CSCE 4610: Computer Architecture

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

Help Session Friday May 5 1:30pm  F 223

Optional HW #8 Due April 27, 2017
4.1 and 4.2

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

Review
Vector processors (pipelined arithmetic units and use vector registers)
SIMD (array processors)
SSE and AVX instructions
GPUs

---

CSCE 4610: Computer Architecture

Convoys and Chimes
Chain pipelines

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

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB.D</td>
<td>F0, F0, F0</td>
</tr>
<tr>
<td>LV</td>
<td>$SVL, 64$; assume $n = 64$</td>
</tr>
<tr>
<td>LV</td>
<td>$SV1, R1$; load Array X into V1</td>
</tr>
<tr>
<td>LV</td>
<td>$SV2, R2$; load Array Y into V2</td>
</tr>
<tr>
<td>MULVV</td>
<td>$SV3, SV1, SV2$; $SV3 = SV1 \cdot SV2$</td>
</tr>
<tr>
<td>VREDPLUS</td>
<td>F0, SV3; F0= sum of elements of $SV3$</td>
</tr>
<tr>
<td>ST</td>
<td>0(R3), F0; store Result</td>
</tr>
</tbody>
</table>

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

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 $S \rightarrow VV$ or $S \rightarrow VS$ instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV1</td>
<td>$R1, (R2+I)$; Load $V1$ with vector whose elements are at $R2 + k \cdot I$ (i.e., $V2$ is an index).</td>
</tr>
<tr>
<td>SV1</td>
<td>$(R1+V2), V1$; Store $V1$ to vector whose elements are at $R1 + k \cdot V2$ (i.e., $V2$ is an index).</td>
</tr>
<tr>
<td>CEI</td>
<td>$V1, R1$; Create an index vector by storing the values 0, 1, $R1$, $R1+1$, ..., $R1+V2$ into $V2$.</td>
</tr>
<tr>
<td>S--VV_D</td>
<td>$V1, V2$; Compare the elements (0-R0, R1-R7, R8-R15) of $V1$ and $V2$. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (MR). The instruction S--VV_D performs the same compare but using a scalar value as one operand.</td>
</tr>
<tr>
<td>S--VS_D</td>
<td>$V1, V2$; Compare the elements (0-R0, R1-R7, R8-R15) of $V1$ and $V2$. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (MR). The instruction S--VS_D performs the same compare but using a scalar value as one operand.</td>
</tr>
<tr>
<td>PDP</td>
<td>R1, MR; Count the 1s in vector-mask register MR and store count in R1.</td>
</tr>
<tr>
<td>CP1</td>
<td>Set the vector-mask register to all 1s.</td>
</tr>
<tr>
<td>HVC1</td>
<td>VLR, R1; Move contents of V1 to vector-length register F1.</td>
</tr>
<tr>
<td>HVC1</td>
<td>R1, VLR; Move the contents of vector-length register V1 to R1.</td>
</tr>
<tr>
<td>HVC1</td>
<td>V0, F0; Move contents of F0 to vector-mask register V0.</td>
</tr>
<tr>
<td>HVC1</td>
<td>F0, V0; Move contents of vector-mask register F0 to V0.</td>
</tr>
</tbody>
</table>

### Example

For the inner loop of a matrix multiplications looks like

for ($k = 0; k < n; k++)$ \(Z[i][j] = Z[i][j] + X[i][k] \cdot Y[k][j];\)

See page 278
CSCE 4610: Computer Architecture

**Strided load and store:** Store vector registers with data from memory locations separated by a fixed distance

\[ \text{LVWS } V_1, (R1, R2) \text{ Load } V_1 \text{ with vector with stride } R2 \]
\[ \text{so } V_{1_0} \text{ gets data from address } (R1+0*R2) \]
\[ V_{1_1} \text{ gets data from address } (R1+1*R2) \ldots \]

Likewise we can store a vector result in an array with a stride

\[ \text{SVWS } (R1,R2), V1 \]

Loading vectors using indirect addresses

Indexes of an array are contained in another vector

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

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

---

CSCE 4610: Computer Architecture

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

We introduce new instructions called Load Vectors Indirect

\[ \text{LV } V_k, R_k \text{ : load } K \text{ vector} \]
\[ \text{LVI } V_a, R_a, V_k \text{ : Load } A \text{ using } K \text{ or (Ra+Vk) as address} \]
\[ \text{LV } V_m, R_m \text{ : load } M \text{ vector} \]
\[ \text{LVI } V_c, R_c, V_m \text{ : load } C \text{ using } M \]
\[ \text{ADDVV } V_a, V_a, V_c \text{ : vector add} \]
\[ \text{SVI } V_a, R_a, V_k \text{ : store } A \text{ using } K \]

---
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

AVX Instruction Description

<table>
<thead>
<tr>
<th>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>VFADDPD</td>
<td>Multiply and add four packed double-precision operands</td>
</tr>
<tr>
<td>VFMSUBPD</td>
<td>Multiply and subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VCMPPX</td>
<td>Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...</td>
</tr>
<tr>
<td>VMULXPD</td>
<td>More aligned four packed double-precision operands</td>
</tr>
<tr>
<td>VINSERTPS</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 64-bit AVX means four 64-bit operands executed in SIM mode. As the width increases with AVX, it is

CSCE 4610 April 20, 2017

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 \( a^\times X + Y \) (DAXPY) where \( X \) and \( Y \) are vectors
We complete 4 iterations at a time

\[
\text{L.D} \quad F0,a \quad \text{;load scalar a} \\
\text{MOV} \quad F1,F0 \quad \text{;copy a into F1 for SIMD MUL} \\
\text{MOV} \quad F2,F0 \quad \text{;copy a into F2 for SIMD MUL} \\
\text{MOV} \quad F3,F0 \quad \text{;copy a into F3 for SIMD MUL} \\
\text{ADD} \quad R4,Rx,#12 \quad \text{;last address to load} \\
\text{Loop:} \quad \text{L.4D} \quad F4,0(Rx) \quad \text{;load } X[i], X[i+1], X[i+2], X[i+3] \\
\text{MUL} \quad F4,F2,F0 \quad \text{;multiply A \times X[i], A \times X[i+1], A \times X[i+2], A \times X[i+3]} \\
\text{BR} \quad F2,0(Ry) \quad \text{;load } Y[i], Y[i+1], Y[i+2], Y[i+3] \\
\text{ADD} \quad F8,F0,F4 \quad \text{;add A \times X[i] + Y[i]} \\
\text{S.4D} \quad F8,0(Ry) \quad \text{;store into } Y[i], Y[i+1], Y[i+2], Y[i+3] \\
\text{DADDU} \quad Rx,Rx,#32 \quad \text{;increment index to X} \\
\text{DADDU} \quad Ry,Ry,#32 \quad \text{;increment index to Y} \\
\text{SUBU} \quad R20,R4,Rx \quad \text{;compute bound} \\
\text{BNEZ} \quad R20,Loop \quad \text{;check if done} \\
\]

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

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

CSCE 4610 April 20, 2017
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
How do we achieve this copying of data

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

```c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
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
for (i=0; i<size; i++) { c[i] = a[i]+b[i];}

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

```c
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

Likewise GPU can copy data from GPU memory to CPU

```c
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
```
CSCE 4610: Computer Architecture

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

\[
\text{add} \leftarrow \{N/\text{THREADS\_PER\_BLOCK},\text{THREADS\_PER\_BLOCK}\} \leftarrow (d\_a, d\_b, d\_c);
\]

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

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

CSCE 4610 April 20, 2017 13

CSCE 4610: Computer Architecture

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

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

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
CSCE 4610: Computer Architecture

GPU Memory Hierarchy

CPU memory – host memory
GPU – device memory

Need to move data from host to device (mapped memory)

No sharing across SM’s
Sharing within a SM is possible

Each “lane” (or thread) in SM is given a set of physical registers

Data moves from L2 to L1/local memory
And then to registers

CSCE 4610: Computer Architecture

No sharing of data among SM – so GPUs are good if you can exploit independent threads
for (i=…..) { loop body with no dependencies among iterations }

You lose performance if you have conditions in loop iterations
for (i=…) { if () { loop body} }

We have to execute the loop body twice: once for iterations satisfying the condition
once for iterations that do not satisfy conditions.

GPUs have special instructions called predicated instructions
These allow you to define if a thread should participate in the execution or not

Note that either for multithreading or vector processing we often rely on “loop” parallelism
This is sometimes called Data Parallelism

Note that this is different from Instruction Level Parallelism
although loop unrolling is often used to increase ILP
Thread Level Parallelism can be based on loops or “task parallelism”
if we have independent functions that can be executed in parallel

Lately the community is talking about Memory Level Parallelism
where we can perform independent memory accesses in parallel.

But for now we will focus on loop or data parallelism
And more specifically we will see how we can modify loops to get higher performance
these optimizations are done by compilers or manually not by hardware

The material covered comes from section 4.5 of the text, starting on page 315

Loop iterations can be completely independent
for (i=99; i>=0; i=i-1) {x[i] = x[i]+s;}

Note that here we DO have a dependency among instructions in the same iteration
WAR dependency on x[i]
You may also have RAW, WAW dependencies among statements inside the loop body

for (i=99; i>=0; i=i-1) {x[i] = x[i]+s;
  y[i] = x[i]*f;}

But in many cases we may have dependencies from iteration to iteration
for (i=99; i>=0; i=i-1) {x[i] = x[i-k]+s;
  y[i] = x[i]*f;}

Such dependencies are called loop carried dependencies

In some cases we may have both types of dependencies

Consider a loop like:
for (i=1; i<= 100; i++) {
  A[i+1] = A[i] + C[i]; /* S1 */
  B[i+1] = B[i] + A[i+1]; /* S2 */
}

S1: contains a loop-carry dependency. The value of A[i] on right hand side depends on the previous iteration
may prevent us from executing loop iterations in parallel (we can unroll for ILP)

S2: uses the value computed in S1 in the same iteration -- sequential data dependency
may prevent us from reordering instructions

Together we have a circular dependency and such loop iterations cannot be executed in parallel
CSCE 4610: Computer Architecture

If there is no circular dependency we MAY be able to rewrite loops to eliminate loop-carried dependency.

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

How can we parallelize this loop?

There is no circular dependency.

Can we reorder the statements?

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

Now we still have a loop-carried dependency

Can we rewrite the loop to eliminate this?

```
for (i=1; i<=99; i++) {
    B[i+1] = C[i] + D[i]; /* S2 */
    A[i+1] = A[i+1] + B[i+1]; /* S1 */
}
B[101] = C[101] + D[101];
```

Now we have no loop-carried dependency

How do we find if there are loop carried dependencies?

If we can express the indexes of array elements accessed in a loop as an expression, we can try to find if there is a potential loop carried dependency.

Sometimes it is more complex to see the dependency

Say a[i] = ...a[i-k]...

or a[i] = a(f(i)):
Consider simple relationship for array elements accessed as a function of loop iteration.
Two array elements accessed in a loop may be expressed as 
\[ A[a*i+b] \text{ and } A[c*i+d] \]
Or the array indexes are a linear (affine) function of loop iteration.
Given such relationships between iteration number and array elements
we can determine if there will be a “loop carried” dependency between them?

What we are trying to find is:
does the index \([a*i+b]\) in iteration \(i\) equal the index \([c*j+d]\) in iteration \(j\)?
If this happens, we have a dependency between iterations \(i\) and \(j\).

There is theoretical proof that if the GCD of \((c, a)\) does not divide \((d-b)\)
than the two array references cannot be the same for any \(i\) and \(j\) – different loop iterations.

If the relationship is not linear then it will not be possible to
determine dependencies at compile time
say \(A[f(i)] = A[g(i)]\) where \(f\) and \(g\) are functions

Most compilers assume that the loop is not parallelizable
Or we may use speculation and execute loop iterations in parallel

Consider the example on page 319
for \((i=0; i<100; i=i+1)\) {
    \[X[2*i+3] = X[2*i]*5.0;\]
}
So we have \(a=2, b=3, c=2\) and \(d=0\)
GCD(a,c) = 2 and \(d-b= -3\)
Since 2 does not divide -3, there is no loop carried dependency.
Other possible optimizations to improve loop level parallelism
We already talked about register renaming to eliminate WAR, WAR
We can do the same at a higher level by “renaming” variables
Consider the example on page 320
for (i=0; i<100; i++) {
    Y[i] = X[i]/c; /*S1 */
    X[i] = X[i]+c; /*S2*/
    Z[i] = Y[i] +c; /*S3 */
    Y[i] = c-Y[i]; } /*S4*/

RAW (true dependency): S1 to S3; S1 to S4 on Y[i] -- not loop carried
WAR (anti dependency) S1 to S2 on X[i]
    S3 to S4 on Y[i]
WAW (output dependency) S1 to S4 on Y[i]
Can we eliminate anti and output dependencies using different names?

for (i=0; i<100; i++) {
    T[i] = X[i]/c; /*S1 */
    T2[i] = X[i]+c; /*S2*/
    Z[i] = T[i] +c; /*S3 */
    Y[i] = c-T[i]; /*S4*/
}
for (i=0; i<100; i++) X[i] = T2[i];

There are many other optimizations that compiler can perform to improve parallelism
when loop carried dependencies exist -- such loops are called DO ACROSS
(without loop carried, we call those loops DO FOR ALL)
Note that when we use Software Pipelining, we are actually causing loop carried dependency
instructions from different iterations are used to create a new loop
so data from previous iterations would be needed in current iteration
But the goal there is to improve ILP on a SINGLE core or single pipeline
For multi-core or multi-threaded systems, we should try to eliminate loop carried
dependencies and execute LOOP ITERATIONS IN PARALLEL
CSCE 4610: Computer Architecture

There are many more compiler techniques to extract parallelism from loops either for Vector processors or array processors
Or even for thread parallelism on MIMD systems – particularly shared memory architectures

Amdahl’s law for parallel programs

\[
\text{Speedup} = \frac{\text{Execution time on one processor}}{\text{Execution time on } n \text{ processors}}
\]

Let \( f \) be a fraction of the program that cannot be executed in parallel

\[
\text{Speedup} = \frac{1}{f + \frac{(1-f)}{n}}
\]

In addition serial fraction, parallel execution incurs overheads
communication
waiting for locks and barriers

\[
T_{\text{overhead}} = (\text{Serial fraction}) + (\text{other overheads})
\]

CSCE 4610 April 20, 2017

Amdahl’s law of parallelism

see example on page 349
we want a speed up of 80 with 100 processors
how much serialization can we tolerate?

\[80 = \frac{1}{f + \frac{(1-f)}{100}}\]

serial_fraction = 0.0025 or 0.25%

Another example 5% serial fraction

<table>
<thead>
<tr>
<th>N</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>16.8</td>
</tr>
<tr>
<td>32</td>
<td>12.55</td>
</tr>
</tbody>
</table>

Measuring performance of parallel processors
Communication overhead -- bandwidth and latency
Shared memory -- delays due to synchronization