Homework Assignment #4
SAMPLE SOLUTION

Q1. (8 points) Please briefly explain what a vector architecture is.

ANSWER: A vector architecture uses vector registers, vector functional units, and vector load-store units, to operate on different data items concurrently to solve a problem.

Q2. (10 points) Please briefly explain what the below two RV64V vector instructions do.

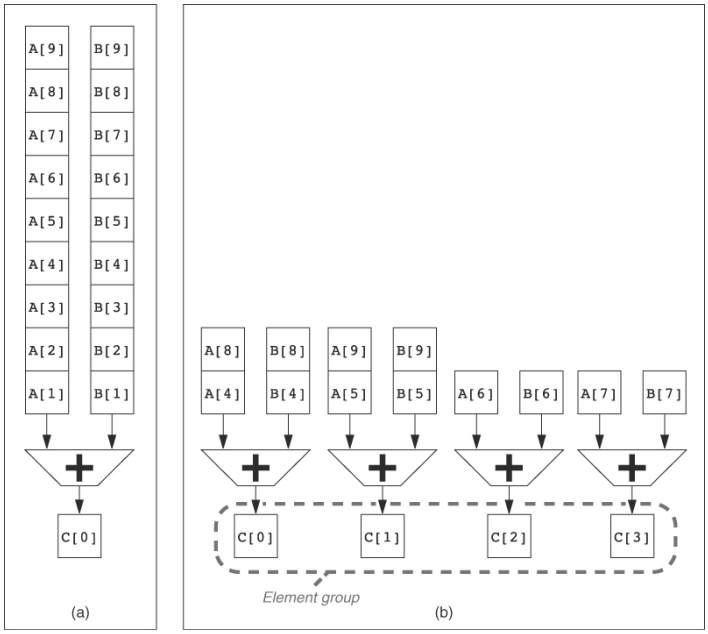
```
vadd v3, v1, v2
vld  v1, r1
```

ANSWER: The first instruction performs adding two vectors, in v1 and v2 registers, and the result goes to v3 vector register.

The second instruction loads a vector (32 elements, 64-bit (8 bytes) data each element) from memory starting at address r1.

Q3. (12 points) Please use a vector add example to explain what multiple lanes is in a vector architecture.

ANSWER: Multiple lanes in a vector architecture are to use multiple functional units to improve the performance of vector operations. For instance, as shown in the figure below, a vector processor with multiple lanes shown in (b) can have four add pipelines and can complete four additions per cycle.



Q4. (8 points) Please briefly explain what grid and thread block are in the GPU architecture.

ANSWER: A grid is a vectorized (or parallelized) code that works over all data elements, which is composed of thread blocks.

A thread block is composed of threads, where one thread execution can operate on multiple data elements at a time.

Q5. (15 points) In a CUDA program doing vector addition, let's assume that we would like each thread to calculate one output element of the addition. How would you map the data index to thread id and/or block id? Which expression provides the best mapping below? Please explain and justify your choice in your own words.

a) i = blockIdx.x + threadIdx.x;
b) i = threadIdx.x + threadIdx.y;
c) i = blockIdx.x * threadIdx.x;
d) i = blockIdx.x * blockDim.x + threadIdx.x;

ANSWER: Organizing threads generally reflects the dimensionality of the data, and based on that, dimensional structures are chosen. blockDim.x variable gives the total number of threads in all block. threadIdx is a variable that assigns a unique coordinate to each thread. The blockIdx variable assigns a common block coordinate to all threads in a block.

Each thread can construct a unique global index for itself within the grid by combining its threadIdx and blockIdx values. This will map thread or block indices to data index. And the expression will be:
i = blockIdx.x * blockDim.x + threadIdx.x;

Hence, the correct answer is option d).

Q6. (20 points) In the following loop, find all the true dependences, output dependences, and anti-dependences. Eliminate the output dependences and anti-dependences 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 */
}
```

ANSWER:
True dependencies (read-after-write dependencies):
S2 and S1 through A[i]
S4 and S3 through A[i]

Output dependencies (write-after-write dependencies):
S3 and S1 through A[i]

Anti-dependencies (write-after-read dependencies):
S2 and S1 through B[i]
S3 and S2 through A[i]
S4 and S3 cause an anti-dependency through C[i]

Re-written code (other solutions exist):

```
for (i=0; i<100; i++) {
     A[i] = A[i] * B[i];      /* Remains the same */
     B1[i] = A[i] + c;        /*  Creating  B1  to  eliminate
     the anti-dependence */
     A1[i] = C[i] * c;        /*  Creating  A1  to  eliminate
     the anti-dependence and output dependence */
     C1[i] = D[i] * A1[i];    /*  Creating  C1  to  eliminate
     the anti-dependence */
}
```

Q7. (15 points) 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:
S1 uses the value (`B[i]`) computed by S2 in the previous iteration (`B[i+1]`). As such, there is a loop-carried dependence, and the loop is not parallel.

The loop can be made parallel with a loop transformation as below:

```
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];
```

Note that, in the transformed for loop, there is a true dependence within each iteration (`B[i+1]`), but there is no loop-carried dependence anymore. Therefore, the loop is parallel now.

Q8. (12 points) Assume a hypothetical GPU with the following characteristics:

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

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

Assuming each single precision operation requires four-byte two operands and outputs one four-byte result, is this throughput sustainable given the memory band-width limitation?

ANSWER: Without considering memory bandwidth, this GPU has a peak throughput of:

$$1.5 \times 16 \times 16 = 384\ GFLOPS/s$$

of single-precision throughput.

Assuming each single precision operation requires four-byte two operands and outputs one four-byte result (i.e., assuming there is no reuse of data, or no temporal locality), sustaining this throughput 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 given the 100 GB/s off-chip memory bandwidth limitation.

THE END.