# A case study in CUDA programming: Running a statistical test on a distribution of galaxies

Oskar Lappi, 37146

February 2020

# Preface

This report is written for the 2019-2020 course *GPU Programming* at Åbo Akademi, organized by Jan Westerholm. The assignment is to write a CUDA program that calculates a statistic measure on two distributions of coordinates. The coordinates happen to be galactic coordinates and the result may support the existence of dark matter in the universe, but this problem context is not the object of study — the purpose of the assignment is to write an efficient program that runs on a GPU.

I chose to write a theoretical **Part I** so that I could better reason about the program and make educated decisions while designing it. Just jump straight to **part III** if you're familiar with CUDA and interested in the work and results.

The report is divided into four parts:

**Part I** describes the CUDA programming model.

**Part II** describes the assignment, the data to be processed, and the work that the program has to do.

**Part III** contains all the practical work, including the final design, steps I took to get there, and the results.

**Part IV** contains my final thoughts and conclusions, I try to derive some rules of thumb from the work.

## Definitions

In this report, I often use CUDA as synonymous with the CUDA C++ language extensions. E.g. when I write ”CUDA program” I refer to a program written in C++ with CUDA extensions. The report is written with CUDA Compute Capability 7.0 in mind.

## Typographic conventions

- Terms and titles are in *italic* the first time they appear
- Text in a CUDA program appears `like this`
- Programs and command line options appear `like this`

# Part I
# The CUDA programming model

## 1 Dramatis personæ: The host and the device

The CUDA programming model is *heterogeneous*. This means the processors that a CUDA program runs on may have different architectures.

A CUDA program runs on one processor (the CPU), which is called the *host*. Coprocessors (typically GPUs) to the host are called *devices*. In a CUDA program, the host is explicitly instructed by the programmer to run program segments on a device. The work given to the devices typically constists of the heaviest workloads in the program — the host holds the reins and the devices pull the cart.

The host and device are logically separated from each other. They have separate symbol tables, they operate on separate instruction streams, they even have separate memory.

### 1.1 Kernels

The segments of a CUDA program that will run on a device are called *kernels*. A kernel is defined as a function decorated with the `__global__` *execution space specifier* that will be called from the host.

### 1.2 Asynchronous execution

Memory transfers and computation on the host and device use resources which are independent from each others'. To use this to our advantage, the CUDA programming model allows for organizing memory management and execution calls into *streams*. All operations within a stream are performed synchronously, but operations in separate streams are (essentially) asynchronous and can execute . I used streams but they weren't terribly useful.

### 1.3 Runtime of a typical CUDA program

A CUDA program begins its runtime on the host with a call to main, just like any other C++ program. The host manages memory on both itself and devices. The host may also copy memory from itself to the host and vv. In any nontrivial CUDA program, the host will launch a kernel on the device, specifying how many threads should be executed with which block and grid dimensions.

# 2 Units of computation

## 2.1 Warp

A warp is a set of 32 threads (on modern NVidia GPUs), and it is generally the smallest unit of threads worth reasoning about. The warp differs from other units of computation in that it is managed by the hardware, not the programmer.

Every thread in a warp is on an individual *warp lane*, an ID ranging from 0 to 31. The lane is the threads 1D index modulo 32. The 1D index is calculated as $z_i|X \times Y| + y_i|X| + x_i$.

Threads in a warp used to be run simultaneously on the GPU on a single multiprocessor, but they are now independent as of compute capability 7.0. This is not the only way that threads in a warp are special, a GPU programmer can use *warp level programming* to create a more efficient program. E.g. memory access patterns, as I discuss in 4.1.1 and 5.3.

## 2.2 Thread blocks

Threads are logically grouped into blocks. Every thread in a block has a 3D block-index. The dimensions of the block index are up to the programmer to decide (1D and 2D indexing space have a widht of 1 for superfluous dimensions). All threads in a block witll be local to one multiprocessor. Threads within a block can share memory through *shared memory*, and synchronize at low cost with `__syncthreads()`. Threads within a block are executed in any order, but one warp at a time.

## 2.3 Grids

Blocks are further grouped into a grid. A block has a 3D index within a grid (just like threads do within blocks). Blocks within a grid can be executed in any order.

## 2.4 Cooperative groups

Since CUDA 9.0, threads can also be synchronized at other levels than thread blocks. A programmer can define a cooperative group of threads within or across thread blocks. The functionality is exposed in the header `<cooperative_groups.h>`.
We can start from the block or the grid by creating a cooperative group out of them:

- `thread_block b = this_thread_block();`
- `grid_group g = this_grid();`

The thread block can then be split into smaller groups of 8 with a call to:

```
thread_group tile = tiled_partition(b,8)
```

(I found little reference to whether a grid can be split into smaller pieces.) These structures offer a unified interface for synchronization and indexing of threads within a group. The interface of `thread_group` is fairly simple:

- `sync()` to synchronize the group
- `thread_rank()` gives a unique id within the group from 0 to size -1
- `size()` returns the size of the group

# 3 Physical memory layout

The useful distinction of where memory is physically mapped are: *off-chip* and *on-chip*. Both physical memory spaces are on the device, but on-chip memory is on the same chip as the multiprocessors. On-chip memory can additionally be split (on a physical level) into registers and shared memory.

In addition to this split there are on-chip (per multiprocessor) caches that make reads quicker.

- constant, read only cache
- unified data cache (128KB for Volta)
  - L1
  - textures
  - shared memory
- L2 cache

# 4 Globally scoped memory

Device memory can be categorized into two different classes:

- Dynamically allocated, globally scoped memory. Off-chip (but may be cached on-chip)
- Statically allocated, locally scoped memory. On-chip (except for local memory)

In general the program must allocate memory to the global, constant, and texture memory segments. Shared memory, registers, and local memory can be statically determined at compile time.

I first discuss globally scoped segments of memory, and then the locally scoped ones.

Global memory is read write; constant and texture memory is read only.

## 4.1 Global memory

Global memory is the biggest chunk of memory available. It is also in the slowest category.

Global memory is cached in L2.

### 4.1.1 Global memory access within a warp

Global memory operations within a warp are *coalesced*. For compute capability $\geq 6.0$, memory access will be grouped into 32-byte transactions. Even though L1-caching is enabled by default, transactions will still be units of 32 bytes.

## 4.2 Constant memory

Constant memory is off-chip read-only memory. Constant memory is cached in the *constant cache*, but cache misses will read all the way from the off-chip device memory. Constant memory is a good candidate when all threads in a warp access the same location in constant memory, because reads within a warp are only performed once for each unique address in constant memory. Such cases are arguments to kernels, and constants that can't be put in instructions. It's mostly used by the compiler, but programmers can use it to load one or two constants into if they want to.

Constant memory reads are however at best the same cost as register reads, and there is rather little of it available (some 64 kB), so it is a bad substitute for global memory. Best to leave constant memory to the compiler.

## 4.3 Texture memory

The caching of texture memory in the *unified data cache* is 2D-spatially local. Texture memory can be preferrable to global memory when processing 2D-spatially local data. I did not conider using texture memory for this assignment.

# 5 Locally scoped memory

Memory within kernel or device functions are locally scoped, and have a lifetime limited by the lifetime of the scope. Unlike the persistent pointers allocated by globally scoped memory, these memory segments are on-chip (except for local memory, which is mapped to off-chip memory).

## 5.1 Single thread scope

All declared variables in a kernel or device function will be scoped to a single thread, unless they are declared `__shared__`.

### 5.1.1 Registers

Registers are the fastest memory available, generally requiring 0 clock cycles.Registers will be partitioned along concurrent threads. The amount of registers allocated per thread can be controlled using the `-maxrregcount=N` option to nvcc, or the `__launch_bounds__()` qualifier. The register count per thread will limit the number of blocks per multiprocessor. More blocks is better (because it minimizes warp latency), but more register use is better locally (because it's faster).

### 5.1.2 Local memory

Local memory is not locally situated, it is off chip (on the device but not near a multiprocessor). This means local memory is slower than other types of locally scoped memory.

Since this is true, local memory is only allocated in special cases. The compiler will attempt to use registers as much as it can, but in some cases, it will allocate to local (off-chip) memory:

- non-constant sized arrays will be put in local memory
- if there aren't enough registers left, variables will be put in local memory (*register spilling*)

Since this behaviour is implicit, and we want to minimize it, it is best to check. You can measure the usage of local memory, as described in section 6.

## 5.2 Shared memory

Shared memory has to be explicitly declared by a programmer with the `__shared__` decorator. Shared memory is grafted out of the on-chip *unified data cache* (in fact, CUDA refers to it as a *carveout* from the unified data cache). The remainder of the udc is used as and L1 cache.

### 5.2.1 Bank conflicts

Shared memory is organized into banks, which can be simultaneously accessed. But access to a single bank by many theads will be serialized. Banks are spread out 32-bit.

### 5.3 Warp level memory constructs

While not explicit, there is a warp level memory space. A program can perform operations on registers used by a warp using *warp shuffle functions* and *warp matrix functions*.

These operations use registers assigned accross warp lanes in a coordinated way.

```
T __shfl_up_sync(unsigned mask, T var, unsigned int delta)
```

will return the value of `var` in the lane with id `delta` less than the current lane id. These accesses happen simultaneously. Since registers are quicker than shared memory, this is a faster way to share memory accross warps.

## 6 Checking and measuring memory use

### 6.1 From source code

The so called *address space predicate functions* can be used determine the memory space of a pointer. The all have the signature `unsigned int ... (const void *ptr)`

- `__isGlobal`
- `__isShared`
- `__isConstant`
- `__isLocal`

### 6.2 From the compiler

nvcc reports total local memory usage, register usage, shared memory usage, etc. when run using the `–ptxas-options=-v`.

# 7 Making things parallel

## 7.1 Atomic functions

CUDA provides atomic functions for performing arithmetic and logical operations atomically. These operations are atomic because they lock the address being modified in the operation. These operations may be atomic on a block, device, or system level.

## 7.2 Overlap

As I described in section 1.2, host, kernels, and memcpy are asynchronous with regard to each other. If we had a lot of data transfer to do, we could interleave that with kernel runs. We can also interleave host and device operations, running them in parallel. To do this we may have to allocate pinned memory and use streams.

# Part II
# The problem

## 8 Background

The hypothesis of the existence of dark matter in the universe has been made because traditional astrophysical models fo not explain astrophysical observations. Therefore — the astrophysicists argue — there must be some unobserved and hitherto unobservable (or dark) quantities that cause the observable quantities in the universe to behave unlike what the astrophysical models predict.

We are tasked with running a statistical test to support this claim, using empirically measured and randomly generated. The specific models and measurements we will consider are:

- models of gravitation
- the position of galaxies

If the models and initial state are valid, the two data sets should be ”similar”. If not, then either the models are wrong, or the modeled state is incomplete. Since there is supporting evidence for the models being right, it’s more likely that there is some quantities missing from the state. (This is my understanding of the logic of the argument)

So, what does the data look like, and what does ”similar” mean? We look at these in sections 9 and 10 respectively.

## 9 Data

There are two data sets with 100 000 coordinates each. One contains real **D**ata, the other **R**andomly generated data. We call these the **D** and **R** data sets.

The program will have to load these from a file when the program is run. The data in both consists of galactic coordinates in the form of right ascension and declination. I will transform this data into cartesian 3D points.

**My program will also exploit** the fact that the points all lie within $90^\circ$ of each other.

## 10 The work

We will consider the cartesian products: $D \times D$, $R \times R$, and $D \times R$. For each set of pairs, we will calculate the angle between each pair and generate the histograms: **DD**,**RR**, **DR**. Each histogram should have a resolution of a quarter of a degree.

To test the hypothesis, we will calculate a statistical measure, which is part of a statistical test. I don’t know what the test actually *is*, or the test statistic, the statistic test in the context of this report simply determines the work that has to be done.

The statistical test described above is calculated as:

$$\omega_i = (DD_i - 2DR_i + RR_i)/RR_i)$$

If $omega_i$ is greater than, say, 0.5, this means that the distributions are dissimilar for the bin $i$.

The greater part of the work will be calculating the angles and histograms. Calculating the test statistic is several orders of magnitude simpler.

# Part III
# Design, Implementation and validation

## 11 Overview

The overview of the program is as follows:

1. load data from files
2. transform into cartesian coordinates
3. move data to the GPU
4. calculate angles between pair of coordinates
5. increment the corresponding histogram bin for each angle
6. move data from the GPU
7. calculate the statistical measure

List item 4 and 5 on the above list are performed on the device, while the rest is done on the host. This is because there are 30 000 000 000 angles to calculate, while there are only 200 000 coordinates, and 1080 bins.

### 11.1 Process

The version which I present to you is about the third iteration. I had some trouble early on getting good performance out of the program, but I am somewhat satisfied with this version.

I used `nvcc` to compile the program, `time` to time the program, `nvprof` to profile, and I tried to debug with `cuda-gdb`, but that didn't work so well. In order to profile host code, I used the NVIDIA Tools Extension Library. I used make and git to organize my work on dione, which is where I eventually did all of my development. My makefile is on the next page.

```
CXX=/usr/local/cuda/bin/nvcc
CPPFLAGS=-lm -arch=compute_70 -code=sm_70,sm_72 -lineinfo
   --ptxas-options=-v -lnvToolsExt
srun_flags=-p gpu -o log/out.log -e log/error.log
jobid_find=s/.*job \([[:digit:]]*\) queued.*/\1/p
REAL=data/real/data_100k_arcmin.txt
FAKE=data/real/flat_100k_arcmin.txt

all : load_cuda darkmatter

darkmatter: src/darkmatter.cu src/*.h
        $(CXX) $(CPPFLAGS) -o $@ src/darkmatter.cu

.PHONY: run
run: darkmatter
        @mkdir -p log
        $(eval PROG_ID=$(shell /bin/bash -c ”srun $(srun_flags) $^
           $(REAL) $(FAKE) 2>&1 | tee /dev/tty |sed -n
           ’$(jobid_find)’”))
        @./report_run.sh $(PROG_ID)

.PHONY: debug
debug: darkmatter
        @mkdir -p log
        srun -p gpu --pty cuda-gdb --args $^ $(REAL) $(FAKE)

.PHONY: time
time: darkmatter
        srun -p gpu --pty time ./$^ $(REAL) $(FAKE)

.PHONY: prof
prof: darkmatter
        srun -p gpu --pty nvprof ./$^ $(REAL) $(FAKE)

.PHONY: vprof
vprof: darkmatter
        srun -p gpu --pty nvprof -f -o darkmatter.prof
           --cpu-profiling on ./$^ $(REAL) $(FAKE)

.PHONY : clean
clean:
        rm -rf build
        rm -f src/**/*.o
        rm -rf  log
```

Listing 1: the project makefile

# 12 Analysis

## 12.1 What device is available to us

Using the cudaGetDeviceProperties function from the runtime API, we get the following information about a GPU partition on the SLURM cluster dione.utu.fi

```
Device Number: 0
  Compute capability version:   7.0
  Device name:                  Tesla V100-PCIE-16GB
  Memory Clock Rate (KHz):      877000
  Memory Bus Width (bits):      4096
  Peak Memory Bandwidth (GB/s): 898.048000


  Total global memory:          16945512448 bytes
  Total constant memory:        65536 bytes
  L2 cache size:                6291456 bytes
  Warpsize:                     32
  32bit regs per block:         65536
  Shared memory per block:      49152 bytes

  Max #threads per block:       1024
  Max block dims (in threads):  1024 x 1024 x 64
  Max grid size:                2147483647 x 65535 x 65535
  #Multiprocessors:             80
  #Threads per multiprocessor:  2048

  ECC:                          1
  Concurrent kernel support:    1
  Unified addressing:           1
  Memcpy/kernel concurrency support(asyncEngineCount): 7
----
Device 0 can access device 0’s memory = 0
Device 0 can access device 1’s memory = 1
Device 0 can access device 2’s memory = 1
Device 0 can access device 3’s memory = 1
```

These numbers were very useful to me when I designed the program.

## 12.2 Optimal numbers

### 12.2.1 Histogram size

As I alluded to earlier, the maximum angle between two coordinates in the datasets was 90 degrees. This means we only need a histogram with $90 \cdot 4 = 360$ bins.

### 12.2.2 Block size

For each of the histograms, we are calculating an angle for each pair in $Z_{10^5} \times Z_{10^5}$. Luckily for us $10^5 \mid 2^5$ and $10^{10} \mid 2^{10}$, and $2^{10} = 1024$ happens to be the maximum number of threads in a block. Since $2^{10}$ is a square, I will make my blocks 2-dimensional, 32 by 32. This can be used for an elegant mapping from memory to computation.

#### 12.2.3 # of registers

There is a total of 65536 registers. We have 1024 threads in a block. Each thread can therefore use at most $65536/1024 = 64$ registers, and preferably 32, 21, or 16, since this would allow us to pack 2 and 4 blocks into one multiprocessor respectively. Values just underneath these numbers would also be ok.

I have used a single kernel. The kernel both calculates angles and increments the corresponding bins. Each thread calculates one angle and increments one bin. The same kernel is run three times, with the arguments (D,D), (R,R), and (D,R).

### 12.3 Memory

We can imagine the cartesian product of the two datasets as a 100 000 by 100 000 matrix. Since my blocks are 32 by 32, we can map the blocks to 32 by 32 submatrices of this matrix. A subset of threads can fetch a value each from global memory. This means each block only has to fetch 64 values from global memory, and they can all be shared within the block to calculate 1024 results.

These 1024 results will be atomically added to a histogram within the block. Finally the block level histogram is added to a histogram in global memory.

We are performing more adds in total, but fewer to global memory by a factor of 1024.

### 12.4 Warp level

The most important thing to consider on a warp level for this program is that threads within a warp not diverge, as this can slow down the overall performance of the program.

Since the blocks are 32 by 32, the warp is identified by `threadIdx.y`, and the lane by `threadIdx.x`. If my ifs use `threadIdx.y` as the condition, and my loops contain as many whole warps as possible, the situation is optimal.

# 13 Design and Implementation

I list the interesting bits of the program below. First, definitions and host code, then device code. The host is responsible for:

- Preprocessing
- The calculation of the statistic
- File I/O (trivial, not presented)
- Validation (not profiled, not presented)

## 13.1 Macros and definitions

```
#define GOOD_ENOUGH_PI 3.14159
#define ARCMIN_CONV_RATE 0.000291
#define RAD2DEG_CONV_RATE 57.296
#define RAD2QUARTDEG_CONV_RATE 229.184

#define BIN_DIM 360
```

```
typedef struct galactic_coordinate {
  float right_ascension;
  float declination;
} galactic_coordinate;

typedef struct cart_coordinate {
  float x;
  float y;
  float z;
} cart_coordinate;

typedef unsigned long long BIN_TYPE;
```

`BIN_TYPE` is typedefed because I didn't want to type `uint64_t` or `unsigned long long` every time I refered to it. It also makes it easier to switch types, should the size of the dataset change.

## 13.2 Preprocessing

Preprocessing takes care of transforming the coordinates from spherical coordinates to cartesian coordinates.

```
void preprocess_cart(
  galactic_coordinate coords_in[],
  cart_coordinate coords_out[],
  int n_coords)
{
  int i;
  for (i = 0;i < n_coords; i++) {

    float phi = coords_in[i].right_ascension*ARCMIN_CONV_RATE;
    float theta =
      GOOD_ENOUGH_PI/2 - coords_in[i].declination*ARCMIN_CONV_RATE;

    float sin_theta = sinf(theta);

    coords_out[i].x = sin_theta*cosf(phi);
    coords_out[i].y = sin_theta*sinf(phi);
    coords_out[i].z = cosf(theta);
  }
}
```

### 13.2.1 Calculating the statistic

```
void calc_kolmogorov(
  double *kolm_smir,
  BIN_TYPE *dr,
  BIN_TYPE *dd,
  BIN_TYPE *rr)
{
  for (int i = 0; i < 360; i++){
    kolm_smir[i] = ((double)dd[i] - 2*(double)dr[i] +
       (double)rr[i])/(double)rr[i];
    if (kolm_smir[i] > 10){
      printf("Bin %d suspiciously high: (%lld - 2*(%lld) + 
          %lld)/%lld = %lf\n",
         i,dd[i],dr[i],rr[i],rr[i],kolm_smir[i]);
    }
  }
}
```

## 13.3 The kernel

```
__global__ void calc_DR_bins(
  cart_coordinate *D,
  cart_coordinate *R,
  BIN_TYPE *bins)
{
  uint64_t t_id_in_block = threadIdx.y*blockDim.x+threadIdx.x;
  uint64_t x_angle_start = blockDim.x*blockIdx.x*ANGLE_PER_THREAD;
  uint64_t y_angle_start = blockDim.y*blockIdx.y*ANGLE_PER_THREAD;
  __shared__ uint32_t block_bins[BIN_DIM];
  __shared__ cart_coordinate shD[ANGLE_PER_THREAD*32];
  __shared__ cart_coordinate shR[ANGLE_PER_THREAD*32];

  if (threadIdx.y == 31)
    for (int i = threadIdx.x;
         i < 32*ANGLE_PER_THREAD && x_angle_start+i < 100000;i+=32
    )
      shD[i] = D[x_angle_start+i];
  else if (threadIdx.y == 30)
    for (int i = threadIdx.x;
         i < 32*ANGLE_PER_THREAD && y_angle_start+i < 100000;i+=32
    )
      shR[i] = R[y_angle_start+i];
  else if (t_id_in_block < BIN_DIM)
    block_bins[t_id_in_block] = 0;


  __syncthreads();
  for(int i = threadIdx.x;
      i <  32*ANGLE_PER_THREAD && x_angle_start+i < 100000;i+=32){
    for(int j = threadIdx.y;
        j < 32*ANGLE_PER_THREAD && y_angle_start+j < 100000;j+=32){

      uint16_t bin_id =(uint16_t) (
        acosf(
          shD[i].x*shR[j].x+
          shD[i].y*shR[j].y+
          shD[i].z*shR[j].z
        )*RAD2QUARTDEG_CONV_RATE);
      bin_id = MIN(bin_id,359);
      atomicAdd_block(&block_bins[bin_id],1);
  }}

  __syncthreads();

  if (t_id_in_block < BIN_DIM)
    atomicAdd(&bins[t_id_in_block],block_bins[t_id_in_block]);
}
```

Reads are written to be optimally coalesced in warps (and for warps not to diverge), but it's done in two warps only, since I don't know ahead of time how many angles per thread there are. Also, since the reads are conditioned on which warp a thread is in warps won't diverge, which is more efficient. The condition for clearing a bin is not exactly warp-specific, there will be one warp with mixed branching, but overall the condition will map to as many whole warps as possible.

By changing the `ANGLE_PER_THREAD` constant, we can decide how many angles are run per thread. Per thread means how many angles are fetched per thread, every thread actually calculates $ANGLE\_PER\_THREAD^2$ angles.

Every thread block is mapped to a segment in the complete matrix of pairs `D` × `R` consisting of $ANGLE\_PER\_THREAD^2$ counts of $32 \times 32$ submatrices. And every thread maps to an angle in the same position in each submatrix. The coordinate in the submatrix is the `threadIdx.x` for `D` and the `threadIdx.y` for `R`.

### 13.3.1 Writes

After the reading is done, every thread calculates its bins, and increments the thread blocks shared histogram. At the end, the same threads that zeroed the shared histogram will add the shared histogram to the histogram in global memory, one thread at a time.

## 13.4 The main function

```
int main(int argc, char *argv[]) {
  //...Allocations left out for legibility
  unsigned int grid_len
     =((3125)+ANGLE_PER_THREAD-1)/ANGLE_PER_THREAD;
  dim3 grid_dims(grid_len,grid_len);
  dim3 block_dims(32,32);

  cudaStream_t stream[2];
  for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

  //Read one file
  galactic_coordinate *raw_coords =
    (galactic_coordinate *)
       calloc(SET_DIM,sizeof(galactic_coordinate));
  read_dataset(raw_coords,argv[1],SET_DIM);
  preprocess_cart(raw_coords,real_coords,SET_DIM);

  //Start DD-calculation
  cudaMemcpyAsync(
        d_real_coords,real_coords,
        SET_DIM*sizeof(cart_coordinate),
        cudaMemcpyHostToDevice,
        stream[0]);
  calc_DR_bins<<<grid_dims, block_dims,0,stream[0]>>>
    (d_real_coords,d_real_coords,d_dd_bins);
  //...
```

```
//...

//While the first kernel runs, read second file
read_dataset(raw_coords,argv[2],SET_DIM);
preprocess_cart(raw_coords,real_coords+SET_DIM,SET_DIM);

//..and start the RR-calculation
cudaMemcpyAsync(
  d_fake_coords,real_coords+SET_DIM,
  SET_DIM*sizeof(cart_coordinate),
  cudaMemcpyHostToDevice,
  stream[1]);

//Make sure stream 1 has finished copying over R before running
   DR on stream 0
cudaStreamSynchronize(stream[1]);
calc_DR_bins<<<grid_dims, block_dims,0,stream[1]>>>
  (d_fake_coords,d_fake_coords,d_rr_bins);


//Finally, run the DR-calculation
calc_DR_bins<<<grid_dims, block_dims,0,stream[0]>>>
  (d_real_coords,d_fake_coords,d_dr_bins);

//Copy over bins
cudaMemcpyAsync(h_dd_bins,d_dd_bins,
      BIN_DIM*sizeof(BIN_TYPE),
      cudaMemcpyDeviceToHost);
cudaMemcpyAsync(h_rr_bins,d_rr_bins,
      BIN_DIM*sizeof(BIN_TYPE),
      cudaMemcpyDeviceToHost);
cudaMemcpyAsync(h_dr_bins,d_dr_bins,
      BIN_DIM*sizeof(BIN_TYPE),
      cudaMemcpyDeviceToHost);

for (int i = 0; i < 2; ++i)
  cudaStreamDestroy(stream[i]);

//Calculate result
double kolm_smir[BIN_DIM];
calc_kolmogorov(kolm_smir, h_dr_bins, h_dd_bins, h_rr_bins);
write_kolmogorov(”out/distribution.tsv”,kolm_smir,BIN_DIM);
//...free() left out for legibility
```

Listing 2: The main function

# 14 Metrics

Unfortunately, timing with `time` is a little unreliable, it depends greatly on the alloted CPU resources (CPU%). I think the profiles are better, but I'm not sure.

```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z12calc_DR_binsP15cart_coordinateS0_Py'
        for 'sm_70'
ptxas info    : Function properties for _Z12calc_DR_binsP15cart_coordinateS0_Py
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 22 registers, 5280 bytes smem, 376 bytes cmem[0], 8 bytes cmem[2]
```

Timing with `time`: angles_per_thread = 1:

```
0.37 user
0.53 system
0:00.99elapsed
91%CPU

0.18 user
0.40 system
0:00.64 elapsed
91%CPU
```

### 14.0.1 Profiles of version B

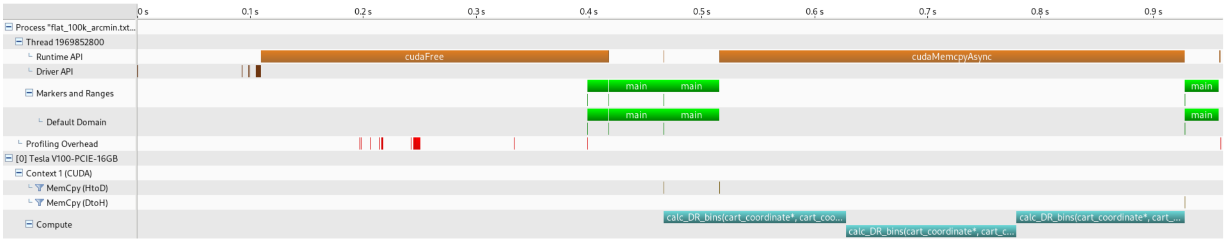

Figure 1: Profile of version B, with angles_per_thread = 1

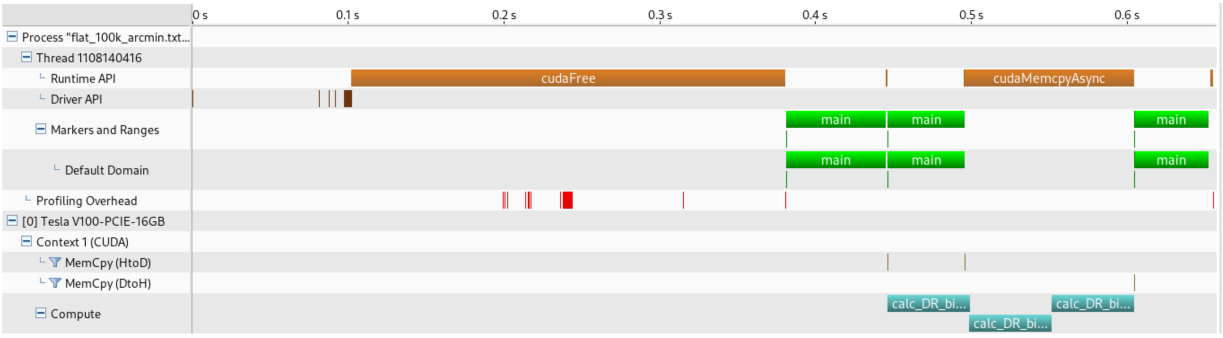

Figure 2: Profile of version B, with angles_per_thread = 5

The second version is faster by far, it appears that letting each thread do more work is better. Or perhaps the relieved pressure on the internal memory bus is causing the gain in performance. Here we can see how the host code (green) is interleaved with the kernels (blue). There is also some small interleaving among the kernels themselves. I've added the host sections using the NVIDIA Tools Extension library.

# Part IV
# Conclusion

## 15 Discussion

I am happy with the results, the computation takes less than a second ($\approx$ 640ms). This was a very fun project which I learned a lot from. I am now both a better CUDA and C/C++ programmer. I also understand the GPU mode of computation much better now.

Another thing I enjoyed was profiling my application, and partly competing against my peers in the course. I am now much more proficient at profiling.

Some things I still wonder about: A significant portion of this is spent initializing the CUDA context $\approx$ 390ms! I wish I could speed this up somehow, but I suspect I can't. If only we could preseed this context somehow, to save time. Even though I compiled the program for the sm_70 (and even sm_72 just to be sure) architectures, I found calls to JIT-compilers in the profiles. I don't understand this at all.

## 16 Rules of thumb and other lessons learned

- Preprocess data at the scale of the data, instead of processing it at the scale of the work later
- Maximize occupancy by keeping warps "intact" (not divergent) by using warp boundaries as conditions for divergent behaviour
- Load data from global memory to shared memory before working on it (if many threads in a block need the same data)
- **In a kernel**: load all data; synchronize; do work; synchronize; write all data
- Allocate to pinned memory for faster data transfer (only ˜1 MiB for this assignment, so did not see a significant effect)
- Don't parallelize too much, sometimes it's better to have fewer kernels doing more work
- Interleave host and device computation as much as possible
- There is an interesting dynamic when optimising heterogenous parallel code. If a synchronization depends on a procedure completing on both the host and the device, then optimising the faster procedure isn't worth it. You have to pull in the two parallel procedures in tandem, or one at a time, always taming the wilder of the two horses.
- Finally, gather as much data as you need to understand the optimisation problem. Testing different configurations help you develop an intuition for the dynamic at play

## 17 Improvements

I could still write a calc_DD_bins kernel to use the fact that some angles are duplicate in the DD and RR histograms. I could try to use cudaMemPrefetchAsync or a pointer to the pinned host memory for asynchronous behaviour. However, since data transfers are cheap for this assignment, it wouldn't make a difference. I could run the program on several devices and maybe tops double the speed of the kernel execution. This may however cause a longer initialization sequence, and may ultimately not be worth it.