Portable Parallelization with the Bohrium Runtime System

Mads R. B. Kristensen

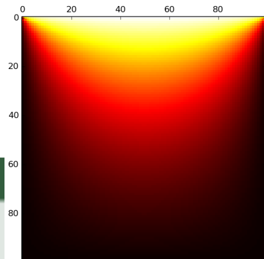
eScience Center
Niels Bohr Institute
University of Copenhagen

UNIVERSITY OF COPENHAGEN

Design Philosophy of Bohrium

- Do science, not programming
- No bugs
  - Correctness above all
- Future proof
  - Don't optimize for today's hardware

Python / NumPy



Heat Equation – a 5-point stencil

```
import numpy
def heat2d(G, epsilon):
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(G[1:-1,1:-1]+G[-2:,1:-1]+\
            G[2:,1:-1]+G[1:-1,:2]+G[1:-1,2:]
    delta = numpy.sum(numpy.abs(tmp-center))
    center[:] = tmp
```

UNIVERSITY OF COPENHAGEN

Python

Heat Equation – a 5-point stencil

```python
import numpy
def heat2d(G, epsilon):
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(G[1:-1,1:-1]+G[-2:,1:-1]+\
          G[2:,1:-1]+G[1:-1,:2]+G[1:-1,2:]
    delta = numpy.sum(numpy.abs(tmp-center))
    center[:] = tmp
```

OpenMP

Multi-core

```c
#include <math.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = size+2; //Size + borders.
  double *T = malloc(gsize*gsize*sizeof(double));
  double delta = epsilon+1;
  while(delta > epsilon)
  {
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ +
*down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
  }
}
```

OpenMP+MPI

Multi-core, Multiprocessor

```c
#include <math.h>
#include <mpi.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = SIZE+2; //Size + borders.
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
  MPI_Comm comm;
  int periods[] = {0};
  MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                  periods, 1, &comm);
  int l_size = SIZE / worldsize;
  if(myrank == worldsize-1)
    l_size += SIZE % worldsize;
  int l_gsize = l_size + 2;//Size + borders.

  double delta = epsilon+1;
  while(delta > epsilon)
  {

    int p_src, p_dest;
    //Send/receive - neighbor above
    MPI_Cart_shift(comm,0,1,&p_src,&p_dest);
    MPI_Sendrecv(grid+gsize,gsize,MPI_DOUBLE,
                 p_dest,1,grid,gsize, MPI_DOUBLE,
                 p_src,1,comm,MPI_STATUS_IGNORE);
    //Send/receive - neighbor below
    MPI_Cart_shift(comm,0,-1,&p_src,&p_dest);
    MPI_Sendrecv(grid+(l_gsize-2)*gsize,
                 gsize,MPI_DOUBLE,
                 p_dest,1,grid+(l_gsize-1)*gsize,
                 gsize,MPI_DOUBLE,
                 p_src,1,comm,MPI_STATUS_IGNORE);
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ +
*down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
    MPI_Allreduce(delta, delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  }
}
```

+Latency Hiding

Double Buffering

```c
#include <math.h>
#include <mpi.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = SIZE+2; //Size + borders.
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
  MPI_Comm comm;
  int periods[] = {0};
  MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                  periods, 1, &comm);
  int l_size = SIZE / worldsize;
  if(myrank == worldsize-1)
    l_size += SIZE % worldsize;
  int l_gsize = l_size + 2;//Size + borders.

  double delta = epsilon+1;
  while(delta > epsilon)
  {
    int p_src, p_dest;
    MPI_Request reqs[4];
    //Initiate send/receive - neighbor above
    MPI_Cart_shift(comm, 0, 1, &p_src, &p_dest);
    MPI_Isend(grid+gsize, gsize, MPI_DOUBLE, p_dest,
              1, comm, &reqs[0]);
    MPI_Irecv(grid, gsize, MPI_DOUBLE, p_src,
              1, comm, &reqs[1]);
    //Initiate send/receive - neighbor below
    MPI_Cart_shift(comm, 0, -1, &p_src, &p_dest);
    MPI_Isend(grid+(l_gsize-2)*gsize, gsize,
              MPI_DOUBLE,
              p_dest, 1, comm, &reqs[2]);
    MPI_Irecv(grid+(l_gsize-1)*gsize, gsize,
              MPI_DOUBLE,
              p_src, 1, comm, &reqs[3]);
    //Handle the non-border elements.
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size-1; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ +
*down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    //Handle the upper ghost line
    MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
    {
      int a = 0 * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ +
*down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    //Handle the lower ghost line
    MPI_Waitall(2, reqs+2, MPI_STATUSES_IGNORE);
    {
      int a = (size-1) * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ +
*down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
    MPI_Allreduce(delta, delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  }
}
```

OpenCL

GPGPU

```cpp
#include <iostream>
#include <sys/time.h>
#include <CL/cl.hpp>
#include <stdexcept>
#include <sys/time.h>
#include <cstdio>
#include <fstream>
#include <sstream>
#include <utility>

cl::Context context;
std::vector<cl::Device> devices;
cl::CommandQueue commandQueue;
cl::Kernel kernel;

#define TPB 32
#ifndef DTYPE
#define DTYPE double
#endif
#define STR(s) #s
#define xSTR(s) STR(s)

const char* kernel_str[] = "
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void heat_eq_jacobi(uint width, uint height
                             , __global DTYPE* in
                             , __global DTYPE* out
                             , __global DTYPE* delta)
{
    uint gid = get_global_id(0);
    if (gid >= width)
        return;
    uint w = width + 2;
    uint h = height + 2;
    DTYPE d = 0.0;
    for (uint i = 0; i < height; ++i)
    {
        uint offset = i*w;
        DTYPE up     = in[gid+1+offset];
        DTYPE left   = in[gid+w+offset];
        DTYPE right  = in[gid+w+2+offset];
        DTYPE down   = in[gid+1+w*2+offset];
        DTYPE center = in[gid+w+1+offset];
        DTYPE out_center = (center + up + left + right + down) * 0.2;
        out[gid+w+1+offset] = out_center;
        d += fabs(out_center - center);
    }
    delta[gid] = d;
}
"

void heat2d(int size, double *grid, double epsilon)
{
    // Commandline arguments
    char *opencl_file = argv[1];
    unsigned int width  = size;
    unsigned int height = size;
    unsigned int iter   = atoi(argv[4]);

    // Setup OpenCL execution system
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    bool foundPlatform = false;
    std::vector<cl::Platform>::iterator it;
    for (it = platforms.begin(); it != platforms.end(); ++it)
    {
        try {
            cl_context_properties props[] = {CL_CONTEXT_PLATFORM, (cl_context_properties)(*it)(),0};
            context = cl::Context(CL_DEVICE_TYPE_GPU, props);
            foundPlatform = true;
            break;
        }
        catch (cl::Error e)
        {
            foundPlatform = false;
        }
    }
    if (foundPlatform)
    {
        devices = context.getInfo<CL_CONTEXT_DEVICES>();
        commandQueue = cl::CommandQueue(context,devices[0],0);
    } else {
        throw std::runtime_error("Could not find valid OpenCL platform.");
    }

    // Compile kernel source
    cl::Program::Sources source(1,std::make_pair(kernel_str,0));
    cl::Program program(context, source);
    try {program.build(devices);}
    catch (cl::Error e)
    {
        std::cout << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]) << std::endl;
    }
    kernel = cl::Kernel(program, "heat_equation");

    // Set up buffers and copy data
    cl::Buffer inDev = cl::Buffer(context, CL_MEM_READ_WRITE, grid_size, NULL);
    cl::Buffer outDev = cl::Buffer(context, CL_MEM_READ_WRITE, grid_size, NULL);
    cl::Buffer deltaDev = cl::Buffer(context, CL_MEM_READ_WRITE, delta_size, NULL);
    cl::Buffer deltaHost = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, delta_size, NULL);
    DTYPE* deltaHostPtr = (DTYPE*)commandQueue.enqueueMapBuffer(deltaHost, CL_TRUE, CL_MAP_WRITE, 0, delta_size);
    commandQueue.enqueueWriteBuffer(inDev, CL_FALSE, 0, grid_size, grid);
    commandQueue.enqueueCopyBuffer(inDev, outDev, 0, 0, grid_size);

    DTYPE delta = 0.0;
    while(delta > epsilon)
    {
        kernel.setArg(0,width);
        kernel.setArg(1,height);
        kernel.setArg(2,inDev);
        kernel.setArg(3,outDev);
        kernel.setArg(4,deltaDev);
        commandQueue.enqueueNDRangeKernel(kernel, cl::NullRange,
                                          cl::NDRange((width/TPB+1)*TPB), cl::NDRange(TPB));
        commandQueue.enqueueReadBuffer(deltaDev, CL_FALSE, 0, delta_size, deltaHostPtr);
        commandQueue.finish();
        delta = 0.0;
        for (unsigned int i = 0; i < width; ++i)
            delta += deltaHostPtr[i];
        cl::Buffer tmp = inDev;
        inDev = outDev;
        outDev = tmp;
    }
    commandQueue.enqueueReadBuffer(inDev, CL_TRUE, 0, grid_size, grid);
    return 0;
}
```

UNIVERSITY OF COPENHAGEN

Python / NumPy

```
import numpy
def heat2d(G, epsilon):
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(G[1:-1,1:-1]+G[-2:,1:-1]+\
          G[2:,1:-1]+G[1:-1,:2]+G[1:-1,2:]
    delta = numpy.sum(numpy.abs(tmp-center))
    center[:] = tmp
```

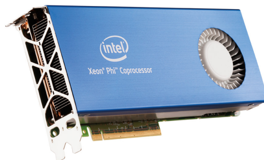
python -m bohrium heat2d.py

Bytecode

```
ADD t1, center, north
ADD t2, t1, south
FREE t1
ADD t3, t2, east
FREE t2
ADD t4, t3, west
FREE t3
MUL tmp, t4, 0.2
FREE t4
MINUS t5, tmp, center
ABS t6, t5
FREE t5
ADD_REDUCE t7, t6
FREE t6
ADD_REDUCE delta, t7
FREE t7
COPY center, tmp
FREE tmp
SYNC delta
```

OpenMP

OpenCL



 UNIVERSITY OF COPENHAGEN

Performance – Speedup VS NumPy

Speedup vs NumPy

30
25
20
15
10
5
0

Leibnitz Pi
Black Scholes
Monte Carlo Pi
Heat Equation
Rosenbrock

C99
Bohrium
Bohrium*4

Freely Available and Open Source

www.bh107.org

www.github.com/bh107/bohrium

UNIVERSITY OF COPENHAGEN

Performance – Bohrium vs C

	C99	Bohrium		NumPy
Threads	1	1	4	1
Leibnitz Pi	1	0.99	0.25	1.79
Black Scholes	1	0.95	0.25	4.76
Monte Carlo Pi	1	0.98	0.25	1.11
Heat Equation	1	1.45	0.87	5.00
Rosenbrock	1	0.96	0.27	7.69

Table: Normalized runtime of Bohrium using C99 as baseline **(lower is better)**. The timings are for full runs.

ANSI C – Sequential

Sequential

```
#include <math.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = size+2; //Size + borders.
  double *T = malloc(gsize*gsize*sizeof(double));
  double delta = epsilon+1;
  while(delta > epsilon)
  {
    double *a = grid;
    double *t = T;
    delta = 0;
    for(i=0; i<size; ++i)
    {
      double *up     = a+1;
      double *left   = a+gsize;
      double *right  = a+gsize+2;
      double *down   = a+1+gsize*2;
      double *center = a+gsize+1;
      double *t_center = t+gsize+1;
      for(j=0; j<size; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
      a += gsize;
      t += gsize;
    }
    memcpy(A, T, gsize*gsize*sizeof(double));
  }
}
```

ANSI C – OpenMP

Multi-core

```
#include <math.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = size+2; //Size + borders.
  double *T = malloc(gsize*gsize*sizeof(double));
  double delta = epsilon+1;
  while(delta > epsilon)
  {
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
  }
}
```

UNIVERSITY OF COPENHAGEN

ANSI C – OpenMP + MPI

Multi-core, Multiprocessor

```
#include <math.h>
#include <mpi.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = SIZE+2; //Size + borders.
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
  MPI_Comm comm;
  int periods[] = {0};
  MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                  periods, 1, &comm);
  int l_size = SIZE / worldsize;
  if(myrank == worldsize-1)
     l_size += SIZE % worldsize;
  int l_gsize = l_size + 2;//Size + borders.

  double delta = epsilon+1;
  while(delta > epsilon)
  {

    int p_src, p_dest;
    //Send/receive - neighbor above
    MPI_Cart_shift(comm,0,1,&p_src,&p_dest);
    MPI_Sendrecv(grid+gsize,gsize,MPI_DOUBLE,
                 p_dest,1,grid,gsize, MPI_DOUBLE,
                 p_src,1,comm,MPI_STATUS_IGNORE);
    //Send/receive - neighbor below
    MPI_Cart_shift(comm,0,-1,&p_src,&p_dest);
    MPI_Sendrecv(grid+(l_gsize-2)*gsize,
                 gsize,MPI_DOUBLE,
                 p_dest,1,grid+(l_gsize-1)*gsize,
                 gsize,MPI_DOUBLE,
                 p_src,1,comm,MPI_STATUS_IGNORE);
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
    MPI_Allreduce(delta, delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  }
}
```

UNIVERSITY OF COPENHAGEN

ANSI C – OpenMP + MPI + Latency Hidding

Double Buffering

```c
#include <math.h>
#include <mpi.h>
void heat2d(int size, double *grid, double epsilon)
{
  int gsize = SIZE+2; //Size + borders.
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
  MPI_Comm comm;
  int periods[] = {0};
  MPI_Cart_create(MPI_COMM_WORLD, 1, &worldsize,
                  periods, 1, &comm);
  int l_size = SIZE / worldsize;
  if(myrank == worldsize-1)
     l_size += SIZE % worldsize;
  int l_gsize = l_size + 2;//Size + borders.

  double delta = epsilon+1;
  while(delta > epsilon)
  {

    int p_src, p_dest;
    MPI_Request reqs[4];
    //Initiate send/receive - neighbor above
    MPI_Cart_shift(comm, 0, 1, &p_src, &p_dest);
    MPI_Isend(grid+gsize, gsize, MPI_DOUBLE, p_dest,
              1, comm, &reqs[0]);
    MPI_Irecv(grid, gsize, MPI_DOUBLE, p_src,
              1, comm, &reqs[1]);
    //Initiate send/receive - neighbor below
    MPI_Cart_shift(comm, 0, -1, &p_src, &p_dest);
    MPI_Isend(grid+(l_gsize-2)*gsize, gsize,
              MPI_DOUBLE,
              p_dest, 1, comm, &reqs[2]);
    MPI_Irecv(grid+(l_gsize-1)*gsize, gsize,
              MPI_DOUBLE,
              p_src, 1, comm, &reqs[3]);
    //Handle the non-border elements.
    delta = 0;
    #pragma omp parallel for shared(grid,T) reduction(+:delta)
    for(i=0; i<size-1; ++i)
    {
      int a = i * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    //Handle the upper ghost line
    MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
    {
      int a = 0 * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    //Handle the lower ghost line
    MPI_Waitall(2, reqs+2, MPI_STATUSES_IGNORE);
    {
      int a = (size-1) * gsize;
      double *up     = &grid[a+1];
      double *left   = &grid[a+gsize];
      double *right  = &grid[a+gsize+2];
      double *down   = &grid[a+1+gsize*2];
      double *center = &grid[a+gsize+1];
      double *t_center = &T[a+gsize+1];
      for(j=0; j<size-1; ++j)
      {
        *t_center = (*center + *up++ + *left++ + *right++ + *down++) / 5.0;
        delta += fabs(t_center+center);
      }
    }
    memcpy(grid, T, gsize*gsize*sizeof(double));
    MPI_Allreduce(delta, delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  }
}
```

UNIVERSITY OF COPENHAGEN

C++11 – OpenCL

GPGPU

```
#include <iostream>
#include <sys/time.h>
#include <CL/cl.hpp>
#include <stdexcept>
#include <sys/time.h>
#include <cstdio>
#include <fstream>
#include <sstream>
#include <utility>

cl::Context context;
std::vector<cl::Device> devices;
cl::CommandQueue commandQueue;
cl::Kernel kernel;

#define TPB 32
#ifndef DTYPE
#define DTYPE double
#endif
#define STR(s) #s
#define xSTR(s) STR(s)

const char* kernel_str[] = "
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void heat_eq_jacobi(uint width, uint height
                             , __global DTYPE* in
                             , __global DTYPE* out
                             , __global DTYPE* delta)
{
    uint gid = get_global_id(0);
    if (gid >= width)
        return;
    uint w = width + 2;
    uint h = height + 2;
    DTYPE d = 0.0;
    for (uint i = 0; i < height; ++i)
    {
        uint offset = i*w;
        DTYPE up     = in[gid+1+offset];
        DTYPE left   = in[gid+w+offset];
        DTYPE right  = in[gid+w+2+offset];
        DTYPE down   = in[gid+1+w*2+offset];
        DTYPE center = in[gid+w+1+offset];
        DTYPE out_center = (center + up + left + right + down) * 0.2;
        out[gid+w+1+offset] = out_center;
        d += fabs(out_center - center);
    }
    delta[gid] = d;
}
"

void heat2d(int size, double *grid, double epsilon)
{
    // Commandline arguments
    char *opencl_file = argv[1];
    unsigned int width  = size;
    unsigned int height = size;
    unsigned int iter   = atoi(argv[4]);

    // Setup OpenCL execution system
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    bool foundPlatform = false;
    std::vector<cl::Platform>::iterator it;
    for (it = platforms.begin(); it != platforms.end(); ++it)
    {
        try {
            cl_context_properties props[] = {CL_CONTEXT_PLATFORM, (cl_context_properties)(*it)(),0};
            context = cl::Context(CL_DEVICE_TYPE_GPU, props);
            foundPlatform = true;
            break;
        }
        catch (cl::Error e)
        {
            foundPlatform = false;
        }
    }
    if (foundPlatform)
    {
        devices = context.getInfo<CL_CONTEXT_DEVICES>();
        commandQueue = cl::CommandQueue(context,devices[0],0);
    } else {
        throw std::runtime_error("Could not find valid OpenCL platform.");
    }

    // Compile kernel source
    cl::Program::Sources source(1,std::make_pair(kernel_str,0));
    cl::Program program(context, source);
    try {program.build(devices);}
    catch (cl::Error e)
    {
        std::cout << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]) << std::endl;
    }
    kernel = cl::Kernel(program, "heat_equation");

    // Set up buffers and copy data
    cl::Buffer inDev = cl::Buffer(context, CL_MEM_READ_WRITE, grid_size, NULL);
    cl::Buffer outDev = cl::Buffer(context, CL_MEM_READ_WRITE, grid_size, NULL);
    cl::Buffer deltaDev = cl::Buffer(context, CL_MEM_READ_WRITE, delta_size, NULL);
    cl::Buffer deltaHost = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, delta_size, NULL);
    DTYPE* deltaHostPtr = (DTYPE*)commandQueue.enqueueMapBuffer(deltaHost, CL_TRUE, CL_MAP_WRITE, 0, delta_size);
    commandQueue.enqueueWriteBuffer(inDev, CL_FALSE, 0, grid_size, grid);
    commandQueue.enqueueCopyBuffer(inDev, outDev, 0, 0, grid_size);

    DTYPE delta = 0.0;
    while(delta > epsilon)
    {
        kernel.setArg(0,width);
        kernel.setArg(1,height);
        kernel.setArg(2,inDev);
        kernel.setArg(3,outDev);
        kernel.setArg(4,deltaDev);
        commandQueue.enqueueNDRangeKernel(kernel, cl::NullRange,
                                          cl::NDRange((width/TPB+1)*TPB), cl::NDRange(TPB));
        commandQueue.enqueueReadBuffer(deltaDev, CL_FALSE, 0, delta_size, deltaHostPtr);
        commandQueue.finish();
        delta = 0.0;
        for (unsigned int i = 0; i < width; ++i)
            delta += deltaHostPtr[i];
        cl::Buffer tmp = inDev;
        inDev = outDev;
        outDev = tmp;
    }
    commandQueue.enqueueReadBuffer(inDev, CL_TRUE, 0, grid_size, grid);
    return 0;
}
```