ECE 586
Practice Problems

Chapter 4:

4.9 Consider the following code, which multiplies two vectors that contain single precision complex values.

```
for (i=0; i < 300; i++) {
        c_re[i] = a_re[i] *  b_re[i] –  a_im[i] * b_im[i];
        c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
}
```

Assume that processor runs at 700MHz 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/subract unit, 5 cycles.

a. What is the arithmetic intensity of the kernel?
b. Convert this loop into RV64 assembly code using strip mining.
c. Assuming chaining and a single memory pipeline, how many chimes are required? How many clock cycles are required per complex result value, including start-up overhead?
d. If the vector sequence is chained, how many clock cycles are required per complex result value, include overhead?
e. Now assume the processor has three memory pipelines and chaining. If there are no bank conflicts in the loop's accesses, how many clock cycles are required per result?

4.12 The following kernel performs a portion of the finite difference time-domain (FDTF) method for computing Maxwell's equations in a three dimensional space, part of one of the SPEC06fp benchmarks.

```
for (int x=0; x<NX-1; x++) {
        for(int y=0; y<NY-1; y++) {
                for (int z=0; z<NZ-1; z++) {
                        int index = x*NY*NZ + y*NZ +z;
                        if ( y>0 && x>0 ) {
                                material = IDx[index];
                                dH1 = (Hz[index] – Hz[index-incrementY]) / dy[y];
                                dH2 = (Hy[index] – Hy[index-incrementZ]) / dy[z];
                                Ex[index]=Ca[material]*Ex[index]+Cb[material]*(dH2-dH1);
}}}}
```

Assume that dH1, dH2, Hy, Hz, dy, dz, Ca, Cb and Ex are all single precision floating point arrays.Assume IDx is an array of unsigned int.

a. What is the arithmetic intensity of this kernel?
b. Is this kernel amenable to vector or SIMD execution? Why or Why not?
c. Assume this kernel is to be executed on a processor that has 30 GB/s of memory bandwidth. Will this kernel be memory bound or compute bound?
d. Develop a roofline model for this processor, assuming it has a peak computational throughput of 85 GFLOP/s.

4.13 Assume a GPU architecture that contains 10 SIMD processors. Each SIMD instruction has a width of 32 and each SIMD processor contains 8 lanes for single precision arithmetic and load/store instructions, meaning that each non-diverged SIMD instruction can produce 32 results every 4 cycles. Assume a kernel that has divergent branches that causes, on average, 80% of threads to be active. Assume that 70% of all SIMD instructions executed are single precision arithmetic and 20% are load/store. Because not all memory latencies are covered, assume an average SIMD instruction issue rate of 0.85. Assume that GPU has a clock speed of 1.5GHz.

a. Compute throughput, in GFLOP/s for this kernel on this GPU.
b. Assume that you have the following choices:
    i. Increasing number of single precision lanes to 16
    ii. Increasing number of SIMD processors to 15 (assume this change doesn't affect any other performance metrics and that the code scales to the additional processors)
    iii. Adding a cache that will effectively reduce memory latency by 40%, which will increase the instruction issue rate to 0.95.

What is the speed-up in throughput for each of these improvements?

4.16. Assume a hypothetical GPU with the following characteristics:

- Clock rate 1.5GHz.
- Contains 16 SIMD processors, each containing 16 single precision floating point units.
- Has 100GB/s off-chip memory bandwidth.

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

ECE 586
Practice Problems

Chapter 4:

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 chimes × 64 elements + 15 cycles (load/store) × 6 + 8 cycles (multiply) × 4 + 5 cycles (add/subtract) × 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.

4.12 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 innermost 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 throughout > (30 GB/s) × (0.18 FLOPs/byte) = 5.4 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 FLOPs/byte.

4.13 a. 1.5 GHz × .80 × .85 × 0.70 × 10 cores × 32/4 = 57.12 GFLOPs/s

b. **Option 1:**

1.5 GHz × .80 × .85 × .70 × 10 cores × 32/2 = 114.24 GFLOPs/s (speedup = 114.24/57.12 = 2)

Option 2:

1.5 GHz × .80 × .85 × .70 × 15 cores × 32/4 = 85.68 GFLOPs/s (speedup = 85.68/57.12 = 1.5)

Option 3:

1.5 GHz × .80 × .95 × .70 × 10 cores × 32/4 = 63.84 GFLOPs/s (speedup = 63.84/57.12 = 1.11)

Option 3 is best.

4.16 This GPU has a peak throughput of 1.5 × 16 × 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 bytes/FLOP × 384 GFLOPs/s = 4.6 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.

ECE 586
Chapter 5 Practice Problems

A multicore SMT multiprocessor is illustrated in Figure 5.37. Only the cache contents are shown. Each core has a single, private cache with coherence maintained using the snooping coherence protocol of Figure 5.7. Each cache is direct-mapped, with four lines, each holding 2 bytes (to simplify diagram). For further simplification, the whole line addresses in memory are shown in the address fields in the caches, where the tag would normally exist. The coherence states are denoted M, S, and I for Modified, Shared, and Invalid.

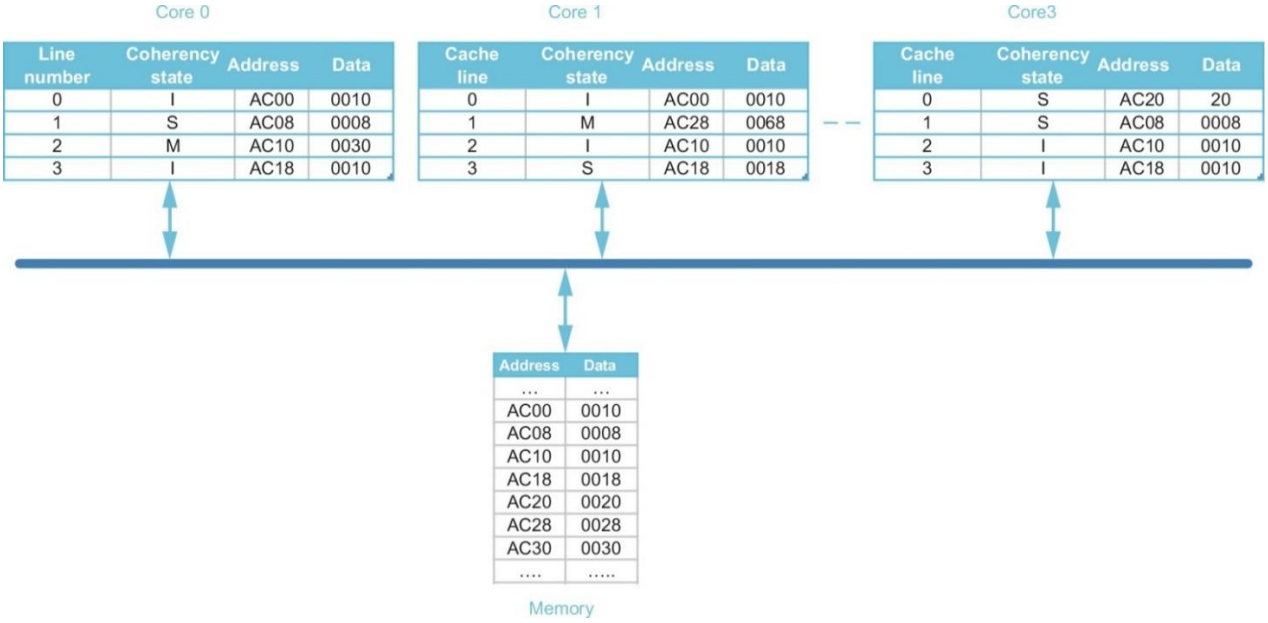

Figure 5.37 Multicore (point-to-point) multiprocessor.

5.1. For each part of this exercise, the initial cache and memory state are assumed to initially have the contents shown in Figure 5.37. Each part of this exercise specifies a sequence of one or more CPU operations of the form

Ccore#: R, < address > for reads
and
Ccore#: W, < address > ← < value written > for writes.

For example,
C3: R, AC10 & C0: W, AC18 ← 0018

Read and write operations are for 1 byte at a time. Show the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the actions given below. Show only the cache lines that experience some state change; for example:
C0.L0: (I, AC20, 0001) indicates that line 0 in core 0 assumes an “invalid” coherence state (I), stores AC20

from the memory, and has data contents 0001. Furthermore, represent any changes to the memory state as M: < address > <-- value.

Different parts (a) through (g) do not depend on one another: assume the actions in all parts are applied to the initial cache and memory states.

a. C0: R, AC20

b. C0: W, AC20 <-- 80

c. C3: W, AC20 <-- 80

d. C1: R, AC10

e. C0: W, AC08 <-- 48

f. C0: W, AC30 <-- 78

g. C3: W, AC30 <-- 78

5.2. The performance of a snooping cache-coherent multiprocessor depends on many detailed implementation issues that determine how quickly a cache responds with data in an exclusive or M state block. In some implementations, a processor read miss to a cache block that is exclusive in another processor's cache is faster than a miss to a block in memory. This is because caches are smaller, and thus faster, than main memory. Conversely, in some implementations, misses satisfied by memory are faster than those satisfied by caches. This is because caches are generally optimized for “front side” or CPU references, rather than “back side” or snooping accesses. For the multiprocessor illustrated in Figure 5.37, consider the execution of a sequence of operations on a single processor core where

- read and write hits generate no stall cycles;
- read and write misses generate Nmemory and Ncache stall cycles if satisfied by memory and cache, respectively;
- write hits that generate an invalidate incur Ninvalidate stall cycles; and
- a write-back of a block, either due to a conflict or another processor's request to an exclusive block, incurs an additional Nwriteback stall cycles.

Consider two implementations with different performance characteristics summarized in Figure 5.38.

Parameter	Implementation 1 Cycles	Implementation 2 Cycles
N_{memory}	100	100
N_{cache}	40	130
$N_{invalidate}$	15	15
$N_{writeback}$	10	10

Figure 5.38 Snooping coherence latencies.

To observe how these cycle values are used, we illustrate how the following sequence of operations, assuming the initial caches' states in Figure 5.37, behave under implementation 1.

C1: R, AC10
C3: R, AC10

For simplicity, assume that the second operation begins after the first completes, even though they are on different processor cores.
For Implementation 1,

- the first read generates 50 stall cycles because the read is satisfied by C0's cache: C1 stalls for 40 cycles while it waits for the block, and C0 stalls for 10 cycles while it writes the block back to memory in response to C1's request; and
- the second read by C3 generates 100 stall cycles because its miss is satisfied by memory.

Therefore, this sequence generates a total of 150 stall cycles.
For the following sequences of operations, how many stall cycles are generated by each implementation?

a. C0: R, AC20
C0: R, AC28
C0: R, AC30
b. C0: R, AC00
C0: W, AC08 <-- 48
C0: W, AC30 <-- 78
c. C1: R, AC20
C1: R, AC28
C1: R, AC30
d. C1: R, AC00
C1: W, AC08 <-- 48
C1: W, AC30 <-- 78

5.3. Some applications read a large dataset first and then modify most or all of it. The base MSI coherence protocol will first fetch all of the cache blocks in the Shared state and then be forced to perform an invalidate operation to upgrade them to the Modified state. The additional delay has a significant impact on some workloads. The MESI addition to the standard protocol (see Section 5.2) provides some relief in these cases. Draw new protocol diagrams for a MESI protocol that adds the Exclusive state and transitions to the base MSI protocol's Modified, Shared, and Invalidate states.

5.7. For the following code sequences and the timing parameters for the two implementations in Figure 5.38, compute the total stall cycles for the base MSI protocol and the optimized MESI protocol in Exercise 5.3. Assume state transitions that do not require bus transactions incur no additional stall cycles.

a. C1: R, AC10
C3: R, AC10
C0: R, AC10

b. C1: R, AC20
C3: R, AC20
C0: R, AC20

c. C0: W, AC20 <-- 80
C3: R, AC20
C0: R, AC20

d. C0: W, AC08 <--88
C3: R, AC08
C0: W, AC08 <-- 98

ECE 586
Chapter 5 Practice Problems

5.1.

Cx.y is cache line y in core x.

a.	C0: R AC20	C0.0: (S, AC20, 0020), returns 0020
b.	C0: W AC20 <-80	C0.0: (M, AC20, 0080) C3.0: (I, AC20, 0020)
c.	C3: W AC20<-80	C3.0: (M, AC20, 0080)
d.	C1: R AC10	C1.2: (S, AC10, 0030) C0.2: (S, AC10, 0030) M: AC10 0030 write back
e.	C0: W AC08 <- 48	C0.1 (M, AC08, 0048 C3.1: (I, AC08, 0008)
f.	C0: W AC30 <-78	C0.2: (M, AC30, 0078) M: AC10 0030 (write-back to memory)
g.	C3: W AC30 <-78	C3.2 :(M, AC30, 0078)

5.2.

a. C0: R AC20 Read miss, satisfied by memory
   C0: R AC28 Read miss, satisfied by C1's cache
   C0: R AC30 Read miss, satisfied by memory, write-back 110
   Implementation 1: 100 + 40+ 10 + 100 + 10 = 260 stall cycles
   Implementation 2: 100 + 130 + 10 + 100 + 10 = 350 stall cycles

b. C0: R AC00 Read miss, satisfied by memory
   C0: W AC08 48 Write hit, sends invalidate
   C0: W A30 78 Write miss, satisfied by memory, write back 110
   Implementation 1: 100 + 15 + 10 + 100 = 225 stall cycles
   Implementation 2: 100 + 15 + 10 + 100 = 225 stall cycles

c. C1: R AC20 Read miss, satisfied by memory
   C1: R AC28 Read hit
   C1: R AC30 Read miss, satisfied by memory
   Implementation 1: 100 + 0 + 100 = 200 stall cycles
   Implementation 2: 100 + 0 + 100 = 200 stall cycles

d. d. C1: R AC00 Read miss, satisfied by memory
   C1: W AC08 48 Write miss, satisfied by memory, write back AC28
   C1: W AC30 78 Write miss, satisfied by memory
   Implementation 1: 100 + 100 + 10 + 100 = 310 stall cycles
   Implementation 2: 100 + 100 + 10 + 100 = 310 stall cycles

5.3 See Figure S.1

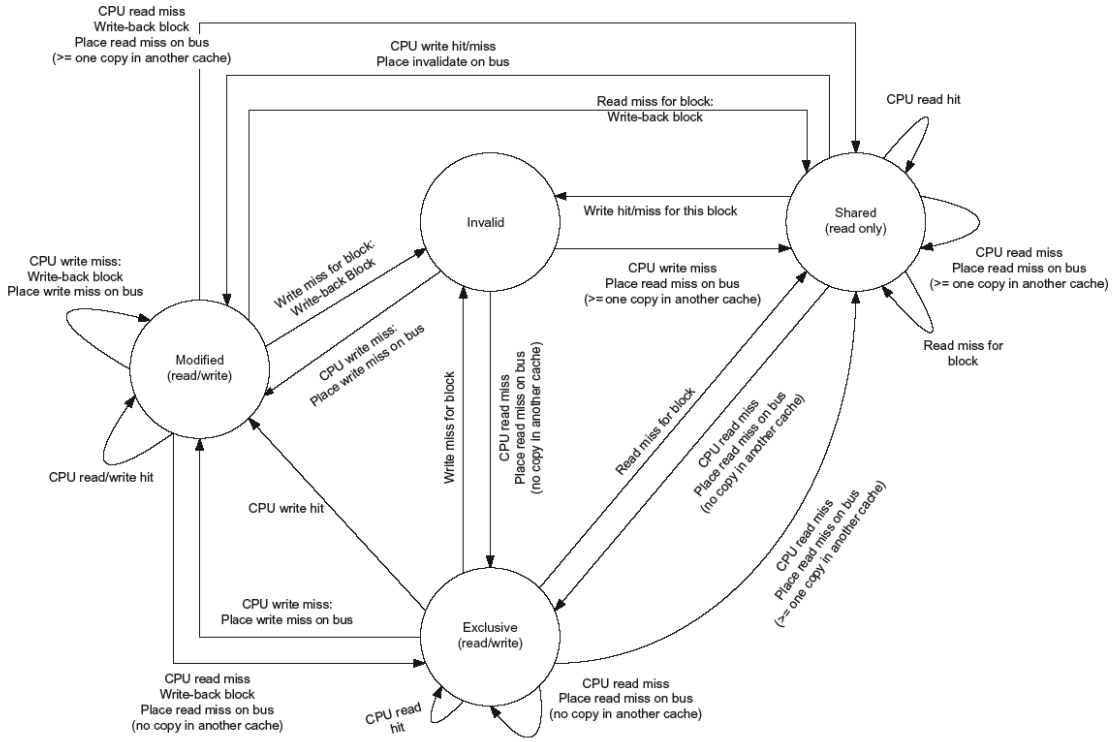

Figure S.1 Protocol diagram.

5.7.

a. C1: R, AC10: C0 cache
writeback + C1 cache miss: 40 +10 = 50 cycles for implementation 1.
130 +10 = 140 cycles for implementation 2
C3: R, AC10 C3 cache miss: 40 cycles for implementation 1.
130 cycles for implementation 2.
C0: R, AC10 C0 cache hit: 0 extra cycles for both implementations 1 & 2.
Total:
Implementation 1: 90 cycles, Implementation 2: 270 cycles
(Results same for MSI and MESI)

b. C1: R, AC20 : C1 cache miss: 40 cycles for implementation 1.
130 cycles for implementation 2.
C3: R, AC20: C3 cache hit: 0 extra cycles for both implementations 1 & 2.
C0: R, AC20: C0 cache miss: 40 cycles for implementation 1.
130 cycles for implementation 2.
Total:
Implementation 1: 80 cycles, Implementation 2: 260 cycles

(Results same for MSI and MESI)

c. C0: W, AC20 <- 80: C0

cache miss + C3 (S-> I):	40 cycles for implementation 1.
	130 cycles for implementation 2.
C3: R, AC20: C0 (M-> S)	
writeback + C3 miss:	10 + 40 = 50 cycles for implementation 1.
	10 + 130 = 140 cycles for implementation 2.
C0: R, AC20: C0 hit:	0 extra cycles for both implementations 1 & 2.

Total:

Implementation 1: 90 cycles, Implementation 2: 270 cycles

(Results same for MSI and MESI)

d. C0:W, AC08 <- 88 C0 write hit – send inv (S-> M), C3 (S->I): 15 cycles for both implementations 1 & 2.

C3: R, AC08 C0 writeback (M->S), C3 cache miss: 10 + 40 = 50 cycles implementation 1.

10 + 130 = 140 cycles implementation 2.

C0: W, AC08 <-98 C0 write hit – send inv (S-> M), C3 (S->I): 15 cycles for both implementations 1 & 2.

Total:

Implementation 1: 80 cycles, Implementation 2: 170 cycles

(Results same for MSI and MESI)

Difference between MSI and MESI would show if there is a read miss followed by a write hit with no intervening accesses from other cores.