Arquitetura de Computadores 2 – 2022/1
Lista 3 – DLP

4.9)
a) Este código lê quatro floats e escreve dois floats para cada 6 FLOPs, então:
Intensidade aritmética = 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 de ciclos por iteração = 6 chimes × 64 elementos + 15 ciclos (load/store) × 6 + 8 ciclos (multiplicação) × 4 + 5 ciclos (add/subtract) × 2 = 516
Ciclos por resultado = 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
```

Alguns ciclos por resultado como na letra c, adicionar unidades de load/store não melhora o desempenho.

4.10
Vector processor requires:
■ (200 MB + 100 MB)/(30 GB/s) = 10 ms for vector memory access +
■ 400 ms for scalar execution.
Assuming that vector computation can be overlapped with memory access, total time = 410 ms.
The hybrid system requires:
■ (200 MB + 100 MB)/(150 GB/s) = 2 ms for vector memory access +
■ 400 ms for scalar execution +
■ (200 MB + 100 MB)/(10 GB/s) = 30 ms for host I/O
Even if host I/O can be overlapped with GPU execution, the GPU will require 430 ms and therefore will achieve lower performance than the host.

4.11
a) Para este item, a redução seria dobrar a recorrência, somando-se vetores mais curtos.
Para este caso a solução seria paralelizar em: 2x32, 4x16; 8x8 ou 16x4, 32x2.
Por exemplo, para 2x32, teríamos:
for (i=0;i<32;i+=2) dot[i] = dot[i]+dot[i+1];
A soma aqui é dobrada a cada iteração do loop: dot[0] = dot[0] + dot[1] e depois dot[2] = dot[2]+dot[3] ...
Solução:

```
dot=0.0;
for (i=0; i<64;i++) dot = dot + a[i] + b[i]
Redução:
for (i=0; i<64;i++) dot = a[i] + b[i]
for (i=1; i<64;i++) dot[0] = dot[0] + dot[i]
```

a) A redução para paralelizar seria: 2x32, 4x16; 8x8 ou 16x4, 32x2.
Logo, o código em C pode ser:

```
for (i=0;i<32;i+=2) dot[i] = dot[i]+dot[i+1];
for (i=0;i<16;i+=4) dot[i] = dot[i]+dot[i+2];
for (i=0;i<8;i+=8) dot[i] = dot[i]+dot[i+4];
for (i=0;i<4;i+=16) dot[i] = dot[i]+dot[i+8];
for (i=0;i<2;i+=32) dot[i] = dot[i]+dot[i+16];
dot[0]=dot[0]+dot[32];
```

Comentário: Dividir o vetor de 64 posições em dois de 32 posições, chamar a função para cada um deles e ao final somar as posições 0 de cada vetor.
Resultado após primeiro for:
0[0,1], 2[2,3], 4[4,5], 6[6,7], 8[8,9], 10[10,11], 12[12,13] .... 30[30,31]
Resultado após segundo for:
0[0,1,2,3], 4[4,5,6,7], 8[8,9,10,11] ,12[12,13,14,15]
.... 28[28,29,30,31]
Resultado após terceiro for:
0[0,1,2,3,4,5,6,7] ,8[8,9,10,11,12,13,14,15]
.... 24[24,25,26,27,28,29,30,31]
Resultado após quarto for:
0[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], 16[16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]
Resultado após quinto for:
0[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]

b) A solução aqui é usar somas parciais, o que pode ser feito dividindo em vetores dentro do mesmo registrador.Então a soma de dois vetores de precisão simples no mesmo registrador $v0 deve ser indexada pelas posições dentro do vetor.Supondo que as somas parciais seja de 4 posições (indexando de 4 em 4 dentro do endereçamento) :li $VL,4A soma será entre os elementos múltiplos de 4:addvv.s$v0(0),$v0(4); addvv.s $v0(8),$v0(12) ...addvv.s $v0(56),$v0(60)
Solução:

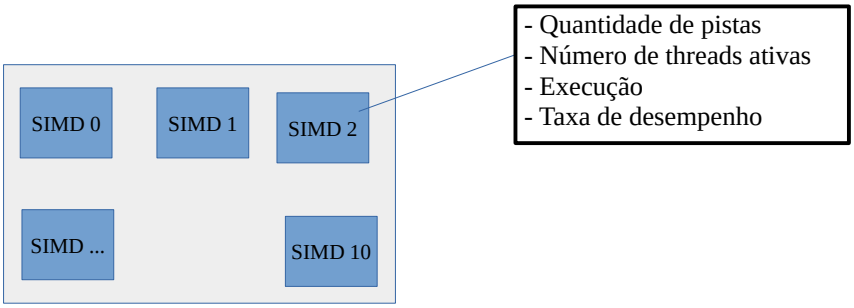
```
li $VL,4
addvv.s $v0(0),$v0(4)
addvv.s $v0(8),$v0(12)
addvv.s $v0(16),$v0(20)
addvv.s $v0(24),$v0(28)
addvv.s $v0(32),$v0(36)
addvv.s $v0(40),$v0(44)
addvv.s $v0(48),$v0(52)
addvv.s $v0(56),$v0(60)
```

c) Cada thread grava seu valor na memória compartilhada.
Dentro do loop cada thread soma cada par de valores, reduzindo a quantidade de threads pela metade,então a dimensão do bloco será: blockDim.x/2,com decremento s=s/2, enquanto s > 0.

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

4.13
Uma GPU contém 10 processadores SIMD.
Cada instrução SIMD contém uma largura de 32, com 8 pistas produzindo resultados a cada 4 ciclos.
Aplicação com desvios divergentes com threads ativas em 80% do tempo, com 70% de instruções aritméticas e 20% armazenamento.
Taxa de despacho do SIMD = 0,85
GPU com frequência de 1,5 GHz ou 1,5 Gciclos/s

a) Calcular a vazão



Vazão = f_GPU X número cores x quantidade de pistas x número de threads x NI_FP x despacho
Vazão = 1,5 Gciclos/s x 10 cores x 8 pistas x 0,8 threads ativas x 0,7 instruções x 0,85 instruções/ciclo = 57,12 GFLOPS
Não contabiliza a latência da memória.

b) Não contabiliza a latência da memória

Aumentar o número de pistas para 16:
Vazão = 57,12 x 2 = 114,24 GFLOPS (DOBRA O VALOR ANTERIOR)
Speed-up = 2

Aumentar o número de cores para 15:
Vazão = 1,5 Gciclos/s x 15 cores x 8 pistas x 0,8 threads ativas x 0,7 instruções x 0,85 instruções/ciclo = 85,68 GFLOPS
Speed-up = 85,68/57,12 = 1,5

Inserir cache que reduz a latência da memória em 40% e a taxa de despacho para 0,95

Vazão = 1,5 Gciclos/s x 10 cores x 8 pistas x 0,8 threads ativas x 0,7 instruções x 0,95 instruções/ciclo = 63,84 GFLOPS
Speed-up = 63,68/57,12 = 1,11

4.14
a) Pela teoria de compiladores, o teste do maior divisor comum GDC (Greatest Common Divisor)

Se um loop sequencial for paralelo para ser executado em mais que um processador, certas dependências devem ser verificadas para saber se o loop pode ser paralelizável. De acordo com o teste do maior divisor comum GDC (Greatest Common Divisor), comparando os índices de duas matrizes presentes em duas ou mais instruções, pode-se calcular se é possível ou não paralelizar o loop.
Se X [a * i + b] e X [c * i + d] (onde X é a matriz; a, b, c e d são números inteiros e i é a variável do loop), então GCD (c, a) deve dividir (d – b)

```
for (i=0;i<100;i++) {
        A[i] = B[2*i+4];
        B[4*i+5] = A[i]
}
```

Dependência entre os loops está entre B[4*i+5] e B[2*i+4], GDC(c,a) = GDC (2,4) é 4 que deve dividir (d-b) = 4-5 = -1, Como 4 divide por -1, o loop é paralelizável!

b)
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 */
}
RAW: S2 com S1 e S4 com S3 em A[i]
WAW: S3 com S1 em A[i]
WAR: S4 com S3 em C[i] e S2 com S1 em B[i]
Reescrendo o código
for (i=0;i<100;i++) {
T[i] = A[i] * B[i]; /* S1 */
B[i] = T[i] + c; /* S2 */
A1[i] = C[i] * c; /* S3 */
C1[i] = D[i] * A1[i]; /* S4 */} RAW – S4 com S3 em A[i] e S2 com S1 em T[i]
c)

for (i=0;i<100;i++) {
A[i] = A[i] * B[i]; /* S1 */
B[i+1] = C[i] + D[i]; /* S2 */

WAW: S2 com S1 em B[i] a cada nova iteração B[i+1] com o próximo B[i].

Há uma anti-dependência entre a iteração i e i+1 para o arrray B.
Isso pode ser evitado renomeando o array B no S2.

for (i=0;i<100;i++) {
A[i] = A[i] * B[i]; /* S1 */
T[i+1] = C[i] + D[i]; /* S2 */

4.15
Listar fatores que influenciam o desempenho de código (kernels) de GPU.
Que fatores devem ser considerados para que tempo de execução do kernel que provocam redução do uso de recursos durante a execução do kernel?
a) Desvios divergentes: faz com que as pistas do SIMD sejam mascaradas quando as threads seguem diferentes caminhos de controle

b) Latência da memória: um número suficiente de threads ativos pode ocultar a latência da memória e aumentar a taxa de emissão de instruções

c) Referências de memória agrupadas fora do chip: os acessos à memória devem ser organizados consecutivamente dentro dos grupos de threads do SIMD

d)Uso de memórias on-chip: o uso de localidades de referências na memória devem tirar proveito de memórias on-chip, referências à memória no chip dentro de um grupo de threads SIMD devem ser organizadas para evitar conflitos no banco de memórias

4.16
GPU: taxa de 1,5 GHz, 16 processadores SIMD com 16 unidades de FP.s
Memória fora do chip: largura de 100GB/s
Qual é o pico de vazão de FP.s para a GPU considerando que todas as latências foram ocultadas?

Vazão = f_GPU X número cores x quantidade de pistas x NI_FP x ~~número de threads x~~ ~~despacho~~

Vazão = 1,5 GHz x 16 cores x 16 pistas = 384 GFLOPS

Cada operação de FP requer dois registradores de quatro bytes para a operação e um registrador de destino de 4 bytes, logo requer 12 bytes /FLOP:
Vazão = 384 GFLOP/s x 12 bytes/FLOP = 4,6 TB/s

Logo, a vazão seria insustentável, uma vez que uma rajada de 4,6 TB/s não seria absorvida por um conjunto de memórias fora do chip com capacidade de 100GB/s