



**Course Name:** *Computer Architecture Lab*

**Course Number and Section:** **14:332:333:04**

**Experiment:** *6 – GPU Parallelism and Performance*

**Lab Instructor:** *Christos Mitropoulos*

**Date Performed:** *December 1<sup>st</sup>, 2016*

**Date Submitted:** *December 15<sup>th</sup>, 2016*

**Submitted by:** *FAHD HUMAYUN – 168000889 – fh186*

-----For Lab Instructor Use ONLY-----

GRADE: \_\_\_\_\_

COMMENTS:

Electrical and Computer Engineering Department  
School of Engineering  
Rutgers University, Piscataway, NJ 08854

## **Introduction:**

The lab is based on learning basic concepts of the Graphics Processing Unit (GPU), and a simplified GPU Assembly language. PTX a new instruction set architecture has been introduced in the lab, so, the lab assignments are based on applying the concept/understanding of PTX ISA to be able to write and practice a simple program in this simplified language.

GPU is a processor optimized for graphics, video, and visual computing and display. It acts like a supplement of a CPU that does not need to be able to perform all the tasks the CPU does. Its resources are dedicated to graphics and rely on hardware multithreading and high parallelism. Dynamic random access memory chips used for GPUs have wider and higher bandwidth than DRAM chips for CPUs.

The instruction set architecture here is a simplified version of the Tesla architecture PTX ISA. It is a register-based load or store scalar instruction set consisting of floating points, integer, logical, memory access etc.

The instruction type is *opcode.type d,a,b,c*; where, *opcode* can be *add,sub,mov,ld,st,etc.*, *type* is either un-typed, unsigned, signed, and floating point with the number of bits 8,16, 32, 64 (floating point doesn't have 8). '*d*' is destination, '*a*','*b*', and '*c*' are the operands/sources. For a load and store, the instruction format is (*ld for load*).(*memory space*).(*type*) (*destination*),(*source*); and (*st for store*).(*memory space*).(*type*) (*destination*),(*source*); where, memory space can be local, shared, or global.

Predicate registers and *setp* instructions are used for conditional/branch instructions, where predicate registers are 1-bit which can either be 0 (false) or 1 (true). The instruction *setp* sets the bit of the predicate register.

## Assignment - 1:

The assignment is to write a program in PTX ISA of a given pseudo-code.

In the program un-typed 32 bits register *r1* is used as the base address of the array. There are five signed integer 32 bits type registers used for the integer variables, and to predicate registers used – one for the condition of the loop and the other for the *if/else* condition. The predicate register is used to determine whether to execute the *if* or *else* condition, where, *@p* (*p* is the predicate register) is used if the condition is true after executing *setp* instruction, and *@!p* is used if the condition is false, and the value is then stored in array (i.e. in memory)

```
//Assignment_1
//Fahd Humayun - 168000889 - fh186

/*
//cpp code
void main()
{
    int a, b, c, d;
    int e[10];
    for (int i = 0; i < 10; i++)
    {
        if(a<=b)
        {
            a = a + c;
            e[i] = a;
        }
        else
        {
            a = a - d;
            e[i] = a;
        }
    }
}
*/

//Program in ISA described in lab manual
```

```

/*
- data in local memory starting from the address that is stored in register r0,r1
- total number of registers available -> 64 (32-bits)
  registers numbered from 0 to 63
- For a 64-bit data use pair of registers -> for example 64-bit number stored in
  registers r0 and r1, then, refer to this number by r0
-
*/

/*
Registers Used (32-bit):
- r1: (assuming 32-bit address) base address of data in local memory (used for e[i])
- r2: used for the integer variable 'a' (and assuming some data initialized)
- r4: used for the integer variable 'b' (and assuming some data initialized)
- r6: used for the integer variable 'c' (and assuming some data initialized)
- r8: used for the integer variable 'd' (and assuming some data initialized)
- r10: used for the loop counter
- r12: used to check the condition for 'if' and 'else' (true or false) (predicate operand 1-bit)
- r13: used to check the condition of loop (predicate operand 1-bit)
*/

//Variable/register Declarations, assigning values to the variables/registers
.reg    .b32    r1;
.reg    .s32    r2, r4, r6, r8;
.reg    .s32    r10;
.pred   r12, r13;

.local  .s32    e[10]; //array e[10] will have some values declared to it

mov.b32    r1, e;      //move base address of array i.e. address of e[0] into r1
mov.s32    r10, 0;     //r10 = 0 (i = 0)

//Similarly, use 'mov.s32' for registers r2, r4, r6, and r8 to assign values
//if 64-bits is to be used then use '.reg .s64' and 'mov.s64'

...

loop:
    //(i < 10 in c++ for loop, if condition is false then end the loop)
    setp.ge.s32    r13, r10, 10;      //r13 = 1, if r10 >= 10
    @r13 bra end;

    //check condition (a <= b) if true jump to if_ label, else continue from next instruction
    setp.le.s32    r12, r2, r4;      //r12 = 1, if r2 <= r4

    @r12    add.s32    r2, r2, r6 //if (a <= b), then (a = a + c) r2 = r2 + r6
    @!r12    sub.s32    r2, r2, r8; //else (a = a - d) r2 = r2 - r8

    st.local.s32    [r1 + 0], r2;    //store r2 in local memory at address [r1 + 0] (e[i] = a)

    //32-bit integer stored in the local memory so increment the address by 4
    //to store next integer (4*8 = 32)
    //if 64-bit integer then increment by 8 i.e. 8*8 = 64
    add.b32    r1, r1, 4;
    add.s32    r10, r10, 1;    //increment counter, i++

    bra loop;

end:
...

```

## Assignment - 2:

The assignment is similar to the previous one, with the changes in the type memory stores that is 2 bytes which is half word (full word or 32 bits integer is 4 bytes), so, the register for saving the base address is used same as before, while the register that is used to store the data in memory is used as signed 16 bits integer.

```
//Assignment_2
//Fahd Humayun - 168000889 - fh186

/*
//cpp code
void main()
{
    int x[10];
    int neg=0, pos=0, neg_sum=0, pos_sum=0;
    int neg_mean=0, pos_mean=0;

    for (int i = 0; i < 10; i++)
    {
        if(x[i] <= 0)
        {
            neg++;
            neg_sum += x[i];
        }
        else
        {
            pos++;
            pos_sum += x[i];
        }
    }
    neg_mean = neg_sum / neg;
    pos_mean = pos_sum / pos;
}
*/

/*
- data in local memory starting from the address that is stored in register r1

- each memory address can store 2 bytes = 16 bits
*/

/*
Registers Used:
- r1: base address of data in local memory (used for x[i])
- r2: used for the integer variable 'neg'
- r3: used for the integer variable 'pos'
- r4: used for the integer variable 'neg_sum'
- r5: used for the integer variable 'pos_sum'
- r6: used for the integer variable 'neg_mean'
- r7: used for the integer variable 'pos_mean'
- r8: used for loading the data from memory (16-bit)
- r10: used for the loop counter
- r11: used to check the condition of loop
- r12: used to check the condition for 'if' and 'else' (true or false)
*/

/*
r2, r3, r4, r5, r6, r7, r8 (type .s16) will have value 0
*/
```

```

r10 (type .s32) will have a value 0
r13 will have a value 1 (true)
*/

/*
8-bit or 16-bit values may be held directly in 32-bit or 64-bit registers when
being loaded, stored, or converted to other types and sizes.
*/

//Variable/register Declarations, assigning values to the variables/registers
.reg    .b32    r1;
.reg    .s16    r8;

.reg    .s32    r2, r3, r4, r5, r6, r7, r10;
.pred   r11, r12;

.local  .s16    x[10]; //array x[10] will have some values declared to it

mov.b32    r1, x;      //move base address of array i.e. address of x[0] into r1
mov.s32    r2, 0;      //neg  = 0;
mov.s32    r3, 0;      //pos  = 0;
mov.s32    r4, 0;      //neg_sum = 0;
mov.s32    r5, 0;      //pos_sum = 0;
mov.s32    r6, 0;      //neg_mean = 0;
mov.s32    r7, 0;      //pos_mean = 0;
mov.s32    r10, 0;     //i  = 0;

//if 64-bits is to be used then use '.reg .s64' and 'mov.s64'
...

loop:
// (i < 10 in c++ for loop, if condition is false then end the loop)
setp.ge.s32    r11, r10, 10;      //r11 = 1, if r10 >= 10
@r11 bra end;

//load data from memory
ld.local.s16    r8, [r1 + 0];      //r8 = x[i]

//check condition (x[i] <= 0) if true jump to if_label, else continue from next
instruction
setp.le.s64    r12, r8, 0;      //r12 = 1, if r8 <= 0

//16-bit integer stored in the local memory so increment the address by 2
//to store next integer (2*8 = 16)
add.b32    r1, r1, 2;
add.s32    r10, r10, 1;      //increment counter, i++

@r12 bra if_;
@!r12 bra else_;

if_:
// (neg++)
add.s16    r2, r2, 1;      //r2 = r2 + 1

// (neg_sum += x[i])
add.s16    r4, r4, r8;      //r4 = r4 + r8

bra loop;

else_:
// (pos++)
add.s16    r3, r3, 1;      //r3 = r3 + 1

// (pos_sum += x[i])

```

```

    add.s16    r5, r5, r8;    //r4 = r4 + r8

    bra loop;

end:
    //(neg_mean = neg_sum / neg)
    div.s16    r6, r4, r2;    //r6 = r4/r2

    //(pos_mean = pos_sum / pos)
    div.s16    r7, r5, r3;    //r7 = r5/r3

    //for the above division instructions it would be better to use floating point
    //registers as the result stored would not have the fraction part because of the

    //integer type used, but, in the pseudocode given the variables used are integer
    //types, that is why same type used here in the code.

```

### Assignment – 3 (a):

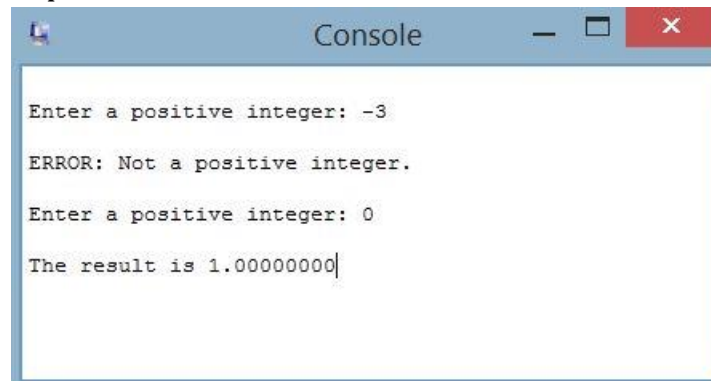
The assignment is based on programming in MIPS and PTX for computing a given geometric series, i.e.

$$x = \sum_{k=0}^n \frac{1}{2^k}$$

32-bit Floating points have been used in the program, and the user is asked to enter a positive value for n. Some of the values for n has been summarized in the table below:

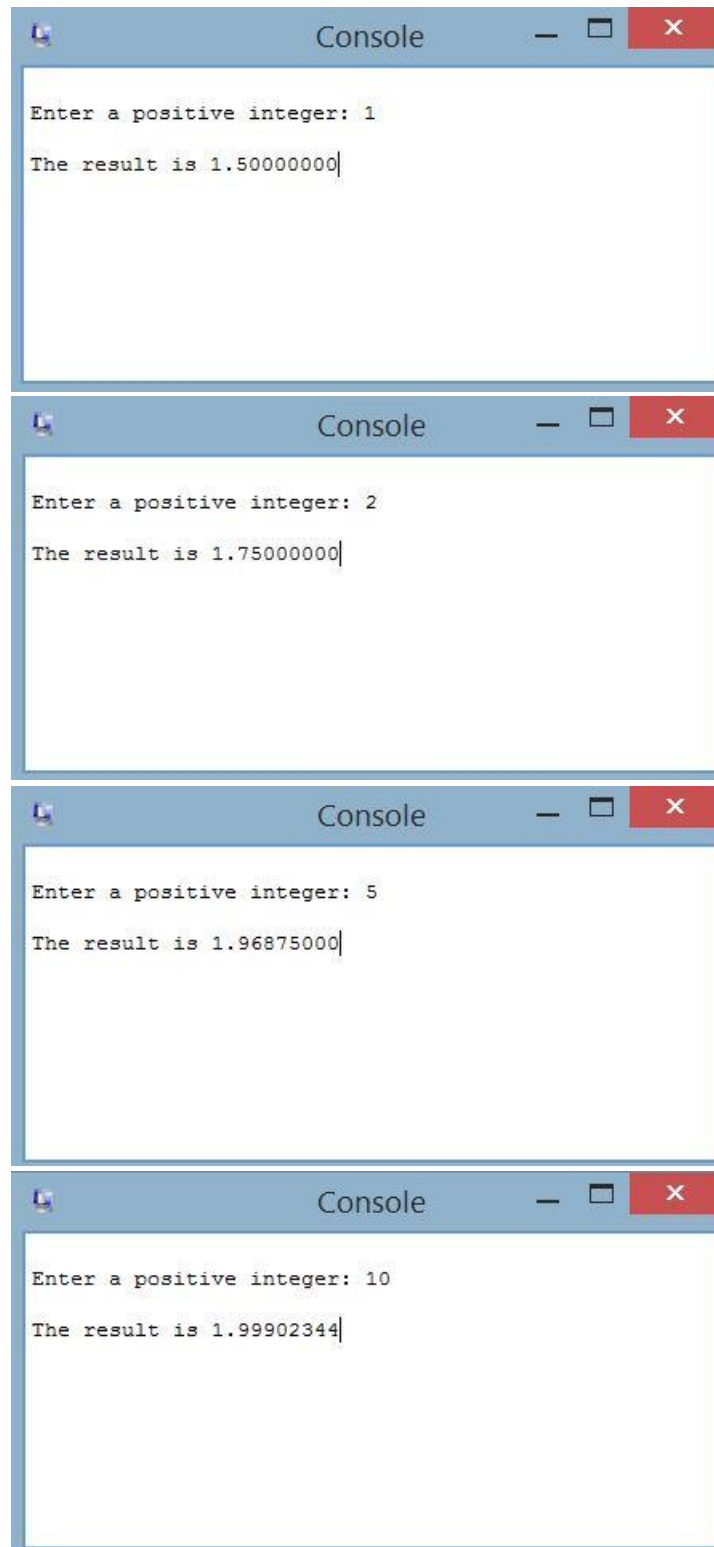
<i><b>n</b></i>	<i><b>x</b></i>
0	$\frac{1}{2^0} = 1$
1	$\frac{1}{2^0} + \frac{1}{2^1} = 1.5$
2	$\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} = 1.75$
5	$\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} = 1.969$
10	$\frac{1}{2^0} + \frac{1}{2^1} + \dots + \frac{1}{2^9} + \frac{1}{2^{10}} = 1.999$

Screenshots of the output:

A screenshot of a console window titled "Console". The window has a blue title bar with standard Windows window controls (minimize, maximize, close). The console text is as follows:

```
Enter a positive integer: -3
ERROR: Not a positive integer.
Enter a positive integer: 0
The result is 1.00000000|
```





## Code in MIPS:

```
#Assignment_3 (MIPS)
#Fahd Humayun - 168000889 - fh186

.data 0x10000000
ask:      .asciiz      "\nEnter a positive integer: "
err:      .asciiz      "\nERROR: Not a positive integer.\n"
result:    .asciiz      "\nThe result is "

.text
.globl main

main:

prompt:
    li      $v0, 4      # print string
    la      $a0, ask     # load address of ask for printing
    syscall

    li      $v0, 5      # read int
    syscall
    move     $t0, $v0    # move integer read to t0

    blt     $t0, $0, error # if integer read negative, jump to error

    li.s     $f9, 1.0    # f9 = 1.0
    li.s     $f2, 0.5    # f2 = 1/2 = 0.5

outer_loop:
    beqz     $t0, end     # if t0 == 0, end outer loop
    move     $t1, $t0     # copy t0 to t1
    li.s     $f3, 1.0    # f3 = 1.0

    inner_loop:
        beqz     $t1, end_iLoop # if t1 == 0, end inner loop

        mul.s     $f3, $f3, $f2 # f3 = f3 * f2 (f3 = f3 * 1/2)

        addi     $t1, $t1, -1   # decrement inner loop counter

        j        inner_loop
```

```

end_iLoop:
    add.s    $f9, $f9, $f3    # f9 = f9 + f3
    addi     $t0, $t0, -1     # decrement outer loop counter
    j        outer_loop

end:
    li       $v0, 4           # print string
    la       $a0, result      # load address of result
    syscall

    li       $v0, 2           # print float
    mov.s    $f12, $f9        # f12 = f9
    syscall

    li       $v0, 10
    syscall

error:
    li       $v0, 4           # print string
    la       $a0, err         # load address of err
    syscall

    j prompt

```

---

## Code in PTX:

```
//Assignment_3 (PTX)
//Fahd Humayun - 168000889 - fh186

//input n in register r1

/*
Registers Used:
- r1: input n (as well as inner loop counter)
- r2: result
- r3: inner loop counter
- r4: used for 2^k
- r5: 1/2
- r10: outer loop condition
- r11: inner loop condition
*/

.reg    .f32    r2,r4,r5;
.reg    .u32    r1, r3;
.pred    r10, r11;

mov.f32    r2, 1;        //r2 = 1
mov.f32    r5, 0.5;      //r5 = 1/2

outer_loop:
    setp.gt.u32 r10, r1, 0; //r10 = 1, r1 > 0

    @!r10    bra    end;

    mov.u32    r3, r1;    //r3 = r1
    mov.f32    r4, 1;    //r4 = 1

    inner_loop:
        sub.u32    r3, r3, 1; //decrement inner loop counter
        setp.gt.u32 r11, r3, 1; //r11 = 1, r3 > 1
        @r11    mul.f32    r4, r4, r5; //r4 = r4 * 1/2
        @r11    bra    inner_loop;
        @!r11    mad.f32    r2, r4, r5, r2;    // r2 = r4*r5 + r2

        sub.u32    r1, r1, 1; //decrement outer loop counter

        bra    outer_loop;

end:
    ...
```

### **Assignment – 3 (b):**

Assuming that each instruction in both programs require 1 cycle, then the number of cycles each program takes to calculate a polynomial of degree  $n = 10$  is below:

*(considering the instructions in the loops only)*

MIPS:

There are 4 instructions in the inner loop, and 7 instructions in the outer loop (7 because the instruction `li $f3, 1.0` is achieved in two instructions in MIPS i.e. `lui $1, 16256` & `mtc1 $1, $f3` (can be seen in QTSpim when the program is loaded).

If  $n = 10$ , the outer loop will run  $n_{outer} = n = 10$  times, which makes the inner loop to run  $n_{outer} + (n_{outer} - 1) + (n_{outer} - 2) + \dots + (n_{outer} - (n_{outer} - 1))$  or in other words this can be given by or simplified to  $\frac{n_{outer}(n_{outer}-1)}{2} = 45$  times the inner loop will run.

Each instruction is 1 cycle, so, total cycles for the inner loop =  $45 * 4 = 180$  cycles and the total cycles for the outer loop =  $7 * 10 = 70$  cycles, and the total number of cycles for both loops =  $180 + 70 = 250$  cycles.

PTX:

There are 5 instructions in the inner loop (the last or 5<sup>th</sup> instruction i.e. `@! r11 mad. f32 r2, r4, r5, r2;` is in the inner loop but it is only executed once in the inner loop when the loop is ending), and 6 instructions in the outer loop.

If  $n = 10$ , the outer loop will run  $n_{outer} = n = 10$  times, which makes the inner loop to run  $n_{outer} + (n_{outer} - 1) + (n_{outer} - 2) + \dots + (n_{outer} - (n_{outer} - 1))$  or in other words this can be given by or simplified to  $\frac{n_{outer}(n_{outer}-1)}{2} = 45$  times the inner loop will run. (the loops in both programs run same number of times)

Each instruction is 1 cycle, so, total cycles for the inner loop =  $(45 - 9) * 4 + 9 = 153$  cycles (the - 9 in parenthesis is because the inner loop counter is first decremented and then the condition is checked in the beginning of the loop, this is because of the instruction `@! r11 mad` as it accounts for the last loop run, and the +9 in the end is because of the same reason that this instruction will run depending upon the number of times - 1 the outer loop runs). The total cycles for the outer loop =  $7 * 10 = 70$  cycles, and the total number of cycles for both loops =  $153 + 70 = 223$  cycles.

## **Assignment – 4:**

The assignment is based on using the bubble sort algorithm to sort an array, and the algorithm is written in both MIPS and PTX.

Array given is {2, -4, 4, 7, 11, 8}.

Screenshot of the unsorted and sorted array in data memory:

Unsorted:

```
User data segment [10000000]..[10040000]
[10000000]..[1000085f] 00000000
[10000860]      0000000002      -4 0000000004 0000000007
[10000870]      0000000011 0000000008 0000000000 0000000000
[10000880]..[1003ffff] 00000000
```

Sorted:

```
User data segment [10000000]..[10040000]
[10000000]..[1000085f] 00000000
[10000860]      -4 0000000002 0000000004 0000000007
[10000870]      0000000008 0000000011 0000000000 0000000000
[10000880]..[1003ffff] 00000000
```

*(considering the instructions in loops only)*

Number of instructions in MIPS = 20 (when run in QTSpim the instructions split to execute can be seen and ignoring the three instructions used to check if swap occurred)

Number of instruction in PTX = 18

## Code in MIPS:

```
#Assignment_4 (MIPS)
#Fahd Humayun - 168000889 - fh186

#Sort the array -> A = {2,-4,4,7,11,8}

#c++ code:
#void BubbleSort(int data_array[], int size_of_array)
#{
#for (int i = 1; i < data_array.size() - 1; i++)
# {
#     swap_flag = false;
#     for (int j = 0; j < data_array.size() - i; j++)
#     {
#         if ( data_array[j] > data_array[j + 1] )
#             swap(data_array, j, j+1);
#     }
#     if ( !swap_flag )
#         break;
# }
#}

.data 0x10000860
Array: .word 2 -4 4 7 11 8

.text
.globl main

main:
    li    $7, 6           #size of array
    li    $8, 1           #counter for outer loop (int i = 1)
    addi   $10, $7, -1     #size of array - 1 (for outer loop condition)
```

```

outer_loop:
    #loop condition
    bge    $8, $10, end_oLoop

    li     $12, 0        #used to check if any swap occurred in a loop
    li     $9, 0         #counter for inner loop (int j = 0)

    sub    $11, $7, $8   #size of array - i (for inner loop condition)
    la     $4, Array     #load base address of array to R4

    inner_loop:
        #loop condition
        bge    $9, $11, end_iLoop

        lw     $13, 0($4)    #R13 = Array[i]
        lw     $14, 4($4)    #R13 = Array[i+1]

        bgt    $13, $14, swap #Array[i] > Array[i+1]

        ret:
        addi   $4, $4, 4     #increment address
        addi   $9, $9, 1     #j++
        j      inner_loop

    end_iLoop:
        beqz   $12, end_oLoop #if no swap occurred
        addi   $8, $8, 1     #i++
        j      outer_loop

    swap:
        sw     $14, 0($4)    #Array[i] = Array[i+1]
        sw     $13, 4($4)    #Array[i+1] = Array[i]
        addi   $12, $12, 1   #set to 1 as swap occurred

end_oLoop:
    li     $2, 10
    syscall

```



Code in PTX:

```
//Assignment_4 (PTX ISA)
//Fahd Humayun - 168000889 - fh186

//Sort the array -> A = {2,-4,4,7,11,8}

/*
Registers Used:
- r1: base address of array in local memory (used for Array[i])
- r2: Array[i]
- r3: Array[i + 1]
- r4: size of array
- r5: outer loop counter (i)
- r6: outer loop condition
- r7: inner loop counter (j)
- r8: inner loop condition
- r10: predicate for outer loop
- r11: predicate for inner loop
- r12: predicate for if condition
*/

//array declaration
.local .s32 Array[] = {2,-4,4,7,11,8};

.reg    .b32    r1;
.reg    .s32    r2,r3,r4,r5,r6,r7,r8;

.pred    r10,r11,r12

mov.s32    r4, 6;        //size of array = 6
mov.s32    r5, 1;        //i = 1
sub.s32    r6, r4, 1;    //size of array - 1

outer_loop:
    setp.lt.s32    r10, r5, r4;        //r10 = 1, i < (size of array - 1)

    @!r10    bra    end_oLoop;

    mov.b32    r1, Array; //move base address of array into r1
    mov.s32    r7, 0;     //j = 0
    sub.s32    r8, r4, r5 //size of array - i
```

```

inner_loop:
    setp.lt.s32    r11, r7, r8;    //r11 = 1, j < (size of array - i)

    @!r11    bra    end_iLoop;

    ld.local.s32   r2,[r1 + 0];    //Array[i]
    ld.local.s32   r3,[r1 + 4];    //Array[i+1]

    add.s32        r1, r1, 4;       //increment address
    add.s32        r7, r7, 1;       //j++

    //if(Array[i] > Array[i+1])
    setp.gt.s32    r12, r2, r3;    //r12 = 1, r2 > r3

    @!r12    bra    inner_loop;

    st.local.s32   [r1 + 0], r3    //Array[i] = Array[i+1]
    st.local.s32   [r1 + 4], r2    //Array[i+1] = Array[i]

    bra    inner_loop;

end_iLoop:
    add.s32        r5, r5, 1;       //i++
    bra    outer_loop;

end_oLoop:
    ...

```

## **Conclusion:**

In the lab, programs were written in MIPS and then written in PTX to observe the simplification and reduction of cycles and instructions from MIPS to PTX. The first assignment was a basic code written in PTX to get familiar with the language and to see how predicate registers and instructions can simplify the code compared to MIPS, the second assignment was similar to the first one this time with more than one different instruction to be executed for the conditions of if/else. Load and store instructions were used in the assignments, which is similar to the way MIPS is with the differences in the instruction format discussed in the introduction section above. Third assignment was to write a program in both MIPS and PTX to compute a geometric series and then analyze the number of instructions in both programs and calculate the number of cycles each program would take – PTX was simplified version and could be seen that it took less number of instructions and specially number of cycles to compute the geometric series. The last assignment was to sort an array using the bubble sort algorithm, the bubble sort algorithm compares two adjacent elements in the array and then swap the elements if the next element is smaller than the current element (to sort in ascending order), there was not a lot of difference in the number of instructions for the program written in MIPS and PTX.