VSCSE Summer School 2010 : Proven Algorithmic Techniques for Many-core Processors

Hands-on Labs

Nasser Anssari

Li-Wen Chang

Hee-Seok Kim

Deepthi Nandakumar

Nady Obeid

Christopher Rodriguez

John Stratton

I-Jui Sung

Xiao-Long Wu

The IMPACT Research Group

Coordinated Science Laboratory

Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign

The purpose of the hands-on labs is to assist an online short course that teaches proven algorithmic techniques for many-core processors. This manual includes an introductory lab, followed by several well-known problems like 7-point stencil computation, Lattice-Boltzmann method, Columbic potential calculation, and MRI reconstruction.

In order to cater to students with different levels of understanding of CUDA programming, we provide 4 hands-on labs, each of which reinforces one or several algorithmic techniques and performance optimizations. Most labs are designed with levels of different difficulties. Due to limited time in a hands-on lab section, we suggest that you choose the difficulty level that best fits your programming skills. If you have time, you can take on another level. Here is the table of contents of this manual.

If you have questions during the lab sections, please raise your hand or post your questions on the discussion boards. The TA's will assist you as soon as they can.

Lab 0: Package Download and Environment Setup

Xiao-Long Wu (xiaolong@illinois.edu)

1. Objective

The purpose of this lab is to check your environment settings and to make sure you can compile and run CUDA programs on the NCSA AC cluster. The objectives of this lab are summarized below:

- To download the assignment package, unpack it and walk through the directory structure.
- Set up the environment for executing the assignments.

2. Preliminary work

Step 1: Use an **SSH program** to login to **ac.ncsa.uiuc.edu**, using the training account login and initial password given to you. Your home directory can be organized in any way you like. To unpack the SDK framework including the code for all of the lab assignments, execute the unpack command in the directory you would like the SDK to be deployed.

```
$> tar  –zxf  ~xiaolong/CUDA_WORKSHOP_UIUC1008.tgz
```

Step 2: Go to the lab0 directory and make sure it exists and is populated.

```
$> cd  CUDA_WORKSHOP_UIUC1008/benchmarks/deviceQuery/src/cuda
```

There should be at least two files:

- deviceQuery.cu
- Makefile

Note: **If you are a remote user, we recommend you use an FTP program with the SFTP protocol and your account/password to retrieve the source files on ac.ncsa.uiuc.edu for editing because the network may be unstable and disconnected during lab sections.**

3. Make the first CUDA program and execute it on the AC cluster

The execution steps of this lab are listed below. You shall use the commands listed here in the remaining labs in the manual.

Step 1: Set environment variables. **You have to do this step whenever you start a new terminal window.**

```
$> cd  CUDA_WORKSHOP_UIUC1008/
$> source  env.sh
Parboil library path is set: /home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008
```

Step 2: Compile the lab.

```
$> ./parboil  compile  deviceQuery  cuda
```

Step 3: Execute the lab.

```
$> ./parboil  run  deviceQuery  cuda  default
```

You shall see the following message.

```
PARBOIL_ROOT=/home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008 make -C
/home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008/common/src
make[1]: Entering directory
`/home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008/common/src'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory
`/home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008/common/src'
nvcc -L/home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008/common/lib -lm -lpthread -
lcuda -L/lib -L/usr/local/cuda/lib -
L/home/ac/xiaolong/CUDA_WORKSHOP_UIUC1008/common/lib
build/cuda_default/deviceQuery.o -o build/cuda_default/deviceQuery -
lparboil_cuda -lcuda -lparboil
There is 1 device supporting CUDA

Device 0: "Tesla T10 Processor"
  Major revision number:                         1
  Minor revision number:                         3
  Total amount of global memory:                 4294770688 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
```

```
  Clock rate:                                     1.30 GHz
  Concurrent copy and execution:                  Yes

Test PASSED
Parboil parallel benchmark suite, version 0.1
```

4. Understanding the Parboil framework

In the hands-on labs for the ManyCore Processors course, we use a framework called Parboil to organize the labs. This is different from the lab organization framework used in the Intro-to-CUDA hands-on labs. The Parboil framework is developed by the IMPACT Research group at the University of Illinois, and is meant to provide an easier interface to manipulate benchmarks and measure performance on a GPU platform.

All lab assignments shall be compiled and run by the parboil script under the unzipped lab package directory, as listed in the previous sections. The parboil script is designed to submit jobs to the AC cluster in a batch mode fashion.

The unzipped lab package is composed of the following directory structures. For simplicity, here only lists the information you may need to know.

File “env.sh”: This is the environment setting file.

File “parboil”: This is the main script to compile and run the labs.

Directory “benchmarks/”: This directory stores the labs.

Subdirectory list: cp/ deviceQuery/ lbm/ mri/ stencil/

Each lab is composed of the directories, “build”, “input”, “output”, “src”, and “tools”. Directory “build” stores the executables of each lab. Directory “input” stores the problem parameters and input data sets. Directory “output” stores the problem output data sets and golden results for comparison. Directory “src” stores the source files of the problem of different versions or phases. Directory “tools” stores the tools to assist the lab result comparison.

Directory “common/”: This directory contains libraries shared by all labs.

Subdirectory “src/”: This directory contains the main parboil library source file. If you plan to port the lab package to another environment, you should rebuild the parboil library at this place.

Directory “driver/”: Here lists the related tools to manipulate the compilation and execution of the labs.

The output log of each lab execution shall list the execution time taken for each of the following sections, IO, GPU, Copy, and Compute. Their meanings are listed below.

IO: Time spent in input/output.

GPU: Time spent computing on the GPU, recorded asynchronously.

Copy: Time spent synchronously moving data to/from GPU and allocating/freeing memory on the GPU.

Driver: Time spent in the host interacting with the driver, primarily for recording the time spent queueing asynchronous operations

Copy_Async: Time spent in asynchronous transfers

Compute: Time for all program execution on the host CPU other than parsing command line arguments, I/O, GPU, and copy.

CPU/GPU Overlap: Time double-counted in asynchronous and host activity: automatically filled in, not intended for direct usage**.**

Lab 1: 2-D blocking and Register Tiling

Li-Wen Chang (lchang20@illinois.edu), Deepthi Nandakumar (nandaku2@illinois.edu)

1. Objective

The purpose of this lab is to provide a real application environment to explore the performance effects of 2-D blocking with data reuse and register tiling transformations. This lab introduces the 7-point stencil probe application, a micro-benchmark and test-bed for large stencil grid applications.

This lab will draw on the lecture material on

Increasing locality in dense arrays (with a focus on tiling).

Improving efficiency and vectorization in dense arrays.

The source file directory for lab 1.1 contains two options for students:

Difficulty Level 1: Students who are experienced in parallel program optimization and have a good understanding of the lecture material should choose this section. Students choosing Lab 1.1 will need to implement Optimizations 1, 2 and 3 detailed in Section 3 below in the file **kernels1.1.cu**.

Difficulty Level 2: Students who are not very experienced in parallel programming and/or are not very familiar with the lecture material should choose this section. Students choosing Lab 1.2 will need to implement only Optimizations 2 and 3 detailed in Section 3 below in the file **kernels1.2.cu**.

2. Examine and Understand the 7 pt. Stencil Kernel

The parboil benchmark 'stencil' contains the data and source code, which should compile and run correctly with the parboil interface as it is. Note that the output comparison step will take several seconds.

```
$ ./parboil run stencil cuda default
```

All source code for the lab can be found in the benchmarks/stencil/src/cuda subdirectory of the provided lab package. The 7-pt stencil probe application is an example of nearest neighbor computations on an input 3-D array. Every element of the output array is described as a weighted linear combination of itself and 6 neighboring values as shown in Fig.1.

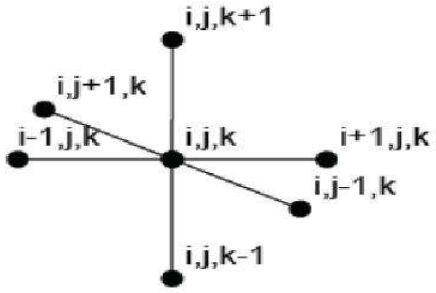


Fig 1: 7-point Stencil Illustration

The **main** function in the file **main.cu** contains the major components of kernel setup and execution including global memory allocation, input data copied to global memory, kernel launch and output data copied back into host memory. The input data is generated in the generate_data function in file main.cu.

The naive kernel **block2D_naive** is provided for reference in **kernels.cu**, and is invoked to compute the output grid as a weighted combination of elements of the input grid. The kernel configuration parameters and launch are shown in the **main.cu** file in the **main** function.

```
dim3 block (tx, ty, 1);
dim3 grid ((nx+tx-1)/tx, (ny+ty-1)/ty,1);
block2D_naive<<<grid, block>>>(fac,d_A0,d_Anext,nx,ny,nz,tx,ty,1);
```

Note that each thread block processes a **BLOCK_SIZE_X** by **BLOCK_SIZE_Y** block, within a for-loop that iterates in the z-direction. In the given kernel **block2D_naive**, each thread computes a single output point, by a weighted combination of 7 global memory elements that from the nearest neighbors.

To simplify boundary conditions, the outer boundary of each x-y plane (topmost and bottommost rows, and leftmost and rightmost columns) is initialized to zeros, and the thread blocks process the elements within this boundary. Thus, the thread blocks that form the outer boundary of the x-y plane have a fraction of threads that are idle.

3. Modifying the Kernel

Difficulty Levels

This lab is organized into two difficulty levels, which students can choose from, depending on their experience in parallel program optimization and/or their confidence and understanding of the lab material. The students will be required to implement the required optimizations in a manner that best exploits the performance potential of the application. The functions for Optimizations 1, 2, and 3 (detailed below) are declared as **block2D_opt_1**, **block2D_opt_2**, and **block2D_opt_3** in the file **kernels.cu**.

Lab 1.1: Students choosing this lab will need to implement Optimizations 1, 2, and 3 in the file **kernels1.1.cu**. The naive kernel **block2D_naive** (explained in Section 2) is provided for reference. Make sure that the file **kernels1.1.cu** is included in the **main.cu** file.

Lab 1.2: Students choosing this lab will be provided with the kernels **block2D_naive** and **block2D_shared** for reference. The students will need to implement only Optimizations 2 and 3 detailed below in the file **kernels1.2.cu**. Make sure that the file **kernels1.2.cu** is included in the **main.cu** file.

Optimization 1: Analyze the given kernel in detail. Every thread loads 7 global memory elements, thus meaning that even neighboring threads that share data points load the same point repetitively. Thus, this indicates good potential for performance improvement from data reuse. Provided below is a rough outline of the logical steps needed to arrive at an optimized kernel. Note that multiple lines of code may be necessary to implement each of these logical steps.

Step1: Declare a shared memory structure to be used for data sharing. Determine the appropriate size from your analysis of the problem.

Step 2: For each frame along the z-axis, threads should load data points in collaborative fashion into the shared memory structure. Ensure that synchronization is effectively used to ensure data consistency.

Step 3: For each frame along the z-axis, compute the weighted combination of nearest neighbor elements for the output point.

```
//Pseudo Code: Load elements into shared memory
for(each frame along z-axis){
      if(indices within range)
            shared_mem[index1] = data[index2];
      syncthreads();

      if(indices within range)
            output[index] = Weighted combination of neighboring
                              elements;
      syncthreads();
}
```

Fig 3: Pseudo Code for Optimization 1, Step 2 and Step 3

Step 4: Change the kernel launch commands within the main function to invoke kernel **block2D_opt_1**.

Step 5: Once you get a working implementation of the above methodology, experiment with different combinations to find the best performing solution.

Table 1

	Before Optimization 1	**After Optimization 1**	**Speedup**
GPU Time			

Optimization 2: Analyze the kernel you are working with in detail. Note that for each z-frame k, you load the (k-1), k, and $(k+1)^{th}$ frame (alternatively called the bottom, current and top frames). This means that for the next frame, only the $(k+2)^{th}$ frame needs to be loaded from global memory, as the k and $(k+1)^{th}$ frame were already fetched in the previous iteration. Provided below is a rough outline of the logical steps needed to arrive at an optimized kernel. Note that multiple lines of code may be necessary to implement each of these logical steps. **Note that the kernel block2D_opt_1 (defined in kernels.cu) is provided as additional reference, for those students choosing Lab 1.2.**

Step 1: Determine the appropriate size of the declared shared memory structure.

Step 2: Insert a prologue that loads the very first 2 required z-frame values from global memory into registers/shared memory respectively. This is to make sure that the first iteration of the inner loop has all the required z-frames. You may also decide to store all frames(top, bottom and current) in shared memory in which case, your code will look different from the example pseudo-codes shown.

```
//Declaring shared memory structures to hold 3 frames
__shared current;
float bottom, top;

//Prologue
if(indices within range){

      bottom = data[index]; //Corresponding data from Frame 0

      //Threads load collaboratively into shared memory
      current frame = {Load frame 1};
      __syncthreads();
 }
```

Fig 4: Pseudo Code for Optimization 2, Step 1 and Step 2

Step 3: Within the inner for-loop that iterates over the frames on the z-axis, load the next required value from the top frame into a register before computation. After computation, for the next iteration, the current frame becomes the bottom one, and the top frame becomes the current frame. Thus, note that within each iteration of the for-loop, only the required top frame needs to be loaded from global memory.

```
for(each frame along z-axis){
      if(indices within range)
            top = data[index]; //Load next frame data from global mem

      syncthreads();

      if(indices within range)
            output[index] = Weighted combination of
                  neighboring elements;

      syncthreads();

      bottom = Current[index];

      //Threads copy data collaboratively into shared memory
      current frame = {Copy data from top};
}
```

Fig 5: Pseudo Code for Optimization 2, Step 3

Step 4: Change the kernel launch commands within the **main** function to invoke kernel **block2D_opt_2**.

Table 2

	Before Optimization 2	**After Optimization 2**	**Speedup**
GPU Time			

Optimization 3: In the kernel that you are working with, note that each thread computes exactly one output point. By having each thread compute multiple points, we would be implementing the register tiling optimization detailed in the lecture. In this optimization, the student will modify the kernel to compute 2 output points instead of 1.

Step 1: Make sure that the sizes of the declared shared memory elements are large enough to hold input data for 2 output points.

Step 2: Modify the loads from global memory to shared memory to load the additional data points required for computing output point 2. If required, you may also need to modify the prologue and the frame update for the next iteration.

```
//Declaring shared memory structures to hold 3 frames for 2 //output
points

__shared current_combined;
float top1, top2, bottom1, bottom2;

//Prologue
if(indices within range){
      Bottom1 = data[index1];//Load Frame 0 for output point 1
      bottom2 = data[index2];//Load Frame 0 for output point 2

      //Threads load collaboratively into shared memory
      Current_combined = {Load frame 1 for o/p points 1 and 2};

 }
```

Fig 6: Pseudo Code for Optimization 3, Step 1 and 2

Step 3: Add additional computation steps to load the top frames corresponding to the 2 output points compute the 2nd output point.

Step 4: Change the kernel launch commands within the main function to invoke kernel **block2D_opt_3,** and also make necessary changes (if required) to the kernel configuration parameters, such as **block**, **grid** etc.

```
for(each frame along z-axis){

   if(indices within range)
       top1 = data[index1]; //Load data from next frame for o/p point 1

       top2 = data[index2]; //Load data from next frame for o/p point 2


      syncthreads();

      if(indices within range){
            output[index1] = Weighted combination of index1
                        neighboring elements;
      }
      if(indices within range){
            output[index2] = Weighted combination of index2
                        neighboring elements;
      }

      syncthreads();

      bottom1 = Current[index1];
      bottom2 = Current[index2];

      //Threads copy data collaboratively from top1 and top2
      Current_combined = {Copy data from top1 and top2 collaboratively};
}
```

Fig 7: Pseudo Code for Optimization 3, Step 3

Table 3

	Before Optimization 3	After Optimization 3s	Speedup
GPU Time			

4. Questions

Of all the configurations you tested, which ones performed best? A few questions that could get you to analyze the performance better are: What is the optimal size of the shared memory structure? Which are the nearest neighbor elements that have a high degree of sharing with neighboring threads? How can global memory loads be performed so as to make best use of the underlying memory infrastructure? (Hint: coalesced accesses). For Optimization 1, what patterns of the global memory loads performed best. Why? Can you think of other ways to load data into a 2-D shared memory structure that could give better performance? For Optimization 3, how did you choose the 2 output points that a single thread computes? Are there other combinations that you could have chosen?

Lab 2A: Data Layout Transformation

John Stratton (stratton@crhc.uiuc.edu), I-Jui Sung (sung10@illinois.edu)

1. Objective

The purpose of this lab is to provide a real application environment (Lattice-Botlzmann Method) to explore the performance effects of changing the layout of a multidimensional array of multi-element data structures affects performance of GPU code accessing that array. This lab will draw on the lecture material on data layout, coalescing, and warp scheduling.

2. Examine and Understand the LBM Kernel

The parboil benchmark 'lbm' contains the data and source code, which should compile and run correctly with the parboil interface as it is. Note that the output comparison step will take several seconds.

```
$> ./parboil run lbm cuda short
Parboil parallel benchmark suite, version 0.2
LBM_allocateGrid: allocated 130.9 MByte
MAIN_printInfo:
        grid size      : 100 x 100 x 130 = 1.30 * 10^6 Cells
        nTimeSteps     : 100
        result file    : run/short/reference.dat
        action         : store
        simulation type: lid-driven cavity
        obstacle file  : input/short/100_100_130_ldc.of


LBM_allocateGrid: allocated 130.9 MByte
LBM_allocateGrid: allocated 130.9 MByte
LBM_showGridStatistics:
        nObstacleCells:  112539 nAccelCells:   18432 nFluidCells: 1169029
        minRho:   1.0000 maxRho:   1.0000 mass: 1.300000e+06
        minU: 0.000000e+00 maxU: 0.000000e+00

timestep: 64
LBM_allocateGrid: allocated 130.9 MByte
LBM_showGridStatistics:
       nObstacleCells:  112539 nAccelCells:   18432 nFluidCells: 1169029
       minRho:   0.9344 maxRho:   1.0662 mass: 1.299998e+06
       minU: 3.789891e-08 maxU: 2.634398e-02

IO: 0.025683
GPU: 1.399837
Copy: 1.117329
Driver: 0.000507
Compute: 0.976847
CPU/GPU Overlap: 0.110463
Pass
```

All source code for the lab can be found in the benchmarks/lbm/src/cuda subdirectory of the provided lab package. The LBM application of this lab assignment simulates a closed, lid-driven cavity system. Aside from initialization and finalization, the entire simulation is performed in the loop in the 'main' function in the file main.cu.

```
for( t = 1; t <= param.nTimeSteps; t++ ) {
    pb_SwitchToTimer(&timers, pb_TimerID_GPU);
    CUDA_LBM_performStreamCollide( *CUDA_srcGrid, *CUDA_dstGrid );
    …
}
```

The kernel in lbm_kernels.cu will be invoked to advance the simulation forward one timestep. The kernel configuration parameters and launch are shown in the lbm.cu file in the CUDA_LBM_performStreamCollide function from lines 37 to 44.

```
void CUDA_LBM_performStreamCollide( LBM_Grid srcGrid, LBM_Grid dstGrid ) {
        dim3 dimBlock, dimGrid;
        dimBlock.x = SIZE_X;
        dimGrid.x = SIZE_Y;
        dimGrid.y = SIZE_Z;
        dimBlock.y = dimBlock.z = dimGrid.z = 1;
        performStreamCollide_kernel<<<dimGrid, dimBlock>>>(srcGrid,
dstGrid);
}
```

Examining the kernel in lbm_kernels.cu, observe that the kernel initially copies all fields of one cell from global memory to private variables, computes some output based on those private variables, and then writes output to global memory. For this lab, the computation is not particularly important to focus on.

The LBM simulation grid is a regular lattice division of physical space, where each cell contains 19 floating-point values recording the fluid flow in the 18 3-D adjacent and diagonal directions (North, South, East, West, Top, Bottom, and every compatible pair in that set), plus a fluid density value for the cell overall (“Center” or “C”, to keep naming convention). Finally, each cell contains a word of flags for determining whether the cell is an obstacle, fluid or driving cell. For the purposes of this lab, all you need to understand is that the data structure for a cell contains 20 values referenced by one- or two-character names.

The macros SWEEPX, SWEEPY and SWEEPZ name the variables defining the X, Y and Z coordinate of a cell. The macros SRC_* and DST_* use those variables to compute an index for where that cell should find its input and output values for each named field.

3. Changing the Data Layout

a) Gather vs. Scatter

Description: Each thread index must read in inputs from all neighboring cells in the previous time step, and write an output to all neighboring cells for the next time step. In the "gather" method, the data structure for a particular cell holds all the values that cell computed in the previous time step. Therefore, every thread must "gather" its own inputs from fields in its neighboring cells. The "scatter" method instead stores in each cell the values all its neighboring cells computed for it on the last time step. Therefore the input data for a cell in the current time step is all within its own fields, but it must write its output into its neighboring cell's fields.

The code has been designed such that only the preprocessor definition in line 33 of layout_config.h need be changed to choose between gather and scatter.

```
#if 1
#define GATHER
#else
#define SCATTER
#endif
```

Assignment: Test both gather and scatter compilations, and record the performance as a baseline for future experiments. These commands will be used to recompile and execute LBM for each test.

```
(edit benchmarks/lbm/src/cuda/layout_config.h)
$> ./parboil clean lbm cuda
$> ./parboil run lbm cuda short
```

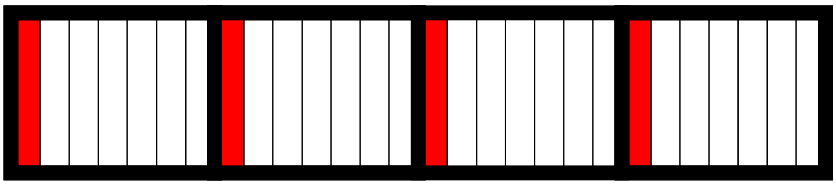
	Gather	Scatter
GPU time		

b) Array of Structures vs. Structure of Arrays

Description: Note that the flattening function CALC_INDEX computes a 1D index from a 4D index, with the dimensions, in order, being *element*, *x*, *y*, and *z*. In other words, it computes a 1D index from x, y and z according to row-major layout rules (see lectures slides on layout), multiplies that index by the number of elements in a cell, and adds the particular offset of the element requested.

```
#define CALC_INDEX(x,y,z,e) ( e + (N_CELL_ENTRIES *  \
                                   ((x)+(y)*PADDED_X+(z)*PADDED_X*PADDED_Y) ) )
```

This is called the “array of structures” layout, because if the cells were declared as C structures with named fields, this is the layout that would result, with fields highlighted if several threads each accessed a particular field of their own cell.

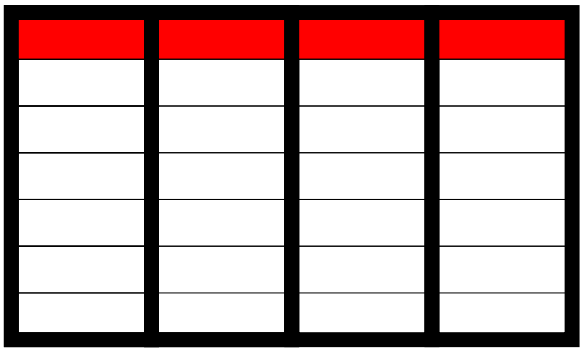


Assignment: Change the CALC_INDEX macro on lines 29 and 39 of layout_config.h so that it instead represents a “structure of arrays”. To do so, the mapping function must compute a 1D index from x, y and z as before, but then multiply element by the total size of the grid in the x, y and z dimensions and add that number to the 1D index.

Solution:

```
#define CALC_INDEX(x,y,z,e) ( TOTAL_PADDED_CELLS*e + \
                              ((x)+(y)*PADDED_X+(z)*PADDED_X*PADDED_Y) )
```

The 1D index from x, y and z is not scaled at all, in contrast with the previous case. This layout is called the “structure of arrays” layout because it is the layout that would result in declaring a C structure containing, for each component, an array for the entire simulation lattice's values of that component. The transformed layout conceptually looks more like this.



Measure the performance again, with both scatter and gather variations, and record them. Begin by copying your previous run times in the “Array of Structures” column.

GPU Time	Scatter	Gather
Array of Structures		
Structure of Arrays		

c) Adding Padding

Description: In addition to changing the order of indexes, you can add padding to any of the x, y or z dimensions to change the alignment properties of the data structure. For instance, by adding a padding of 28 elements to the X dimension, the 1D index computed from x, y and z will always be a multiple of 128 if x is 0.

Assignment: Set the padding of the x dimension to 28 by changing the value of PADDING_X in line 15 of layout_config.h, and measure the performance again.

Solution: change layout_config.h line 15 to

```
#define PADDING_X (28)
```

GPU Time: Padded Arrays	Scatter	Gather
Array of Structures		
Structure of Arrays		

4. Questions

Of all the configurations you tested, which ones performed best? Of the SoA/AoS and gather/scatter combinations, which ones were most improved by padding? Why? Can you think of any other padding or flattening function combinations that may perform even better? Try them out, and see if you can explain their performance as well.

Lab 2B: Binning with Uniform Distributions

Hee-Seok Kim (kim868@illinois.edu), Christopher Rodriguez (cirodrig@crhc.uiuc.edu)

1. Objective

This lab is an introduction to binning as a technique to help solve problems efficiently. As a case study, we investigate the calculation of electrostatic potential maps, which consists of a regularly spaced lattice of points in a space containing the summed potential contributions from the atoms. You will learn how binning can help solve problems on both CPU and GPU efficiently. Also you will discover how different tweaks in binning could lead to significant performance variation. Finally, you will be challenged with non-uniform input data, which does not lend itself to uniform binning thus motivating new and innovative approaches.

2. Prerequisite

In order to go through this lab, you need to have understanding of basic C and CUDA programming skills.

3. Brief Review: What is Binning? What Binning is for?

Binning is a process that groups data to form a chunk called *bin*. Each bin can have a representative property of data inside the bin. When data are properly binned, the problem solving could be coarsened due to the representative property of bin. This can bring great optimization opportunity with higher abstraction on input data.

Binning is useful for various algorithms dealing with huge data. Ray tracing, for example, uses KD-tree which is a kind of non-uniform sized bins that divides a scene into multiple bounding boxes to group polygons close to each other. We can ignore many polygons inside bounding boxes that have no chance of colliding with a ray being traced.

4. Problem Statement: Columbic Potential

The electrostatic interactions between atoms are of particular importance in molecular simulation. Atoms are modeled as point charges by assigning to each atom I at position $\vec{r_i}$, a fixed partial charge q_i. Closely related to the electrostatic force and interaction energy between atoms is the electrostatic potential V at a position $\vec{r}$ expressed here as a sum over all N atoms,

$$V(\vec{r};\vec{r}_1,\vec{r}_2,\ldots,\vec{r}_N)=\sum_{i=1}^{N}\frac{q_i}{4\pi\epsilon_0|\vec{r}-\vec{r}_i|}s(|\vec{r}-\vec{r}_i|)$$

……. Eq 1

In this lab, Eq. 1 is sampled at regularly spaced points over a volume to generate a map of the electrostatic potential in that volume. In this case, ~~r~~ ranges over a set of *M* regularly spaced lattice points. Setting the function $s(r) \equiv 1$ calculates the full (infinite distance) electrostatic potential contributed by all atoms to position ~~r~~, requiring quadratic computational work. Algorithmic efficiency is improved by choosing *s(r)* to yield a cutoff potential truncated beyond a fixed cutoff distance r_c. A common choice is given by

$$s(r) = \begin{cases} 1 - \frac{r}{r_c^2} & ,if\, r < r_c \end{cases}$$ Eq. 2

which smoothly shifts the potential to zero beyond r_c and regains the full electrostatic potential in the limit as r_c approaches infinity. Figure 1 illustrates how the electrostatic potential map is computed and the rendering of the map accordingly.

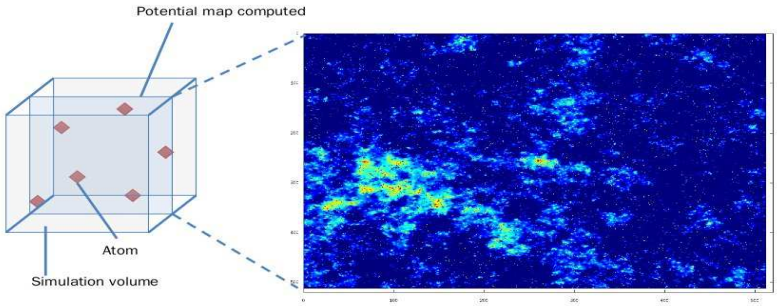


Figure 1: An Electrostatic Potential Map in a Simulation Volume

The computation of electrostatic potential maps is directly applicable to ion placement methods used to initially setup a biomolecular system. Moreover, it is useful for the analysis and visualization of the completed simulation; for instance, when visualizing the electrostatic contour lines around a molecular surface.

5. Approaches: From Naive to Binning

5.1 Naive Approach

The naive approach follows the equations shown above as Eq. 1 and Eq. 2. It iterates over every point on the output grid and on each point it sums up the Columbic Potential from atoms around it. The implementation is given in `benchmarks/cp/src/1`.

5.2 Binning Approach

We could get several important observations from the naive implementation as below.

1) It is wasteful to visit all atoms when the cutoff distance is relatively small.

- We can enhance the performance if we efficiently get the list of atoms within the cutoff distance for any given output grid point.

2) The cut-off distance is fixed through the algorithm run.

- We can create a list of neighboring points that lie within or along the cut off distance in a fixed size grid space.

With these observations, we could optimize the naive approach if we divide the simulation volume into a set of fixed size cubes that are uniformly distributed. It involves performing spatial hashing of the atoms into bins of a uniform grid. Then there is a neighborhood of bins that lie within or straddle the cut off radius for some points in the output grid. Therefore, we could derive another implementation as below.

Phase 1: Perform spatial hashing of the atoms into bins

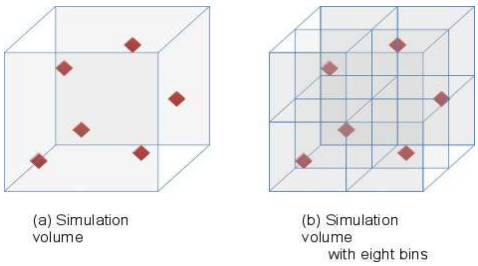


Figure 2: Binning Example of the Simulation Volume

In this phase, the simulation volume is divided into bins of the same size and capacity as illustrated in Figure 2. The number of atoms per bin is almost equal since we assume that atoms are uniformly distributed in the simulation volume. You will be given the initial configuration of binning such as size and capacity of a bin in the code. Nevertheless, some bins might not contain all atoms that fall into them due to the fixed capacity of a bin. Obviously these atoms should not be omitted anyway, so let’s collect them and process naively later.

When performing the binning, we would like to add extra bins surrounding the simulation volume. This comes in order to make the computation more regular around the edges of the simulation volume. Once we extend the simulation volume in this way, we don't need to care about edges and clipping conditions and so on. We will also use a bounding cube rather than a sphere for the search space of an output grid point. Note that these are the basic implementation ideas. Figure 3 illustrates these ideas.

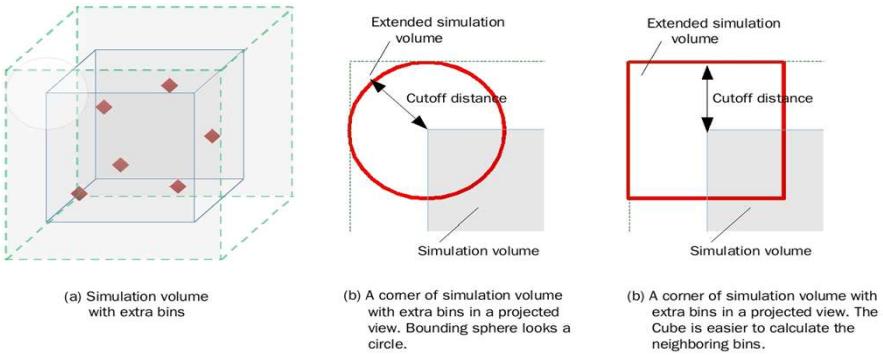


Figure 3: Illustration of the Simulation Volume Extension

Pseudo code below shows how to create bins.

```
    bins = [][BIN_CAPACITY];            // List of array
    extra_atoms = [];                   // To be handled naively later
    for each atom a,
        bin_loc = a.loc / BIN_SIZE;     // Spatial hashing, radix sort
        bin = bins[bin_loc]
        if bin is not full,
            bin.add(a);
        else
            extra_atoms.add(a);
```

Phase 2: Create neighborhood list

This phase deals with identification of the neighborhood of a region to be used in cutoff summation. The shape of neighborhood is precomputed with respect to the binning lattice in the form of a list of neighbor offsets. As shown in Figure 3(a), the area within the cut off radius forms a sphere in 3-dimensional space. In this lab, however, we assume that the shape of neighborhood is a bounding cube that has the sphere inside as shown in Figure 3(b) and (c) to reduce the programming effort, though it might cause some degree of inefficiency to the performance.

The following pseudo-code computes the neighborhood list for a bounding cube.

```
    neighbor_list = [];
    For offset from -(cut off radius) to +(cut off radius),
        // NOTE: uncomment below to make it a bounding sphere.
        // if (distance from zero to offset <= cut off radius) then
        neighbor_list.add(offset);
    }
```

Phase 3: Calculate the electrostatic potential map

This phase calculates the electrostatic potential map using the bins and neighbor list that are done in the previous phases. The idea is to iterate the output grid and for each point neighboring bins are to

be identified and summed up. The atoms that did not fit into bins need to processed by the CPU using the naive approach. The pseudo-code is shown as below.

```
// Part 1. compute with bins
For each output grid point p,
// identify central bin from the location of p
    center_bin = bins[p.loc / BIN_SIZE];
    for n in neighbor_list,
          bin = bins[center_bin.pos + n.pos];
          for each atom in atoms in bin,
               dist = |p.loc – atom.loc|
               p.energy += atom.q/dist*s(dist    //s(dist) by Eq. 2

// Part 2. Handle extra atoms
For each output grid point p,
     for each atom in extra_atoms,
          dist = |p.loc – atom.loc|
          p.energy += atom.q / dist * s(dist)
```

6. Implementation

6.1. Review the Naive Implementation

The purpose of this step is to get you familiar with basic idea to solve the problem. In this step, you don't need to implement anything. Instead, you should be comfortable with the source codes and some data structures that are provided. Please open `benchmarks/cp/src/1/cenergy.c`. Function `calc_energy()` is the common interface to solve the problem. The name and description of the function parameters are shown in Table 1.

Table 1: A Few Important Data Structures

Name	Type	Description
energygrid	**float***	One dimensional array to contain output points, which is as big as grid.x * grid.y * grid.z in linear address form.
grid	**voldim3i**	Dimension of output energy grid where cumulative potential is to be summed up. Note that grid.z equals to 1 to make the output grid a single plane. Voldim3i is defined as: typedef struct _tag { int x, y, z; } voldim3i;
atoms	**float***	Atoms information as follows. atoms[4 * n + 0] : x coordinate atoms[4 * n + 1] : y coordinate atoms[4 * n + 2] : z coordinate atoms[4 * n + 3] : particle charge (0 <= n < numatoms)
numatoms	**int**	Number of atoms in the simulation volume
gridspacing	**float**	Size of output grid lattice space
k	**int**	Z-axis of the output grid plane

Fill in the execution time in the following instruction table.

Source code to work	$PARBOIL_ROOT/benchmarks/cp/src/1/cenergy.c
How to build & run	# cd $PARBOIL_ROOT # ./parboil run cp 1 uniform
Time to run (measure it!)	

6.2 Binning

Throughout this step, you will implement the binning algorithm you have studied previously. This step has three sub-steps each of which is related to a phase of the algorithm described in Section 5.2 (Binning Approach). The following code snippet is the body of `calc_energy()` function in

```
// Phase 1. Perform the binning process
 create_uniform_bin(grid, num_atoms, gridspacing, atoms, cutoff);

// Phase 2. Create the neighbor list
 sol_create_neighbor_list(gridspacing, cutoff);

// Phase 3. Calculate energy using the bins and the neighbor list
//{
      sol_calc_energy_with_bins(
      energygrid, grid, atoms, num_atoms, gridspacing, cutoff, k);
      calc_extra(energygrid, grid, gridspacing, cutoff, k);
//}
```

`benchmarks/cp/src/2.1/cenergy.c` that computes the result which reveals all steps of the algorithm.

You will be asked to implement the functions called in `calc_energy()`. Note that it is strongly recommended to follow all the steps provided here because later steps might require the result of previous steps.

6.2.1 Implement uniform bins and spatial hashing atoms onto them

You need to create bins of uniform grid which divide the extended simulation volume. As mentioned earlier, this stage corresponds to Phase 1 of the algorithm shown in Phase 1 of Section 5.2. You need to fill Function **create_uniform_bin()** accordingly. Open the source code and search for "`Step 2.1`". Once you do it correctly, it will print out "`Pass`".

Source code to work	$PARBOIL_ROOT/benchmarks/cp/src/2.1/cenergy.c
How to build & run	# cd $PARBOIL_ROOT # ./parboil run cp 2.1 uniform

6.2.2 Implement list of neighborhood bins

Next, you need to create list of neighboring bins for a given location in the simulation volume. In this step, you need to fill Function **create_neighbor_list()** accordingly. Open the source code and search for "Step 2.2". Similarly, once you do it correctly, it will print out "Pass". Note that you may skip the previous stage by calling sol_create_uniform_bin() instead of create_uniform_bin() in Function calc_energy().

Source code to work	`$PARBOIL_ROOT/benchmarks/cp/src/2.2/cenergy.c`
How to build & run	`# cd $PARBOIL_ROOT` `# ./parboil run cp 2.2 uniform`

6.2.3 Implement columbic potential kernel

With the bins and neighborhood list you have done so far, finally you can implement a kernel which corresponds to Phase 3 of the algorithm. In this stage, you need to implement calc_energy_with_bins() accordingly. Open the source code and search for "Step 2.3". You might need to reuse what you have done in Steps 2.1 and 2.2. If you want to skip them and use the reference implementation, use sol_create_uniform_bin() and sol_create_neighbor_list() in Function calc_energy(). Write down the execution time when you succeed for later comparison with the GPU version.

Source code to work	`$PARBOIL_ROOT/benchmarks/cp/src/2.3/cenergy.c`
How to build & run	`# cd $PARBOIL_ROOT` `# ./parboil run cp 2.3 uniform`
Time to run (measure it!)	

6.3 Optimize the Performance on the GPU

6.3.1 Naive Approach

Now let's move on to the GPU version. Among the three phases of the algorithm, the last one seems a proper candidate for a GPU to handle. As such, we also focus on the last phase of the algorithm in this step. Your mission is to implement a CUDA version of what you have done previously. Open the source code and search for "Step 3.1". The strategy is to map output grid points onto CUDA grids and let the threads for each grid visit all the atoms within the cutoff range. This should look very similar what you have done in Step 2.3. Write down the execution time when you get the correct result. Again, you may use the reference implementation for 2.1 and 2.2.

Hint: Try to allocate a thread block for an output grid point. Then let the threads in the block contribute to the output. **If you haven't finished 2.3 yet, you can start from 3.1-help which implements the idea of this step.**

Source code to work	$PARBOIL_ROOT/benchmarks/cp/src/3.1/cenergy.cu
How to build & run	# cd $PARBOIL_ROOT # ./parboil run cp 3.1 uniform
Time to run (measure it!)	

6.3.2 Tiling Approach

Simply peeling the loop and mapping each loop index to a thread index or a block index should work nicely; however, it can also be optimized further. The observation from Step 3.1 is that atoms being loaded from the global memory will be used only once which wastes the memory bandwidth. The goal of this step is to optimize the memory bandwidth.

Once you load a bin and the atoms in it, they can be used for multiple output grid points nearby due to the coarse binning. Following is pseudo-code for the modified implementation. You should be able to configure the grid and the blocks for the CUDA kernel properly.

Hint: If you haven't finished 2.3 or 3.1 yet, you can start from 3.2-help which implements the idea of this step. Following is the pseudo-code.

```
For each output grid block b,            // blocks, not points
     center_bin = bins[b.loc / BIN_SIZE];
     for n in neighbor_list,
     bin = bins[center_bin.pos + n.pos];
     for each atom in bin.atoms,
          for each point p in b,   // iterate points in the block
          dist = | p.loc – atom.loc |
          p.energy += atom.q / dist * s(dist)
```

Please refer to the instruction as below.

Source code to work	$PARBOIL_ROOT/benchmarks/cp/src/3.2/cenergy.cu
How to build & run	# cd $PARBOIL_ROOT # ./parboil run cp 3.2 uniform
Time to run (measure it!)	

Once it works, you can cache atoms in the shared memory. It will provide you with slightly better performance.

7. Questions

1. Why does Step 3.2 have better potential than Step 3.1 in terms of memory performance?

2. Can you point out data structures that leverage faster memory system such as context memory, texture memory or shared memory? Explain and implement your answer. How much performance improvement can you expect?

3. How does the performance change when we adjust the binning configuration? Specifically, what happens when we change the capacity of the bin from 8 to 4? Why does that happen?

4. Try using non-uniformly distributed input data by changing input from uniform to non_uniform. What happens? What will you do to rectify this situation? What do you think are problems with your solution?

Lab 3: Binning with Non-Uniform Distributions

Nasser Anssari (anssari1@illinois.edu), Nady Obeid (obeid1@illinois.edu)

1. Objectives

In Lab 2B (Binning with Uniform Distributions), you optimized the execution of the Columbic Potential application using the binning technique and explored the benefits of spatially decomposing the computation and data in a manner which naturally maps to CUDA thread blocks and efficiently uses the memory system. Moreover, you took advantage of the uniform distribution of the input data to hone your implementation. In this lab, you will look into another example of applications which uses window functions (functions which are zero-valued outside some chosen interval), namely MRI reconstruction. In particular, you will analyze the impact of its non-uniform data distribution on the performance of the algorithmic approach of Lab 2B and consider other alternatives which may be better-suited for such distributions.

2. Lab Applications

Efficient computation of window functions is of particular importance due to their prominence in molecular modeling applications in domains that span biochemistry, materials science, thermal science, and astrophysics. Typically involving a large number of data points in multiple dimensions, such applications require novel spatial data structures and search algorithms, such as binning, for efficient data representation and querying.

3. Prerequisite Knowledge

1) Basic C Language and CUDA programming skills

2) Familiarity with major data structures and algorithms such as sorting and vector reduction

4. Theoretical Background

The MRI Reconstruction application, the example application of this lab, transforms MR data samples from the k-space into the image space using IFFT. Since MRI scanners typically use spiral trajectories in a cylindrical or spherical coordinate system (Figure 1), the image cannot be reconstructed by directly applying IFFF to the k-space samples.

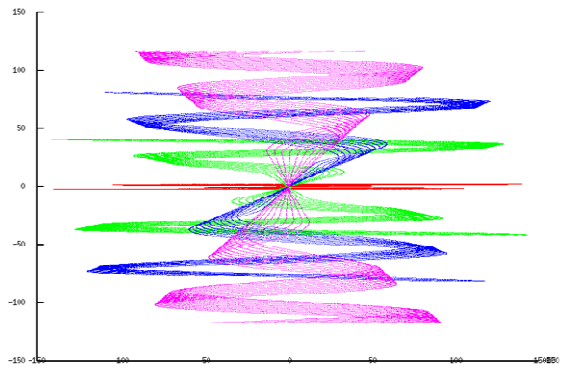


Figure 1: Typical Data Acquisition Paths in MRI Scanning

Instead, in a commonly used approach called gridding, the samples are first interpolated onto a uniform Cartesian grid and then reconstructed using IFFT (Figure 2). A convolution approach to gridding takes a k-space data point, convolves it with a gridding kernel, and accumulates the results on a Cartesian grid. The gridding kernel uses Kaiser-Bessel function, a window function.

The chief advantage of window functions comes from skipping input points which are known to be outside the cutoff radius for a localized region of the output grid. Still, a distance test is needed to check for the input points which satisfy this condition. To minimize the volume whose input data needs

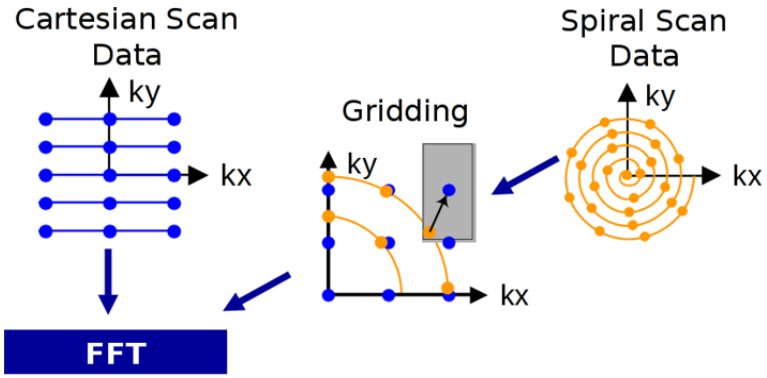


Figure 2: MRI Reconstruction

to be processed, input points can be spatially hashed into bins prior to output computation. A near-uniform distribution of input data, such as that of the Columbic Potential application of Lab 2B, allows using a pre-allocated array of bins with equal capacities to optimize the implementation of the algorithm. Such a data structure, however, is not suitable for non-uniform data distributions for reasons which will become evident through the course of this lab. This renders the previously tailored implementation inefficient and thus motivates probing alternative approaches.

One example approach substitutes “implicit dynamic” bins for the “explicit static” ones. While the output lattice is still decomposed into a set of predetermined bins, the input data is sorted based on the indices of these bins rather than distributed over a pre-allocated array thereof. The varying sizes of the resulting implicit bins require a subsequent reduction step to determine the beginning of each bin in the sorted input array (Figure 3). To improve the load balance across the bins processed on the GPU, a limit can be imposed on the bin capacity so that the superfluous data points from all bins are grouped and offloaded to the CPU.

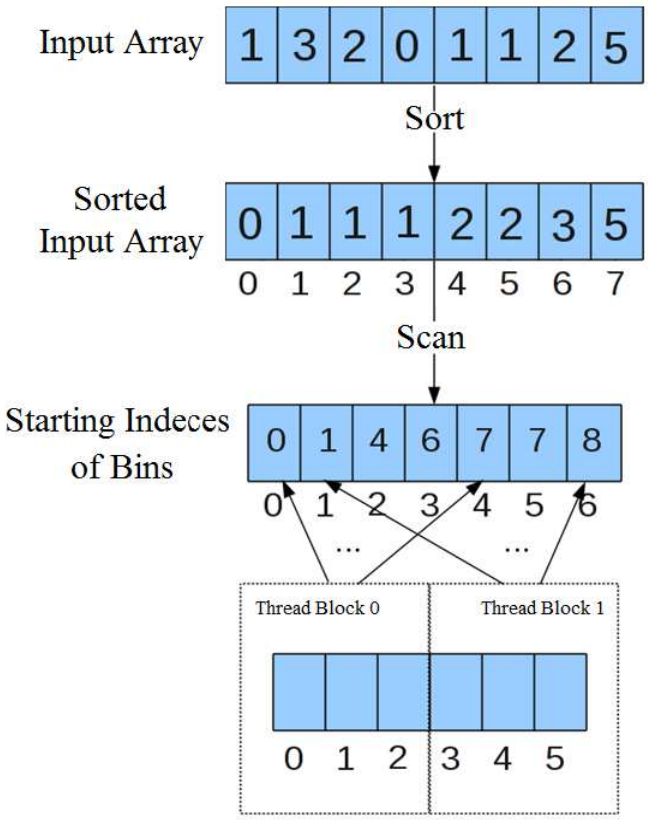


Figure 3: Sort-Reduce Variant of Binning

5. Implementation Details

This section details the implementation of the “sort-reduce” algorithmic variant of binning in the context of the MRI reconstruction application as provided in the ancillary lab material. Each of the steps described below, except the last, corresponds to a separate GPU kernel.

1) Spatial Hashing and Partitioning

The implementation proceeds with determining the closest bin to every input sample while maintaining this information in two interrelated arrays: one for the indices of the input samples, and the other for the indices of the bins to which they are mapped. A third array is used to keep track of the number of sample points mapped to each bin. If a bin reaches maximum capacity, any additional sample points intended for it are mapped to an overflow list to be processed on the CPU. This list thus serves as a single bin which spans the entire output grid.

2) Sorting

The bin indices array and the sample indices array from Step 1 are sorted as a key-value pair to congregate the indices of the samples belonging to the same bin.

3) Reordering

Using the sorted key-value arrays from Step 2, the sample points in the original input array are shuffled to form implicit bins. Sorting and reordering are broken into two steps because the sorting kernel only accepts values of the Integer data type.

4) Scanning

The implicit bins from Step 3 have variable capacities and thus their starting locations are unknown. A reduction operation is performed on the array of bin indices to tally the number of samples in each bin and determine its starting location.

5) Gridding

A Kaiser-Bessel function is used to construct the output grid from the binned input (Figure 4). The CPU and the GPU process their respective portions of the input concurrently, thereby exploiting the full computational power of the system besides utilizing the CPU to improve the load balance on the GPU.

```
for (every sample point s){
   for(z range){
      for(y range){
         for(x range){
            weight = kaiser_bessel(|<s.coords>-<x,y,z>|)
            grid[z][y][x] += s.value * weight;
         }
      }
   }
}
```

Figure 4: Pseudo-code of the Gridding Step

6) Merging

The partial results from the CPU and the GPU from Step 5 are merged into the final output grid.

6. Procedure

1) You will start with investigating the limitations of the binning approach of Lab 2B with non-uniformly distributed data. To this end, you are provided with an implementation of the MRI Reconstruction application which uses this approach. The implementation is fully functional, so you only need to familiarize yourself with the source code (Figure 5). You are given a default input set of **30144488 samples** which the application maps onto a **128x128x128 Cartesian grid**.

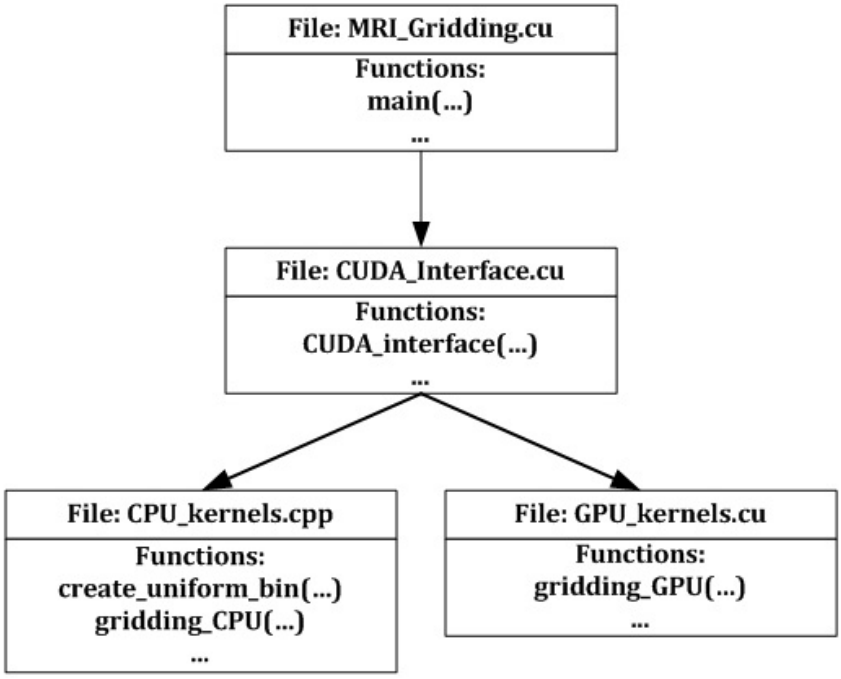


Figure 5: File Hierarchy of Lab 3.1

Run the application (Table 1) using different bin capacities and standard deviations and record the corresponding execution information (Table 2). The parameters field in the input description file specifies the bin capacity (first value in the field) and the standard deviation (second value).

Table 1

Source Code Path	`$PARBOIL_ROOT/benchmarks/3/src/3.1`
Input Description File Path	`$PARBOIL_ROOT/benchmarks/3/input/default/DESCRIPTION`
Compilation and Execution Command	`#cd PARBOIL_ROOT` `#./parboil run 3 3.1 default`

Table 2

Bin Capacity		Standard Deviation				
		16	32	48	64	80
16	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					
48	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					
80	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					
112	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					

Did you identify any patterns for the changes in the measured quantities? Can you explain such patterns? Keep in mind that larger standard deviations mean more uniformly-distributed data.

2) Next, you will explore the advantages of the binning algorithmic variant described in this lab. As a starting point, you are given a skeleton implementation (Figure 6) comprising five GPU kernels which correspond to the first five steps outlined in Section 5.

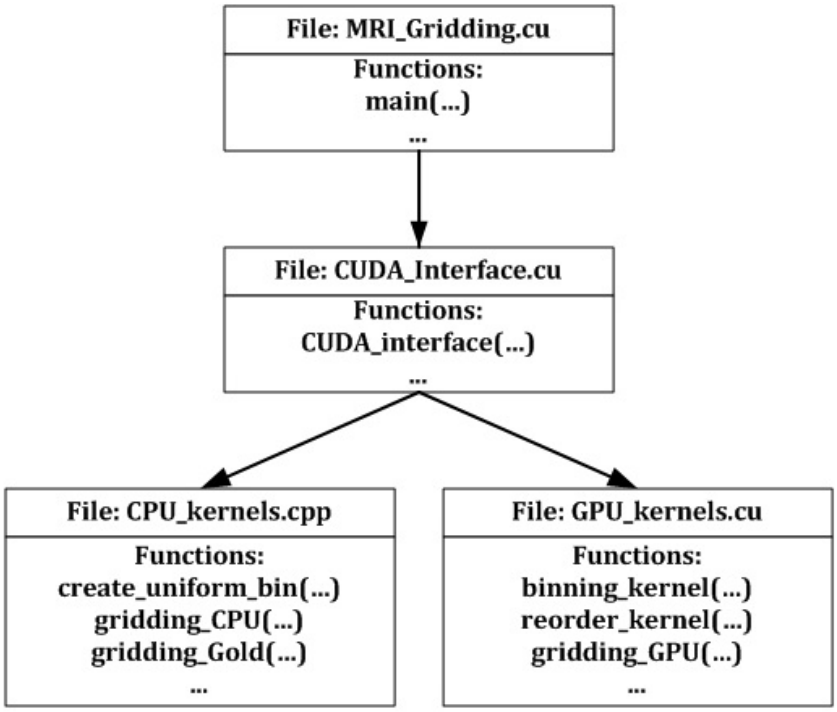


Figure 6: File Hierarchy of Lab 3.2

Complete the GPU kernels (Table 3) corresponding to **Step 1 (spatial hashing and partitioning)** and **Step 3 (reordering)** described in Section 5. The kernels corresponding to Step 2 (sorting), Step 4 (scanning), and Step 5 (gridding) have been already done for you. The correctness of your solution for a **bin capacity of 80** and **standard deviation of 16** will be confirmed with a **"Pass"** message.

Table 3

Source Code Path	`$PARBOIL_ROOT/benchmarks/3/src/3.2`
Source File to Modify	`GPU_kernels.cu`
Input Description File Path	`$PARBOIL_ROOT/benchmarks/3/input/default/DESCRIPTION`
Compilation and Execution Command	`#cd PARBOIL_ROOT` `#./parboil run 3 3.2 default`

Once again, run the application using different bin capacities and standard deviations and record the corresponding execution information (Table 4).

Table 4

Bin Capacity		Standard Deviation				
		16	32	48	64	80
80	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					
112	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					
144	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					
176	% of Samples Processed on CPU					
	% of Wasted Bin Locations					
	Binning Time					
	GPU Computing Time					
	Total Execution Time					

How do the results compare to those of Lab 3.1? Did you notice that bins with larger capacities can be used?

7. Questions

1) What are the limitations of the binning approach of Lab 2B when used with non-uniform data distributions? How does the algorithmic variant presented in this lab circumvent these limitations?

2) In this lab, the MRI Reconstruction application is parallelized by dividing the output points across the parallel threads of execution. Conversely, the application can be parallelized by dividing the input samples across the threads. Since a single input sample affects multiple output points in its vicinity, however, atomic operations are required for correct execution in this case. While atomic operations typically induce performance penalties, such a parallelism structure still proves to be more efficient for certain data distributions. Do you expect this observation to materialize towards the uniform or the non-uniform end of the data distribution? Try to validate your conclusion by implementing this parallelism structure and running the resultant kernels.