Report on GPU Assignment

Necessary GPU Device Specification

Name: Tesla V100-SXM2-32GB

GPU count: 1

- Maximum number of threads per block: 1024
- Maximum dimension size of thread block: (1024, 1024, 64)
- Total amount of shared memory per block: 49152 Bytes
- Warp size: 32
- Global Memory: 32 GB
- Maximum dimension of grid size: (2147483647, 65535, 65535)

1. launchKernel() optimization

Given the input matrix dimension as follows,

$$(x, y) = (8 \times 1024, 8 \times 1024) = (8192, 8192) \tag{1}$$

As the block can have 1024 threads, we can take the shape of the block as

$$(32 \times 32) = 1024 \tag{2}$$

32 is exactly divisible by 8192. So, the grid size will be

$$((8192/32), (8192/32)) = (256, 256) \tag{3}$$

Finding: For both of the cases we got integers. So, due to the grid and block for the input matrix, there will not be any *warp divergence* problem. There can be other satisfiable shapes possible for both grid and block dimensions, however, the performance is better for these dimensions upon trial and error basis due to high *occupancy*.

Following is the respective code that automatically calculates based on the device property.

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);
//int num_sms = prop.multiProcessorCount;
int num_th = (int)ceil(sqrtf(prop.maxThreadsPerBlock));
int num_bkx = (int)ceil(dimx/num_th);
int num_bky = (int)ceil(dimy/num_th);

dim3 block(num_th, num_th);
dim3 grid(num_bkx , num_bky);
kernel_A<<<grid, block>>>(d_data, dimx, dimy, niterations);
```

2. Kernel_A() optimizations

2.1 Usage of intrinsic function

REGULAR	INTRINSIC
logf	__logf
cosf	__cosf
sinf	__sinf
tanf	__tanf

2.2 Removal of outer two loops

In the original unoptimized code, two outer loops were there to process multiple elements of the given matrix by a single thread. As we designed the grid and block in such a way (described in Section 1) that each item in the given matrix is processed by a single thread, we don't need to iterate over multiple items using an outer for loop. So, it was removed.

```
for (int iy = blockIdx.y * blockDim.y + threadIdx.y; iy < dimy;
       iy += blockDim.y * gridDim.y) {
    for (int ix = blockIdx.x * blockDim.x + threadIdx.x; ix < dimx;
         ix += blockDim.x * gridDim.x) {
```

The corresponding necessary code has been replaced by the above code to maintain accuracy.

2.3 Mod operator replacement with bitwise AND

The modulus operator (%) has been replaced by bitwise AND as we know mod operator is expensive. The logic is,

$$\text{something } \%n = \text{something } \&(n-1) \tag{4}$$

where & is bitwise AND.

2.4 Utilization of shared memory

Shared memory speeds up the access of data. As the block is (32 * 32), the size of shared memory is (32 * (32 +1)). Here additional element in each row helps to resolve the Bank conflict when we were reading from VRAM to shared memory (cache) and writing from Cache to VRAM in a transposed way. The size does not overflow the limit shown in Section 1.

```
// Transpose reading from global memory to shared memory
shared_data[threadIdx.x * paddedWidth + threadIdx.y] = g_data[idx];
.
.
.
// Transponse write from shared memory to global memory
g_data[idx] = shared_data[threadIdx.x * paddedWidth + threadIdx.y];
```

2.5 Memory Coalesce and Fixing Warp Divergence

We know when threads in the warp do different things, it hinders the performance. We cannot avoid the conditional statement altogether. However, we can make sure the threads in a warp do the same instruction execution. However, when memory is read from VRAM to the Cache, in a cycle a set of values is read (locality). If threads in warp work on the data in the Cache (shared memory), we can avoid multiple cycles to fetch the data into the Cache.

Reason for Warp Divergence

Initially, the conditional statements were given in such a way that for a single row, the operations on the data for *nterations* are varied for the consecutive column. For example, if the column (x-axis) is divisible by 4, then log function, otherwise if the reminder is one, cos function, and so on. However, as the GPUs are row-major. That means threads in a warp will follow row-major order, each thread working on consecutive data will process different functions which cause warp divergence, which is a hindrance to performance.

Solution

One idea is to transpose while reading into Cache. Transpose reading and writing can cause Bank conflict which has been handled by padding 1 additional memory space for each row in shared memory allocation. In subsection 2.4, we have shown the transpose reading.

Discussion

Due to memory coalesce, less number of clock cycle is needed to get the data from VRAM to Cache. The threads in Warp will perform the same operations in row-major order for transposing the 32* 32 matrix in Cache.

2.6 Loop Unrolling

As the *niterations* are small, we can use loop unrolling to utilize the vectorized operation. We cannot parallelize this portion due to having dependency of ith iteration on (i-1)th iteration.

Speedup:

The unoptimized code was taking 39.7 ms and our optimized code took only 1.75 ms.

Speed up = **22.69** X

code > cuda-assignment-public > cuda_prog.cu > computeCpuResults(float *, int, int, int, int)

```
61
62  __global__ void kernel_A(float *g_data, int dimx, int dimy, int niterations) {
63    // Using shared memory to improve memory access patterns
64
65    const int paddedWidth = 33;
66    extern __shared__ float shared_data[32 * 33];
67
68    // Calculate global thread coordinates
69    int ix = blockIdx.x * blockDim.x + threadIdx.x;
70    int iy = blockIdx.y * blockDim.y + threadIdx.y;
71    int idx = iy * dimx + ix;
72
73    // Load data into shared memory for better coalescence
74    if (ix < dimx && iy < dimy) {
75      shared_data[threadIdx.x * paddedWidth + threadIdx.y] = g_data[idx];
76    }
77
78    // Synchronize threads to ensure all data is loaded into shared memory
79    __syncthreads();
80
81    // Perform calculations on data in shared memory
82    if (ix < dimx && iy < dimy) {
83      float value = shared_data[threadIdx.y * paddedWidth + threadIdx.x];
84      int iy_mod_4 = iy & 3;
85      #pragma unroll
86      for (int i = 0; i < niterations; i++) {
87        float temp;
88        switch (iy_mod_4) {
89          case 0: temp = sqrtf(__logf(value) + 1.f); break;
90          case 1: temp = sqrtf(__cosf(value) + 1.f); break;
91          case 2: temp = sqrtf(__sinf(value) + 1.f); break;
92          case 3: temp = sqrtf(__tanf(value) + 1.f); break;
93        }
94        value += temp;
95      }
96      shared_data[threadIdx.y * paddedWidth + threadIdx.x] = value;
97    }
98
99    // Synchronize threads again before writing back to global memory
100   __syncthreads();
101
102   // Write back the results from shared memory to global memory
103   if (ix < dimx && iy < dimy) {
104     g_data[idx] = shared_data[threadIdx.x * paddedWidth + threadIdx.y];
105   }
```

PROBLEMS 58 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) sskg8@cuda-15-31:~/code/cuda-assignment-public$ make && ./baseline.x
make: Nothing to be done for 'all'.
allocated 256.00 MB on GPU
allocated 512.00 MB on CPU
Verifying solution
Rep: 0
Results are correct
A:      1.75 ms
CUDA: no error
(base) sskg8@cuda-15-31:~/code/cuda-assignment-public$
```