

## Chapter 4 Solutions

### Case Study: Implementing a Vector Kernel on a Vector Processor and GPU

#### 4.1 MIPS code (answers may vary)

```

        li      $r1,#0          # initialize k
loop:   l.s     $f0,0($RtipL)    # load all values for first
                                # expression

        l.s     $f1,0($Rc1L)
        l.s     $f2,4($RtipL)
        l.s     $f3,4($Rc1L)
        l.s     $f4,8($RtipL)
        l.s     $f5,8($Rc1L)
        l.s     $f6,12($RtipL)
        l.s     $f7,12($Rc1L)
        l.s     $f8,0($RtipR)
        l.s     $f9,0($Rc1R)
        l.s     $f10,4($RtipR)
        l.s     $f11,4($Rc1R)
        l.s     $f12,8($RtipR)
        l.s     $f13,8($Rc1R)
        l.s     $f14,12($RtipR)
        l.s     $f15,12($Rc1R)
        mul.s   $f16,$f0,$f1    # first four multiplies
        mul.s   $f17,$f2,$f3
        mul.s   $f18,$f4,$f5
        mul.s   $f19,$f6,$f7
        add.s   $f20,$f16,$f17  # accumulate
        add.s   $f20,$f20,$f18
        add.s   $f20,$f20,$f19
        mul.s   $f16,$f8,$f9    # second four multiplies
        mul.s   $f17,$f10,$f11
        mul.s   $f18,$f12,$f13
        mul.s   $f19,$f14,$f15
        add.s   $f21,$f16,$f17  # accumulate
        add.s   $f21,$f21,$f18
        add.s   $f21,$f21,$f19
        mul.s   $f20,$f20,$f21  # final multiply
        st.s    $f20,0($Rc1P)   # store result
        add     $Rc1P,$Rc1P,#4  # increment c1P for next
                                # expression
        add     $RtipL,$RtipL,#16 # increment tiPL for next
                                # expression

```

```

add    $RtiPR,$RtiPR,#16    # increment tiPR for next
                             expression
addi   $r1,$r1,#1
and    $r2,$r2,#3           # check to see if we should
                             increment c1L and c1R (every
                             4 bits)
bneq   $r2,skip
add    $Rc1L,$Rc1L,#16     # increment tiPL for next loop
                             iteration
add    $Rc1R,$Rc1R,#16     # increment tiPR for next loop
                             iteration
skip:  blt    $r1,$r3,loop   # assume r3 = seq_length * 4

```

**VMIPS code (answers may vary)**

```

li      $r1,#0              # initialize k
li      $VL,#4              # initialize vector length
loop:   lv      $v0,0($Rc1L)
        lv      $v1,0($Rc1R)
        lv      $v2,0($RtipL)    # load all tiPL values
        lv      $v3,16($RtipL)
        lv      $v4,32($RtipL)
        lv      $v5,48($RtipL)
        lv      $v6,0($RtipR)    # load all tiPR values
        lv      $v7,16($RtipR)
        lv      $v8,32($RtipR)
        lv      $v9,48($RtipR)
mulvv.s $v2,$v2,$v0          # multiply left
                             sub-expressions
mulvv.s $v3,$v3,$v0
mulvv.s $v4,$v4,$v0
mulvv.s $v5,$v5,$v0
mulvv.s $v6,$v6,$v1          # multiply right
                             sub-expression
mulvv.s $v7,$v7,$v1
mulvv.s $v8,$v8,$v1
mulvv.s $v9,$v9,$v1
sumr.s  $f0,$v2              # reduce left sub-expressions
sumr.s  $f1,$v3
sumr.s  $f2,$v4
sumr.s  $f3,$v5
sumr.s  $f4,$v6              # reduce right
                             sub-expressions
sumr.s  $f5,$v7
sumr.s  $f6,$v8
sumr.s  $f7,$v9
mul.s   $f0,$f0,$f4          # multiply left and right
                             sub-expressions

```

```

mul.s    $f1,$f1,$f5
mul.s    $f2,$f2,$f6
mul.s    $f3,$f3,$f7
s.s      $f0,0($Rclp)      # store results
s.s      $f1,4($Rclp)
s.s      $f2,8($Rclp)
s.s      $f3,12($Rclp)
add      $RtiPL,$RtiPL,#64 # increment tiPL for next
                           # expression
add      $RtiPR,$RtiPR,#64 # increment tiPR for next
                           # expression
add      $RclP,$RclP,#16   # increment clP for next
                           # expression
add      $RclL,$RclL,#16   # increment clL for next
                           # expression
add      $RclR,$RclR,#16   # increment clR for next
                           # expression
addi     $r1,$r1,#1
blt      $r1,$r3,loop      # assume r3 = seq_length

```

4.2 MIPS: loop is 41 instructions, will iterate  $500 \times 4 = 2000$  times, so roughly 82000 instructions

VMIPS: loop is also 41 instructions but will iterate only 500 times, so roughly 20500 instructions

```

4.3  1.      lv                      # clL
      2.      lv                      # clR
      3.      lv      mulvv.s        # tiPL 0
      4.      lv      mulvv.s        # tiPL 1
      5.      lv      mulvv.s        # tiPL 2
      6.      lv      mulvv.s        # tiPL 3
      7.      lv      mulvv.s        # tiPR 0
      8.      lv      mulvv.s        # tiPR 1
      9.      lv      mulvv.s        # tiPR 2
     10.      lv      mulvv.s        # tiPR 3
     11.      sumr.s
     12.      sumr.s
     13.      sumr.s
     14.      sumr.s
     15.      sumr.s
     16.      sumr.s
     17.      sumr.s
     18.      sumr.s

```

18 chimes, 4 results, 15 FLOPS per result,  $18/15 = 1.2$  cycles per FLOP

- 4.4 In this case, the 16 values could be loaded into each vector register, performing vector multiplies from four iterations of the loop in single vector multiply instructions. This could reduce the iteration count of the loop by a factor of 4. However, without a way to perform reductions on a subset of vector elements, this technique cannot be applied to this code.
- 4.5
- ```

__global__ void compute_condLike (float *c1L, float *c1R, float
*c1P, float *tiPL, float *tiPR) {
    int i,k = threadIdx.x;
    __shared__ float c1L_s[4], c1R_s[4];

    for (i=0;i<4;i++) {
        c1L_s[i]=c1L[k*4+i];
        c1R_s[i]=c1R[k*4+i];
    }

    c1P[k*4] = (tiPL[k*16+AA]*c1L_s[A] +
tiPL[k*16+AC]*c1L_s[C] + tiPL[k*16+AG]*c1L_s[G] +
tiPL[k*16+AT]*c1L_s[T])*(tiPR[k*16+AA]*c1R_s[A] +
tiPR[k*16+AC]*c1R_s[C] + tiPR[k*16+AG]*c1R_s[G] +
tiPR[k*16+AT]*c1R_s[T]);

    c1P[k*4+1] = (tiPL[k*16+CA]*c1L_s[A] +
tiPL[k*16+CC]*c1L_s[C] + tiPL[k*16+CG]*c1L_s[G] +
tiPL[k*16+CT]*c1L_s[T])*(tiPR[k*16+CA]*c1R_s[A] +
tiPR[k*16+CC]*c1R_s[C] + tiPR[k*16+CG]*c1R_s[G] +
tiPR[k*16+CT]*c1R_s[T]);

    c1P[k*4+2] = (tiPL[k*16+GA]*c1L_s[A] +
tiPL[k*16+GC]*c1L_s[C] + tiPL[k*16+GG]*c1L_s[G] +
tiPL[k*16+GT]*c1L_s[T])*(tiPR[k*16+GA]*c1R_s[A] +
tiPR[k*16+GC]*c1R_s[C] + tiPR[k*16+GG]*c1R_s[G] +
tiPR[k*16+GT]*c1R_s[T]);

    c1P[k*4+3] = (tiPL[k*16+TA]*c1L_s[A] +
tiPL[k*16+TC]*c1L_s[C] + tiPL[k*16+TG]*c1L_s[G] +
tiPL[k*16+TT]*c1L_s[T])*(tiPR[k*16+TA]*c1R_s[A] +
tiPR[k*16+TC]*c1R_s[C] + tiPR[k*16+TG]*c1R_s[G] +
tiPR[k*16+TT]*c1R_s[T]);
}

4.6 c1P[threadIdx.x*4 + blockIdx.x+12*500*4]
c1P[threadIdx.x*4+1 + blockIdx.x+12*500*4]
c1P[threadIdx.x*4+2+ blockIdx.x+12*500*4]
c1P[threadIdx.x*4+3 + blockIdx.x+12*500*4]

c1L[threadIdx.x*4+i+ blockIdx.x*2*500*4]
c1R[threadIdx.x*4+i+ (blockIdx.x*2+1)*500*4]

```

```

tipL[threadIdx.x*16+AA + blockIdx.x*2*500*16]
tipL[threadIdx.x*16+AC + blockIdx.x*2*500*16]
...
tipL[threadIdx.x*16+TT + blockIdx.x*2*500*16]
tipR[threadIdx.x*16+AA + (blockIdx.x*2+1)*500*16]
tipR[threadIdx.x*16+AC +1 + (blockIdx.x*2+1)*500*16]
...
tipR[threadIdx.x*16+TT +15+ (blockIdx.x*2+1)*500*16]

```

4.7

|               |                      |                                       |
|---------------|----------------------|---------------------------------------|
| mul.u64       | %r1, %ctaid.x, 4000  | # compute address of c1L              |
| mul.u64       | %r2, %tid.x, 4       | # multiply block index by 4000        |
| add.u64       | %r1, %r1, %r2        | # multiply thread index by 4          |
| ld.param.u64  | %r2, [c1L]           | # add products                        |
| add.u64       | %r1, %r2, %r2        | # load base address of c1L            |
|               |                      | # add base to offset                  |
|               |                      | # compute address of c1R              |
| add.u64       | %r2, %ctaid.x, 1     | # add 1 to block index                |
| mul.u64       | %r2, %r2, 4000       | # multiply by 4000                    |
| mul.u64       | %r3, %tid.x, 4       | # multiply thread index by 4          |
| add.u64       | %r2, %r2, %r3        | # add products                        |
| ld.param.u64  | %r3, [c1R]           | # load base address of c1R            |
| add.u64       | %r2, %r2, %r3        | # add base to offset                  |
| ld.global.f32 | %f1, [%r1+0]         | # move c1L and c1R into shared memory |
| st.shared.f32 | [c1L_s+0], %f1       | # (unroll the loop)                   |
| ld.global.f32 | %f1, [%r2+0]         |                                       |
| st.shared.f32 | [c1R_s+0], %f1       |                                       |
| ld.global.f32 | %f1, [%r1+4]         |                                       |
| st.shared.f32 | [c1L_s+4], %f1       |                                       |
| ld.global.f32 | %f1, [%r2+4]         |                                       |
| st.shared.f32 | [c1R_s+4], %f1       |                                       |
| ld.global.f32 | %f1, [%r1+8]         |                                       |
| st.shared.f32 | [c1L_s+8], %f1       |                                       |
| ld.global.f32 | %f1, [%r2+8]         |                                       |
| st.shared.f32 | [c1R_s+8], %f1       |                                       |
| ld.global.f32 | %f1, [%r1+12]        |                                       |
| st.shared.f32 | [c1L_s+12], %f1      |                                       |
| ld.global.f32 | %f1, [%r2+12]        |                                       |
| st.shared.f32 | [c1R_s+12], %f1      |                                       |
|               |                      | # compute address of tiPL:            |
| mul.u64       | %r1, %ctaid.x, 16000 | # multiply block index by 4000        |
| mul.u64       | %r2, %tid.x, 64      | # multiply thread index by 16         |
|               |                      | floats                                |
| add.u64       | %r1, %r1, %r2        | # add products                        |

```

ld.param.u64    %r2, [tiPL]           # load base address of tiPL
add.u64         %r1,%r2,%r2          # add base to offset

add.u64         %r2, %ctaid.x,1       # compute address of tiPR:
mul.u64         %r2, %r2, 16000       # multiply block index by 4000
mul.u64         %r3, %tid.x, 64       # multiply thread index by 16 floats
add.u64         %r2, %r2, %r3        # add products
ld.param.u64    %r3, [tiPR]          # load base address of tiPL
add.u64         %r2,%r2,%r3          # add base to offset

mul.u64         %r3, %r3, 24000       # compute address of clP:
mul.u64         %r4, %tid.x, 16       # multiply block index by 4000
add.u64         %r3, %r3, %r4        # multiply thread index by 4 floats
ld.param.u64    %r4, [tiPR]          # add products
add.u64         %r3,%r3,%r4          # load base address of tiPL
ld.global.f32   %f1,[%r1]            # add base to offset
ld.global.f32   %f2,[%r1+4]          # load tiPL[0]
...
ld.global.f32   %f16,[%r1+60]        # load tiPL[15]
ld.global.f32   %f17,[%r2]          # load tiPR[0]
ld.global.f32   %f18,[%r2+4]        # load tiPR[1]
...
ld.global.f32   %f32,[%r1+60]        # load tiPR[15]
ld.shared.f32   %f33,[c1L_s]         # load c1L
ld.shared.f32   %f34,[c1L_s+4]
ld.shared.f32   %f35,[c1L_s+8]
ld.shared.f32   %f36,[c1L_s+12]
ld.shared.f32   %f37,[c1R_s]         # load c1R
ld.shared.f32   %f38,[c1R_s+4]
ld.shared.f32   %f39,[c1R_s+8]
ld.shared.f32   %f40,[c1R_s+12]

mul.f32         %f1,%f1,%f33         # first expression
mul.f32         %f2,%f2,%f34
mul.f32         %f3,%f3,%f35
mul.f32         %f4,%f4,%f36
add.f32         %f1,%f1,%f2
add.f32         %f1,%f1,%f3
add.f32         %f1,%f1,%f4
mul.f32         %f17,%f17,%f37
mul.f32         %f18,%f18,%f38
mul.f32         %f19,%f19,%f39
mul.f32         %f20,%f20,%f40
add.f32         %f17,%f17,%f18
add.f32         %f17,%f17,%f19
add.f32         %f17,%f17,%f20
st.global.f32   [%r3],%f17          # store result

```

```

mul.f32      %f5,%f5,%f33      # second expression
mul.f32      %f6,%f6,%f34
mul.f32      %f7,%f7,%f35
mul.f32      %f8,%f8,%f36
add.f32      %f5,%f5,%f6
add.f32      %f5,%f5,%f7
add.f32      %f5,%f5,%f8
mul.f32      %f21,%f21,%f37
mul.f32      %f22,%f22,%f38
mul.f32      %f23,%f23,%f39
mul.f32      %f24,%f24,%f40
add.f32      %f21,%f21,%f22
add.f32      %f21,%f21,%f23
add.f32      %f21,%f21,%f24
st.global.f32 [%r3+4],%f21      # store result

mul.f32      %f9,%f9,%f33      # third expression
mul.f32      %f10,%f10,%f34
mul.f32      %f11,%f11,%f35
mul.f32      %f12,%f12,%f36
add.f32      %f9,%f9,%f10
add.f32      %f9,%f9,%f11
add.f32      %f9,%f9,%f12
mul.f32      %f25,%f25,%f37
mul.f32      %f26,%f26,%f38
mul.f32      %f27,%f27,%f39
mul.f32      %f28,%f28,%f40
add.f32      %f25,%f26,%f22
add.f32      %f25,%f27,%f23
add.f32      %f25,%f28,%f24
st.global.f32 [%r3+8],%f25      # store result

mul.f32      %f13,%f13,%f33      # fourth expression
mul.f32      %f14,%f14,%f34
mul.f32      %f15,%f15,%f35
mul.f32      %f16,%f16,%f36
add.f32      %f13,%f14,%f6
add.f32      %f13,%f15,%f7
add.f32      %f13,%f16,%f8
mul.f32      %f29,%f29,%f37
mul.f32      %f30,%f30,%f38
mul.f32      %f31,%f31,%f39
mul.f32      %f32,%f32,%f40
add.f32      %f29,%f29,%f30
add.f32      %f29,%f29,%f31
add.f32      %f29,%f29,%f32
st.global.f32 [%r3+12],%f29      # store result

```

- 4.8 It will perform well, since there are no branch divergences, all memory references are coalesced, and there are 500 threads spread across 6 blocks (3000 total threads), which provides many instructions to hide memory latency.

### Exercises

- 4.9 a. This code reads four floats and writes two floats for every six FLOPs, so arithmetic intensity =  $6/6 = 1$ .

- b. Assume MVL = 64:

```

        li          $VL,44          # perform the first 44 ops
        li          $r1,0           # initialize index
loop:   lv          $v1,a_re+$r1     # load a_re
        lv          $v3,b_re+$r1     # load b_re
        mulvv.s     $v5,$v1,$v3     # a_re*b_re
        lv          $v2,a_im+$r1     # load a_im
        lv          $v4,b_im+$r1     # load b_im
        mulvv.s     $v6,$v2,$v4     # a_im*b_im
        subvv.s     $v5,$v5,$v6     # a_re*b_re - a_im*b_im
        sv          $v5,c_re+$r1     # store c_re
        mulvv.s     $v5,$v1,$v4     # a_re*b_im
        mulvv.s     $v6,$v2,$v3     # a_im*b_re
        addvv.s     $v5,$v5,$v6     # a_re*b_im + a_im*b_re
        sv          $v5,c_im+$r1     # store c_im
        bne         $r1,0,else       # check if first iteration
        addi        $r1,$r1,#44      # first iteration,
                                     # increment by 44
        j loop                    # guaranteed next iteration
else:   addi        $r1,$r1,#256     # not first iteration,
                                     # increment by 256
skip:   blt         $r1,1200,loop    # next iteration?

```

- c.

```

1.      mulvv.s     lv          # a_re * b_re (assume already
                                # loaded), load a_im
2.      lv          mulvv.s     # load b_im, a_im*b_im
3.      subvv.s     sv          # subtract and store c_re
4.      mulvv.s     lv          # a_re*b_im, load next a_re vector
5.      mulvv.s     lv          # a_im*b_re, load next b_re vector
6.      addvv.s     sv          # add and store c_im

```

6 chimes



d. total cycles per iteration =

$6 \text{ chimes} \times 64 \text{ elements} + 15 \text{ cycles (load/store)} \times 6 + 8 \text{ cycles (multiply)} \times 4 + 5 \text{ cycles (add/subtract)} \times 2 = 516$

cycles per result =  $516/128 = 4$

e.

```

1. mulvv.s          # a_re*b_re
2. mulvv.s          # a_im*b_im
3. subvv.s  sv      # subtract and store c_re
4. mulvv.s          # a_re*b_im
5. mulvv.s  lv      # a_im*b_re, load next a_re
6. addvv.s  sv  lv  lv  lv  # add, store c_im, load next b_re, a_im, b_im

```

Same cycles per result as in part c. Adding additional load/store units did not improve performance.

4.10 Vector processor requires:

- $(200 \text{ MB} + 100 \text{ MB})/(30 \text{ GB/s}) = 10 \text{ ms}$  for vector memory access +
- 400 ms for scalar execution.

Assuming that vector computation can be overlapped with memory access, total time = 410 ms.

The hybrid system requires:

- $(200 \text{ MB} + 100 \text{ MB})/(150 \text{ GB/s}) = 2 \text{ ms}$  for vector memory access +
- 400 ms for scalar execution +
- $(200 \text{ MB} + 100 \text{ MB})/(10 \text{ GB/s}) = 30 \text{ ms}$  for host I/O

Even if host I/O can be overlapped with GPU execution, the GPU will require 430 ms and therefore will achieve lower performance than the host.

4.11 a. 

```

for (i=0; i<32; i+=2) dot[i] = dot[i]+dot[i+1];
for (i=0; i<16; i+=4) dot[i] = dot[i]+dot[i+2];
for (i=0; i<8; i+=8) dot[i] = dot[i]+dot[i+4];
for (i=0; i<4; i+=16) dot[i] = dot[i]+dot[i+8];
for (i=0; i<2; i+=32) dot[i] = dot[i]+dot[i+16];
dot[0]=dot[0]+dot[32];

```

b. 

```

li          $VL, 4
addvv.s     $v0(0), $v0(4)
addvv.s     $v0(8), $v0(12)
addvv.s     $v0(16), $v0(20)
addvv.s     $v0(24), $v0(28)
addvv.s     $v0(32), $v0(36)
addvv.s     $v0(40), $v0(44)
addvv.s     $v0(48), $v0(52)
addvv.s     $v0(56), $v0(60)

```

- c. 

```
for (unsigned int s= blockDim.x/2;s>0;s/=2) {
    if (tid<s) sdata[tid]=sdata[tid]+sdata[tid+s];
    __syncthreads();
}
```
- 4.12 a. Reads 40 bytes and writes 4 bytes for every 8 FLOPs, thus 8/44 FLOPs/byte.
- b. This code performs indirect references through the Ca and Cb arrays, as they are indexed using the contents of the IDx array, which can only be performed at runtime. While this complicates SIMD implementation, it is still possible to perform type of indexing using gather-type load instructions. The inner-most loop (iterates on z) can be vectorized: the values for Ex, dH1, dH2, Ca, and Cb could be operated on as SIMD registers or vectors. Thus this code is amenable to SIMD and vector execution.
- c. Having an arithmetic intensity of 0.18, if the processor has a peak floating-point throughput  $> (30 \text{ GB/s}) \times (0.18 \text{ FLOPs/byte}) = 5.4 \text{ GFLOPs/s}$ , then this code is likely to be memory-bound, unless the working set fits well within the processor's cache.
- d. The single precision arithmetic intensity corresponding to the edge of the roof is  $85/4 = 21.25 \text{ FLOPs/byte}$ .
- 4.13 a.  $1.5 \text{ GHz} \times .80 \times .85 \times 0.70 \times 10 \text{ cores} \times 32/4 = 57.12 \text{ GFLOPs/s}$
- b. **Option 1:**  
 $1.5 \text{ GHz} \times .80 \times .85 \times .70 \times 10 \text{ cores} \times 32/2 = 114.24 \text{ GFLOPs/s}$  (speedup =  $114.24/57.12 = 2$ )
- Option 2:**  
 $1.5 \text{ GHz} \times .80 \times .85 \times .70 \times 15 \text{ cores} \times 32/4 = 85.68 \text{ GFLOPs/s}$  (speedup =  $85.68/57.12 = 1.5$ )
- Option 3:**  
 $1.5 \text{ GHz} \times .80 \times .95 \times .70 \times 10 \text{ cores} \times 32/4 = 63.84 \text{ GFLOPs/s}$  (speedup =  $63.84/57.12 = 1.11$ )
- Option 3 is best.
- 4.14 a. Using the GCD test, a dependency exists if GCD (2,4) must divide  $5 - 4$ . In this case, a loop-carried dependency does exist.
- b. Output dependencies  
 S1 and S3 cause through A[i]
- Anti-dependencies  
 S4 and S3 cause an anti-dependency through C[i]
- Re-written code
- ```
for (i=0;i<100;i++) {
    T[i] = A[i] * B[i]; /* S1 */
    B[i] = T[i] + c; /* S2 */
    A1[i] = C[i] * c; /* S3 */
    C1[i] = D[i] * A1[i]; /* S4 */
}
```

True dependencies

S4 and S3 through A[i]

S2 and S1 through T[i]

- c. There is an anti-dependence between iteration  $i$  and  $i+1$  for array B. This can be avoided by renaming the B array in S2.
- 4.15
- a. Branch divergence: causes SIMD lanes to be masked when threads follow different control paths
  - b. Covering memory latency: a sufficient number of active threads can hide memory latency and increase instruction issue rate
  - c. Coalesced off-chip memory references: memory accesses should be organized consecutively within SIMD thread groups
  - d. Use of on-chip memory: memory references with locality should take advantage of on-chip memory, references to on-chip memory within a SIMD thread group should be organized to avoid bank conflicts
- 4.16 This GPU has a peak throughput of  $1.5 \times 16 \times 16 = 384$  GFLOPS/s of single-precision throughput. However, assuming each single precision operation requires four-byte two operands and outputs one four-byte result, sustaining this throughput (assuming no temporal locality) would require  $12 \text{ bytes/FLOP} \times 384 \text{ GFLOPs/s} = 4.6 \text{ TB/s}$  of memory bandwidth. As such, this throughput is not sustainable, but can still be achieved in short bursts when using on-chip memory.
- 4.17 Reference code for programming exercise:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <cuda.h>

__global__ void life (unsigned char *d_board,int iterations) {
    int i,row,col,rows,cols;
    unsigned char state,neighbors;

    row = blockIdx.y * blockDim.y + threadIdx.y;
    col = blockIdx.x * blockDim.x + threadIdx.x;
    rows = gridDim.y * blockDim.y;
    cols = gridDim.x * blockDim.x;
    state = d_board[(row)*cols+(col)];
    for (i=0;i<iterations;i++) {
        neighbors=0;
        if (row!=0) {
            if (col!=0) if (d_board[(row-1)*cols+(col-1)]==1) neighbors++;
            if (d_board[(row-1)*cols+(col)]==1) neighbors++;
            if (col!=(cols-1)) if (d_board[(row-1)*cols+(col+1)]==1) neighbors++;
        }
        if (col!=0) if (d_board[(row)*cols+(col-1)]==1) neighbors++;
```

```

        if (col!=(cols-1)) if (d_board[(row)*cols+(col+1)]==1) neighbors++;
        if (row!=(rows-1)) {
            if (col!=0) if (d_board[(row+1)*cols+(col-1)]==1) neighbors++;
            if (d_board[(row+1)*cols+(col)]==1) neighbors++;
            if (col!=(cols-1)) if (d_board[(row+1)*cols+(col+1)]==1) neighbors++;
        }
        if (neighbors<2) state = 0;
        else if (neighbors==3) state = 1;
        else if (neighbors>3) state = 0;
        __syncthreads();
        d_board[(row)*cols+(col)]=state;
    }
}

int main () {
    dim3 gDim,bDim;
    unsigned char *h_board,*d_board;
    int i,iterations=100;

    bDim.y=16;
    bDim.x=32;
    bDim.z=1;

    gDim.y=16;
    gDim.x=8;
    gDim.z=1;

    h_board=(unsigned char *)malloc(sizeof(unsigned char)*4096*4096);
    cudaMalloc((void **)&d_board,sizeof(unsigned char)*4096*4096);

    srand(56);
    for (i=0;i<4096*4096;i++) h_board[i]=rand()%2;
    cudaMemcpy(d_board,h_board,sizeof(unsigned char)*4096*4096,cudaMemcpyHostToDevice);
    life <<<gDim,bDim>>> (d_board,iterations);
    cudaMemcpy(h_board,d_board,sizeof(unsigned char)*4096*4096,cudaMemcpyDeviceToHost);
    free(h_board);
    cudaFree(d_board);
}

```