Name: Yifan Wang
PID: A53298382
Email: yiw021@ucsd.edu

ECE 277 GPU Programming
Midterm Report

Due on 2021-02-25

1 Introduction

The plan of doing optimization for midterm was first to study the GPU hardware information, to do tests for Lab3 code with midterm parameters setting as benchmark, and then, optimized the code.
In the midterm, I tried techniques include **reorganizing parallel threads**, **thread mapping**, **shared memory**, **parallel reduction**, **dynamic parallelism**, **understanding the nature of warp execution**, **avoiding branch divergence**, **consider coalescing**, **pragma unrolling**, **bit operators**, **data structures**, and corrected some mistakes I did in Lab2 or Lab3, such as **removed memory copy**, **adjust cuRAND range**.

2 System Information

First, I searched the CUDA Occupancy Calculator and get our lab machine system info, and tested Lab3 code with parameters (agents 512, size 46 x 46) as performance benchmark (results in Part 5 experiments).
Our lab machine info is below:

Physical Limits for GPU Compute Capability:	6.1
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

3 CUDA Programming

I listed my development progress into two parts, initialize agent and agent action or update. In the first initial part, it's difficult to find issues and debug but we can do Nsight profile test to see if our code is effective. In the second part, I listed my development progress which includes many ideas not in my final version code but helps a lot to understand how it works.

3.1 Initialize Agent

- Initialize parallel (line 50 - 58)

  In kernel functions, we can use every thread to do initialize work parallel, instead of using for-loop as it's in CPU host function. (This technique I used in Lab2, Lab3 and Midterm.)

- Understanding warp execution and occupancy (line 63 - 73)

  In midterm, the environment size is 46×46. When we initialized the Q-table, we can see the Q-table size is $46 \times 46 \times 4 = 8464$, which doesn't perfectly match the warp size 32. It means the occupancy for `init_qtable` may only reach 50% occupancy, which not effective.

  I consider the "golden rule" and adjusted the gridDim to ceiling size of $\lceil \frac{qSize}{\#threads} \rceil$ (line 95) and add a condition `tid < QSIZE` (line 72) to make it in the range of size. Then, I tested $32, 64, 128, 256, 512$ as the number of threads, which are times of warp size, and chose the size can reach occupancy 75%.

- Removed unnecessary variable `d_epsilon` (Lab3 line 50, 72, 113 - removed)

  In Lab3, I declared a pointer in global and did `cudaMemcpyDeviceToHost` after update the epsilon, which was unnecessary step. In midterm, the epsilon is a parameter which passes to kernel functions directly.

- The cuRAND range issue (line 151)

  Note that the cuRAND will generated from 0 to 1 which exclude 0.0 and include 1.0 which I didn't realized in previous Lab. So, basically we need to adjust the range from $(0.0, 1.0]$ to $[0.0, 1.0)$. We can use a if-condition to check if the ACTION value is 4 then change it to 0 to keep the random sampling uniform still. Or we can simply make `1.0f - curand_uniform()` then multiple by the number of actions in integer type, so the ACTION value will only generated as 0, 1, 2, 3.

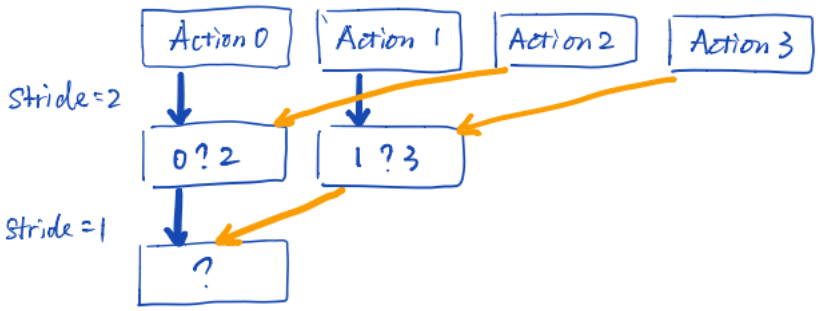
3.2 Agent Action & Update

3.2.1 Development Progress

In Lab3, my `Agent_action` kernel launched by `<<<1, MUN_AGENTS>>>`. In every thread, there was a for-loop to do linear search for the best action which had the max value in Q-table (Lab3 line 147 - 154). But I found it could be optimized by reduction as the finding max values is a typical **Parallel Reduction Problem**.
Here are the techniques which I tried and helped me to develop the final version code:

- Parallel Reduction for Single Agent's Actions (line 318 - 378, not use in final version)

  First, I tried to assign the number of agents as grid size and the number of actions for single agent as block size, which was `<<<NUM_AGENTS, ACTIONS>>>`. In each grid, there was 4 threads. So we could use shared memory in this block (an agent), and stored the 4 actions into a cache then did parallel reduction.



  At the 1st round, set the stride as 2 to compare (idx_0, idx_2) and (idx_1, idx_3). Keep larger values into left one. Then, at the 2nd round, set the stride as 1 to compare (idx_0, idx_1). Keep the larger value at idx_0.

  However, in this way, the occupancy was very low. Then, I tried the **Dynamic Parallelism**.

- Dynamic Parallelism (line 269 - 313, not use in final version)

  The idea of using dynamic parallelism was to set the number of agents as the block size then to launch `device GetMax` kernel `<<<1, ACTIONS>>>` in a `global` kernel function `<<<1, NUM_AGENTS>>>`, so the occupancy of global kernel functions might improve. Then, in the `device` function, we could do a **nested parallel reduction** for single agent, so it shared memory (size of the 4 actions) in the block to do reduction. It seems can work but I found there maybe a better way.

  I didn't choose this dynamic parallelism method, but this progress do help a lot for me to understand how memory shared works and how to avoid bank conflicts etc.

- Data Structure and AoS Struct (line 273 - 276)

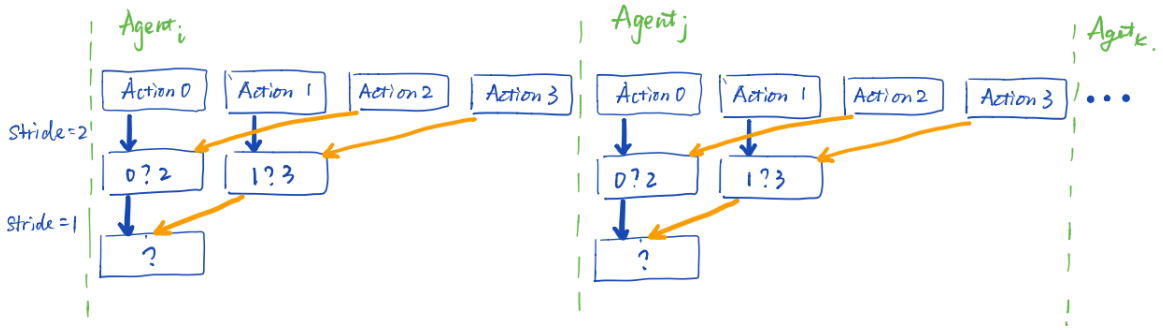
  I also tried other data structures, such as made `float4` as Q-table structure or created a data structure to store the best action and max q-value. However, I found it might cause troubles for parallel threadIdx as when storing in cache and do parallel reduction, we need to get the value by `qtable.x, qtable.y, qtable.z, qtable.w` separately by if-condition. So I didn't use this in my final version code.

3.2.2 Final Version - Parallel Reduction for Multiple Agents

Here I listed the final version of my midterm code. The code line here are mainly in kernel function `Agent_action`. The techniques I used in kernel function `Agent_update` are similar (line 204 - 255).

- Parallel Reduction for Multiple-Agents' Actions (line 171 - 186)

  The idea is similar as previous one, but this time I put several agents and their actions into threads and use **shared memory cache** to store their actions and q-values which are **row-wise read** and **coalesced access**. Then do decreasing stride 2 then 1, not increasing stride 1 then 2. This is because in more complicate case, dong decreasing stride (e.g. 8, 4, 2, then 1) usually will be wiser than the increasing way (e.g. 1, 2, 4, then 8) which may cause divergence issue (as the nature of warp execution, the entire warp schedule will do the same work). If it does parallel reduction by decreasing stride, when the "right side" half threads or warps finish their work early, the "right side" warps may start some other works.



- Reorganized Threads and Thread Mapping (line 141 - 146)

  Instead of using fixed number of agents, actions and environment size, I reorganized threads and used the thread mapping which made my code quite fixable to test by different size of threads and grids.

- Stencil Calculation (line 160 - 166)

  As in my strategy, I need to calculate the agents' id and locate their actions index. Note that some agents may go exploration or inactive, which means some of they won't go into the parallel reduction step to get the best action or max q-value. So the stencil calculation to located their index and positions correctly.

- Shared Memory Allocation (line 155 - 157)

  I used the **Static Shared Memory** as I found it's good for index stencil calculation This stencil calculation can located the agents' id and their action very stable and clear, which might not suit to use dynamic shared memory. Also, it avoids bank conflict.

- Synchronization (line 167, 180)

  The synchronization used in storing cache and every round of parallel reduction, as we need to wait all threads done their work to continue do parallel reduction.

- Pragma Unrolling (line 172)

  Unrolling technique usually is useful in loop. As in a for-loop, there would be about 3 conditions which will cost source. So the unrolling will expend the steps and reduce the cost.

- Bit Operators (line 160, 169, 181)

  As we know bit operations are faster, I changed mod and divide to bit manipulation:

  1. Mod (`sid % ACTIONS`) to (`sid &(ACTIONS - 1)`); 2. divided (`stribe / 2`) to (`stribe >> 1`)

  It's improve performance a lot. Its testing results show in the Part 5 experiments.

- Variable Type (line 30 - 36, 151)

  Declare variable as `0.0f` to avoid cost of covert type from double to float.

4 Algorithm Parameters

I didn't do much tuning parameter work as I think the point of this midterm is to use CUDA techniques.
I found if the delta epsilon is too small, the agents would do exploration but many of them not so useful if most of mines were found. However, if the delta epsilon is too large, the agents might lack of explorations and stuck in corners as they might prefer to the blank place. Also, the learning rate may effect how rewards effect the q-value. In lab3 and midterm algorithm, learning rate may balance rewards and blank space.
In the final version, I picked the learning rate as 0.5 and delta epsilon as 0.01.

5 Experiments

5.0.1 Test Lab3 Code

Before doing optimization, I tested the Lab3 code with the environment size and the number of agents of Midterm setting. The Lab3 code testing results in (Figure 1). Here, the kernel was simply launched by `COLS 46, ROWS 46, ACTIONS 4, NUM_AGENTS 512`. There was no very fixable thread mapping work, and no shared memory technique was used. We can see the occupancy of `Init_agent` and `Init_qtable` were both only 50%.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{1, 1, 1}	{512, 1, 1}	544,478.321	2,711.488	50.00%	39	0
2	Init_qtable	{46, 46, 1}	{4, 1, 1}	686,580.145	24.704	50.00%	8	0
3	Agent_update	{1, 1, 1}	{512, 1, 1}	22,087,371.025	18.656	100.00%	24	0
4	Agent_update	{1, 1, 1}	{512, 1, 1}	14,746,968.465	12.864	100.00%	24	0
5	Agent_update	{1, 1, 1}	{512, 1, 1}	50,339,928.241	12.064	100.00%	24	0
6	Agent_update	{1, 1, 1}	{512, 1, 1}	85,815,579.537	11.968	100.00%	24	0
7	Agent_update	{1, 1, 1}	{512, 1, 1}	75,888,663.761	11.360	100.00%	24	0
8	Agent_update	{1, 1, 1}	{512, 1, 1}	209,429,090.737	11.360	100.00%	24	0
9	Agent_update	{1, 1, 1}	{512, 1, 1}	235,564,068.817	11.328	100.00%	24	0
10	Agent_update	{1, 1, 1}	{512, 1, 1}	38,297,442.097	11.136	100.00%	24	0

Figure 1: Lab3 original code with environment size 46x46 and 512 agents.

5.0.2 Parallel Reduction for Single-Agent's Actions

First, I tried the parallel reduction with single agent's actions. Every block only process the actions for one agent, and 512 (agents) blocks were launched. The actions for each agent were shared itself memory (size 4) actions in cache to do reduction. However, the occupancy were all quite low and duration were longer than the Lab3 strategies did in (Figure 2). So I continued to trying new methods.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{1, 1, 1}	{512, 1, 1}	535,518.398	2,689.952	50.00%	40	0
2	Agent_action	{512, 1, 1}	{4, 1, 1}	33,552,969.982	23.424	50.00%	25	24
3	Agent_update	{512, 1, 1}	{4, 1, 1}	19,526,877.438	22.464	50.00%	19	16
4	Agent_update	{512, 1, 1}	{4, 1, 1}	62,322,673.726	21.440	50.00%	19	16
5	Agent_update	{512, 1, 1}	{4, 1, 1}	2,131,820.414	20.832	50.00%	19	16
6	Agent_update	{512, 1, 1}	{4, 1, 1}	46,778,935.294	20.128	50.00%	19	16
7	Agent_action	{512, 1, 1}	{4, 1, 1}	72,397,824.414	20.128	50.00%	25	24
8	Agent_update	{512, 1, 1}	{4, 1, 1}	55,631,491.294	20.096	50.00%	19	16
9	Agent_action	{512, 1, 1}	{4, 1, 1}	50,588,043.070	19.232	50.00%	25	24
10	Agent_update	{512, 1, 1}	{4, 1, 1}	39,852,372.894	18.976	50.00%	19	16

Figure 2: Parallel Reduction for Single-Agent's Actions (in development progress)

5.0.3 Parallel Reduction for Multiple-Agents' Actions (w/o Bit Operators.)

The final version I used in midterm code is parallel reduction with multiple-agents. In each block, there will be several agents with their actions' values storing in shared memory cache and then do reduction parallel to get their max values of their actions or q-values. In this version, I also used quite fixable thread mapping technique, so we can easily reorganize the grid and block sizes, and pick the best one. I chose the 256 threads one as my final version of midterm, which also shows a good performance in CUDA occupancy calculator.

Note that in the Q-learning case itself, the reduction we need is only for 2 rounds, so we can not see significant improvement. But in more complicate cases, the parallel reduction will show its power. We can see that the time complexity of the parallel reduction will be $O(N/P + logP)$ which is better than for-loop linear search. For details, consider a local reduction on each processors, which would take $O(N/P)$, and then compute a reduction of the P local results, which takes $O(logP)$ steps .

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{4, 1, 1}	{128, 1, 1}	558,611.263	2,654.048	75.00%	40	0
2	Agent_update	{16, 1, 1}	{128, 1, 1}	26,768,152.958	15.648	100.00%	19	512
3	Agent_update	{16, 1, 1}	{128, 1, 1}	18,061,155.839	14.591	100.00%	19	512
4	Agent_update	{16, 1, 1}	{128, 1, 1}	54,391,753.055	14.208	100.00%	19	512
5	Agent_update	{16, 1, 1}	{128, 1, 1}	70,950,331.039	12.319	100.00%	19	512
6	Agent_update	{16, 1, 1}	{128, 1, 1}	1,912,919.551	10.559	100.00%	19	512
7	Agent_update	{16, 1, 1}	{128, 1, 1}	48,702,184.031	10.367	100.00%	19	512
8	Agent_update	{16, 1, 1}	{128, 1, 1}	6,225,217.630	9.984	100.00%	19	512
9	Agent_update	{16, 1, 1}	{128, 1, 1}	30,949,653.854	9.792	100.00%	19	512
10	Agent_update	{16, 1, 1}	{128, 1, 1}	10,985,829.630	9.760	100.00%	19	512

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{2, 1, 1}	{256, 1, 1}	522,375.031	2,667.136	75.00%	40	0
2	Agent_update	{8, 1, 1}	{256, 1, 1}	43,667,262.775	15.200	100.00%	19	1024
3	Agent_update	{8, 1, 1}	{256, 1, 1}	22,330,657.207	12.384	100.00%	19	1024
4	Agent_update	{8, 1, 1}	{256, 1, 1}	44,256,875.575	10.272	100.00%	19	1024
5	Agent_update	{8, 1, 1}	{256, 1, 1}	16,318,619.703	9.856	100.00%	19	1024
6	Agent_update	{8, 1, 1}	{256, 1, 1}	100,394,370.167	9.856	100.00%	19	1024
7	Agent_update	{8, 1, 1}	{256, 1, 1}	10,503,585.911	9.664	100.00%	19	1024
8	Agent_update	{8, 1, 1}	{256, 1, 1}	31,778,806.903	9.312	100.00%	19	1024
9	Agent_update	{8, 1, 1}	{256, 1, 1}	93,862,396.695	9.280	100.00%	19	1024
10	Agent_update	{8, 1, 1}	{256, 1, 1}	203,472,309.623	9.280	100.00%	19	1024

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{1, 1, 1}	{512, 1, 1}	534,604.246	2,688.544	50.00%	40	0
2	Agent_update	{4, 1, 1}	{512, 1, 1}	52,060,777.718	31.840	100.00%	19	2048
3	Agent_update	{4, 1, 1}	{512, 1, 1}	23,325,719.894	14.944	100.00%	19	2048
4	Agent_update	{4, 1, 1}	{512, 1, 1}	56,652,596.790	13.536	100.00%	19	2048
5	Agent_update	{4, 1, 1}	{512, 1, 1}	74,207,840.342	12.192	100.00%	19	2048
6	Agent_update	{4, 1, 1}	{512, 1, 1}	181,780,869.462	10.880	100.00%	19	2048
7	Agent_update	{4, 1, 1}	{512, 1, 1}	86,797,823.574	10.720	100.00%	19	2048
8	Agent_update	{4, 1, 1}	{512, 1, 1}	149,608,845.654	10.720	100.00%	19	2048
9	Agent_update	{4, 1, 1}	{512, 1, 1}	132,206,101.270	10.560	100.00%	19	2048
10	Agent_update	{4, 1, 1}	{512, 1, 1}	192,267,889.942	10.560	100.00%	19	2048

Figure 3: Parallel Reduction for Multiple-Agents's Actions with 128, 256, 512 Threads **(w/o Bit Operators)**

5.0.4 Parallel Reductions for Multiple-Agents' Actions (with Bit Operators)

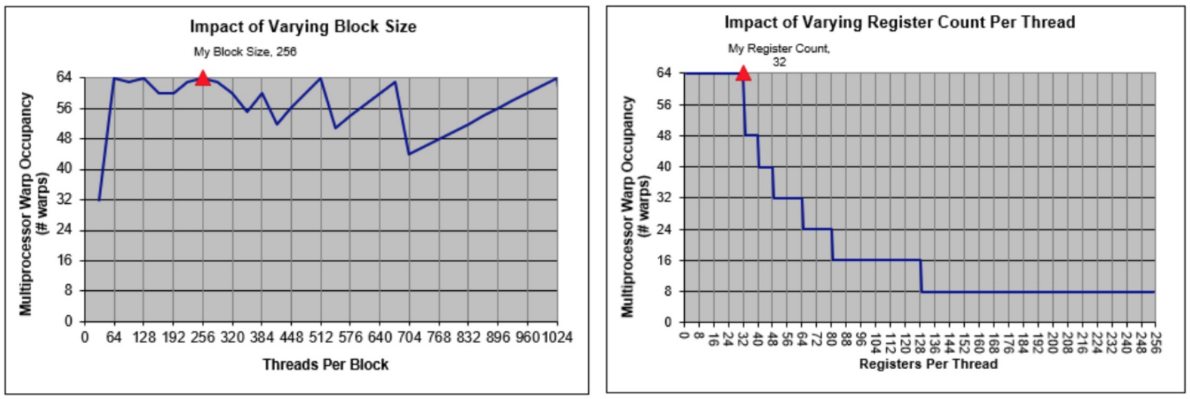


Figure 4: CUDA Occupancy Calculator Results (Compute Capability 6.1, Threads 256)

Finally, I rewrote the operators mod and divide to bit operators and the testing results show in (Figure 5). The performance (w bit) improved about at least 20% compared with the one w/o bit operators, and about 50% compared with the Lab3 code. We can see all `Agent_action` and most of `Agent_update` kernel functions run faster than the `env_step` functions.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{2, 1, 1}	{256, 1, 1}	525,843.085	2,665.952	75.00%	40	0
2	Agent_update	{8, 1, 1}	{256, 1, 1}	45,629,523.149	8.544	100.00%	19	1024
3	Agent_update	{8, 1, 1}	{256, 1, 1}	8,270,069.741	8.288	100.00%	19	1024
4	env_step	{1, 1, 1}	{512, 1, 1}	11,360,970.285	8.288	100.00%	22	0
5	env_step	{1, 1, 1}	{512, 1, 1}	125,240,141.293	8.288	100.00%	22	0
6	env_step	{1, 1, 1}	{512, 1, 1}	77,071,061.709	8.160	100.00%	22	0
7	env_step	{1, 1, 1}	{512, 1, 1}	206,270,521.677	8.160	100.00%	22	0
8	Agent_update	{8, 1, 1}	{256, 1, 1}	50,849,411.501	7.936	100.00%	19	1024
9	Agent_update	{8, 1, 1}	{256, 1, 1}	76,729,365.805	7.936	100.00%	19	1024
10	env_step	{1, 1, 1}	{512, 1, 1}	6,588,733.581	7.840	100.00%	22	0

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	Init_agent	{2, 1, 1}	{256, 1, 1}	525,843.085	2,665.952	75.00%	40	0
2	Agent_update	{8, 1, 1}	{256, 1, 1}	45,629,523.149	8.544	100.00%	19	1024
3	Agent_update	{8, 1, 1}	{256, 1, 1}	8,270,069.741	8.288	100.00%	19	1024
4	Agent_update	{8, 1, 1}	{256, 1, 1}	50,849,411.501	7.936	100.00%	19	1024
5	Agent_update	{8, 1, 1}	{256, 1, 1}	76,729,365.805	7.936	100.00%	19	1024
6	Agent_update	{8, 1, 1}	{256, 1, 1}	28,588,505.229	7.808	100.00%	19	1024
7	Agent_update	{8, 1, 1}	{256, 1, 1}	8,472,773.069	7.776	100.00%	19	1024
8	Agent_action	{8, 1, 1}	{256, 1, 1}	218,213,342.445	7.776	100.00%	25	1536
9	Agent_update	{8, 1, 1}	{256, 1, 1}	3,413,670.637	7.648	100.00%	19	1024
10	Agent_update	{8, 1, 1}	{256, 1, 1}	12,879,187.309	7.648	100.00%	19	1024

Figure 5: Parallel Reduction for Multiple-Agents's Actions for 256 Threads (**with Bit Operators**)

It's very interesting midterm :D

6 Reference

[1] John Cheng, Max Grossman, Ty McKercher, "Professional CUDA C Programming", Wiley, 2014.
[2] NVIDIA CUDA C++ Programming Guide