January 25, 2016

CUDA for Scientists

Parallel Programming on GPUs

Dr Sam Coveney (Scienceable Skills)
coveney.sam@gmail.com ☎ 07794 702092

Contents

1 Introduction

1.1 CUDA and GPUs

GPU stands for **G**raphics **P**rocessing **U**nit a.k.a. Graphics Card.
NVIDIA, one of the leaders in high-end graphics cards, make graphics cards suited for HPC (High Performance Computing) and for Gaming. Maybe obviously, HPC and Gaming are rather similar in many respects, which is why GPUs (originally developed to deal with the large number of floating point calculations needed for graphics rendering) are being utilised today for HPC.

- Using graphics cards for general purpose computing is often called **GPGPU: General Purpose computing on Graphics Processing Units**
- **CUDA** stands for "**C**ompute **U**nified **D**evice **A**rchitecture" (CUDA is also known as CUDA C). It is a C++ like software platform and application programming interface (API) for writing code that can be executed on CUDA-enabled GPUs.
- The CUDA architecture is how Nvidia has built its Graphics Cards such that they can do both graphics-rendering and general-purpose computing.
- **nvcc** is "**nv**idia's **c**++ **c**ompiler", which compiles CUDA into executable code. NVIDIA also provides a specialized driver to utilise their CUDA architecture. Both are provided for free (!) by NVIDIA (of course, they want you to buy their GPUs!)

1.2 (Simplified) Hardware Paradigm

For the meantime, we will need a simplified paradigm of hardware in mind (we'll get more technical later). For good reason, I will refer to the hardware as a binary: **Host** and **Device**.

- **Host** hardware - everything we'd normally think about as being a computer (and therefore not really think about)
- **Device** hardware - our GPU.

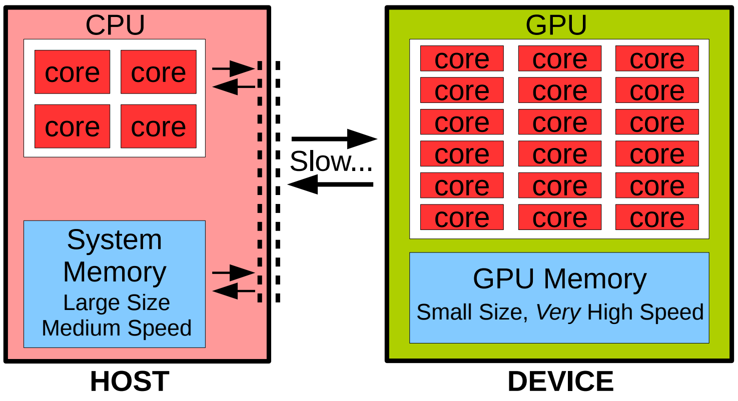


Figure 1: Simplified Hardware Paradigm of the Host and the Device.

The Host and Device are connected together via a PCI port on the motherboard.
To summarize:

- CPUs are great at processing SERIAL code: high clock speed, good at branching etc. but there is very little time dedicated to floating points calculations
- GPUs are great at processing (certain types of) parallel code: can perform the same operation on many different data elements at once, loads of cores on modern GPUs, and cores can retrieve data from the GPU memory very quickly.

- transferring memory between the host and the device is relatively slow, so we don't really want to do this very often.
- **The Idea: do Parallel stuff on Device, do Serial stuff on Host**

1.3 KERNELs

The **Kernel** is at the heart of CUDA programming. It is **a special function** written in CUDA and compiled by nvcc, which is then **executed P times on the GPU by P different CUDA THREADS**
Kernels are **launched** by the host (we say that we 'call' a function, but we ought to say that we 'launch' a kernel), and even though our CPU is telling our GPU to do something, **a launch only takes about 0.01ms, which is extremely fast!**
This means that we can call kernels many many times (which is what cards need to do to provide real-time graphics) and so **we can design kernels to do very specific work**: better to call several kernels designed to do several tasks very efficiently, rather than attempt to call one kernel that does lots of work inefficiently...

1.3.1 Threads, Blocks, and Grids

- **Thread**: an execution of a kernel, with certain **Index** information that identifies the execution from other executions. The index information can be used to get particular threads to do particular jobs (e.g. "If you have thread index 0, please fetch data element a[0], if you have thread index 1, please fetch data element a[1]", and so on).
- **Blocks**: groups of threads, which are executed on the same physical part of a GPU (a Streaming Multiprocessor) and so are able to share certain resources (such as shared memory). We can specify how many threads get grouped into a block, and how many blocks we want e.g. perhaps I want 1000 threads that are grouped into 10 blocks of 100 threads.
  The GPU will deal with all of its tasks **'blocks-at-a-time'**. Depending on the resources available to a particular GPU, it may be able to launch all blocks at once, or perhaps it will have to put them in a queue (as soon as one block is finished, another block can run, until all the work is done).
- **Grid**: a group of blocks, which can be handled by a single GPU chip. The grid is defined by the number of blocks, and the size of these blocks e.g. 10 blocks of 100 threads $\rightarrow$ there are 1000 threads in my grid. There's no particular order for the execution of blocks.

1.3.2 SIMT: Single Instruction Multiple Threads

We should write Kernel code from the perspective of a single thread, since each thread runs the same kernel but with different Index information. Using the Index information available to each thread, we can write code such that **each thread operates on different data, but in the same way**.
e.g. we need to operate on every element of an array, which is 1000 elements long, so we can use 1000 threads, each working on a single element of the array. Thread 0 performs an operation on element $a[0]$, thread 1 performs *the same* operation on element $a[1]$, and so on...

This is what GPUs are good at: Single Instruction Multiple Threads. This is quite different from writing code that runs on multiple CPUs, in which there might potentially be a lot of branching e.g. if-else statements, loops with different numbers of iterations etc. which means that different parallel executions of code might actual result in different work being done. To understand why this is the case, we need to look a bit more closely at the hardware in GPUs.

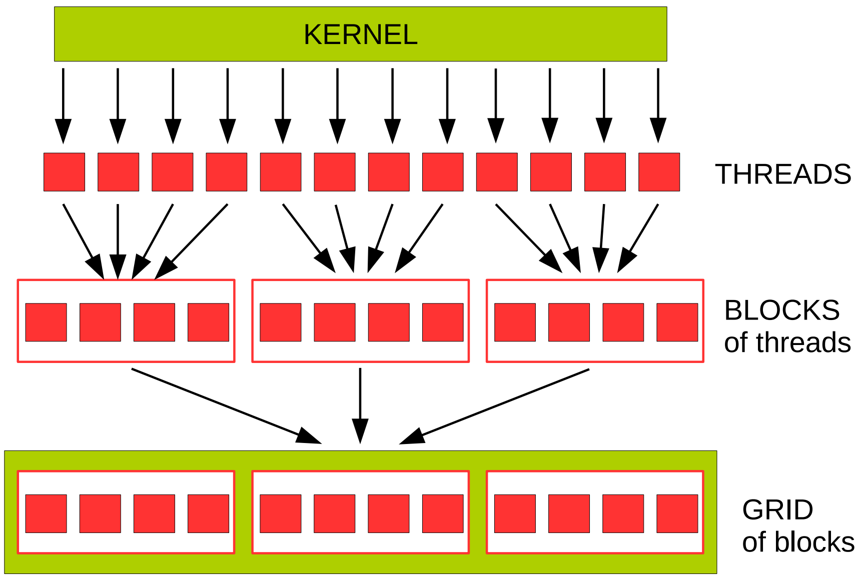


Figure 2: Representation of grouping into threads and blocks - it should be noted that we are limited to 512 threads per block, but to 65535 blocks! Technically, the scheduling of tasks happens firstly at the block level and secondly at the thread level, but as a programmer it is probably easier to imagine how our code looks like from a single thread's perspective and imagine the grouping represented here.

1.4 Vectors and Warps

1.4.1 Vector Units

When we get down to the hardware, **GPUs are Vector Processors**: they perform the same operations on many pieces of data at once (parallel), rather than processing the data pieces one at a time (serial).

One instruction results in a single basic operation on several data elements at once. More specifically, due to the hardware, **one instruction results in 32 operations at once**.

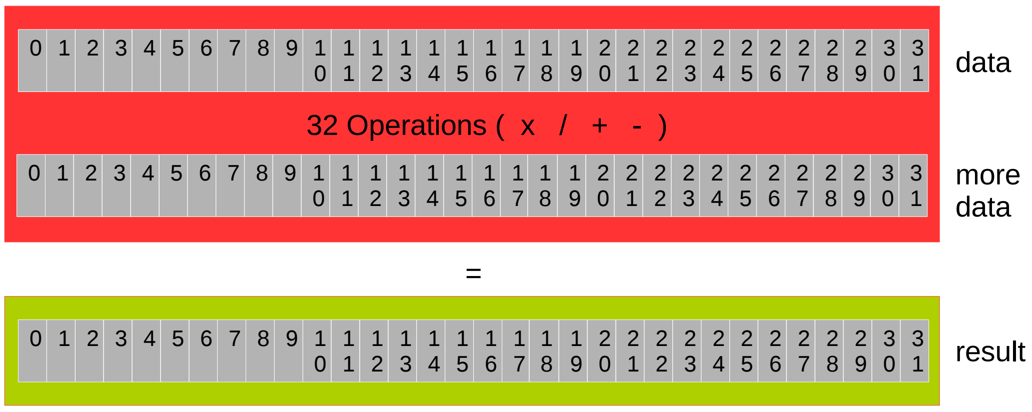


Figure 3: 32 basic operations being applied at the same time.

1.4.2 Warps

A group of 32 consecutive threads is called a Warp (the name is taken from weaving).

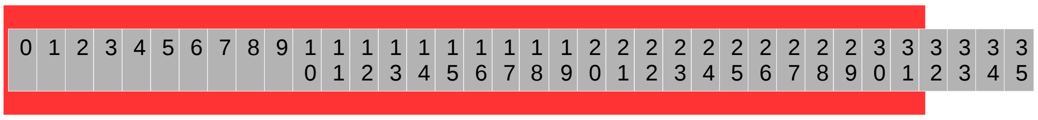


Figure 4: 32 threads constitute a Warp

Therefore, **one warp represents a full hardware vector unit** i.e. an entire warp is processed at once by the hardware.
We won't be too technical now, but remember blocks of threads? Well, each block is scheduled to run on a particular area of the GPU called the 'Streaming Multiprocessor' or 'SM'. On the SM, **Blocks are sliced up into Warps**, since execution of instructions requires execution of a warp e.g. a Block of $129 = 128 + 1 = 32 * 4 + 1$ threads requires 5 warps (4 lots of 32, and an entire warp just for the 129th thread).

Think about the following Block of 22 threads:

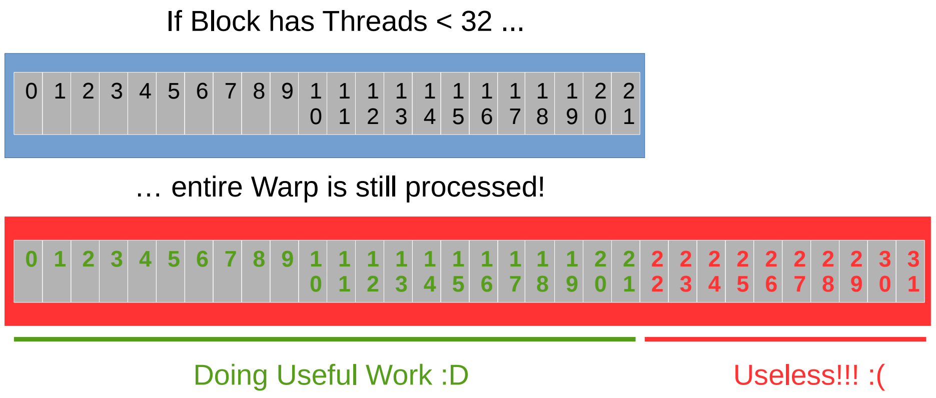


Figure 5: We can only perform operations in Warps - don't waste resources!

We've asked the GPU to do work for 22 threads, but it can only perform tasks in warps. We could have had a block of 32 threads, and it would've taken the same amount of time! (Providing there was no branching - we'll get onto that in a moment...)
What a total waste of resources!
The same argument applies to a block of 50 threads: while we seem to ask for 50 instruction to run, really 2 warps worth of instructions are run, so it wouldn't have taken any more time for 64 threads (2*32) than for 50 threads.
Since we only asked for 50 threads, only the results of the 50 threads that we specified for the block will be used (the other 64-50=14 results? Gone to the void...)

Occupancy: Active Threads / Maximum Active Threads
We want to maximize occupancy, such that we are using as many threads as possible.

A small technicality...

Due to the number of Scalar Processor **Cores** on each Streaming Multiprocessor, all threads in a warp may not *technically* run concurrently... but it *honestly* suffices to assume that they do! As far as we're concerned as a programmer, the execution of threads within a warp happens concurrently. I almost left this out, because I don't want you to think about it.

1.4.3 GPUs and BRANCHING

When we think about parallel processing, we tend to think about many separate programs running at the same time. This is a good paradigm in some respects, but we have to think a little harder if we want to write good GPU code.

e.g. we have 32 threads running (one warp) and they approach an **if-else** statement. That's fine, right? Some threads will do If and some threads will do Else? WRONG! (well, mostly...)

ALL threads in the warp will follow If, and then ALL threads in the warp will follow Else. The execution of code in the branches is serialised.

The threads follow **both** branches and the correct results (based on the conditional statements of the If-Else) will be kept.
Why? Remember the Vector Units in the hardware? An instruction causes 32 operations. We can't simply take some threads in a warp and have them do one thing, and have the other threads in a warp do another.

If all the threads in a warp satisfy IF (or ELSE) then they all follow only IF (or ELSE), and then we don't have to follow both branches, since the threads in a warp wouldn't have to follow both branches (single instruction, only need to do IF provided they all satisfy the conditions for IF), but otherwise we might be doing a lot more work than we might first think...

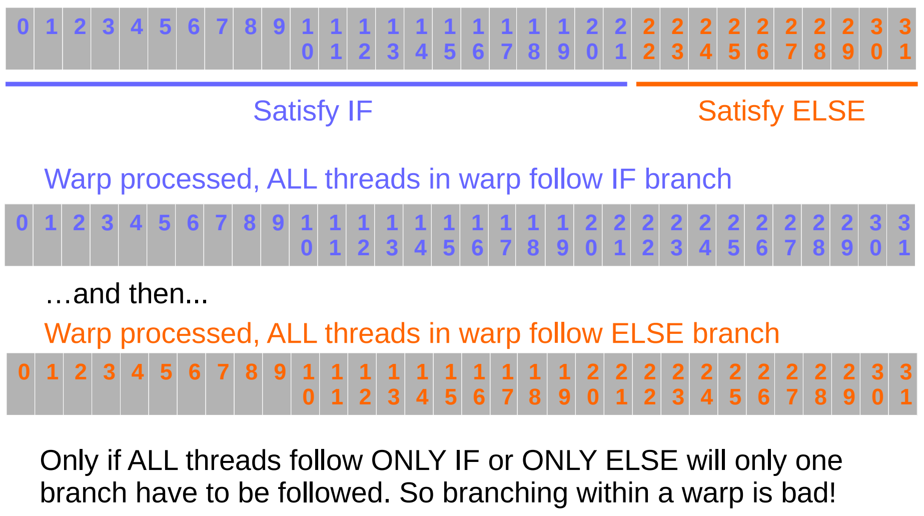

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Satisfy IF
Satisfy ELSE
Warp processed, ALL threads in warp follow IF branch
…and then...
Warp processed, ALL threads in warp follow ELSE branch
Only if ALL threads follow ONLY IF or ONLY ELSE will only one branch have to be followed. So branching within a warp is bad!

Figure 6: The dangers of divergence and branching *within* a Warp. However, when one entire Warp follows if, and another entire Warp follows else, then there's no problem!

Each thread *could* execute a totally different program, with no relation to the program that other threads are running e.g. "If you are thread 0 then run this bit of code, If you are thread 1 then run this totally different bit of code etc." but we see now why we *don't* want to do this!

Avoid branching, but it's not the end of the world - sometimes it's necessary, but if every thread is following a completely different set of instructions, your code will perform much much slower than you might expect...
Hence the principle: **Single Instruction Multiple Threads**.

1.4.4 A word on Double Precision...

The CUDA core count represents the total number of single precision floating point or integer thread instructions that can be executed per cycle. At most, 4 of the 12 groups of cores in a multiprocessor can be executing double-precision instructions concurrently.

This means the double-precision peak speed is one-third of the single-precision peak. Use floats! They perform much better than doubles!

Need doubles? Perhaps use **Mixed Precision Code: calculations done with floats, data stored as doubles**. (By some sorcery, this can provide the required accuracy, without resorting to calculating with doubles... This has lead me to suspect that GPUs are possessed...)
Also, bear in mind that functions like exp() are also much faster with floats than with doubles, just like on CPUs. Really (for real really) need doubles? Okay, use doubles. But are you sure you couldn't find another way to improve accuracy? (e.g. use higher-order accuracy discrete differencing)

2 The Basics

2.1 Navigating the Linux Server

If you are making use of a server for this course, here are some useful things to know.

2.1.1 Log In to Server

Connect to my wireless (I'll tell you how) and Log In to the server:
ssh -X username@serveraddress
(the option -X allows graphics to be sent to your screen from the server; useful). I will give you this server address, and tell you what your user name and password is.
To log out, type **logout** or **ctrl-d**

2.1.2 Useful Linux Commands

To navigate around the server, we need several commands:

ls : lists the folders/directories within the current folder/directory

cd : change folder/directory e.g.

- **cd helloworld** to 'go into' the helloworld dir (within current dir)
- **cd ./helloworld** does the same as above (**.** indicates the current dir)
- **cd ..** to go back one directory from the current dir
- **cd /home/username/anotherfolder** is changing dir using the *absolute* path (begins with / Also note that ˜ (tilde) means the same as /home/username)

mkdir : creates a directory e.g.

- **mkdir folder1**
- **mkdir folder1 folder2 folder_3**
- (for programming, don't use spaces in folder names, use underscores if need be)

cp : copy a file e.g. **cp file destination** to copy 'file' into folder 'destination'

mv : move a file, and perhaps also rename it e.g.

- **mv file destination**
- **mv file destination/newname** (unless newname is a folder in destination) ;
- **mv file newname**

rm : delete a file e.g. **rm file** - be very careful with this!

***** : a *wildcard*, for matching patterns e.g.

- **ls *.txt** will list any files ending in .txt
- **rm ./*** deletes everything in current directory!

./bin/executablename : will run the script or program 'executablename' in the folder 'bin'

2.1.3 Uploading/Downloading files

The easiest way to do this is with **sftp: secure file transfer protocol**.
From a terminal on our local machine:
sftp username@serveraddress

and we can then **put** files on our local machine onto the server
put path/to/file/on/local/machine

or **get** files from the server to our local machine:
get path/to/file/on/server/machine.

To change directory etc. on the server, use the standard linux commands (limited in sftp), and to change directory etc. on the local machine begin the commands with **l** e.g. **lls, lcd**

For up/downloading entire directories, it is best to **zip** the folders, from **ssh**, with:
zip -r zipname folder/
and then use sftp with for zipname.zip e.g. textbfput zipname.zip

If you are on windows, you'll want to be using **putty/WinSCP** to do ssh and sftp.

2.1.4 Text Editors

We'll need to make use of text editors to write our code. Personally, I think **nano** is simplest, though I prefer vim.

- To open nano, type **nano**, or to open the 'file.txt' with nano use **nano file.txt** (this will either open file.txt if it exists, or open a new file called file.txt).
- To open nano with nice colours, do **nano –syntax=c /path/to/file** (c syntax colours CUDA pretty well - there will be a CUDA color syntax file out there somewhere...)
- To save output in nano, hit **ctrl-o** and you can either overwrite the original file content or choose a new filename.
- To exit nano, hit **ctrl-x**.

2.1.5 Screen

Since we'll be editting and compiling code, it's good idea to open several ssh sessions in different terminal windows, so that you can edit in one window and compile in another. However, if we use **screen** then we don't need to log in with ssh multiple times, we can open multiple shell windows from one ssh session.
To use screen, type **screen**. We are now in the screen environment, which looks exactly the same. The special key combination for screen commands (as opposed to commands to the shell) is **ctrl-a** followed by something else:

- **c** - this opens a new shell session
- **k** - this kills a screen session (it will ask for confirmation)
- **n** - this flips to the next screen session
- **p** - this flips to the previous screen session
- **d** - this detaches the screen session - this is powerful! We can set something running, detach the entire screen session (all shell windows included) and log out! Then we can come back later and reattach! To reattach, use **screen -r**

2.2 Practical Work

2.2.1 Project organisation

All of the example code is organised into project folders called 'ProjectName' (e.g. "helloworld") filled with these folders:

- bin (for executables and run scripts)
- source (for source code)
- utils (for utility scripts e.g. compile, plotting)
- data (for data)

We can then work from the directory 'ProjectName', using scripts in the subdirectories like this:

- compile code: **./utils/compile**
- submit program to run: **./utils/submit** (calls script ./bin/run which calls the program)
- plot data: **./utils/plot $time** (plots ./data/$time.txt)

Remember, don't switch to the subdirectories, or these scripts won't work!
I (should!) have provided incomplete code for you to build on throughout the course, which will hopefully allow you to focus on CUDA and CUDA paradigms, rather than having to worry about anything else.
Complete code will be provided afterwards, so its not hugely important that you finish all of the practical work. In order to assist you further, I have provided scripts to do all of the other work (compile, plot etc.) for you.

2.2.2 Compiling

We can compile with **nvcc main.cu -o bin/executablename**
(if main file is 'main.cu' and you want the binary placed in directory 'bin' in current dir).
We can then run our code **./bin/executablename**

However, we can also use the script **./utils/compile**:

```
#!/bin/bash

src="./source"
main="main.cu"
ex_name="${PWD##*/}"
bin="./bin"

# invoke nvcc compiler
# -w means warnings will NOT show
nvcc -arch=sm_20 -w $src/$main -o $bin/$ex_name
```

The ${PWD##*/} just makes sure that the executable has the name of the project directory.

2.2.3 Running

We can also use the script **./bin/run** to run our code:

```
#!/bin/bash

# Error checking
EXIT_STATUS=2
# write program to Return:
# 0 when no-errors , 1 when commandline interrupt...

# run program from project directory
./bin/${PWD##*/}

# get the exit status of the CUDA program (last executed program)
EXIT_STATUS=$?

# print various things bases on exit status
if [ $EXIT_STATUS == "2" ]
then
        printf "\n>>> EXIT STATUS $EXIT_STATUS, did program run?"
fi
```

This script just collects the return status of our program, which is handy.

2.2.4 Submit job to a job engine

However, its better, perhaps, to run the job with **./utils/submit**, since this script can submit the run-script to a job-queue (such as the sun-grid engine on the White Rose Iceberg cluster). This script must contain the appropriate commands for the particular system you are submitting your job to. For iceberg, the script would look like this:

```
#!/bin/bash

qsub -l gpu=1 -l arch=intel* -P gpu ./bin/run
```

so we can simply type **./utils/submit** and everything happens automatically.
(Bear in mind that the run script that is submitted will usually need some special comments after the #!/bin/bash for the job engine to know certain things e.g. how many cores? etc.)

For us, this script simply calls **./bin/run**, which makes it a little superfluous, but this way of organising projects means that you can almost seamlessly move your project from a local computer to a cluster and only have to comment/uncomment 1 or 2 lines in your submit/run scripts in order for everything to work fine. Great!

2.3 Hello, World! (with some GPU stuff!)

While there is incomplete code for this section that you can build upon, this section is written so that you could write all of the source code yourself. However! I would only write the **main.cu** and **myprint.h** if I were you, and use the provided header files (explained below) rather than write them yourself.

Create a file called **main.cu**. The main program looks like a C++ program:

```
int main(void)
{
// this is a comment
/* this is a
multi-line comment */
}
```

To save us messing around later, please write all of:

```
using namespace std;
#include <iostream>
#include <stdio.h>
#include <cuda.h>
#include <assert.h>
#include <cmath>
#include <sstream>
#include <signal.h>
```

at the top of every main.cu file for this course.
We can then write our first program:

```
printf("Hello, World!\n");
```

We can compile it from the directory 'helloworld' with:
nvcc source/main.cu -o bin/helloworld or **./utils/compile**
and run it with:
./bin/helloworld or **./utils/submit**
Create the header file **myprint.h** or something similar, and fill it with:

```
#ifndef MYPRINT_H
#define MYPRINT_H

void myprintfunc()
{
   printf("Hello, World!\n");
   return;
}

#endif
```

Then in **main.cu** include the header file with:

```
   #include "myprint.h"
```

Our function myprintfunc() can then be called from main.cu

2.4 Device Setup (and cudaFunctions())

By **Device** I mean the **GPU**.
It is a good idea to 'set-up' the device properly by *obtaining the device number*, *setting the device*, and *resetting the device* prior to use.
The reasons are many, the main one being that GPUs are haunted, another being that bugs in your code (usually writing/reading from memory you're not meaning to) may be easier to identify after a device reset, and so on...

I've written the following **CUDA header file** "device_setup.cuh" for resetting the device:

```
#ifndef DEVICE_SETUP_CUH
#define DEVICE_SETUP_CUH

void device_setup()
{
    // this lets us set up the device - good practice
    cout << "Setting up GPU device..." << endl;
    int cuda_device;
    assert( cudaGetDevice(&cuda_device) == cudaSuccess);
    assert( cudaSetDevice(cuda_device) == cudaSuccess);
    printf("    Set Device %d...", cuda_device );
    printf(" Resetting Card...");
    assert( cudaDeviceReset() == cudaSuccess);
    printf("successful!\n");
}

#endif
```

Usually, CUDA chooses the best GPU for the job (best not to have that GPU plugged into a monitor!) For this course, each user has been given a different GPU number on my server, and the code above is slightly modified to look for the environment variable giving the number of this GPU, so that the selected device is automatically the correct one. We learn from this example:

- CUDA header files are like C++ header files, but have extension '.cuh'
- CUDA has built in functions, such as **cudaGetDevice()**
- **use assert (found in assert.h) to check if CUDA functions were successful**

This (user defined) header file can be included with

```
#include "device_setup.cuh"
```

under the header files I asked you to include.

2.5 Error Checking

2.5.1 Assert

Above, we saw that we can use

```
assert( cudaSomeFunction() == cudaSuccess );
```

to check if a CUDA function returned successfully. This works because a CUDA function will return cudaSuccess if successful, but will return another error message otherwise. If the conditional is not met, assert will fail and the program will be stopped.

2.5.2 Other useful Error-y stuff

There are other things that we can do that are pretty useful (**feel free to skip this section, it's not strictly necessary**).
Here is my own header file "signal_handler.cuh" for catching ctrl-c interupt signals.

```
#ifndef SIGNAL_HANDLER_CUH
#define SIGNAL_HANDLER_CUH

extern int no_sigint;
int no_sigint=1;

void sigint_handler (int sig)
{
          if(sig == SIGINT)
          {
             std::cerr << std::endl
                << "!!! Forcing exit from main loop !!!" << std::endl;
             no_sigint=0;
          }
          return;
}

int return_exit_status(bool no_problem)
{
    if (no_sigint == 1 && no_problem)
    {
            cout << "Return 0 for completion without errors" << endl;
        return 0;
    }
    if (no_sigint == 0 && no_problem)
    {
            cout << "Return 1 for user ctrl-c signal interupt" << endl;
            return 1;
    }
    if (!no_problem)
    {
        cout << "Found a bad number, return 3!" << endl;
        return 3;
    }
    return 99;  // we'll return 99 if nothing else
}
#endif
```

If we include “signal_handler.cuh” in our main program, we can then call (as early as possible):

```
    signal(SIGINT, sigint_handler);
    bool no_problem = true;
```

and

```
    return return_exit_status(no_problem);
```

at the end of our program (instead of writing return 0;)

Why? The main work of our programs will probably be done in a loop, so:

```
 while(no_sigint && no_problem)
   {
    // main program, probably looping again and again...
   }
```

This means that we’ll keep looping until we get a signal interrupt from the keyboard (can’t do this if we’ve submitted a job to a job engine, of course...) or we find ‘a problem’.

For me, when I print data, or take an average, I check to see if any of the data is bad:

```
if (isnan(av_data) || isinf(av_data) )    no_problem = false;
```

and then the main program loop will exit, and I’ll know why the program exited. Useful!
This also allows us to perform other tasks (e.g. free memory, print data) after our interrupt or ‘crash’. Also useful!

2.6 Main CUDA template for this course

Putting everything above together, we have

```
using namespace std;
#include <iostream>
#include <stdio.h>
#include <cuda.h>
#include <assert.h>
#include <signal.h>
#include <cmath>
#include <sstream>

#include "device_setup.cuh"
#include "signal_handler.cuh"

int main(int argc, char **argv)  // or just int main(void)
{
   device_setup();
   signal(SIGINT, sigint_handler);
   bool no_problem = true;

   // stuff like declaring memory and parameters goes here

   while(no_sigint && no_problem)  // infinite loop!
   {
      // main body of program (can use IF instead of WHILE...)
   }

   // stuff like freeing memory goes here

   return return_exit_status(no_problem);
}
```

3 CUDA 101

3.1 Memory

Remember the (over-simplified) Hardware Paradigm in the first section? It shows that there is ‘normal’ memory (the memory you would use in ordinary C++ program) and there is GPU memory (physically on the Graphics Card).
The ‘normal’ memory is **Host** memory, and the GPU memory is **Device** memory.

3.1.1 Arrays and Pointers

In C++, when we allocate memory to a pointer, that memory is (almost always) **contiguous**, meaning that the memory is allocated as a single chunk.
Say I create an array of three elements in C++ as follows:

```
int *vec;
vec = new int[3];
vec[0] = 1; vec[1] = 2; vec[2] = 3;
```

then I know that *offsetting* and *dereferencing* like this

```
*(vec);      // equiv to vec[0]
*(vec + 1);  // equiv to vec[1]
*(vec + 2);  // equiv to vec[2]
```

will give the values 1, 2, and 3 respectively, since the values lie next to each other in memory.
The following declaration would *not* give data stored contiguously in memory

```
int a, b, c;
```

For our purposes, we will probably use the terms ‘arrays’ and ‘pointers’, by which we mean ‘pointers-that-point-to-the-beginning-of-a-block-of-allocated-memory’, fairly interchangeably (you can’t change the memory that the array points to, but you can change the address that pointers point to - when you’re just storing data, this distinction doesn’t really matter).

3.1.2 Host Memory

For the Host to work on data, that data must be in Host Memory.
(Newer cards allow memory accessible by both host and device using Unified Addressing, but this is probably slow and won’t teach you how to be a good GPU programmer).

To declare and free Host memory, we will use the C++ commands **new** and **delete**:

```
float *h_array;  // pointer to float

h_array = new float[cols]; // allocates 'cols' floats to h_array

delete[] h_array; // deletes the array
```

Notice that I’ve used **h_** to indicate that this is a Host array.
This isn’t necessary, but it’s recommended practice to name all Host variables in this way.
We could have also used the CUDA function **cudaMallocHost()**

```
float *h_array_pagelocked;

size_t width = sizeof(float)*cols;
cudaMallocHost(&h_array_pagelocked, width);
```

```
  cudaFreeHost(h_array_pagelocked)
```

which would have given us 'page-locked' host memory, but I would rather we practice using the standard C++ method to help us **conceptually separate the CUDA-bit from the C++-bit**.

3.1.3 Device Memory

For the Device to work on data, that data must be in Device Memory. Here is how to declare '1D' device memory (we'll look at '2D' device memory later):

```
  float *d_array; // point to float

  size_t width = sizeof(float)*cols;
  cudaMalloc( (void**)&d_array, width); // 'width' bytes alloc on GPU

  cudaFree(d_array);  // frees the memory on the device
```

The (void**) part has to do with passing around the return status of cudaMalloc, not to do with d_array (the first argument is a pointer to the pointer that you want to hold the address of the newly allocated parameter - don't think about it too much!).

If we are writing device code (that is, code that executes on the device) then we can access elements of this array as with would in C++

```
   d_array[2];
   // or, equivalently
   *(d_array + 2);
```

3.1.4 CUDA memory copies

In order to copy memory between host and device arrays, we need to use CUDA functions.
For the memory above, we can copy the data between the host and device arrays like this:

```
   // copy from host to device
   cudaMemcpy(d_array, h_array, width, cudaMemcpyHostToDevice) ;
   // copy from device to host
   cudaMemcpy(h_array, d_array, width, cudaMemcpyDeviceToHost) ;
   // we can also copy device-to-device and host-to-host
```

The function takes the following arguments

```
   cudaMemcpy(*destination, *source, width_of_transfer, direction);
```

It is much safer to use assert when using memory copies

```
   assert(
      cudaMemcpy(d_array, h_array, width, cudaMemcpyHostToDevice)
      == cudaSuccess
   );
```

For this course, you should always use assert with memory copies.

3.2 Functions

When compiling code, the compiler needs to know what it is compiling code for.
Is this code to be executed on the device/GPU? Or on the host machine? Or, is the code a Kernel (the special function that gets executed multiple times in parallel, according to our configuration for the Grid?)
We can use special qualifiers to tell the compiler how to compile our functions.

3.2.1 Host Functions

Host functions, which execute on the host machine, can be specified with:

```
__host__ type( arguments )
```

or not specified i.e.

```
type( arguments )
```

in which case the compiler will assume a host function.
Host functions can only be called from Host Code!

3.2.2 Device Functions

Device functions must always be explicitly specified as device functions

```
__device__ type( arguments )
```

Device functions can only be called from Device Code or Global Code i.e. they can only be called from other code that runs on the device, and so cannot be called from host code (it is possible to give both device and host qualifiers to a function, but we won't be doing this.)

3.3 Kernels : Global Functions

To repeat what was said earlier:
The **Kernel** is at the heart of CUDA.
It is **a function** written in CUDA and compiled by nvcc, **which is then executed P times on the GPU by P different CUDA THREADS**.

Kernels must be qualified as 'global' and must be of type void

```
__global__ void example( arguments )
{
    // some code
    return;
}
```

Kernels can only be called from Host code, but the code executes on the GPU.
They must be called, or **launched**, like this:

```
example<<< grid_spec,block_spec >>>( arguments );
```

Notice the **special syntax <<< >>> within which we must specify the dimensions of the grid**. This is what we'll look at next.

3.3.1 Specifying the Grid

We must provide Grid specifications for our kernel, which determines how the kernel is executed. **How many threads do we want, and how do we want them configured?** In 1 batch of 100, or in 10 batches of 10? Or maybe in 2X5 batches of 5X2?

The Grid is specified as follows:

- Threads are grouped into **Blocks**, and the threads within a block execute concurrently on a GPU. The order that threads run on the GPU is not determined, and not all the threads will necessarily execute at the same time e.g. 1000 threads, so maybe 10 blocks of 10 threads?
- The threads within a block all run on the same streaming multiprocessor (part of GPU). If there are enough resources, the blocks all run at the same time. Otherwise, blocks are queued while other blocks run, and as soon as a block finishes another block can be run;
- All blocks must be the same size for a specific grid;
- We must specify the dimensions of blocks (how many threads in each block?) and the grid dimension *in terms of blocks* (how many blocks in the grid?). **The Grid is the result of our specification of blocks** e.g. each block has 32 threads, 10 blocks, and the result of that is a grid of 320 threads.

We must specify the Grid using a special CUDA variable called **dim3**

```
// tpb : threads per block
// 2D block: tpb_x X tpb_y
dim3 block_spec = dim3( tpb_x , tpb_y );

// 2D grid: numblocks_x X numblocks_y
dim3 grid_spec  = dim3( numblocks_x , numblocks_y );

// kernel
example<<< grid_spec , block_spec >>>( arguments );
```

A dim3 variable can be 1D, 2D, or 3D:

```
dim3(32);      //valid
dim3(8,4);     //valid
dim3(4,4,2);   //valid
```

Remember, **blocks should be multiples of warps** i.e. multiples of 32, because any less means that we're wasting resources.

3.3.2 Unique Grid ID

Remember, a Kernel is executed P times on the GPU by P different CUDA THREADS.
So if there are 10 blocks of 32 threads, then the kernel is executed 320 times in parallel (although, due to resources, not all of these execution can necessarily run at once).

We can think of the execution of the kernel from each thread's point of view, as if each thread were encountering the same code.
So how do we get different threads to do different things? e.g. act on different piece of data?
We can use built-in **Index Variables** in our Kernel code:

```
blockIdx.x;    // the x-index of a block within the grid
blockDim.y;    // the y-size of the block dimension
threadIdx.z;   // the z-index if a thread within a block
```

From these variables, we can calculate unique indexes for each thread in the grid:

```
int x = blockIdx.x*blockDim.x + threadIdx.x;
int y = blockIdx.y*blockDim.y + threadIdx.y;
```

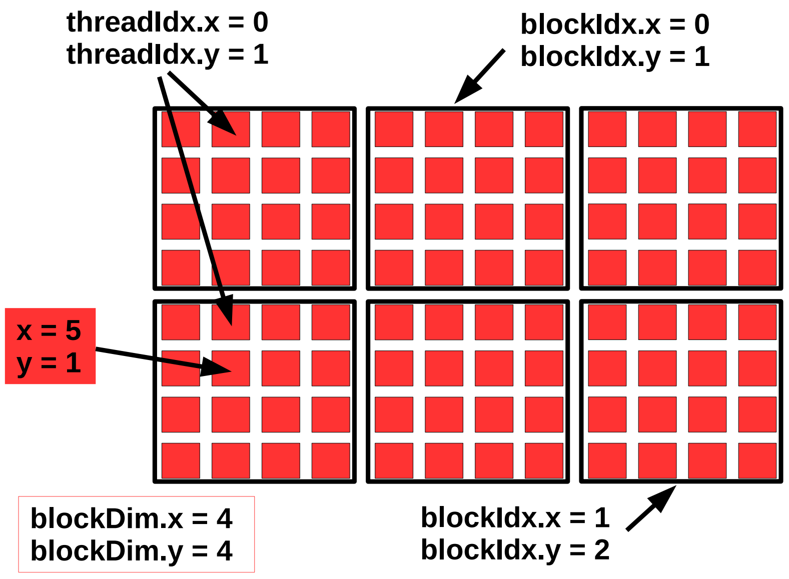


Figure 7: We can calculate a unique global index (x, y) for every single thread in our grid.

3.3.3 Mapping the (Thread) Grid to the (Data) Array

So Kernels have access to several built in variables that allow us to specify a unique ID for each kernel. So **if we calculate a unique ID for each thread, then we can 'map' the grid of threads to our data** (e.g. 32 threads operating on 32 pieces of data i.e. 1 datum per thread). We can imagine a Grid overlaid on top of an Array:

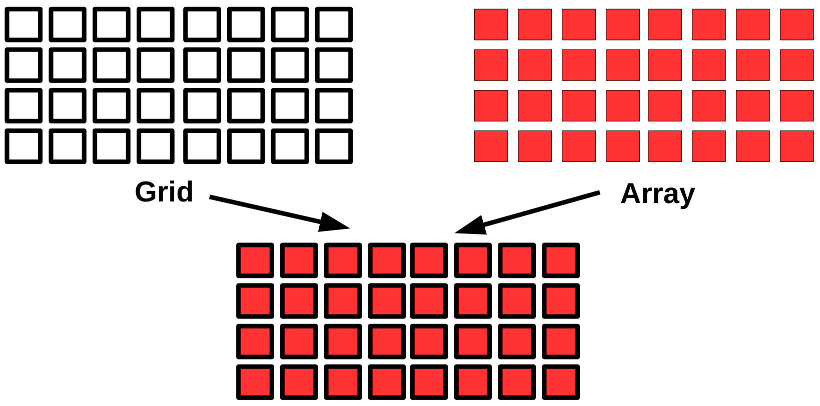


Figure 8: Idea of 'Mapping' a grid to a set of data, such that we process all the data.

As an example, let's say we are squaring an array with 'rows' elements.
With a host function, our code would probably look like this:

```
    for (int i = 0; i < rows; i++)
    {
        a[i] = a[i] * a[i] ;
    }
```

With a Kernel, our code could look like this:

```
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    a[x] = a[x] * a[x] ;
```

3.3.4 Specifying a big enough grid

If we declare, say, a '3D' array that is rows long, cols wide, and depth deep, what size should we make our 3D blocks? Well, we need enough threads in each direction to map to each element... I would recommend the following procedure:

```
int rows, cols, depth;

int bx, by, bz; // you'll have to decide these...

dim3 tpb(tpb_x, tpb_y, tpb_z);
dim3 grid(rows/tpb_x, cols/tpb_y, depth/tpb_z);
// rows must be multiple of tpb_x to cover al rows etc.
```

This code ensures that we have one thread per element of the array (assuming this is what you need, which it probably will be).

3.4 TASK: Write GPU code to square an array

1. Write a normal C++ program (compiled with nvcc - see the template code for this course) in which an array of 'row' elements is defined, sent to a function, squared, and the results printed out *from the main code (not from the function)*;

2. In the same code, allocate an array on the device for 'row' elements.

3. Using a cudaMemcpy, copy the data in the host array onto the device array.

4. Write a kernel (a function qualified with __global__) which defines a unique grid ID, and uses that ID to square an element of an array.

5. In the main host code, specify the Grid using dim3 variables;

6. Call the kernel from the main host code, with the grid specification you just made;

7. Copy the data from the device array back to the host array, and print the results in the host array. Did it work?

I will provide a complete working example that you can use to guide you, but it is worth trying to follow the steps above. Below, I've provided bits of code that, combined with the course template code, should guide you through most of what to do.

3.4.1 Example Code for main.cu

In this example code, there are functions ("populate_host_array" and "print_host_array") which you will either have to write, or take from the appropriate header file I have provided in the example code (or, just #include this header file).

```
int n = 3; // power to multiply array by

// host array stuff
float *h_array;
int cols = 100;
h_array = new float[cols];

// put numbers in the host array
populate_host_array(h_array, cols);
print_host_array(h_array,cols);

// device array stuff
float *d_array;
cudaMalloc(&d_array, cols*sizeof(float));

// copy data from host to device
assert(
    cudaMemcpy(d_array, h_array, (size_t)(cols*sizeof(float)),
                        cudaMemcpyHostToDevice)
== cudaSuccess );

// grid specifications
// we'll keep it simple here: 1 block with enough threads
dim3 grid_specs(1);
dim3 block_specs(cols);
```

```
    // ******** body of main program *********
    if(no_sigint)
    {
       printf("****** Array Multiplying KERNEL *******\n");

       arraymultiply<<<grid_specs , block_specs>>>(d_array, cols, n);
    }

    // copy data from device to host
    assert(
       cudaMemcpy(h_array, d_array, (size_t)(cols*sizeof(float)),
                  cudaMemcpyDeviceToHost)
    == cudaSuccess );

    print_host_array(h_array,cols);

    // free all memory
    delete[] h_array;
    assert( cudaFree(d_array) == cudaSuccess);
```

3.4.2 Example Code for KERNEL

This kernel makes a call to a device function "to_the_power", which multiplies an array by itself n times. You'll either have to write such a device function, copy it from the example source code, or include the appropriate header file in the example code.

```
__global__ void arraymultiply(float* device_array,
                              int columns, int power)
{
        int j = threadIdx.x;

        float array_value = device_array[j];

        // here we call our own device function
        device_array[j] = to_the_power(array_value,power);

        __syncthreads();  // syncs our threads - explained later
        return;
}
```

4 Project: Diffusion Simulation

4.1 The Physics of Simple Diffusion

Here is a polymer blend made of two components Polymer A and Polymer B. It can be represented on a 2D grid by the value of one of the components, say $\phi \equiv \phi_A$, at each point on the grid.

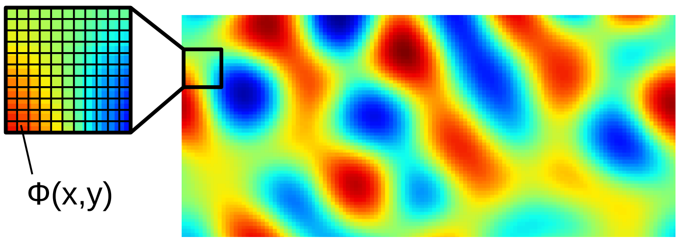


Figure 9: A 2D grid containing values of ϕ: the volume fraction of a polymer fluid. If we have a mixture of different polymers A and B, then $\phi = 1$ means all A, $\phi = 0$ means all B, and $\phi = 0.5$ means 50:50 A and B.

Without too much detail, I can tell you that under some conditions the blend remains mixed, and under other conditions it unmixes, or 'phase separates'.
The composition of the blend ϕ at each point in space (x, y) changes with time, and we would like to know how $\phi(x, y)$ changes with time. This must be done on a finite difference numerical grid, with grid spacing Δx and Δy, such that each cell represents the value of ϕ at the point (x, y), as shown in the diagram.
If we use the indices i and j for x and y, then we can write the following equations for the rate of change of fluid:

$$\phi_{ij}(t + \Delta t) = \phi_{ij}(t) + \Delta t \frac{\partial \phi}{\partial t}|_{ij} \tag{1}$$

$$\frac{\partial \phi}{\partial t}|_{ij} = \nabla^2 \mu|_{ij} \tag{2}$$

$$\approx \frac{\mu_{i+1,j} + \mu_{i-1,j} - 2\mu_{ij}}{\Delta x^2} + \frac{\mu_{i,j+1} + \mu_{i,j-1} - 2\mu_{ij}}{\Delta y^2} \tag{3}$$

The best way to approach this problem is to first calculate the chemical potential μ at every point, and then update the values of the composition ϕ at every point. Do not fear! I will provide the functions that return the value of the chemical potential!

4.1.1 Project Tasks

The rest of this section contains the information we will need to build a 2D diffusion simulation, such as how to use 2D memory on GPUs. We'll be going through all of the theory before you start, but take a look at what we'll be doing later:

- declare a host array for storing values of ϕ, and initialise it. I've provided the function for initialising the ϕ array with a value $\phi = \phi_{av} + \delta\phi$ where $\delta\phi$ is a random number; I'd recommend a 32*32 array to begin with.
- declare two device arrays, one for storing values of ϕ and one for storing values of μ, and copy the data in the host ϕ array into the device ϕ array (we'll look at 2D memory in a moment);
- take a look at the 'neighbours' device function I have written: it takes value of x and y (corresponding to an element in a rows$\times$columns array with **periodic boundary conditions** - 2D PBC means the grid is wrapped into a torus) and returns the next nearest

neighbours. Since a thread working on position x, y will need to retrieve not only ϕ at (x, y), but $\phi(x+1, y)$, $\phi(x-1, y)$, $\phi(x, y+1)$, $\phi(x, y-1)$, this will be useful

- Write a kernel to calculate the chemical potential. Your kernel will need to read in values from the (device) ϕ array, call the function to return the chemical potential, and store the results in the (device) μ array;
- Write a kernel to update values of ϕ. Your kernel will need to read in values from the μ array, calculate the value $\nabla^2\mu$, and use this to update the values in the ϕ array; How many blocks do you need, and of what size?
- every so often, you will need to copy data back from the device ϕ array into the host ϕ array, and print out the host ϕ array - I've provide a function for printing the ϕ array to a file. You can plot it with gnuplot, for example (see plotting script in ./utils).

We'll be using this code tomorrow when we learn how to optimise GPU code, but if you don't manage to finish it then I can give you some completed code that you can use.

4.1.2 1D Dewetting Simulation EXAMPLE CODE

Rather than start you off too blind, I've written a Dewetting Simulation that you can take a look at for guidance. It is diffusion-based, and so the code is actually very similar to the code you will need for the Diffusion Simulation we're building. However, this dewetting simulation is only 1D!
I don't want you copying bits of code over without understanding what we're doing, but hopefully this code might serve as a guide. Feel free to spend lots of time getting your head around this 1D dewetting simulation before moving onto the tasks outlined above.

4.2 Using '2D' memory

With C++, you have probably been shown to create a 2D array (there are differences between arrays and pointers, but it doesn't matter to us here) using a declaration like this:

```
        type **array2D_pointer; // pointer method
        array2D_pointer[1][2] = value;

        type array2D_array[size_x][size_y]; // array method
        array2D_array[1][2] = value;
```

We are not going to do anything like this.

4.2.1 Flattened 2D array

We are going to use a **flattened 2D array**. This means that we use a 1D array, and make sure we use an **access pattern** so that we can use it to store 2D (or 3D, 4D, 5D etc.) data.
Why? For some types of CUDA memory, we simply must access the data this way, and I want us to get used to it. The other reason is that the memory is contiguous i.e. all laid out in order in memory. This makes memory access much faster, and aren't we using GPUs to make things faster, after all?

To be a good CUDA programmer requires that you get good with memory.

Imagine a 2D array of rows×cols. Now take every row, and stick them end to end, and you'd have a 1D array of rows×cols elements.
But if we were careful to access the elements, we could still use two indexes i and j to easily access it in a 2D pattern.

```
   int rows = 10, cols = 10;
   float *array;
   array = new float[rows*cols];

   array[i*cols + j]; // access element i,j of the array
```

The last line essential skips over i rows (requiring us to count through i×cols since each row contains 'cols' elements) in order to take us to the row we want, and then counts along that row to the column we want.

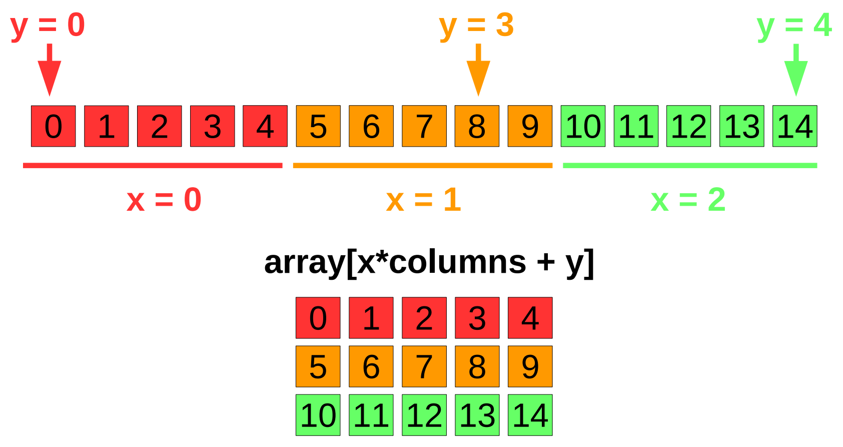


Figure 10: Flattened 2D array - by indexing correctly, we can use a 1D array to hold 2D data. x*cols effectively skips over x rows, and then +y gets us to the correct column.

We can also do the same things using the number of bytes. This is more complicated, but will make it easier for us to understand 2D device memory.

```
int rows = 10, cols = 10;
float *array;
array = new float[rows*cols];

size_t width = cols * sizeof(float); // width of our array

// access element i,j of the array
*( (float*)( (char*)array + i*width )  + j );
```

I'd like to take this opportunity to apologise for the last line of code...
The 'width' of the array is how many bytes are taken up by each row. As for the last bit, I'll explain what's going on:

- array is cast to a character pointer: char*
- we add i*width bytes in order to move along memory to another row
- we then cast everything to a float pointer: float*
- we then increment this pointer with j to get to the column we want
- finally, we dereference that pointer to get the stored value

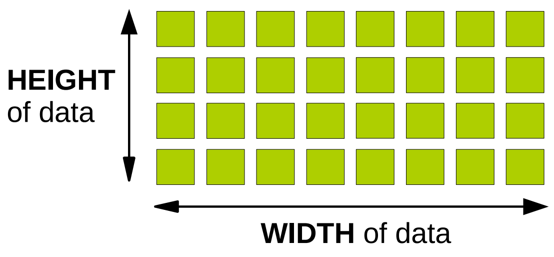
Why do we have to cast to char* ? It assures that the offset i*width is applied to the pointer in 1 byte increments, because a char is one byte.

4.2.2 2D device memory

From NVIDIA CUDA library (found on dem internets):
"For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cudaMallocPitch(). Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays)."

```
// number of data elements for array
int rows = 32*4, cols = 32*4;

// height and width of this array in terms of data
int height = rows;
size_t width = sizeof(float) * cols;

// array
float *d_phi;

// the actual 'width' of array row on device
size_t pitch_phi;

// allocate '2D' memory on device
cudaMallocPitch((void**)&d_phi, (void**)&pitch_phi, width, height);
```

What is the pitch? When we allocate 2D memory with cudaMallocPitch, the resulting memory is usually padded. This is done to improve memory alignment on the device (such that the beginning of rows are aligned to the beginning of the 128 byte = warp*sizeof(float) memory cache line of global 'GDDR5' memory, which improves performance).



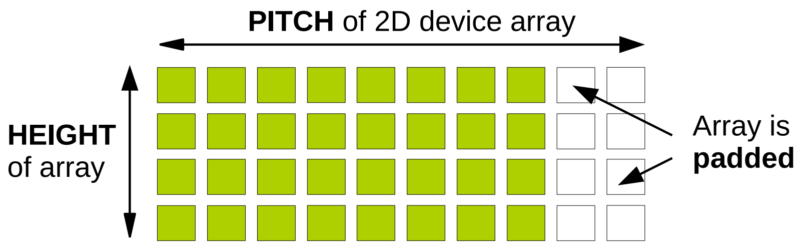


Figure 11: When we allocate memory via cudaMallocPitch, rows are padded to guarantee that the beginning of each row begins at a 128 memory boundary - this generally improves performance when we are accessing our memory in a flattened-2D-like fashion.

The **height** is an integers counting the number of rows, while the **width** is measured in bytes.

We can't predict the pitch. Even two seemingly identical arrays might be given a different pitch! (It happened to me...) We must send the pitch address to cudaMallocPitch, so that it is returned to us and we can use it to access the 2D device array properly.

For every 2D array "d_array1" we should also declare "pitch_array1" (this naming convention stops us getting confused).

How do we access 2D memory from our kernels and device functions? It is quite similar to the awkward memory access I showed for a flattened 2D array:

```
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;

    *( (float*)( (char*)array + x*pitch_array ) + y)  = 123.456;


    // same as above
    *(
        (float*)(
                    (char*)array + x*pitch_array
                )
        + y
     )  = 123.456;
```

This is how we must access 2D memory declared with cudaMallocPitch. Why do we have to cast to char* ? It assures that the offset i*width is applied to the pointer in 1 byte increments, because a char is one byte. Therefore, this assures the pointer arithmetic using the pitch will actually skip i rows, which is what we're trying to do.
To re-quote *"For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cudaMallocPitch(). Due to pitch alignment restrictions in the hardware..."*

4.2.3 2D memory copies

```
  int rows = 32*4, cols = 32*4;
  int height = rows;  size_t width = sizeof(float) * cols;

  float *h_array;     // host arrays
  h_array = new float[rows*cols];

  // for 2D device memory
  float *d_array; size_t pitch_array;
  cudaMallocPitch(&d_array, &pitch_array, width, height);

  // copy from host to device
  assert(  cudaMemcpy2D
          (d_array, pitch_array, h_array, width,
           width, height, cudaMemcpyHostToDevice)
           == cudaSuccess  );

  // copy from device to host
  assert(  cudaMemcpy2D
          (h_array, width, d_array, pitch_array,
           width, height, cudaMemcpyDeviceToHost)
           == cudaSuccess  );
```

We can also copy memory between two host arrays, and between two device arrays.
What's going on?

```
 cudaMemcpy2D
 (destination, width/pitch, source, width/pitch,
  width of transfer, height of transfer, transfer direction);
```

The destination is the pointer to the destination array (similarly for the source), and the width/pitch parameter tells us how many bytes occupy each row of the array. Normally this would be the width = sizeof(type) * cols, but for 2D device memory this is the pitch, since the array is padded for better alignment.
The width of the transfer refers to the width of the rows in terms of data elements, so this is usually going to be width = sizeof(type) * cols (since this is how much data is in each row).

4.2.4 2D memory in practice

So we have a 2D array that we need to access from our kernels. Each thread with grid indices x and y also needs to know nearest neighbour information.

```
 int x = blockIdx.x*blockDim.x + threadIdx.x;
 int y = blockIdx.y*blockDim.y + threadIdx.y;

 float phi_x_y = *((float*)((char*)array + x*pitch_array) + y);

 int next_x, prev_x, next_y, prev_y;
 neighbours(x, &next_x, &prev_x, y, &next_y, &prev_y, rows, cols);
 float phi_px_y = *((float*)((char*)array + prev_x*pitch_array) + y);
 float phi_x_ny = *((float*)((char*)array + x*pitch_array) + next_y);
```

The reason for accessing nearest neighbours with values of prev_x etc. is that we require 2D periodic boundary conditions, so prev_x is not always x-1.

4.2.5 Use Macros

There is a way that we don't have to keep writing out this long, complicated way of accessing our flattened 2D array, and that's by using Macros. These are commands that are read and processed by the *preprocessor* before the code is properly compiled.

```
#define get2D(array,pitch,x,y) *((double*)((char*)array+x*pitch)+y)
```

We can use our macro in our code like this, for example:

```
    get2D(arrayname, arrayname_pitch, 2, 5)
```

and it'll all work like magic, as if you had written

```
    *( (double*)( (char*)arrayname + 2*arrayname_pitch ) + 5 )
```

5 Optimising our GPU code

5.1 Timing Code and Auto-tuning

We could guess that some grid specs are faster than others, but its best to number-crunch: **Auto-Tuning** "EventTime" means computing time, "SimTime" means simulation time.

```
int best_tpb_x, best_tpb_y;
cudaEvent_t start, stop;
float minEventTime=20000.0f;

int actual_rows = rows, actual_cols = cols;
for(tpb_x=4; tpb_x<32; tpb_x+=2)
  for(tpb_y=4; tpb_y<32; tpb_y+=2)
  {
      block_spec = dim3(tpb_x,tpb_y);
      numblks_x = actual_rows/tpb_x; numblks_y = actual_cols/tpb_y;
      grid_spec = dim3(numblks_x,numblks_y);
      rows = numblks_x*tpb_x; cols = numblks_y*tpb_y;
      height = rows; width = sizeof(float)*cols;
      sm_size = sizeof(float)*(tpb_x+2)*(tpb_y+2);
      // *** re-intialise device array, if necessary ***

      cudaEventCreate(&start); cudaEventCreate(&stop);
      cudaEventRecord(start,0);
      float sim_time = 0.0, run_to_sim_time=0.1f;
      float EventTime = 0.0f;
      while(sim_time < run_to_sim_time)
      {
        // *** our simulations, should be mostly kernels ***
        sim_time = sim_time + dt;
      }
      cudaEventRecord(stop,0); cudaEventSynchronize(stop);
      cudaEventElapsedTime(&EventTime,start,stop);
      printf("tpb_x=%d, tpb_y=%d, %.3f \n", tpb_x,tpb_y,EventTime);
      if(EventTime < minEventTime)
      {
        best_tpb_x = tpb_x; best_tpb_y = tpb_y;
        minEventTime = EventTime;
      }
  }
cudaEventDestroy(start); cudaEventDestroy(stop);
printf("Best block x,y: %d,%d \n", best_tpb_x,best_tpb_y);
tpb_x = best_tpb_x; tpb_y = best_tpb_y;

block_spec = dim3(tpb_x,tpb_y);
numblks_x = rows/tpb_x;  numblks_y = cols/tpb_y;
grid_spec = dim3(numblks_x,numblks_y);
rows = numblks_x*tpb_x; cols = numblks_y*tpb_y;
height = rows; width = sizeof(float)*cols;
sm_size = sizeof(float)*(tpb_x+2)*(tpb_y+2);
// *** re-intialise device array before main simulations ***
```

In essence, we are performing short simulations to see what grid specifications run fastest. This doesn't take very long, but makes sure your simulation code will run as fast as possible (2 days instead of 3 days? Quite a difference!)

In this example, we actually redefine the number of rows and cols we're using at the end, so that they match our grid. There may be reasons that we don't want to redefine our rows and cols, in which case we may need additional conditions for whether we can select certain a block size as a best size e.g. is the block size a factor of rows? but best to keep things simple here.
By number crunching in this way, we might very well select block sizes that are not multiples of warps. This might be because our redefinition of the number of rows and cols results in fewer calculations over all.

While large blocks are usually better, there must be some compromise between the number of blocks and the size of blocks. Auto-Tuning will find this compromise.

5.2 A better hardware paradigm

A better hardware paradigm for our Host-Device system includes the cache:

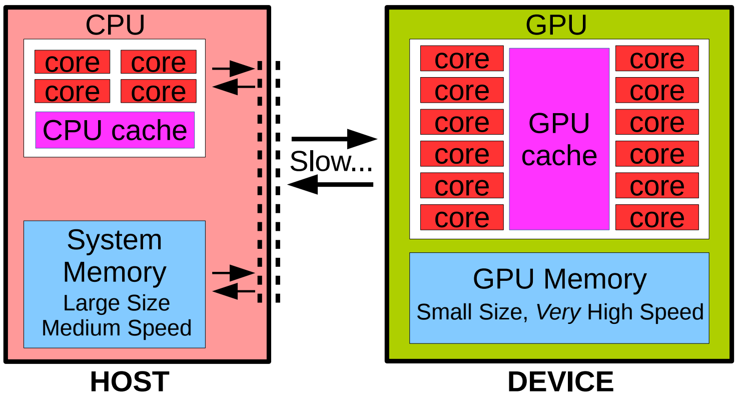


Figure 12: Better hardware paradigm including the cache. Actually, there are four types of 'on-chip memory'/'GPU Cache': registers, shared memory, constant memory (read only), and texture memory (read only). Also, as before, we also have the Global memory ("GPU Memory") that we've been using up until now.

The **cache is on-chip memory** that is similar to RAM. When data is read in from main memory (normal-RAM/system memory for the Host, Global/GPU Memory for the GPU) it will often be stored in the cache, in case it is needed again.
When data is requested, the cache is first checked and if the data is there we have a **Cache Hit** and data can be read in from cache (very fast!), and if it isn't there we have a **Cache Miss** and data is fetched from main memory.

Different types of memory on the GPU that we will look at for this course are:

- Device/GPU Memory, also called global memory (include device variables)
- Shared memory: not much, but on chip and very fast
- Local memory: Thread-private memory, held in L1 and L2 cache

There are actually other types of cache memory that we won't address here: constant memory and texture memory. Texture Memory is actually extremely useful - we can bind 'textures' to global memory so that reads to global are routed through the GPUs texture unit, which helps us exploit data locality in memory. When Constant Memory is read, up to 16 threads reading the *same* data element can benefit from *broadcasting* - one read provides the single data element to all 16 threads. Sorry though, we won't look at Texture or Constant memory...

5.2.1 Device Memory

The device arrays that we have been allocating are stored in GPU/device/global memory.

We can also declare device variables in a .cu or .cuh file (arrays must be of fixed size).

```
__device__ float my_value = 123.456;
__device__ float more_values[123];
```

We can then call these variables from a kernel (so long as they are within scope of that kernel). I used device variables once, but now I just pass my parameters as arguments to my kernels and functions (if you've got loads, you could always put them in a user defined data type). It can be a bit dodgy using device variables that aren't constant, since you have no idea what order they are being read/written from, but it should be okay if they are constants.

5.2.2 Shared Memory

Fetches From and Writes To Global Memory are relatively slow compared to other memory we could be using and compared to calculations.
Although our arrays are stored in global memory, it would be better to read from global memory as little as possible. We can achieve this by reading values into shared memory, and then using that shared memory instead of fetching from global memory. This has several advantages:

- Shared memory is **very fast**, since its on chip
- Shared memory is **available to all threads within a block** i.e. it is shared
- We can get threads to **cooperatively read from global memory into shared memory**

But beware, there's not all that much shared memory! 16KB or 48KB depending on card - this may limit the size of blocks that you can use.

Within a kernel we can declare shared memory like this:

```
__shared__ float my_value = 123.456;
// array size must be known at compile time
__shared__ float more_values[123];
```

and use it in the usual way:

```
    float x = my_value;
    more_values[10] = x;
```

Actually, shared memory declared like this can be indexed in a 2D fashion!

```
__shared__ float even_more_values[10][10];  // hurray!
```

Again, the size must be declared at compile time.

We can also declare **"external shared memory" within a kernel (one per kernel)**:

```
    extern __shared__ float shared[];
```

Notice that the size is not specified. This is a good thing. We can **use an additional kernel configuration parameter to specify the amount of 'external' shared memory** we need:

```
    // amount of shared memory required
    sm_size = sizeof(float)*(tpb_x)*(tpb_y);

    my_kernel<<<grid_spec, block_spec, sm_size>>>(arguments);
```

In this case, if we had a 4×4 block, we have enough shared memory for 4×4 floats.
However, for **extern __shared__** we can only declare one array, and it must be 1D. We can use this shared memory as a flattened 2D array, providing we use the right access pattern:

```
  extern __shared__ float shared[];
  __syncthreads();

  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  // 2D access pattern of 'shared'
  shared[threadIdx.x*blockDim.y + threadIdx.y] =
      *((float*)((char*)array + x*pitch_array) + y);
  __syncthreads();
```

Accesses to shared memory must be **SYNCHRONISED**. This can be done using a __syncthreads() statement. **When threads within a block reach a __syncthreads() statement, they will wait until *all threads in the block have reached that statement*.**
Be careful that all __syncthreads() statements are reachable, or your code will hang forever! Also, don't overuse __syncthreads() or you'll slow down your code.

5.2.3 Local Memory

Within a kernel, we can do what we might do in any ordinary function, and declare local variables:

```
    float my_value;
    float more_values[10];
```

It's good to minimise doing this: although we effectively have as much local memory as we like, it is still better to use less-not-more local memory as it's a bit slow apparently (not sure why it's slow, as it's cached in L1/L2 memory, but its definitely slower than shared memory).
However, it can be used to simplify code, and you should only remove it later when optimising - there are better ways to optimise than writing code that is difficult to follow!

5.3 Shared Memory: HALOs and nearest neighbours

If we want to utilise shared memory in our kernels, we need to take account of the size of our blocks, since shared memory is shared between all threads in a block.
Since we need nearest neighbours, then we need shared memory of size:

```
    size_t sm_size = sizeof(float)*(tpb_x+2)*(tpb_y+2);
```

5.3.1 Threads fetch 'their' data

In our code, we can easily get all the threads with indices x and y to fetch appropriate data, as we did above. Except that since we've declared more shared memory than the size of our blocks, **we need to adjust the indices when referencing the shared memory** (see figure below/next page):

```
  shared[(threadIdx.x+1)*(blockDim.y+2) + (threadIdx.y+1)] =
    *((float*)((char*)phi + x*pitch_phi) + y);
```

This figure may help us understand why we needed the '+1's next to the thread index:

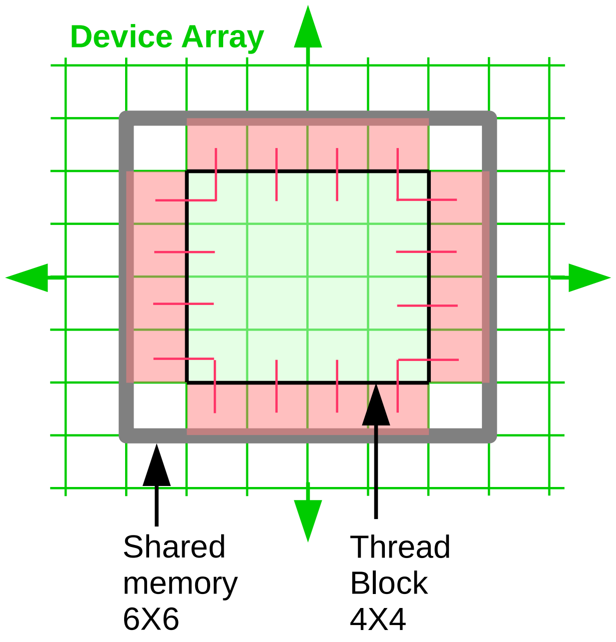


Figure 13: Schematic of using shared memory to read a 2D tile from a global array. Although the shared memory is technically 1D, we can imagine it as 2D. The 4×4 square is our block of threads, the 6×6 square is the shared memory belonging to this block.

The threads with unique grid indices x and y will fetch data from the global array at x and y, but they need to store it in the correct place in the shared memory array: they must skip a row and a column.

5.3.2 Some threads fetch halo data

But what about nearest neighbours? As indicated in the diagram, certain threads will need to fetch information in the **Halo** around the data elements that have already been fetched.
To do this, we will use *if-else* statements as necessary to get certain threads to fetch the extra data that we need. A simple implementation (though we can do a bit better if we get way more complicated) would be as follows:

```
  int  next_x, prev_x, next_y, prev_y;
  neighbours(x,&next_x,&prev_x,y,&next_y,&prev_y,rows,cols);

  if(threadIdx.x==0)
  {
      shared[(threadIdx.x+1 -1)*(blockDim.y+2) + (threadIdx.y+1)] =
        *((float*)((char*)phi + prev_x*pitch_phi) + y);
  }

  if(threadIdx.x==blockDim.x-1)
  {
      shared[(threadIdx.x+1 +1)*(blockDim.y+2) + (threadIdx.y+1)] =
        *((float*)((char*)phi + next_x*pitch_phi) + y);
  }

  if(threadIdx.y==0)
  {
      shared[(threadIdx.x+1)*(blockDim.y+2) + (threadIdx.y+1 -1)] =
        *((float*)((char*)phi + x*pitch_phi) + prev_y);
  }

  if(threadIdx.y==blockDim.y-1)
  {
      shared[(threadIdx.x+1)*(blockDim.y+2) + (threadIdx.y+1 +1)] =
        *((float*)((char*)phi + x*pitch_phi) + next_y);
  }
  __syncthreads();
```

5.3.3 Use Macros

Once again, to help us with referencing the shared memory properly, we can write a macro:

```
#define  getshared(x_incr,y_incr) \
shared[(threadIdx.x+1+x_incr)*(BLOCK_Y+2)+(threadIdx.y+1+y_incr)]
```

(the "\" just allows me to continue writing on a new line).
What this macro does is fetch elements from our shared memory array relative to the (x, y) of threads. So a thread corresponding to (x, y) can fetch data from shared memory corresponding to (x, y) with $x\_incr = 0$ and $y\_incr = 0$. To fetch the neighbouring element for the same x and the previous y, we can use $x\_incr = 0$ and $y\_incr = -1$:

```
    somenumber = getshared(0, -1) ;
```

This also works for reading into shared memory, so for the 'left' halo elements:

```
  if(threadIdx.x==0)
  {
  get_shared(-1, 0) = *((float*)((char*)phi + prev_x*pitch_phi) + y);
  }
```

5.3.4 Save in local memory

So we don't get lost, we can read the shared memory into local memory so we can give the values some better names. This is good when developing code, sometimes, because it can help you avoid typo bugs.
Of course there is some performance penalty, but on the other hand we don't have to worry about synchronising everything all the time (so there may not be such a performance hit, actually - case by case).
Either way, **we still have a speed-up compared to reading directly from global into local, because we got the threads to cooperatively fetch the data from global memory into shared memory - from the point of view of each thread, there have been fewer reads from global memory**.

```
float phi_x_y, phi_px_y, phi_nx_y, phi_x_py, phi_x_ny;
// Read the data from shared memory into local memory
phi_x_y=shared[(threadIdx.x+1)*(blockDim.y+2) + (threadIdx.y+1)];
phi_px_y=shared[(threadIdx.x+1-1)*(blockDim.y+2) + (threadIdx.y+1)];
phi_nx_y= get_shared(shared, +1 ,0);  // can use our macro of course!
phi_x_py= get_shared(shared, 0 ,-1);  // can use our macro of course!
phi_x_ny=shared[(threadIdx.x+1)*(blockDim.y+2) + (threadIdx.y+1+1)];
```

When you've developed your code, you may want to remove these superfluous reads into local memory, and see if you get an additional (and worthwhile) speed up.

5.4 Efficient data fetching

This section is about **writing cache-friendly code**: if data can be found in the cache instead of in global memory, it can be read in much faster.
Due to the size of the **Cache Line**, reading in some data from global memory can give us access to lots of other data for free! But are we making use of that?
The Rule: **Use Structures of Arrays, Not arrays of structures!**
This will allow **Coalesced Access** of memory by threads.

5.4.1 Cache Lines

When data is read from main memory an entire Cache Line is transferred (this is also true when writing to global memory).
For GPUs, the size of the cache line is 32 floats, which is the size of a warp when we are working in floats. In a kernel, if I read a single float from global memory into local memory (e.g. reading 1 element of a float array) then 32 floats of data are actually transferred. This means that I can actually read 32 floats in the same time as reading 1 float. Phew!

To be more correct, whenever data is requested, the cache line is checked first in case the data is in the cache memory, and if it is there, then it is loaded from there: very fast. This is called a **Cache Hit**. If the data is not there, then we have a **Cache Miss**, and the data is then is read from global memory. It will then be in the Cache memory afterwards, along with a load of other data that may have be read in when the entire cache line containing the requested data was transferred.
There is only so much Cache Memory, not very much in fact, so data that is used regularly is kept in there at the expense of data that is not used very often (there are other ways to organise priority of data storage in cache, but this will do for us now). These optimisations happen in the background, but we can program in a Cache-friendly way to improve our code.
We can **exploit data locality to write cache-friendly code**.

5.4.2 Array of Structures (bad...)

Let's take a simple structure with just two fields:

```
typedef struct planet
{
    float mass;
    float radius;
} planet;
```

(Side note: if using structure like this, you may need to consider memory alignment for performance, which would require __align__(n) after 'struct', where n is number of bytes).
Then perhaps in our main code we make an 'Array of Structures':

```
planet *d_planets;
size_t width = sizeof(planet)*cols;
cudaMalloc((void**)&d_planets, width);
```

If we consider how we might access fields from this array in a kernel, we might come up with something like this:

```
int x = blockIdx.x*blockDim.x + threadIdx.x;

float mass = planet[x].mass;
float radius = planet[x].radius;
```

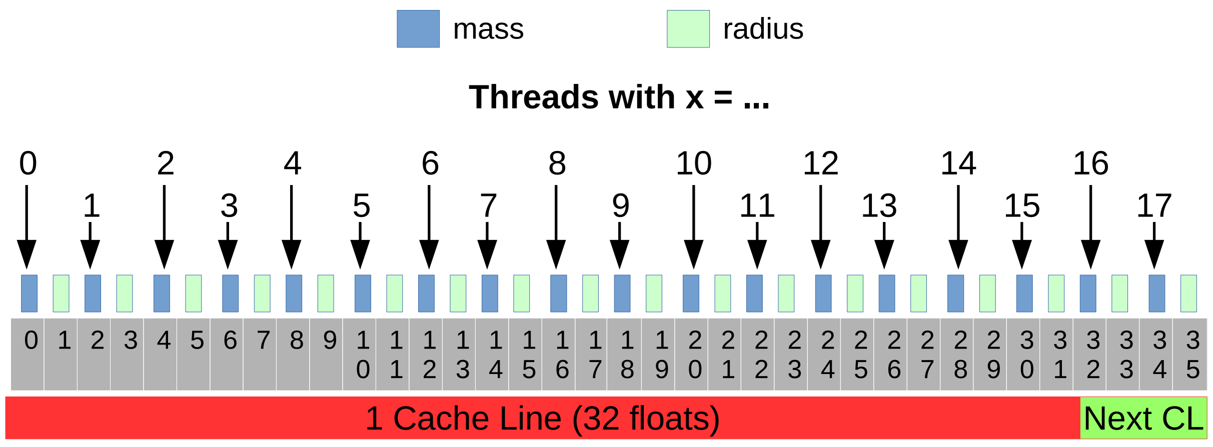


Figure 14: Schematic of reading data from an Array of Structures - not efficient

We can imagine that the different threads access the memory like this:
and then once all of the threads have read in the mass field, they would then have to all read in the radius field. It'll work, but **the problem is with efficiency** (disclaimer: there may be other padding between each array element, but we see the point here anyway, I hope).

If we have 17 threads (for some reason) and only 1 cache line's worth of cache memory (for some reason) then we see that we have to read in a cache line for threads 0 to 15, and then read in another cache line for threads 16 and 17. Next, though, we'll want to read in the mass field, so threads 0 to 15 will have to read the first cache line from global memory again, and then threads 16 and 17 will have to read the second cache line in. We've just done 4 cache line transfers from global memory.

This is a shame, because threads 1 to 15 had access to the data they needed just by having thread 0 read in its data: an entire cache line was transferred. But because of our structure, that cache line contained lots of data that we didn't use, which is a waste. **It's wasteful because we're not utilising coalesced access, but doing staggered access**.

5.4.3 Structure of Arrays (better...)

Consider that we now make a structure of arrays, like this:

```
 #define N 18
 typedef struct planet
 {
     float mass[N];
     float radius[N];
 } planet;
```

and create a device pointer like this:

```
planet *d_planets;
size_t width = sizeof(planet);
cudaMalloc((void**)&d_planets, width);
```

Then we would probably write the kernel code like this:

```
 int x = blockIdx.x*blockDim.x + threadIdx.x;

 float mass = (*planet).mass[x];
 float radius = (*planet).radius[x];
```

Our data accesses would then be like this:

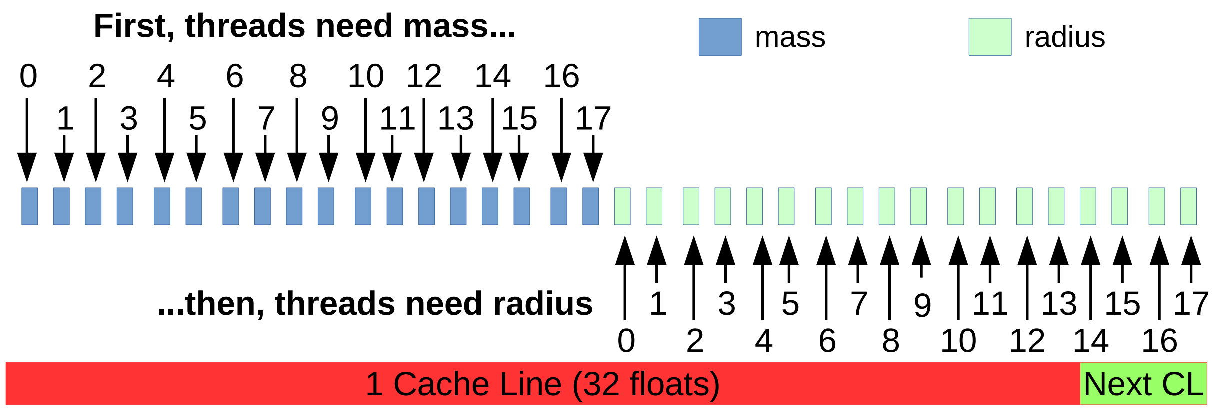


Figure 15: Schematic of reading data from a Structure of Arrays - efficient!

In this simple example, we would need to read in two cache lines from global memory into the cache, but that's a lot better than four. In fact, it might be *a lot* better than four, since we don't have to keep throwing away cache lines in the cache in order to get new ones from global memory. This can potentially make a big difference in our programs.

5.4.4 The Best Way

The best way to write cache-friendly code and ensure coalesced access is: **just create different arrays for each variable, and don't bother with structures**.
e.g. don't have an array of vectors with fields x, y, z, just have three arrays: one for x, one for y, and one for z - this is by far the most efficient way to GPU program, and it's exactly what we did for ϕ and μ in our diffusion simulation.

5.5 Boundary Conditions

Implementing Boundary Condition might not be something you would think to do differently on a GPU, but because we want to stick to the principle of **SIMT: Same Instruction Multiple Threads** and avoid branching, we need to reconsider the way we might implement boundary conditions.

It might be quite normal to implement a boundary condition with a conditional *if-else* statement. But when we consider that if any single thread in a warp follows an IF (or ELSE) then all threads must follow that IF (or ELSE), we see that we might be doing a lot of useless work - we don't want all threads to follow both branches, but it happens anyway because of the hardware. Since it's often the case that only a few data elements require boundary conditions to be applied, we could perhaps do a better job than to build boundary conditions into our code with if-else statements.

We use separate kernels to implement boundary conditions to minimise branching.

The physics

We won't worry about the branching that due to our 2D periodic boundary conditions.
Instead, we're going to add two surfaces into our diffusion simulation, so that the fluid is a film. This will require

- an **additional surface term** contributing to the chemical potential
- finding a way to **calculate gradients** of quantities at these surfaces
- implementing a **no-flux** condition at the surfaces

The first task is easy. Go into the provided code that returns the chemical potential, and uncomment the line that makes a call to the function that returns the surface energy.
The rest is slightly more involved, so trust me when I say that **the principles of what we are going to do are very useful to know**.
The other two tasks are going to require something I call **'virtual cells'**, and they are generally very useful for implementing boundary conditions (in any programming language).

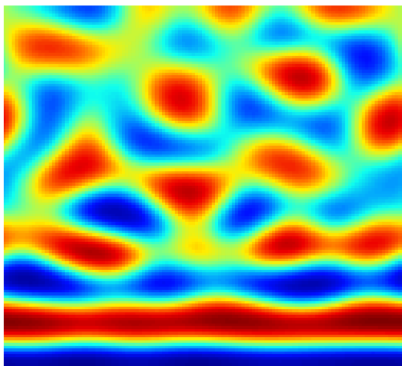


Figure 16: A lovely picture of the effects of some surfaces - the lower surface prefers blue, the upper surface doesn't mind.

5.5.1 Virtual Cells

What are virtual cells?
Do not get them confused with anything to do with CUDA. They are simply cells in our array (representing an area of binary fluid) that don't contain 'real values' corresponding to the simulation results, but are used as part of our the calculations (so we wouldn't want to plot the values in these virtual cells).
We will need them to implement a no-flux condition and calculate gradients at the surfaces. In this case, the surfaces will be represented by the imaginary boundary between our 'real simulation cells' and our 'virtual simulation cells'.
We should pick our surfaces to run *row-wise* along our arrays, so that the elements in the virtual cells are contingious for each surface (so memory accesses of these virtual cells are coalesced i.e. cache friendly code).

TASK: update your code to so the arrays for ϕ and μ have virtual cells:

- increase the height of your arrays to make space for the two extra rows of virtual cells. We will refer to the number of rows of 'real' cells with 'rows', so I suggest you do not simply add 2 to the value of rows, but increase the value of 'height':

```
int cols = 32*4, rows = 32*4;
size_t width = cols* sizeof(float); int height = rows+2;
```

  Now the virtual cells exist at $x = 0$ and $x = rows + 1$, and the real cells exist between and including $x = 1$ and $x = rows$. All the memory allocations on the host and device should take care of everything properly, provided they use 'height' and 'width' as defined above.

- for the kernels we have already been using, we should make sure to 'skip' the first virtual row (we don't have to skip the last one: our Grid Specs will ensure that no thread is charged with changing those values)

```
int x = blockIdx.x*blockDim.x + threadIdx.x+1;
int y = blockIdx.y*blockDim.y + threadIdx.y;
```

- where we previously had periodic boundary conditions in the x-direction, we will now not need any such conditions: for any given x, the neighbours are the cells either side of it (the periodic boundary conditions for y remain the same)

```
*next_x = x + 1; *prev_x = x - 1;
```

- any initialisation function, or printing function, will now need to be adapted so that it 'skips' the first virtual row. Where a routing would have read:

```
// for(int i = 0; i < rows; i++) - old code
for(int i = 1; i < rows+1; i++) // new code
```

  This is why it is useful to refer to the number of rows of real cells with *rows*, even though the height of our array containing those values is now $rows + 2$

5.5.2 Boundary conditions for a surface - gradients

We've not explicitly had to calculate many gradients because these calculations are done inside some of the functions I provided, but to understand why we need to make adaptations due to the surfaces we need to take a look at calculating gradients with finite differencing.
Consider the following problem of calculating a central difference gradient near a surface:

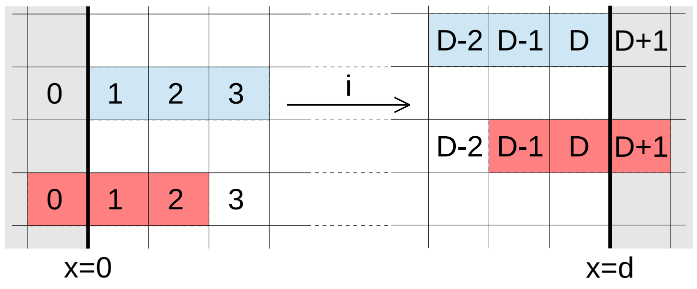


Figure 17: Central Differencing near a surface - how are we going to do this?

For 'bulk' cells not directly next to a surface, we can calculate the gradient using simple central differencing:

$$\nabla\phi|_2 = \frac{\phi_3 - \phi_1}{2\Delta z} \tag{4}$$

But what about for cells next to a surface?

$$\nabla\phi|_1 = \frac{\phi_2 - \phi_0}{2\Delta z} \tag{5}$$

In this case, the cells containing ϕ_0 are virtual cells: the data in them is not part of the simulation. If we didn't have these virtual cells, we would need to do forward differencing at this surface instead. Using forward differencing to second order accuracy (to help remove artefacts of using forward differencing) we have:

$$\nabla_f\phi_1 = \left(-\frac{3}{2}\phi_1 + 2\phi_2 - \frac{1}{2}\phi_3\right)/\Delta z. \tag{6}$$

Imagine the kernel code though: it would need an IF-ELSE statement: *if we are in the bulk then use central differencing, else use forward differencing.* The memory fetches would need to be different in that case... things get messy quickly.

But consider if we set $\nabla_c\phi_1 = \nabla_f\phi_1$ then we find that if we set

$$\phi_0 = +3\phi_1 - 3\phi_2 + \phi_3. \tag{7}$$

then we can just do central differencing for cells next to the surfaces, and we don't need to treat any of the cells any differently. Similarly, for the other surface:

$$\phi_{D+1} = +3\phi_D - 3\phi_{D-1} + \phi_{D-2}. \tag{8}$$

So if we set the values in our virtual cells ϕ_0 (and ϕ_D) correctly, we don't need to make exceptions at the surfaces.

To implement these conditions, **we ought to call a kernel, prior to calculations needing $\nabla\phi$, in which we set all of the virtual cells to appropriate values**.

5.5.3 Boundary conditions for a surface - no flux

What about implementing a no flux boundary condition at the surfaces?
Without going into the physics too much, this requires that the row of virtual cells on the outside of the surface have values set equal to the row of cells on the inside of the surface.
So after we calculate the chemical potentials everywhere, we need to set the virtual cells in the chemical potential array as follows:

$$\mu_0 = \mu_1 \tag{9}$$
$$\mu_D = \mu_{D+1} \tag{10}$$

5.5.4 Kernels for BCs

Finally, to apply our boundary conditions, we need kernels that work on the device arrays. Take a look at the extra kernel functions you've been given in the file run_sim.cuh. They are called 'chembc' and 'phibc'.
TASK: call these kernels from your main program in order to apply the boundary conditions.
You'll need to apply the boundary conditions for ϕ before calculating the chemical potential, and apply the boundary conditions on the chemical potential before updating ϕ.

What Grid Specifications should you use to call these kernels? We're only working on a couple of rows from the array, so wouldn't it make sense to choose grid and block specification to match that? I would suggest:

```
<<<dim3(2,1),dim3(1,cols)>>>
```

5.5.5 PRINCIPLE: Configure kernels appropriately!

Kernels should be configured with an appropriate grid for the work they are doing, hence any one simulation may have many different grid specifications for many different kernels doing many different bits of work.

6 The End

Now you know how to CUDA! If you need to consult me about CUDA, either for CUDA specific advice or for advice on how you could parallelize your program, you can contact me.