

PARALLEL COMPUTING

PPAR: GPU Lab1 and Lab2

Team

Ishan Tiwari

Shashi Mohan Reddy Ravula

1.Optimizing memory access.

Question 1. Program the simulation without optimizing memory accesses.

Answer

Below is the code without memory optimization of the function life_kernel in kernel code.

```
int neighbors=0;
int red=0, blue=0, blank=0;
    for (int i=-1; i<2; i++)
    {
        for (int j=-1; j<2; j++)
        {
            //control flow divergence
            if ((i !=0) || (j !=0))
            {
                neighbors = read_cell(shared_source_domain, tx, ty, i, j,
domain_x, domain_y);

                if (neighbors == 1)
                {
                    red++;
                }
                else if (neighbors == 2)
                {
                    blue++;
                }
                else if (neighbors == 0)
                {
                    blank++;
                }
            }
        }
    }
    // TODO: Compute new value

    int all_neighbors = red + blue;
    //control flow divergence
    if ((all_neighbors < 2) || (all_neighbors > 3))
    {
        myself = 0;
    }
    //control flow divergence
    else if ((all_neighbors == 2) || (all_neighbors == 3))
    {
        if ( blue >= 2)
        {
            myself = 2;
        }
        else
```

```

    {
        myself = 1;
    }
}
// TODO: Write it in dest_domain
dest_domain[(ty * domain_x) + tx] = myself;
}

```

Question 2. How many reads are performed for each cell ? are they coalesced reads ?

Answer 9

Each cell will be read 8 times by its neighbors and it will read itself while knowing its neighbor so in total 9 times. And yes they are coalesced reads which is implemented in below code.

```

__device__ int read_cell(int * source_domain, int x, int y, int dx, int dy,
    unsigned int domain_x, unsigned int domain_y)
{
    x = (unsigned int)(x + dx) % domain_x;    // Wrap around
    y = (unsigned int)(y + dy) % domain_y;
    return source_domain[y * domain_x + x];
}

```

Question3. Consider 1 thread block. What is the set of all location that are read by threads of the block ?

Answer

Since we are using 2 dimensional grid with `dim3 grid(1, 128)` so it will make an array 1*128 matrix of blocks and each block will act as an 1-D array with 128 elements.

```

GRID
[0120120120...till 128 elements] Block 1
[0120120120...till 128 elements] Block 2
.
.
.
. 128 blocks
.
.
.
[0120120120...till 128 elements] Block 128

```

So all the threads of a block will read elements of a block which in the grid means will read a row of the grid.

Question4. If blocks are 2 Dimensional ? Which block shape would minimize the number of unique locations read ?

Answer

The best shape of the block will be a square matrix since with a square matrix a cell can have all its neighbors in one block which would eventually minimize the unique locations read.

The grid dimension can be 16 x 16 and the block dimension can be 8 x 8 so each block will have 64 threads. So in total $16 \times 16 = 256$ blocks and $256 \times 64 = 128 \times 128$ threads.

```
int threads_per_block = 16;
int blocks_x = domain_x/(threads_per_block/2 * cells_per_word);
int blocks_y = domain_y/(threads_per_block/2 * cells_per_word);
dim3 grid(blocks_x, blocks_y);      // CUDA grid dimensions
dim3 threads(threads_per_block/2, threads_per_block/2);    // CUDA
block dimensions
```

Question 5. Use shared memory to remove duplicates.

Answer

We can reduce the number of duplicate reads from global memory by utilizing shared memory as a cache . For a 8 X 8 block we would have had to perform 576 reads from global memory (9 for each 64 threads) . But loading the values for the same block into shared memory for the same 64 output values will reduce the number of reads.

To remove duplicates we have used a shared variable which will have all the values from a block and will be accessed only by the threads of a block.

We have highlighted the code for shared memory usage below

```
int tx = blockIdx.x * blockDim.x + threadIdx.x;
int ty = blockIdx.y * blockDim.y + threadIdx.y;

extern __shared__ int shared_source_domain[];
// we put elements of a block by traversing source domain

for (int i=tx; i<tx+8; i++)
{
    for (int j=ty; j<ty+8; j++)
    {
        shared_source_domain[i * 8 + j] = source_domain[i *
domain_x + j];
    }
}
```

```

    }
    }
__syncthreads();

// Read cell
int myself = read_cell(shared_source_domain, tx, ty, 0, 0,
                        domain_x, domain_y);

```

Question 6. Where is there control flow divergence in your kernel? Can you reduce divergence?

Answer

As the code which we have given in the question 1. We have given comments(in green) where there is a possibility of a divergence in the control. Well, divergence can lead to bad performance since few threads will have different flow and others will have different. But in our code there is minimal computations under divergence. We have some other if-else blocks inside divergence but using the ternary operator will not solve anything in terms of performance. So we cannot reduce the divergence in our code.

Question 7. Edit the compiler options to pass the option -xptxas=-v to nvcc. It will now print the number of registers/threads used in the compiler output window. How many registers do you get?

Answer

The option will enable verbose mode which prints code generation statistics.

we know that there are 512 threads per CTA, and there are 15 registers per thread. 9 threads out of 15 threads contains redundant data. Local and shared memory are listed by two numbers each, cmem[1] and smem respectively.

sm_35 is the GPU architecture annotated with functional capabilities that they provide, like in _35 it supports for dynamic parallelism.

We see there are 7 and 16 registers in starting and then 25 and 7 registers being used all the later part of the execution.

```

>
>H:/Documents/PPAR/gpu_lab3/life.cu(20): warning : variable "blocks_y" was declared but never referenced
>
> ptxas : info : 0 bytes gmem
> ptxas : info : Compiling entry function '_Z11init_kernelPii' for 'sm_10'
> ptxas : info : Used 7 registers, 32 bytes smem, 20 bytes cmem[1]
> ptxas : info : Compiling entry function '_Z11life_kernelPiS_ii' for 'sm_10'
> ptxas : info : Used 16 registers, 32 bytes smem, 16 bytes cmem[1]
> life.cu
>h:\documents\ppar\gpu_lab3\life_kernel.cu(52): warning : variable "blank" was declared but never referenced
>
>H:/Documents/PPAR/gpu_lab3/life.cu(19): warning : variable "blocks_x" was declared but never referenced
>
>H:/Documents/PPAR/gpu_lab3/life.cu(20): warning : variable "blocks_y" was declared but never referenced
>

```

```
Output
Show output from: Build Order
1> life.cu
1>h:\documents\ppar\gpu_lab3\life_kernel.cu(52): warning : variable "blank" was
1>
1>H:/Documents/PPAR/gpu_lab3/life.cu(19): warning : variable "blocks_x" was dec
1>
1>H:/Documents/PPAR/gpu_lab3/life.cu(20): warning : variable "blocks_y" was dec
1>
1> ptxas : info : 0 bytes gmem
1> ptxas : info : Function properties for _Z9read_cellPiijjj
1> 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1> ptxas : info : Compiling entry function '_Z11life_kernelPiS_ii' for 'sm_35'
1> ptxas : info : Function properties for _Z11life_kernelPiS_ii
1> 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1> ptxas : info : Used 25 registers, 336 bytes cmem[0]
1> ptxas : info : Compiling entry function '_Z11init_kernelPii' for 'sm_35'
1> ptxas : info : Function properties for _Z11init_kernelPii
1> 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
1> ptxas : info : Used 7 registers, 328 bytes cmem[0]
1> tmpxft_00000580_00000000-11_life.compute_10.cudafec1.cpb
1> tmpxft_00000580_00000000-39_life.compute_10.ii
1>CudaBuild:
1> Skipping CUDA source file life_kernel.cu (excluded from build).
1>CudaBuild:
1> Compiling CUDA source file utils.cu...
1>
1> H:\Documents\PPAR\gpu_lab3>"c:\Program Files\NVIDIA GPU Computing Toolkit\CI
1> utils.cu
1>H:/Documents/PPAR/gpu_lab3/utils.cu(39): warning : variable "timerStart" was
```

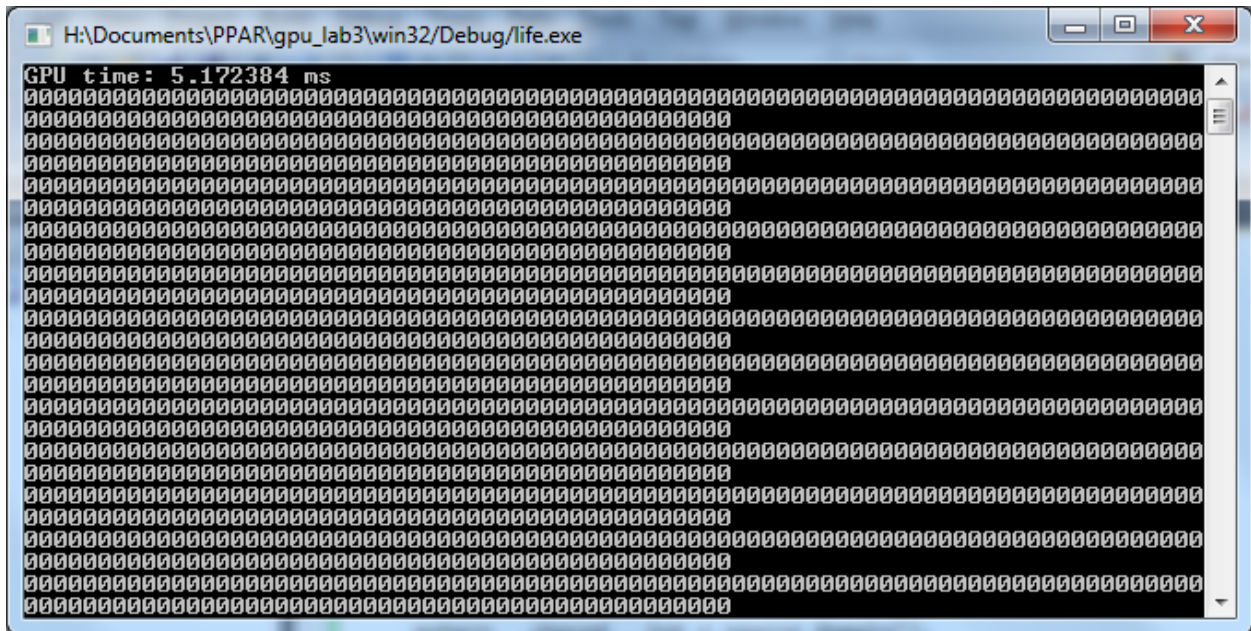
Question 8. Instead of having one thread per cell, have each thread compute 16 cells. What is the runtime and register usage?

Answer

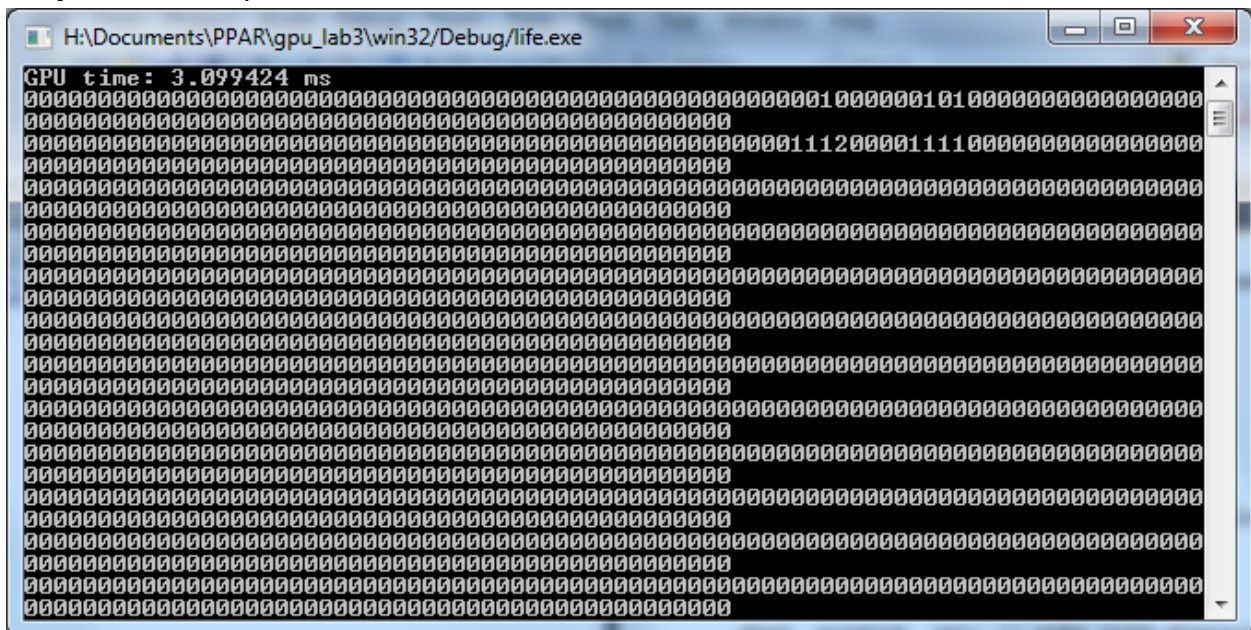
We have now, 2x2 threads in a block where a thread is computing 16 cells at a time in a block. We see that there is significant decrease in the runtime of the code(snapshots below). But we saw no change in the use of number of registers.

```
int threads_per_block = 16;
int blocks_x = domain_x/(threads_per_block/2 * cells_per_word);
int blocks_y = domain_y/(threads_per_block/2 * cells_per_word);
dim3 grid(blocks_x, blocks_y); // CUDA grid dimensions
dim3 threads(threads_per_block/8, threads_per_block/8);
```

Output: 1 thread per cell



Output: 1 thread per 16 cells



Question 9. Use data packing optimization mentioned above . How many reads are performed for each cell .

Answer

We can pack the values 0,1,2 in a 32-bit int. The data packing optimization can be done using left shift operator and we can store 3 numbers inside a single 32-bit int. We are packing

three numbers in the initialization part and then we are unpacking the values in the read_cell() function of the kernel.

```
int cells_per_word = 3;
int steps = 2;
int threads_per_block = 16;
int blocks_x = domain_x/(threads_per_block/2 * cells_per_word);
int blocks_y = domain_y/(threads_per_block/2 * cells_per_word);
dim3 grid(blocks_x, blocks_y);      // CUDA grid dimensions
dim3 threads(threads_per_block/2, threads_per_block/2);    // CUDA
block dimensions
dim3 threads(threads_per_block/8, threads_per_block/8);    // CUDA
block dimensions for 1 thread computing 16 cells
// Allocation of arrays
int * domain_gpu[2] = {NULL, NULL};
// Arrays of dimensions domain.x * domain.y
size_t domain_size = domain_x * domain_y / cells_per_word *
sizeof(int);
```

life_kernel.cu

```
__global__ void init_kernel(int * domain, int domain_x)
{
    // Dummy initialization
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * domain_x + ix;
    int x=1,y=1,z=1;
    if( 1664525ul * (blockIdx.x + threadIdx.y + threadIdx.x) +
        1013904223ul) % 3== 0)
    {
        x=0;
    }
    else if(1664525ul * (blockIdx.x + threadIdx.y + threadIdx.x) +
        1013904223ul) % 3==1)
    {
        y=1;
    }
    else
    {
        z=z<<1;
    }
    //Packing the values in single value
    domain[idx] = x | y | z;}
```