  
 OpenCLData::conc\_new (C++ member), 15  
 OpenCLData::conc\_old (C++ member), 15  
 OpenCLData::context (C++ member), 15  
 OpenCLData::convolution\_kernel (C++ member), 16  
 OpenCLData::convolution\_program (C++ member), 16  
 OpenCLData::diffusion\_kernel (C++ member), 16  
 OpenCLData::diffusion\_program (C++ member), 16  
 OpenCLData::mask (C++ member), 15

## P

param\_parser (C++ function), 12  
 print\_progress (C++ function), 12

## R

read\_out\_result (C++ function), 15, 22, 24  
 report\_error (C++ function), 14

## S

set\_mask (C++ function), 10  
 slow\_nine\_point\_Laplacian\_stencil (C++ function), 10  
 StartTimer (C++ function), 12  
 Stopwatch (C++ struct), 13  
 Stopwatch::conv (C++ member), 13

Stopwatch::file (*C++ member*), [13](#)  
Stopwatch::soln (*C++ member*), [13](#)  
Stopwatch::step (*C++ member*), [13](#)  
swap\_pointers (*C++ function*), [9](#)  
swap\_pointers\_1D (*C++ function*), [10](#)

## U

update\_composition (*C++ function*), [11](#), [18](#), [19](#)

## W

write\_csv (*C++ function*), [12](#)  
write\_png (*C++ function*), [12](#)