AMS 148 Chapter 4: Precision Support, Matrix Operations, and Stencil

Steven Reeves

1 Floating Point Precision

Floating point numbers come in a variety of precisions. Essentially, the amount of data each float carries defines how precise the number is. CUDA takes the floating point types that its base language has. So for CUDA C/C++ the floating point types are:

- half, 16 bit floating point type, carries 4 digits
- float, 32 bit floating point type, carries 8 digits
- double, 64 bit floating point type, carries 16 digits

These floating point types are defined by the IEEE 754 standard.

1.1 What are floats?

Floating point data types are formulaic representation of real numbers as an approximation. That is, technically, computers only really understand rational numbers. The precision of a float gives the relative "goodness" of a real number approximation. Take for instance π, the most loved irrational number. A computer will never be able to fully represent this number. Instead it is represented by a floating point number. So for each precision π can be represented as

$$\pi = \pi_{mach} + \mathcal{O}(\epsilon_{mach})$$

here ϵ_{mach} is the precision cut-off, or **machine precision**. This machine precision depends on the floating point type the user chooses. Let's write out π in terms of our 3 main floating point types.

- half, $\pi_{mach} = 3.141$
- float, $\pi_{mach} = 3.141592$
- double, $\pi_{mach} = 3.141592653589793$

Formally, the computer understands floating point numbers as a combination of **exponent** and **mantissa**(significand). The exponent is stored as a binary representation of an integer, as well as the mantissa. So if we wanted to represent the half-precision π using the matissa and exponent we would have:

$$3.141 = \underbrace{3141}_{\text{mantissa}} \times 10^{\overbrace{-3}^{\text{exponent}}}$$

.

1.2 Half Precision

Half precision floating point numbers are a fairly recent development. These are typically useful for some types of deep-learning applications. However, not all GPUs support half precision, and some that do may not be particularly performant. If you find that your application warrants the use of half precision, you will probably need to use a Pascal architecture *Tesla* series GPU, like the ones found in Hummingbird. Any GPU that was made before the Pascal architecture, like Kepler or Maxwell, will not have half precision support. Additionally, other types of Pascal GPUs will be able to understand half precision arguements, but their half precision performance is greatly reduced. These GPUs are often optimized for gaming *GeForce* or graphics rendering *Quaddro*, and will usually only excel single precision floats.

To use half precision floats in C++ we need to download the library from sourceforge, and include the correct header. For example:

```
#include <half.hpp>
int main(){
        using half_float::half;
        half pi(3.141);
        std::cout<<"This is half precision pi!"<<pi<<std::endl;
        }
```

The header `half.hpp` contains all the basic math functions overloaded for half precision arguments. For more information, read the documentation included with in the sourceforge site. This library is C++98 compatible, but runs best on C++11 or higher.

In CUDA, we'll need to use `cuda_fp16.h`. With this we can use the `half` data type directly. **Note this only works for CUDA 7.5 or higher!**

1.3 Single Precision

Single precision, or 32-bit floats are still GPUs' bread and butter. No extra library is needed to load the standard float, and any GPU will be performant for single precision floats. Single precision floating point data is useful in image processing, machine learning, and some scientific applications.

1.4 Double Precision

Double precision, 64-bit floats were not supported by GPU's the first Tesla branded GPU was invented. Double precision is the standard when it comes to most scientific applications. The Tesla range was specifically designed to be used for scientific and machine learning applications. However, some double precision capabilities have leaked into the other ranges. However, in GeForce and Quaddro, the double precision capabilities do not mirror their Tesla counterparts - since those GPUs are optimized for actual graphics.

1.5 Mixed Precision Computing

Suppose that you're a science student, who plays some games on your computer with your Nvidia Graphics card. You find yourself in a position where your research code would benefit from GPU acceleration but your graphics card doesn't have strong double precision performance. You want double precision accuracy, but have loads of single precision performance. An example would be the addition: double(a) + double(float(b) + float(c)).

You can simulate a double using two floats for addition, or four floats for multiplication. So we can define a double adder as a two float adder, and a double multiplier as a four float multiplier. You need to be careful of how you do your operations. In this case, round off error can destroy the accuracy of your program if you are not careful. For more information on this you can contact me in my office hours, this would be a good project!

1.6 Choice of Precision

You application may require a specific type of precision, more or less. This will dictate the type of GPU that you can feasible use. If you have a double precision application, you will probably need a Tesla series GPU, or the new Titan V. For half precision, you will also need a Tesla GPU. For plain single precision, you will

be able to use any Nvidia GPU. So if you don't need double precision, use a float, your application will be much faster and more portable to different architectures. If you find yourself doing deep learning, then you probably are well funded and have access to a half precision compatible GPU - luckily we do! And, if you're clever and don't have access to a double precision performer, you can develop your code to take advantage of mixed precision programming.

2 Matrix Operations in CUDA

In the previous homework were wrote a CUDA matrix transpose, and matrix-vector multiplication. As an example, we did matrix addition. In this section we will tackle a more complex matrix operation - matrix multiplication. To do this effectively we'll need a couple tools:

- tiling
- shared memory
- Matrix struct

But first, we'll look into a naive approach.

2.1 Naive Matrix Multiplication

The simplest way to do Matrix Multiplication in CUDA would be to examine the serial implementation and expose concurrency. Suppose we have two matrices, $\mathbf{A}, \mathbf{B}$ where $\mathbf{A}$ is $N \times M$ and $\mathbf{B}$ is $M \times L$. Then, $\mathbf{C} = \mathbf{AB}$ is $N \times L$. To write out the serial algorithm, recall that $C_{i,j} = \mathbf{a}_i^T \mathbf{b}_j$ where $i \leq N$ and $j \leq L$.

Algorithm 1: A sequential Matrix multiply

```
   Data: A, B
   Result: C
1  for i = 0 → N − 1 do
2  |  for j = 0 → L − 1 do
3  |  |  summ = 0;
4  |  |  for k = 0 → M − 1 do
5  |  |  |  summ+= a_ik b_kj;
6  |  |  end
7  |  |  c_ij = summ;
8  |  end
9  end
```

we see that the sum over k is the dot product in action. Figure 1 illustrates this product pictorally.

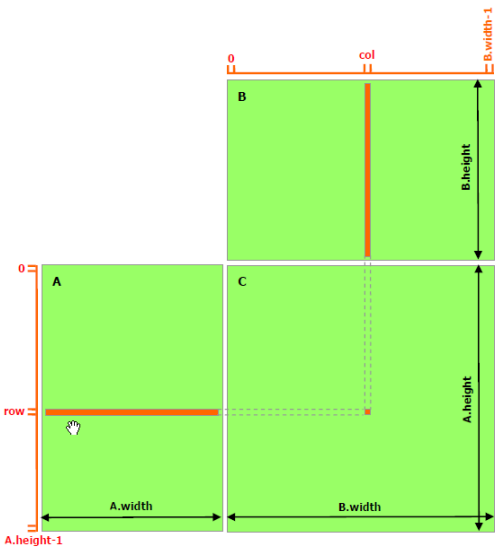

Figure 1: Visual Representation of Matrix Multiplication, CUDA Programming Guide

Often it is advantageous to examine an algorithm graphically. When the matrix multiplication is visuallized it is easy to see how one can make it parallelized. Each element depends only on a row of **A** and a column of **B**. Therefore, we can distribute the calculations over the rows of **A** and the columns of **B**.

When doing matrix multiplication it is convenient to create a structure called *Matrix*.

```
typedef struct{
        int width;
        int height;
        float* elements;
} Matrix;
```

This makes the kernel call cleaner, and allows direct access to the elements, as well as the dimensions of our matrix. Asuming we use this struct, let us construct the CUDA kernel for execution.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
        // Each Thread computes one element of C
        // by accumulating results into Cvalue
        float Cvalue = 0.0f;
        int row = blockIdx.y*blockDim.y + threadIdx.y;
        int col = blockIdx.x*blockDim.x + threadIdx.x;
        for (int e = 0; e<A.width; e++)
                Cvalue += A.elements[row*A.width + e]*B.elements[e*Bwidth + col];
        C.elements[row*C.width + col] = Cvalue;
}
```

In this implementation, everything is stored in global memory. Think about how many times an element is called from memory. For large matrices, this will certainly be faster than the triple looped serial implementation, however, memory transactions will bog it down from it's true potential. This seems like a job for the trusty shared memory.

2.2 Shared Memory Matrix Multiplication

The following code is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub matrix $\mathbf{C}_{sub}$ of **C**, and each thread within the block is responsible for computing one element of the submatrix. Again, let's look at this pictorally. Figure 2 illustrates the tiled matrix multiplication using shared memory.

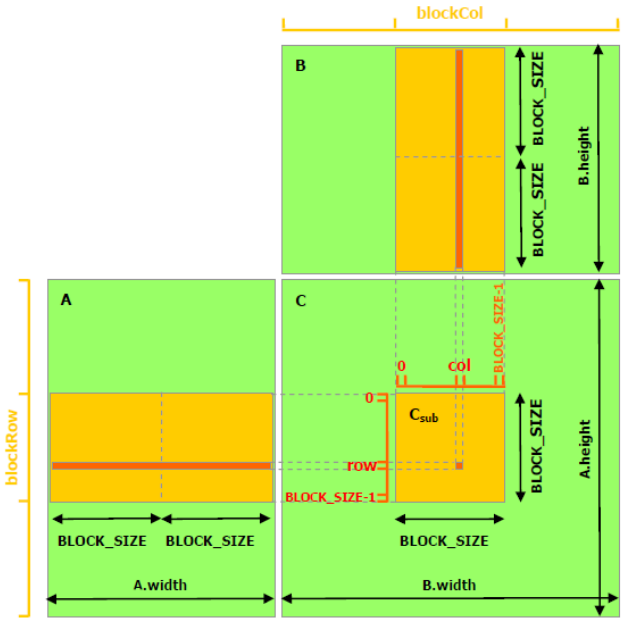

Figure 2: Visual Representation of Matrix Multiplication using tiling with shared memory, CUDA Programming Guide

In figure 2, the gold bands represents the data in shared memory, and the block in the matrix $\mathbf{C}$ is the tile. We can write the tile $\mathbf{C}_{sub} = \mathbf{A}_{sub}\mathbf{B}_{sub}$, where $\mathbf{A}_{sub}$ is of dimesion `A.width, block_size` and $\mathbf{B}_{sub}$ is of size `B.height,block_size`. Just like in the naive case, $\mathbf{C}_{sub}$ has the row indices of $\mathbf{A}_sub$, and the column indices of $\mathbf{B}_{sub}$.

For the shared memeory kernel we modify our Matrix struct.

```
typedef struct{
        int width;
        int height;
        int stride;
        float* elements;
} Matrix;
```

In this case we'll need an additional sizing element, `stride`. Also for this implementation we will create a couple `__device__` functions. The first will be a fetching algorithm for elements.

```
__device__ float GetElement(const Matrix A, int row, int col)
        {
                return A.elements[row*A.stride + col];
        }
```

Next we define a function to set a matrix element.

```
__device__ void SetElement(Matrix A, int row, int col, float value)
        {
                 A.elements[row * A.stride + col] = value;
        }
```

Lastly we write a function to select a submatrix

```
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
        {
                Matrix Asub;
                Asub.width = BLOCK_SIZE;
                Asub.height = BLOCK_SIZE;
                Asub.stride = A.stride;
                Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
                return Asub;
         }
```

The main reasoning for these functions is to keep the matrix multiplication modular, and clean.

Now we need to write our kernel. In this instance we know *explicitly* how much shared memory we need, so we will imploy the static shared memory model.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
        // Block row and column;
        int blockRow = blockIdx.y;
        int blockCol = blockIdx.x;

        //Thread block computes one sub matrix Csub of C
        Matrix Csub = GetSubmatrix(C, blockRow, blockCol);

        // Each thread computes one element of Csub
        // By accumulating results into Cvalue
        float Cvalue = 0.0f;

        //Thread row and column index within the submatrix
        int row = threadIdx.y;
        int col = threadIdx.x;

        // Loop over submatrices of A and B that are required for Csub
        //Multiply each pair of sub-matrices together
        //and summ the results
        for (int m = 0; m < (A.width/BLOCK_SIZE); m++){

                //Get A submatrix
                Matrix Asub = GetSubMatrix(A, blockRow, m);

                //Get B submatrix
                Matrix Bsub = GetSubMatrix(B, m ,blockCol);

                //Static shared memory for Asub and Bsub
                __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
                __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE]; //Great name for an array

                //Load Asub and Bsub from global memory into shared;

                As[row][col] = GetElement(Asub,row,col);
                Bs[row][col] = GetElement(Bsub,row,col);

                //Always sync threads when loading shared memory before doing computation
                __syncthreads();

                //Multiply the submatrices
                for (int e = 0; e < BLOCK_SIZE; e++)
                        Cvalue += As[row][e]*Bs[e][col];

                //synchronize to make sure all threads are done computing
                __synchthreads();
        }
        //write Csub back into global memory
        //each thread writes one element
        SetElement(Csub, row, col, Cvalue);
}
```

Recall that in figure 2 the submatrices of **A** and **B** may be multiple block sizes tall or wide, this requires us to loop over these sections. This technique is called tiling, and is a form of domain decomposition.

2.3 Is it worth it?

As you can see, it is a lot more work to use shared memory in our algorithms. Is it worth it? All we need to do to test the fruit of our labour is to check the timing.

	Serial	OpenMP	Naive CUDA	Shared Memory CUDA
$N = 32$	1.72×10^{-4}	2.102×10^{-3}	2.2×10^{-5}	1.1×10^{-5}
$N = 64$	6.15×10^{-4}	2.19×10^{-3}	2.6×10^{-5}	1.4×10^{-5}
$N = 128$	6.39×10^{-3}	3.19×10^{-3}	3.9×10^{-5}	1.6×10^{-5}
$N = 256$	5.51×10^{-2}	1.96×10^{-2}	1.43×10^{-4}	7.4×10^{-5}
$N = 512$	5.35×10^{-1}	1.58×10^{-1}	7.35×10^{-4}	2.24×10^{-4}
$N = 1024$	3.60713	1.52667	5.794×10^{-3}	1.545×10^{-3}
$N = 2048$	111.053	38.3684	4.6233×10^{-2}	1.2963×10^{-2}
$N = 4096$	—	—	3.45668×10^{-1}	7.2939×10^{-2}
$N = 8192$	—	—	4.16188	5.92996×10^{-1}

Table 1: Table for run times for square matrices with varying N.

In table 1 we see run times for four different implementations of the matrix multiplication. The serial implementation, is one that you would code up with one core. The second is using OpenMP with 8 threads using an intel i7-4790K. The next two are the GPU based matrix multiplications, the two codes we built in this chapter, using an Nvidia Tesla Xp GPU. Note that the matrices are of size N^2 for this test. As the matrix sizes enlarge, we see that the shared memory based matrix multiplication is roughly ten times faster than the naive implementation. The CPU times are for our benefit, truely showing that the GPUs are the best hardware for this type of problem.

Now, for a single matrix multiplication, a ten times speed up may not seem like much. However, for applications that require thousands of matrix multiplications (think machine learning, or molecular dynamics), using shared memory could be the difference between simulations on the orders of hours vs a week.

3 Stencil

Stencil is a class of algorithms that combines the functionality of map and gather. Many applications in scientific computing as well as machine learning use this type of parallel primative. Within stencil type algorithms, each data point is "updated" using a fixed set of data points, known as a stencil. Generally, the stencil is small compared to the overarching data set, revealing concurrency. Usually, stencil type algorithms are close to embarrassingly parallel, making them ideal for offloading onto GPUs.

A couple of examples:

- In scientific computing Finite difference schemes for numerical differential equations are a prime example of stencil, as each new timestep is calculated using a set of data points from the previous timestep.
- In machine learning Convolutions in Convolutional Neural Networks involve stencil type algorithms.
- In image processing: Applying image filters, such as a "Gaussian Blur"

A nice property of stencil type algorithms is that they are easily visualizable. Take Figure 3 for example. It displays the stencil scheme for a 3D Von-Neumann type algorithm.

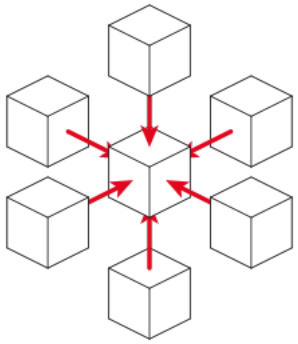

Figure 3: A 7-point Von-Neumann stencil.

3.1 Modeling Heat in a 1D Rod

To illustrate stencil, we will model heat in a 1D uniform rod of length L. Mathematically, we solve the heat-equation:

$$\frac{\partial f}{\partial t} - k\frac{\partial^2 f}{\partial x^2} = 0$$

Numerical solutions to the heat equation in the finite difference framework has been extensively studied within the last century. For a more inclusive look into numerical PDEs and ODEs, the reader is encouraged to look into AMS213b. For this example, we will use the Forward time - central space finite difference method. This scheme follows directly from using the definition of derivative:

$$\frac{\partial f}{\partial t} = \lim_{\delta t \to 0} \frac{f(t+\delta t) - f(t)}{\delta t}$$

And we decompose the second derivative as:

$$\frac{\partial^2 f}{\partial x^2} = \lim_{\delta x \to 0} \frac{f(x+\delta x) - 2f(x) + f(x-\delta x)}{\delta x^2}$$

Since computers do not understand continuous functions, we decompose the definition of derivative into an approximation:

$$\frac{\partial f}{\partial t} \approx \frac{f(t+\delta t) - f(t)}{\delta t} = \frac{f_i^{n+1} - fn_i}{\delta t}$$

and

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x+\delta x) - 2f(x) + f(x-\delta x)}{\delta x^2} = \frac{f_{i+1}^n - 2f_i^n + f_{i-1}^n}{\delta x^2}.$$

Where

$$f_i^n = f(t^n, x_i); \qquad f_i^{n+1} = f(t^n + \delta t, x_i); \qquad f_{i+1}^n = f(t^n, x_i + \delta x)$$

and $\{t^n\}_{n=0}^N$ is the set of time samples, and $\{x_i\}_{i=0}^{M-1}$ is the set of spacial locations. So using the heat equation and the finite difference approximations we arrive at the forward-time central-space (FTCS)

$$f_i^{n+1} = f_i^n + \frac{k\delta t}{\delta x^2}\left[f_{i+1}^n - 2f_i^n + f_{i-1}^n\right].$$

This form illustrates the type of stencil used, that is, we update the center data point using itself and two neighbouring data points.

When solving a partial differential equation, boundary conditions must be used. This is apparent within our algorithm. Consider the point x_0, then we cannot use the stencil provided as it access a point beyond our data set. In this case, we invoke the boundary conditions. In this case we will use what are called Neumann boundary conditions, that is

$$\left.\frac{\partial f}{\partial x}\right|_{x=-L/2,L/2} = 0.$$

Physically, it means that the rod is perfectly insolated - that is, there's no heat flux through the ends. Numerically, we apply the finite difference to get

$$f_0^n = f_1^n \qquad f_{M-1}^n = f_{M-2}^n.$$

We should note that there is a condition on the values we choose for δx and δt. This condition requires that

$$\frac{k\delta t}{\delta x^2} \leq \frac{1}{2}.$$

We have this condition to keep the solution *numerically stable*, that is if we increase beyond a half, the numerical solution will abscond to infinite values and not converge to a desirable solution. For more information on how this works, take AMS213b.

3.1.1 CUDA Kernel

We will implement the stencil within the CUDA kernel. In general it is not advisable keep every time step within the solution array. So for this example, we will use $L = 2$, $k = 1$, and $M = 128$. So using M we find that

$$\delta x = \frac{L}{M} = \frac{2}{128} = \frac{1}{64}.$$

We can use the stability condtion to find δt, that is

$$\delta t = \frac{1}{2k}\delta x^2.$$

Note that some applications may yield better results with using smaller δt.

So now that we have the prescription for each timestep we can write a kernel for our solver:

```
__global__ void ftcs(float* f, const float dx, const float k, const float dt )
{
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        if(tid> 0 && tid<M-1)
        {
                extern __shared__ float temp[];
                id = threadIdx.x;
                temp[id] = f[tid] + k*dt/(dx*dx)*(f[tid+1] - 2*f[tid] + f[tid-1]);
                __syncThreads();
                f[tid] = temp[id];
        }
}
```

Note that this kernel does not touch the boundaries. It is better to write another kernel to handle the boudary conditions than to transfer the data back to the host and have the CPU deal with them. This is even more apparent in multi-dimensional PDEs, when the boundary conditions become array operations.

```
__global__ void bc(float* f)
{
        int tid = threadIdx.x + blockIdx.x*blockDim.x;
        if(tid ==0) //use only one thread.
        {
                f[0] = f[1];
                f[M-1] = f[M-2];
        }

}
```

One reason we use two different kernels, is to insure that f[1] and f[M-2] are updated before applying boundary conditions. Since each thread is run simultaneously, there's no garuantee that this will happen before thread 0 executes.

3.1.2 Host Code

With a stencil type algorithm, the host code needs to do a couple of things: initialize data, launch kernels for each iteration, and control io-operations. In this scenario we will assume that the heat distribution in the rod follows a Gaussain profile, that is, our initial condition is

$$f(x, 0) = \exp\left[-\frac{1}{2}x^2\right]$$

where $x \in [-1, 1]$. So the host code will be:

```
int main()
{
        float k = 1.0f;
        float dx = 2.0f/float(M);
        float dt = 0.5f/(k*dx*dx);
        float x[M];
        float tmax = 2.0f;
        float t = 0.0f, tio = 0.5f;

        //Allocate Memory
        size_t sz = M*sizeof(float);
        float *f
        f = (float*)malloc(sz);
        float *d_f;
        cudaMalloc(&d_f, sz);

        //Kernel parameters
        dim3 dimBlock(16,1,1);
        dim3 dimGrid(M/dimBlock.x, 1,1);

        //Apply Initial Condition You could also create a kernel for this
        for(int i=0; i < M; i++)
        {
                x[i] = -1.0f + i*dx;
                f[i] = exp(-0.5f*pow(x[i],2));
        }

        //Transfer to device
        cudaMemcpy(d_f, f, sz, cudaMemcpyHostToDevice);

        /* IO Operations for IC */


        /*Perform Integration */

        while(t<tmax)
        {
        //Call the stencil routine
                ftcs<<<dimGrid, dimBlock. dimBlock.x*sizeof(float)>>>(d_f, dx, k, dt);
                cudaDeviceSyncronize();
        //Call BC
                bc<<dimGrid, dimBlock>>>(d_f);
                cudaDeviceSynchronize();
                if(fmod(t, tio) == 0.0f)
                {
                //IO function
                }
                t+=dt;
        }

        if(fmod(tmax,tio) != 0.0f)
                //IO Function

        //deallocate memory
        free(f);
        cudaFree(d_f);
}
```

Do not forget to use the necessary headers for the use of *exp*, the `cmath` header should suffice.

4 Exercises

4.1 Faster Matrix Multiplication

Many fast linear algebra libraries apply a transpose to one of the matrices before multiplication, that way the "fast" index is used on both matrices. For example, in C arrays are stored in row major form, so the matrix multiplication would look like this:

```
//transpose B
for(i = 0; i < n, i++)
{
        for(j = 0; j < n; j++)
        {
                for(k = 0; k < n; k++) //here B is transposed
                        C[i][j] += A[i][k]*B[j][k];
```

```
        }
}
```

Use the shared memory matrix multiplication outlined in this chapter, and the transpose you wrote for a previous homework to take advantage of fast indexing. Compare the timing between the transposed matrix multiplication and the regular as in table 1. Note: you will have to change some aspects of the shared matrix multiplication to use transposed indexing.

4.2 Gaussian Blur for Image Processing

Download the image code from the course repository. Complete the missing code segments, notably the Gaussian Kernel. In this application we will use the following stencil:

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

This operation is useful in filtering out high frequency noise from images or signals. It can also be used in machine learning via Convolutional Neural Networks.

This is a 3x3 Gaussian Blur matrix. We can also think of this as a convolutional step. In this application, the center pixel i, j is multiplied by $1/4$, as so on. So the updated pixel at i, j should be the sum of the 3x3 stencil multiplied by their respective coefficients. In your report, you should compare the image vs the blurred image.

The image that you will use is of Ana Carrasco, she made history in 2017 for being the first woman to win an individual world championship motorcycle race, at the World SuperSport 300 race in Portimao Circuit in Portugal.

Figure 4: Ana Carrasco Gabarrón, the first woman to win a World Champsionship motorcycle race

4.3 Numerical Solutions to the 2D Heat Equation

Typically in scientific applicaitons PDEs are multidimensional. In this excercise you will write a CUDA program using the 2D Forward Time Central Space finite difference algorithm to solve

$$\frac{\partial f}{\partial t} = k\left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right)$$

in a perfectly insulated square. That is, there are Neumann Boundary conditions on each edge:

$$\frac{\partial f(t,-L/2,y)}{\partial x} = \frac{\partial f(t,L/2,y)}{\partial x} = 0$$

and

$$\frac{\partial f(t,x,-H/2)}{\partial y} = \frac{\partial f(t,x,H/2)}{\partial y} = 0$$

The FTCS scheme in 2D is:

$$f_{i,j}^{n+1} = f_{i,j}^{n} + \frac{k\delta t}{\delta x^2}\left(f_{i+1,j}^{n} - 2f_{i,j}^{n} + f_{i-1,j}^{n}\right) + \frac{k\delta t}{\delta y^2}\left(f_{i,j+1}^{n} - 2f_{i,j}^{n} + f_{i,j-1}^{n}\right)$$

Assume that $\delta x = \delta y = h$ for this assignment, then $\delta t \leq \dfrac{h^2}{4k}$.

Use a Guassian Heat profile as an initial condition:

$$f(0, x, y) = \exp\left(-\frac{1}{2}||x + y||^2\right)$$

Model your 2D solver after the 1D solver presented in this lecture.

4.3.1 Solution

Let $tmax = 2.0$ and output the solution every 0.5 time units. Let In this portion let $M = 128$ so, $h = \frac{1}{128}$. So the solution will be a 128×128 array.

4.3.2 Timing

Examine how the execution time scales as M increases. For this test let

$$M = 16, 32, 64, 128, 256, 512, 1024$$

You do not need to output the solution for this test, just the execution times.