CAASTRO GPU Workshop 2012

Preliminary

It is assumed that you have the basic knowledge of programming in Linux environment. If you have difficulty in starting to program in Linux, please let us know.

Before you starting the exercises below, please head to the website http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation and download the CUDA_C_Programming_Guide, CUDA_C_Best_Practices_Guide (v4.2), CUDA_Toolkit_Reference_Manual (v4.2), Compute Command Line Profiler User Guide and CUDA_Occupancy_Calculator.xls We will also use CUFFT Library User Guide (v4.2) and CUBLAS Library User Guide (v4.2) for exercises of utilizing pre-written programs. These are some useful reference when doing CUDA programming and we will refer to these documentations fairly extensively.

The exercises in each part has a description and tasks section, it is quite obvious as what is meant by description part. While tasks are made up of a few types of exercises, they might be programming exercises, or execution of some programs and results observation, or simply some open questions that you can attempt to answer and later discussed together.

There are also some tasks or a whole part labelled as o*ptional*. These are the parts or tasks that you should attempt, but most of them will be fairly time consuming, and some of them depend on previous optional exercises, so do not spend too much time when you get stuck. You can come back to these exercises later if time permits or outside of the workshop sessions if you are interested. We will go through the summary or conclusion of these exercises nonetheless.

Part 1: Basics of CUDA programming

Part 1 covers the basics of CUDA programming including:

i. Compile and execute a CUDA program.
ii. Memory management in CUDA programs.
iii. Writing a kernel function.
iv. Timing the CUDA functions.

Unzip the zip file provided and you will find all the files that will be used throughout this workshop. Most of these files have .cu extension, which are source files for CUDA programs.

1.1: Compiling and executing a CUDA program

Description:

We will compile a very simple "Hello World" program with the file name *ex11.cu*. It can be seen that this program is just the same as a normal C Hello World program. In CUDA, not all the code are executed in GPU, there will be a big portion of the code executed in CPU. Code that is executed by CPU is named *host code* (and host function) while the code that is executed by GPU is named *kernel code* (or kernel function).

Tasks:

i. Compile the program with the command

```
nvcc -o ex11 ex11.cu
```

ii. Replace the last line with the program name in the job script and submit the job script

```
./ex11
```

or

If you are in an interactive PBS session, just run it with the command above in terminal.

1.2: Memory management

Description:

We will learn the basic memory management mechanisms in CUDA from a working example, file ex12_example.cu.

Details of the memory management functions can be found in page 50 of CUDA Toolkit Reference Manual. When just starting programming in CUDA, it is vital for anyone to have a working example of GPU memory management. You will need to refer to your previous working examples along the way. The same is applicable to part 1.3.

You may find the term “device” used throughout the CUDA C Programming Guide, and it can also be seen in the `cudaMemcpy()` function. You can already guess that it just means the GPU, and we will use the term “device” and “GPU” interchangeably in this workshop. Similarly, we will also use the term “host” and “CPU” interchangeably.

Tasks:

i. Open the file ex12_example.cu and study the memory functions used.
ii. Open the file ex12.cu side by side. In *vim*, this can be achieved by the either one of the following command:

```
:vs filename
:sp filename
```

or simply open up another terminal.
iii. Type the functions by hand (do not just copy and paste) from the file ex12_complete.cu to ex12.cu, try to change the variable name and get it to compile.
iv. Run the program and make sure that you are able to get the "Correct output".

1.3: Writing kernel functions

Description:

We will extend our previous hello world program to use the actual kernel functions. CUDA introduces a few keywords on top of C language. For instance the keyword `__global__` defines a function that is callable from host function, and executed in a GPU. These keywords are called Function Type Qualifiers, for details about these keywords, refer to chapter B.1 in CUDA C Programming Guide. When a kernel function is invoked, the thread and block numbers must be specified (refer to chapter 2.1 of the CUDA C Programming Guide). It defines how many copies the program are executed.

We will also introduce the concept of compute capability, defined by an *-arch* option in compilation. The compute capability defines the core architecture that you are using. For instance, the *printf* feature is only available since compute capability 2.0. Refer to chapter 2.5, Appendix A and F in the CUDA C Programming Guide for more details about compute capability. Practically we should always choose the latest compute capability that is supported, which in this case, it is 2.0 or sm_20.

Lastly, we will investigate the usage of 3 dimensional numbers in thread and block numbers.

Tasks:

i. In file ex13.cu, look for the kernel function which is labelled by the keyword `__global__`.
ii. Try to compile the program. Do you get an error?
iii. Try again, but with the following command instead:

```
nvcc -arch sm_20 -o ex13 ex13.cu
```

Note: For GPUs with lower compute capability, you will need to use certain setup. Example can be found from the *simplePrintf* module in the SDK source directory.

iv. Change with the number for threads and blocks, and observe how it affects the output.
v. Study the way that the kernel gets invoked in file ex13_3d.cu, then play with the thread and block numbers and observe how it affect the output.
vi. Add in some simple maths operation in the kernel and print out the results using *printf*, as an example:

```
int result = threadIdx.x + blockIdx.x;
```

vii. Observe how it obtains different results from each thread.
viii. You can also use `cuda-memcheck` command to have CUDA checking the memory access for you. In your job script or in interactive PBS session, use this command instead:

```
cuda-memcheck ./programname
```

1.4: Vector Addition Program

Description:

We will revise a bit on what we have learnt so far, by writing a simple CUDA program to perform integer vector addition. In file ex14.cu, you will find a C implementation of the vector addition. It uses a *for-loop* to run through all the integers in the vector, *arr1* and *arr2.* We will reinforce the concept of multi-threaded nature in GPU.

There is a verification function for checking your results automatically. Try not to cheat yourself by changing the verification function :).

Tasks:

i. Implement a GPU vector addition using kernel functions. With one requirement: you must use one thread to calculate only one integer in the result vector. The steps below can be used as a guide:
   a. Allocate memory for GPU variables using the pre-chosen variables *cudaArr1, cudaArr2* and *cudaResults*. Also use *arrSize* to define the size of your memory allocation.
   b. Copy the memory from both *arr1* and *arr2* into your newly allocated variables.
   c. Write a kernel function for doing the vector addition.
   d. Choose your number of threads and blocks. Make sure that
      *number of threads * number of blocks = arrSize*
   e. Copy the memory back from GPU to CPU, *resultsFromGPU.*

Tips: The key is to use the thread index and block index, and another preset variable, *blockDim* to identify the integer that need to be performed by a thread. There is an example in chapter 3.2.2 in CUDA C Programming that you can refer to.

ii. (Optional) Work out the correct indexing formula when either the thread or block size is 2D. Meaning that either *threadIdx.y* or *blockIdx.y* is not 1.
   a. Use the *dim3* function from the 3D version from exercise in part 1.3.
   b. Sometimes the GPU memory will not clear itself after the program finish executing. Therefore, it might show correct results when it should not, please insert a *memset* function after you allocate the GPU memory:

```
cudaMemset( cudaResults, 0, arrSize * sizeof(int) );
```

Tips: It might be helpful to draw out a simplified thread and block map on a paper. You should start by working out the correct formula when *threadIdx* is 2D, but *blockIdx* is 1D. Then proceed to work out the formula when both *threadIdx* and *blockIdx* are 2D.

1.5: Timing CUDA functions

Description:

Timing is a very important aspect in CUDA programming. It is because most programmers are interested in how much can be improved with CUDA implementation. Therefore we will go through some useful timing methods.

We will look at two different timing methods, one uses a very general Linux timing function, `gettimeofday()` while the other uses the CUDA event timing functions, which utilizes the GPU clock. For more information about the CPU timing function, use the command `man gettimeofday` under a terminal, while for GPU timing function, it is found in chapter 5.1.2 in CUDA C Best Practices Guide.

Tasks:

i. Open and file ex15.cu and study the two timing methods used in this program. You may try to compile and run this program.

Note: Most CUDA function calls (except normal `cudaMemcpy()` functions) are asynchronous. It means that when you invoke a function, it will return immediately with a success or failure return value, while the actual computations are still being executed in the background. Therefore you need to use the function `cudaDeviceSynchronize()` or `cudaEventSynchronize()` to make sure that your program waits for the actual computation to finish and provide you with the appropriate timing results.

ii. Use the any timing function and apply to the program you have completed in part 1.4. Compare the execution time of the CPU and your GPU kernel implementation.
iii. Observe the effect of the timing results without synchronization using the function `cudaDeviceSynchronize()` or `cudaEventSynchronize()`.
iv. What conclusion can you draw from the timing comparison between CPU and GPU implementation (is GPU faster or slower)?

1.6: Array with 2 or More Dimension

Description:

Often it might be more convenient to use 2 or more dimensional arrays. We will look 2-dimensional arrays usage in CUDA. In file ex16.cu, it is pre-written with a wrong way to allocate 2D array in CUDA. With the way error handling works in GPU, you might not see anything wrong when compiling, or even when executing it. But when you start doing something in a kernel, you might get incorrect results or some memory access errors.

The correct way is to not use double pointer. In CUDA, it is much more efficient to allocate a large contiguous memory than using pointers pointing to various different locations in memory. Hence the indexing of this 2D array is done by a simple calculation. Say you would like to access the item in `[x][y]` position, you will need to do something like `[x * ncols + y]` in order to obtain the correct index. There is an example in page 7, CUBLAS Library, at the line `#define IDX2C` which defines a macro for 2-dimensional indexing (we will come back to this same example later).

Tasks:

i. Study the memory allocation in file ex16.cu, observe the method of 2D array allocation in CPU, but 2D array in GPU cannot be created the same away.

ii. Comment out the wrong way and uncomment the correct way (which was comment out by the `#if 0` line). You may execute the program. Here are the steps that were taken in the process:

a) Allocate enough memory for *nrows * ncols* floating point number in *arr1D* (CPU memory) and *cudaArr* (GPU memory).

b) Transfer the contents *arr2D* to *arr1D* using `memcpy()`. A for-loop is used in this part.

c) Copy the memory from *arr1D* to *cudaArr.*

1.7: (Optional) Simple Matrix Multiplication

Description:

After allocating 2D array, we can write a simple matrix multiplication. We will employ one assumption though, all matrices used are square. This is to reduce the complexity of our program. But of course if you are feeling adventurous, feel free to use matrices with different height and width.

The file ex17.cu is pre-written with CPU implementation of matrix multiplication. For simplicity, the CPU implementation does not use a double pointer for array storage that eliminates the need to re-arrange the array. For a simple matrix multiplication, you should use one thread to calculate one item in the result matrix, and do not worry about the inefficiency of global memory.

Tasks:

i. By referring to the CPU implementation of matrix multiplication, implement your GPU version of matrix multiplication. The steps are similar to the implementation process of previous exercises where you need to perform GPU memory allocation and transfer.
ii. Measure the execution time for your GPU implementation and compare with the CPU implementation. Are you able to get close to the CPU implementation in terms of execution time?

Part 2: Useful Libraries in CUDA

There are many useful libraries in CUDA, in most cases you don't even need to know about the kernel programming or anything fancy to start utilizing your GPU. With these libraries under your sleeves, you can already make full use of your GPUs without the need to learn about some advance optimization methods. We will use two great examples, the CUDA Basic Linear Algebra Subprogram (CUBLAS) and CUDA Fast Fourier Transform (CUFFT). We will also cover atomic functions which strictly speaking is not really a library, but a group of useful functions.

2.1: Linear Algebra Libraries, CUBLAS

Description:

CUBLAS library provides functions for calculating linear algebra operations, including vector scaling, dot products, matrix-vector multiplication, matrix-matrix multiplication and so on. Have a look at the example code in page 6 of CUBLAS Library User Guide, example 1 uses the 1-based indexing, which is more compatible with Fortran. Example 2 uses 0-based indexing which is more suitable for C programming, therefore we will only look at example 2.

The example might seem a bit over complicated at first glance, but in fact it is very simple. Most of the code is for error handling purpose. The file ex21.cu implements a simplified version of the example using the vector scaling function.

Tasks:

i. Study the usage of CUBLAS function in file ex21.cu, try using the CUBLAS dot product functions in page 26-27 of CUBLAS Library User Guide for calculating dot product of *vec1* and *vec2*. The CUBLAS library needs to be linked in the compile command:

```
nvcc -arch sm_20 -lcublas -o ex21 ex21.cu
```

Note: The two variable *cudaVec1* and *cudaVec2* have been pre-allocated for your use in the dot product implementation. Be aware that these two variables are of type *double*, hence you need to choose the correct function for handling *doubles*.

ii. (Optional) Try out the CUBLAS matrix multiplication function.

Note: Functions for performing operation between two matrices are labelled as level 3 functions in CUBLAS, try to look for the matrix multiplication function in that section. You can either use the existing variables *cudaVec1* and *cudaVec2,* or define a new set of variables. Just be aware that you

need some extra variables for the width of both matrices (once you have defined a width, vector becomes matrix), and also the rows and columns that is required by the functions.

iii. (Optional) Measure the time for executing the CUBALS matrix multiplication, then compare to your previously written GPU simple matrix multiplication. Is there any difference between your implementation and CUBLAS library?

2.2: FFT functions in CUDA

Description:

FFT (Fast Fourier Transform) is a very popularly used method for signal processing. It is used to transform time series into frequency series or vice-versa. The CUDA FFT or CUFFT is modelled after the very popular FFTW (Fastest Fourier Transform in the West) implementation.

In the file *ex22.cu,* a C function wrapper for FFTW execution (in *exfft.c*) was used. This can be used as a reference for comparison between CPU and GPU FFT operations. We will investigate the time performance for these two implementations.

Note that FFTW came with multi-threaded implementations as well, but we will not look at that in this workshop. Performance in multi-threaded CPU implementation is much more predictable though, the execution speed usually scales linearly with the number of CPUs (or cores).

Tasks:

i. Compile and execute the program ex22.cu. You will need to load the *fftw/3.3.2* modules, and the file needs to be compiled with *-lfftw3f* (for FFTW) and *-lcufft* (for CUDA FFT).
ii. Observe the execution time for each implementation. Which implementation is faster?
iii. Try increase the size of the array used in the function and compare the timing, you may also try numbers that are not of power of 2. How does execution get affected? For instance, how does the execution time for both FFT implementations scale with data size? Do you get twice the execution time when you use arrays that are twice as large?
iv. Implement the *batch* version of CUFFT. Refer to work examples in chapter 5 of CUFFT Library, you can find usage of the *BATCH* variable, which tells the program how many FFTs you want to perform in parallel.

    Note: You will need to change the array size to
    *array size = data size * batch*

v. Enclose the FFTW function with a loop that will run *BATCH* times, and then compare the execution time with your batch CUFFT implementation. How does the performance scale now with data?
vi. You may also try to slightly optimize the CPU version, that make use of the *setup* and *destroy* variables, where the very first run of the loop should setup the plan, and the last loop should destroy the plan. Then compare the performance again.

Exercise 2.3: Atomic Function

Description:

Atomic function is used for performing some operations in certain memory space accessed many threads. It might not be the best option out there, but it is certainly very useful in some situations.

Tasks:

i. Examine the file ex23.cu. Then compile and execute the program, which will output some wrong answers. Can you work out what is the expected output?
ii. Comment out the line in the kernel function which is performing some addition operations, uncomment the line with `atomicAdd()` function.
iii. Try experiment a bit with different number of threads and blocks, with or without using the `atomicAdd()` function.
iv. Can you guess what the potential overhead with using atomic functions is? And why it happens?

Part 3: Simple Optimization Technique

In part 3, you will learn the following:

1. Usage of CUDA profiler.
2. Choice of number of threads, blocks and grids.
3. Accumulate more data for more optimum execution.
4. Optimal memory access pattern.
5. Memory access pattern for 2D array.
6. Utilize shared memory.

In the process of optimizing our CUDA programs, we almost always need to do this step, *identify bottlenecks*. It will be undesirable to spend a lot of time and effort to gain little speed-up. Therefore we will start off with the CUDA Profiler provided by NVIDIA.

3.1: CUDA Profiler

Description:

CUDA includes a very useful tool, which is really handy for us to identify bottlenecks. It is very easy to use, you just need to set a few environment variables, then CUDA will automatically do the profiling whenever you run a CUDA program with memory copying between CPU or GPU, or with a kernel launch. Setting the environment variable *COMPUTE_PROFILE* will enable or disable the profiling, *COMPUTE_PROFILE_CSV* will make the output comma separated version format, *COMPUTE_PROFILE_CONFIG* will enable you do use a config file for a few extra performance counters. For more details about these variables, refer to the Compute Command Line Profiler User Guide.

Tasks:

i. In your job script, add in a line before your program name to enable the profiler:
```
export COMPUTE_PROFILE=1
```
or execute the same command in an interactive PBS session:
```
export COMPUTE_PROFILE=1
```

ii. Run any previously written program that has memory copying and kernel launch.

iii. Open and study the log file generated, should be named *cuda_profiler_0.log* by default.

iv. Create a file for setting the profiler options, say *profiler.conf*. Insert a few lines such as:
```
threadblocksize
regperthread
```

v. Then set the environment variable `COMPUTE_PROFILE_CONFIG` to a chosen file name.

```
export COMPUTE_PROFILE_CONFIG=filename
```

vi. Rerun your program and observe the difference in your profiler log.

vii. Play with your programs as much as you like, like adding more variable numbers in kernel, adding more maths operation in each thread, changing your memory copying size and so on. Observe how these actions affect your memory copy time and number of registers per thread.

3.2: Choose Appropriate Numbers for Threads, Blocks and Grids

Description:

You may have asked this question before, "what is the best number of threads and blocks should be chosen?" You may have come across the term "grid" as well, this is basically the number of GPUs you have, that you will not be able to choose freely, and it is not managed through the same mechanism as threads and blocks.

But for number of threads (block dimension) and blocks (grid dimension), they should not be chosen randomly as well. There are some guidelines as to how you should choose these numbers. Do you remember executing the *deviceQuery* program from previous exercise? We will go through some of the crucial specifications that form the guidelines.

The file ex32.cu contains a simple kernel program that you can use to experiment with the effect appropriate or inappropriate numbers for threads. Note that the numbers in this program is designed to magnify the effect of the choice of number of threads. And it is pre-written with a simple mechanism to accept command line argument for changing the number of threads without recompiling the program.

Tasks:

i. Open the output file from the execution of *deviceQuery* program, or re-run the program.

   Note: It can be seen that there are a few properties regarding the number of threads and dimension of block and grid. The meaning of these numbers are obvious, you cannot have more than 1024 threads per block, or more than 65535 in each block dimension.

ii. Compile and execute the program ex32.cu. Try running vary the number of threads by just passing an argument (best run in an interactive PBS session), for example:

```
./ex32 255
```

iii. Try the number 255, and also 257. Did you notice the vast difference from 256 to 257 and no difference from 255 to 256? Experiment with as many numbers as you like. Are you able to spot a pattern?

   Tips: *Warp size* from *deviceQuery*.

   Note: The rules for number of blocks are much looser, most of the time block numbers will adapt to the choice of the number of threads. But, generally you should at least have about twice as many as the number of multiprocessors in a GPU.

3.3: Accumulate More Data

Description:

In part 2.2, we have actually already implemented the concept of accumulation more data. In the batched version of CUFFT, multiple FFT operations will be performed in parallel, giving us huge performance gain. But at the same time, it also means that more data will be needed.

In this part, we will look at vector addition operations again. File ex33.cu is pre-written with a simple implementation of CUDA, which should be similar to what you have done in part 1.4, with some slight variations to incorporate multiple sets of data, plus some timing functions.

Tasks:

i. First compile and execute the program without any argument, then try inputting *batch* argument. You may start from 2, then 4, 8, 16 and also larger numbers like 1024, 2048. Do you see any advantage from the batch implementation (compare to your previous CUDA implementation)?

   Tips: Do not always expect improvement from optimization technique.

ii. Now if you have not enabled the CUDA profiler, enable it as per instructed in part 3.1, then re-run your program. Can you find out what is taking most of the time?

iii. Is there any way to speed-up the bottleneck? You may try to use a similar mechanism from part 1.6, also you may try using page-locked or pinned memory (by replacing `malloc()` functions with `cudaMallocHost()`, refer to page 63 of CUDA Toolkit Reference Manual). Another option is to use Texture Memory which is not covered in this workshop.

iv. The final question would be, is it worth using GPU for vector addition?

3.4: Aligned Memory Access

Description:

You may or may not notice that we always use array or data size with certain pattern, which are numbers such as 256, 1024 and other powers of two. One of the reason was already discussed earlier, which was due to the choice of number of threads and blocks (multiple of warp size). The other reason (which is not as important as previous though) is the memory access pattern of CUDA.

There are detailed explanations in chapter 6.2.1 in CUDA C Best Practices Guide, named *Coalesced Access to Global Memory*. It might be a bit complicated at first glance, but what it is saying is basically this: whenever your threads in a warp access a certain space in *global memory*, CUDA will always fetch the whole 128-byte segment into the cache. We will just do a little bit of experiment with the simple example provided under this chapter.

Tasks:

i. Compile and execute the file ex34.cu. Then try experimenting with some arguments used for changing the variable *offset*, using numbers such as 1, 2, ..., 32, 33, ..., 64, but not exceeding the preset macro *OFFSETLIMIT*.
ii. Observe how it changes your kernel execution time (by using CUDA profiler or inserting your own timing functions), we will ignore the memory copy in this part.
iii. You may have a read on chapter 6.2.1 in CUDA C Best Practices Guide, are you able to figure out why we choose certain memory size in our programs?
iv. Try running ex34_1.cu as well. This is the experiment for strided access pattern. You can vary the stride (without exceeding *STRIDELIMIT*) and observe how much wastage it cost.

3.5: Aligned Memory Access for 2D Array

Description:

When we use 1D data, we do not need to worry about the memory being not aligned to the exact segments as discussed in previous part. But when we start using 2D arrays, which was allocated in a big array block, it may not be very efficient in terms of access time.

CUDA provides a function called `cudaMallocPitch()` for this exact purpose. It will ensure the memory alignment requirement for each row of your 2D arrays. In this part, we will only try out the functions, and observe how the *pitch* value changes.

Tasks:

i. First have a read on the `cudaMallocPitch()` function in page 63 of the CUDA Toolkit Reference Manual.
ii. The file from part 1.6 was re-used here, but modified to use `cudaMallocPitch()`. Compile and run the program ex35.cu, the variable *ncols* can be changed using command line argument. Try changing it and observe the change in the pitch value.
iii. Using a piece of paper, or any drawing program on your computer, identify the index of each item in your matrix in terms of *row (or i)*, *col (or j)* and *pitch*.

    Note: The *index* here is what you will use when addressing your matrix as in `matrix[index]`.
iv. Proceed to replace the row and columns with threads and blocks indices.
v. Replace the index *i* in the simple kernel, to see if you got the correct index. Remember to pass in the *pitch* variable in the function parameter.

3.6 Shared Memory

Description:

Shared memory is different from the memory allocated from `cudaMalloc()`. It is resides in a block, only shareable between threads in the same block. It has very limited size that can be checked from the *deviceQuery* program.

The file ex36.cu contains a program for demonstrating the usage of shared memory. What it does is just getting each thread in the same block reading an input and does some floating point operations on it.

We will also look at another example that calculates simple moving average. Moving average is used very often in financial markets. We will take the simplest version of it for our shared memory example. The equation is:

$$MA_k = \frac{x_k + x_{k+1} + x_{k+2} + x_{k+3} + x_{k+4} + x_{k+5} + x_{k+6} + x_{k+7}}{8}$$

Detailed explanation of how to declare shared memory can be found in Appendix B.2.3 CUDA C Programming Guide.

Tasks:

i. Compile and execute the program ex36.cu. Then examine the kernels execution time in the CUDA profile. Observe the difference in the two kernels that are doing basically the same operations.

ii. From this program, can you think of a case where utilizing shared memory is not useful?

iii. Open the file ex36_ma.cu. You will find a simple program that calculates 8-day moving average from a set of input. Without changing the number of blocks and threads, implement your own kernel function utilizing shared memory, similar to previous task. And check the execution time from the CUDA profile.

Tips: They key is issue you need to figure out is how you should load the shared memory. You may use this line to print out previous error from CUDA when the kernel execution did not get profiled:

```
printf( "%s\n", cudaGetErrorString(cudaGetLastError()) );
```

And there is an error checking method, you should be able to get 0 difference between the two method.

iv. How much speed-up are you able to obtain in this moving average example? Try increasing the number of *ndays* through command line argument, can you achieve higher speed-up?

4: Other Optimization Technique and Examples

In part 4, we will take a look at several other optimization techniques, they are:

1. Overlap kernel execution and memory copy.
2. Avoid bank conflicts in using shared memory.
3. Avoid branches within a warp, or warp divergence.

Note that the list optimization techniques in this workshop are not exhaustive. There are new techniques (or new implementation of old techniques) or optimization for newer architecture published from time to time. It is beneficial if you are able to stay up-to-date with programs version where the libraries (such as CUFFT) might get further optimized, and be aware of new publications.

4.1: Overlap Kernel Execution and Memory Copy

Description:

We have already looked at the concept and effect of asynchronous executions of CUDA kernels in part 1.5. This feature can of course be used to our advantage when writing CUDA programs. Without much effort, we can already execute a CUDA kernel and do some CPU operations in parallel. We will not look at that in this part of exercise though.

For CUDA uses *streams* to manage the multiple operations of memory copy and kernel executions. A CUDA stream defines a sequence of operations that get executed in order, while different streams are not guaranteed to execute in order. There are some good explanations in Chapter 3.2.5.5.5 in CUDA C Programming Guide, where we will use its examples in this exercise.

Tasks:

i. Inspect the code in ex41.cu, it contains examples of executing two kernels in sequential order, and also utilizing *stream* for overlapping kernel and memory copy plus concurrent kernel. They are found in the kernel launching code in `runtest()` function.

   Note: You may also check the CUDA profile time which will probably not show the speed improvement from overlapping memory copy and kernel execution or concurrent kernel.

ii. Compile and execute the program, observe the difference in execution time between the example methods.

iii. Vary the number of stream with command line argument and re-run the program. You may also play with different number of data size and also threads and blocks.

4.2: Avoid Bank Conflict

Description:

When using shared memory, there is one issue that you might want to watch out for. It is something called bank conflict. From the CUDA C Programming Guide (chapter 5.3.2.3), bank conflict occurs when two addresses of a memory request (within the same warp) fall in the same memory bank, where banks are the memory modules that can be accessed simultaneously.

We will take a look at a very common case that might cause bank conflicts in shared memory: complex number of *float* type. There are a few examples given in Appendix F.3.3 of CUDA C Programming Guide. Complex number would fall under the category of *larger than 32-bit access.* The GPUs used in this workshop have 32 banks, instead of 16 banks (explained in CUDA C Best Practices Guide). Therefore, the warps are not split into two, and bank conflicts can occur between the first half and the second half of the warp.

Tasks:

i. Inspect the file ex42.cu. Look for the line where shared memory was read. Can you draw out on a paper (or a computer drawing board) that how did a bank conflict occur?
ii. How would you solve the bank conflict problems with *float* type complex number?
iii. Now inspect the file ex42_1.cu, and inspect the different structure used for storing the complex number. You may execute both programs to see the difference in the kernel executions. Is this the idea you got for avoiding bank conflict?
iv. In this way, bank conflict problem is avoided, but is it worth changing the whole program to use this kind of structure?
v. Try play with the program a bit, such as increasing the iterations in the kernel to increase the frequency of shared memory reading. When do you think it is worth modifying the complex number structure just for avoiding bank conflicts in this example?

4.3: Avoid Branches within Warp

Description:

As explained in previous exercises, threads are being executed in warps. In other words, a multiprocessor in CUDA will (randomly) choose a warp to execute. All the threads in this warp will execute the SAME command at any given time. Therefore, when you have different branches, or *if* cases in threads of the same warp, the whole warp will have to execute all the different branches together, this is also called warp divergence.

Note that this does not mean that you cannot have any branching operation or *if* case in a kernel, the results will correctly calculated. But it will waste some operations if the threads within the same warp do not enter the same branch. For instance, you should avoid having an *if* case like this:

```
if( threadIdx == 0 ) /* Do some operations */

else if( threadIdx == 1 ) /* Do some other operations */
```

This kind of *if* cases are very common in CPU programs, but should be avoided as much as possible in kernel functions, unless you are very certain that all threads in the same warp will take the same branch route. You can also replace some possible *if* case with maths operations. We will look at one example of this type, and also an artificially chosen (sort of) extreme case.

In file ex43.cu, you can see that there is a classic example of setting the maximum of a set of numbers, might be very useful in colours rendering as you usually want to limit the magnitude to only 255. Note that the numbers used in here does not really give a high amount of warp divergences, but you can do modify the code (add more if-cases) to increase the amount of warp divergences.

Tasks:

i. Inspect the code in ex43.cu. Locate the code where there are warp divergence. Draw out a simple map (of just a few threads) of what operations are the threads performing.
ii. You may compile and execute the program, and take note of the kernel execution time from the CUDA profile. Again we will only look at the kernel executions time, and not the memory transfer time.
iii. You may vary the either kernel (add more if-cases) to either increase or decrease warp divergences. Based on your observation, is it an effective technique for the first kernel in the program? Do you think it is worth eliminating the warp divergences using this method?
iv. We have only looked at some simple examples of maths operations. Now take a look at the code below, can you figure out what is the appropriate maths operations for replacing the "if"?

```
// Consider x with values ranging from 0.0 to 1.0
if( x < 0.5 ) y = 2 * x;
else y = x;
```

Tips: The `floor()` or `floorf()` functions may be of use here.

This concludes the exercises in this workshop. Hope you find it to be useful. Good luck.