Programming Massively Parallel Hardware

Assignment 2

Viktor Hansen

Task I - CUDA Programming w. Reduce and Segmented Scan

Task I.1 - Exclusive scan and exclusive segmented scan

Exclusive scan was implemented by having the host invoke `scanInc` with a choice of parameters displacing the output by 1 index to the right in the output array.

```
template<class OP, class T>
void scanExc(unsigned int  block_size,
             unsigned long d_size,
             T*            d_in,  // device
             T*            d_out  // device
) {
    // Compute inclusive scan for all elements, but displace indices by 1
    // in d_out. This works as scanIncKernel checks if gid is within
    // bounds of d_size.
    if (d_size > 0) {
      scanInc<OP,T>(block_size, d_size-1, d_in, d_out+1);
      cudaThreadSynchronize();

      // Set first element to the identity element
      scanExcFixUp<OP,T><<< 1, 1 >>>(d_out, d_size);
    }
}
```

Listing 1: Exclusive scan implemented by scanning inclusively and inserting the neutral element at index 0.

Subsequently, the kernel `scanExcFixUp` is called on the output array to insert the neutral element at the first index of the scanned output, yielding the desired result:

```
template<class OP, class T>
__global__ void
scanExcFixUp(T* d_out, unsigned int d_size) {
    const unsigned int gid = blockIdx.x*blockDim.x + threadIdx.x;
    // First element of exclusive scan should be the identity element.
    if (gid == 0)
      d_out[0] = OP::identity();
}
```

Listing 2: Kernel for inserting the neutral element at the first index of an array.

Segmented inclusive scan was implemented by first computing an adjacent array, containing flat subarrays with the neutral element as their first one. This is the same solution as the one in the Haskell implementation from the previous assignment. The adjacent array is computed by the kernel shown in Listing 3.

```
template<class OP, class T>
__global__ void
sgmScanExcAdj(T* d_in, int* flags, T* d_adj, const unsigned long d_size)  {
    const unsigned int gid = blockIdx.x*blockDim.x + threadIdx.x;
    d_adj[gid] = flags[gid] ? OP::identity() : d_in[gid];
}
```

Listing 3: Kernel for computing the adjacent subarrays.

The problem of computing the exclusive segmented scan now reduces to computing the inclusive segmented scan of the flag array and the adjacent array. The host interface to the computation is shown in Listing 4.

```
template<class OP, class T>
void sgmScanExc( const unsigned int  block_size,
                 const unsigned long d_size,
                 T*             d_in,  //device
                 int*           flags, //device
                 T*             d_out  //device
) {
    unsigned int num_blocks;
    //unsigned int val_sh_size = block_size * sizeof(T  );
    unsigned int flg_sh_size = block_size * sizeof(int);

    num_blocks = ( (d_size % block_size) == 0) ?
                   d_size / block_size      :
                   d_size / block_size + 1 ;

    // Compute the adjacant array - result could be stored in-place
    // in d_in, however, we choose to preserve it.
    T *d_adj;
    cudaMalloc((void**)&d_adj, d_size*sizeof(T));
    sgmScanExcAdj<OP, T> <<< num_blocks, block_size >>>(d_in, flags, d_adj,
    ↪  d_size);
    cudaThreadSynchronize();

    // Perform inclusive segmented scan of adjacent array to get exclusive
    // segmented scan result of original array.
    sgmScanInc<OP,T>(block_size, d_size, d_adj, flags, d_out);
}
```

Listing 4: Host function for computing the segmented scan of an array.

The already implemented tests were extended to incorporate the exclusive scans, and the implementations pass the tests.

Task I.2

A solution to the maximum segment problem was implemented both for the CPU and the GPU. MSSP related code was moved to separate source files, one file containing the host-functions responsible for invoking the kernels, `code/MsspHost.cu.h`, and another containing the actual kernels, found in `code/MsspKernels.cu.h`. The `MssOp` class modeling the homomorphism operator was implemented pr. the semantics presented in last week's assignment, see Listing 5.

```
class MsspOp {
  public:
    typedef MyInt4 BaseType;
    static __device__ inline MyInt4 identity() { return MyInt4(0,0,0,0); }
    static __device__ inline MyInt4 apply(volatile MyInt4& t1, volatile
     ↪  MyInt4& t2) {
        int mssx = t1.x, misx = t1.y, mcsx = t1.z, tsx = t1.w;
        int mssy = t2.x, misy = t2.y, mcsy = t2.z, tsy = t2.w;
        int mss = max(max(mssx,mssy), mssx+mssy);
        int mis = max(misx,tsx+misy);
        int mcs = max(mcsy,mcsx+tsy);
        int t   = tsx+tsy;
        return MyInt4(mss, mis, mcs, t);
    }
};
```

Listing 5: The MSSP operator.

The map for lifting the integers of a segment into the domain of the homomorphism is implemented in the `msspTrivialMap` kernel seen in Listing 6. The kernel simply lifts non-negative numbers into quads containing segment sum informations, so that the we may reduce using the `MsspOp` operator.

```
__global__ void
msspTrivialMap(int* inp_d, MyInt4* inp_lift, int inp_size) {
    const unsigned int gid = blockIdx.x*blockDim.x + threadIdx.x;
    if(gid < inp_size) {
      int x = inp_d[gid];
      inp_lift[gid] = (x > 0) ? MyInt4(x,x,x,x) : MyInt4(0,0,0,x);
    }
}
```

Listing 6: The (trivial) map kernel.

The reduction is implemented with a scan, as returning the last element of the scanned array equals the reduction. The described computation is implemented in the `Mss` function in `code/MsspHost.cu.h` shown in Listing 7.

```
int Mss( const unsigned int  block_size,
         const unsigned long d_size,
         int* d_in  //device
) {
    int res;
    unsigned int num_blocks;
    //unsigned int val_sh_size = block_size * sizeof(T  );
    unsigned int flg_sh_size = block_size * sizeof(int);

    num_blocks = ( (d_size % block_size) == 0) ?
                    d_size / block_size     :
                    d_size / block_size + 1 ;

    MyInt4 *inp_lift;
    MyInt4 *out_lift;
    cudaMalloc((void**)&inp_lift, d_size*sizeof(MyInt4));
    cudaMalloc((void**)&out_lift, d_size*sizeof(MyInt4));

    // Map to lift the input array
    msspTrivialMap <<< num_blocks, block_size >>> (d_in, inp_lift, d_size);
    cudaThreadSynchronize();

    // If we scan, we can simply return the last element, which is the
    ↪  reduction
    scanInc<MsspOp, MyInt4>(block_size, d_size, inp_lift, out_lift);
    cudaThreadSynchronize();

    // Get the result
    cudaMemcpy(&res, &out_lift[d_size-1].x, sizeof(int),
    ↪  cudaMemcpyDeviceToHost);

    return res;
}
```

Listing 7: The implemented MSS solution.

The solution was tested by comparing the results of the parallel algorithm with that of a "serial" implementation found in listing 8 and produces the correct results.

```
MyInt4 mssSerial(int *in, int length) {
    if (length == 1) {
      int x = *in;
      if (x > 0)
        return MyInt4(x,x,x,x);
      else
```

```
        return MyInt4(0,0,0,x);
    }
    int half = length/2;
    MyInt4 t1 = mssSerial(&in[0], half);
    MyInt4 t2 = mssSerial(&in[half], length-half);
    int mssx = t1.x, misx = t1.y, mcsx = t1.z, tsx = t1.w;
    int mssy = t2.x, misy = t2.y, mcsy = t2.z, tsy = t2.w;
    int mss = max(max(mssx,mssy), mssx+mssy);
    int mis = max(misx,tsx+misy);
    int mcs = max(mcsy,mcsx+tsy);
    int t   = tsx+tsy;
    return MyInt4(mss, mis, mcs, t);
}

int matVecMultTest() {
    const unsigned int block_size  = 512;

    int matrix_flag[] = { 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 };
    MyPair<int,float> matrix_flat[]  = {
      MyPair<int,float>(0,2.0),  MyPair<int,float>(1,-1.0),
      MyPair<int,float>(0,-1.0), MyPair<int,float>(1, 2.0),
       ↪  MyPair<int,float>(2,-1.0),
```

Listing 8: The implemented serial MSS solution.

The functions were tested with arrays of length 100 with values drawn randomly from the range [0..9] as the input.

Task I.3

Solutions to subtasks 1 and two are implemented in `code/PrimesQuicksort.hs` and produce the correct results when run with the input of the handed out test cases. The code for the flat implementation is shown in Listing 9.

```
flatSparseMatVctMult :: [Int] -> [(Int,Double)] -> [Double] -> [Double]
flatSparseMatVctMult flags mat b =
    let
      (inds,as)   = unzip mat
      bs          = map (\i -> b !! i) inds
      ps          = map (\(a,b) -> a*b) $ zip as bs
      sums        = segmScanInc (+) 0.0 flags ps
      flagl       = (tail flags) ++ [head flags]
      offs        = scanInc (+) 0 flags
      (ind,vals)  = unzip $ map (\(f,o,s) -> if f > 0 then (o,s) else
       ↪  (0,0.0)) (zip3 flagl offs sums)
    in tail $ write ind vals $ replicate (length b+1) 0
```

Listing 9: The flat Haskell implementation of matrix-vector multiplication.

The CUDA translation of the code is implemented in `code/MatVecMultHost.cu.h` and `code/MatVecMultKernels.cu.h` and also produce the correct result when run with the data of the handed out test cases.

Task II - Hardware Track Exercises

There are data dependencies between instructions 1-2, 2-3 and 5-6. Control hazards are resolved by the hardware support for branching, that is, stages IF, ID of the pipeline is flushed whenever a branch is taken. We need one NOOP between instructions 1 and 2 as instruction 1 must be in the WB stage when instruction 2 is in the EX stage for the data in R5 to be available. We need another NOOP between instructions 2 and 3 as well as 5 and 6 by the same reasoning.

Task II.1.a

```
1. SEARCH:  LW R5, 0(R3)
            NOOP
2.          SUB R6, R5, R2     ; RAW hazard on R5
            NOOP
3.          BNEZ R6, NOMATCH   ; RAW hazard on R6
4.          ADDI R1, R1, #1
5. NOMATCH: ADDI R3, R3, #4
            NOOP
6.          BNE R4, R3, SEARCH ; RAW hazard on R3
```

Task II.1.b

A single iteration of the loop that is not the last iteration, takes 11 clock cycles to complete when the there is a match. This is depicted in Table 1. As branches are predicted as untaken, the pipeline is flushed whenever the branch at the end of each iteration is taken incurring a 2 cycle penalty. The instructions following immediately after the loop are denoted by i_6 and i_7 in the tables.
In the case when there is no match, the branch is taken, and an iteration takes 12 cycles. The pipeline diagram in Table 4 depicts this for a single iteration.

Task II.1.c

Operand forwarding allows results of ALU operations to be forwarded to the EX stage of subsequent instructions before the results reach the write-back stage. This reduces the number of NOOPs introduced into the pipeline by the hazard detection unit when true dependencies are present. The pipeline diagram for a match and a no match are shown in Tables 3 and 4 and take 9 and 11 cycles, respectively.

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}
`LW`	IF	ID	EX	ME	WB						
`SUB`		IF	ID	ID	EX	ME	WB				
`BNEZ`				IF	ID	ID	EX	ME	WB		
`ADDI`						IF	ID	EX	ME	WB	
`ADDI`							IF	ID	EX	ME	WB
`BNE`	ME	WB						IF	ID	ID	EX
i_6										IF	ID
i_7											IF

Table 1: Pipeline diagram for a non-final iteration in a pipeline with hazard detection. The total number of cycles is 11 when there is a match.

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
`LW`	IF	ID	EX	ME	WB							
`SUB`		IF	ID	ID	EX	ME	WB					
`BNEZ`				IF	ID	ID	EX	ME	WB			
`ADDI`						IF	ID					
`ADDI`							IF	IF	ID	EX	ME	WB
`BNE`	ME	WB							IF	ID	ID	EX
i_6											IF	ID
i_7												IF

Table 2: Pipeline diagram for a non-final iteration in a pipeline with hazard detection. The total number of cycles is 12 when there is a no match.

Task II.1.d

With full forwarding, values that are loaded into registers from memory can be forwarded to the EX stage of a subsequent instruction reading from that register. The total number of cycles are required pr. iteration are the same as in the previous minus one in both cases because the stall between instruction 1 and 2 is eliminated. That is, 8 cycles are needed pr. iteration when there is a match and 10 cycles is needed when there is no match.

Task II.1.e

Unrolling the loop would result in less cycles wasted at the end of each iteration from flushing the pipeline due to the predict-untaken strategy. By using delayed branching, the compiler can move instructions to the branch-delay slots following a branch, as long as the resulting code does not change the semantics of the program. In our case, instruction 5 (`ADDI R3, R3, #4`) does not have any dependencies within the iteration, and can safely be placed after the last branch while preserving semantics. However, a NOOP has to be

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
`LW`	IF	ID	EX	ME	WB				
`SUB`		IF	ID	ID	EX	ME	WB		
`BNEZ`				IF	ID	EX	ME	WB	
`ADDI`					IF	ID	EX	ME	WB
`ADDI`	WB					IF	ID	EX	ME
`BNE`	ME	WB					IF	ID	EX
i_6								IF	ID
i_7									IF

Table 3: Pipeline diagram for a non-final iteration in a pipeline with hazard detection and operand forwarding. The total number of cycles is 9 when there is a match.

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}
`LW`	IF	ID	EX	ME	WB						
`SUB`		IF	ID	ID	EX	ME	WB				
`BNEZ`				IF	ID	EX	ME	WB			
`ADDI`					IF	ID					
`ADDI`	WB						IF	IF	ID	EX	ME
`BNE`	ME	WB							IF	ID	EX
i_6										IF	ID
i_7											IF

Table 4: Pipeline diagram for a non-final iteration in a pipeline with hazard detection and operand forwarding. The total number of cycles is 11 when there is a no match.

placed after the BNE instruction, to prevent the instruction following the ADDI from being introduced into the pipeline.

```
1. SEARCH:   LW R5, 0(R3)
2.           SUB R6, R5, R2     ; RAW hazard on R5
3.           BNEZ R6, NOMATCH   ; RAW hazard on R6
4.           ADDI R1, R1, #1
6. NOMATCH:  BNE R4, R3, SEARCH ; RAW hazard on R3
5.           ADDI R3, R3, #4
             NOOP
```

This results in a better utilization of the pipeline, however one cycle is still wasted at the end of each iteration. By unrolling the loop to include 4 iterations, the compiler might produce something like this:

```
SEARCH:     LW R5, 0(R3)
            SUB R6, R5, R2
            BNEZ R6, NOMATCH_0
            LW R5, 4(R3)
            SUB R6, R5, R2
            ADDI R1, R1, #1
NOMATCH_0:  BNEZ R6, NOMATCH_1
            LW R5, 8(R3)
            SUB R6, R5, R2
            ADDI R1, R1, #1
NOMATCH_1:  BNEZ R6, NOMATCH_2
            LW R5, 12(R3)
            SUB R6, R5, R2
            ADDI R1, R1, #1
NOMATCH_2:  BNEZ R6, NOMATCH_3
            ADDI R3, R3, #4
            ADDI R3, R3, #4
            ADDI R1, R1, #1
NOMATCH_3:  BNE R4, R3, SEARCH
            ADDI R3, R3, #4
            ADDI R3, R3, #4
```

Here, the `ADDI R3, R3, #4`, `LW R5, 4(R3)` and `SUB R6, R5, R2` instructions were moved into the branch-delay slots of the branch-instructions, and the pipeline is fully utilized regardless of whether a match if found or not. This means each iteration takes 21 cycles/4 iterations $= 5.25\frac{\text{cycles}}{\text{iteration}}$.

Task II.2.a

```
LOOP:
  L.V   V1,0(R1), R6  ; p += x[k+j] * y[k+j]
  L.V   V2,0(R2), R6  ; ...
  MUL.V V3,V1,V2      ; ...
  ADD.V V4,V3,V4      ; ...
  ADDI R1,R1,#512     ; x+=64 i.e. (sizeof(double) * 64)
  ADDI R2,R2,#512     ; y+=64 i.e. (sizeof(double) * 64)
  ADDI R3,R3,#64      ; k+=64
  CMP  R3,R4          ; R4 == 1024
  JL   LOOP
```

Task II.2.b

The time before the first pair of operands are loaded is $T^{\text{S}}_{\text{load}} = 30 + 1$. The added one comes from the second load instruction being executed one cycle later than the first one. As vector operations are chained, we can direct the loaded operands into the multiplication FU pipeline as soon the first pair of operands are available. The multiplication FU pipeline has a startup time $T^{\text{S}}_{\text{mult}} = 10$. Next, the results of the multiplication are directed into the addition FU, which has a startup time $T^{\text{S}}_{\text{add}} = 5$. This gives a processing time for 64 elements of:

$$\begin{aligned} T^{\mathrm{S}}_{\mathrm{load}} + T^{\mathrm{S}}_{\mathrm{mult}} + T^{\mathrm{S}}_{\mathrm{add}} + N &= 31 + 10 + 5 + 64 \\ &= 110 \text{ cycles} \end{aligned}$$

As the vector machine is a register-to-register vector machine, the number of cycles used for the computation is proportional to the total number of loop iterations $N_{\mathrm{iter}} = 1024/64 = 16$ iterations, and the total time spent computing the dot product of a vector of size 1024 is 110 cycles/iteration $\cdot$ 16 iterations = 1760 cycles.

Task II.2.c

The matrix product of two 1024×1024 matrices reduces to computing the dot products of all the combinations of rows in the first matrix and columns in the second matrix. That is $1024 \cdot 1024$ dot products of vectors of length 1024 are needed. As the total number of cycles in computing one dot product is 1760, we have:

$$1024^2 \text{ dot products } \cdot 1760 \text{ cycles/dot product} = 1,845,493,760 \text{ cycles}$$

For computing the matrix product.