CS232 roadmap

- Here is what we have covered so far
  1. Understanding the relationship between HLL and assembly code
  2. Processor design, pipelining, and performance
  3. Memory systems, caches, virtual memory, I/O

- The next major topic is: **performance tuning**
  - How can I, as a programmer, make my programs run fast?
  - First step: where/why is my program slow?
    - Program profiling

- How does one go about optimizing a program?
  - Use better algorithms (do this first!)
  - Exploit the processor better (3 ways)
    1. Write hand-tuned assembly versions of hot spots
    2. Getting more done with every instruction
    3. Using more than one processor
“We should forget about small efficiencies, say about 97% of the time.”

-- Sir Tony Hoare
Collecting data

- The process is called “instrumenting the code”

- One option is to do this by hand:
  - record entry and exit times for suspected “hot” blocks of code
  - but this is tedious and error prone

- Fortunately, there are tools to do this instrumenting for us:
  - Gprof: The GNU profiler (compile with the \texttt{-pg} flag)
  - \texttt{gcc} keeps track of source code $\leftrightarrow$ object code correspondence
  - also links in a profiling signal handler
    - the program requests OS to periodically send it signals
    - signal handler records instruction that was executing (\texttt{gmon.out})
  - Display results using gprof command
    - Shows how much time is being spent in each function
    - Shows the path of function calls to the hot spot
Performance Optimization, cont.

How do we fix performance problems?

1. Create a Benchmark
2. Collect Data
3. Analyze Data and Identify Performance Problems

4. Fix the problems in your code or system

5. Is Problem Fixed?
   - No
   - Yes

6. Are performance requirements met?
   - No
   - Yes

Done!
Exploiting Parallelism

- We can exploit parallelism in two ways:

1. At the instruction level
   - Single Instruction Multiple Data (SIMD)
   - Make use of extensions to the ISA

2. At the core level
   - Rewrite the code to parallelize operations across many cores
   - Make use of extensions to the programming language
Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++ i) {
        C[i] = A[i] + B[i];
    }
}
```

Operating on one element at a time
Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++ i) {
        C[i] = A[i] + B[i];
    }
}
```

Operating on one element at a time
Consider adding together two arrays:

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++ i) {
        C[i] = A[i] + B[i];
    }
}
```

Exploiting Parallelism at the Instruction level (SIMD)

Operate on MULTIPLE elements

Single Instruction,
Multiple Data (SIMD)
Intel SSE/SSE2 as an example of SIMD

- Added new 128 bit registers (XMM0 - XMM7), each can store
  - 4 single precision FP values (SSE) \( 4 \times 32\)b
  - 2 double precision FP values (SSE2) \( 2 \times 64\)b
  - 16 byte values (SSE2) \( 16 \times 8\)b
  - 8 word values (SSE2) \( 8 \times 16\)b
  - 4 double word values (SSE2) \( 4 \times 32\)b
  - 1 128-bit integer value (SSE2) \( 1 \times 128\)b

\[
\begin{array}{cccc}
4.0 \ (32 \text{ bits}) & 4.0 \ (32 \text{ bits}) & 3.5 \ (32 \text{ bits}) & -2.0 \ (32 \text{ bits}) \\
+ & -1.5 \ (32 \text{ bits}) & 2.0 \ (32 \text{ bits}) & 1.7 \ (32 \text{ bits}) & 2.3 \ (32 \text{ bits}) \\
2.5 \ (32 \text{ bits}) & 6.0 \ (32 \text{ bits}) & 5.2 \ (32 \text{ bits}) & 0.3 \ (32 \text{ bits}) \\
\end{array}
\]
More than 70 instructions. Arithmetic Operations supported: Addition, Subtraction, Mult, Division, Square Root, Maximum, Minimum. Can operate on Floating point or Integer data.
Is it always that easy?

- No, not always. Let’s look at a little more challenging one:

```c
unsigned sum_array(unsigned *array, int length) {
    int total = 0;
    for (int i = 0 ; i < length ; ++i) {
        total += array[i];
    }
    return total;
}
```

- Is there parallelism here?
  - Yes, we could split the loop across two cores
How much faster?

- We’re expecting a speedup of 2

- OK, perhaps a little less because of Amdahl’s Law
  – overhead for forking and joining multiple threads

- But it's actually slower!! Why??

- Here’s the mental picture that we have - two processors, shared memory
This mental picture is wrong!

- We’ve forgotten about caches!
  - The memory may be shared, but each processor has its own L1 cache
  - As each processor updates total, it bounces between L1 caches

![Diagram showing multiple processors and caches](image-url)
The code is not only slow, its WRONG!

- Since the variable total is shared, we can get a data race.

- Increment operation: `total+= ...`  
  MIPS equivalent:  
  
  ```
  lw $t0, total
  addi $t0, $t0, $t1
  sw $t0, total
  ```

- A data race occurs when data is accessed and manipulated by multiple processors, and the outcome depends on the sequence or timing of these events.

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor 1</strong></td>
<td><strong>Processor 2</strong></td>
</tr>
<tr>
<td>lw $t0, total</td>
<td>lw $t0, total</td>
</tr>
<tr>
<td>addi $t0, $t0, $t1</td>
<td>addi $t0, $t0, $t1</td>
</tr>
<tr>
<td>sw $t0, total</td>
<td>sw $t0, total</td>
</tr>
<tr>
<td>lw $t0, total</td>
<td></td>
</tr>
<tr>
<td>addi $t0, $t0, $t1</td>
<td></td>
</tr>
<tr>
<td>sw $t0, total</td>
<td></td>
</tr>
</tbody>
</table>

counter increases twice  

counter increases once !!
unsigned sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    for ( ; i < length ; ++ i) {
        total += array[i];
    }
    return total;
}
Then we can write SIMD code for the hot part

```c
unsigned sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    for ( ; i < length ; ++i) {
        total += array[i];
    }
    return total;
}
```
Exploiting a multi-core processor

- Hardware can guarantee correctness with atomic operations, but its slow

```cpp
parallel_for (int i = 0; i < length; ++i){
    total += array[i];
}
```

- What if each thread had its own copy of `total`? (private, not shared)

```cpp
parallel_for (int i = 0; i < length; ++i) private(total) { 
    total += array[i];  // increment local copy
} 
```

// Now reduce the local copies of counter into a single variable

- This works because “+” is associative and commutative
  – fortunately, common operations have these properties
Summary

- Performance is of primary concern in some applications
  - Games, servers, mobile devices, super computers

- Many important applications have parallelism
  - Exploiting it is a good way to speed up programs.

- Single Instruction Multiple Data (SIMD) does this at ISA level
  - Registers hold multiple data items, instruction operate on them
  - Can achieve factor or 2, 4, 8 speedups on kernels
  - May require some restructuring of code to expose parallelism

- Exploiting core-level parallelism
  - May require atomic operations to avoid data races
  - Can sometimes be sped up using reductions