/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2018 - Lecture 1: “Introduction”

Welcome!

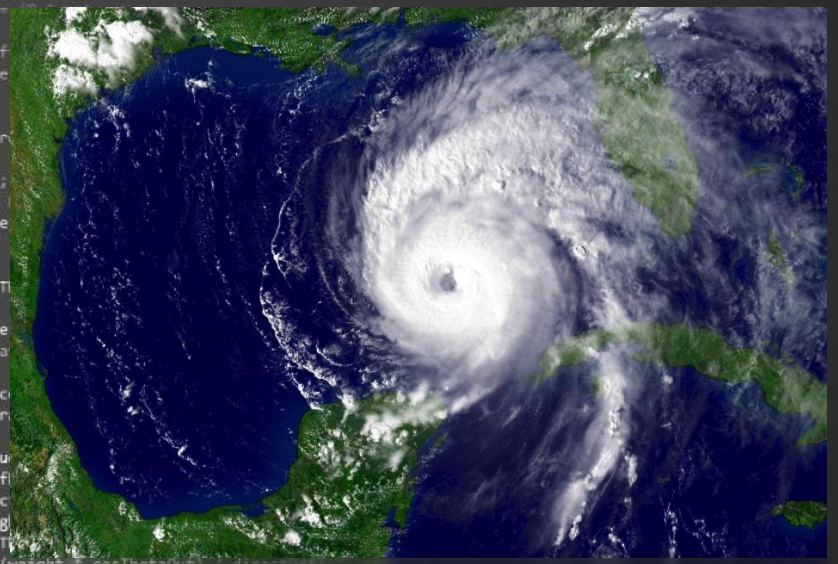
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling

Introduction

Why?

Some problems require the supercomputer of the future.



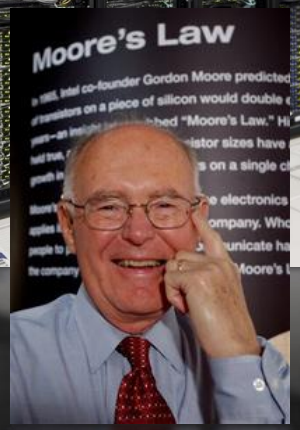
Introduction

Why?

Some problems require the supercomputer of the future.

- Anything that depends on Moore’s Law and time to become feasible.

AlphaGo Parallel, ELO rating 3140
Running on 1202 CPUs, 176 GPUs

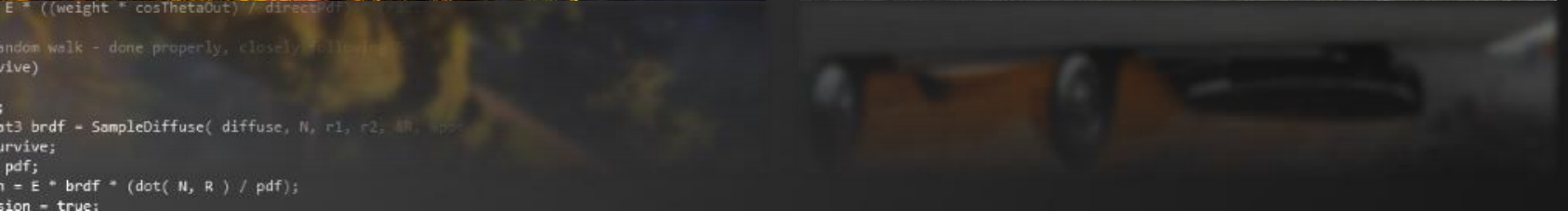


Introduction

Why?

Games want to raise the bar.

- More, better, faster. Also: be scalable.



Introduction

Why?

Some software needs to run on pretty weak hardware.

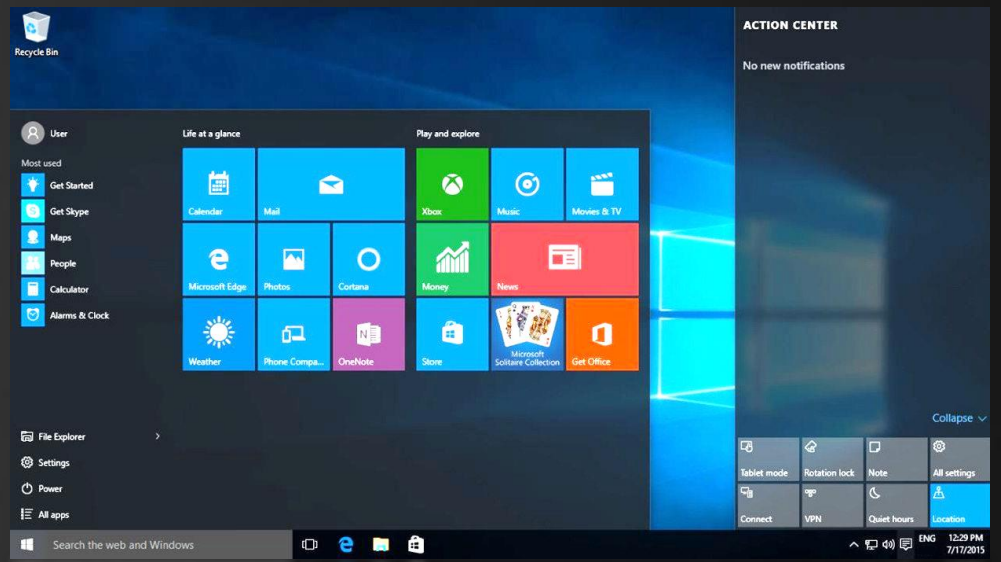
- Limited CPU, limited RAM (limited controls).

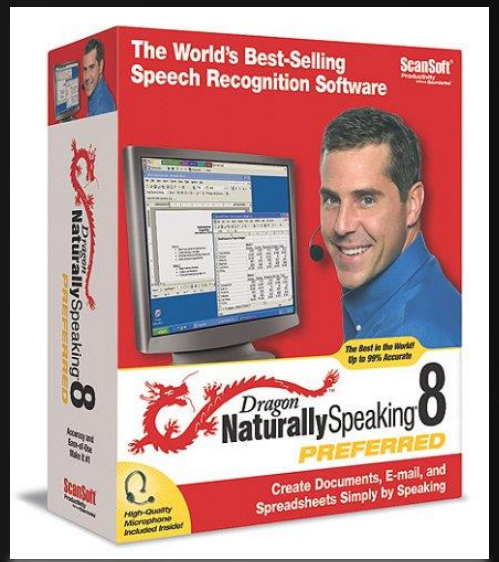
Introduction

Why?

Some software should not use 90% of your CPU.

- Leave room for other applications, be invisible.





Introduction

Why?

Sometimes the cheapest / lowest power CPU is the best.

- What is the lowest end CPU this will still run on? Can we go lower?

Introduction

Why?

Some things are done so frequently, they must be efficient.

- Memory manager
- Garbage collector
- JIT compiler
- Compilers in general
- Image processing in Photoshop
- OS startup / resume
- …
- …

Introduction

What is optimization?

Part of it is:

- INFOB3CC - Concurrency
- INFONW - Computerarchitectuur en netwerken
- INFOB3TC - Talen en compilers

And of course: any course that deals with improving existing algorithms.

Specific purpose of INFOMOV:

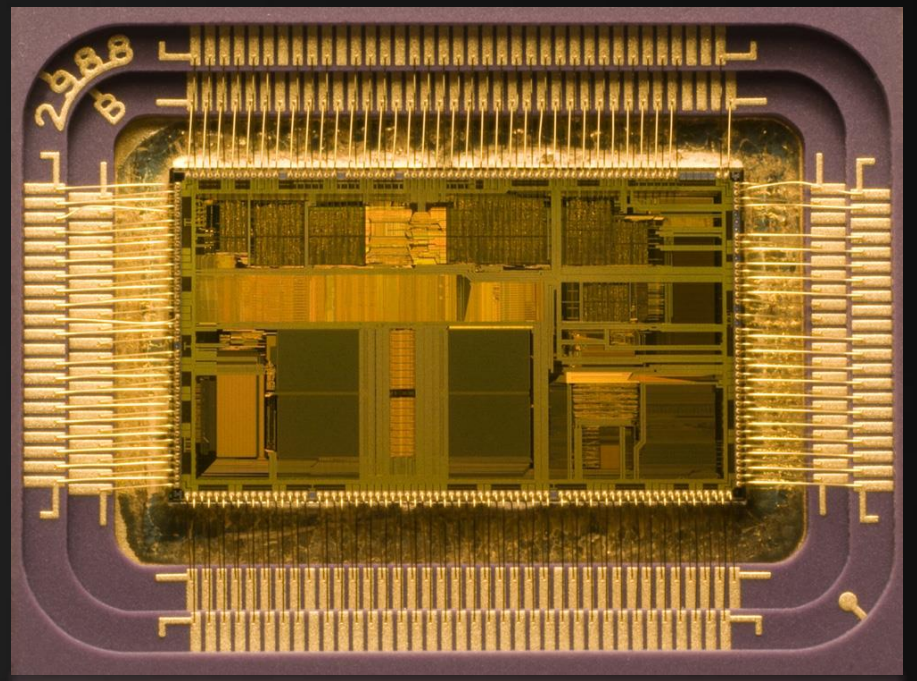
- To gain understanding of performance aspects of the hardware we use;
- To gain an intuition for what affects performance;
- To learn to apply a structured process to improve performance.

Introduction

What is optimization?

Think like a CPU

- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

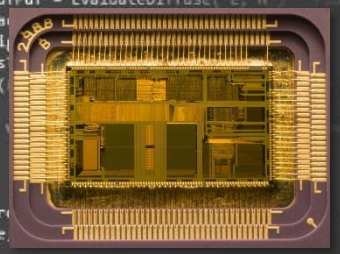


Introduction

What is optimization?

Work smarter, not harder: algorithm scalability

- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory



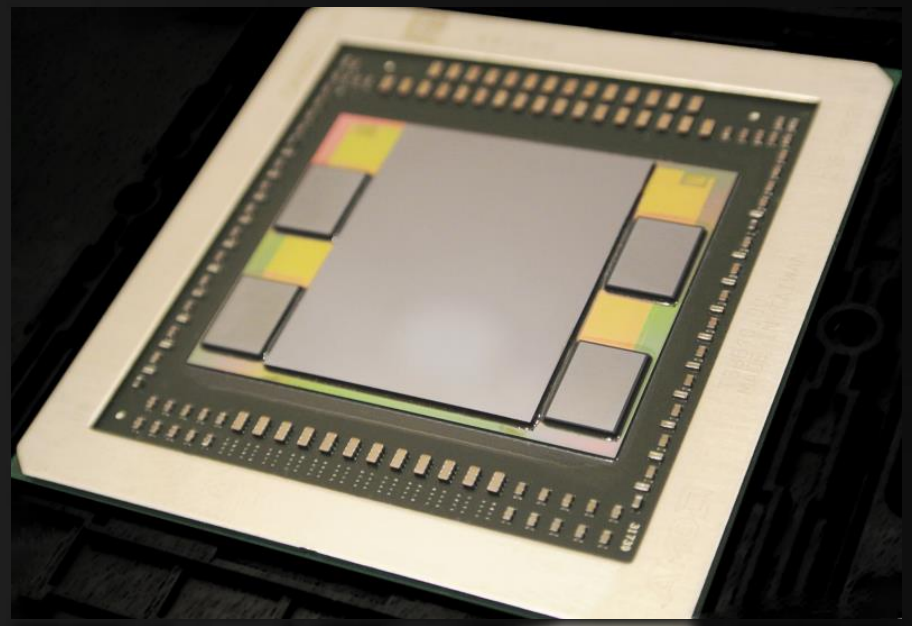
- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

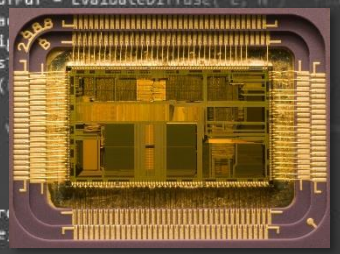
Introduction

What is optimization?

Memory hierarchy: caches

- Cache architecture
- Cache lines
- Hits, misses and collisions
- Eviction policies
- Prefetching
- Cache-oblivious
- Data-centric programming





- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

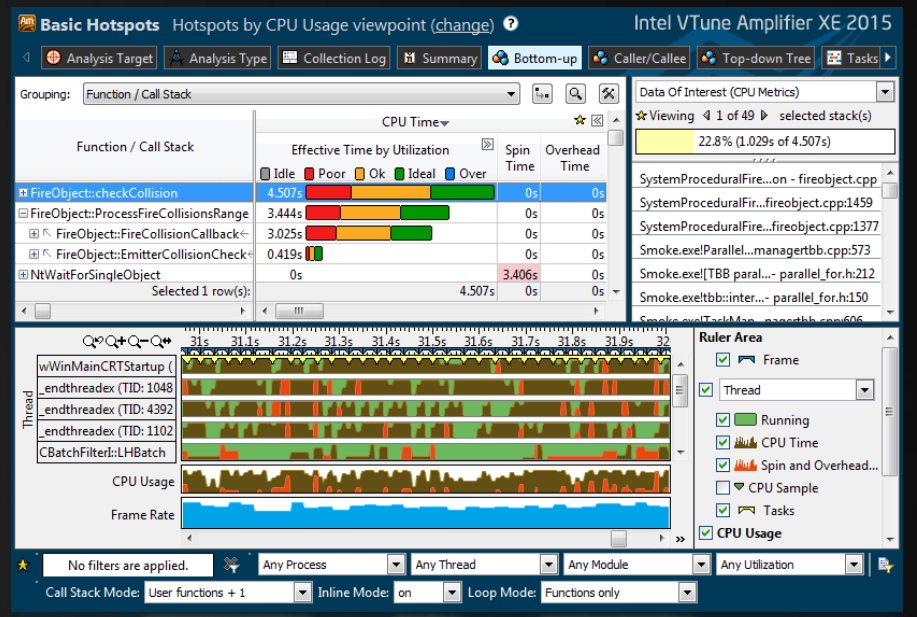
- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory

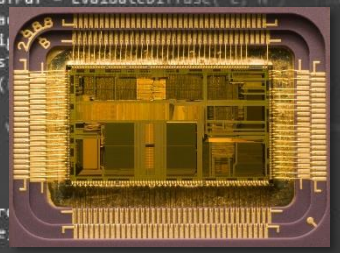
Introduction

What is optimization?

Don’t assume, measure

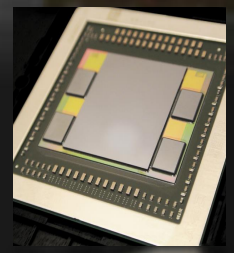
- Profilers
- Interpreting profiling data
- Instrumentation
- Bottlenecks
- Steering optimization effort





- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory



- Cache architecture
- Cache lines
- Hits, misses and collisions
- Eviction policies
- Prefetching
- Cache-oblivious
- Data-centric programming

Introduction

What is optimization? – Project Management

Keeping code maintainable

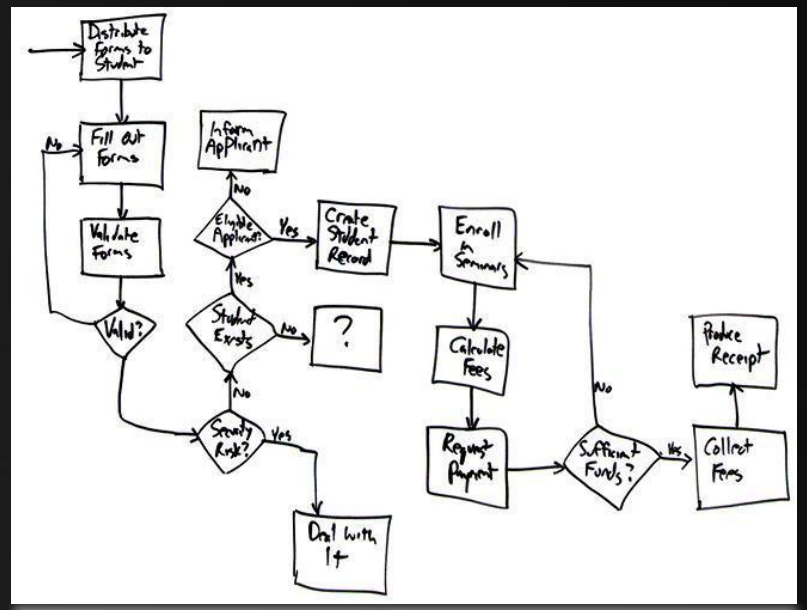
- Pareto principle / 80-20 rule: roughly 80% of the effects are caused by 20% of the causes.
- 1% of the code takes 99% of the time.

“The curse of premature optimization”

- Optimization, rule 1: “Don’t do it”.
- Rule 2 (for experts only!), “Don’t do it yet”.

Optimization as a deliberate process

- *Get predictable gains using a consistent approach.*



Introduction

What is optimization?

“Perceived Performance”

1. Wait for user input
2. Respond to user input *as quickly as possible*
3. Execute requested operation.

Introduction

At the end of this course:

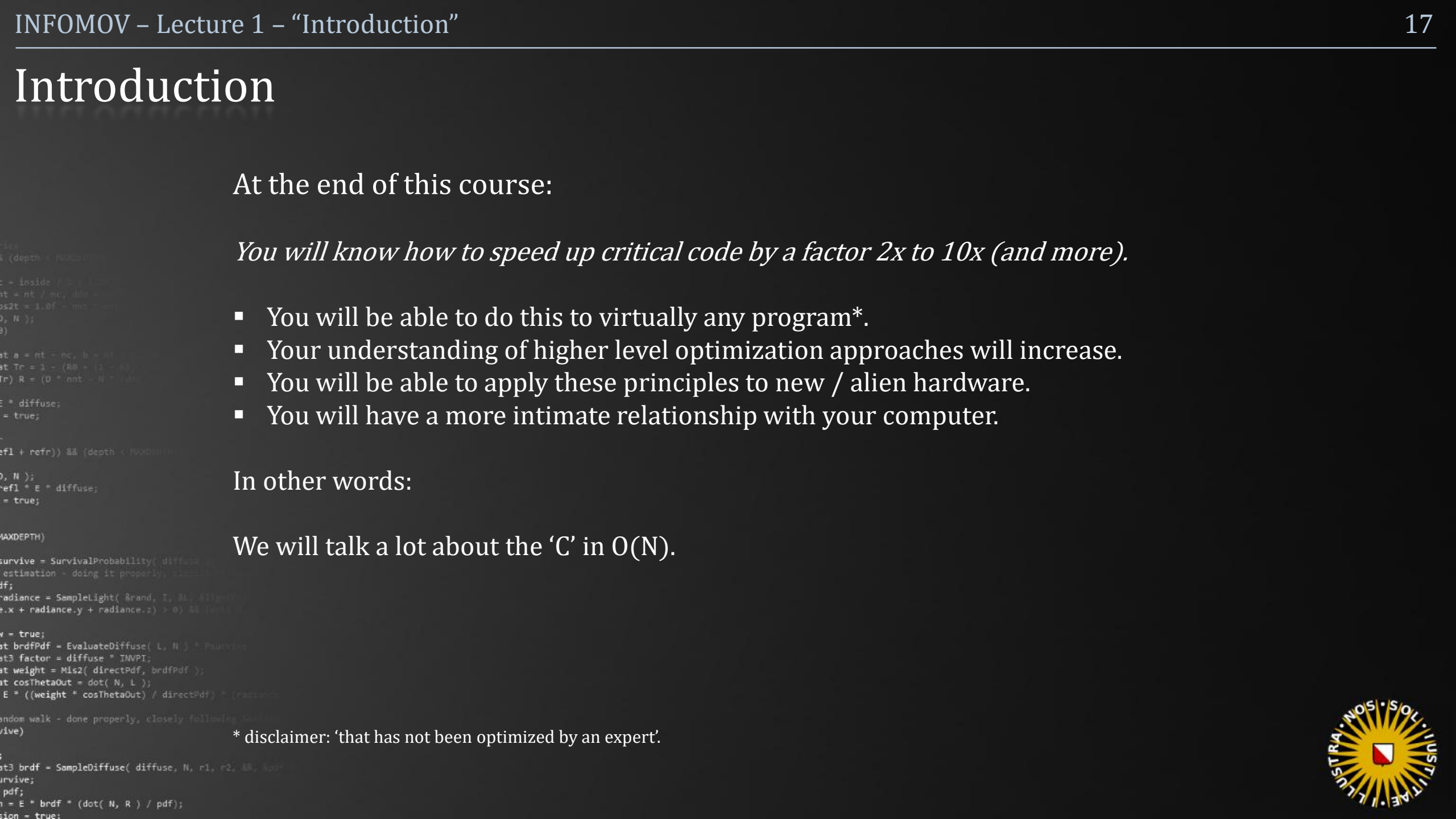
You will know how to speed up critical code by a factor 2x to 10x (and more).

- You will be able to do this to virtually any program*.
- Your understanding of higher level optimization approaches will increase.
- You will be able to apply these principles to new / alien hardware.
- You will have a more intimate relationship with your computer.

In other words:

We will talk a lot about the ‘C’ in O(N).

* disclaimer: ‘that has not been optimized by an expert’.



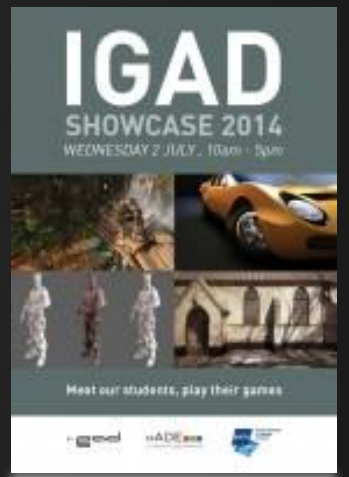
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling

Formalities

Lecturer

Jacco Bikker
j.bikker@uu.nl
Room 4.24 BBG



Formalities

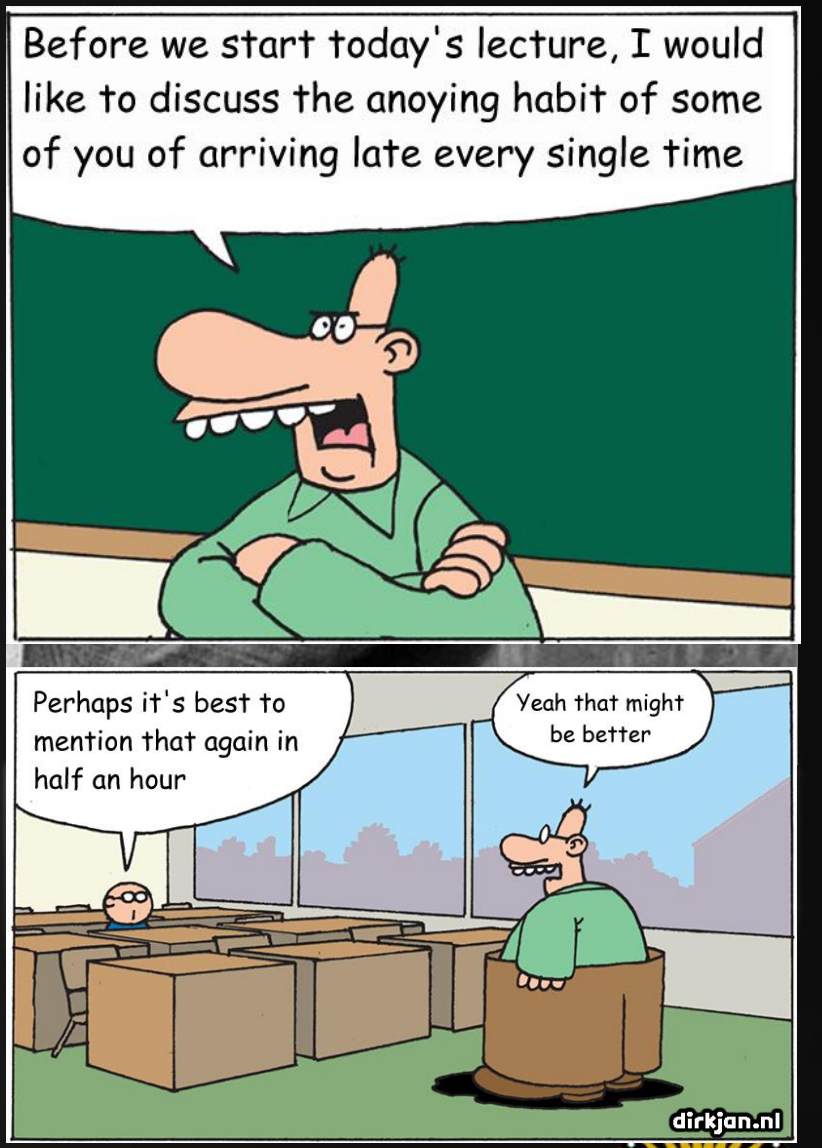
Course Layout

9 weeks + exam week:

- 2 lectures per week (for exceptions: see website)
- 1 guest lecture
- Lectures start at 09:00...
- Working class starts at 09:00, lecture at 11:00. ☺

Assessment:

- 3 assignments (20% each, individual or pairs);
- 1 final assignment (40%, individual or pairs);
- 1 final theory exam.



Formalities

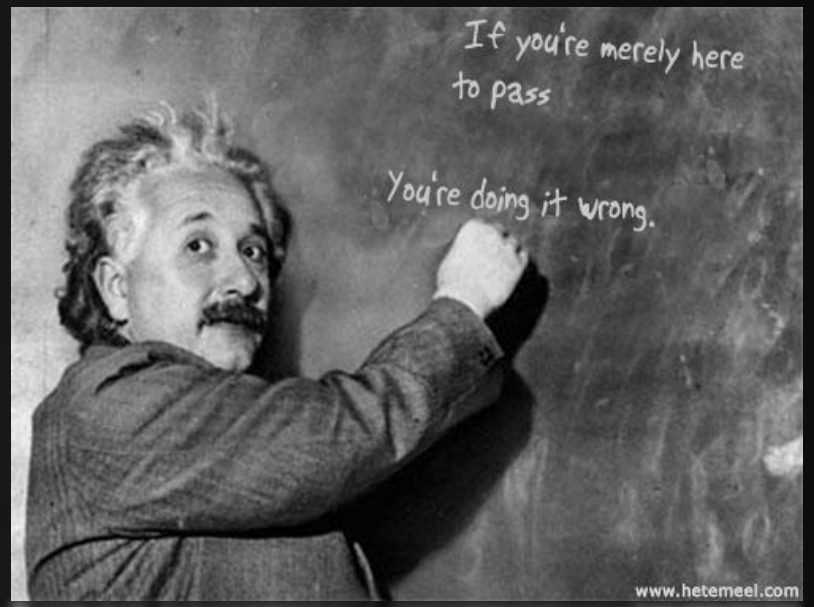
Prerequisites

C++
English

Hardware / software

You’ll need access to a computer with a CPU that supports SSE2 and OpenCL.
Obtaining VTune (Intel CPU) or CodeXL (AMD CPU) is beneficial (VTune is free to try for 30 days).
We will use Visual Studio 2017 (community edition).

Other tools will (also) be free.



Formalities

Literature

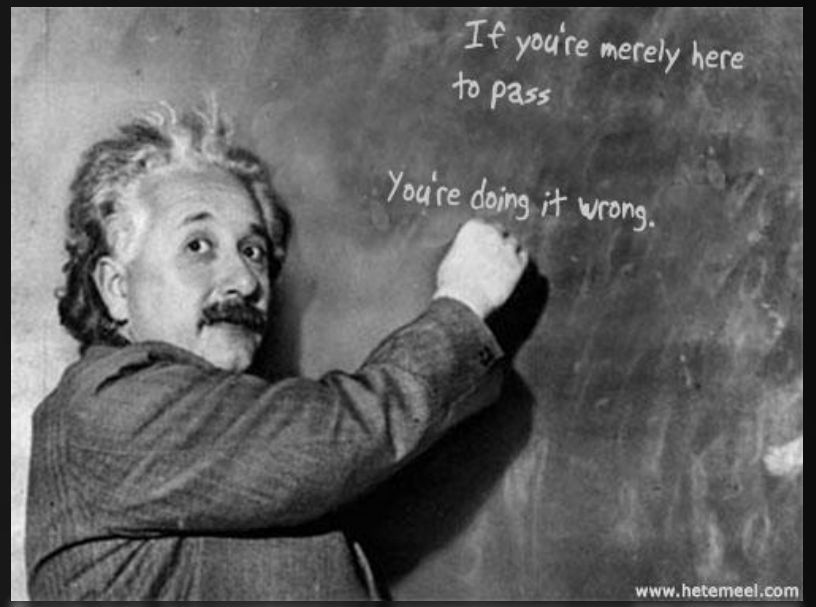
No book!
But that doesn’t mean you won’t be reading.

Main documents:

Agner Fog, 2004-2018, “Optimizing Software in C++”
(also see his website: http://agner.org)

Ulrich Drepper, 2007, “What Every Programmer Should Know About Memory”

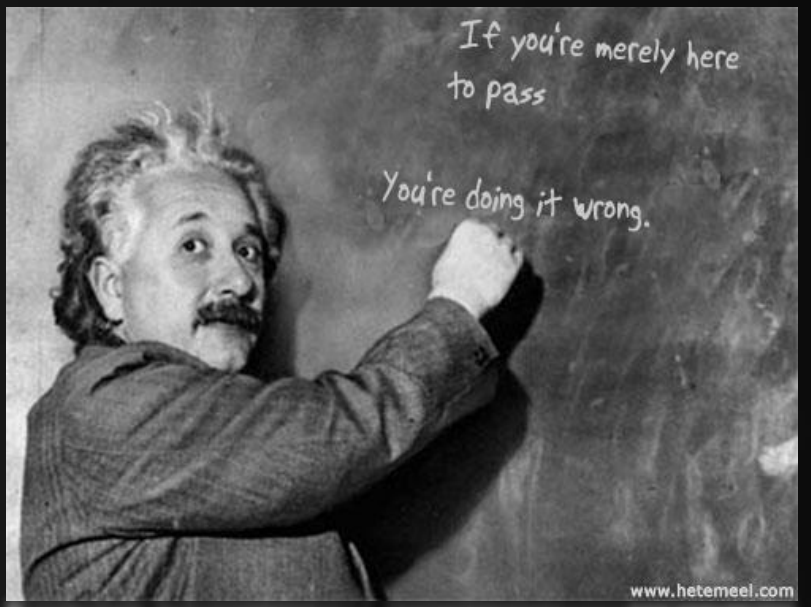
You are encouraged to do research into specific topics of interest yourself, and to report on this in class.



Formalities

Audience

Any computer science student
(with a slight bias towards games)



Make sure <u>you</u> get as much as possible out of this course. This automatically includes a free pass.

Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling

Overview

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.

Overview

Consistent Approach

(0.) Determine optimization requirements

- Target hardware (or range of hardware)
- Target performance
- Time available for optimization
- Constraints related to maintainability / portability
- …

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 steps 6 and 7 until time runs out
9. Report.

From here on, we will assume that:

- the code is ‘done’ (feature complete);
- a speed improvement is required;
- we have a finite amount of time for this.

Overview

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 steps 6 and 7 until time runs out
9. Report.

Overview

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 / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
   - caching, data-centric programming,
   - removing superfluous functionality and precision,
   - aligning data to cache lines, vectorization,
   - checking compiler output, fixed point arithmetic,
   - …
8. Repeat steps 6 and 7 until time runs out
9. Report.

Overview

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 steps 6 and 7 until time runs out
9. Report.

Profiling

High Level

Basic Low Level

Cache & Memory

Data-centric

Compilers

Fixed-point Arithmetic

CPU architecture

SIMD

GPGPU

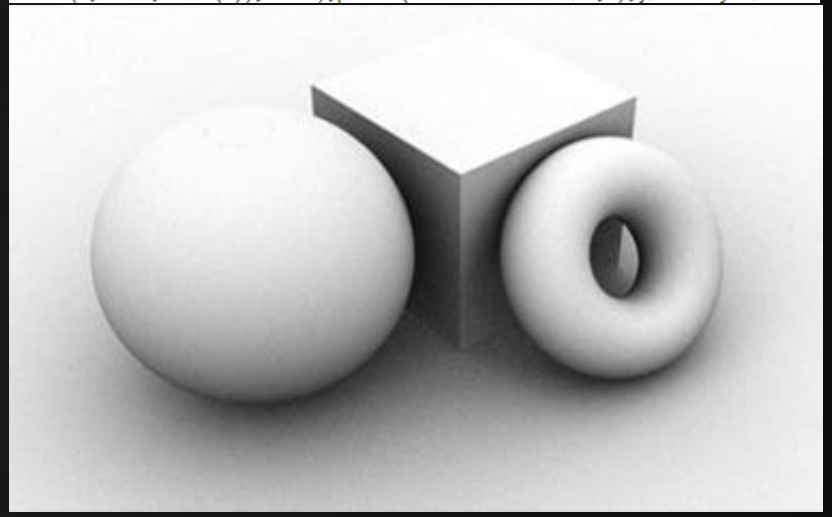
Overview

Assembler

In this course, we will <u>not</u> write assembler:

- It takes a pro to outperform the compiler
- You will be fighting the compiler
- You will have to redo the optimization for every target processor
- Maintainability will be zero.

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*
A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec vunit(A)vec A;{return vcomb(1./sqrt(
vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s
->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l-->sph)if((e=l
->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e
,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*
eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,
color,vcomb(s->kl,U,black))));}main(){printf("%d %d\n",32,32);while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*minray!*/
```

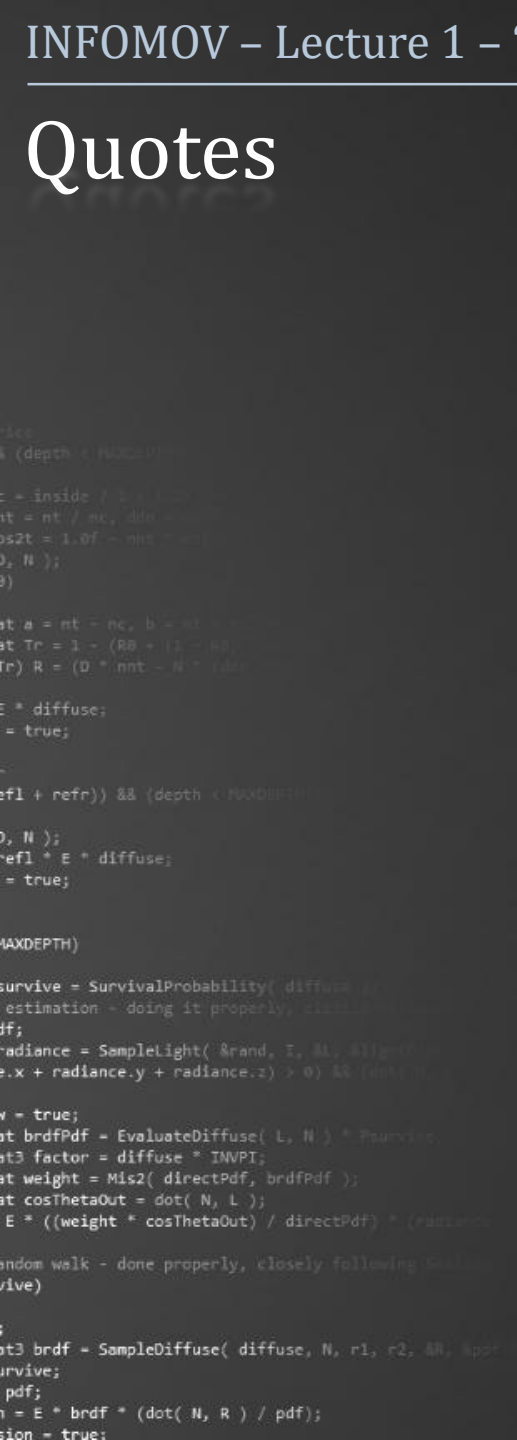


Quotes

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”
(Donald Knuth)

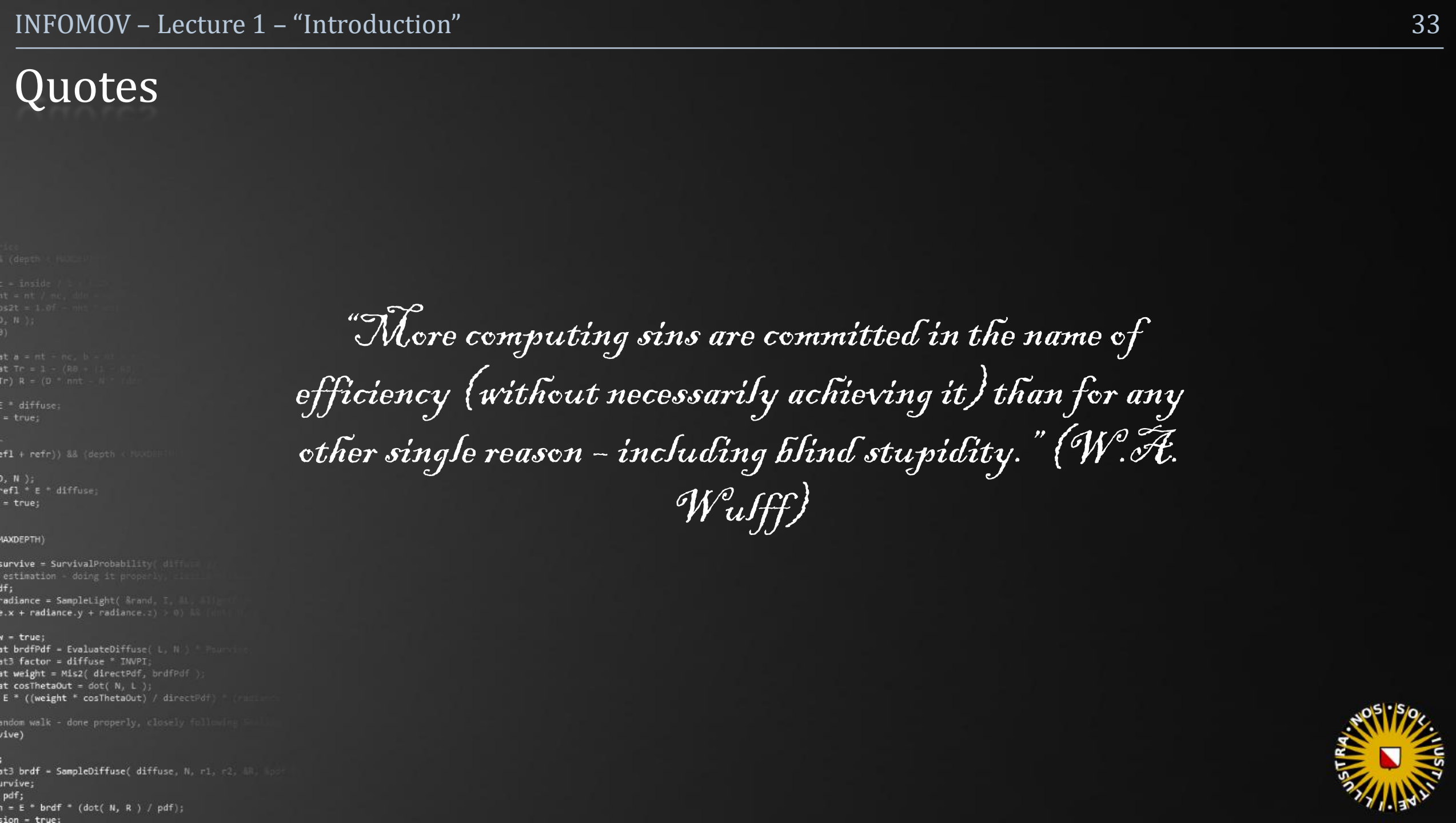
Quotes

“A significant improvement in performance can often be achieved by solving only the actual problem and removing extraneous functionality.” (Wikipedia)

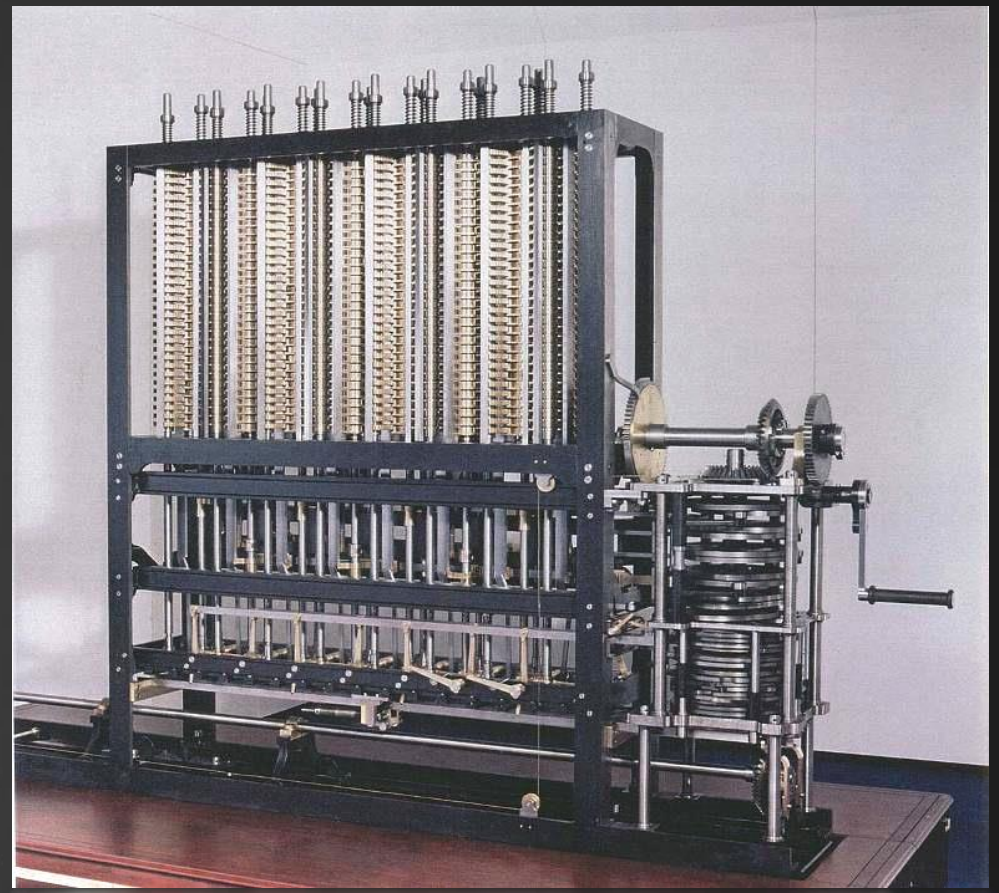


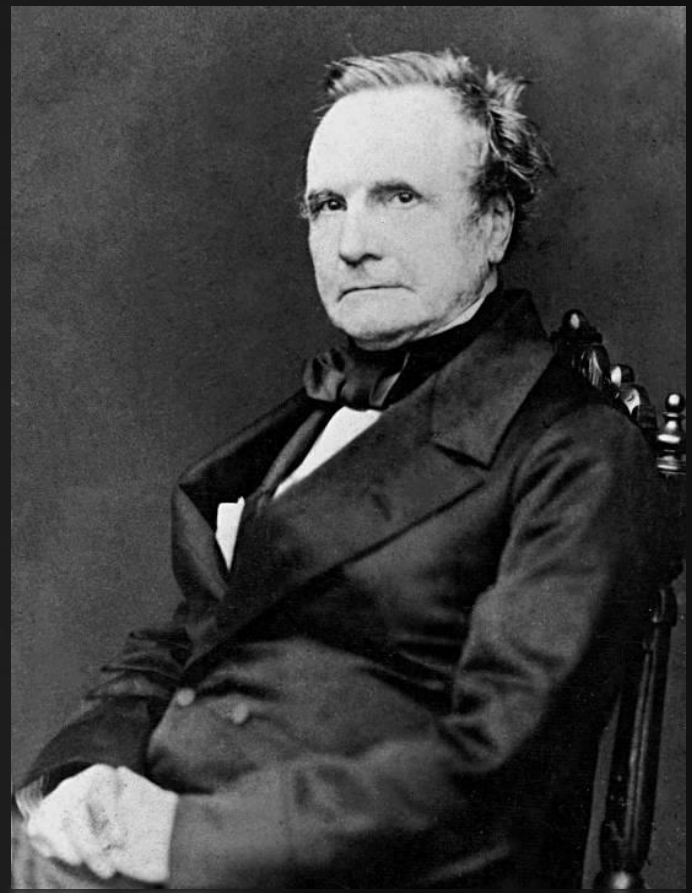
Quotes

“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)



Quotes





Quotes

“Dear Charles,

In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them (...).

One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Therefore, one should attend INFOMOV and learn from it.

Love, Ada.”

Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling

Never Assume

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.

Never Assume

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*.

Never Assume

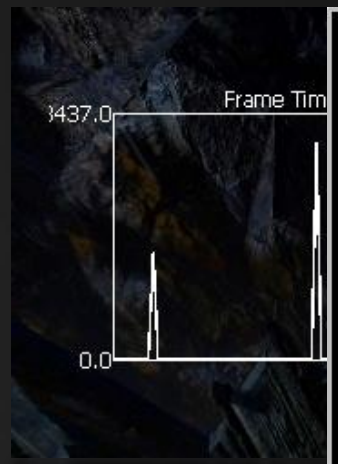
Profiling

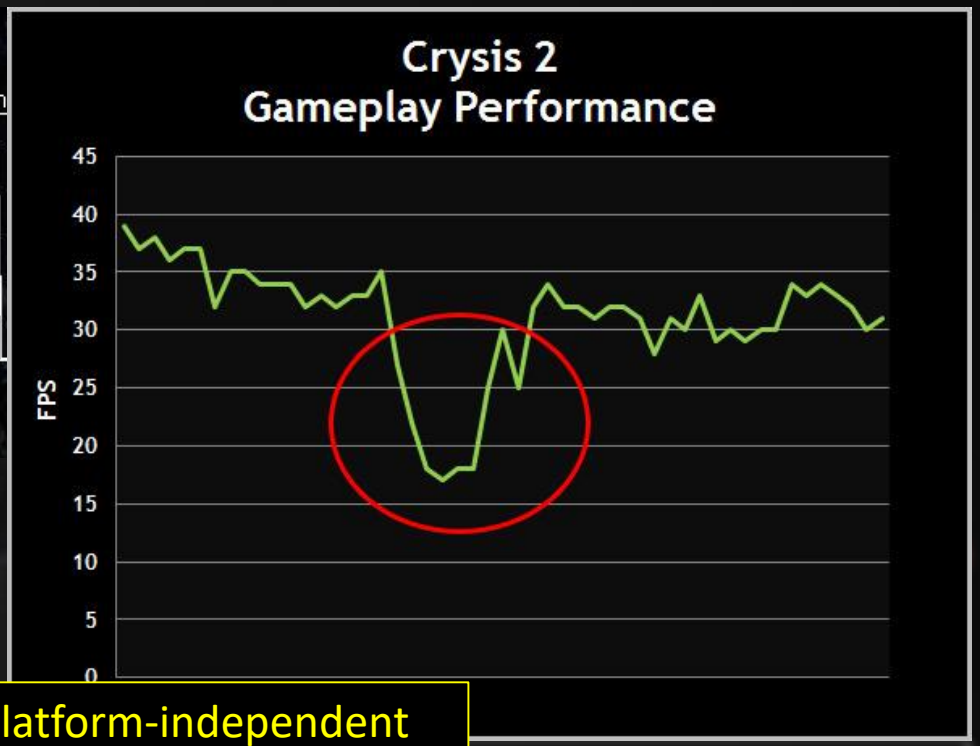
Measuring application performance

- Using external tools
- Using timers in the code

Measurements:

- How much time is spent were? (inclusive / exclusive, cycles, percentage)
- How often is each function called?
- Low level behavior: stalls / latencies, branch mispredictions, occupation, …
- Performance over time: lag, spikes, stutter





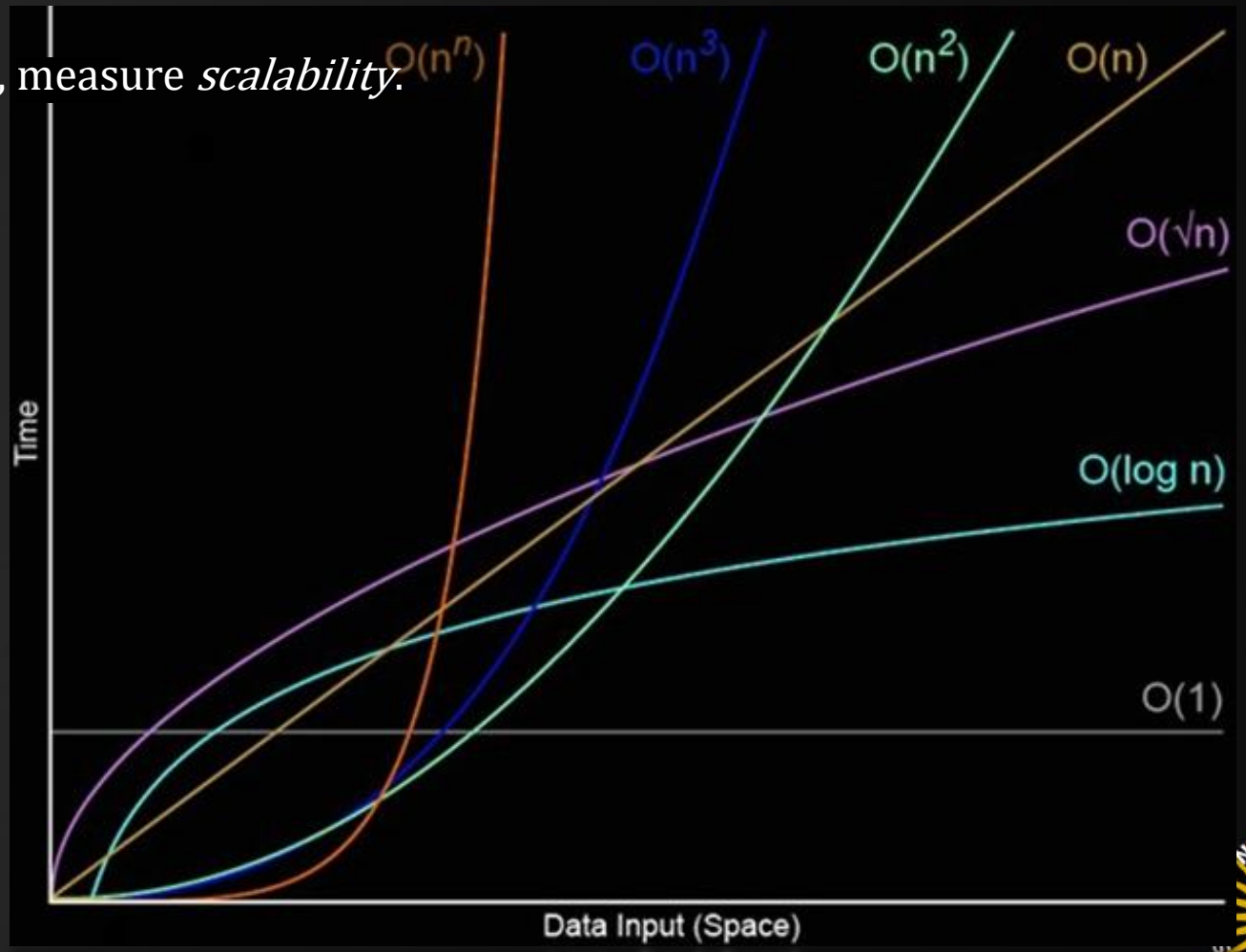
Platform-independent

Platform-dependent

Never Assume

What if the goal is to have a 10x larger army in your RTS?

Don’t just measure performance, measure *scalability*.



Never Assume

Profiling – getting accurate results

A profiler needs information about your code:
this is typically available in *debug* builds.

However:
Debug builds have very different performance characteristics, for many reasons. We need to profile in *release* mode.

Enabling debug information in release mode in Visual Studio:

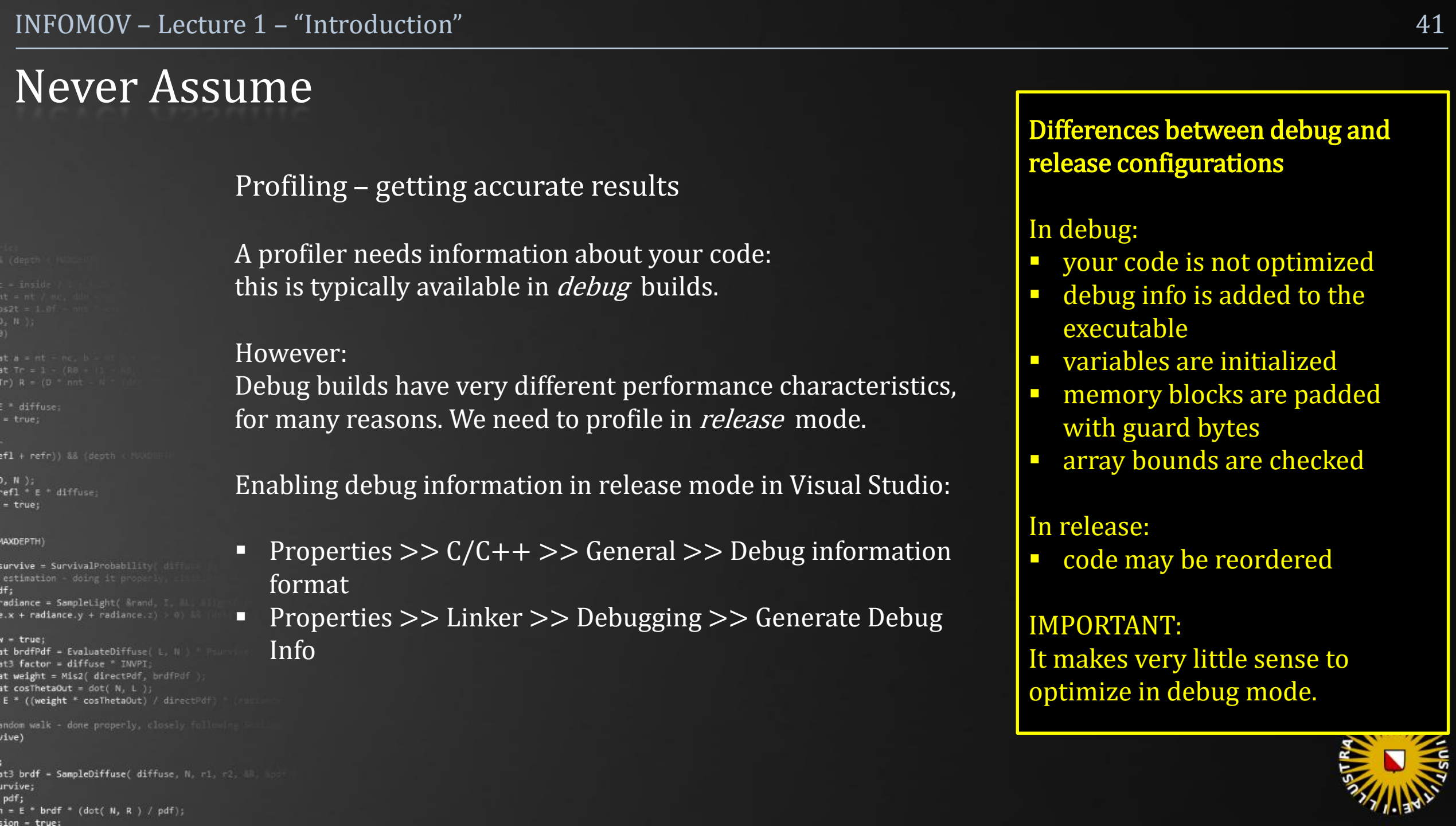
- Properties >> C/C++ >> General >> Debug information format
- Properties >> Linker >> Debugging >> Generate Debug Info

Differences between debug and release configurations

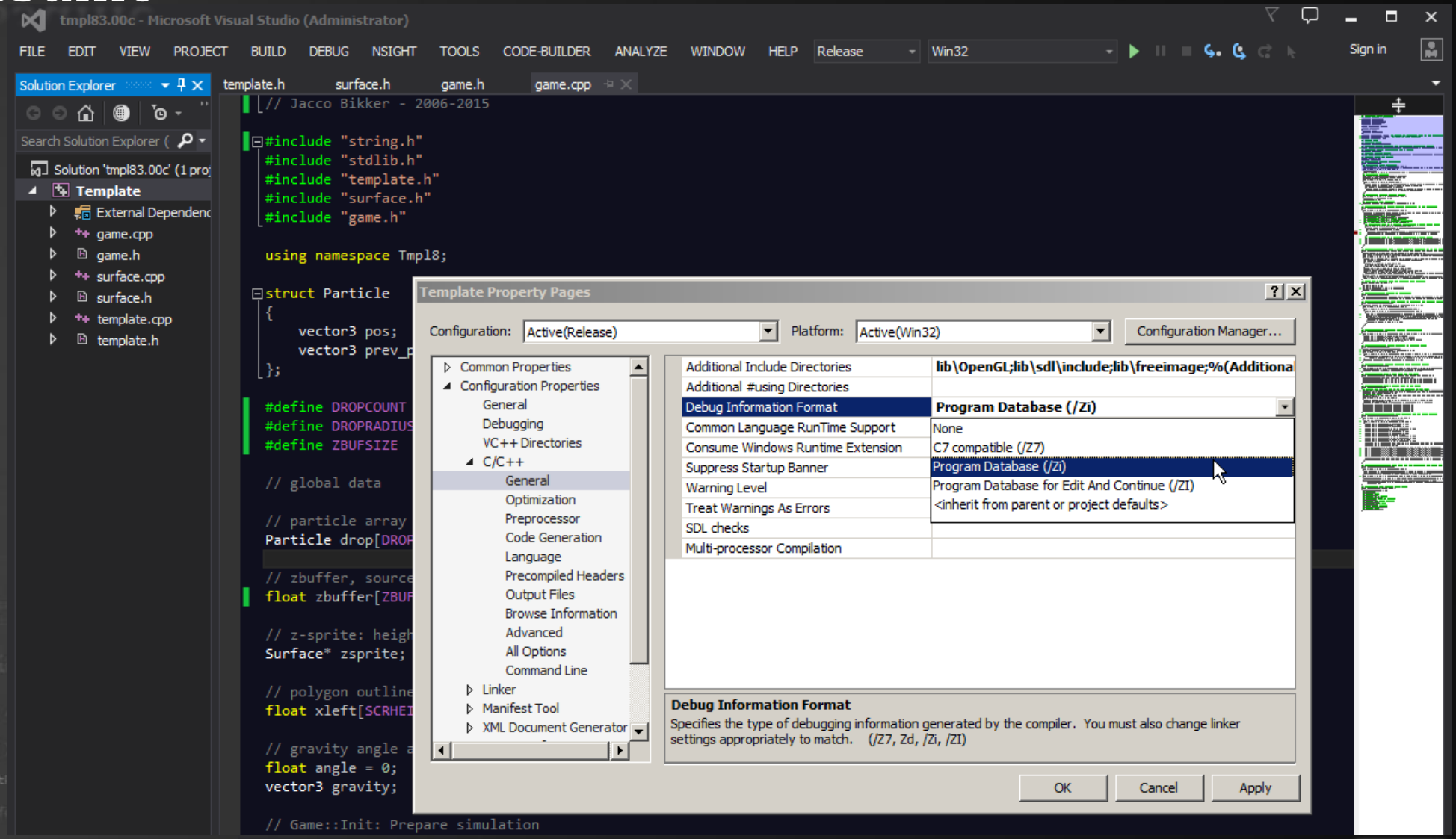
In debug:
- your code is not optimized
- debug info is added to the executable
- variables are initialized
- memory blocks are padded with guard bytes
- array bounds are checked

In release:
- code may be reordered

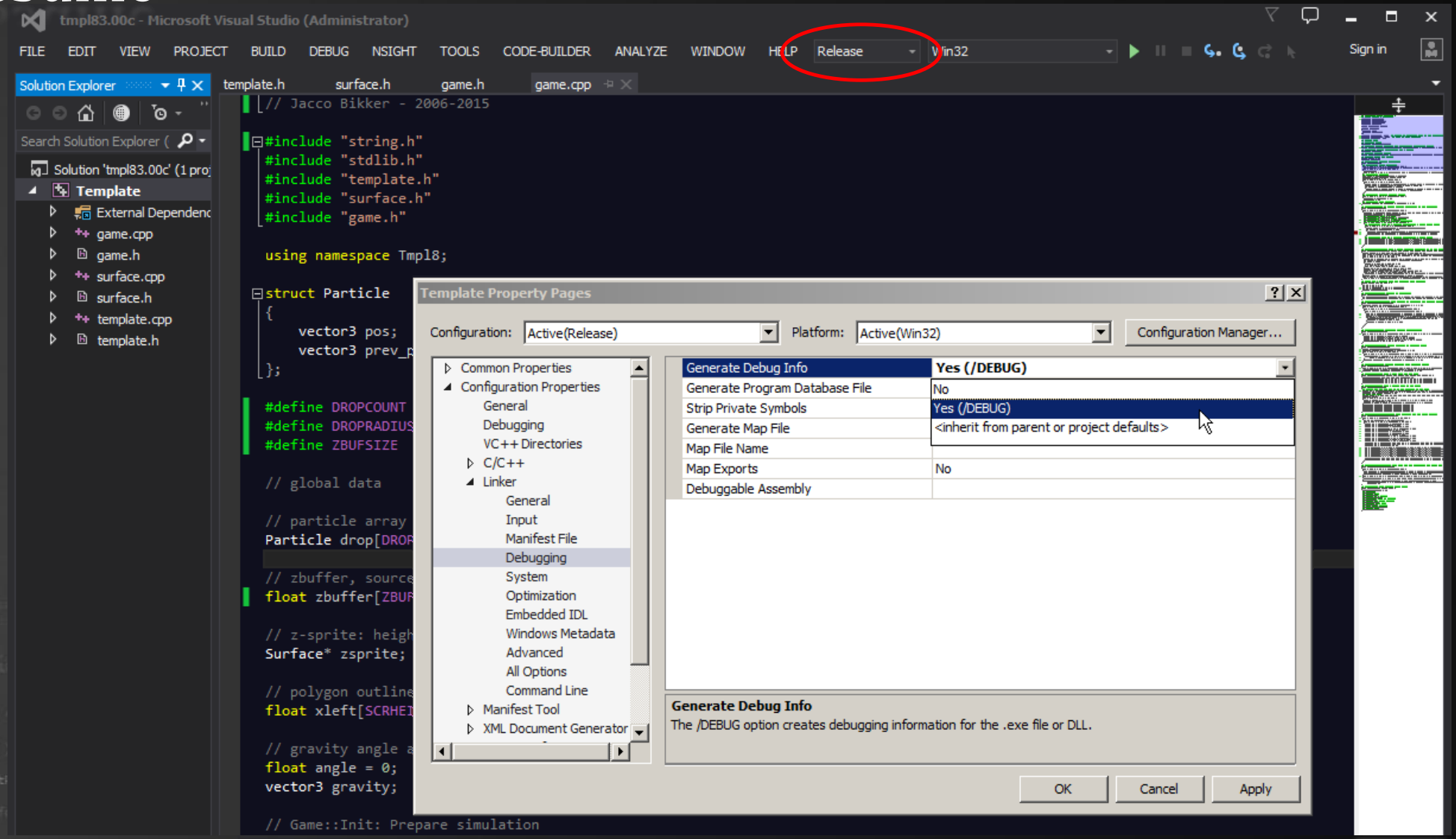
IMPORTANT:
It makes very little sense to optimize in debug mode.



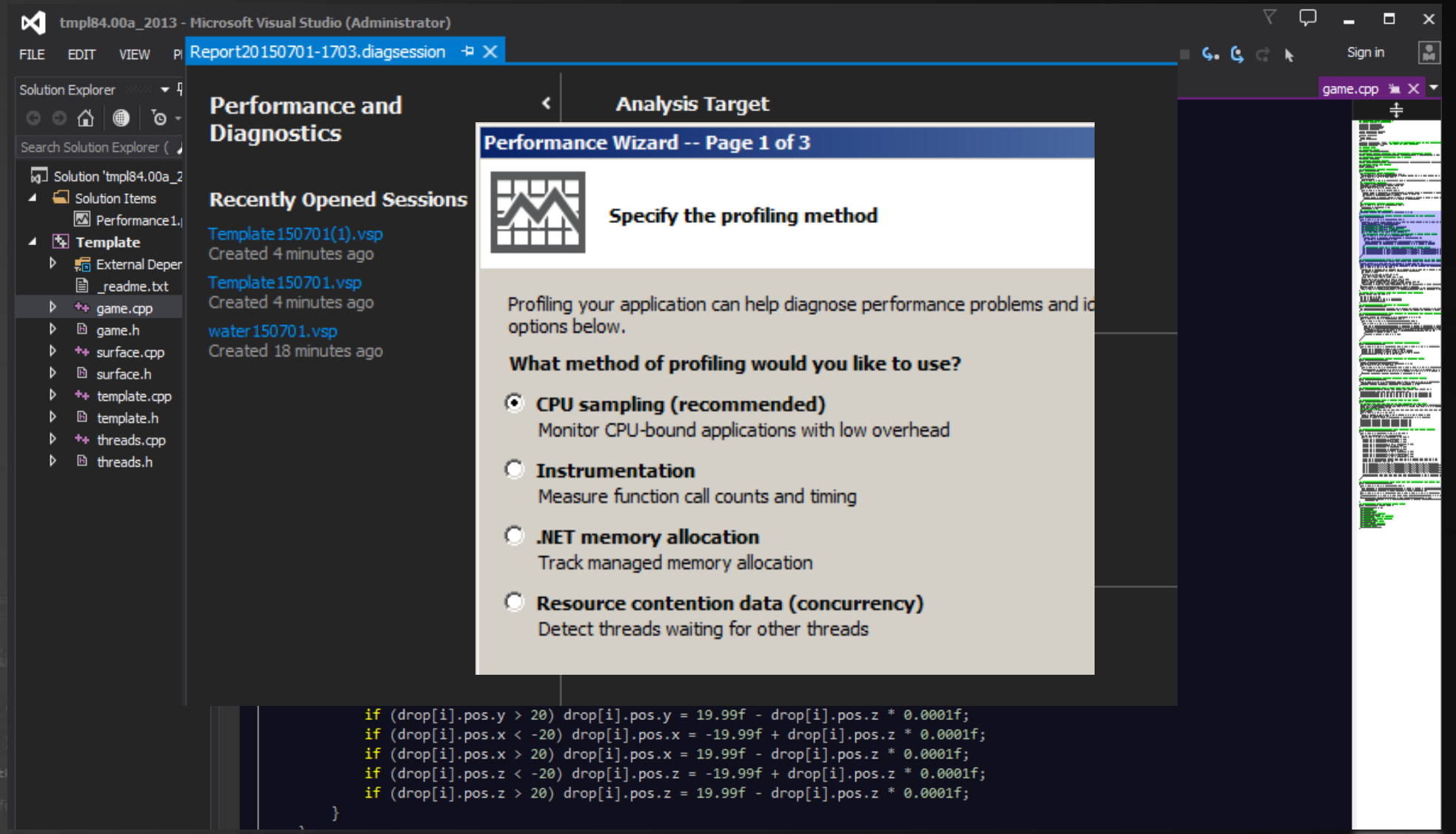
Never Assume



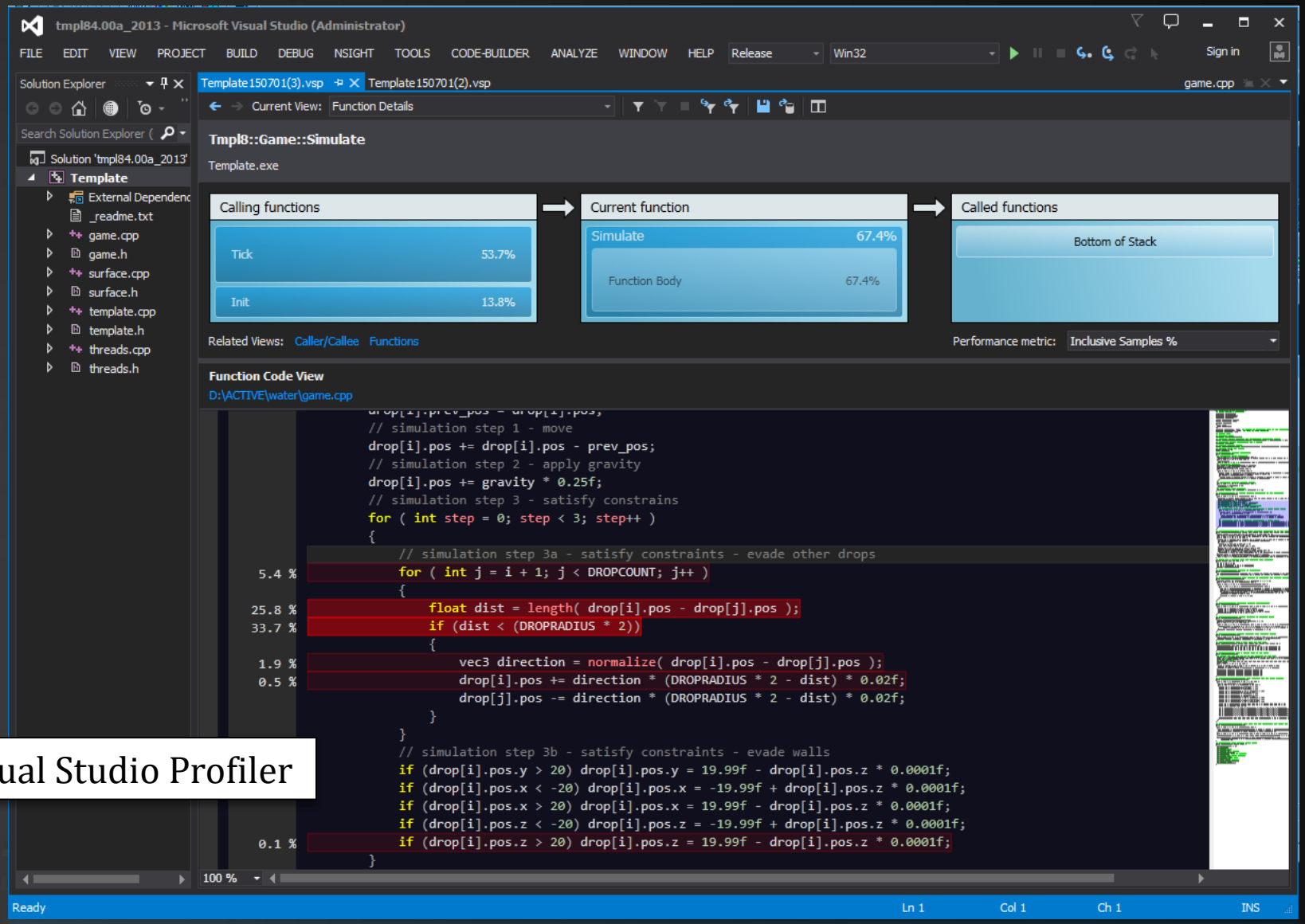
Never Assume



Tools

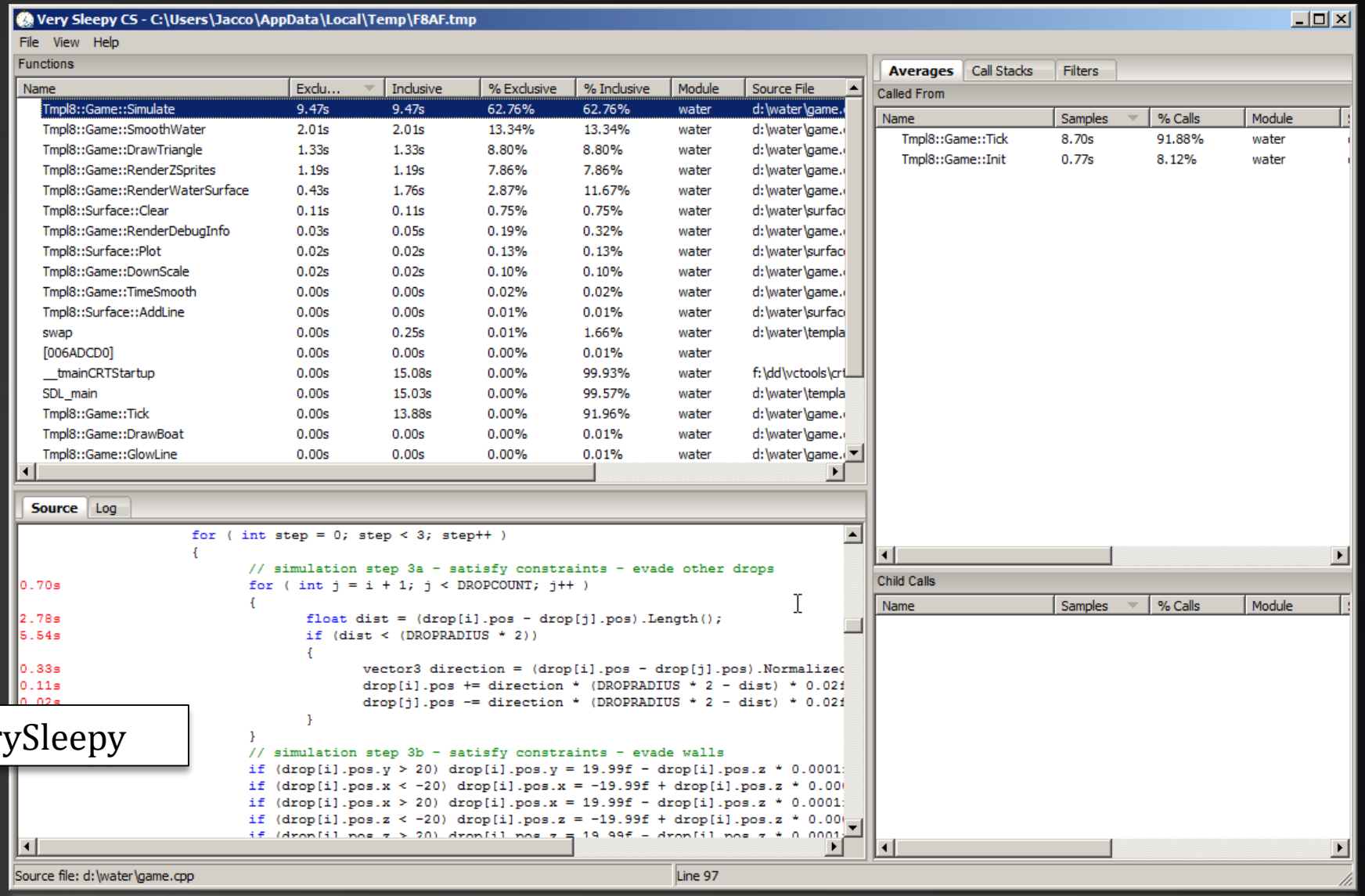

tmpl84.00a_2013 - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW
Report20150701-1703.diagsession
Sign in
Solution Explorer
Search Solution Explorer
Solution 'tmpl84.00a_2
Solution Items
Performance1.
Template
External Deper
_readme.txt
game.cpp
game.h
surface.cpp
surface.h
template.cpp
template.h
threads.cpp
threads.h
Performance and Diagnostics
Recently Opened Sessions
Template150701(1).vsp
Created 4 minutes ago
Template150701.vsp
Created 4 minutes ago
water150701.vsp
Created 18 minutes ago
Analysis Target
Performance Wizard -- Page 1 of 3
Specify the profiling method
Profiling your application can help diagnose performance problems and id
options below.
What method of profiling would you like to use?
CPU sampling (recommended)
Monitor CPU-bound applications with low overhead
Instrumentation
Measure function call counts and timing
.NET memory allocation
Track managed memory allocation
Resource contention data (concurrency)
Detect threads waiting for other threads
game.cpp
if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.0001f;
if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.0001f;
if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.0001f;
if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.0001f;
if (drop[i].pos.z > 20) drop[i].pos.z = 19.99f - drop[i].pos.z * 0.0001f;

Tools



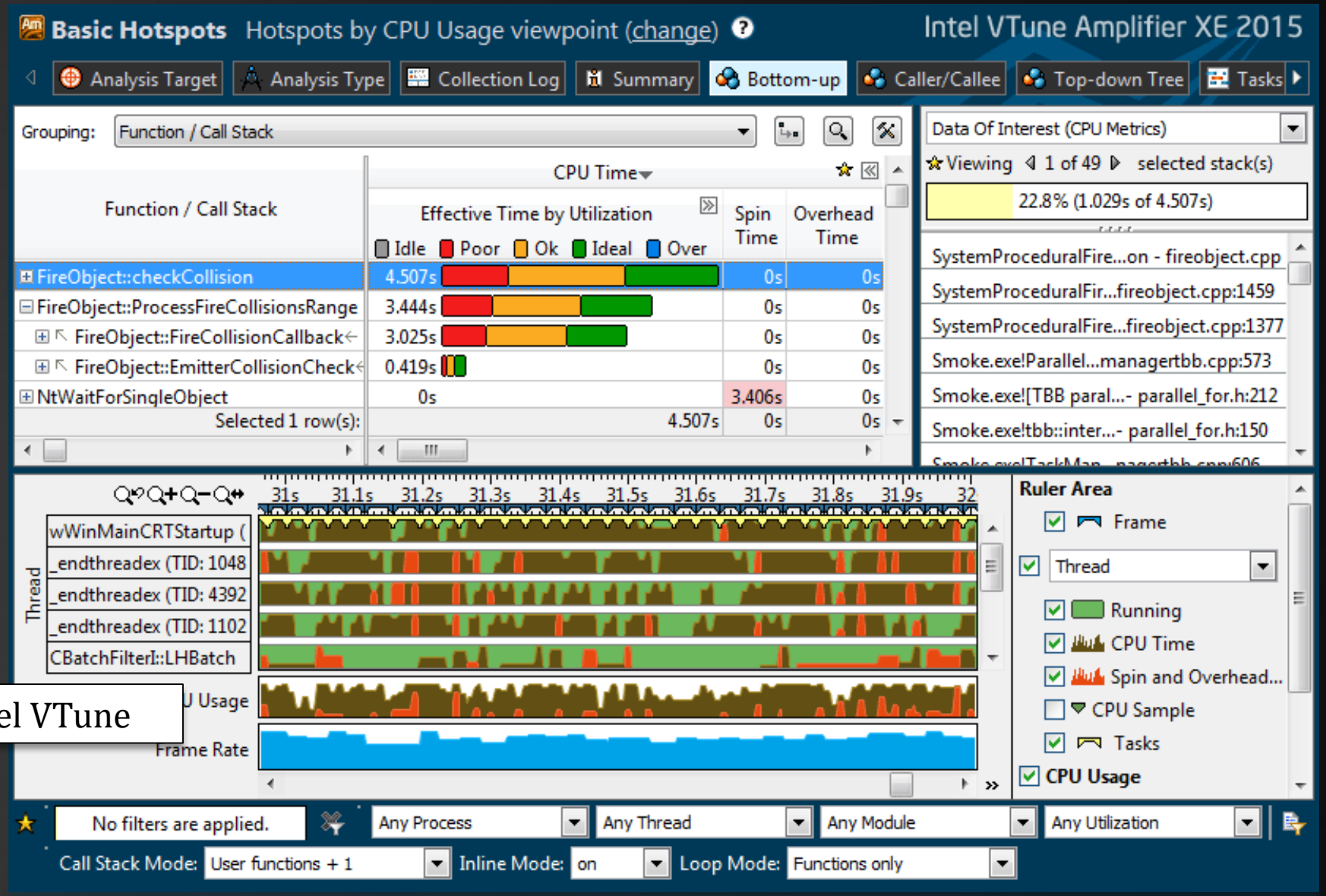
Visual Studio Profiler

Tools



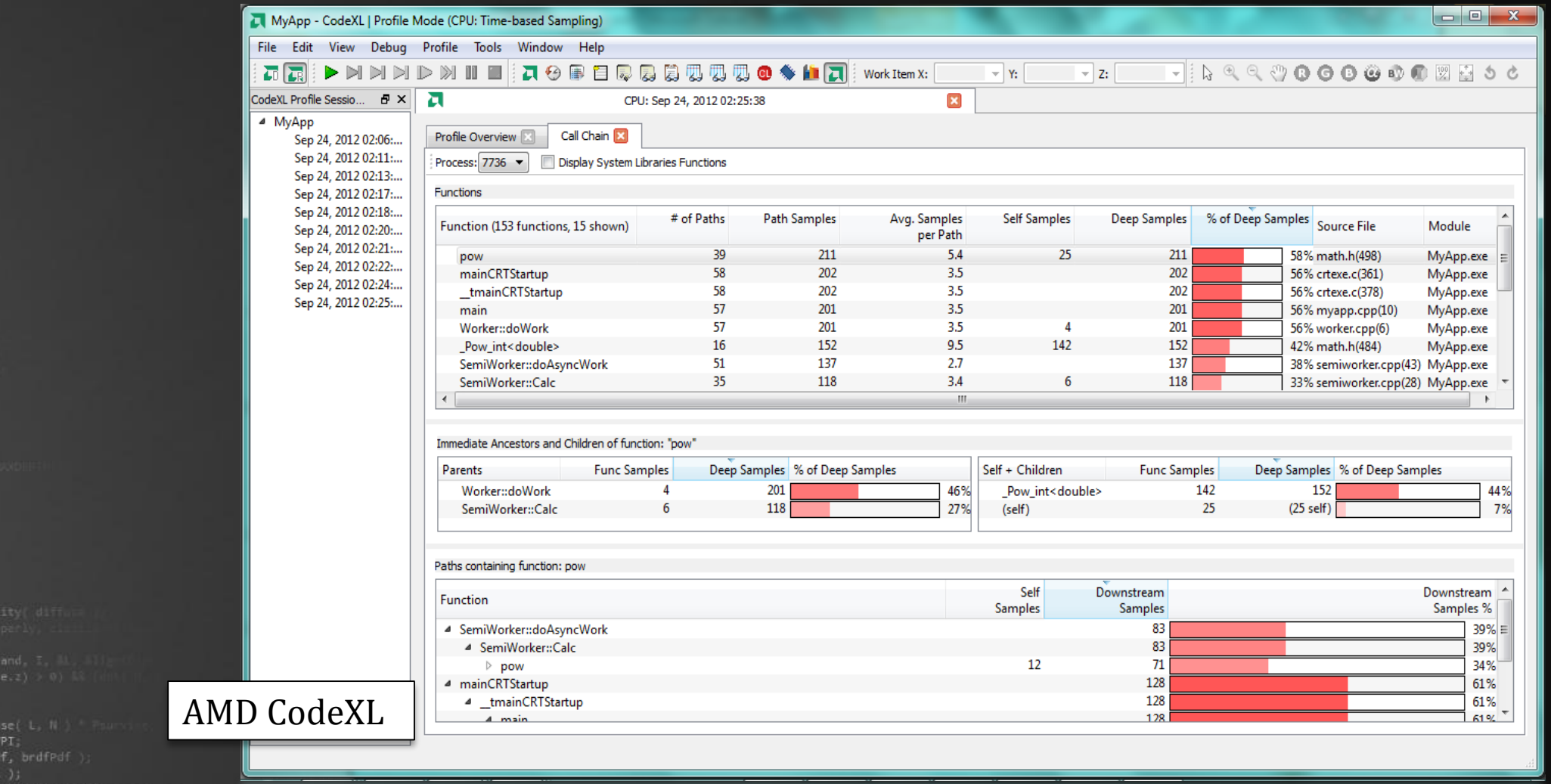
VerySleepy

Tools



Intel VTune

Tools


MyApp - CodeXL | Profile Mode (CPU: Time-based Sampling)
File Edit View Debug Profile Tools Window Help
CodeXL Profile Sessio...
MyApp
Sep 24, 2012 02:06:...
Sep 24, 2012 02:11:...
Sep 24, 2012 02:13:...
Sep 24, 2012 02:17:...
Sep 24, 2012 02:18:...
Sep 24, 2012 02:20:...
Sep 24, 2012 02:21:...
Sep 24, 2012 02:22:...
Sep 24, 2012 02:24:...
Sep 24, 2012 02:25:...
CPU: Sep 24, 2012 02:25:38
Profile Overview
Call Chain
Process: 7736
Display System Libraries Functions
Functions
Function (153 functions, 15 shown) # of Paths Path Samples Avg. Samples per Path Self Samples Deep Samples % of Deep Samples Source File Module
pow 39 211 5.4 25 211 58% math.h(498) MyApp.exe
mainCRTStartup 58 202 3.5 202 56% crtexe.c(361) MyApp.exe
__tmainCRTStartup 58 202 3.5 202 56% crtexe.c(378) MyApp.exe
main 57 201 3.5 201 56% myapp.cpp(10) MyApp.exe
Worker::doWork 57 201 3.5 4 201 56% worker.cpp(6) MyApp.exe
_Pow_int<double> 16 152 9.5 142 152 42% math.h(484) MyApp.exe
SemiWorker::doAsyncWork 51 137 2.7 137 38% semiworker.cpp(43) MyApp.exe
SemiWorker::Calc 35 118 3.4 6 118 33% semiworker.cpp(28) MyApp.exe
Immediate Ancestors and Children of function: "pow"
Parents Func Samples Deep Samples % of Deep Samples
Worker::doWork 4 201 46%
SemiWorker::Calc 6 118 27%
Self + Children Func Samples Deep Samples % of Deep Samples
_Pow_int<double> 142 152 44%
(self) 25 (25 self) 7%
Paths containing function: pow
Function Self Samples Downstream Samples Downstream Samples %
SemiWorker::doAsyncWork 83 39%
SemiWorker::Calc 83 39%
pow 12 71 34%
mainCRTStartup 128 61%
__tmainCRTStartup 128 61%
AMD CodeXL

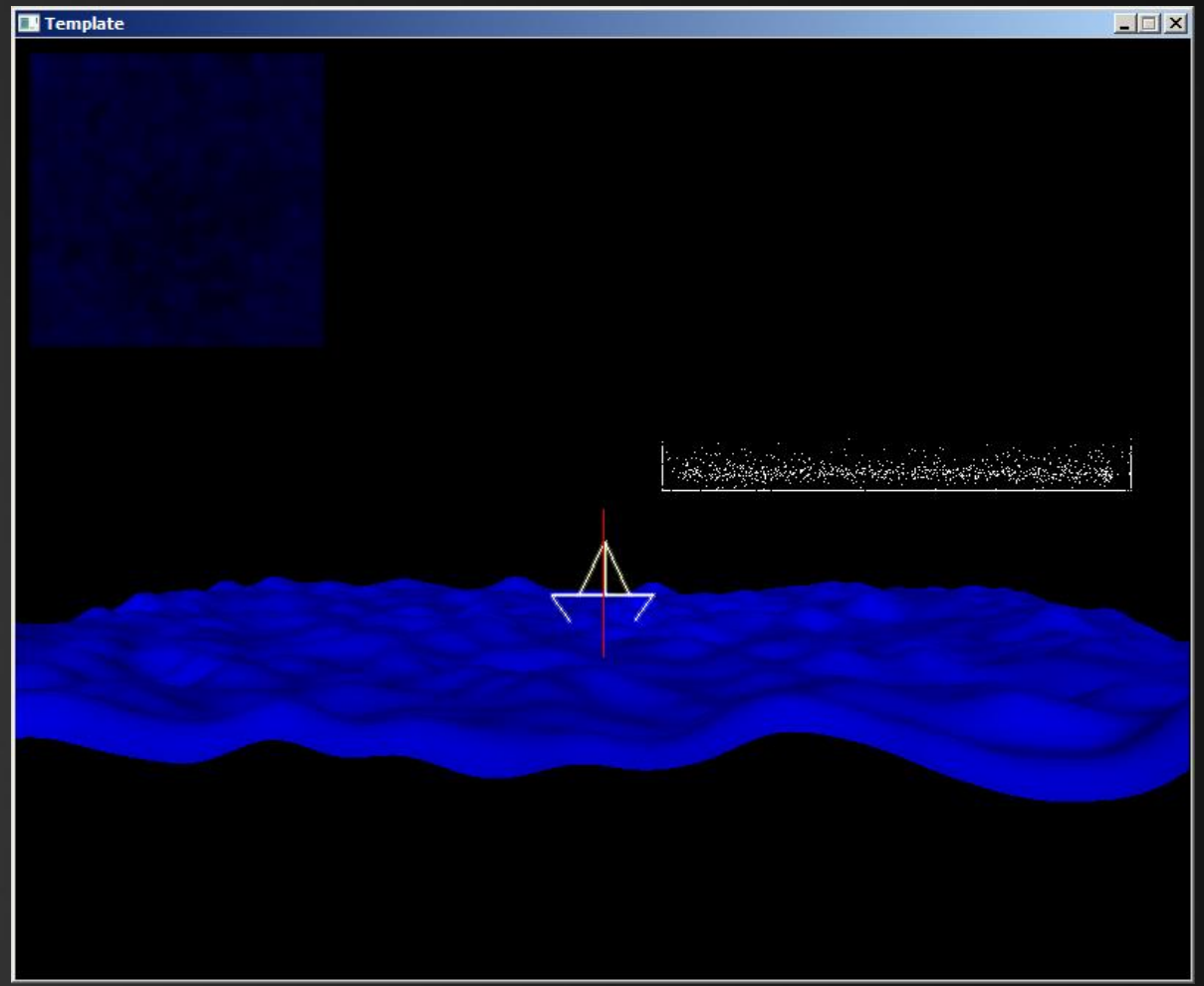
Never Assume

Take-away:

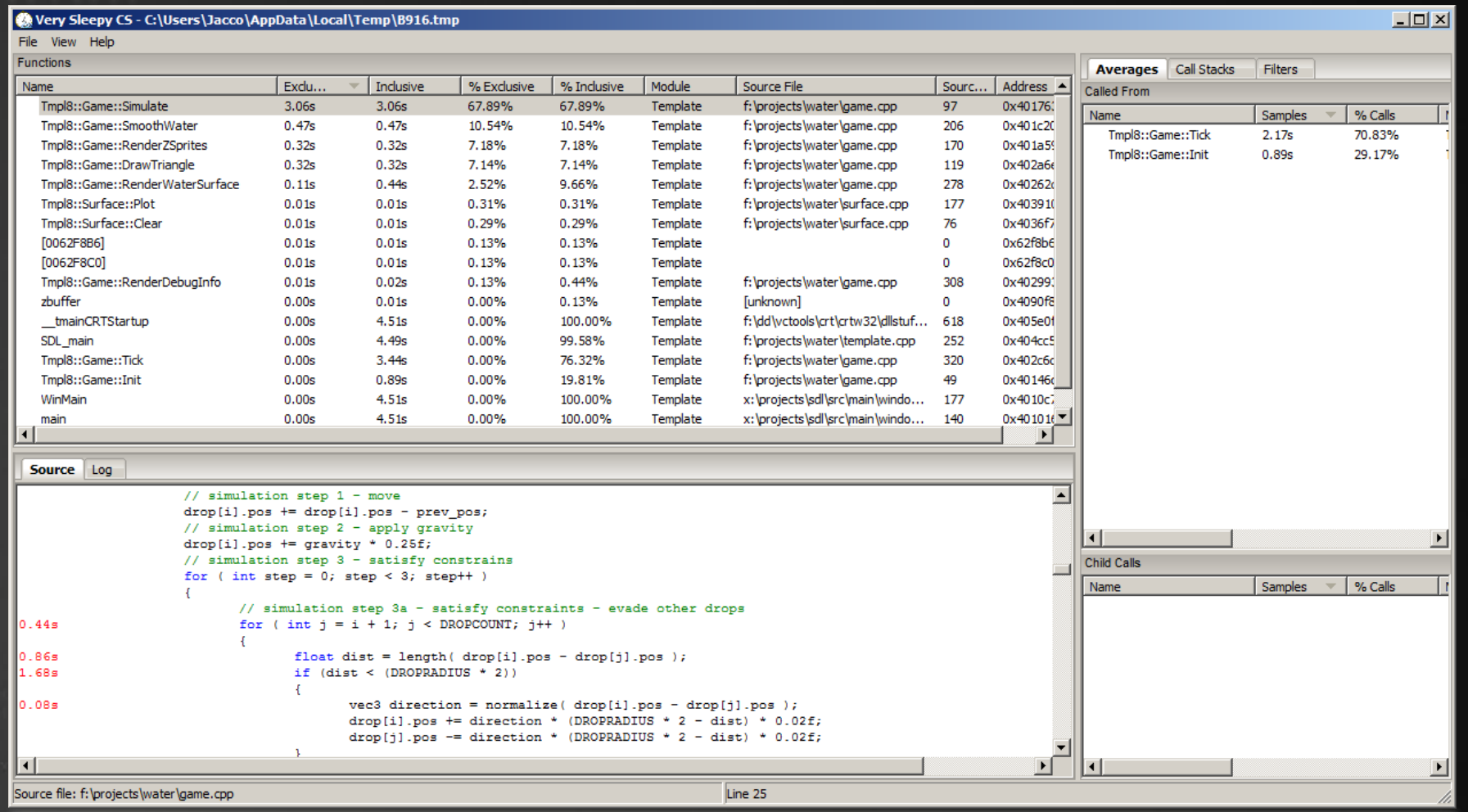
Never assume. Profiling *always* steers optimization.

Optimize in release mode. Enable debug info during this process. Don’t forget to turn it off before distribution.

Profiler Output



Profiler Output



Profiler Output

Profiling – Results

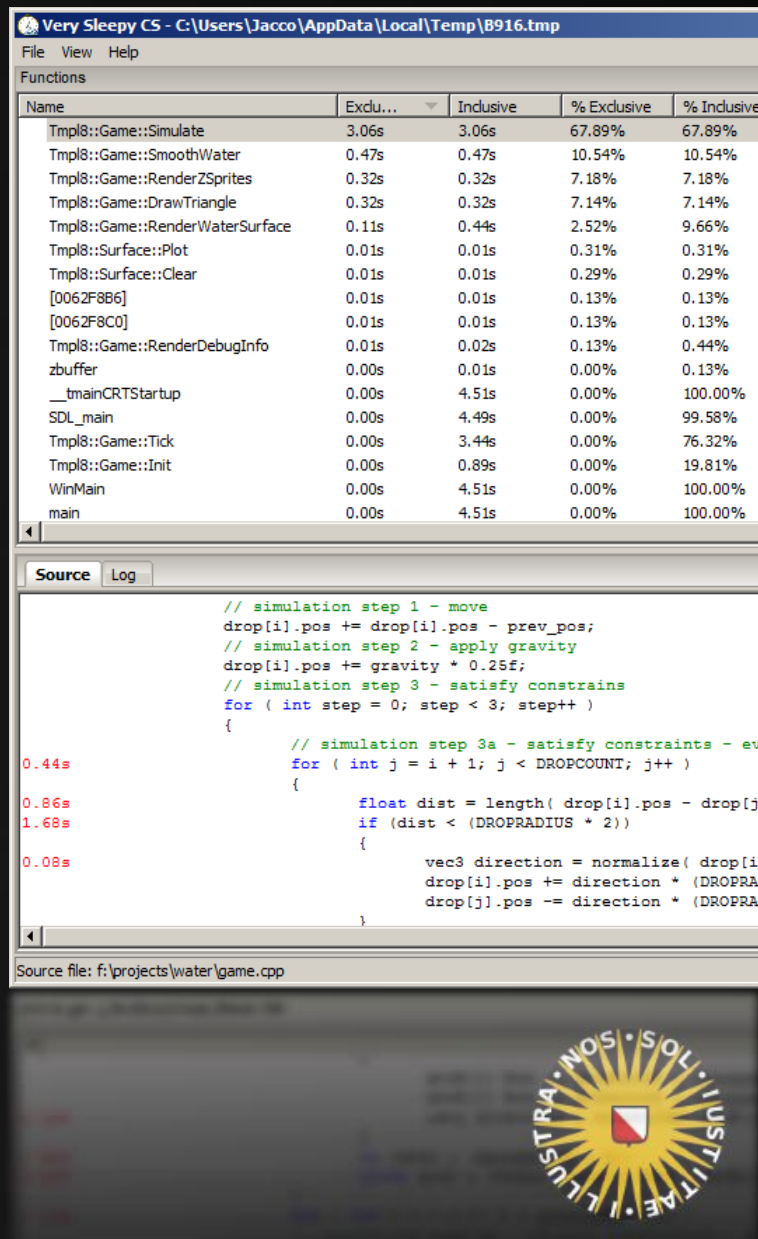
Game::Simulate	67.89%	67.89%
Game::SmoothWater	10.54%	10.54%
Game::RenderZSprites	7.18%	7.18%
Game::Tick	0.00%	76.32%

Running ~3 seconds, we spent 0.86s on this line:

```
float dist = length( drop[i].pos - drop[j].pos );
```

and 1.68s on this line:

```
if (dist < (DROPRADIUS * 2))
```



Profiler Output

Profiling – finding hotspots

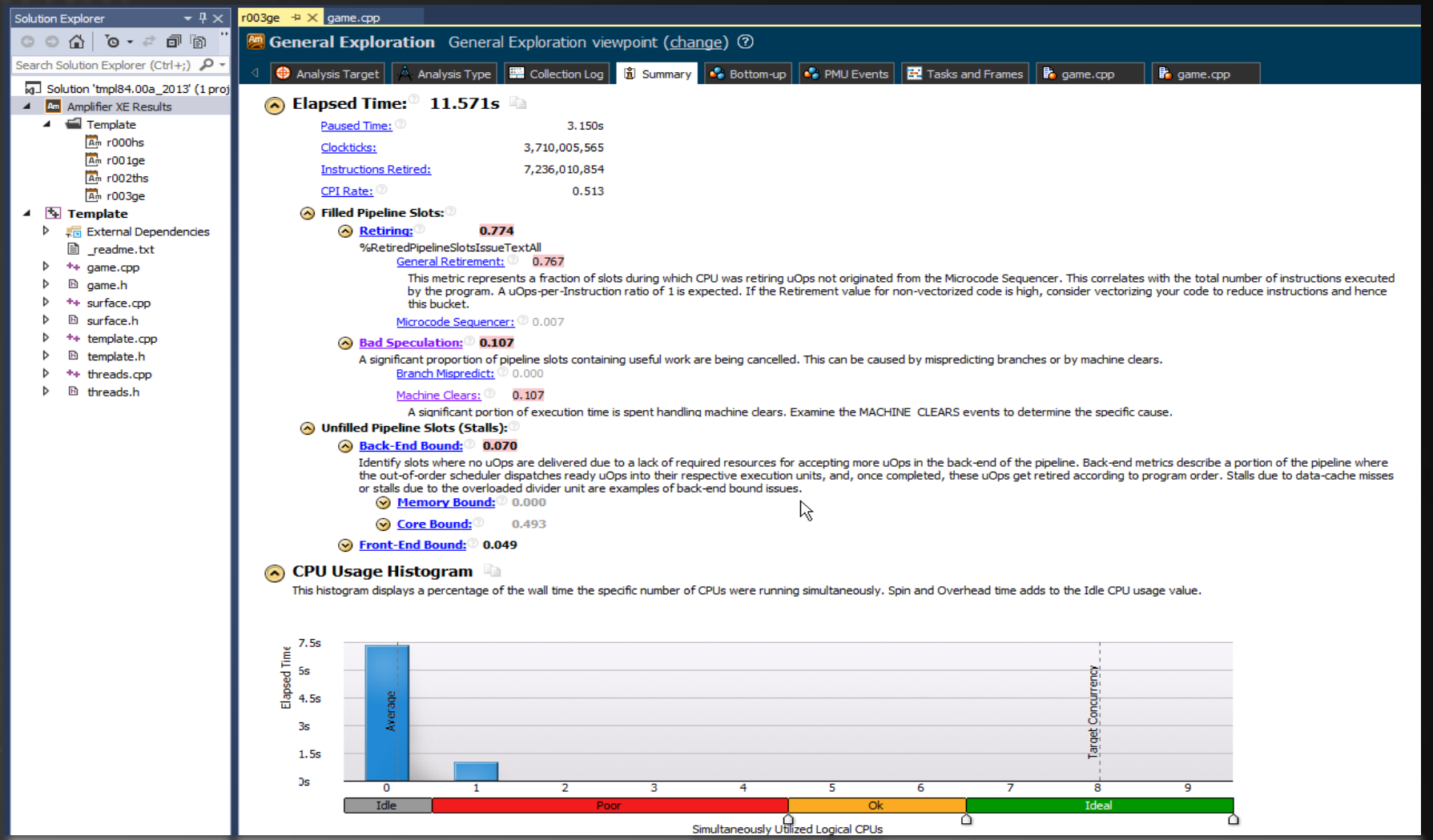
The profiler allows you to quickly find the parts of your program that take most time.

But:

- Mind debug versus release;
- The profiler doesn’t tell you *why* a function is costly
- The profiler doesn’t report scalability
- There is no ‘cost over time’ information

➔ Scalability analysis requires running the program with different work sets (i.e., change N in O(N)).
➔ Determining why a section takes a lot of time requires more in-depth knowledge.
➔ *Solving* the performance issue requires even more in-depth knowledge.

Profiler Output



Profiler Output

So. Lin.	Source	Clockticks	Instructions Retired	CPI Rate	Filled Pipeline Slots		Unfilled Pipeline Slo...	
					Retiring	Bad Spec...	Back-End Bound	Fr. Bo.
129								
130	`// Game::Simulate: do Verlet physics simulation on the particles`							
131	`void Game::Simulate()`							
132	`{`							
133	`ENTER(SIMULATE);`							
134	`for ( int i = 0; i < DROPCOUNT; i++ )`							
135	`{`							
136	`gravity = vec3( sin( angle * PI / 180 ), cos( angle * PI / 180 ), 0 );`							
137	`vec3 prev_pos = drop[i].prev_pos;`							
138	`drop[i].prev_pos = drop[i].pos;`							
139	`// simulation step 1 - move`							
140	`drop[i].pos += drop[i].pos - prev_pos;`							
141	`// simulation step 2 - apply gravity`							
142	`drop[i].pos += gravity * 0.25f;`							
143	`// simulation step 3 - satisfy constrains`							
144	`for ( int step = 0; step < 3; step++ )`	0	2,000,003	0.000	0.000	0.000	1.000	0.000
145	`{`							
146	`// simulation step 3a - satisfy constraints - evade other drops`							
147	`for ( int j = i + 1; j < DROPCOUNT; j++ )`	128,000,192	148,000,222	0.865	0.703	0.000	0.824	0.059
148	`{`							
149	`float dist = length( drop[i].pos - drop[j].pos );`	780,001,170	208,000,312	3.750	1.000	0.000	0.231	0.010
150	`if (dist < (DROPRADIUS * 2))`	1,042,001,563	2,818,004,227	0.370	0.115	0.446	0.381	0.058
151	`{`							
152	`vec3 direction = normalize( drop[i].pos - drop[j].pos );`	64,000,096	86,000,129	0.744	0.820	0.000	0.180	0.000
153	`drop[i].pos += direction * (DROPRADIUS * 2 - dist) * 0.02f;`	22,000,033	0		0.000	1.000	0.000	1.000
154	`drop[j].pos -= direction * (DROPRADIUS * 2 - dist) * 0.02f;`	6,000,009	2,000,003	3.000	0.000	0.000	1.000	0.000
155	`}`							
156	`}`							
157	`// simulation step 3b - satisfy constraints - evade walls`							
158	`if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.000`							
159	`if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.0`							
160	`if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.000`							
161	`if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.0`							
162	`if (drop[i].pos.z > 20) drop[i].pos.z = 19.99f - drop[i].pos.z * 0.000`							
163	`}`							
164	`}`							
165	`LEAVE(SIMULATE);`							
166	`}`							

Profiler Output

Take-away:

Free, vendor-agnostic profilers tell you where time is spent in your program (but not *why*).

Vendor-specific tools provide a wealth of information, but generally require knowledge about the hardware processes.

Stalls are generally not vendor-specific and will be similar on similar hardware.

Just timing information is often sufficient to make an educated guess towards improvements.

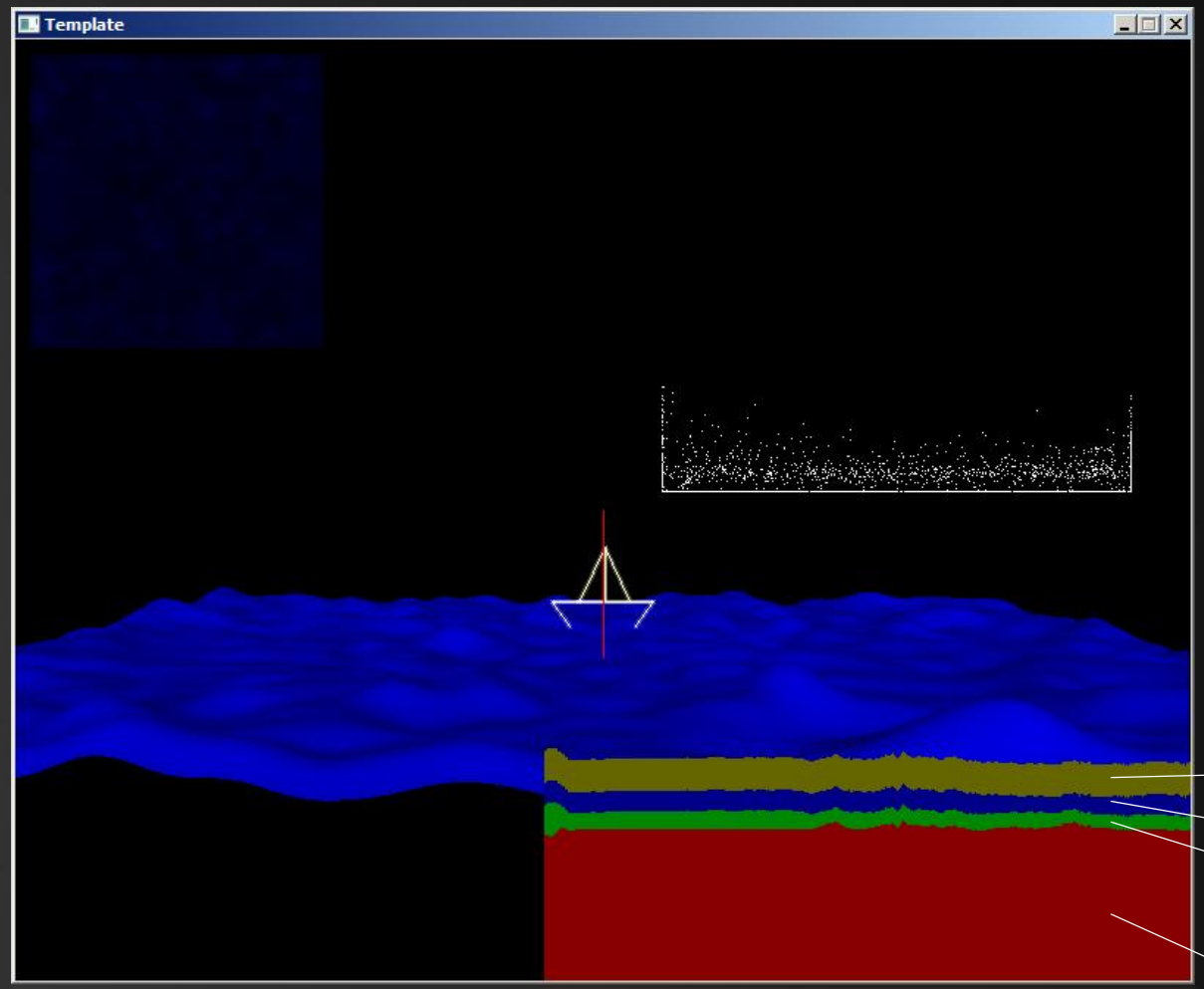
Custom Profiling

Generic Profiler Downsides

- No ‘performance over time’ measurements
- Requires inclusion of debug information (including source code)
- Not real-time
- Not very intuitive

Using a custom in-app profiler we can drastically improve our profiling information.

Custom Profiling



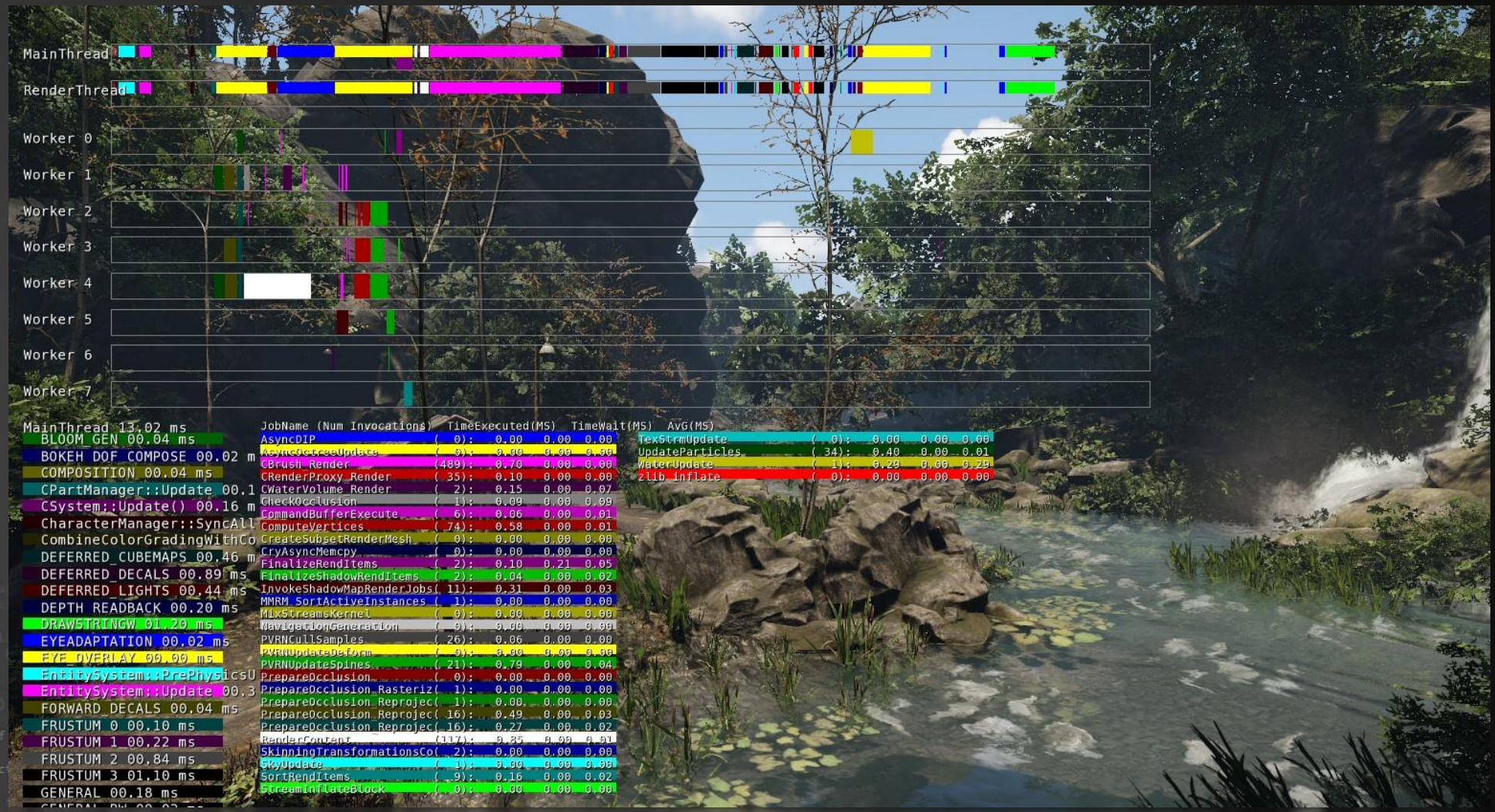
Custom Profiling

Minecraft

Custom Profiling

UnrealEngine 3

Custom Profiling



CryEngine

Custom Profiling

StarCraft II

Custom Profiling

StarCraft II

Custom Profiling

Take-away:

In-app profiling provides advantages over external profilers:

- You get real-time information, which is easily associated with what is going on in the app;
- You can measure statistics that are not available to the profiler;
- You can present the data in a form that is also useful to people not familiar with the intricacies of the profiler.

Considerations

Custom timers: what to measure?

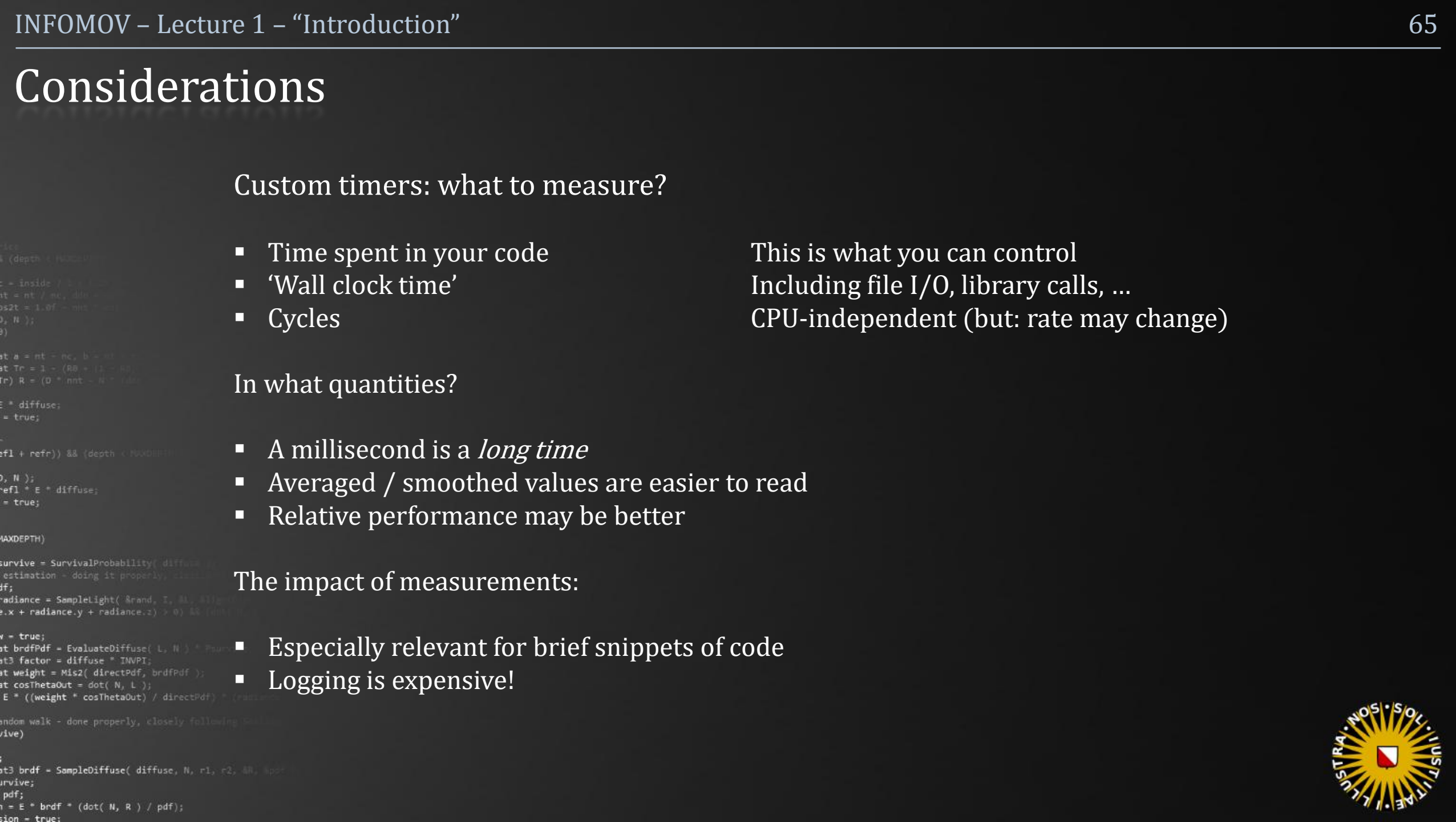
- Time spent in your code — This is what you can control
- ‘Wall clock time’ — Including file I/O, library calls, …
- Cycles — CPU-independent (but: rate may change)

In what quantities?

- A millisecond is a *long time*
- Averaged / smoothed values are easier to read
- Relative performance may be better

The impact of measurements:

- Especially relevant for brief snippets of code
- Logging is expensive!



Considerations

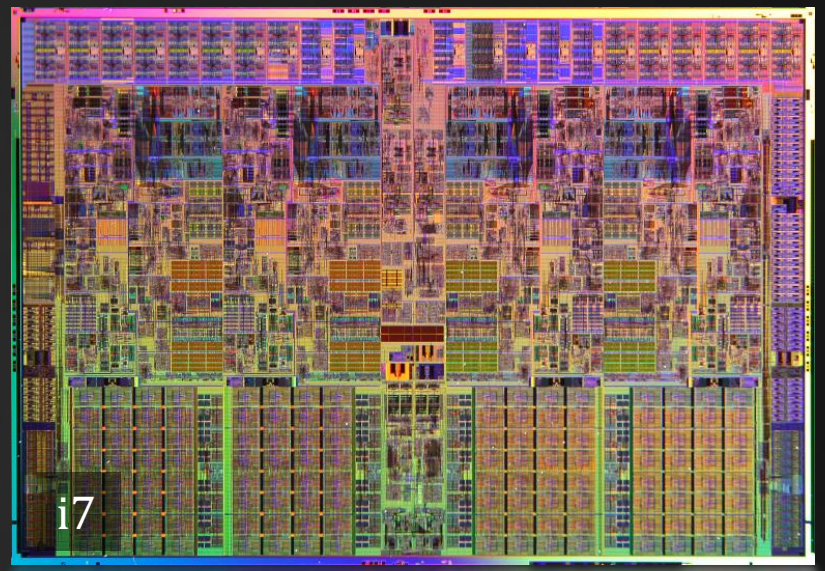
What if the CPU isn’t the problem?

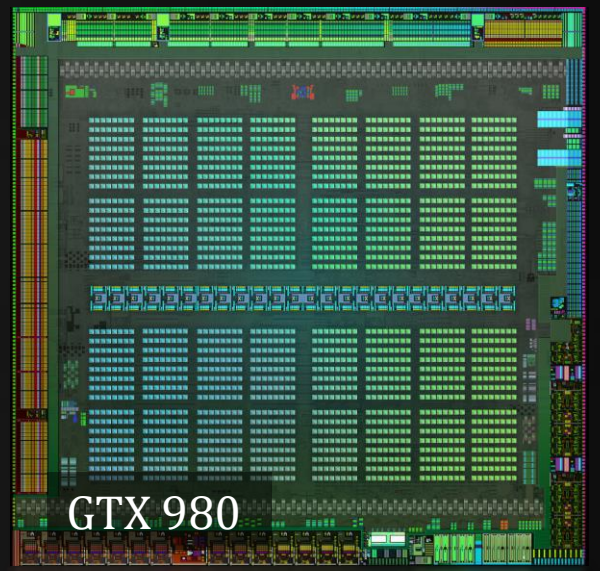
A modern system consists of:

- A multicore CPU;
- A many-core GPU.

If one of them is idling, your application isn’t running at maximum performance.

Move work, even if the other processor can’t do it as efficient.





Considerations

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 steps 6 and 7 until time runs out
9. Report.

And Finally:

Profiling:

Without it, no optimization – we need to *know*

How to profile: tools, custom timers, CPU + GPU

What to profile: realistically (release!), raw performance, scalability
(but also: cache misses, pipelining, branch prediction)

Keep in mind: profiling takes time too.

Repeated profiling: things change, if you’re doing it right. Stay informed.

/INFOMOV/

END of “Introduction”

next lecture: “Low Level”