CS 6850 - Assignment - OpenMP/CUDA PortabilityThrough Kokkos
Brad Peterson - Weber State University

Overall goal: Utilize a portability tool to implement a graphics blurring problem capable of execution on multiple threading backends.

Your job is to simply load in a bitmap, then perform a Gaussian Smoothing algorithm. A good explanation is found here: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

For 5000 level students

Complete only the Gaussian Smoothing algorithm on your assignment

For 6000 level students

Run the program using MPI, splitting each MPI rank to use its own quadrant of the image. For each MPI rank, complete the Gaussian Smoothing algorithm.

You do not need to split the .bmp into 4 separate chunks. You also do not need to have the 0th MPI rank splice the image back together. Instead, simply save 4 images back out to file, each rank saving a blurred region of its own assigned area. For example, if rank 2 is assigned to blur on the 3/4th set of rows, then save an output file is sized with the same dimensions as the original, and only output new pixels for the 3/4th set of rows (the other fourths can be left as original or simply written as black).

For all students

The background of his homework is the Gaussian Smoothing algorithm, which determines a new pixel color value by utilizing nearby pixel values. The closer the pixel, the larger its weight in affecting the target pixel. The 2D Gaussian function used here is given by:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

For example, suppose you are working on finding a new smoothed pixel value for the pixel at row 3 column 3. That will be our new origin of (0,0) for this function. To find the weight of the pixel at row 1 column 2, the x and y would be (-2, -1), as they are offset that direction from the origin.

For this assignment, you do not compute the above formula, instead, you can simply use a 5x5 “kernel”. While you can compute your own, the link above gave one for you. (Theirs was computed by summing together a million points within each pixel, so theirs will be different from just merely running the formula on a single discrete integer (x,y) value.)

$$\frac{1}{273}$$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Thus, if your origin is a pixel at row 2 and column 2, then multiply the pixel value by 41. Then multiply the pixel value at row 2 column 3 by 26. Multiply the pixel value at row 2 column 4 by 7, and so on. You must multiply all 25 pixel values, sum the results, then divide by 273. The resulting integer is your new pixel value for row 2 and column 2.

Some bitmap caveats and details:

- Pixel values are not a single integer, but rather a 3-tuple of integers in a red-green-blue format. For each pixel, you must repeat the process three times, once for red, once for green, and once for blue. You will store the result as the red value, the green value, and the blue value, respectively.
- Do not store your results in-place. Rather, create a new bitmap that is a copy of the first. Write your results into the new bitmap.
- For this assignment, the bitmap uses 24-bit colors, so red has a range of values 0-255, green gets 0-255, and blue gets 0-255. Thus, each color component of a pixel is 8 bits.
- Bitmaps store the bottom left pixel first, and the top right pixel last. The data is row-major, so the second pixel in the bitmap is going to be the pixel to the right of the bottom left pixel.
- Bitmaps store the bits in blue/green/red order. Not red/green/blue.
- For this assignment, do not worry about any edges. Only process pixels that are at least 3 pixels in from an edge in any direction.
- Many programmers confuse *i*/*j* with width/height. Since height is the number of rows, *i* should work with height, and *j* should work with width. That means your view should be allocated to have a height dimension first, and a width dimension second. An outer *i* loop should iterate up to height, and an inner *j* loop should iterate up to width.
- If you wish to debug your pixel values against the image, remember that bitmap's (0,0) is bottom left, and graphic viewers put (0,0) top left. I fixed this debugging dilemma by flipping my image vertically in an image viewer, as now the coordinates line up. Also, I used Gimp, which indicates the pixels in (column, row), so I had to mentally flip them in my head.

Overall, your assignment just needs to iterate across all appropriate pixels, compute a new pixel value using its neighbors, and then store the new pixel value out to file.

You can find the image at: /share/HK-7_left_H6D-400c-MS.bmp. It is 1.2 GB and is a 23200x17400 image = 404 megapixel image.

For testing purposes, I strongly recommend you utilize
You can find the image at: /share/HK-7_left_H6D-400c-MS_screw.bmp. It is a 288x328 pixel image.

You must utilize Kokkos parallel for loops and Kokkos Views. Your parallel loop must be fully portable with no OpenMP/CUDA specific code found in your assignment. You should not create one single binary that works for both OpenMP and CUDA. Rather, you should build for OpenMP using g++ and verify that. Then build for CUDA using nvcc_wrapper and verify that.

Verification

Verify if your code is computing correctly by printing the values at these pixels. The output should also match.

```
The red, green, blue at (8353, 9111) (origin bottom left) is (252, 249, 248)
The red, green, blue at (8351, 9113) (origin bottom left) is (141, 76, 68)
The red, green, blue at (6352, 15231) (origin bottom left) is (67, 35, 18)
The red, green, blue at (10559, 10611) (origin bottom left) is (191, 187, 172)
The red, green, blue at (10818, 20226) (origin bottom left) is (73, 75, 69)
```

Submission

Obtain overhead wall times for the following:

1) Loading the image into a Kokkos View
2) Writing the image back out to file from a Kokkos View
3) For CUDA runs, the time it takes to copy data from host to GPU memory
4) For CUDA runs, the time it takes to copy data from GPU to host memory

Obtain wall time for processing the image in these scenarios:

1) Serial execution (1 thread)
2) OpenMP execution (let Kokkos manage its own threading)
3) CUDA execution (do not time the cost to copy data in and out of GPUs, just the kernel run)
4) OpenMP + MPI execution (Not needed for 5000 level students)
5) CUDA + MPI execution (Not needed for 5000 level students)

Prepare a short report giving these wall times. Also submit your code file.

Below is starter code which should assist you:

```
#include <Kokkos_Core.hpp>
#include <fstream>
#include <chrono>
//#include <filesystem>
#include <cstdio>
#include <string>

using std::string;
using std::ios;
using std::ifstream;
using std::ofstream;

double duration(std::chrono::high_resolution_clock::time_point start, std::chrono::high_resolution_clock::time_point end )
{
  return std::chrono::duration<double, std::milli>(end - start).count();
}

int main(int argc, char ** argv)
{

  Kokkos::initialize(argc, argv);
  {
    string filename = "HK-7_left_H6D-400c-MS.bmp";
    //std::uintmax_t filesize = std::filesystem::file_size(filename);
    //printf("The file size is %ju\n", filesize);
```

```
    // Open File
    ifstream fin(filename, ios::in | ios::binary);

    if(!fin.is_open()) {
      printf("File not opened\n");
      return -1;
    }
    // The first 14 bytes are the header, containing four values.  Get those four values.
    char header[2];
    uint32_t filesize;
    uint32_t dummy;
    uint32_t offset;
    fin.read(header, 2);
    fin.read((char*)&filesize, 4);
    fin.read((char*)&dummy, 4);
    fin.read((char*)&offset, 4);
    printf("header: %c%c\n", header[0], header[1]);
    printf("filesize: %u\n", filesize);
    printf("dummy %u\n", dummy);
    printf("offset: %u\n", offset);
    int32_t sizeOfHeader;
    int32_t width;
    int32_t height;
    fin.read((char*)&sizeOfHeader, 4);
    fin.read((char*)&width, 4);
    fin.read((char*)&height, 4);
    printf("The width: %d\n", width);
    printf("The height: %d\n", height);
    uint16_t numColorPanes;
    uint16_t numBitsPerPixel;
    fin.read((char*)&numColorPanes, 2);
    fin.read((char*)&numBitsPerPixel, 2);
    printf("The number of bits per pixel: %u\n", numBitsPerPixel);
    if (numBitsPerPixel == 24) {
      printf("This bitmap uses rgb, where the first byte is blue, second byte is green, third byte is red.\n");
    }
    //uint32_t rowSize = (numBitsPerPixel * width + 31) / 32 * 4;
    //printf("Each row in the image requires %u bytes\n", rowSize);

    // Jump to offset where the bitmap pixel data starts
    fin.seekg(offset, ios::beg);

    // Read the data part of the file
    unsigned char* h_buffer = new unsigned char[filesize-offset];
    fin.read((char*)h_buffer, filesize-offset);
    std::chrono::high_resolution_clock::time_point start;
    std::chrono::high_resolution_clock::time_point end;
    printf("The first pixel is located in the bottom left.  Its blue/green/red values are (%u, %u, %u)\n", h_buffer[0],
h_buffer[1], h_buffer[2]);
    printf("The second pixel is to the right.  Its blue/green/red values are (%u, %u, %u)\n", h_buffer[3], h_buffer[4],
h_buffer[5]);

    // TODO: Read the image into Kokkos views

    delete[] h_buffer;


    start = std::chrono::high_resolution_clock::now();
    // TODO: Perform the blurring
    end = std::chrono::high_resolution_clock::now();
    printf("Time - %g ms\n", duration(start,end));

    // TODO: Verification
    printf("The red, green, blue at (8353, 9111) (origin bottom left) is (%d, %d, %d)\n", 0, 0, 0);
    printf("The red, green, blue at (8351, 9113) (origin bottom left) is (%d, %d, %d)\n", 0, 0, 0);
    printf("The red, green, blue at (6352, 15231) (origin bottom left) is (%d, %d, %d)\n", 0, 0, 0);
    printf("The red, green, blue at (10559, 10611) (origin bottom left) is (%d, %d, %d)\n", 0, 0, 0);
    printf("The red, green, blue at (10818, 20226) (origin bottom left) is (%d, %d, %d)\n", 0, 0, 0);

    //Print out to file output.bmp
```

```
    string outputFile = "output.bmp";
    ofstream fout;
    fout.open(outputFile, ios::binary);

    // Copy of the old headers into the new output
    fin.seekg(0, ios::beg);
    // Read the data part of the file
    char* headers = new char[offset];
    fin.read(headers, offset);
    fout.seekp(0, ios::beg);
    fout.write(headers, offset);
    delete[] headers;

    fout.seekp(offset, ios::beg);
    // TODO: Copy out the rest of the view to file (hint, use fout.put())
    fout.close();

  }
  Kokkos::finalize();
}
```

Below is Kokkos code which demonstrates host views and portable views for parallel regions of code.

```
#include <Kokkos_Core.hpp>
#include <cstdio>
#include <typeinfo>

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc,argv);
  {
  printf("Kokkos execution space is %s\n", typeid(Kokkos::DefaultExecutionSpace).name());
  int rows{10};
  int cols{16};
  Kokkos::View<float**, Kokkos::LayoutRight> portableInput("portableInput", rows, cols);
  Kokkos::View<float**, Kokkos::LayoutRight> portableOutput("portableOutput", rows, cols);
  Kokkos::View<float**, Kokkos::LayoutRight>::HostMirror hostInput = create_mirror(portableInput);
  Kokkos::View<float**, Kokkos::LayoutRight>::HostMirror hostOutput = create_mirror(portableOutput);

  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      hostInput(i,j) = i + j;
    }
  }
  Kokkos::deep_copy(portableInput, hostInput);

  Kokkos::parallel_for(Kokkos::RangePolicy<>(0, rows*cols), KOKKOS_LAMBDA (const int n) {
    int i = n / cols;
    int j = n % cols;
    portableOutput(i, j) = portableInput(i, j)*2;
    printf(" -%g- ", portableOutput(i,j));
  });

  Kokkos::deep_copy(hostOutput, portableOutput);

  printf("\nVerifying the view from host-side to verify\n");
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      printf("%3g ", hostOutput(i,j));
    }
    printf("\n");
  }
  }

  Kokkos::finalize();
  return 0;
}
```

Some compiling examples:

Kokkos OpenMP:

```
g++ -mavx512f -I/opt/kokkos-openmp/include -fopenmp -O3 -c hw4.cpp -o hw4.o ; g++ -mavx512f -fopenmp -L/opt/kokkos-openmp/lib hw4.o -lkokkoscore -ldl -o hw4.host
```

OpenMP Kokkos + MPI:

```
g++ -mavx512f -I/opt/kokkos-openmp/include -I/home/bpeterson/opt/openmpi-4.0.4/include -fopenmp -O3 -c hw4.cpp -o hw4.o ; g++ -mavx512f -fopenmp -L/opt/kokkos-openmp/lib -L/opt/openmpi-4.0.4/lib hw4.o -lkokkoscore -ldl -lmpi -o hw4.host
```

Note, this one requires running via: /opt/openmpi-4.0.4/bin/mpiexec -n 4 ./hw4.host

Kokkos CUDA:

```
/opt/kokkos/bin/nvcc_wrapper -DKOKKOS_ENABLE_CUDA_LAMBDA -I/opt/kokkos-cuda-openmp/include -mavx512f -arch=sm_75 -expt-extended-lambda -Xcompiler -fopenmp -O3 -c hw4.cpp -o hw4.o ; /opt/kokkos/bin/nvcc_wrapper -mavx512f -fopenmp -L/opt/kokkos/lib hw4.o -lkokkoscore -lcudart -ldl -o hw4.cuda
```

Kokkos CUDA + MPI:

```
/opt/kokkos/bin/nvcc_wrapper -DKOKKOS_ENABLE_CUDA_LAMBDA -I/opt/kokkos-cuda-openmp/include -mavx512f -arch=sm_75 -expt-extended-lambda -Xcompiler -L/opt/openmpi-4.0.4/lib/ -Wl,-rpath-/opt/openmpi-4.0.4/lib -I/opt/openmpi-4.0.4/include -g -Wall -fopenmp -O3 -c hw4.cpp -o hw4.o -lmpi ; /opt/kokkos/bin/nvcc_wrapper -mavx512f  -fopenmp -L/opt/kokkos/lib -L/opt/openmpi-4.0.4/lib hw4.o -lkokkoscore -lcudart -ldl -lmpi -o hw4.cuda
```

Note, this one requires running via: /opt/openmpi-4.0.4/bin/mpiexec -n 4 ./hw4.cuda