CSCE 4610: Computer Architecture

**Final Exam: May 9 10:30-12:30**
**Term projects Due: May 4, 2017.**

What to include in project report

HW #7 Due April 20, 2017
3.14 part a and b
3.18, 3.31

Two Surveys to Complete:
1. UNT SPOT – evaluate the class
2. CSE exit survey – you need to let us know how well you learned the course objectives

---

CSCE 4610: Computer Architecture

Review

Multithreading and Simultaneous multithreading

A simple model for multithreading

\( N = \) number of threads, \( N_{\text{saturated}} = \) enough threads to saturate

\( R = \) Average time between thread switches due to long latency operations

\( L = \) average time for long latency operation

\( C = \) context switching overhead

\[
\text{Speedup} = S_N = \frac{U_N}{U_1} = \begin{cases} \frac{N}{R + L} & \text{if } N < N_{\text{saturated}} \\ R + C & \text{Otherwise} \end{cases}
\]

\[
\text{Speedup} = S_N = \frac{U_N}{U_1} = \begin{cases} \frac{N}{1 + m \times L} & \text{if } N < N_{\text{saturated}} \\ \frac{1 + m \times C}{L} & \text{Otherwise} \end{cases}
\]

If we context switch on a cache miss

\( R = 1/\text{miss rate} \)

\( L = \text{miss penalty} \)
CSCE 4610: Computer Architecture

If we use threads from the same program (shared memory) we need
Mutual exclusion (or locks) to modify shared data
Barriers (or rendezvous) to coordinate thread execution

Parallel Processing Architectures

Flynn’s Classification

SISD
SIMD or SPMD
MISD
MIMD

Array Processors versus Vector Processors

Array processors execute the same instruction on different data elements
Vector processors have special vector instructions
may or may not have multiple functional units
Better use of pipelined arithmetic unit

Cray Vector Unit

Vector Registers (64 words per register)
VADD Vi, Vj, Vk
Additional registers for specifying vector length and mask
Need to load vector elements into these vector registers
Vector Code Example

C code

for (i=0; i<64; i++)
C[i] = A[i] + B[i];

Scalar Code
LI R4, 64
loop:
L.D F0, 0(R1)
L.D F2, 0(R2)
ADD.D F4, F2, F0
S.D F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 1
BNEZ R4, loop

Vector Code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3
Also let us look at how to use the vector instructions to compute the following vector operation

\[ Y = a \times X + Y \]

where \( X \) and \( Y \) are arrays (vectors)

In C

```c
for (i=0; i<n; i++) { Y[i] = a*X[i] + Y[i]; }
```

In standard MIPS instructions we use a loop

1. \( \text{LD} \) \( F0, a \) for scalar load
2. \( \text{DADDIU} \) \( R4, Rx, \#512 \) for scalar load
3. \( \text{MULD} \) \( F2, F2, F0 \) for \( a \times X[i] \)
4. \( \text{ADD} \) \( F4, F4, F2 \) for \( a \times X[i] + Y[i] \)
5. \( \text{SD} \) \( 0(Ry), F4 \) for store
6. \( \text{BNEZ} \) \( R20, \text{Loop} \) for check bounds

On a vector processor or using VMIPS instructions

1. \( \text{LD} \) \( V0, \#512 \) for scalar load
2. \( \text{DADDIU} \) \( V1, V0, \#8 \) for scalar load
3. \( \text{MULB} \) \( V2, V1, F0 \) for \( a \times X \)
4. \( \text{ADD} \) \( V3, V2, V3 \) for \( a \times X + Y \)
5. \( \text{SD} \) \( V4, Ry \) for store
6. \( \text{BNEZ} \) \( R20, \text{r-loop} \) for check bounds

CSCE 4610: Computer Architecture

Text uses instructions similar to Cray but calls them Vector MIPS (VMIPS) -- see page 266
CSCE 4610: Computer Architecture

Note that all functional units are pipelined
  Vector-Vector units
  Vector-Scalar unit
  Load and Store Vector Units

In most vector processors, we can “forward” data from one vector operation to another.

For example we can forward the data from MULSV to ADDVV
Or, as we compute a*X[i] this can be forwarded so that we can do Y[i]+a*X[i]
Such forwarding is known as “Chaining” – we are connecting the output of one pipelined functional unit to another.

How do we estimate the performance of Vector processors?

Convoys and Chimes

CSCE 4610: Computer Architecture

Convey is a set of vector instructions that can be issued together
A vector instruction cannot be issued if there is a structural hazard
We will assume that data dependencies (RAW) can be handled with chaining
And other data hazards with register renaming.

The time it takes to complete a convoy is known as a chime.

Consider the example we have seen already (and ignore the scalar load)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, a</td>
<td>load scalar a into F0</td>
</tr>
<tr>
<td>LV V1, Rx</td>
<td>load vector X into V1</td>
</tr>
<tr>
<td>MULSV V2, V1, F0</td>
<td>compute a*X</td>
</tr>
<tr>
<td>LV V3, Ry</td>
<td>load vector Y into V3</td>
</tr>
<tr>
<td>ADDVV V4, V2, V3</td>
<td>compute a*X + Y</td>
</tr>
<tr>
<td>SV V4, Ry</td>
<td>store Y</td>
</tr>
</tbody>
</table>

We have 3 convoys (ignoring scalar load)

1. LV V1, Rx  MULSV V2, V1, F0
2. LV V3, Ry  ADDVV V4, V2, V3
3. SV
CSCE 4610: Computer Architecture

Since we have only one Load/Store vector unit, we have structural hazards.

The code takes 3 chimes.
How many operations did we complete (in total) = 5
How many floating point operations = 2

Or we achieved 1.5 chimes per FLOP.

Actually the number of actual operations we can complete depends on the number of vector elements in a vector register

Another Example
Consider the problem 4.9 on page 337

Consider the following code, which multiplies two vectors that contain single-precision complex values:

```c
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];
}
```

We are dealing with complex numbers – hence real and imaginary parts of a number. We can only do 64 elements at a time.

The clock frequency is 700Mhz.
Load/store has a startup overhead of 15 cycles; multiply latency of 8 cycles, add/sub 5 cycles

a). What is the arithmetic intensity – how many arithmetic operations
    there are a total of 6 arithmetic operations in the loop body
    And the total number of operations is also 6
    So arithmetic intensity = 6/6 =1
b). Show the VMPI code

Assume MVL = 64:

Note the loop executes 300 times 300 = 44 + 64*4
So the code does 44 first

```
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
    mulv.s $v5,v1,v3  # a+re*b_re
    lv  $v2,a_im+$r1  # load a_im
    lv $v4,b_im+$r1  # load b_im
    mulv.s $v5,v2,v4  # a+im*b_im
    subv.s $v5,v5,v6  # a+re*b_re - a+im*b_im
    sv $v5,c_re+$r1  # store c_re
    mulv.s $v5,v1,v4  # a+re*b_im
    mulv.s $v6,v2,v3  # a+im*b_re
    addv.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,#352  # first iteration, increment by 44*8
    j loop  # guaranteed next iteration else:
else:
    addi $r1,$r1,#512  # not first iteration 64*8
    li  $VL, 64  # increment by 512
    # set vector length
skip:
    blt $r1, 2400,loop  # next iteration?
```

---

C: How many chimes are required, assuming chaining

```
# load a_re
lv 
# load b_re
lv
mulv.s lv  # a_re * b_re and , load a_im
# load b_im,  and a_im*b_im
lv mulv.s sv  # subtract and store c _re
subv.s sv
# a_re*b_im, load next a_re vector
mulv.s lv  # a_im*b_re, load next b_re vector
addv.s sv  # add and store c_im
```

8 chimes

---

D. How many cycles (ignore first iteration)

8 chimes, and each chime handles 64 elements. We have 4 loads and 2 stores
4 multiply and 2 add/sub
8*64 + 15 (load/store)*6 + 8 (multiply)*4 + 5 (add/sub)*2 = 644

We are producing 2 results per iteration or 128 results since there are 64 elements
cycles per result = 644/128 = approximately 5 cycles
One more example and the concept of reduction

Consider a simple vector dot product $X \cdot Y$

$$\text{sum} = 0.0;$$

for $(i=0; i<n; i++) \text{sum} = \text{sum} + X[i]^*Y[i];$

```
SUB.D   F0, F0, F0  ; assume n = 64
LI      $VL, 64      ; load Array X into V1
LV      $V1, R1      ; load Array Y into V2
LV      $V2, R2      ;
MULVV   $V3, $V1, $V2 ; $V3 = $V1*$V2
VREDPLUS F0, $V3     ; F0= sum of elements of $V3
ST      0(R3), F0    ; store Result
```

Reduction is a very important operation in vector arithmetic

Consider now how we extend this for Matrix Vector product

$Y = A \cdot X$ (A is a matrix, X and Y are) vectors or one dimensional arrays

a). Can we use multiple functional units to improve the performance of a vector operation?
   for example multiple vector LD/ST units
   multiple floating point ADD and MULT units

b). What if the array size is not a multiple of 64?
   We already have seen this using Vector length

c). How to handle if conditions that require vector operations on some array elements only?
   may be only certain elements of the vector are involved
   for $(i=0; i<n; i++)$
   \[ \text{if} \ (A[i] <> 0) \ C[i] = B[i]/C[i]; \]

d). How to supply the memory bandwidth to load vector registers?

e). What if the array elements are not adjacent?
CSCE 4610: Computer Architecture

a). Can we use multiple functional units to improve the performance of a vector operation?
Yes – text calls this multiple lanes -- each functional unit operates on a set of vector elements
See page 272-273

CSCE 4610: Computer Architecture

b). What if the array size is not a multiple of 64?
Use a vector length register to indicate the number of elements less than 64
So if you have 132 elements, we first operate on 64 elements two times and then set VL to 4

c). How to handle if conditions that require vector operations on some array elements only?
Use vector mask register – zero’s for elements not involved in the operation
use S—VV or S—VS instructions
CSCE 4610: Computer Architecture

d). How to supply the memory bandwidth to load vector registers?

Use multiple banks – and use interleaved addressing to spread different element of the vector to different banks
Consider the example on page 277.
  32 vector functional units (or lanes), each capable of 4 loads and 2 stores per cycle
  Total load/stores = 32*6 = 192
Processor clock is 2.167ns (461 MHz?) while memory clock is 15ns
  or memory = approximately 7 processor cycles

So, to keep up with processor requests we need 192*7 = 1344 banks!
  So, you may need to slow the processor clock, or use 3D DRAMs

c). What if the array elements are not adjacent?
This can happen either because how the array is accessed
For example the inner loop of a matrix multiplications looks like
  for (k = 0; k<n; k++) {Z[i][j] = Z[i][j] + X[j][k] * Y[k][j];}
See page 278

CSCE 4610: Computer Architecture

In programming language C matrices are stored row – major; so elements of X in the inner loop are adjacent to each other while those of Y are not.

If we have 100*100 matrices, and each element is 8 bytes, the elements of Y for inner loop are separated by 800 bytes (while those of X are separated by only 8 bytes)
  this is called a stride

We can use hardware to learn the distance of the next element (stride) and prefetch data using the stride (strided prefetch).

But for our purpose here, we can indicate the stride for loading vectors
  see LVWS V1, (R1, R2) Load V1 with vector with stride R2
  so V1i gets data from address (R1+0*R2)
  V1i gets data from address (R1+1*R2)….

Likewise we can store a vector result in an array with a stride
  SVWS (R1,R2), V1
### CSCE 4610: Computer Architecture

There is another possibility why vector elements accessed are not adjacent. "sparse matrices" – lots of zeros in a matrix.

Consider some common Engineering or scientific applications that use the Finite Difference Method.

In this method, we model a physical system with a grid of finite points.

In order to solve the partial differential equation for any point \( u \), given by

\[
(\partial^2 u / \partial X^2) + (\partial^2 u / \partial Y^2) = f
\]

This for two dimensional grids. We can extend this to 3 dimensional grids also.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV. D</td>
<td>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>ADDV. D</td>
<td>V1, V2, V0</td>
<td>Add V0 to each element of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBV. D</td>
<td>V1, V2, V3</td>
<td>Subtract elements of V3 from V1, then put each result in V1.</td>
</tr>
<tr>
<td>SUBV. D</td>
<td>V1, V2, V0</td>
<td>Subtract V0 from elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>MULV. D</td>
<td>V1, V2, V3</td>
<td>Multiply elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>MULV. D</td>
<td>V1, V2, V0</td>
<td>Multiply each element of V2 by V0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV. D</td>
<td>V1, V2, V3</td>
<td>Divide elements of V2 by V3, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV. D</td>
<td>V1, V2, V0</td>
<td>Divide V0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
</tr>
<tr>
<td>LK</td>
<td>V1, V2</td>
<td>Load V1 from address at R1 with stride in R2 (i.e., R1 + s * R2).</td>
</tr>
<tr>
<td>V0</td>
<td>(R1, R2), V1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + s * R2).</td>
</tr>
<tr>
<td>V1</td>
<td>V1, R1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + s * R2).</td>
</tr>
<tr>
<td>CSX</td>
<td>V1, R1</td>
<td>Create an index vector by storing the values 0, 1, 2, 3, ..., 63 * R1 into V1.</td>
</tr>
<tr>
<td>CSX</td>
<td>V1, R1</td>
<td>Create an index vector by storing the values 0, 1, 2, 3, ..., 63 * R1 into V1.</td>
</tr>
<tr>
<td>S—VY. D</td>
<td>V1, R2</td>
<td>Compare the elements (0, 6, 12, 18, 24, 30, 36, 40) in V2 and V1.</td>
</tr>
<tr>
<td>S—VZ. D</td>
<td>V1, R0</td>
<td>Divide elements of V0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>S—VZ. D</td>
<td>V1, R0</td>
<td>Divide elements of V0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>S—VZ. D</td>
<td>V1, R0</td>
<td>Divide elements of V0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>PDP</td>
<td>R1, VM</td>
<td>Count the 1s in vector mask register VM and store count in R1.</td>
</tr>
<tr>
<td>CMP</td>
<td>V0, R1</td>
<td>Set the vector mask register to all 1s.</td>
</tr>
<tr>
<td>MVC</td>
<td>V0, R1</td>
<td>Move contents of V0 to vector-length register V1.</td>
</tr>
<tr>
<td>MVL</td>
<td>R1, V1</td>
<td>Move the contents of vector-length register V1 to R1.</td>
</tr>
<tr>
<td>MVR</td>
<td>VM, R0</td>
<td>Move contents of V0 to vector-mask register VM.</td>
</tr>
<tr>
<td>MVR</td>
<td>V0, VM</td>
<td>Move contents of vector-mask register VM to V0.</td>
</tr>
</tbody>
</table>

Here \( h \) is the distance between grid points.

\[
h^2 f = x[i+1,j] + x[i-1,j] + x[i,j+1] + x[i,j-1] - 4 x[i,j]
\]

CSCE 4610  April 18, 2017
CSCE 4610: Computer Architecture

If you stored the coefficients of X’s that are used in the above computation it may look like

Note that the off-the-main diagonal elements are at a distance dependent on how the grid points are numbered.

CSCE 4610: Computer Architecture

To save space, we only want to represent non-zero elements.
There are several ways of doing this, but let us look at one simple example

Coordinate form

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>

`VAL` | 8  | 6  | 12 | 1  | 15 | 14 | 9  | 2  | 3  | 5  | 15 | 4  | 11 | 7  | 10 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Here we have 3 arrays. The first array VAL contains only the non zero elements
The other two arrays I and J define the IJ subscripts for the values
Note: we can store the values in any order.
CSCE 4610: Computer Architecture

In other words, the elements we are accessing are indirectly addressed using I and J arrays.

Consider the code example on page 279

\[
\text{for } (i=0; i<n; i++)
A[K[i]] = A[K[i]] + C[M[i]]
\]

We have two arrays \( K \) and \( M \) and they store the “indexes” of A and C.

Many vector processors have instructions to access memory using the indirect pointers. They are called “gather-scatter” instructions.

note strided Load and Store are also called Gather-scatter instructions

Consider the example of vector code on page 280 for the above C segment

\[
\begin{align*}
\text{LV } & V_k, R_k : \text{load } K \text{ vector} \\
\text{LVI } & V_a, R_a, V_k : \text{Load } A \text{ using } K \text{ or } (R_a + V_k) \text{ as address} \\
\text{LV } & V_m, R_m : \text{load } M \text{ vector} \\
\text{LVI } & V_c, R_c, V_m : \text{load } C \text{ using } M \\
\text{ADDVV } & V_a, V_a, V_c : \text{vector add} \\
\text{SVI } & V_a, R_a, V_k : \text{store } A \text{ using } K
\end{align*}
\]
CSCE 4610: Computer Architecture

Remember vector machines use vector instructions
We can think of them as SIMD instructions
*Apply the same instruction on different data items*
However Vector machines rely on pipelining
*SIMD processors use independent parallel execution*

First attempt at SIMD implementation in CPU’s to aid in graphical processing
graphical processing relies on pixel level parallelism
and in many cases the operands are small (8, 16 bits)

Instruction extensions for multimedia by Intel of MMX

If you have a 64-bit ALU, can we treat that as a unit that can perform arithmetic on
1 pair of 64-bit operands
2 pairs of 32-bit operands
4 pairs of 16-bit operands
8 pairs of 8-bit operands

The idea is to think of ALUs formed with independent units

Initially, Intel used floating point unit and floating point registers
A 64-bit register is viewed as
one 64-bit, two 32-bit, four 16-bit or eight 8-bit registers

This has several limitations
You have to fix the number of parallel operations with the instruction
No masking like vector processors
No strided load/store (or scatter/gather addressing)

Also, conflict between floating point instructions and MMX instruction using the same floating point registers
CSCE 4610: Computer Architecture

Later Intel introduced Streaming MIMD extension (SSE) separate 128-bit registers (not use floating point registers)
new load/store to these registers

More recent versions include Advanced Vector Extensions (AVX)
use 256-bit or 512-bit registers

increasing the SIMD width

<table>
<thead>
<tr>
<th>AVX Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADDPD</td>
<td>Add four packed double-precision operands</td>
</tr>
<tr>
<td>VSUBPD</td>
<td>Subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VMULPD</td>
<td>Multiply four packed double-precision operands</td>
</tr>
<tr>
<td>VDIVPD</td>
<td>Divide four packed double-precision operands</td>
</tr>
<tr>
<td>VPADDPD</td>
<td>Multiply and add four packed double-precision operands</td>
</tr>
<tr>
<td>VPADDSUBPD</td>
<td>Multiply and subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VCMPXx</td>
<td>Compare four packed double-precision operands for EQ, NE, LT, LE, GT, GE, etc.</td>
</tr>
<tr>
<td>VPSRRCPPD</td>
<td>More aligned four packed double-precision operands</td>
</tr>
<tr>
<td>VRUDORDS4PS1D</td>
<td>Broadcast one double-precision operand to four locations in a 256-bit register</td>
</tr>
</tbody>
</table>

Figure 4.9 AVX instructions for x86 architecture useful in double-precision floating-point programs. Packed-double for 512-bit AVX means four 64-bit operands executed in SIMD mode. As the width increases with AVX, it is

CSCE 4610 April 18, 2017 29

---

CSCE 4610: Computer Architecture

An example from page 284. This example assumes that we have AVX instructions added to MIPS instructions (as shown by AD with instructions)

The code computes \( a^*X + Y \) (DAXPY) where \( X \) and \( Y \) are vectors

We complete 4 iterations at a time

```
L.D F0,a ;load scalar a
MOV F1,F0 ;copy a into F1 for SIMD MUL
MOV F2,F0 ;copy a into F2 for SIMD MUL
MOV F3,F0 ;copy a into F3 for SIMD MUL

DADDU R4,Rx,#12 ;last address to load

Loop: L.4D F4,0(Rs) ;load X[i], X[i+1], X[i+2], X[i+3]
        L.4D F5,0(Rs) ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        MUL.4D F4,F2,F0 ;ax[1], ax[1+1], ax[1+2], ax[1+3]
        ADD.4D F8,0(Ra) ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]

DADDU Rx,Rx,#32 ;increment index to X
DADDU Ry,Ry,#32 ;increment index to Y
SUBU R20,R4,Rx ;compute bound
BNEZ R20,Loop ;check if done
```

Making copies of \( a^*X \) so that we can do \( a^*X \) on 4 elements of \( X \)

Note that this actually means F1*F0; F2*F0; F3*F0; F4*F0
This means F8+2F4; F9+2F5; F10+2F6; F11+2F7

CSCE 4610 April 18, 2017 30
CSCE 4610: Computer Architecture

GPU architecture
most implementations are similar to that of NVIDIA processors

Consider the following diagram

Note we have 3 dimensional parallelism
Wave front (or SM)
Work group (thread block)
Grid

Another view of GPU architectures
Page 293

Each SIMD thread is a streaming processor
32-wide data items at a time

Each thread block has 16 SM processors
A grid has 16 thread blocks

So we have a total of 8192 threads
Or process that many loop iterations at a time
Another view of a SM

Let us think of how a loop is allocated

A loop is divided into slices and each slice is allocated to a different SM.

Multiple iterations per SM more than its width

Warp scheduler schedules ready iterations (as a group)

Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has 48 independent threads of SIMD instructions that is schedules with a table of 48 PCs.
CSCE 4610: Computer Architecture

Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance

How do we achieve this copying of data

First we need to allocate memory both on CPU (use normal malloc) and on GPU

\[
\text{cudaMalloc((void **)&d_a, size);}
\]
\[
\text{cudaMalloc((void **)&d_b, size);}
\]
\[
\text{cudaMalloc((void **)&d_c, size);}
\]

Here we are allocating 3 arrays; two inputs and one output

This taken from an example that adds two arrays

\[
\text{for (i=0; i<size; i++)} \{ \text{c[i] = a[i]+b[i];} \}
\]

Then we need to copy data from CPU memory into GPU memory

\[
\text{cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);}
\]
\[
\text{cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);}
\]

Likewise GPU can copy data from GPU memory to CPU

\[
\text{cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);}
\]
CSCE 4610: Computer Architecture

To launch a kernel (code or loop or function) to execute on GPU

\[ \text{add} \langle< \langle N/\text{THREADS\_PER\_BLOCK} \rangle, \text{THREADS\_PER\_BLOCK} \rangle \rangle(d_a, d_b, d_c); \]

Here the code block to be executed by GPU is “add”

Then you specify how many thread blocks and how many threads per block

The actual code for “add” must use the dimensions of the arrays, threads per block and block id to determine which indexes a particular thread will work on

```c
__global__ void add(int *a, int *b, int *c, int n)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

CSCE 4610: Computer Architecture

Consider how we can think of mapping a program into these architectural concepts

// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n + 255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

Assign 256 iterations per thread block
Each thread block in hardware has 16 Streaming multiprocessors
So each SM will get 16 iterations

Need to compute indexes assigned to each SM.
SM schedules those iterations that are ready to its SMID cores