CSCI 510: Computer Architecture
Written Assignment 3

Due November 10, 2016 23:59:59PM

1. **(5 pts)** Assume a hypothetical GPU with the following characteristics:
   - Clock rate 1.5 GHz
   - Contains 16 SIMD processors, each containing 32 single-precision floating point units
   - Has 100 GB/sec off-chip memory bandwidth

   Without considering memory bandwidth, what is the peak single-precision floating-point throughput for this GPU in GLFOP/sec, assuming that all memory latencies can be hidden? Is this throughput sustainable given the memory bandwidth limitation?

Answer:

The peak single-precision floating-point throughput is

$$1.5 \times 16 \times 32 = 768 \text{ GFLOPS/s}$$

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 memory bandwidth of

$$12 \text{ bytes/FLOP} \times 768 \text{ GFLOPs/s} = 9.216 \text{ TB/s}$$

Since 9.216 TB/s >> 100 GB/s, this throughput is not sustainable, but can still be achieved in short bursts when using on-chip cache.

2. The following code multiplies two vectors that contain single-precision complex values:

```
for (i = 0; i < 400; ++ i) {
  u[i] = a[i] * b[i] – c[i] * d[i];
  v[i] = a[i] * d[i] + c[i] * b[i];
}
```

   Assume that the processor runs at 1GHz and has a maximum vector length of 64. The load/store unit has a start-up overhead of 15 cycles; the multiply unit, 8 cycles; and the add/subtract unit, 5 cycles.

   a. **(5 pts)** What is the arithmetic intensity of this piece of code? Justify your answer.
   b. **(10 pts)** Convert this loop into VMIPS assembly code using strip mining.
   c. **(15 pts)** Assuming chaining and a single memory pipeline, how many chimes are required? How many clock cycles are required per result value, including start-up overhead?
   d. **(15 pts)** Now assume chaining and three memory pipelines. If there are no bank conflicts in the loop's accesses, how many clock cycles are required per result, including start-up overhead?

Answer:

a. Every single-precision number takes 4 bytes. In each iteration, the code reads 4 × 4 = 16 bytes from the main memory (the second access of a[i], b[i], c[i] and d[i] is from the cache) and writes 2 × 4 = 8 bytes to the main memory, in total 24 bytes. In each iteration, six floating point operations are executed. Therefore, the arithmetic intensity is 6 / 24 = 1/4 = 0.25.

b.

```
      li $VL,16              # perform the first 16 ops
      li $r1,0               # initialize index
loop: lv $v1,a+$r1           # load a
      lv $v3,b+$r1           # load b
      mulvv.s $v5,$v1,$v3    # a*b
      lv $v2,c+$r1           # load c
      lv $v4,d+$r1           # load d
      mulvv.s $v6,$v2,$v4    # c*d
      subvv.s $v5,$v5,$v6    # a*b - c*d
      sv $v5,u+$r1           # store u
      mulvv.s $v5,$v1,$v4    # a*d
      mulvv.s $v6,$v2,$v3    # c*b
      addvv.s $v5,$v5,$v6    # a*d + c*b
      sv $v5,v+$r1           # store v
      bnez $r1,else          # check if first iteration
      li $VL, 64             # perform 64 ops for every iteration
      addi $r1,$r1,#64       # first iteration, increment by 16*4=64
      j loop                 # guaranteed next iteration
else: addi $r1,$r1,#256      # not first iteration,
                             # increment by 64*4 = 256
skip: blt $r1,1600,loop      # next iteration?
```

c.

1) lv # load a
2) lv # load b
3) mulvv.s lv # a * b, load c
4) lv mulvv.s # load d, c * d
5) subvv.s sv # a*b – c*d, store u
6) mulvv.s # a * d
7) mulvv.s # c * b
8) addvv.s sv # a * d + c * b, store v

It takes $\lceil 400 / 64 \rceil \times 6 = 7 \times 6 = 42$ chimes.

In the first iteration,
The first two chimes take 2 × (15 + 16) = 62 cycles.
The next six chime takes (8 + 15 + 16) × 2 + (5 + 15 + 16) + (8 + 16) × 2 + (5 + 15 + 16) = 196 cycles.
In other iterations,
The first two chimes take 2 × (15 + 64) = 158 cycles.

The next six chime takes (8 + 15 + 64) × 2 + (5 + 15 + 64) + (8 + 64) × 2 + (5 + 15 + 64) = 486 cycles.

The total number of cycles taken is 62 + 196 + $\lfloor 400/64 \rfloor$ × 486 = 3174 cycles.
The number of cycles per result = 3174 / 400 = 7.935 cycles.

d.

- Part 1: For the first iteration:

1) lv, lv, mulvv.s lv # load a and b, a * b, load c
2) lv mulvv.s # load d, c * d
3) subvv.s sv # a*b – c*d, store u
4) mulvv.s # a * d
5) mulvv.s # c * b

- Part 2: For all but the last iteration:

1) addvv.s sv, lv, lv # a * d + c * b, store v, load a and b

- Part 3: For all but the first iteration:

2) mulvv.s lv, lv # a * b, load c and d
3) mulvv.s # c * d
4) subvv.s sv # a*b – c*d, store u
5) mulvv.s # a * d
6) mulvv.s # c * b

- Part 4: For the last iteration:

7) addvv.s sv # a * d + c * b, store v

It takes $\lceil 400 / 64 \rceil$ × 6 = 7 × 6 = 42 chimes.

In part 1,
The first chime takes 15 + 8 + 15 + 16 = 54 cycles.
The second chime takes 15 + 8 + 16 = 39 cycles.
The third chime takes 8 + 15 + 16 = 39 cycles.
The fourth and fifth chime take 2 × (8 + 16) = 48 cycles
In part 2,
The chime takes 8 + 15 + 64 = 87 cycles.
In part 3,
The first chime takes 8 + 15 + 64 = 87 cycles.
The second chime takes 8 + 64 = 72 cycles
The third chime takes 8 + 15 + 64 = 87 cycles.
The fourth and fifth chime take 2 × (8 + 64) = 144 cycles
In part 4,
The chime takes 8 + 15 + 64 = 87 cycles.

$\lfloor 400/64 \rfloor$ = 7

The total number of cycles taken is 54 + 39 + 39 + 48 + 87 * 6 + (87 + 72 + 87 + 144) * 6 + 87 = 3129 cycles.
The number of cycles per result = 3129 / 400 = 7.8225 cycles.

3. **(7 pts)** Convert the loop in Problem 2 into MIPS SIMD. (Refer to the example on page 284 ~ 285 on the textbook.)

Answer:

```
        LA     R1, a          ; load base address of a
        LA     R2, b          ; load base address of b
        LA     R3, c          ; load base address of c
        LA     R4, d          ; load base address of d
        LA     R5, u          ; load base address of u
        LA     R6, v          ; load base address of v
        DADDIU R7, R1, #1600
LOOP:   L.4S   F0, 0(R1)      ; load a[i]..a[i+3]
        L.4S   F4, 0(R2)      ; load b[i]..b[i+3]
        MUL.4S F16, F4, F0    ; a * b
        L.4S   F8, 0(R3)      ; load c[i]..c[i+3]
        L.4S   F12, 0(R4)     ; load d[i]..d[i+3]
        MUL.4S F20, F12, F8   ; c * d
        SUB.4S F20, F20, F16  ; a * b - c * d
        S.4S   F4, 0(R5)      ; store u[i]..u[i+3]
        MUL.4S F16, F12, F0   ; a * d
        MUL.4S F20, F8, F4    ; c * b
        ADD.4S F20, F20, F16  ; a * d + c * b
        S.4S   F20, 0(R6)     ; store v[i]..v[i+3]
        DADDIU R1, R1, #16
        DADDIU R2, R2, #16
        DADDIU R3, R3, #16
        DADDIU R4, R4, #16
        DADDIU R5, R5, #16
        DADDIU R6, R6, #16
        DSUBU  R8, R7, R1
        BNEZ   R8, LOOP
```

4. **(7 pts)** Convert the loop in Problem 2 into CUDA.

Answer:

```
// Invoke remi() with 64 threads per Thread Block
__host__
int nblocks = (400 + 63) / 64;
remi<<<nblocks, 64>>>(400, a, b, u, c, d, v);
// remi in CUDA
__device__
void remi(int n, double *a, double *b, double *u,
          double *c, double *d, double *v)
```

```
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) {
    u[i] = a[i] * b[i] – c[i] * d[i];
    v[i] = a[i] * d[i] + c[i] * b[i];
  }
}
```

5. A reduction is a special type of a loop recurrence. An example is shown below:

```
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 = 0; 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. **(7 pts)** One technique is called recurrence doubling, which adds sequences of progressively shorter vectors (i.e., 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. **(7 pts)** 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.
c. **(7 pts)** When performing a reduction on a GPU, one thread is associated with each element in the input vector. The first step is for each thread to write its corresponding value into shared memory. Next, each thread enters a loop that adds each pair of input values. This reduces the number of elements by half after each iteration, meaning that the number of active threads also reduces by half after each iteration. In order to maximize the performance of the reduction, the number of fully populated warps should be maximized throughout the course of the loop. In other words, the active threads should be contiguous. Also, each thread should index the shared

array in such a way as to avoid bank conflicts in the shared memory. The following loop violates only the first of these guidelines and also uses the modulo operator which is very expensive for GPUs:

```
unsigned int tid = threadIdx.x;
for (unsigned int s=1; s < blockDim.x; s *= 2) {
  if ((tid % (2*s)) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

Rewrite the loop to meet these guidelines and eliminate the use of the modulo operator. Assume that there are 32 threads per warp and a bank conflict occurs whenever two or more threads from the same warp reference an index whose modulo by 32 are equal.

Answer:

a.

```
for (k = 32; k > 0; k /= 2) {
  for (i = 0; i < k; ++ i) {
    dot[i] = dot[i] + dot[i + k];
  }
}
```

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();
}
```

6. In this exercise, we will examine several loops and analyze their potential for parallelization.

a. **(5 pts)** 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. **(8 pts)** 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. **(5 pts)** 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.

Answer:

a.

To check whether or not loop-carried dependency exists between B[2*i+4] and B[4*i+5], we use the GCD test. A dependency exists if GCD (2,4) divides 5 – 4. As GCD(2,4) = 2, and (5 – 4) mod 2 = 1, i.e. GCD(2,4) does not divide 5 – 4. Therefore, no loop-carried dependency exists for B[].

The same index for A[] is used in the loop body. Therefore, no loop-carried dependency exists for A[] either.

b.

True dependencies:

S1 and S2 through A[i]

S3 and S4 through A[i]

Output dependencies:

S1 and S3 through A[i]

Anti-dependencies

S1 and S2 through B[i]

S2 and S3 through A[i]

S3 and S4 through C[i]

Re-written code:

```
/* Assume A1,B1,C1 are copies of A,B,C */
for (i=0;i<100;i++) {
  X[i] = A1[i] * B1[i]; /* S1 */
  B[i] = X[i] + c; /* S2 */
  A[i] = C1[i] * c; /* S3 */
  C[i] = D[i] * A[i]; /* S4 */
}
```

c.

There is an anti-dependence between iteration i and i+1 for array B

Re-written code:

```
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];
```