/INFOMOV/

Optimization & Vectorization

J. Bikker - April-June 2024 - Lecture 3: “Profiling”

Welcome!

Today’s Agenda:

- Introduction
- Metrics, Clocks and Counters
- Scalability Issues
- Environment and Tools
- Dirty Hands

Introduction

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
10. Report.

Do you actually need to speed it up?
By how much?

Things to consider:

- You have a finite amount of time for this
- You don’t want to break anything
- You don’t want to reduce maintainability

➔ Focus on ‘low hanging fruit’ – typically a small portion of the code.

Introduction

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
10. Report.

Don’t trust your intuition

- Not even when optimizing your own code.
- *Especially* not when you are proficient at optimizing.

Blind changes may *reduce* the performance of the code.

Needless to say: *use version control*.

Introduction

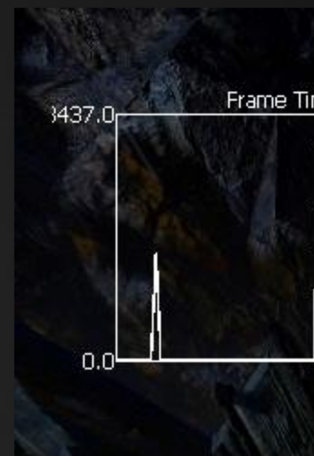
Profiling

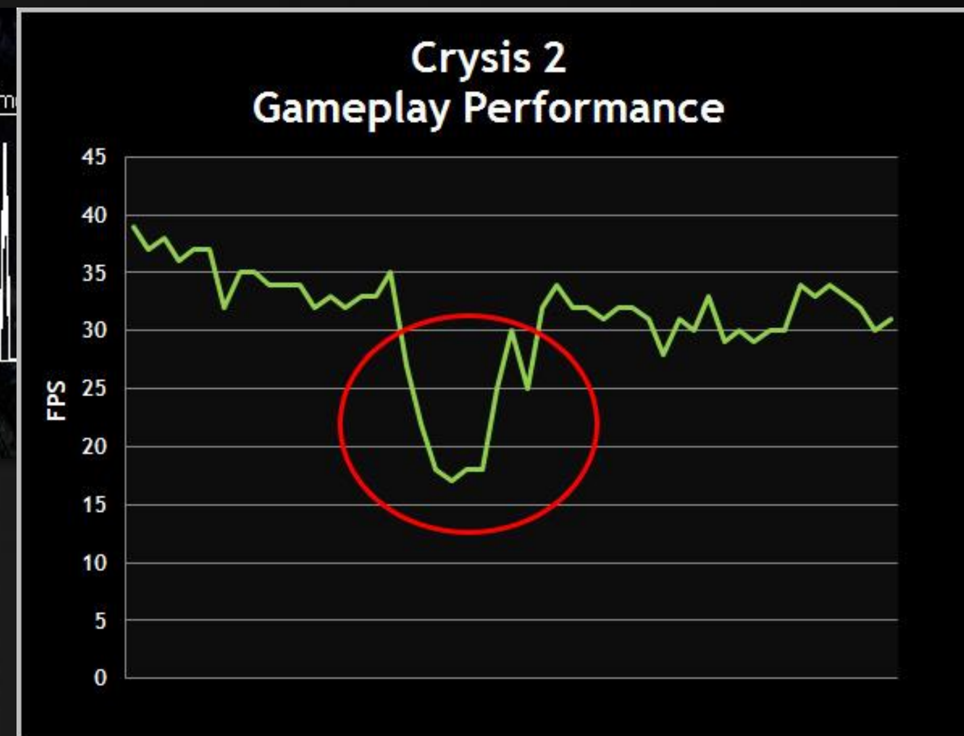
Measuring application performance

- Using timers in the code
- Using external tools

Details:

- Overall, average or time analysis
- What to measure
- Doing proper experiment setup
- Interpreting the data
- Acting on the findings
- Repeating the whole process





Focus on 'low hanging fruit'

Introduction

Subject

Application breakdown, Tick:

```
t.reset(); Transform(); elapsed1 = t.elapsed();
t.reset(); Sort();      elapsed2 = t.elapsed();
t.reset(); Render();    elapsed3 = t.elapsed();
```

Transform:

```
for ( int i = 0; i < DOTS; i++ )
   m_Rotated[i] = m.Transform( m_Points[i] );
```

Sort:

```
for ( int i = 0; i < DOTS; i++ )
   for ( int j = 0; j < (DOTS - 1); j++ )
      if (m_Rotated[j].z > m_Rotated[j + 1].z) Swap( j, j + 1 );
```

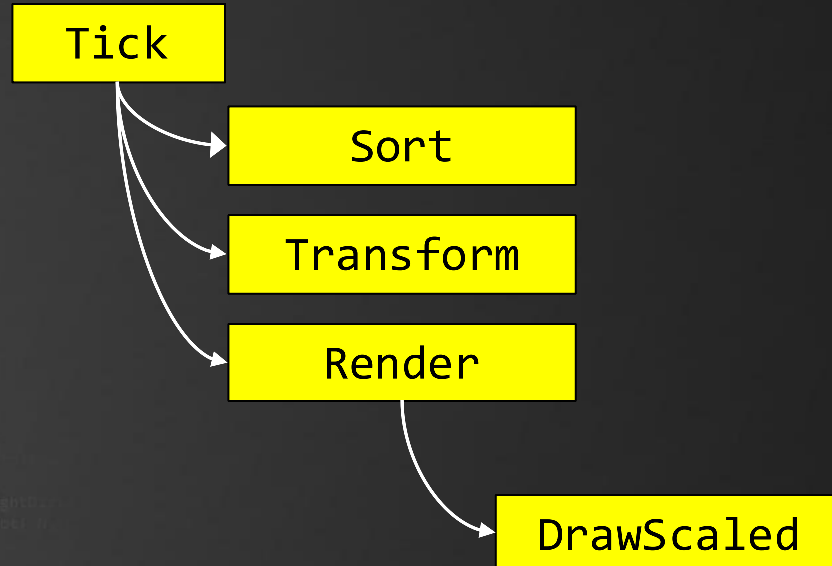
Render:

```
for ( int i = 0; i < DOTS; i++ )
   m_Dot->DrawScaled( (sx - size / 2), (sy - size / 2), size, size, screen );
```

Introduction

DotCloud

Application breakdown:



Today's Agenda:

- Introduction
- Metrics, Clocks and Counters
- Scalability Issues
- Environment and Tools
- Dirty Hands

Metrics

Timers

The template includes a ‘Timer’:

```
// timer
struct Timer
{
    Timer() { reset(); }
    float elapsed() const
    {
        chrono::high_resolution_clock::time_point t2 = chrono::high_resolution_clock::now();
        chrono::duration<double> time_span = chrono::duration_cast<chrono::duration<double>>(t2 - start);
        return (float)time_span.count();
    }
    void reset() { start = chrono::high_resolution_clock::now(); }
    chrono::high_resolution_clock::time_point start;
};
```

Based on some quick research:

- Windows: 100ns
- Linux: 10ns

@ 5Ghz: 100ns ≈ 500 CPU cycles.

Metrics

Timers

Note that resolution is not very important.

If it takes very little time to execute some code:

```
t.reset(); Transform(); elapsed1 = t.elapsed();
t.reset(); Sort();      elapsed2 = t.elapsed();
t.reset(); Render();    elapsed3 = t.elapsed();
```

Then we just execute it multiple times:

```
t.reset(); for( int i = 0; i < 1000; i++ ) Transform(); elapsed1 = t.elapsed();
t.reset(); for( int i = 0; i < 1000; i++ ) Sort();      elapsed2 = t.elapsed();
t.reset(); for( int i = 0; i < 1000; i++ ) Render();    elapsed3 = t.elapsed();
```

(Note that there's issues with that approach)

Metrics

Timers

However, there is a different problem with timers.

CPU-Z

CPU | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name: Intel Core i9 13900KS
Code Name: Raptor Lake — Max TDP: 150.0 W
Package: Socket 1700 LGA
Technology: 10 nm — Core VID: 0.893 V
Specification: 13th Gen Intel® Core™ i9-13900KS
Family: 6 — Model: 7 — Stepping: 1
Ext. Family: 6 — Ext. Model: B7 — Revision: B0
Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, AVX-VNNI, FMA3, SHA

Clocks (P-Core #0)

Core Speed: 3700.00 MHz
Multiplier: x 37.0 (8.0 - 56.0)
Bus Speed: 100.00 MHz
Rated FSB:

Cache

L1 Data: 8 x 48 KB + 16 x 32 KB
L1 Inst.: 8 x 32 KB + 16 x 64 KB
Level 2: 8 x 2 MB + 4 x 4 MB
Level 3: 36 MBytes

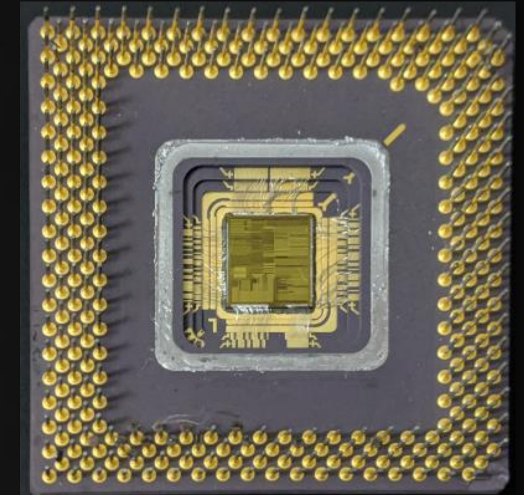
Selection: Socket #1 — Cores: 8P + 16E — Threads: 32

CPU-Z Ver. 2.09.0.x64 — Tools — Validate — Close

Question is…

- Is that in fact something we want to include in our experiments?
- If not, can we counter it, i.e., can we make clock speed a stable number?
- If not, can we measure something that is independent of clock speed?

Metrics



CPU Clock

There is a way to directly access the CPU clock:

```
uint64_t start = __rdtsc();
...
uint64_t elapsed = __rdtsc() – start;
```

__rdtsc is an assembler instruction that *returns the number of CPU cycles since the last reset.*

Using __rdtsc():

- We *still* get fluctuating results
- We still benefit from measuring longer periods

Metrics

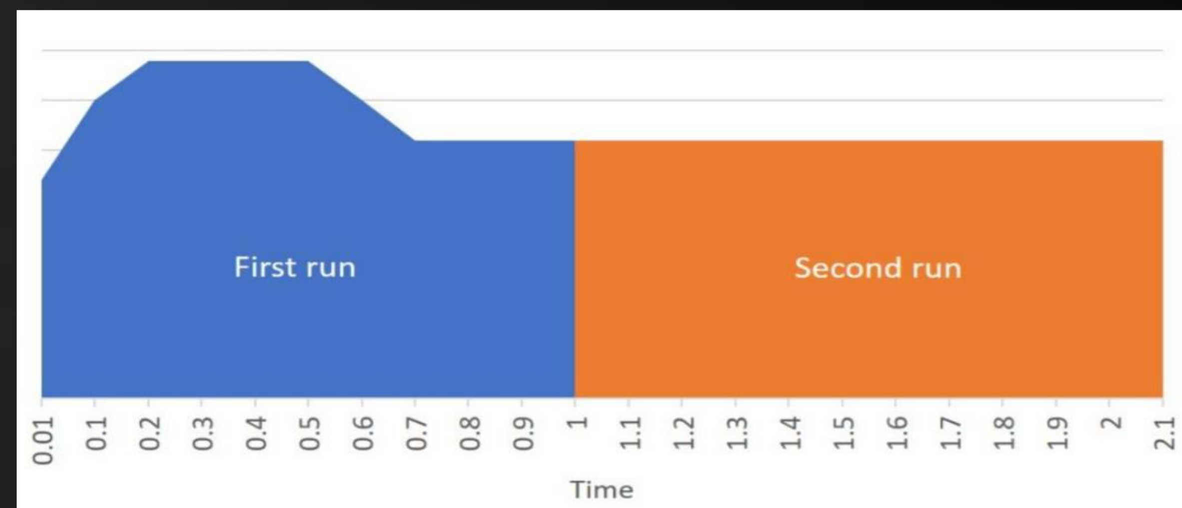
Sources of Noise

When using wallclock time, the source of noise is obvious:

- The CPU clock is not stable.

The timing variations when measuring CPU cycles are less obvious.

- OS task switching
- Caches
- Interrupts
- Subtleties of the memory system



Metrics

Deal with Uncertainty

To get reliable results:

1. Take multiple measurements..
2. …until standard deviation lies within desired range.
3. Mind ‘cache warming’: skip the first couple of seconds.
4. Try to use a stable testing environment.

Today’s Agenda:

- Introduction
- Metrics, Clocks and Counters
- Scalability Issues
- Environment and Tools
- Dirty Hands

Scalability

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat step 6 and 7 until time runs out
9. Report.

Scalability

Consistent Approach

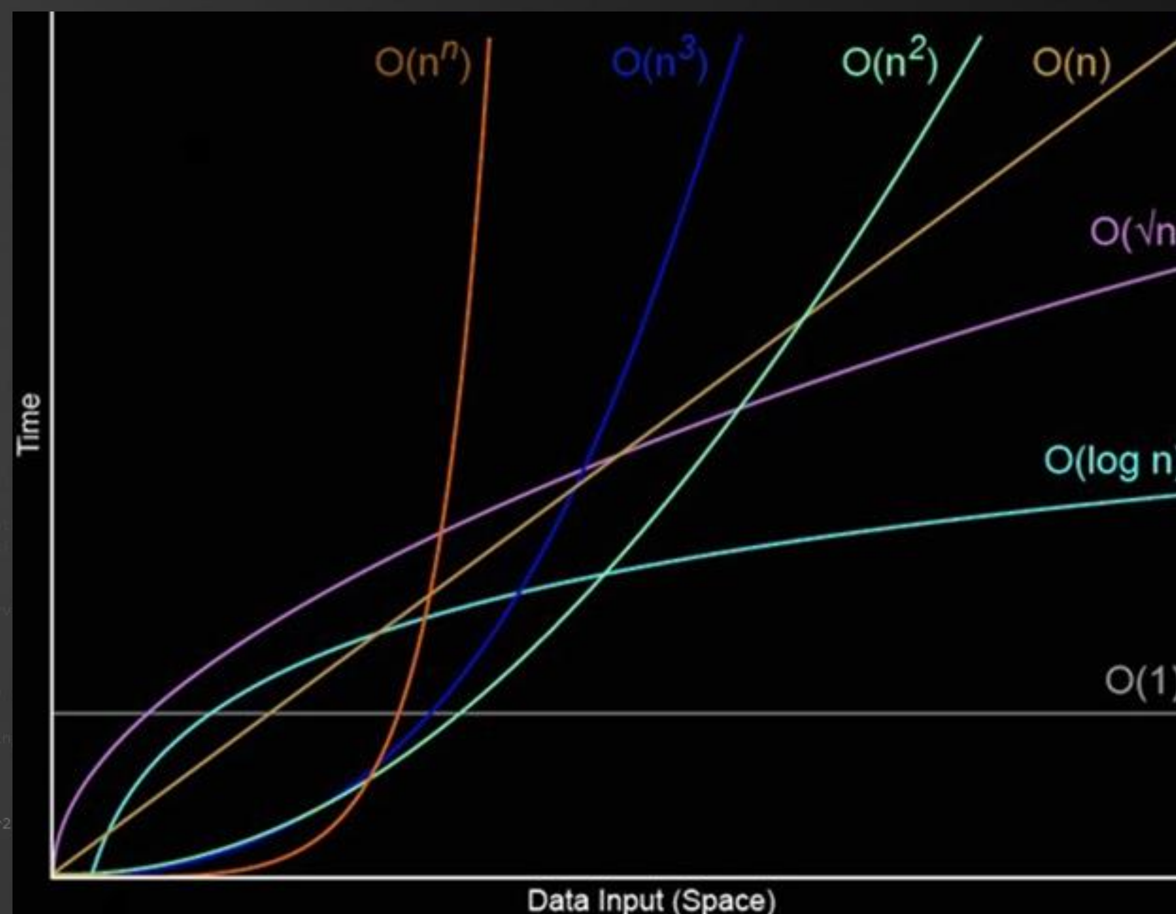
(0.) Determine optimization requirements
1. Profile: determine hotspots
2. **Analyze hotspots: determine scalability**
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
10. Report.

Scalability

Scalability

The Importance of Testing with the Target Workload

Code sections typically don’t scale linearly when you increase the size of the dataset.



Scalability

Improving Scalability

Remember:

Focus on low-hanging fruit.

The absolute best sorting algorithm depends on the dataset. Here:

- Somewhat sizeable
- Uniformly distributed
- Simple keys (depth: float)
- Mostly already sorted
- We don't need a perfect nor consistent sort

Optimal choice: probably Radix Sort, which can run in O(N) under these circumstances.

Scalability

Improving Scalability

So:

I chose Quicksort. ☺

```
void Swap( vec3& a, vec3& b ) { vec3 t = a; a = b; b =
int Pivot( vec3 a[], int first, int last )
{
    int p = first;
    vec3 e = a[first];
    for (int i = first + 1; i <= last; i++) if (a[i].z <= e.z) Swap( a[i], a[++p] );
    Swap( a[p], a[first] );
    return p;
}
void QuickSort( vec3 a[], int first, int last )
{
    int pivotElement;
    if (first >= last) return;
    pivotElement = Pivot( a, first, last );
    QuickSort( a, first, pivotElement - 1 );
    QuickSort( a, pivotElement + 1, last );
}
```

That seems an odd choice. Reasoning:

- Quicksort is much better than Bubblesort
- I can add Quicksort in minutes.
- I don't expect sorting to be a bottleneck after that.

The time required to add RadixSort is better spent elsewhere…

Today’s Agenda:

- Introduction
- Metrics, Clocks and Counters
- Scalability Issues
- Environment and Tools
- Dirty Hands

Tools

Profilers

VerySleepy
Intel VTune
AMD µProf

Both profilers are sampling profilers:

They work by interrupting your program X times per second (typically: each millisecond).
At this point, the profiler records at which address the CPU is executing code.

- A sampling profiler makes your software slower.
- A sampling profiler must store data, which affects the caches.
- Code that is visited infrequently may be missed by the sampling process.

Tools

Very Sleepy - C:\Users\bikke\AppData\Local\Temp\64D1.tmp

File View Help

Functions

Name	Exclu...	Inclusive	% Exclusive	% Inclusive	Module
Tmpl8::Sprite::DrawScaled	0.07s	0.07s	1.32%	1.32%	Tmpl8_2019-01
Tmpl8::Game::Tick	0.02s	0.14s	0.32%	2.89%	Tmpl8_2019-01
Tmpl8::Surface::Clear	0.02s	0.06s	0.32%	1.17%	Tmpl8_2019-01
Tmpl8::Game::Transform	0.00s	0.00s	0.04%	0.08%	Tmpl8_2019-01
__libm_sse2_sincosf_	0.00s	0.00s	0.04%	0.04%	Tmpl8_2019-01
SDL_main	0.00s	0.46s	0.00%	9.17%	Tmpl8_2019-01
main_getcmdline	0.00s	0.46s	0.00%	9.17%	Tmpl8_2019-01
__scrt_common_main_seh	0.00s	0.46s	0.00%	9.17%	Tmpl8_2019-01
Tmpl8::timer::get	0.00s	0.00s	0.00%	0.04%	Tmpl8_2019-01

Averages | Call Stacks | Filters

Main

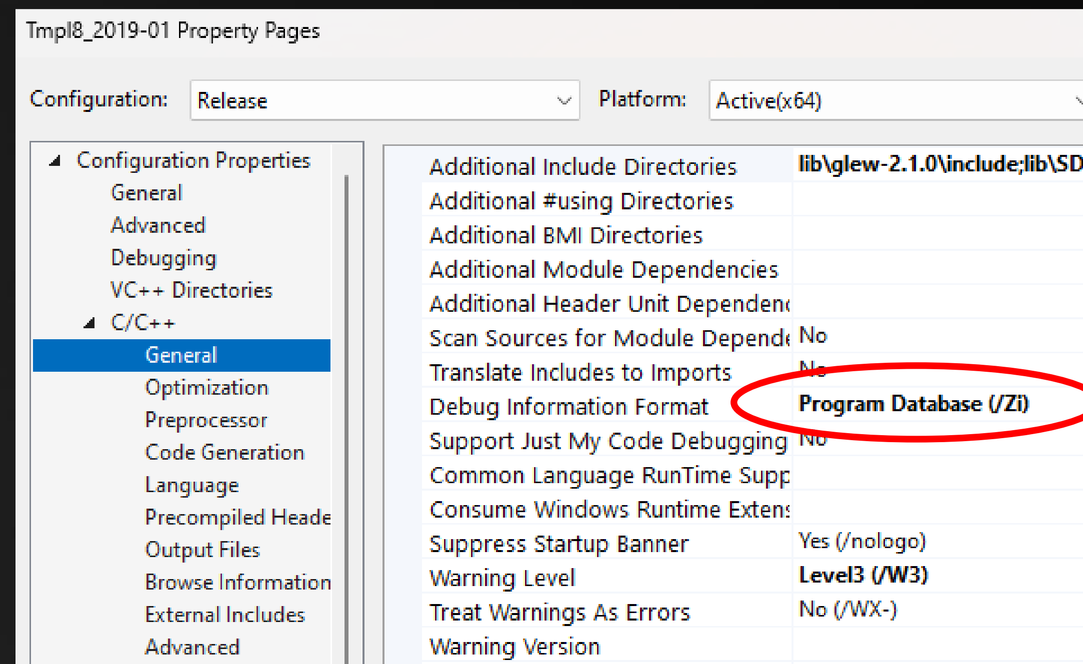
Function Name	
Module	Tmpl8_2019-01
Source File	

Source | Log

```
415          }
416
417          void Sprite::DrawScaled( float a_X, float a_Y, float a_Width, float
418          {
419             Pixel* dest = a_Target->GetBuffer() + (int)a_X + (int)a_Y * a_Ta
420             Pixel* src = GetBuffer() + m_CurrentFrame * 32;
421   0.01s     for ( int y = 0; y < (int)a_Height; y++ ) for ( int x = 0; x < (
422             {
423                int v = (int)((y * 32) / a_Height);
424   0.01s        int u = (int)((x * 32) / a_Width);
425   0.03s        if (src[u + v * 128] & 0xffffff)
426   0.01s           *(dest + x + y * a_Target->GetWidth()) = src[u + v * 128];
427             }
428          }
429
```

Source file: C:\projects\dotcloud - tmp2019\surface.cpp   Line 422

Tools



Profilers

VerySleepy
Intel VTune / AMD µProf

Always profile using the Release Configuration!

However:

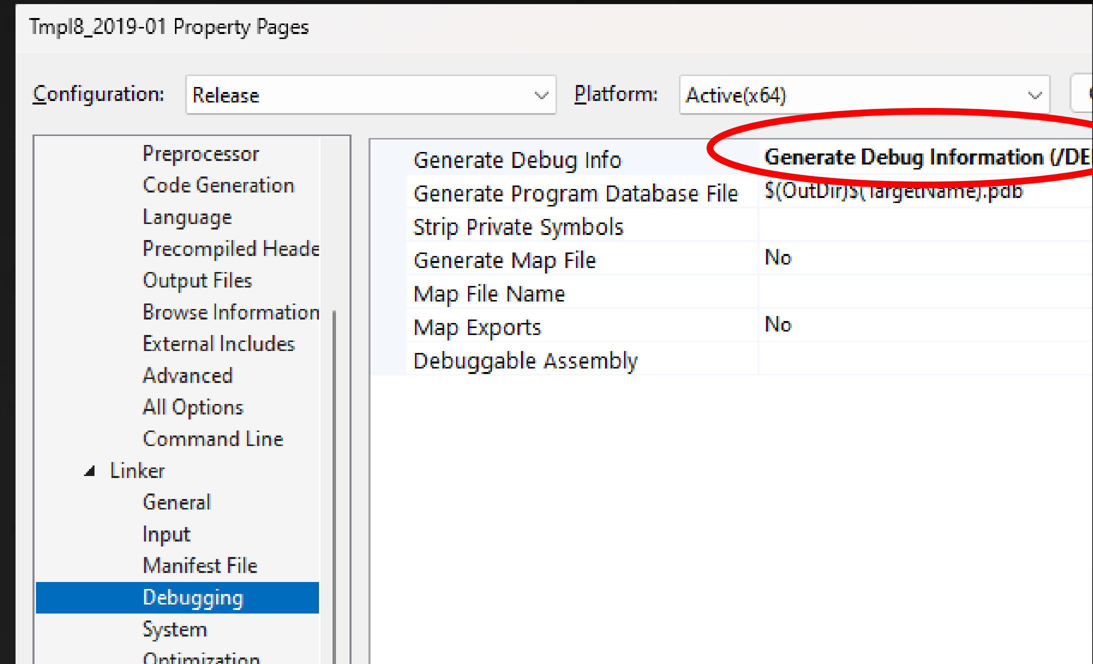
All profilers need your project to be compiled with DEBUG information.

Enabling debug information in release mode in Visual Studio:

- Properties >> C/C++ >> General >> Debug information format
- Properties >> Linker >> Debugging >> Generate Debug Info

NOTE: The Visual Studio working folder is typically not where the .exe is, but where the .sln is.

Tools



Profilers

VerySleepy
Intel VTune
AMD µProf

Always profile using the Release Configuration!

However:

All profilers need your project to be compiled with DEBUG information.

Enabling debug information in release mode in Visual Studio:

- Properties >> C/C++ >> General >> Debug information format
- Properties >> Linker >> Debugging >> Generate Debug Info

Tools

VerySleepy Data

Exclusive time:
time spent in the function, excluding calls to other functions.

Inclusive time:
time spent in the function and called functions.

Source code:
Tells you how much execution time is spent on a particular line. Profile longer to get more detailed results.

VerySleepy will not tell you *why* a particular function or line is ‘expensive’.

Very Sleepy - C:\Users\bikke\AppData\Local\Temp\64D1.tmp

File View Help

Functions

Name	Exclu...	Inclusive	% Exclusive	% Inclusive	Module
Tmpl8::Sprite::DrawScaled	0.07s	0.07s	1.32%	1.32%	Tmpl8_2019-01
Tmpl8::Game::Tick	0.02s	0.14s	0.32%	2.89%	Tmpl8_2019-01
Tmpl8::Surface::Clear	0.02s	0.06s	0.32%	1.17%	Tmpl8_2019-01
Tmpl8::Game::Transform	0.00s	0.00s	0.04%	0.08%	Tmpl8_2019-01
_libm_sse2_sincosf_	0.00s	0.00s	0.04%	0.04%	Tmpl8_2019-01
SDL_main	0.00s	0.46s	0.00%	9.17%	Tmpl8_2019-01
main_getcmdline	0.00s	0.46s	0.00%	9.17%	Tmpl8_2019-01
_scrt_common_main_seh	0.00s	0.46s	0.00%	9.17%	Tmpl8_2019-01
Tmpl8::timer::get	0.00s	0.00s	0.00%	0.04%	Tmpl8_2019-01

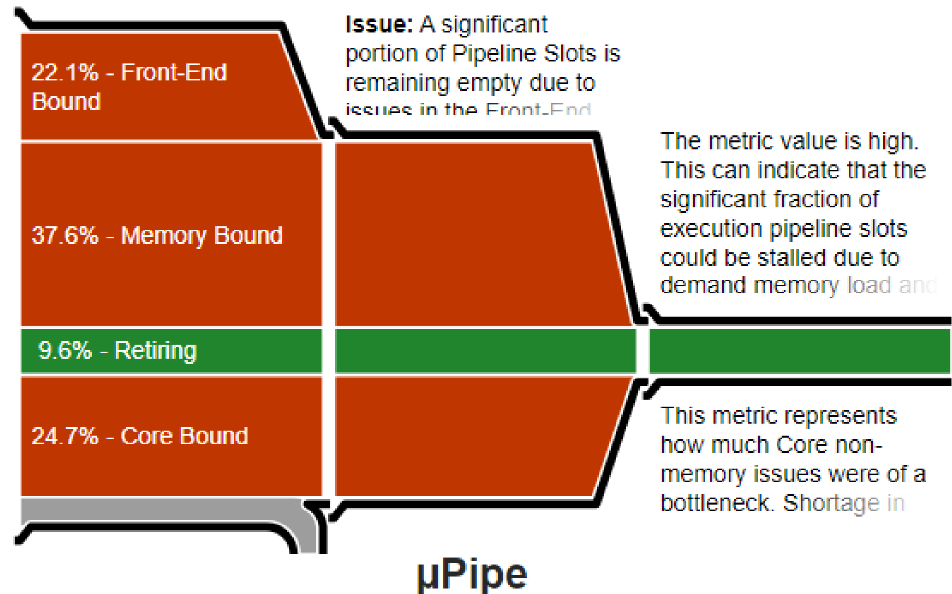
Source Log

```
415         }
416
417         void Sprite::DrawScaled( float a_X, float a_Y, float a_Width, float
418         {
419            Pixel* dest = a_Target->GetBuffer() + (int)a_X + (int)a_Y * a_Ta
420            Pixel* src = GetBuffer() + m_CurrentFrame * 32;
421  0.01s     for ( int y = 0; y < (int)a_Height; y++ ) for ( int x = 0; x < (
422            {
423               int v = (int)((y * 32) / a_Height);
424  0.01s        int u = (int)((x * 32) / a_Width);
425  0.03s        if (src[u + v * 128] & 0xffffff)
426  0.01s           *(dest + x + y * a_Target->GetWidth()) = src[u + v * 128];
427            }
428         }
429
```

Source file: C:\projects\dotcloud - tmp2019\surface.cpp Line 422

Elapsed Time: 4.843s

Metric	Value	Unit
Clockticks:	3,193,200,000	
Instructions Retired:	1,461,600,000	
CPI Rate:	2.185	
MUX Reliability:	0.736	
Retiring:	9.6%	of Pipeline Slots
Front-End Bound:	22.1%	of Pipeline Slots
Bad Speculation:	6.0%	of Pipeline Slots
Back-End Bound:	62.4%	of Pipeline Slots
Memory Bound:	37.6%	of Pipeline Slots
L1 Bound:	0.0%	of Clockticks
DTLB Overhead:	100.0%	of Clockticks
Load STLB Hit:	100.0%	of Clockticks
Load STLB Miss:	6.2%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	1.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.2%	of Clockticks
FB Full:	100.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	11.6%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	7.0%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	40.8%	of Clockticks
Memory Bandwidth:	34.9%	of Clockticks
Memory Latency:	31.1%	of Clockticks
Store Bound:	1.9%	of Clockticks
Core Bound:	24.7%	of Pipeline Slots
Average CPU Frequency:	3.8 GHz	
Total Thread Count:	16	
Paused Time:	0s	



µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Effective Physical Core Utilization: 3.5% (0.140 out of 4)

Effective Logical Core Utilization: 2.2% (0.172 out of 8)

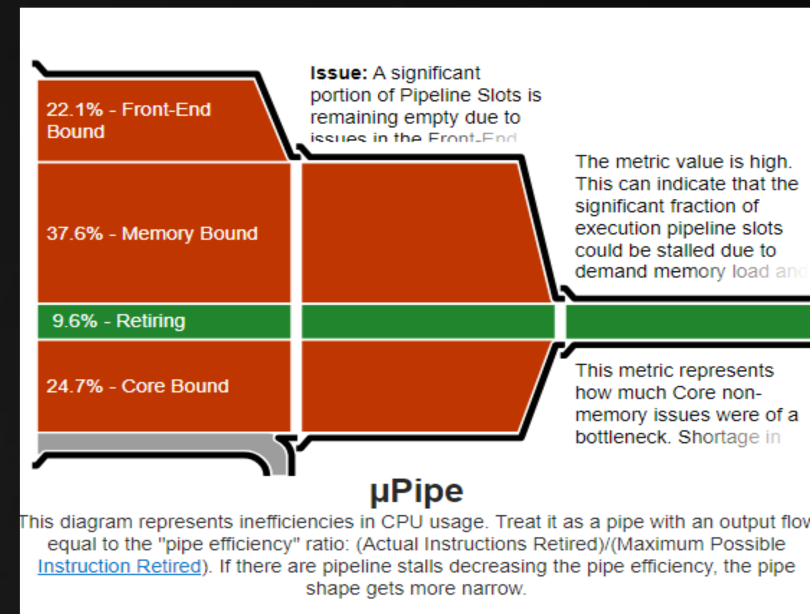
Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Source | Assembly

Source Line ▲	Source	Clockticks	Instructions Retired	CPI Rate	Retiring
416					
417	void Sprite::DrawScaled(float a_X, float a_Y, float a_Width, float a_Height, Surface* a_Target)				
418	{	0	1,800,000	0.000	
419	Pixel* dest = a_Target->GetBuffer() + (int)a_X + (int)a_Y * a_Target->GetPitch();				
420	Pixel* src = GetBuffer() + m_CurrentFrame * 32;	1,800,000	0		
421	for (int y = 0; y < (int)a_Height; y++) for (int x = 0; x < (int)a_Width; x++)	21,600,000	34,200,000	0.632	
422	{				
423	int v = (int)((y * 32) / a_Height);				
424	int u = (int)((x * 32) / a_Width);	30,600,000	106,200,000	0.288	
425	if (src[u + v * 128] & 0xffffff)	100,800,000	192,600,000	0.523	
426	*(dest + x + y * a_Target->GetWidth()) = src[u + v * 128];	57,600,000	120,600,000	0.478	
427	}				
428	}				
429					
430	void Sprite::InitializeStartData()				
431	{				
432	for (unsigned int f = 0; f < m_NumFrames; ++f)				
433	{				
434	m_Start[f] = new unsigned int[m_Height];				
435	for (int y = 0; y < m_Height; ++y)				
436	{				
437	m_Start[f][y] = m_Width;				
438	Pixel* addr = GetBuffer() + f * m_Width + y * m_Pitch;				
439	for (int x = 0; x < m_Width; ++x)				
440	{				
441	if (addr[x])				
442	{				
443	m_Start[f][y] = x;				
444	break;				
445	}				
446	}				
447	}				
448	}				

Tools



VTune Data

VTune gives you a *lot* of information.

Don't expect to understand it all at once! That being said:

Executed	instructions that entered the pipeline
Retired	instructions for which 'WB' was done
CPI/IPC	instructions per cycle / cycles per instruction
Micro-ops	many instructions are converted into several *micro-ops.* In some cases, several instructions are *fused* in a single micro-op. More info: https://easyperf.net/blog/2018/02/23/MacroFusion-in-Intel-CPUs#micro--macro-fusion

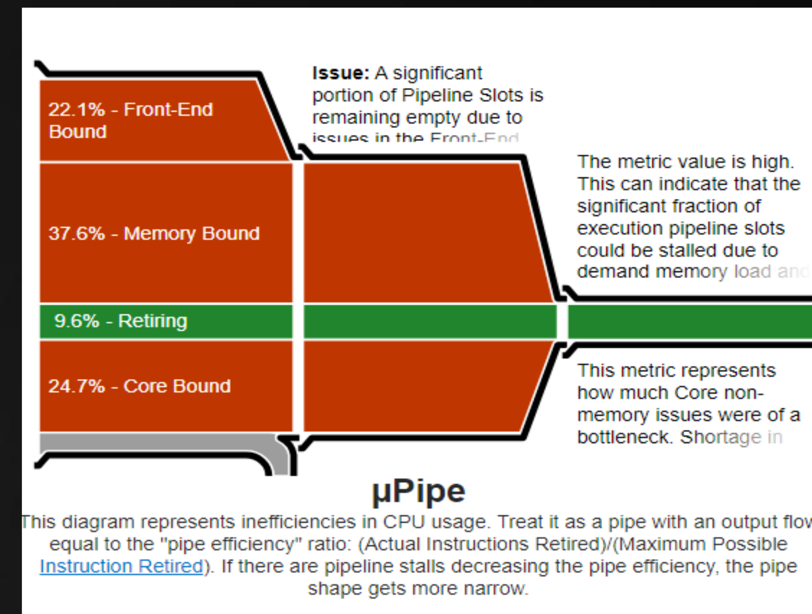
i	1	2	3	4	5	6
Branch (a<b)	IF	ID			EX	WB
Call foo		IF*	ID*	EX*		
// instr. foo			IF*	ID*	EX*	

t →

VTune will also tell you about:

Cache misses (L1, L2, L3)
Branch mispredictions
...and a lot more.

Tools



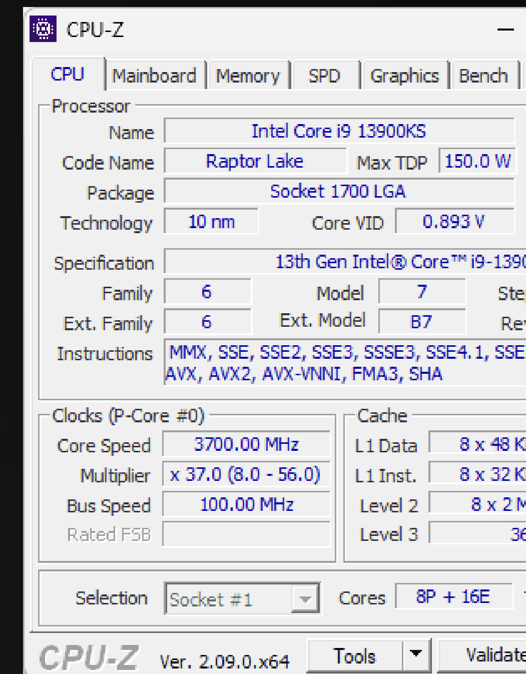
VTune Data

Intel’s VTune gives you a *lot* of information.

Some tips:

1. Only look at specific data for specific problems
2. There’s good online resources for VTune
3. Quite often, VerySleepy gets you everything you need…
4. A latency often shows up late:
   - E.g., when the result of a sqrt is *used*
5. Intel VTune does *not* work on AMD processors:
   It uses CPU-specific hardware counters.

Tools



Environment

To obtain the most stable results:

- Disable all features that change the clock speed of your CPU
  (warning: can be dangerous, use at your own risk)
- Windows: Create a power plan that *underclocks* the CPU significantly
  (verify using CPU-Z)
- Profile the code single-threaded, if possible
  (except when specifically optimizing threading behavior)
- Profile the code on a single core
  (windows: SetThreadAffinityMask)

Today’s Agenda:

- Introduction
- Metrics, Clocks and Counters
- Scalability Issues
- Environment and Tools
- Dirty Hands

Scalability

Blind Stupidity

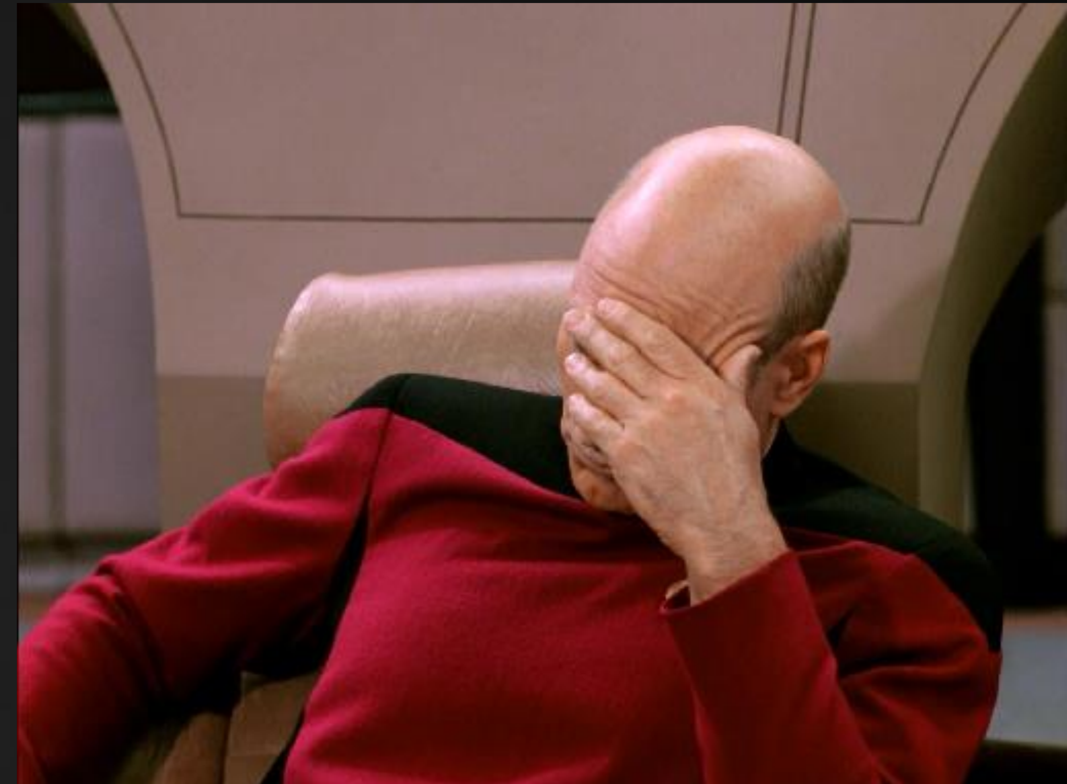
Low Level Optimization of DrawScaled

Now what?

- Plot multiple pixels at a time?
- …

How many different ball sizes do we encounter?

…Why don’t we simply pre-calculate those frames?



Blind Stupidity

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity." (W.A. Wulff)

Blind Stupidity

Low Level Optimization of Sprite::Draw

Pre-scaled sprites are faster than on-the-fly scaling.

But, we still have loops, and if-statements, and look-ups. I wonder…

...

Low Level Optimization of Sprite::Draw

Extreme Optimization:

- We simply generate a function that plots every pixel, without the need for a loop.

```
void Sprite::DrawBall( int x, int y, int size,
                       Surface* target )
{
   uint* a = target->GetBuffer() + x + y * SCRWIDTH;
   switch( size )
   {
   case 1:
      break;
   case 2:
      a[1]=1052688;
      a[800]=1052688;
      a[801]=15724527;
      break;
   case 3:
      a[801]=9737364;
      a[802]=8684676;
      a[1601]=8684676;
      a[1602]=8092539;
      break;
   case 4:
      a[2]=1052688;
      a[801]=6513507;
      a[802]=9737364;
      a[803]=4868682;
      a[1600]=1052688;
      a[1601]=9737364;
      a[1602]=15724527;
      a[1603]=7566195;
      a[2401]=4868682;
      a[2402]=7566195;
      a[2403]=3223857;
      break;
```

...

Low Level Optimization of Sprite::Draw

Extreme Optimization:

- We simply generate a function that plots every pixel, without the need for a loop.

```
FILE* f = fopen( "drawfunc.h", "w" );
fprintf( f, "void Sprite::DrawBall( int x, int y, int size, Surface* target )\n" );
fprintf( f, "{\nuint* a = target->GetBuffer() + x + y * SCRWIDTH;\nswitch( size )\n{\n" );
for( int i = 0; i < 64; i++ )
{
   ...
   fprintf( f, "case %i:\n", size );
   for( int y = 0; y < size; y++) for( int x = 0; x < size; x++ )
   {
      int a = y * SCRWIDTH + x;
      if (scaled[i]->GetBuffer()[x + y * size] & 0xffffff)
         fprintf( f, "a[%i]=%i;\n", a, scaled[i]->GetBuffer()[x + y * size] & 0xffffff );
   }
   fprintf( f, "break;\n" );
}
fprintf( f, "}\n}\n" );
```

```
switch( size )
{
case 1:
break;
case 2:
a[1]=1052688;
a[800]=1052688;
a[801]=15724527;
break;
case 3:
a[801]=9737364;
a[802]=8684676;
a[1601]=8684676;
a[1602]=8092539;
break;
case 4:
a[2]=1052688;
a[801]=6513507;
a[802]=9737364;
a[803]=4868682;
a[1600]=1052688;
a[1601]=9737364;
a[1602]=15724527;
a[1603]=7566195;
a[2401]=4868682;
a[2402]=7566195;
a[2403]=3223857;
break;
case 5:
a[2]=526344;
a[3]=526344;
a[801]=3223857;
a[802]=7039851;
a[803]=6513507;
a[804]=2697513;
a[1600]=526344;
a[1601]=7039851;
a[1602]=12434877;
a[1603]=11908533;
a[1604]=5921370;
a[2400]=526344;
a[2401]=6513507;
a[2402]=11908533;
a[2403]=11382189;
a[2404]=5395026;
a[3201]=2697513;
a[3202]=5921370;
a[3203]=5395026;
a[3204]=2171169;
break;
case 6:
a[3]=1052688;
a[801]=2171169;
a[802]=5395026;
a[803]=6513507;
...
a[2406]=4342338;
a[3200]=526344;
a[3201]=4868682;
a[3202]=10263708;
a[3203]=13553358;
a[3204]=12434877;
a[3205]=9211020;
a[3206]=3750201;
a[4001]=3223857;
a[4002]=7039851;
a[4003]=9211020;
a[4004]=9211020;
a[4005]=6513507;
a[4006]=2171169;
a[4802]=2697513;
a[4803]=4342338;
a[4804]=3750201;
a[4805]=2171169;
break;
case 8:
a[3]=526344;
a[4]=1052688;
a[801]=526344;
a[802]=3223857;
a[803]=4868682;
a[804]=5395026;
a[805]=4342338;
a[806]=1579032;
a[1601]=3223857;
a[1602]=6513507;
a[1603]=9211020;
a[1604]=9737364;
a[1605]=8092539;
a[1606]=4868682;
a[1607]=1052688;
a[2400]=526344;
a[2401]=4868682;
a[2402]=9211020;
a[2403]=12434877;
a[2404]=14079702;
a[2405]=10855845;
a[2406]=7039851;
a[2407]=2697513;
a[3200]=1052688;
a[3201]=5395026;
a[3202]=9737364;
a[3203]=14079702;
a[3204]=15724527;
a[3205]=11908533;
a[3206]=7566195;
a[3207]=3223857;
a[4001]=4342338;
a[4002]=8092539;
a[4003]=10855845;
a[4004]=11908533;
...
```

“Madness!? This is INFOMOV!”

/INFOMOV/

END of “Low Level”

next lecture: “Caches (1)”