Assignment 4

Programming for Performance, Group SELF

Naman Singla 200619 nsingla20@iitk.ac.in

Question 1

Problem 1

[20+30+10+10+20 marks]

Consider the following code.

```
#define N 64
float in[N][N][N], out[N][N][N];
for (i=1; i<N-1; i++) {
 for (j=1; j<N-1; j++) {
  for (k=1; k<N-1; k++) {
   out[i][j][k]=0.8 * (in[i-1][j][k] + in[i+1][j][k] + in[i][j-1][k] +
              in[i][j+1][k] + in[i][j][k-1] + in[i][j][k+1]);
  }
 }
}

```

The above computation pattern is referred to as *stencil* pattern. In the stencil pattern, the value of a point is a function of the eight neighboring points (three in row $i-1$, two in row i, and three in row $i+1$). In the above listing, the access pattern has reuse on the array `in` in 3 dimensions.

You will implement five versions of the above stencil pattern. Your goal is to strive for getting as much speedup as possible with the second optimized kernel. Compare the performance of the different kernel versions, and explain your observations. Initialize the elements of `in` with random values.

(i) Implement a naïve CUDA kernel (i.e., no optimizations are required) that implements the above code.

(ii) Use shared memory tiling to improve the memory access performance of the code. Investigate and describe how the size of the block/tile computed by each thread block influences the performance. Experiment with blocks with sides comprising values from the set $\{1, 2, 4, 8\}$,

(iii) Implement other loop transformations like loop unrolling and loop permutation over version (ii). Explain your optimizations and highlight their impact.

(iv) Implement version (iii) using pinned memory (e.g., `cudaHostAlloc(..., cudaHostAllocDefault)`.

(v) Implement version (iii) using unified virtual memory (e.g., `cudaMallocManaged()`).

Answer:

(i) **Navie CUDA Version**

In this version, each thread computes its own cell(by picking data from Global memory and placing the result in global memory).

```
__global__ void navie_CUDA(float* in, float* out) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int k = blockIdx.z * blockDim.z + threadIdx.z;

  if(i==0||i==N-1||j==0||j==N-1||k==0||k==N-1){
    out[i * N * N + j * N + k] = 0;
  }else{
    out[i * N * N + j * N + k] = 0.8 * (in[(i-1) * N * N + j * N + k] +
                                        in[(i+1) * N * N + j * N + k] +
                                        in[i * N * N + (j-1) * N + k] +
```

```
                                                in[i * N * N + (j+1) * N + k] +
                                                in[i * N * N + j * N + (k-1)] +
                                                in[i * N * N + j * N + (k+1)]);
  }
}
```

(ii) **Tile Version**

```
__global__ void memTile_CUDA(float* in, float* out) {
  const int i = blockIdx.x * BLOCK_SIDE + threadIdx.x;
  const int j = blockIdx.y * BLOCK_SIDE + threadIdx.y;
  const int k = blockIdx.z * BLOCK_SIDE + threadIdx.z;
  const int is = threadIdx.x+1;
  const int js = threadIdx.y+1;
  const int ks = threadIdx.z+1;
  const int n = BLOCK_SIDE+2;
  __shared__ float data[(BLOCK_SIDE+2)*(BLOCK_SIDE+2)*(BLOCK_SIDE+2)];
  float ans = 0;
  __syncthreads();
  data[is * n * n + js * n + ks] = in[i * N * N + j * N + k];
  if(i!=0&&i!=N-1&&j!=0&&j!=N-1&&k!=0&&k!=N-1){
    if(is==1)data[(is-1)*n*n + js*n + ks] = in[(i-1)*N*N + j*N + k];
    if(js==1)data[is*n*n + (js-1) * n + ks] = in[i*N *N + (j-1) * N + k];
    if(ks==1)data[is*n*n + js*n + (ks-1)] = in[i*N*N + j*N + (k-1)];
    if(is==BLOCK_SIDE)data[(is+1)*n*n+js*n+ks] = in[(i+1)*N*N+j*N+k];
    if(js==BLOCK_SIDE)data[is*n*n+(js+1)*n+ks] = in[i*N*N+(j+1)*N+k];
    if(ks==BLOCK_SIDE)data[is*n*n+js*n+(ks+1)] = in[i*N*N+j*N+(k+1)];
  }
  __syncthreads();
  if(i!=0&&i!=N-1&&j!=0&&j!=N-1&&k!=0&&k!=N-1){
    ans += data[(is-1) * n * n + js * n + ks];
    ans += data[(is+1) * n * n + js * n + ks];
    ans += data[is * n * n + (js-1) * n + ks];
    ans += data[is * n * n + (js+1) * n + ks];
    ans += data[is * n * n + js * n + (ks-1)];
    ans += data[is * n * n + js * n + (ks+1)];
  }
  out[i * N * N + j * N + k] = 0.8 * ans;
}
```

In this version we first copy the data required for computation of a block into shared memory(all threads in block will use it). Then we perform all computations in shared memory and transfer the result to global final location.
Lets test its performance with different block size.

BLOCK-SIDE	Navie CUDA(ms)	Tile CUDA(ms)
1	0.483904	0.498144
2	0.366752	0.379296
4	0.302336	0.333792
8	0.325088	0.303968

CPU Time ≈3.5 ms
Hence, we are getting better speed for larger block sizes in case of memory tiling.

(iii) **Loop optimisation Version**

There is nothing much to improve as there is no loop. But we can change the access pattern and make it coleased for some statements (since it is all 6 neighbors it doesn't matter to other statements). Like statement 12 in the tile version(code in the previous part).
Interchange the following pair of variables in the tile version:

(a) i,k (statement 2,4)

(b) is, ks (statement 5,7)

By doing so, we observe a slight improvement(0.005ms) in performance.
NOTE:- Setting banking offset will not impact much here as shared data is a 3D array, and we access all directions in a single go.

(iv) **Pinned Version, Unified Version**

BLOCK-SIDE	Navie CUDA(ms)	Tile CUDA(ms)	LoopOP	Pinned	Unified
1	0.475456	0.484128	0.465312	0.277056	0.187392
2	0.321376	0.350048	0.318336	0.122880	0.030688
4	0.312096	0.323648	0.289920	0.101984	0.013536
8	0.342336	0.317504	0.307104	0.097408	0.012288

Consistent improvement :)

Lets see performance of kernels using nsight tool.

	Navie CUDA(ms)	Tile CUDA(ms)	LoopOP	Pinned	Unified
Compute (SM %)	15.42	19.69	36.82	37.60	37.88
Duration(us)	23.26	19.46	10.69	9.54	10.53
Active Warps Per SM	36.00	37.65	39.27	39.60	40.24

Question 3

Problem 3

[30+30+30+30 marks]

Parallelize the attached C code (`problem3-v0.c`) using CUDA. You will implement three versions.

1. The first version will be a vanilla port of the C code. Your goal in this version is to ensure correctness. The challenges are in mapping the large iteration space of the 10D loop nest and writing back the output data from the device to the host. Implementing optimizations is optional.

2. Improve the performance of version (i) using different possible optimizations (e.g., shared memory tiling, launch multiple kernels, data prefetching and memadvise).

3. Implement the C code with UVM support with CUDA.

4. Implement a version using different transformations provided by Thrust. A few Thrust transformations are well-suited for the given problem.

Compare the performance of the four versions.

Answer:

(i) **Navie CUDA Version**
The idea is to collapse all 10 loops into one. But due to memory limits, we will complete execution in multiple kernel runs(using an offset).
Time Taken: 4357.513434 seconds :')

(ii) **Optimised Version**
The code implemented in previous part is already using titling and multiple kernel execution. I have used pinned memory for more efficiency but still time remains that long. I also tried using streams(much more memory) but it didn't impact much.

(iii) **UVM Version**
Using cudaMallocManaged, performance gets worse(10x increase in execution time).

Device Used

```
Device 0: "NVIDIA A40"
  CUDA Driver Version / Runtime Version          11.7 / 11.8
  CUDA Capability Major/Minor version number:    8.6
  Total amount of global memory:                 45494 MBytes (47704375296
   bytes)
  (084) Multiprocessors, (128) CUDA Cores/MP:    10752 CUDA Cores
  GPU Max Clock rate:                            1740 MHz (1.74 GHz)
  Memory Clock rate:                             7251 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
   65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048
   layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        102400 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 134 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
    simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime
   Version = 11.8, NumDevs = 1
Result = PASS
```