Profiling and 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 `-0s`)
  - 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 to use on deployment
  - All assertion checks are **disabled**
  - **Compiler optimizations** performed (often `–O0`)
  - But breakpoints and watches are unreliable

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

- CUGL 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?
  - TestCUGL is test bed
  - Results surprising (sort of)

```c
class Mat4 {
    #pragma mark Values
    public:
    #if defined CU_MATH_VECTOR_SSE
        __attribute__((__aligned__((16)))) union {
            __m128 col[4];
            float m[16];
        };
    #elif defined CU_MATH_VECTOR_NEON64
        __attribute__((__aligned__((16)))) union {
            float32x4_t col[4];
            float m[16];
        };
    #else
        /* The underlying matrix elements */
        float m[16];
    #endif
```
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

<table>
<thead>
<tr>
<th></th>
<th>SSE Code</th>
<th>Neon Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Debug</td>
<td>Optimized -Os</td>
</tr>
<tr>
<td>Naïve Vec</td>
<td>Naïve Vec</td>
<td></td>
</tr>
<tr>
<td>Vec4</td>
<td>488 μs</td>
<td>525 μs</td>
</tr>
<tr>
<td></td>
<td>412 μs</td>
<td>412 μs</td>
</tr>
<tr>
<td>Mat4</td>
<td>40595</td>
<td>40104</td>
</tr>
<tr>
<td></td>
<td>7271</td>
<td>7159</td>
</tr>
<tr>
<td></td>
<td>Debug</td>
<td>Optimized -Os</td>
</tr>
<tr>
<td>Naïve Vec</td>
<td>Naïve Vec</td>
<td></td>
</tr>
<tr>
<td>Vec4</td>
<td>126 μs</td>
<td>61 μs</td>
</tr>
<tr>
<td></td>
<td>250 μs</td>
<td>60 μs</td>
</tr>
<tr>
<td>Mat4</td>
<td>12033</td>
<td>10038</td>
</tr>
<tr>
<td></td>
<td>10529</td>
<td>9788</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+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

<table>
<thead>
<tr>
<th></th>
<th>SSE Code</th>
<th>Neon Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Debug</td>
<td>Optimized -Os</td>
</tr>
<tr>
<td>Naïve Vec</td>
<td>Vec</td>
<td>Naïve 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>
<tr>
<td></td>
<td>60 μs</td>
<td>250 μs</td>
</tr>
<tr>
<td></td>
<td>9788</td>
<td>10529</td>
</tr>
</tbody>
</table>

**Observations**

- Naïve ARM >> Naïve Intel
- -Os, Vec Intel > -Os, Vec ARM
- -Os does not do much on iOS
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>Naïve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vec</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>
</tbody>
</table>

### Neon Code

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Optimized -Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vec</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>
</tbody>
</table>
### What Can We Measure?

<table>
<thead>
<tr>
<th><strong>Time Performance</strong></th>
<th><strong>Memory Performance</strong></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>
## Analysis Methods

### 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

### 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
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
# 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

## Time-Sampling
- 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

## Instrumentation
- 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

Modern profilers fix with random sampling
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)
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++ new uses malloc
  - Allocates raw bytes

- malloc can be **instrumented**
  - Count number of mallocs
  - Track malloc addresses
  - Look for frees later on

- Finds memory leaks!

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
```
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
Android NDK Profiling

// Non-profiled code
monstartup("your_lib.so");

// Profiled code
moncleanup();

// Non-profiled code

captures everything

Android App  gmon.out  gprof
Android Profiling

![Android Profiling screenshot](image-url)
## 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
### 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 = pair of (calls 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

**Useful in networked setting**
### Using Timing Code

<table>
<thead>
<tr>
<th>clock</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#include &lt;ctime&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>// Get two timestamps</code></td>
<td></td>
</tr>
<tr>
<td><code>clock_t start = clock();</code></td>
<td></td>
</tr>
<tr>
<td><code>clock_t end = clock();</code></td>
<td></td>
</tr>
<tr>
<td><code>// Compute difference in seconds</code></td>
<td></td>
</tr>
<tr>
<td><code>float time = (end-start)</code></td>
<td></td>
</tr>
<tr>
<td><code>time /= CLOCKS_PER_SEC;</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;cugl/util/CUTimestamp&gt;</code></td>
</tr>
<tr>
<td><code>// Get two timestamps</code></td>
</tr>
<tr>
<td><code>Timestamp start; // or start.mark();</code></td>
</tr>
<tr>
<td><code>Timestamp end;</code></td>
</tr>
<tr>
<td><code>// Compute difference in seconds</code></td>
</tr>
<tr>
<td><code>Uint64 micros;</code></td>
</tr>
<tr>
<td><code>micros = end.ellapsedTimeMicros(start);</code></td>
</tr>
<tr>
<td><code>float time = micros/1000000.0f</code></td>
</tr>
</tbody>
</table>
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
```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 {
    }
}
print(sum);
```

Static Analysis: Control Flow

p → q  q may be executed immediately after p
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 {
        done = false;
    }
}
print(sum);
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

```
// If defined (CU_MATH_VECTOR_SSE)
  __m_store_ps(d1, __m_setr_ps(1, __c1(4+asize-4), __c1(4+asize-3),
                             __c1(4+asize-2));
  __m_store_ps(d1+4, __m_setr_ps(0, 1, __c1(4+asize-4),
                              __c1(4+asize-3));
  __m_store_ps(d1+8, __m_setr_ps(0, 0, 1, __c1(4+asize-3));
  __m_store_ps(d1+12, __m_setr_ps(0, 0, 0, 1));

  for(size_t ii = 0; ii < asize; ii++) {
    __m128 rows;
    __m128 temp = __m_load_ps(__c1+ii*4);
    temp = __mmoveln_ps(temp, __mm_setzero_ps());
  }
```
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