Advanced Computer Architecture

HOMEWORK #3

Submitted by: Maliha Arif
PID: 4506817
Submitted to: Dr. Dan C. Marinescu
Submitted on: 30th October 2018

UCF - Fall 2018

Question 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 the processor runs at 700 MHz 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. What is the arithmetic intensity of this kernel? Justify your answer.

Arithmetic intensity of kernel= number of floating point operations
.. Byte of memory accessed

- So, we know, there are 4 multiply operations and 1 add and 1 sub which makes a total of 6 operations
- Memory is accessed when c_im, c_re are being used to save results
  It is accessed when a_im, a_re, b_im, b_re are being accessed hence the total is a 6.

Arithmetic intensity of kernel = 6/6 =1

Hence arithmetic intensity is 1.

b. Convert this loop into RV64V assembly code using strip mining.

Solution: RV64V

My Key:

- mvl=64
- v1,v2,v3,v4 are a_re,b_re,a_im,b_im respectively
- v5 and v6 are c_re and c_im
- vl= vector length

```
        vsetdcfg    6 DP FP         # load 6 64-bit floating point registers
loop:   setvl       t0, a0          # vl=t0=min(mvl,n)
        vld         v1,a_re         # load a_re
        slli        t1,t0,3         #t1=vl*8 ( in bytes)
        add         a_re,a_re,t1    # Increment pointer to a_re by vl*8
        vld         v2,b_re         #load b_re
        add         b_re,b_re,t1    # Increment pointer to b_re by vl*8
        vmul        v5,v1,v2        #a_re*b_re
        vld         v3,a_im         # load a_im
        add         a_re,a_re,t1    # Increment pointer to a_im by vl*8
        vld         v4,b_im         #load b_im
        add         b_re,b_re,t1    # Increment pointer to b_im by vl*8
        vmul        v6,v3,v4        # a_im*b_im
        vsub        v5,v5,v6        # a_re*b_re - a+im*b_im
        vst         v5,c_re         #store in c_re
        vmul        v5,v1,v4        # a_re * b_im
        vmul        v6,v2,v3        #a_im*b_re
        vadd        v6,v5,v6        # a_re*b_im + a+im*b_re
        vst         v6,c_im         #store in c_im
        sub         a0,a0,t0        # n-vl( t0)
        bnez        a0,loop         #Repeat if n!=0
        vdisable                    #disable vector registers
```

c. Assuming chaining and a single memory pipeline, how many chimes are required?

1. vmul a_re,b_re #assuming a_re and b_re have been loaded , both are multiplied
   vld a_im #a load is performed in same convoy
2. vld b_im
   vmul a_im,b_im #second load and multiply is the second convoy
3. vsub a_re,a_im
   vst c_re #we subtract and save in 3rd convoy
4. vmul a_re,b_im #mulpiply in 4th convoy, subsequent multiply will be in separate convoy
5. vmul a_im,b_re #5th convoy
6. vdd a_re,b_im
   vst c_im #add and store in 6th convoy

6 convoys= 6 chimes

d. If the vector sequence is chained, how many clock cycles are required per complex result value, including overhead?

# of clock cycles

multiply – 8 cycles
add/sub – 5 cycles
load/store – 15 cycles

Total clock cycles: 4 x 8 (multiply) =32 +
5 x 2 (+,-) = 10 +
6 x 15 , (load – 4) +2 (save) = 90 +
6 x 64 = 384 (since there are 6 convoys = 6 chimes for 1 element and there are 64 elements that can multiplied together as vector)

32 +10+90+384 = 516 clock cycles

Now to find clock cycles per complex value, we know 516 is the total when 2 complex values are computed. 64 elements for c_re and 64 for c_im hence

Answer: 516/128 = 4 clock cycles required per complex result

e. Now assume that 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?

Now we have 3 memory pipelines

-We observe each element takes 6 chimes again even though there are more load/store units

because multiply and add / subt units are same in number so we end up with same 6 convoys and hence 6 chimes per element

and there are 64 elements that can be computed so 6 x 64 =384 clock cycles

-And each multiply and add/sub unit takes same time hence there is not much difference and we get same clock cycles for them. Performance overall does not improve.

516/128 = 4 clock cycles per complex value

Question 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. Since not all memory latencies are covered, assume an average SIMD instruction issue rate of 0.85. Assume that the GPU has a clock speed of 1.5 GHz.

a. **[10] <4.4> Compute the throughput, in GFLOP/sec, for this kernel on this GPU.**

Throughput will be total # of floating point operations per second
= 80% (active threads) * 70% (arithmetic operations) * 0.85 (instruction issue rate) * 1.5 GHz (clock speed) * 10 cores (SIMD processors are 10) * 32/4 =8 (results produced per clock cycle) =

Answer: 57.12 GFLPOS/s

b. [15] <4.4> Assume that you have the following choices: What is speedup in throughput for each of these improvements?

(1) Increasing the number of single-precision lanes to 16

Throughput will be total # of floating point operations per second
= 80% (active threads) * 70% (arithmetic operations) * 0.85 (instruction issue rate) * 1.5 GHz (clock speed) * 10 cores (SIMD processors are 10) * **32/2 =16 (results produced per clock cycle)** = **114.24 GFLOPS/s**

only results produced per clock cycle will change as there are 16 lanes now, 32 results will be produced after every 2 clock cycles hence 32/2= 16

Answer: 114.24 GFLOPS/s hence

Speed up = 114.24/57.12 = 2

(2) Increasing the 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)

Throughput will be total # of floating point operations per second
= 80% (active threads) * 70% (arithmetic operations) * 0.85 (instruction issue rate) * 1.5 GHz (clock speed) * **15 cores (SIMD processors are 15 now)** * 32/4 =8 (results produced per clock cycle) = **85.68 GFLOPs/s hence**

Speedup = 85.68/57.12 = 1.5

(3) Adding a cache that will effectively reduce memory latency by 40%, which will increase instruction issue rate to 0.95

Throughput will be total # of floating point operations per second
= 80% (active threads) * 70% (arithmetic operations) * **0.95 (instruction issue rate will change only)** * 1.5 GHz (clock speed) * 10 cores (SIMD processors are 10) * 32/4 =8 (results produced per clock cycle) = **63.84 GFLOPs/s hence**

Speedup = 63.84/57.12 = 1.117

Highest speed up is in option 1 hence that's the best option.

Question 4.15:

List and describe at least four factors that influence the performance of GPU kernels. In other words, which runtime behaviors that are caused by the kernel code cause a reduction in resource utilization during kernel execution?

There are few important factors that affect the performance of a GPU at run time. They are also responsible for affecting resource utilization whether it be memory, cache, floating point operational units etc. The 4 factors are as follows:

1. Memory latency: Memory latency affects performance of a GPU at run time. The original philosophy to tackle memory latency was using a sufficient number of threads but al recent GPUs are using caches to reduce latency now. This follows Little's law, the longer the latency, the more threads needed to run which means requiring more registers. GPU caches have been added which lower average memory latency and thereby mask potential shortages of the number of registers.

2. Branch divergence: Branch divergence is done in GPUs so that IF statements are effectively handled. In addition to predicate registers, GPU branch hardware uses internal masks, a branch synchronization stack and instruction markers to manage when a branch diverges onto multiple execution paths. At the PTX assembler level, control flow of one CUDA thread is described by instruction branch, call, return and exit plus individual per thread lane predication of each instruction. The PTX assembler sets a branch synchronization marker on appropriate conditional branch instructions that pushes the current active mask on a stack inside SIMD thread. If the conditional branch diverges, it pushes a stack entry and sets the current internal active mask based on the condition. Hence this causes SIMD lanes to be masked when threads follow different control paths and helps tackle conditional statements at run time.

3. Use of on-chip memory: Memory references with locality should take advantage of on-chip memory. In this way, bank conflicts can be avoided when references to on-chip memory within a SIMD thread group are organized.

4. Lastly, Coalesced data transfers or off-chip memory references: Unlike vector architectures, GPUs don't have separate instructions for sequential data transfers. To tackle this, and gain efficiency of sequential data transfers, GPUs include special Address Coalescing hardware to recognize when the SIMD Lanes within a thread of SIMD instructions are collectively issuing sequential addresses. That runtime hardware then notifies the Memory Interface Unit to request a block transfer of 32 sequential words. To get this important performance improvement, the GPU programmer must ensure that adjacent CUDA Threads access nearby addresses at the same time so that they can be coalesced into one or a few memory or cache blocks.

Question: 4.16

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/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: Throughput= 1.5 GHz * 16 cores (SIMD processors) * 16 (FPUs) = 384 GFLOPS/s

Memory bandwidth required would be, assuming 4 bytes for 2 operands and 4 bytes for one results/output 4*3 = 12 bytes memory access /FLOP

Hence total times memory accessed would be **12*384= 4608 GB/s or 4.6 TB/s** whereas max memory bandwidth supplied is 100 GB/s hence this throughput is not sustainable considering the memory bandwidth limitation. Using on-chip memory, maybe it can be achieved using short bursts.