GPU Programming Assignment (HPC Course)

Neeraj Sathyan (12938262)

February 2020

1 Exercise 1

1.1 a.

Identified the kernel, and added the necessary operation to implement vector addition. The following code is added into the kernel:

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
Result[i] = A[i] + B[i];
```

1.2 b.

Run the code for at 5 different sizes of the array: 256, 1024, 65536, 655360, and 1000000. Report the execution times for each timer. Comment on the performance difference between these values.

1.2.1 Report for array size: 256

Threads per blocks: 256

```
vector-add (sequential):          = 9.33911e-07 seconds
vector-add (kernel):              = 0.000222871 seconds
vector-add (memory):              = 8.50346e-05 seconds
```

1.2.2 Report for array size: 1024

Threads per block: 512

```
vector-add (sequential):          = 8.79238e-07 seconds
vector-add (kernel):              = 0.000209564 seconds
vector-add (memory):              = 8.74672e-05 seconds
```

1.2.3 Report for array size: 65536

Threads per block: 512

```
vector-add (sequential):       = 0.000314457 seconds
vector-add (kernel):           = 0.000238089 seconds
vector-add (memory):           = 0.000825005 seconds
```

1.2.4 Report for array size: 655360

Threads per block: 512

```
vector-add (sequential):          = 0.00209592 seconds
vector-add (kernel):              = 0.000245389 seconds
vector-add (memory):              = 0.00501311 seconds
```

1.2.5 Report for array size: 1000000

Threads per block: 1024 No. of Block: 977 There will be (1024 x 977) - 1000000 threads doing no work here. i.e 448 threads

```
vector-add (sequential):          = 0.00296042 seconds
vector-add (kernel):              = 0.000336525 seconds
vector-add (memory):              = 0.00730678 seconds
```

Based on the different performances of the kernel for different array sizes, it can be inferred that kernel performance is proportional to the array size, although at small array values, the sequential outperforms the kernel as for small array sizes, there is less overhead on CPU for processing and IO, whereas for kernel even though there is performance in computation, the overhead of copy the data back to register makes it a roadblock. We can also infer that for any configuration the kernel overall time (memory + kernel) is far greater than the sequential execution, since the array copying to the GPU is memory intrinsic and limits the performance.

1.3 c.

Run the code for 5 different grid geometries, and report the results. What is the best performing configuration? Any idea why? Taking Array Size as 655360

1.3.1 Grid: 1280 x 512

```
vector-add (sequential):          = 0.00209592 seconds
vector-add (kernel):              = 0.000245389 seconds
vector-add (memory):              = 0.00501311 seconds
```

1.3.2 Grid: 640 x 1024

```
vector-add (sequential):          = 0.00271212 seconds
vector-add (kernel):              = 0.000321288 seconds
vector-add (memory):              = 0.00651095 seconds
```

1.3.3 Grid: 2560 x 256

```
vector-add (sequential):          =  0.003371 seconds
vector-add (kernel):              = 0.000402984 seconds
vector-add (memory):              = 0.00800138 seconds
```

1.3.4 Grid: 5120 x 128

```
vector-add (sequential):                    = 0.00224991 seconds
vector-add (kernel):                = 0.000297877 seconds
vector-add (memory):                = 0.00570816 seconds
```

1.3.5 Grid: 20480 x 32

```
vector-add (sequential):                    = 0.00188922 seconds
vector-add (kernel):                = 0.000250202 seconds
vector-add (memory):                = 0.00452138 seconds
```

From the above the results, it can be concluded as threads per block between 128-512 gives best performance, and low and very high threads per blocks inhibits the performance. The performance of grid geometry depends on SM occupancy. Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The maximums are hardware maximums limit. It's not possible to create a HW design that supports an infinite number of open blocks per SM. A very small block sizes (e.g. 32 threads per block) may limit performance due to less occupancy. Very large block sizes for example 1024 threads per block, may also limit performance, if there are resource limits (e.g. registers per thread usage, or shared memory usage) which prevent some threadblocks from being resident on an SM. Threadblock size choices in the range of 128 - 512 are less likely to run into the aforementioned issues. Usually there are not huge differences in performance for a code between, say, a choice of 128 threads per block and a choice of 256 threads per block.

1.4 d.

Change the operation from addition to subtraction, multiplication, and division, respectively. Compare the performance of these operations for the GPU kernel. Comment on the results.

1.4.1 Addition (Threads per bloc: 512)

```
vector-add (sequential):                    = 0.00209592 seconds
vector-add (kernel):                = 0.000245389 seconds
vector-add (memory):                = 0.00501311 seconds
```

1.4.2 Subtraction (Threads per bloc: 512)

```
vector-add (sequential):                    = 0.00387817 seconds
vector-add (kernel):                = 0.000452247 seconds
vector-add (memory):                = 0.00910407 seconds
GPU Total time (kernel + memory):                   = 0.00536685 seconds
```

1.4.3 Multiplication (Threads per bloc: 512)

```
vector-add (sequential):                    = 0.00246406 seconds
vector-add (kernel):                = 0.00032032 seconds
vector-add (memory):                = 0.00588623 seconds
GPU Total time (kernel + memory):                   = 0.00350761 seconds
```

1.4.4 Division (Threads per bloc: 512)

```
vector-add (sequential):          = 0.00253355 seconds
vector-add (kernel):              = 0.000284774 seconds
vector-add (memory):              = 0.00559786 seconds
GPU Total time (kernel + memory):                  = 0.00331905 seconds
```

The data is inconsistent to show any inference.

1.5 e.

Take a look at the vector-add-events.cu. Notice the difference in measuring performance by comparing the code with the code in vector-add.cu. Comment on the (potential) difference between the actual time measurements for the two versions, for vector-add operations, using the 3 largest array sizes.

1.5.1 For array size of 655360 elements:

```
vector-add.cu :
vector-add (sequential):          = 0.00209592 seconds
vector-add (kernel):              = 0.000245389 seconds
vector-add (memory):              = 0.00501311 seconds

vector-add-events.cu :
kernel invocation took          0.188192 milliseconds
vector-add (CUDA, total):                 =  0.269845 seconds
```

1.5.2 For array size of 65536

```
vector-add.cu :
vector-add (sequential):                  = 0.000179734 seconds
vector-add (kernel):              = 0.000159233 seconds
vector-add (memory):              = 0.000583113 seconds
GPU Total time (kernel + memory):                  = 0.000448573 seconds

vector-add-events.cu :
kernel invocation took          0.126176 milliseconds
vector-add (CUDA, total):                 =  0.260998 seconds
```

1.5.3 For array size of 1000000

```
vector-add.cu:
vector-add (sequential):                  = 0.00339652 seconds
vector-add (kernel):              = 0.000353746 seconds
vector-add (memory):              = 0.00778558 seconds
GPU Total time (kernel + memory):                  = 0.00458828 seconds

vector-add-events.cu:
kernel invocation took    0.213984 milliseconds
vector-add (CUDA, total):                 =  0.208557 seconds
```

From the above, it can be inferred that vector-add-events kernel invocation is faster when using cuda events instead of CPU based timers that works over synchronization. A problem with using host-device synchronization points, such as cudaDeviceSynchronize(), is that they stall the GPU pipeline. For this reason,

CUDA offers a relatively light-weight alternative to CPU timers via the CUDA event API.

1.6 f.

Take a look at the vector-add-streams.cu implementation. Explain what the code does differently, where the performance improvement should come from (if at all), and check whether the performance is actually improved for the streams version. Could you imagine (and implement) a different/better way of using streams for this code? If so, did this new version improve performance?

```
Streams: (655360 x 512)
Adding two vectors of 655360 integer elements.
vector-add (sequential):                    = 0.00250825 seconds
vector-add (kernel):              = 0.000777531 seconds
vector-add (memory):              = 0.00602117 seconds
vector-add (total) :              = 0.0068037 seconds
Overall sequential:               = 0.00256906 seconds
Overall CUDA:             =   0.22451 seconds
results OK!

Normal: (655360 x 512)
Adding two vectors of 655360 integer elements.
vector-add (sequential):                    = 0.00255225 seconds
vector-add (kernel):              = 0.000303381 seconds
vector-add (memory):              = 0.00607753 seconds
GPU Total time (kernel + memory):                   = 0.00355644 seconds
results OK!
```

Normal outperforms Streams. Fix: Comment cudaDeviceSynchronize as it blocks all the executed kernel streams till the last kernel stream is executed, and then continues with the device to host mem-copy. This takes up time which can be avoided using pipelining computation and communication.

Performance after fixing the issue:

```
Adding two vectors of 655360 integer elements.
vector-add (sequential):                    = 0.00198011 seconds
vector-add (kernel):              = 0.000230127 seconds
vector-add (memory):              = 0.00518092 seconds
vector-add (total) :              = 0.00541717 seconds
Overall sequential:               = 0.00202566 seconds
Overall CUDA:             =  0.138584 seconds
results OK!
```

As expected, the performance got improved compared to the trivial method used above.

1.7 Exercise 2.

We now work on the vector-transform folder. Please write the kernel that implements the vector transformation from the sequential version (as seen in function vectorTransformSeq. Run the code for at 5 different sizes of the array: 256, 1024, 65536, 655360, and 1000000. What do you observe, performance-wise? Compare against the performance of vector-add. What do you observe?

Solution:

1.7.1 Array Size of 256:

```
vector-transform (sequential):          = 2.08954e-06 seconds
vector-transform (kernel):              = 0.000278057 seconds
vector-transform (memory):              = 0.000107623 seconds
```

1.7.2 Array Size of 1024:

```
vector-transform (sequential):          = 5.34058e-06 seconds
vector-transform (kernel):              = 0.000233055 seconds
vector-transform (memory):              = 0.000101938 seconds
```

1.7.3 Array Size of 65536:

```
vector-transform (sequential):          = 0.000337775 seconds
vector-transform (kernel):              = 0.00026094 seconds
vector-transform (memory):              = 0.000738987 seconds
```

1.7.4 Array Size of 655360:

```
vector-transform (sequential):          = 0.00286919 seconds
vector-transform (kernel):              = 0.00055613 seconds
vector-transform (memory):              = 0.00449371 seconds
```

1.7.5 Array Size of 1000000:

```
vector-transform (sequential):          = 0.00453909 seconds
vector-transform (kernel):              = 0.000751864 seconds
vector-transform (memory):              = 0.00644128 seconds
```

As the size increases, the performance decreases. vector-add is faster than vector-transform as here it makes use of 5 kernels in synchronized manner to take care of Result[i] dependency.

1.8 Exercise 3.

Please implement a correct encryption and decryption and test it on at least 5 different files. Make a correlation between the size of the files and the performance of the application for both the sequential and the GPU versions. Report speed-up. Comment on the performance you observe.

1.8.1 File Size: 8K

```
The entire file content is in memory.
Encrypting a file of 4986 characters.
Encryption (sequential):                0.000001 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000186 seconds.
Encrypt (memory):               0.000080 seconds.
The entire file content was written to file.
The entire file content is in memory.
```

```
Decrypting a file of 4986characters
Decryption (sequential):                0.000001 seconds.
The entire file content was written to file.
Decrypt (kernel):               0.000029 seconds.
Decrypt (memory):               0.000043 seconds.
The entire file content was written to file.
```

1.8.2 File Size: 24K

```
The entire file content is in memory.
Encrypting a file of 21163 characters.
Encryption (sequential):                0.000018 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000167 seconds.
Encrypt (memory):               0.000089 seconds.
The entire file content was written to file.
The entire file content is in memory.
Decrypting a file of 21163characters
Decryption (sequential):                0.000002 seconds.
The entire file content was written to file.
Decrypt (kernel):               0.000028 seconds.
Decrypt (memory):               0.000055 seconds.
The entire file content was written to file.
```

1.8.3 File Size: 60K

```
The entire file content is in memory.
Encrypting a file of 57435 characters.
Encryption (sequential):                0.000043 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000158 seconds.
Encrypt (memory):               0.000120 seconds.
The entire file content was written to file.
The entire file content is in memory.
Decrypting a file of 57435characters
Decryption (sequential):                0.000006 seconds.
The entire file content was written to file.
Decrypt (kernel):               0.000026 seconds.
Decrypt (memory):               0.000079 seconds.
The entire file content was written to file.
```

1.8.4 File Size: 148K

```
The entire file content is in memory.
Encrypting a file of 148402 characters.
Encryption (sequential):                0.000080 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000159 seconds.
Encrypt (memory):               0.000204 seconds.
The entire file content was written to file.
The entire file content is in memory.
```

```
Decrypting a file of 148402characters
Decryption (sequential):                0.000013 seconds.
The entire file content was written to file.
Decrypt (kernel):               0.000030 seconds.
Decrypt (memory):               0.000136 seconds.
The entire file content was written to file.
```

1.8.5 File Size: 348K

```
The entire file content is in memory.
Encrypting a file of 352439 characters.
Encryption (sequential):                0.000215 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000207 seconds.
Encrypt (memory):               0.000449 seconds.
The entire file content was written to file.
The entire file content is in memory.
Decrypting a file of 352439characters
Decryption (sequential):                0.000042 seconds.
The entire file content was written to file.
Decrypt (kernel):               0.000059 seconds.
Decrypt (memory):               0.000331 seconds.
The entire file content was written to file.
```

Its inferred from here that as file size grows, naturally the computation time increases for both encryption and decryption in the sequential code. This is because of the size complexities as array size increases. However its observed that in the kernel part, the time almost remains same as kernel encryption is independent of the array size and is processed in warps instead of sequentially. For smaller array sizes, sequential program outperforms the kernel encryption and decryption. This is not the case as the file size increases, although there is not much significant speed up. This can be explained with the difference in hardware properties in GPU and CPU. CPU will have higher clock rates compared to that of GPUs, hence for small array sizes, the faster cores of CPUs are able to outperform the slower but large grid cores of GPUs just by sequential execution. But as the size increases, the CPU have to buffer the array characters outside of its buffer in queue during the sequential execution. This reduces the performance. In GPUs, since each character is encoded/decoded by a single GPU core, there is no buffering for execution. Only file sizes higher then the buffer of the GPU SM are scheduled in buffer queue.

1.9 Exercise 4.

A very interesting extension of this encryption algorithm is to use a larger key - i.e., a set of values, applied to consecutive values. Explain how you parallelize the new version, and implement this encryption/decryption algorithm as an extension to the original version. You can assume the key is already known (fixed), but its length (up to 256 characters) is given by the user (thus, should be configurable). Test this extended version for the same few files as in the previous case, and compare again the results against the sequential version. Report speed-up per file. Comment on the performance you observe.

Solution: The parallelization is similar with the previous code. Extra parameters of key array, key array size are passed on to the kernel memory. The indexing logic for accessing respective array character based on the key index is:

```
__global__ void encryptKernel(char* deviceDataIn, char* deviceDataOut,
                              int n, char *key, int keySize) {
    unsigned index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n)
    deviceDataOut[index] = deviceDataIn[index] + key[index % keySize];
}

__global__ void decryptKernel(char* deviceDataIn, char* deviceDataOut,
                              int n, char *key, int keySize) {
    unsigned index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n)
    deviceDataOut[index] = deviceDataIn[index] - key[index % keySize];
}
```

Comparisons were made between the previous case and the extended case:

```
normal caeser cypher:
The entire file content is in memory.
Encrypting a file of 352439 characters.
Encryption (sequential):                0.000215 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000207 seconds.
Encrypt (memory):               0.000449 seconds.
The entire file content was written to file.
The entire file content is in memory.
Decrypting a file of 352439characters
Decryption (sequential):                0.000042 seconds.
The entire file content was written to file.
Blocks: 689
Decrypt (kernel):               0.000059 seconds.
Decrypt (memory):               0.000331 seconds.
The entire file content was written to file.

specialised caeser cypher:
The entire file content is in memory.
Encrypting a file of 352439 characters.
Encryption (sequential):                0.001616 seconds.
The entire file content was written to file.
Encrypt (kernel):               0.000189 seconds.
Encrypt (memory):               0.000472 seconds.
The entire file content was written to file.
The entire file content is in memory.
Decrypting a file of 352439characters
Decryption (sequential):                0.001423 seconds.
The entire file content was written to file.
Blocks: 689
```

```
Decrypt (kernel):                0.000049 seconds.
Decrypt (memory):                0.000364 seconds.
The entire file content was written to file.
```

The normal code gave (encryption+decryption+memory) as: 0.001046 seconds Spcialised caeser cypher code gave (encryption+decryption) as: 0.001074 seconds The specialised version is slower just by a fraction (because of high memory transfers) than the normal code.

Its noted that kernel invocation tends to outperform sequential as file size increases. Throughput wins over the faster CPUs when files are larger. Speed Up compared to sequential programs: File Size 8K: 5% File Size 24K: 66% File Size 60K: 32% File Size 76K: 77% File Size 348K: 35%