4.1

[25] <4.1, 4.2> Assume the constants shown as follows.

Constants	Values
AA,AC,AG,AT	0,1,2,3
CA,CC,CG,CT	4,5,6,7
GA,GC,GG,GT	8,9,10,11
TA,TC,TG,TT	12,13,14,15
A,C,G,T	0,1,2,3

Write code for RISC-V and RV64V. Assume the starting addresses of `tiPL`, `tiPR`, `clL`, `clR`, and `clP` are in `RtiPL`, `RtiPR`, `RclL`, `RclR`, and `RclP`, respectively. Do not unroll the loop. To facilitate vector addition reductions, assume that we add the following instructions to RV64V:
Vector Summation Reduction Single Precision:

```
vsum Fd, Vs
```

This instruction performs a summation reduction on a vector register `Vs`, writing to the sum into scalar register `Fd`.

RISC-V

li t1, #0 # initialize k

loop: flw f0, 0 (RtipL) # load all values for first expression

flw f1, 0 (RclL)

fmul.s f16, f0, f1

flw f2, 4 (RtipL)

flw f3, 4 (RclL)

fmul.s f17, f2, f3

fadd.s f20, f16, f17

flw f4, 8 (RtipL)

flw f5, 8 (RclL)

fmul.s f18, f4, f5

fadd.s f20, f20, f18

flw f6, 12 (RtipL)

flw f7, 12 (RclL)

fmul.s f19, f6, f7

fadd.s f20, f20, f19

flw f8, 0 (RtipR)

flw f9, 0 (RclR)

fmul.s f16, f8, f9

flw f10, 4 (RtipR)

flw f11, 4 (RclR)

fmul.s f17, f10, f11

fadd.s f21, f16, f17

flw f12, 8 (RtipR)

flw f13, 8 (RclR)

fmul.s f18, f12, f13

fadd.s f21, f21, f18

flw f14, 12 (RtipR)

flw f15, 12 (RclR)

fmul.s f19, f14, f15

fadd.s f21, f21, f19

fmul.s f20, f20, f21 # final multiply

fsw f20, 0 (RclP) # store result

add RclP, RclP, 4 # increment clP for next expression

addi t1, t1, 1

and x2, x2, #3

bneq x2, skip

skip: blt t1, x3, loop # assume r3 = seq_length * 4

RV64

li t1, 0

vcfgd 10 * FP32 # 开出是个向量

loop:

vld v0, 0 (RclL)

vld v1, 0 (RclR)

vld v2, 0 (RtipL) # 装载tipL values

```
vmul v2, v2, v0
vsum f0, v2
vld v3, 16 (RtipL)
vmul v3, v3, v0
vsum f1, v3
vld v4, 32 (RtipL)
vmul v4, v4, v0
vsum f2, v4
vld v5, 48 (RtipL)
vmul v5, v5, v0
vsum f3, v5
vld v6, 0 (RtipR) # 装载 tipR values
vmul v6, v6, v1
vsum f4, v6
fmul.s f0, f0, f4
vld v7, 16 (RtipR)
vmul v7, v7, v1
vsum f5, v7
fmul.s f1, f1, f5
vld v8, 32 (RtipR)
vmul v8, v8, v1
vld v9, 48 (RtipR)
vsum f6, v8
fmul.s f2, f2, f6
vmul v9, v9, v1
vsum f7, v9
fmul.s f3, f3, f7
fsw f0, 0 (Rclp)
fsw f1, 4 (Rclp)
fsw f2, 8 (Rclp)
fsw f3, 12 (Rclp)
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 t1, t1, 1

blt t1, x3, loop # assume x3 = seq_length

4.3

4.3 [25] <4.1, 4.2> Assume that the vector reduction instruction is executed on the vector functional unit, similar to a vector add instruction. Show how the code sequence lays out in convoys assuming a single instance of each vector functional unit. How many chimes will the code require? How many cycles per FLOP are needed, ignoring vector instruction issue overhead?

并行规约通常用于处理大输入数据集，***将一组输入值规约一个值。***

数据特点：

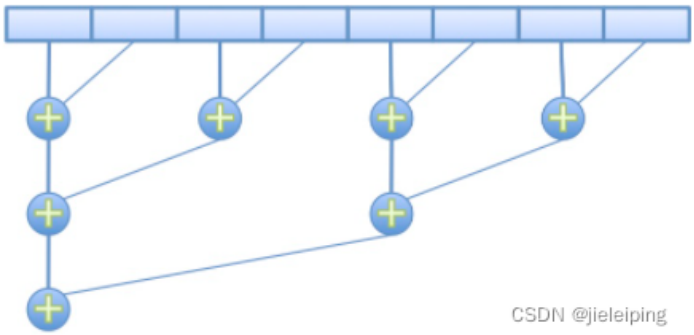
（1）对于数据集中的元素没有顺序要求。

（2）可将数据分为若干小集合，每个线程处理一个集合。

操作可以是：求最大值（Max）、求最小值（Min）、求和（Sum）、求乘（Product）。

并行规约求和

规约求和是常见应用，将输入数据求和得到一个值。如下面简单例子所示：



这里以上图为例，假设要计算该向量的规约

vld A

vld B

vld vadd

vld vadd

vld vadd

vld vadd

vld vadd

vld vadd

vld vadd

vsum

vsum

vsum

vsum

vsum

vsum

vsum

vsum

18 chimes

4results

15 FLOPs per result

18/15 = 1.2 cycles per Flop

4.5

4.5 [25] <4.4> Now assume we want to implement the MrBayes kernel on a GPU using a single thread block. Rewrite the C code of the kernel using CUDA. Assume that pointers to the conditional likelihood and transition probability tables are specified as parameters to the kernel. Invoke one thread for each iteration of the loop. Load any reused values into shared memory before performing operations on it.

```
__global__ void compute_condLike(float *clL, float *clR, float *clP, float
*tiPL, float *tiPR) {
    int k = threadIdx.x;

    // 共享部分
    __shared__ float clL_s[4];
    __shared__ float clR_s[4];

    // 初始化加载
    for (int i = 0; i < 4; i++) {
        clL_s[i] = clL[k * 4 + i];
        clR_s[i] = clR[k * 4 + i];
    }

    // 加权求和并写回数组
    clP[k * 4] = (tiPL[k + AA] * clL_s[A] +
                  tiPL[k + AC] * clL_s[C] +
                  tiPL[k + AG] * clL_s[G] +
                  tiPL[k + AT] * clL_s[T]) *
                 (tiPR[k + AA] * clR_s[A] +
                  tiPR[k + AC] * clR_s[C] +
                  tiPR[k + AG] * clR_s[G] +
                  tiPR[k + AT] * clR_s[T]);
```

```
    clP[k * 4 + 1] = (tiPL[k + CA] * clL_s[A] +
                      tiPL[k + CC] * clL_s[C] +
                      tiPL[k + CG] * clL_s[G] +
                      tiPL[k + CT] * clL_s[T]) *
                     (tiPR[k + CA] * clR_s[A] +
                      tiPR[k + CC] * clR_s[C] +
                      tiPR[k + CG] * clR_s[G] +
                      tiPR[k + CT] * clR_s[T]);
}

int main() {
    //
    float *clL_host, *clR_host, *clP_host, *tiPL_host, *tiPR_host;

    // 分配内存
    float *clL_device, *clR_device, *clP_device, *tiPL_device, *tiPR_device;
    cudaMalloc((void**)&clL_device, sizeof(float) * /*size*/);
    cudaMalloc((void**)&clR_device, sizeof(float) * /*size*/);
    cudaMalloc((void**)&clP_device, sizeof(float) * /*size*/);
    cudaMalloc((void**)&tiPL_device, sizeof(float) * /*size*/);
    cudaMalloc((void**)&tiPR_device, sizeof(float) * /*size*/);

    // 拷贝数据
    cudaMemcpy(clL_device, clL_host, sizeof(float) * /*size*/,
cudaMemcpyHostToDevice);
    cudaMemcpy(clR_device, clR_host, sizeof(float) * /*size*/,
cudaMemcpyHostToDevice);
    cudaMemcpy(tiPL_device, tiPL_host, sizeof(float) * /*size*/,
cudaMemcpyHostToDevice);
    cudaMemcpy(tiPR_device, tiPR_host, sizeof(float) * /*size*/,
cudaMemcpyHostToDevice);

    // 每一次迭代启动一个线程
    int numIterations = /* number of iterations */;
    compute_condLike<<<1, numIterations>>>(clL_device, clR_device, clP_device,
tiPL_device, tiPR_device);

    // Copy the result from device to host
    cudaMemcpy(clP_host, clP_device, sizeof(float) * /*size*/,
cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(clL_device);
    cudaFree(clR_device);
    cudaFree(clP_device);
    cudaFree(tiPL_device);
    cudaFree(tiPR_device);


    return 0;
}
```

4.7

4.7 [15] <4.4> Convert your code from Exercise 4.6 into PTX code. How many instructions are needed for the kernel?

```
# compute address of clL
mul.u64 %r1, %ctaid.x, 4000 # multiply block index by 4000
mul.u64 %r2, %tid.x, 4 # multiply thread index by 4
add.u64 %r1, %r1, %r2 # add products
ld.param.u64 %r2, [clL] # load base address of clL
add.u64 %r1, %r2, %r2 # add base to offset
# compute address of clR
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, [clR] # load base address of clR
add.u64 %r2, %r2, %r3 # add base to offset
ld.global.f32 %f1, [%r1+0] # move clL and clR into shared memory
st.shared.f32 [clL_s+0], %f1 # (unroll the loop)
ld.global.f32 %f1, [%r2+0]
st.shared.f32 [clR_s+0], %f1
ld.global.f32 %f1, [%r1+4]
st.shared.f32 [clL_s+4], %f1
ld.global.f32 %f1, [%r2+4]
st.shared.f32 [clR_s+4], %f1
ld.global.f32 %f1, [%r1+8]
st.shared.f32 [clL_s+8], %f1
ld.global.f32 %f1, [%r2+8]
st.shared.f32 [clR_s+8], %f1
ld.global.f32 %f1, [%r1+12]
st.shared.f32 [clL_s+12], %f1
ld.global.f32 %f1, [%r2+12]
st.shared.f32 [clR_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
# compute address of clP:
mul.u64 %r3, %r3, 24,000 # multiply block index by 4000
mul.u64 %r4, %tid.x, 16 # multiply thread index by 4 floats
add.u64 %r3, %r3, %r4 # add products
ld.param.u64 %r4, [tipR] # load base address of tipL
add.u64 %r3, %r3, %r4 # add base to offset
ld.global.f32 %f1, [%r1] # load tiPL[0]
ld.global.f32 %f2, [%r1+4] # load tiPL[1]
```

```
…
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, [clL_s] # load clL
ld.shared.f32 %f34, [clL_s+4]
ld.shared.f32 %f35, [clL_s+8]
ld.shared.f32 %f36, [clL_s+12]
ld.shared.f32 %f37, [clR_s] # load clR
ld.shared.f32 %f38, [clR_s+4]
ld.shared.f32 %f39, [clR_s+8]
ld.shared.f32 %f40, [clR_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, %11, %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
```

4.11 (a) (b)

4.11 [15/25/25] <4.4, 4.5> Section 4.5 discussed the reduction operation that reduces a vector down to a scalar by repeated application of an operation. A reduction is a special type of a loop recurrence. An example is shown as follows:

```
dot=0.0;
for (i=0;i <64;i++) dot = dot + a[i] * b[i];
```

A vectorizing compiler might apply a transformation called *scalar expansion*, which expands dot into a vector and splits the loop such that the multiply can be performed with a vector operation, leaving the reduction as a separate scalar operation:

```
for (i=0;i <64;i++) dot[i] = a[i] * b[i];
for (i=1;i <64;i++) dot[0] = dot[0] + dot[i];
```

As mentioned in Section 4.5, if we allow the floating-point addition to be associative, there are several techniques available for parallelizing the reduction.

a. [15] <4.4, 4.5> One technique is called recurrence doubling, which adds sequences of progressively shorter vectors (ie, two 32-element vectors, then two 16-element vectors, and so on). Show how the C code would look for executing the second loop in this way.

b. [25] <4.4, 4.5> In some vector processors, the individual elements within the vector registers are addressable. In this case, the operands to a vector operation may be two different parts of the same vector register. This allows another solution for the reduction called *partial sums*. The idea is to reduce the vector to m sums where m is

the total latency through the vector functional unit, including the operand read and write times. Assume that the VMIPS vector registers are addressable (e.g., you can initiate a vector operation with the operand `V1(16)`, indicating that the input operand begins with element 16). Also, assume that the total latency for adds, including the operand read and result write, is eight cycles. Write a VMIPS code sequence that reduces the contents of `V1` to eight partial sums.

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)

```
vadd v0(0), v0(4)

vadd v0(8), v0(12)

vadd v0(16), v0(20)

vadd v0(24), v0(28)

vadd v0(32), v0(36)

vadd v0(40), v0(44)

vadd v0(48), v0(52)

vadd v0(56), v0(60)
```

4.14

4.14 [10/15/15] <4.5> In this exercise, we will examine several loops and analyze their potential for parallelization.

a. [10] <4.5> Does the following loop have a loop-carried dependency?

```
for (i=0;i <100;i++) {
 A[i] = B[2*i+4];
 B[4*i+5] = A[i];
}
```

b. [15] <4.5> In the following loop, find all the true dependences, output dependences, and antidependences. Eliminate the output dependences and antidependences by renaming.

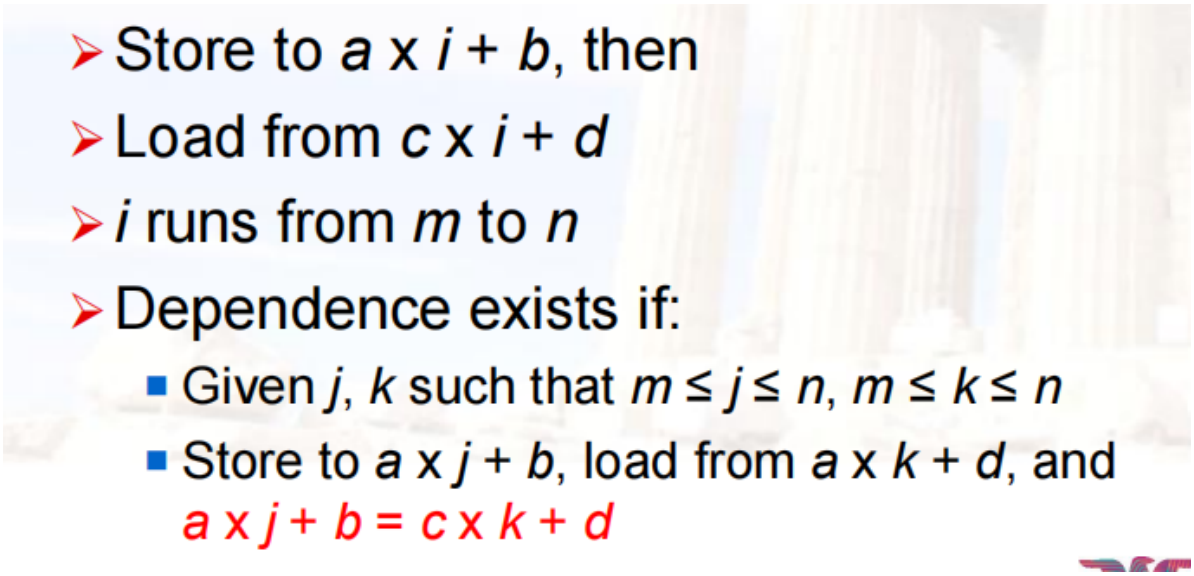
```
for (i=0;i <100;i++) {
A[i] = A[i] * B[i]; /* S1 */
```

```
B[i] = A[i] + c; /* S2 */
A[i] = C[i] * c; /* S3 */
C[i] = D[i] * A[i]; /* S4 */
```

c. [15] <4.5> Consider the following loop:

```
for (i=0;i <100;i++) {
 A[i] = A[i] + B[i]; /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

Are there dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.



a)

根据公式

load 2 i+ 4　a = 2 b =4

store 4 i+ 5 c = 4 d = 5

gcd(c,a) =,d-b = 1, 由于4整除于1 所以依赖存在

b)

S2 S4 其中的Ai真依赖于当前段S1

S1的Bi真依赖于上一段的S2

S1 S3 有WAW依赖

S4 S3在Ai上存在真依赖

通过类似renamed的方式来消除依赖

```
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 */}
```

现在只有真依赖 S2 真依赖S1

S4真依赖S3

c)

由于S1使用S2在前一次迭代中计算的值，但是依赖关系没有形成loop，所以可以改造

A[0] = A[0] + B[0];

for (i=0; i<99; i=i+1) {

B[i+1] = C[i] + D[i];

A[i+1] = A[i+1] + B[i+1];

}

B[100] = C[99] + D[99];

这样就可以并行了