GPGPU Practical

Practical session 1 – Getting your feet wet with some toy problems

This practical session will cover a wide range of topics starting with some basics and progressing to some toy programs:

1. Introduction
2. c/c++ Hello World. (you probably should know this one!)
3. Introduction to cluster computing (this may be new to many of you)
4. Hello World on the cluster
5. CUDA Runtime API
6. Vector addition

1 Introduction

We will be running the practicals in the CS senior computer laboratory, you should be able to login using you're Linux account (if not or you need help speak to Chris). The computers we will be using should have a CUDA enabled GPU. We have also have access to the ICTS High Performance Computing Cluster, but more on that later. We will be developing and testing locally and then testing on the cluster. We expect you to have at least a basic understanding of **c/c++, Linux (console), ssh.**

I will be giving you plenty console commands, they will be in the format below:

```
$ command
```

Where `$` represents the prompt and is not part of the command to be run.

Generally for the practicals I will give you the vast majority of the code already written and you will just have to complete a few sections of code.

The starting code and documents for this Practical, can be downloaded at people.cs.uct.ac.za/~claidler/GPGPU_Ses01.tar.gz and extracted.

```
$ wget http://people.cs.uct.ac.za/~claidler/GPGPU_Ses01.tar.gz
$ tar xfz GPGPU_Day01.tar.gz
```

This will create the directory GPGPU, here you will find this document and a number of subdirectories, containing all the parts of the first practical session.

2 c/c++ Hello World.

If you are doing this workshop you should know this but I have included it to help anyone who may be a bit behind or rusty. If this is new to you get ready for a steep learning curve over the next couple days! You can happily skip this section if you are happy in c/c++, just compile the hello world programs in the `01_Hello_World` subdirectory as you are going to need them in later sections.

2.1 Code

For this hello world section we will be working in the `01_Hello_World` subdirectory.

```
$ cd 01_Hello_World
```

The source code of the program consists of **source** and **header** files. Source files contain the code and header files contain function declarations and class definitions.

First we will make a c hello world program. There is only one source file `main.c`, have a look at it with your favourite editor, personally I like kate I think it is on most of the machines.

```
$ kate main.c &
```

Or

```
$ gedit main.c &
```

Or you can always be old school...

```
$ vim main.c
```

The code is fairly simple and I'm not going to explain it.

2.2 Compile and link

To make an executable the code needs to be **compiled** and **linked**. The various source files are compiled to object files, these will then be combined by a linker to create a binary. These days most compilers can do linking for you. In our c hello world program there is only one score file and we can compile and link with:

```
$ gcc main.c -o HelloWorld
```

This produces the binary `HelloWorld` which can be run with

```
$ ./HelloWorld
```

and should give the output:

```
Hello World
```

Now lets look at something a bit more complicated, two source files. I have included an OO (Object-oriented) c++ hello world program. This has the class `HelloWorld` defined in `helloWorld.h` and with code in `helloWorld.cpp` and the main program is in `main.cpp`, these can be compiled:

```
$ g++ -c helloWorld.cpp -o helloWorld.o
$ g++ -c main.cpp -o main_cpp.o
```

The `-c` flag specifies that no linking will be done. This will create the object files `helloWorld.o` and `main_cpp.o` these can be linked:

```
$ g++  main_cpp.o helloWorld.o -o OO_HelloWorld
```

resulting in the binary `OO_HelloWorld` that can be run:

```
$ ./OO_HelloWorld
```

This should give the output:

```
Hello World!  from a Class.
```

2.3 Makefiles

Now that you know how to compile, link and run a program, you wont have to do it manually, it can get a bit tedious! Makefiles are a way to manage the build process, we have included them for the rest of the practicals. With our makefiles you can simply compile and link everything with the make command:

```
$ make
```

If you need more details, there is plenty material out there on the web, you are just going to have to Google it!

3 Introduction to cluster computing

The computers you are currently working should have a GPU but for some real world compute problems this type of GPU may not be sufficient. So we are going to testing some of our code on a GPU clusters. The cluster we will be using is the University of Cape Town ICTS High Performance Computing Cluster (http://hex.uct.ac.za/). Have a look on the web and see what info you can find on the computers you will be working on.

Some of you may have worked on clusters but I believe most wont have, so I will go over some general details of how to compile and run your code on a cluster.

The examples for the rest of the practicals will assume you are working on the ICTS cluster.

3.1 ICTS Cluster

3.1.1 General structure of a cluster

Generally a cluster will consist of a head node and a number of other nodes that will do the actual computation. When you access the cluster generally you will log into the head node and from here you will launch jobs that will be scheduled to run on the various compute nodes. Generally the head node is configured to be similar to the compute nodes, thus you can compile you're code on the head node and run it on the various compute nodes. Usually there is some form of common file system so you can seamlessly get and retrieve data from the various nodes.

3.1.2 Logging in

You should have been supplied with a temporary user name and password for the UCT ICTS cluster, this you will use for the rest of these practicals. If you have further research that needs to be conducted on the cluster later you can apply for an account here.

To log into the head node of the ICTS cluster:

```
$ ssh -X username@hex.uct.ac.za
```

You should have just logged onto the **head node** of the cluster.

3.1.3 Gather Information

Lets have a look what this head node is ...

First what OS are we running?

```
$ uname -a
```

We have Linux so what distro are we using?

```
$ cat /etc/*release
```

Great now some specs of the hardware:

```
$ lscpu
$ lspci or on hex /sbin/lspci
$ cat /proc/cpuinfo
$ cat /proc/meminfo
```

From all this you should be able to gather some info on the computer you have logged onto, one of the main things you should notice is that it is a virtual machine and has no GPU. So we **definitely can not** run our CUDA code on it! Compare its specs to the specs of the machine you are sitting in front of. Do you have a CUDA enabled GPU?

3.1.4 Copying data to the cluster

Before we get into writing and compiling code on the server, here is a hint to make life a bit easier. Do as much work on your local machine as possible, if possible code, debug and tweak locally. Only run on the server when you have to.

If you do not know about `scp` you can learn about it here, I will not go into to much detail except:

```
$ scp file.xyz username@hex.uct.ac.za:/home/username/
```

This is the standard method of getting files around, personally I prefer **sshfs**, which I will outline below.

sshfs mount

When I am working on a remote server like this I'm not the biggest fan of having to write everything in vim and copy files using scp. So what I do is mount the remote directory using sshfs. This allows me to use my local file browser and favourite local editor while working on remote files.

To make a sshfs mount open a console on your local machine and do the following:

```
$ cd ~
$ mkdir -p ~/remote/Hex
$ sshfs username@Hex.uct.ac.za:/home/username/ ~/remote/Hex/
```

This mirrors all changes in the local directory `~/remote/Hex/` to you're home drive on Hex. If you want to stop the mirroring use:

```
$ fusermount -u ~/remote/Hex/
```

But I would leave it mounted for the moment, so if you just blindly unmounted `~/remote/Hex/` please remount it.

3.1.5 Running a program on a cluster

You should have logged onto the head node of the cluster, now how do you actually run something on the cluster? Naturally time on clusters like this is in high demand, so user 'jobs' are put in a queue and scheduled, this allows efficient use of the compute resources. On the ICTS cluster the resource manager used is Torque (PBS) which monitors the status of the cluster and controls/monitors the various queues and job lists. This is tied into a batch scheduler. ICTS uses (Maui) as its scheduler which decides how a job should be run and its placement in a queue.

It is important to note the `/home` partition on the head node is NFS mounted (i.e. common) to all worker nodes, regardless of series.

The command you will use to submit a job to the cluster is `qsub`. Qsub communicates with the batch server and informs it you have a job to be run on the cluster. Once submitted the scheduler decides where in the queue to place the job and notifies the batch server. Qsub's return status will be whether the job was successfully submitted or not. You may use STDIN for input to `qsub`, but more commonly we use an command file.

The most common commands you will be using are:

`qsub <script_name>`	Submit a job
`qstat -r`	Monitor a job
`qdel <PBS_JOBID>`	Cancelling a Job

3.1.6 Submit a job

The most common way to use qsub is to uses a script file. The script file is a specialized shell script, containing a number of **torque parameters** that define the environment your job will run in and the commands you want to run. First have a look at `qsubScript_01` in subdirectory `02_Cluster` it should look something like the script below.

```
1  #!/bin/sh
2  #
3  # This is an example script
4
5  # These commands set up the Grid Environment for your job:
6  #PBS -N gpu_test_01
7  #PBS -l nodes=1:ppn=1:seriesGPU,walltime=00:00:20
8  #PBS -q GPUQ
9
10 # Now we execute some normal shell commands
11
12 # Print the date and time
13 date
14
15 # Host name of the node we are executing on
16 hostname
17
18 # where is the script running?
19 pwd
```

This is a shell script and if you run it on Hex (the head node):

```
$ ./qsubScript_01
```

It should give you something like:

```
Mon Apr 29 14:20:00 SAST 2013
srvslshpc001
/home/username/02_Cluster
```

This is running on the head node, **Hex** (which has host name srvslshpc001). The comments on line 6, 7 and 8 are special PBS directives. These will be ignored when run as a normal shell script, as the are comments, but will be passed to PBS as command line switches when run with `qsub`.
Line 6 specifies the name of the job (gpu_test_01).
Line 7 the resource list consisting of; list if nodes (nodes=srvslsgpu001) **or** number of nodes (nodes=2) to use number, next is the number processors (ppn=1) cluster (seriesGPU) and maximum time to use them (walltime)
Line 8 is the queue to use(GPUQ).

For more detailed TORQUE/qsub reference have a look here or try this.

Now to run this script on the cluster, use `qsub` (on Hex):

```
$ qsub qsubScript_01
```

This should give us output something like

```
880300.srvslshpc001
```

Where **880300** is the job ID. Running `qsub` with this script, adds this script as a job in the GPU queue and it will run the script on one of the GPU nodes (srvslsgpu001). When the

script is run anything written to `stderr` is written to `gpu_test_01.e880300` and to stdout is written to `gpu_test_01.o880300`, where the number will be the relevant job ID's. Remember that your home drive is NFS mounted so once written on the compute node, you will be able to read the output on Hex. These files will be written to the location `qsub` **was called from**, so in the previous case directory you are in. It may take some time for your script to get run, especially seeing as there are probably 25 other people submitting to the queue right now, so wait a couple seconds then have a look in the directory and the output files will hopefully be there. If not you can use `qstat` to try figure out what is up. Hopefully `gpu_test_01.e880300` is empty and `gpu_test_01.o880300` should look something like:

```
1  Tue Apr 21 14:21:00 SAST 2015
2  srvslsgpu001
3  /home/username
```

Great we ran some shell commands on the server, and got some output. It is important to note **it isn't interactive**, you write up a script add it to the queue and it gets run at some point in the future and you get the output. Now lets have a closer look at the output we got, note the directory the script was run in was **your home directory**. If you want to change to the directory where the script was called from on the head node you can put `cd $PBS_O_WORKDIR` in the script see `qsubScript_02`, have a look at the output generated when you run it.

```
$ qsub qsubScript_02
```

Have a look at the output, especially the last line what about if we do:

```
$ mkdir -p subdir
$ cd subdir
$ qsub ../qsubScript_02
```

Note where the output files are placed and where the script was run, this makes a big difference when you are trying to run your programs in various places on your home drive. It is **often advisable to use absolute paths.**

3.1.7 Monitor a job

The command to use here is `qstat`, it will list the jobs you have active. For a full reference have a look here or good old `man qstat`.

3.1.8 Cancelling a Job

The command to use here is `qdel <PBS_JOBID>`, the job will be sent TERM and KILL signals killing the running processes. For a full reference have a look here or good old `man qdel`.

3.1.9 Get info on the server

We can use what we have learned to get some info on the GPU cluster. Have a look at `qsubScript_03` it has a the commands we used in 3.1.3 and we can now use it to get some info on the GPU nodes.

```
$ qsub qsubScript_03
```

3.2

There is plenty more to know and I'm not going to cover it here, for reference's have a look here:

http://Hex.uct.ac.za/JobSubmit.html

http://docs.adaptivecomputing.com/torque/2-5-12/help.htm

If you need more details, there is plenty other material out there on the web, you are just going to have to Google it!

4 Hello World on the cluster

For this section you are going to have to compile and run a hello world program on the cluster. We will be working in the `03_Hello_World_Cluster` subdirectory, which you will see is basically empty. In the previous sections you were given every detail, here we just give you some task to do, you can refer to the previous sections or Google if you need help.

- Change to the `03_Hello_World_Cluster/src` subdirectory.
- Write a basic c or c++ hello world program.
- Makes sure it compiles and runs on your local machine.
- Copy your code across to the cluster
- Login to Hex
- Compile your program **on Hex** (you will have to '`make clean`' first if you copied across object or binary files!)
- Copy the binary to `03_Hello_World_Cluster` on Hex
- Edit `qsubScript` so that it will run your binary
- Use `qsub` to run your hello world application on the GPU cluster

5 CUDA Runtime API

In this section we will do a little example using the CUDA Runtime API. For work like this your biggest aid will be the official documentation, which can be found here http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

We will be working in the `04_Device_Query_API` directory, where we will be writing a program to find out some of the CUDA specs and the specs of the CUDA device(s). I have written up a skeleton and you will just have to look up the relevant API calls, and add them to the code.

You will first get everything working on you're local machine and then when everything is good you will run it on the cluster.

5.1 Build

On you're local machine you can simply build the project with:

```
$ make
```

This should give output something like:

```
g++ -m64 -O3 -I/usr/local/cuda/include -I. -I..
-I/home/claidler/Day_01/04_Device_Query_API/../../common -o release/main.o -c
main.cpp
nvcc -link release/main.o -o release/deviceQuery -lgomp  -L/usr/local/cuda/lib64
-lcudart
```

Notice we are compiling the `main.cpp` with g++ and linking with libcudart using `nvcc`. For this program we could link with g++, but later we will need to link with `nvcc`, we will discuss this in the next section. libcudart is included as part of the CUDA toolkit.

5.2 Run

I normally include a **run** clause in **my** make files, this compiles and if successful runs the binary, If you try and run the program:

```
$ make run
```

You should get something like:

```
./release/deviceQuery
 --=== GPGPU workshop CUDA Device Query (Runtime API) ==--

ERROR: The variable 'deviceCount' has not been instantiated. Pleas correct line
78 in 'main.cpp' by using an API call to get a relevant value.
make: *** [run] Error 1
```

5.3 Fix error

Have a look at the error and it will give you some insight! So we go to line 78 in `main.cpp` and we see something like

```
76  //TODO 1 - Find an API call to calculate the number of devices . Set: deviceCount
77  {
78    deviceCount = -1*__LINE__;
79  }
```

All the edits you have to make are indicated with a **TODO** comment, you should only have to

edit code in between the curly braces following the **TODO** comment. In this case we have the line `deviceCount = -1*__LINE__;` the idea is to find out how many GPU's there are. You will see I have set `deviceCount`. The value it is set to is `-1*__LINE__` this is just a junk value I use to indicate that `deviceCount` hasn't been correctly instantiated yet. `__LINE__` is a standard predefined compiler macro, and I am using a negative value to indicate an invalid value that will cause the error to be printed.

What you need to do is set `deviceCount` to the correct value. For this we use the API. This is easily done simply open up the on-line reference and in the first section we find:

```
cudaError_t  cudaGetDeviceCount ( int* count )
             Returns the number of compute-capable devices.
```

Great follow the link and you will get a description of cudaGetDeviceCount. So use the API call and do some error checks and you can get something like:

```
  //TODO 1 - Find an API call to calculate the number of devices . Set:
deviceCount
  {
    // API Call
    cudaError_t cudaStat = cudaGetDeviceCount(&deviceCount);

    // Error Checks
    if (cudaStat != cudaSuccess)
    {
      fprintf(stderr, "Error %s at line %d in file %s\n",
              cudaGetErrorString(cudaStat), __LINE__, __FILE__);
      exit(EXIT_FAILURE);
    }
  }
```

Note there is a macro `CUDA_CHECK_RETURN` in `common/GPGPU.h` to do the error checks and so this can become:

```
  //TODO 1 - Find an API call to calculate the number of devices . Set:
deviceCount
  {
    CUDA_SAFE_CALL(cudaGetDeviceCount(&deviceCount),"Failed to get device
count using cudaGetDeviceCount");
  }
```

If you now try run it on you're local machine you should get something like

```
 --== GPGPU workshop CUDA Device Query (Runtime API) ==--

There are 1 CUDA enabled devices on this node
ERROR: The variable 'driverVersion' has not been instantiated. Pleas correct
line 83 in 'main.cpp' by using an API call to get a relevant value.
```

Great you solved the first sub task / TODO. Before you continue with the rest lets try run this on the cluster to see how we do that.

5.4 Run on cluster

Lets try run our incomplete code on the cluster, **note that most of the programs from here will require files found in the common directory**. I would advise mounting Hex and copying the entire GPGPU directory to your home directory on Hex. If you build the code on you're local machine the binaries will have been compiled and linked with local libraries so **always make sure you clean the project** when you are switching computers.

```
$ make clean
```

You should be able to compile on the head node with:

```
$ make
```

Once it has compiled successfully you will find the compiled binary (`deviceQuery`) in the current directory.

Now if you try run this on **Hex**

```
$ ./deviceQuery
```

You will get the error:

```
ERROR: Failed to get device count using cudaGetDeviceCount [ CUDA driver version
is insufficient for CUDA runtime version at line 72 in file
main.cpp ]
```

What's wrong here? You are trying to run a compute program, specifically GPU compute program, on the head node. You do not want to do this, it is a sure fire way to get an aggravated e-mail from an admin telling you to stop abusing the head node. Again **do not run you're programs on the head node!** Luckily in our case we are trying to use the GPU so it will probably error anyway. You want to **submit a job to the cluster** to run the program, so add the relevant lines to the `qsubScript` and submit it to the queue. You should get similar output / error message as you got on you're local machine.

5.5 API Calls

Now the rest is up to you, find all the TODO comments and fix them, most of this will be done with API calls. When all is done the code should compile and run with no errors. TODO 7 and 8, may cause some trouble, speak to a tutor if you need help.

5.6 RUN on cluster

Once you have the code finished try run it on the cluster to to see what resources there are on the cluster. Note there are two nodes in the seriesGPU cluster, try figure out a way to specify winch node you're code gets run on.

6 Vector addition

In this section we will get into the first real CUDA code with the simple case of vector addition. This will introduce you to CUDA *grids* , *blocks*, *threads* and *kernels*.

Consider we have two list of numbers and we want to sum corresponding elements to create a new list, the classic case of vector addition. The addition of a single pair of corresponding elements is independent of all others and can thus be performed in parallel, this is perfectly suited to the SIMD model.

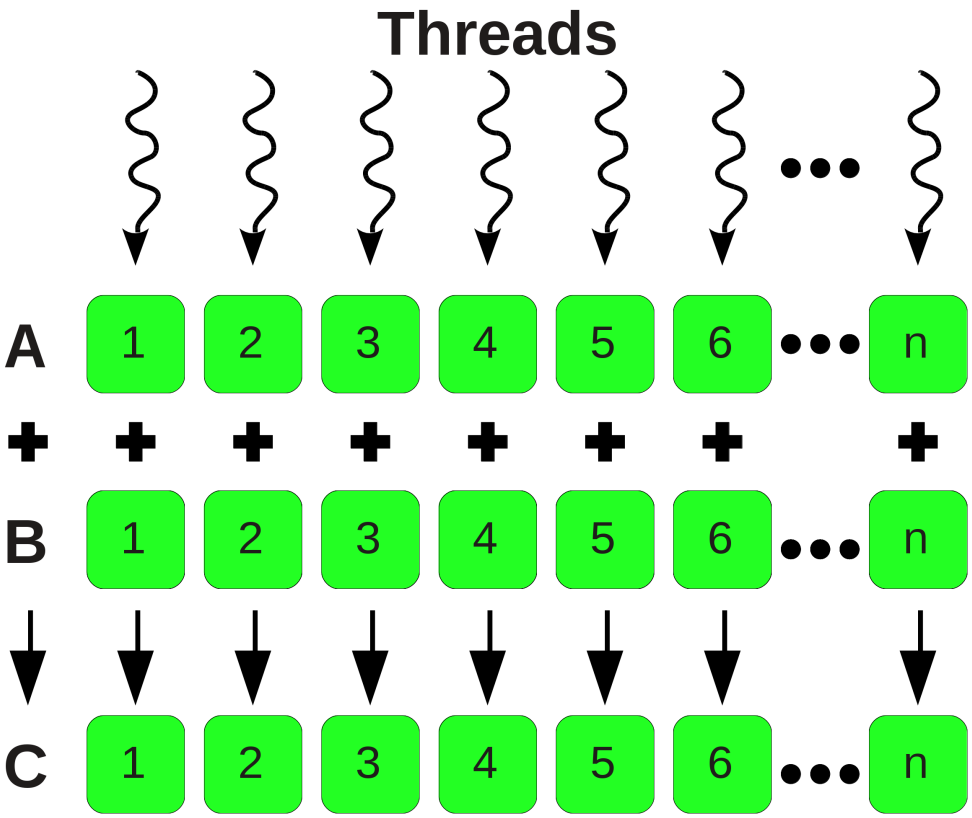

Illustration 1: Vector addition

For this section we will be working in the `05_Vector_Add` directory. Here you will find a couple of source files and a make file, you will be editing `vectorAdd.cu`. Again all the sections you will have to edit are marked by a `TODO` comment. Again this code relies on some code form the common directory, notably I use a TODO macro to note the location of of the next section to be completed, once dealt with you can remove or comment out the relevant call to the macro. For most of the initial sections you should only have to add code to the braces below the comments. I will briefly describe the task here and you will find other details in the comments.

Again when you are writing CUDA code you should find the CUDA C Programming Guide and toolkit documentation a great asset.

6.1 Specify which CUDA device to use

To minimise contention you must specify which GPU you will be using, to do this pleas flip a coin, one side device 0 the other device 1.

6.2 Allocate device memory

You will have to allocate memory on the device, it might be a good idea to do some error and sanity checks as well.

6.3 Copy input data from the host to the device

You will have to copy the data that is in host memory to the device memory.

6.4 Set the dimensions of the blocks and grid

Our data is one dimensional so set up a 1D block and grid structure, you are going to have to decide on your block dimension, try 1D 256 element block. Note that 256 is divisible by 32, why is this important?

6.5 Call the kernel

Well fairly self explanatory.

6.6 In the kernel, calculate the index of the thread

6.7 In the kernel, do the addition

6.8 Copy device memory to host

6.9 Free device memory

Great, at this point you should be able to run and the verification should pass. If verification passes, the program will output some information, something like:

```
Computation speed-up 21.00 real speed-up 0.29   Host malloc: 2ms  Device malloc: 142ms
CPU Gflops 0.5   GPU Gflops 10.0
```

'Computation speed-up', considers only computation time, GPU verse CPU, where as 'real speed-up' includes the time to copy data to and from the device.

6.10 Bigger problems

By default the program runs with arrays of 10 000 000 elements, this can be changed with the `-n` flag. Try some bigger problem sizes say, 12 000 000, 15 000 000 how does this affect your speedups? Now if you try bigger problems you may run into various problems. For each of the sizes below, run yore program, if it fails note why, think of a way to fix it and implement it. Hopefully you implemented some error checking in yore previous sections it may come in very helpful here. You may need to refer to some of the results of the device query you wrote in the previous section to solve your problems.

- 17 000 000 What problems could this cause?
- 70 000 000 What problems could this cause?
- 470 000 000 What problems could this cause?

Once you work through the above, you should be able to run problem sizes of up to +-690M elements, what is the limiting factor if you go over this? Think about how you would go about over coming this, but you don't have to implement it.

6.11 Analyse

You may have noticed the `logger->csvWrite` commands in the code, you can write some output of your run to a comma separated file for easy analysis, this is enabled with the `-l` flag. We can now see how the problem scales, on the size of one GPU, as the problem size increases. For this we need to run tests at number of problem sizes. This can easily be achieved by making the block at TODO 6.11 loop over a variety of problem sizes that cover the full range possible by your program. Do this with the `-e` flag and use the cvs log, create a graph to show your results. What dominates the time taken to run the GPU version, how would you try get a round this type of problem?

Now, how about comparing apples with apples, so far you have been doing the CPU calculation using only 1 processor. How many processors do the various nodes have? If you uncomment the line that starts "`//#pragma omp parallel`" the CPU add will run in parallel across the CPU cores. (Note you will have to change your PBS script to request more CPU's). How does this change your results, what real speed up are you getting now? How about Gflops, what real Gflops are you achieving compare this to the theoretical maximum we calculated earlier?

The take home message from this is that you can get code to run in parallel but it is not always easy to get the performance you expect!