Assignment IV: Advanced CUDA

Start Assignment

Due 9 Jan 2023 by 23:59 **Points** 4 **Submitting** a file upload **File types** pdf
Available after 1 Dec at 0:00

To submit your assignment, prepare and upload a PDF file fulfilling the following requirements:

- Named as *DD2360HT22_HW4_Surname_Name.pdf*
- Include your answers to each exercise
- Add to your previous Git repository *DD2360HT22*. Include the link in your report.
- Use the following folder structure in this Git repo : hw_4/ex_*ExerciseNumber* /*your_source_code_files*

Exercise 1 - Thread Scheduling and Execution Efficiency

In the lectures, we covered that when a CUDA kernel is launched, two levels of parallelism are generated. The top level generates a grid consisting of a one-, two-, or three-dimensional array of blocks. At the bottom level, each block further consists of a one-, two-, or three-dimensional array of threads. When scheduling these threads for parallel execution, each block is further partitioned into *Warps*. Warps are executed in SIMD hardware, i.e., all threads in one warp execute the same instruction but on different data. On current CUDA devices, each warp consists of 32 threads. *Divergence* occurs in the execution path when different threads in a warp take different paths. For instance, in a if-else construct, if some threads take the if-path and the other threads take the else-path, the SIMD hardware would require two passes to cover all paths, which results in higher cost and lower performance. For more details about divergence, refer to '**5.3 WARPS AND SIMD HARDWARE' in the reading material** [Reading: GPU thread scheduling and efficiency (https://canvas.kth.se/courses/36161/pages/reading-gpu-thread-scheduling-and-efficiency)](https://canvas.kth.se/courses/36161/pages/reading-gpu-thread-scheduling-and-efficiency) **.**

In this exercise, you will use the following simple kernel that processes an input picture by doubling the value of each pixel and storing into an output picture.

1. The following kernel process a picture of X pixels in the horizontal direction and Y pixels in the vertical direction. Each thread is in charge of processing for one pixel of the output picture.

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
```

```
   // Calculate the row # of the d_Pin and d_Pout element to process
 int Row = blockIdx.y*blockDim.y + threadIdx.y;
 // Calculate the column # of the d_Pin and d_Pout element to process
 int Col = blockIdx.x*blockDim.x + threadIdx.x;
 // each thread computes one element of d_Pout if in range
 if ((Row < m) && (Col < n)) {
   d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
 }
}
```

Questions to answer in the report

1. Assume X=800 and Y=600. Assume that we decided to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. How many warps will be generated during the execution of the kernel? How many warps will have control divergence? Please explain your answers.

2. Now assume X=600 and Y=800 instead, how many warps will have control divergence? Please explain your answers.

3. Now assume X=600 and Y=799, how many warps will have control divergence? Please explain your answers.

Exercise 2 - CUDA Streams

In this exercise, we will reuse the simple vector addition program that you have written for Lab3-ex1. To ease your understand, we show the relevant code blocks in the following: (1) allocate memory (2) data copy from CPU and GPU (3) launch the GPU kernel (4) data copy from GPU to CPU.

```
__global__ void vecAdd(DataType *in1, DataType *in2, DataType *out, int len) {
  //@@ Insert code to implement vector addition here
}

int main(int argc, char **argv) {
 …
  //@@ Insert code below to allocate Host memory for input and output
 …
  //@@ Insert code to below to Copy memory to the GPU here
 …
  //@@ Launch the GPU Kernel here
 …
  //@@ Copy the GPU memory back to the CPU here
 …
}
```

In this exercise, you will need to use multiple CUDA Streams in your code to improve its

parallelism. You may want to consider using **asynchronous memory copy** if you previously used synchronous copy. Please finish the following implementation based on your code for lab3-ex1:

1. Divide an input vector into multiple segments of a given size (S_seg)

2. Create 4 CUDA streams to copy asynchronously from host to GPU memory, perform vector addition on GPU, and copy back the results from GPU memory to host memory

3. Add timers to compare the performance using different segment size by varying the value of S_seg.

Questions to answer in the report

1. Compared to the non-streamed vector addition, what performance gain do you get? Present in a plot (you may include comparison at different vector length)
2. Use nvprof to collect traces and the NVIDIA Visual Profiler (nvvp) to visualize the overlap of communication and computation. To use nvvp, you can check **Tutorial: NVVP - Visualize nvprof Traces (https://canvas.kth.se/courses/36161/pages/tutorial-nvvp-visualize-nvprof-traces)**
3. What is the impact of segment size on performance? Present in a plot (you may choose a large vector and compare 4-8 different segment sizes)

Exercise 3 - Pinned Memory and Unified Memory

In this exercise, we will reuse the simple matrix-multiplication program that you have written for Lab3-ex2. To ease your understand, we show the relevant code blocks in the following: (1) allocate memory (2) data copy from CPU and GPU (3) launch the GPU kernel (4) data copy from GPU to CPU.

```
__global__ void gemm(DataType *in1, DataType *in2, DataType *out, int len) {

  //@@ Insert code to implement vector addition here

}
```

```
int main(int argc, char **argv) {
  …
  //@@ Insert code below to allocate Host memory for input and output
  …
  //@@ Insert code to below to Copy memory to the GPU here
  …
  //@@ Launch the GPU Kernel here
  …
  //@@ Copy the GPU memory back to the CPU here
  …
}
```

Please finish the following implementation based on your code for lab3-ex2:

1. Use **nvprof** to study the time spent on memory allocation, data movement and actual computation. You may want to test with several different matrix sizes.
2. Change the appropriate memory allocator to use cudaMallocHost().
3. For the same input that you have tested in Step 1, now use nvprof again to study the time spent on memory allocation, data movement and actual computation.

Next: make necessary changes to the program so it uses managed memory (Unified Memory UM):

1. Change the GPU memory allocators to use cudaMallocManaged().
2. Remove explicit data copy and device pointers that were used in the original code
3. For the same input that you have tested in the previous step, now use **nvprof** again to study the time spent on memory allocation, data movement and actual computation.

Questions to answer in the report

1. What are the differences between pageable memory and pinned memory, what are the tradeoffs?
2. Compare the profiling results between your original code and the new version using pinned memory. Do you see any difference in terms of the breakdown of execution time after changing to pinned memory?
3. What is a managed memory? What are the implications of using managed memory?
4. Compare the profiling results between your original code and the new version using managed memory. What do you observe in the profiling results?

Exercise 4 - Heat Equation with using NVIDIA libraries

This exercise implements an explicit finite difference iterative method for the Heat Equation in a one-dimensional metal rod using two CUDA libraries called cuSPARSE, cuBLAS, and Unified Memory. The cuBLAS library (https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublas-api) is used to accelerate BLAS (Basic Linear Algebra Subroutine) operations. The cuSPARSE library (https://docs.nvidia.com/cuda/cusparse/index.html#using-the-cusparse-api) is used in basic linear algebra subroutines in sparse matrices. Both libraries are highly optimized.

The idea is to model a metal rod with constant heat sources at the boundaries and simulate what is happening in the interior of the rod after a long period of time. The heat equation in 1D simulates this problem and is given by the differential equation:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

Using the Finite Differences Method the Heat Equation can be discretized into:

$$T_i^{n+1} = T_i^n + \alpha(T_{i-1}^n - 2T_i^n) + T_{i+1}^n$$

Given the boundary conditions and the discretization we obtain the algorithm:

```
for i from 1 to nsteps:
   tmp = A * temp
   temp = alpha * tmp + temp
   norm = vectorNorm(tmp)
   if tmp < 1e-4 :
      break
end
```

where nsteps is the number of time steps to simulate. The sparse matrix A of dimX rows and dimX columns is given by the tridiagonal matrix:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \end{pmatrix}$$

the relative error is given by:

$$\text{error} = \frac{\|\text{temp} - \text{exact}\|}{\|\text{exact}\|},$$

where temp is the computed approximation and exact is the exact solution of the problem. The exact solution is already given in the template code **lab4-ex4-template.cu (https://canvas.kth.se/courses/36161/files/6045543?wrap=1)** ↓ **(https://canvas.kth.se/courses/36161/files/6045543/download?download_frd=1)** . Please complete the following implementation. Instructions about where to place each part of the code is demarcated by the //@@ comment lines.

- Allocate unified memory needed by the sparse matrix, the temperature array and a temporal array
- Prefetch data to the appropriate locations.
- Add the necessary cuBLAS, cuSPARSE API functions to create and destroy the environments.
- Add the necessary cuBLAS, cuSPARSE routines to perform the algorithm.
- Add the necessary cuBLAS routines to perform the relative error.
- Deallocate memory

Questions to answer in the report

1. Run the program with different dimX values. For each one, approximate the FLOPS (floating-point operation per second) achieved in computing the SMPV (sparse matrix multiplication). Report FLOPS at different input sizes in a FLOPS. What do you see compared to the peak throughput you report in Lab2?
2. Run the program with dimX=128 and vary nsteps from 100 to 10000. Plot the relative error of the approximation at different nstep. What do you observe?
3. Compare the performance with and without the prefetching in Unified Memory. How is the performance impact? [Optional: using nvprof to get metrics on UM]