Bonus3 DLP, IS2202

Linghan Zhao
020717-5563
linghanz@kth.se

February 21, 2024

Problem 1. Data Level Parallelism

a) *LoopLength* $L1 = 100 + Y + y + M + m + D + d = 117$

```
for (i = 0; i < L1; i=i+1) {
    C[i] = X[i]* Y[i]-W[i]*Z[i];
    D[i] = X[i]* Z[i]+Y[i]*W[i];
    B[i]=C[i]*C[i]+D[i]*D[i];
 }

Using Strip-mining:
low = 0;
VL = (L1 % MVL);
for (j=0; j<=L1/MVL; j++) {
    for (i=low; i<(low+VL); i++)
        C[i] = X[i]* Y[i]-W[i]*Z[i];
        D[i] = X[i]* Z[i]+Y[i]*W[i];
        B[i]=C[i]*C[i]+D[i]*D[i];
    low = low + VL;
    VL = MVL ;
}
```

Then write down the VMIPS code for the loop.

```
    LI.D F0, 117.0
    LI Rk, 0
    MODU R4,F0,#64 ; VL=L1%MVL
    DIVU Rj,F0,#64 ; L1/MVL

Loop_j:
    MTC1 VLR,R4 ; Set vector length

    LV.D V1,Rx ; Load X[i]
    LV.D V2,Ry ; Load Y[i]
    MULVV.D V5,V2,V1 ; X[i]*Y[i]
    LV.D V3,Rw ; Load W[i]
```

```
    LV.D V4,Rz ; Load Z[i]
    MULVV.D V6,V4,V3 ; W[i]*Z[i]
    SUBVV.D V9,V5,V6 ; C=XY-WZ
    SV.D V9,Rc ; Store C[i]

    MULVV.D V5,V4,V1 ; X[i]*Z[i]
    MULVV.D V6,V2,V3 ; W[i]*Y[i]
    ADDVV.D V8,V5,V6 ; D=XZ+YW
    SV.D V8,Rd ; Store D[i]

    MULVV.D V5,V7,V7 ; C[i]*C[i]
    MULVV.D V6,V8,V8 ; D[i]*D[i]
    ADDVV.D V9,V5,V6 ; B=CC+DD
    SV.D V9,Rb ; Store B[i]

    ADDIU Rx,Rx,R4 ; Set Rx to X[i+low]
    ADDIU Ry,Ry,R4 ; Set Ry to Y[i+low]
    ADDIU Rw,Rw,R4 ; Set Ry to W[i+low]
    ADDIU Rz,Rz,R4 ; Set Ry to Z[i+low]

    L.D R4,#64 ; VL=MVL
    SUBIU Rj,Rj,#1; ; Decr j
    BGE Rj, Rk, Loop_j ; Branch if still positive
```

.

b) We can estimate the execution time with Convoys, Chimes, MOH, DOH.

```
Convoy 1:
    LV.D V1,Rx ; Load X[i]
Convoy 2:
    LV.D V2,Ry ; Load Y[i]
    MULVV.D V5,V2,V1 ; X[i]*Y[i]
Convoy 3:
    LV.D V3,Rw ; Load W[i]
Convoy 4:
    LV.D V4,Rz ; Load Z[i]
    MULVV.D V6,V4,V3 ; W[i]*Z[i]
    SUBVV.D V9,V5,V6 ; C=XY-WZ
Convoy 5:
    SV.D V9,Rc ; Store C[i]
    MULVV.D V5,V4,V1 ; X[i]*Z[i]
Convoy 6:
    MULVV.D V6,V2,V3 ; W[i]*Y[i]
    ADDVV.D V8,V5,V6 ; D=XZ+YW
    SV.D V8,Rd ; Store D[i]
Convoy 7:
    MULVV.D V5,V7,V7 ; C[i]*C[i]
Convoy 8:
    MULVV.D V6,V8,V8 ; D[i]*D[i]
```

```
    ADDVV.D V9,V5,V6 ; B=CC+DD
    SV.D V9,Rb ; Store B[i]
```

Totally 8 Convoys, $T_{Chimes} = 64(ccs)$,
so $T_0 = 8 * 64 = 512(ccs)$
Load/Store : 7 times
Memory Instruction $MOH = 8 + d = 15(ccs)$
ADD/SUB and MUL : 9 times
ADD and MUL instruction $DOH = 4 + D = 5(ccs)$
$OH = 7 * MOH + 9 * DOH = 7 * 15 + 9 * 5 = 150(ccs)$
The estimated execution time in clock cycles including overhead
$T_0 + OH = 512 + 150 = 662(ccs)$, for every loop.

c) Convert the C-code to CUDA code.
$LoopLength \quad L1 = 100 + Y + y + M + m + D + d = 117$
Number of CUDA threads $NrThreads = 32 * (1 + M) = 32 * 1 = 32(Threads)$

```
// Invoke CALCULATE with NrThreads threads per block
__host__
int L1 = 117;
int NrThreads = 32;
int nblocks = (L1+NrThreads-1)/NrThreads;
calculate<<<nblocks, NrThreads>>>(L1, X, Y, W, Z, C, D, B)

//CALCULATE in CUDA
__device__
void calculate(int L1, double *X, double *Y, double *W, double *Z, double
    *C, double *D, double *B) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < L1) {
        C[i] = X[i] * Y[i] - W[i] * Z[i];
        D[i] = X[i] * Z[i] + Y[i] * W[i];
        B[i] = C[i] * C[i] + D[i] * D[i];
    }
}
```

Problem 2. VMIPS vs GPU

$Picture \quad x\_size = 1920 * (1 + D) = 1920 * 2 = 3840(pixels)$
$Picture \quad y\_size = 1080 * (1 + M) = 1080 * 1 = 1080(pixels)$
a) The target machine is a 32 vector length 32 bit Single Precision VMIPS architecture. Assume that all pixels are 8 bit integers, (but stored as 32 bit integers). Write down the VMIPS code for the Sobel operator.

```
# Constants
LIU R5, #512 ; Row Stride = 32 * 32
LIU Ry, #1078 ; Nr of iterations for y (y_size - 1)
LIU Rx, #3838 ; Nr of iterations for x (x_size - 1)
```

```
LIU Ri, #3 ; Nr of iterations for i
LIU Rj, #3 ; Nr of iterations for j

# Initialize loop_y
Loop_y:

    # Initialize loop_x
    Loop_x:

        # Initialize pixel_value_x and pixel_value_y
        L.D F0, #0.0
        L.D F1, #0.0

        # Initialize gradient_kernel_x and gradient_kernel_y indices
        LIU Rg, #0 ; Start of gradient_kernel_x
        LIU Rh, #0 ; Start of gradient_kernel_y

        # Loop for j
        Loop_j:

            # Loop for i
            Loop_i:

                # Load gradient_kernel_x[j+1][i+1] into scalar register
                LV.D V4, (Rg)

                # Load image[y+j][x+i] into scalar register
                LV.D V5, (Rf, R5)

                # Multiply scalar values
                MULVS.D V6, V4, V5

                # Accumulate result in vector register
                ADDVV.D V3, V3, V6

                # Move to next column in image
                ADDIU Rf, Rf, #8

                # Move to next column in gradient_kernel_x
                ADDIU Rg, Rg, #8

                # Decrement i
                SUBIU Ri, Ri, #1

                # Continue loop if i > 0
                BNEZ Ri, Loop_i

            # Reset i
```

```
        LIU Ri, #3
        SUBIU Ri, Ri, #1

        # Reset image row index
        SUBIU Rf, Rf, #24

        # Reset column index of gradient_kernel_x
        SUBIU Rg, Rg, #24

        # Move to next row in image
        ADDIU Rf, Rf, #1024

        # Move to next row in gradient_kernel_x
        ADDIU Rg, Rg, #1024

        # Decrement j
        SUBIU Rj, Rj, #1

        # Continue loop if j > 0
        BNEZ Rj, Loop_j

    # Reset j
    LIU Rj, #3
    SUBIU Rj, Rj, #1

    # Reset image row index
    SUBIU Rf, Rf, #3072

    # Reset column index of gradient_kernel_x
    SUBIU Rg, Rg, #3072

    # Move to next row in image
    ADDIU Rf, Rf, #1024

    # Move to next row in gradient_kernel_x
    ADDIU Rg, Rg, #1024

    # Calculate pixel_value = abs(pixel_value_x) + abs(pixel_value_y)
    ABS.D F2, F0
    ABS.D F3, F1
    ADD.D F4, F2, F3

    # Compare pixel_value with threshold
    LI.D F5, #threshold
    CGT.D F6, F4, F5

    # Convert boolean result to integer (255 or 0)
    MFC1 V7, F6
```

```
        CVT.W.S V7, V7
        ANDI V7, V7, #255

        # Store result in edge_image[y][x]
        S.D (Re, #0),

        # Continue loop x
        SUBIU Rx, Rx, #1
        BNEZ Rx, Loop_x

    # Continue loop y
    SUBIU Ry, Ry, #1
    BNEZ Ry, Loop_y
```

.

b) Write down the CUDA code for a GPU implementation of the gradient function. Number of CUDA threads $NrThreads = 32 * (1 + M) = 32 * 1 = 32(Threads)$

```
#define NrThreads 32
#define x_size 3840
#define y_size 1080
// Invoke SobelOperator with NrThreads threads per block
__host__
// Determine grid and block dimensions
dim3 NrBlocks((x_size + NrThreads - 1) / NrThreads, (y_size + NrThreads -
    1) / NrThreads);
dim3 NrThreadsPerBlock(NrThreads, NrThreads);
sobelOperator<<<NrBlocks, NrThreadsPerBlock>>>(image, edge_image, x_size,
    y_size, threshold);

//SobelOperator in CUDA
__device__
void sobelOperator(int *image, int *edge_image, int x_size, int y_size,
   int threshold) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if ( y > 0 && y < y_size - 1) {
        if(x > 0 && x < x_size - 1 ){
            double pixel_value_x = 0.0;
            double pixel_value_y = 0.0;

            for (int j = -1; j <= 1; j++) {
                for (int i = -1; i <= 1; i++) {
                    int image_index_x = x + i;
                    int image_index_y = y + j;
                    int kernel_index = (j + 1) * 3 + (i + 1); //3x3 kernel
                    pixel_value_x += gradient_kernel_x[kernel_index] *
                        image[image_index_y * x_size + image_index_x];
```

```
                    pixel_value_y += gradient_kernel_y[kernel_index] *
                        image[image_index_y * x_size + image_index_x];
                }
            }

            // approximate sqrt(x^2+y^2) with |x|+|y|
            double pixel_value = abs(pixel_value_x) + abs(pixel_value_y);
            int new_pixel_value = 255 * (pixel_value > threshold);
            edge_image[y * x_size + x] = new_pixel_value;
        }
    }
}
```

.

c) It is more beneficial to run the algorithm on a GPU using NrThreads CUDA threads than on the VMIPS Machine.

Although the VMIPS machine offers a scalar/vector architecture suitable for processing images pixel by pixel efficiently, this Sobel operator algorithm does not really use much vector instructions to deal with the operands.

This Sobel operator algorithm requires complex branching(like loop_y, loop_x, loop_j, loop_i here) and irregular memory access patterns(of gradient_kernel and iamge here), so it may not benefit as much from the vector processing capabilities of VMIPS.

On the other hand, GPUs are highly parallel processors designed to handle massive amounts of data simultaneously. This algorithm can do the convolution-like operation with gradient kernel to produce results inside the loops without much data dependency on before and after clock cycles, so threads can be blocked together to execute in groups. It can be parallelized across many pixels, so GPU acceleration using CUDA threads can provide significant speedup compared to VMIPS.

References

Computer Architecture: A quantitative approach, 6th ed. by John Hennessy and David Patterson Morgan Kaufmann, 2017 ISBN: 978-0-12-811905-1.