Lecture 17

Profiling & Optimization
Avoid Premature Optimization

- Novice developers rely on ad hoc optimization
  - Make private data public
  - Force function inlining
  - Decrease code modularity

- But this is a very bad idea
  - Rarely gives significant performance benefits
  - Non-modular code is very hard to maintain

- Write clean code first; optimize later
Debug vs Release

**Debug mode** is the default when you run
- All assertion checks are **enabled**
- **No compiler optimizations** are performed
- But works well with breakpoints and watches

**Release mode** is what to use on deployment
- All assertion checks are **disabled**
- **Compiler optimizations** performed (often -Os)
- But breakpoints and watches are unreliable
Debug vs Release

- **Debug mode** is the default when you run
  - All assertion checks are **enabled**
  - **No compiler optimizations** are performed
  - But works well with breakpoints and watches

- **Release mode** is what you want to use on deployment
  - All assertion checks are **disabled**
  - **Compiler optimizations** performed (often -O$s$)
  - But breakpoints and watches are unreliable

*This is often better than anything you can do!*
Debug vs Release

Profiling & Optimization
Performance Tuning

- Code follows an 80/20 rule (or even 90/10)
  - 80% of run-time spent in 20% of the code
  - Optimizing other 80% provides little benefit
  - Do nothing until you know what this 20% is

- Be careful in tuning performance
  - Never overtune some inputs at expense of others
  - Always focus on the overall algorithm first
  - Think hard before making non-modular changes
Case Study: Vectorization Support

- CUtgl has vectorization
  - SSE support for Mac/Win
  - NEON support for ARM
  - But currently turned off…
- Focused on high value areas
  - Vec4 and Mat4 for graphics
  - DSP and Filters for audio
  - Bespoke and hand tuned
- Was it worth it?
  - TestCUegl is test bed
  - Results surprising (sort of)
No Significant Win for Graphics

• **SSE** on 2019 MBook Pro
  • 2.4 GHz 8 core Intel i9
  • 32 Gig Ram

• **Neon** on iPhone XS Max
  • 2x2.5 GHz+4x1.6GHz Arm
  • 4 GB Ram

• Tests are **synthetic**
  • Unit tests for most ops
  • Mix of short & long comps
  • Want a standard workload
  • Vectorization best on long

### SSE Code

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Naïve</td>
<td>Vec</td>
</tr>
<tr>
<td>Vec4</td>
<td>488 µs</td>
<td>525 µs</td>
</tr>
<tr>
<td>Mat4</td>
<td>40595</td>
<td>40104</td>
</tr>
</tbody>
</table>

### Neon Code

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Naïve</td>
<td>Vec</td>
</tr>
<tr>
<td>Vec4</td>
<td>126 µs</td>
<td>61 µs</td>
</tr>
<tr>
<td>Mat4</td>
<td>12033</td>
<td>10038</td>
</tr>
</tbody>
</table>
No Significant Win for Graphics

- **SSE** on 2019 MBook Pro
  - 2.4 GHz 8 core Intel i9
  - 32 Gig Ram
- **Neon** on iPhone XS Max
  - 2x2.5 GHz
  - 4 GB Ram
- Tests are **synthetic**
  - Unit tests for most ops
  - Mix of short & long comps
  - Want a standard workload
  - Vectorization best on long

### Observations

- Naïve ARM >> Naïve Intel
- -Os, Vec Intel > -Os, Vec ARM
- -Os does not do much on iOS

<table>
<thead>
<tr>
<th>SSE Code</th>
<th>Debug</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vec</td>
<td>56 μs</td>
<td>412 μs</td>
</tr>
<tr>
<td>Naïve</td>
<td>367 μs</td>
<td>412 μs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Neon Code</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vec4</td>
<td>126 μs, 61 μs</td>
</tr>
<tr>
<td>Naïve</td>
<td>250 μs, 60 μs</td>
</tr>
<tr>
<td>Mat4</td>
<td>12033, 10038</td>
</tr>
<tr>
<td>Vec</td>
<td>10529, 9788</td>
</tr>
</tbody>
</table>
But Major Win for Audio DSP

- Audio all long comps
  - Adds/mults of long arrays
  - Arrays are audio chunks
  - **DSP:** Basic add/mul
  - **Filters:** IIR and FIR

- Why not graphics too?
  - Transform large meshes?
  - Better to do in shader!
  - Easily parallelizable

### SSE Code

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naïve</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP</td>
<td>27527</td>
<td>7355</td>
</tr>
<tr>
<td>Filter</td>
<td>872186</td>
<td>24392</td>
</tr>
<tr>
<td><strong>Vec</strong></td>
<td>11373</td>
<td>1515</td>
</tr>
<tr>
<td><strong>Naïve</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP</td>
<td>6957</td>
<td>7222</td>
</tr>
<tr>
<td>Filter</td>
<td>385377</td>
<td>378013</td>
</tr>
<tr>
<td><strong>Vec</strong></td>
<td>2059</td>
<td>2016</td>
</tr>
<tr>
<td><strong>Optimized -Os</strong></td>
<td>121061</td>
<td>121061</td>
</tr>
</tbody>
</table>

### Neon Code

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naïve</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP</td>
<td>6957</td>
<td>7222</td>
</tr>
<tr>
<td>Filter</td>
<td>385377</td>
<td>378013</td>
</tr>
<tr>
<td><strong>Vec</strong></td>
<td>2059</td>
<td>2016</td>
</tr>
<tr>
<td><strong>Optimized -Os</strong></td>
<td>121061</td>
<td>121061</td>
</tr>
</tbody>
</table>

Profiling & Optimization
### What Can We Measure?

#### Time Performance
- What code takes most time
- What is called most often
- How long I/O takes to finish
- Time to switch threads
- Time threads hold locks
- Time threads wait for locks

#### Memory Performance
- Number of heap allocations
- Location of allocations
- Timing of allocations
- Location of releases
- Timing of releases
- (Location of memory leaks)
### Analysis Methods

#### Profiling
- Analysis runs with program
  - Record behavior of program
  - Helps visualize this record

#### Static Analysis
- Analyze without running
  - Relies on language features
  - Major area of PL research

#### Advantages
- More data than static anal.
- Can capture user input

#### Disadvantages
- Hurts performance a lot
- May alter program behavior

- Offline; no performance hit
- Can analyze deep properties

- Conservative; misses a lot
- Cannot capture user input
Profiling

- Analysis runs with program
  - Record behavior of program
  - Helps visualize this record

- Advantages
  - More data than static anal.
  - Can capture user input

- Disadvantages
  - Hurts performance a lot
  - May alter program behavior
Time Profiling

Profiling & Optimization
# Time Profiling: Methods

## Software
- Code added to program
  - Captures start of function
  - Captures end of function
  - Subtract to get time spent
  - Calculate percentage at end
- **Not completely accurate**
  - Changes actual program
  - Also, how get the time?

## Hardware
- Measurements in hardware
  - Feature attached to CPU
  - Does not change how the program is run
  - Simulate w/ hypervisors
  - Virtual machine for Oss
  - VM includes profiling measurement features
  - **Example**: Xen Hypervisor
Time Profiling: Methods

<table>
<thead>
<tr>
<th>Time-Sampling</th>
<th>Instrumentation</th>
</tr>
</thead>
</table>
| • Count at periodic intervals  
  • Wakes up from sleep  
  • Looks at parent function  
  • Adds that to the count  
  • Relatively lower overhead  
  • Doesn’t count everything  
  • Performance hit acceptable  
  • May miss small functions  | • Count pre-specified places  
  • Specific function calls  
  • Hardware interrupts  
  • Different from sampling  
  • Still not getting everything  
  • But **exact view** of slice  
  • Used for targeted searches  |
Issues with Periodic Sampling

Real

Sampled
Issues with Periodic Sampling

Real

Sampled

Modern profilers fix with random sampling
## What Can We Measure?

<table>
<thead>
<tr>
<th>Time Performance</th>
<th>Memory Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>- What code takes most time</td>
<td>- Number of heap allocations</td>
</tr>
<tr>
<td>- What is called most often</td>
<td>- Location of allocations</td>
</tr>
<tr>
<td>- How long I/O takes to finish</td>
<td>- Timing of allocations</td>
</tr>
<tr>
<td>- Time to switch threads</td>
<td>- Location of releases</td>
</tr>
<tr>
<td>- Time threads hold locks</td>
<td>- Timing of releases</td>
</tr>
<tr>
<td>- Time threads wait for locks</td>
<td>- (Location of memory leaks)</td>
</tr>
</tbody>
</table>
What Can We Instrument?

### Time Performance
- What code takes most time
- What is called most often
- How long I/O takes to finish
- Time to switch threads
- Time threads hold locks
- Time threads wait for locks

### Memory Performance
- Number of heap allocations
- Location of allocations
- Timing of allocations
- Location of releases
- Timing of releases
- (Location of memory leaks)
Instrumentation: Memory

- Memory handled by malloc
  - Basic C allocation method
  - C++ \texttt{new} uses malloc
  - Allocates raw bytes
- \texttt{malloc} can be \texttt{instrumented}
  - Count number of mallocs
  - Track malloc addresses
  - Look for frees later on
- Finds memory leaks!

\begin{verbatim}
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
\end{verbatim}
Instrumentation: Memory
Profiling and Instrumentation Tools

- **iOS/X-Code**: Profiling Tools (⌘ I)
  - Supports a wide variety of instrumentation tools

- **Visual Studio**: Diagnostic Tools
  - C++ mostly limited to performance and memory

- **Android (Java)**
  - Dalvik Debug Monitor Server (DDMS) for traces
  - **TraceView** helps visualize the results of DDMS

- **Android (C++)**
  - Android NDK Profiler (3rd party)
  - **GNU gprof** visualizes the results of gmon.out
// Non-profiled code
monstartup("your_lib.so");

// Profiled code
captures everything
moncleanup();

// Non-profiled code

Android App  gmon.out  gprof
# Android Profiling

![Android Profiling](image)

<table>
<thead>
<tr>
<th>Name (location)</th>
<th>Samples</th>
<th>Calls</th>
<th>Time/Call</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
<td>3</td>
<td></td>
<td></td>
<td>100.0%</td>
</tr>
<tr>
<td><strong>factorial</strong></td>
<td>3</td>
<td>1000000</td>
<td>30ns</td>
<td>100.0%</td>
</tr>
<tr>
<td><strong>parents</strong></td>
<td>0</td>
<td>1000000</td>
<td>0ns</td>
<td>0.0%</td>
</tr>
<tr>
<td>main (factorial.c:26)</td>
<td>0</td>
<td>1000000</td>
<td>0ns</td>
<td>0.0%</td>
</tr>
<tr>
<td><strong>main</strong></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0.0%</td>
</tr>
<tr>
<td><strong>children</strong></td>
<td>3</td>
<td>1000000</td>
<td>30ns</td>
<td>100.0%</td>
</tr>
<tr>
<td>factorial (factorial.c:13)</td>
<td>3</td>
<td>1000000</td>
<td>30ns</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Poor Man’s Sampling

**Timing**

- Use the processor’s timer
  - Track time used by program
  - System dependent function
  - C-style `clock()` function

- Do not use “wall clock”
  - Timer for the whole system
  - Includes other programs
  - `CUTimestamp` is wall clock

**Call Graph**

- Create a hashtable
  - Keys = pairs (a calls b)
  - Values = time (time spent)

- Place code around call
  - Code inside outer func. a
  - Code before & after call b
  - Records start and end time
  - Put difference in hashtable
Poor Man’s Sampling

Timing
- Use the processor’s timer
  - Track time used by program
  - System dependent function
  - C-style `clock()` function
- Do not use “wall clock”
  - Timer for the whole system
  - Includes other programs
  - `CUTimestamp` is wall clock

Call Graph
- Create a hashtable
  - Keys = pairs (a calls b)
  - Values = time (time spent)
- Code around call
  - Code before & after call b
  - Records start and end time
  - Put difference in hashtable

Useful in networked setting
Using Timing Code

## clock

```c
#include <ctime>

// Get two timestamps
clock_t start = clock();
clock_t end = clock();

// Compute difference in seconds
float time = (end - start) / CLOCKS_PER_SEC;
```

## Timestamp

```c
#include <cugl/util/CUTimestamp>

// Get two timestamps
Timestamp start; // or start.mark();
Timestamp end;

// Compute difference in seconds
Uint64 micros;
micros = end.ellapsedTimeMicros(start);  
float time = micros / 1000000.0f;
```
Analysis Methods

Static Analysis

- Analyze without running
- Relies on language features
- Major area of PL research

**Advantages**
- Offline; no performance hit
- Can analyze deep properties

**Disadvantages**
- Conservative; misses a lot
- Cannot capture user input
int sum = 0
boolean done = false;
for(int ii; ii<=5 &<!done;) {
    if (j >= 0) {
        sum += j;
        if (sum > 100) {
            done = true;
        } else {
            i = i+1;
        }
    } else {
        i = i+1;
    }
}
print(sum);

q may be executed immediately after p
The code snippet is as follows:

```java
int sum = 0
boolean done = false;
for(int ii; ii<=5 && !done;) {
    if (j >= 0) {
        sum += j;
        if (sum > 100) {
            done = true;
        } else {
            i = i+1;
        }
    } else {
        i = i+1;
    }
}
print(sum);
```

The diagram illustrates the flow dependence analysis with the following identifiers:
- `sum = 0`
- `done = F`
- `i = 0`
- `i <= 5 && !done`
- `j >= 0`
- `sum = sum + j`
- `sum > 100`
- `done = T`
- `i = i+1`
- `endif`
- `print sum`

The value assigned at `p` is read at command `q`.
Model Checking

- Given a graph, logical formula $\varphi$
  - $\varphi$ expresses properties of graph
  - Checker determines if is true
- Often applied to software
  - Program as control-flow graph
  - $\varphi$ indicates acceptable paths
Static Analysis: Applications

- **Pointer analysis**
  - Look at pointer variables
  - Determine possible values for variable at each place
  - Can find memory leaks

- **Deadlock detection**
  - Locks are flow dependency
  - Determine possible owners of lock at each position

- **Dead code analysis**
Example: Analyze in X-Code
Summary

- Premature optimization is bad
  - Make code unmanageable for little gain
  - Best to identify the bottlenecks first

- Profiling can find runtime performance issues
  - But changes the program and incurs overhead
  - Sampling and instrumentation reduce overhead

- Static analysis is useful in some cases
  - Finding memory leaks and other issues
  - Deadlock and resource analysis