ECE 786 Advanced Computer Architecture: Data Parallel Processors
Spring 2023 Huiyang Zhou HW 5

# 1

In this exercise, we analyze how the vector instructions of a Vector-Add function run on vector processors. The following is the instruction sequence in the VMIPS format:

```
LV       V1, R1
LV       V2, R2
ADDVV.D  V3, V1, V2
SV       V3, R3
```

Using the pipeline timing diagram to show how these instructions are executed on one vector processor pipeline.

The following assumptions are made:

- the vector length is 64 (i.e., each vector register has 64 elements)
- the latency of the load/store unit is 5 cycles (i.e., 1 AGEN stage and 4 MEM stages); the latency of the adder is 3 cycles (i.e., 3 EX stages); both the load/store unit and adder are fully pipelined and the throughputs are one element per cycle after the pipeline is filled
- there are 2 lanes in the vector processor and each lane has an ALU and a load/store unit.

How many cycles does it take to execute all the instructions in an in-order pipeline with ALL the necessary dependency checking logic, support for chaining, and register read/write ports, counting from when the first instruction enters the IF stage to when the last instruction leaves the WB stage?

It takes a total of 72 cycles to execute the pipeline. The full pipeline diagram is written as a spreadsheet and is attached to the submission. Please note that assumptions were made, such as there being two lanes for load and store unit and the ALU. The question mentioned a throughput of one element per cycle, but I assumed that meant the individual lane throughput, not the vector unit throughput (1 cycle per every element in 2 long vector, or 2 cycles per 2 elements). Although there is vector chaining and bypassing, the writebacks were bottlenecked by the throughput, thus stalls were inserted executing the instructions sequentially. Because there are 2 load/store units, I also assumed this meant that there are 2 register read and write ports, thus why WBs were bottlenecked due to structural hazard.

# 2

Analyze the following kernel function (the naive transpose kernel) to see whether there are scalar operations and whether the AMD GCN architecture will be beneficial compared to the SIMT architecture. Explain your answer (i.e., if you say scalar operations exist, identify them).

The constants `TILE_DIM = 32` and `BLOCK_ROWS = 8`.

```
__global__ void transposeNaive(float *odata, float* idata, int width, int height, int nreps) {
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index_in = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;
    for (int r=0; r < nreps; r++) {
        for (int i=0; i < TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i] = idata[index_in+i*width];
        }
    }
}
```

There are scalar operations or scalar values that can benefit from a separate scalar vector register file, which are the variables `r` and `i`. `r` is not that important, because its data is not used in the transpose, and is used to iterate repetitively. But because a SIMT architecture stores the `r` register repeatedly per thread, it can benefit from space savings on a scalar register file or AMD GCN. The next is `i`, and this can also benefit AMD GCN, because the value `i` can be shared across multiple threads since it's just computing the blocking and tiling offsets. Thus, the scalar operations are the index operators on the two `for` loops, while the transpose operation `odata[index_out+i] = idata[index_in+i*width]` includes vector and scalar operations to compute the indices for each thread.