2019/2020, 1st quarter

INFOMOV: Optimization & Vectorization / *OptmzdSummary*

#0 - Profiling

Author: Jacco Bikker

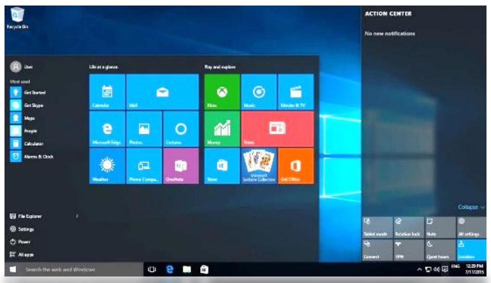
TL;DR

These documents contain an overview of many of the topics discussed in the lectures. Often, additional links to relevant materials are provided. These documents will ultimately provide a full overview of the theory of the course. For the 2019/2020 edition, this is not yet expected to be the case.

In this '#0' a global overview of the optimization process is provided. We also briefly discuss the topic of profiling.

Raison d'être /ˌreɪzɔ̃ ˈdɛtrə/

There are many reasons to optimize software. The *goal* of the optimization affects the optimization process, and, in particular, the 'termination criterion', i.e. when to stop. Some software simply must run as fast as possible, in which case we start with low hanging fruit (as indicated by a profiling session) and proceed with profiling / optimization cycles until time runs out. In other cases, software should run on a CPU that is as simple as possible, e.g., to save on hardware or energy cost. Perhaps we want to optimize for memory usage, or application response time. Software that is already fast enough also exist. And in some cases, optimization effort is more expensive than new hardware. And finally: sometimes we should refrain from optimization to minimize the risk of breaking something that works.



If faster hardware is not an option, we need to make better use of the resources we have. Often this is a matter of improving algorithms, which is the focus of many research areas. In this course however, we focus on the 'C' in '$O(N)$'; on how a clever algorithm uses the hardware. Besides the obvious (multithreading and GPGPU) we dive into the not-so-obvious (vectorization, data locality, low level hardware details) to obtain a substantial and often predictable speedup.

Consistent Approach

Software optimization happens when functionality is complete. This often means that optimization is squeezed in just before the deadline, probably with a tight budget. An important question is: *what can we do in X days?* Answering that question, and delivering what we promise, requires a consistent and reproduceable process, as well as some intuition. E.g.:

- Single-threaded software can run up to five times faster on a quad-core CPU with hyperthreading.
- Code that is clearly suitable for data parallelism can run up to eight times faster on AVX hardware.
- Most software makes poor use of the memory system. Improving this can double application performance.

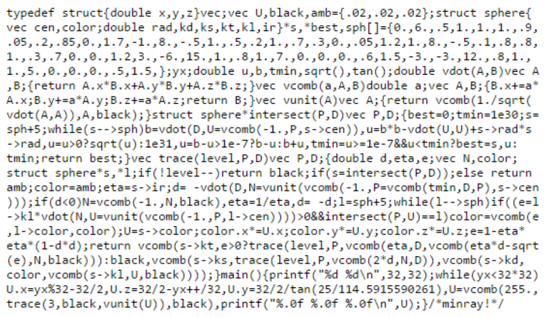
For many programs, it is safe to estimate that a 25x speedup can be achieved in a modest amount of time. If the GPU can be successfully deployed, this can be even more.

Do note that optimization tends to reduce maintainability of the software. Also note that optimization may break functionality.

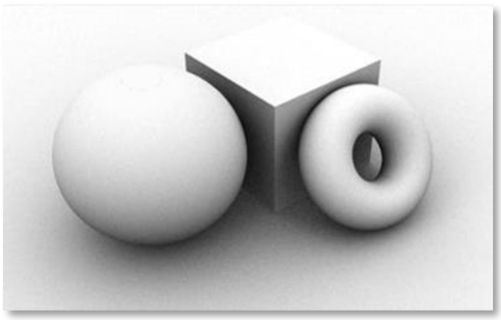
I mentioned a consistent and reproduceable process. This is roughly:

0. Determine optimization requirements.
1. **Profile** the program to determine hotspots.
2. Determine scalability of the hotspots.
3. Improve scalability, if possible.
4. **Profile** the improved code.
5. Parallelize / vectorize the code and/or use GPGPU.
6. **Profile** the improved code.
7. Apply low-level optimizations to hotspots.
8. Repeat step 6 and 7 until time runs out.
9. Report.

A repeating term here is 'profiling'. Without profiling, the effort lacks a clear course and becomes essentially random. It is very likely that you spend your efforts on the wrong code lines. You may break things and slow down sections of the program.



```
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!*/
```



Optimized code can be hard to read.

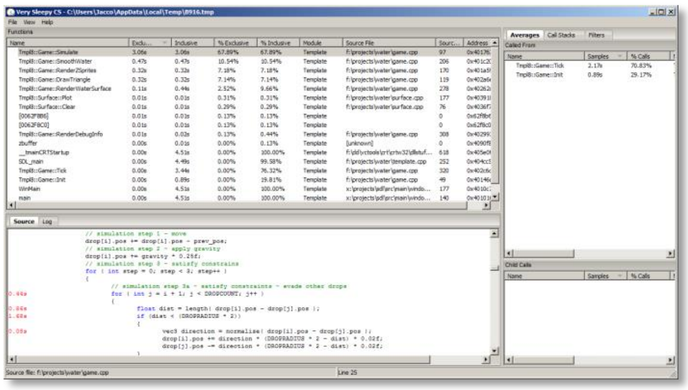
Profiling

Profiling can be as simple as starting a stopwatch when our code starts, and stopping it when the program completes. This can be refined by using a stopwatch per function, or even per line of code.

Luckily, we don't need a timer around each line of code for this. A *profiler* is a program that interrupts running software at fixed intervals. During such an interruption, it probes the *program counter* (PC) of the CPU, which stores the address of the instruction that is about to be executed by the CPU. By storing the PC many times, we get a histogram of where the CPU spends its time.

A popular profiler for Windows is Very-Sleepy. The profiler of Visual Studio is a good alternative.

Be aware that profiling influences your measurements: interrupting the running program and storing profiling information takes time and changes the contents of caches. Also be aware that modern CPUs and GPUs do not run at fixed clock rates: a hot CPU may slow down. This is especially true on laptops.



Name	Exclu...	Inclusive	% Exclusive	% Inclusive	Module	Source File	Sourc...	Address
Tmpl8::Game::Simulate	3.06s	3.06s	67.89%	67.89%	Template	f:\projects\water\game.cpp	97	0x40176
Tmpl8::Game::SmoothWater	0.47s	0.47s	10.54%	10.54%	Template	f:\projects\water\game.cpp	206	0x401c2
Tmpl8::Game::Render2Sprites	0.32s	0.32s	7.18%	7.18%	Template	f:\projects\water\game.cpp	170	0x401a5
Tmpl8::Game::DrawTriangle	0.32s	0.32s	7.14%	7.14%	Template	f:\projects\water\game.cpp	119	0x402a6
Tmpl8::Game::RenderWaterSurface	0.11s	0.44s	2.52%	9.66%	Template	f:\projects\water\game.cpp	278	0x40262
Tmpl8::Surface::Plot	0.01s	0.01s	0.31%	0.31%	Template	f:\projects\water\surface.cpp	177	0x40391
Tmpl8::Surface::Clear	0.01s	0.01s	0.29%	0.29%	Template	f:\projects\water\surface.cpp	76	0x4036f
[0062F8B6]	0.01s	0.01s	0.13%	0.13%	Template		0	0x62f8b
[0062F8C0]	0.01s	0.01s	0.13%	0.13%	Template		0	0x62f8c
Tmpl8::Game::RenderDebugInfo	0.01s	0.02s	0.13%	0.44%	Template	f:\projects\water\game.cpp	308	0x40299
zbuffer	0.00s	0.01s	0.00%	0.13%	Template	[unknown]	0	0x4090f
__tmainCRTStartup	0.00s	4.51s	0.00%	100.00%	Template	f:\dd\vctools\crt\crtw32\dllstuf...	618	0x405e0
SDL_main	0.00s	4.49s	0.00%	99.58%	Template	f:\projects\water\template.cpp	252	0x404cc
Tmpl8::Game::Tick	0.00s	3.44s	0.00%	76.32%	Template	f:\projects\water\game.cpp	320	0x402c6
Tmpl8::Game::Init	0.00s	0.89s	0.00%	19.81%	Template	f:\projects\water\game.cpp	49	0x40146
WinMain	0.00s	4.51s	0.00%	100.00%	Template	x:\projects\sdl\src\main\windo...	177	0x4010c
main	0.00s	4.51s	0.00%	100.00%	Template	x:\projects\sdl\src\main\windo...	140	0x40101

Name	Samples	% Calls
Tmpl8::Game::Tick	2.17s	70.83%
Tmpl8::Game::Init	0.89s	29.17%

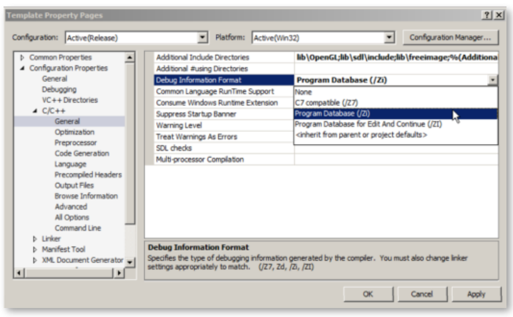
```
                  // simulation step 1 - move
                  drop[i].pos += drop[i].pos - prev_pos;
                  // simulation step 2 - apply gravity
                  drop[i].pos += gravity * 0.25f;
                  // simulation step 3 - satisfy constrains
                  for ( int step = 0; step < 3; step++ )
                  {
                      // simulation step 3a - satisfy constraints - evade other drops
0.44s                 for ( int j = i + 1; j < DROPCOUNT; j++ )
                      {
0.86s                     float dist = length( drop[i].pos - drop[j].pos );
1.68s                     if (dist < (DROPRADIUS * 2))
                          {
0.08s                         vec3 direction = normalize( drop[i].pos - drop[j].pos );
                              drop[i].pos += direction * (DROPRADIUS * 2 - dist) * 0.02f;
                              drop[j].pos -= direction * (DROPRADIUS * 2 - dist) * 0.02f;
                          }
```

Source file: f:\projects\water\game.cpp   Line 25

VerySleepy

PRACTICAL NOTES – In Microsoft Visual Studio you typically chose between a 'Debug' and 'Release' configuration. Note that these are vastly different. You will want to profile the 'Release' version, but this poses a problem: this configuration by default lacks debug information, which is needed for linking instruction addresses to source lines.

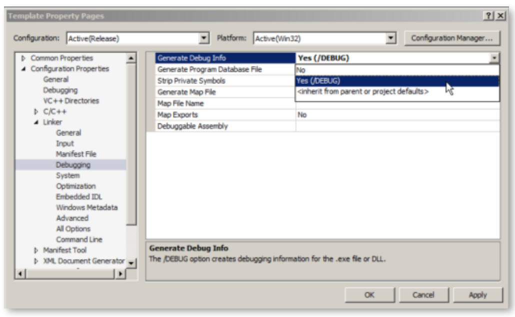
Luckily, you can enable this information in a Release build as well. Right-click on the project in the solution explorer and select 'Properties'. Then, for the Release configuration, enable 'Debug Information Format' (C/C++ - General) and tell the linker to generate debug info (Linker – Debugging).



Configuration: Active(Release)   Platform: Active(Win32)

Additional Include Directories	lib\OpenGL;lib\sdl\include;lib\freeimage;%(Additiona
Additional #using Directories	
Debug Information Format	Program Database (/Zi)
Common Language RunTime Support	None
Consume Windows Runtime Extension	C7 compatible (/Z7)
Suppress Startup Banner	Program Database (/Zi)
Warning Level	Program Database for Edit And Continue (/ZI)
Treat Warnings As Errors	<inherit from parent or project defaults>
SDL checks	
Multi-processor Compilation	

Debug Information Format
Specifies the type of debugging information generated by the compiler. You must also change linker settings appropriately to match. (/Z7, Zd, /Zi, /ZI)



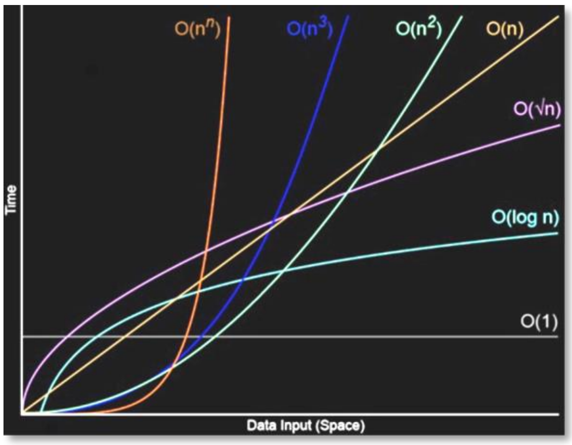
Configuration: Active(Release)   Platform: Active(Win32)

Generate Debug Info	Yes (/DEBUG)
Generate Program Database File	No
Strip Private Symbols	Yes (/DEBUG)
Generate Map File	<inherit from parent or project defaults>
Map File Name	
Map Exports	No
Debuggable Assembly	

Generate Debug Info
The /DEBUG option creates debugging information for the .exe file or DLL.

Scalability

A typical goal of an optimization effort is to make the program fast enough for a larger data set. If this is indeed the case, we may want to measure more than just the performance of the original code. By comparing the performance for different workloads, we get an impression of the *scalability* of the code.



A function in the program that takes little time for 100 input elements may become a bottleneck at 1000 elements. We need to make sure to optimize the software for the *desired* workload, which may not be the current workload.

Temporally Varying

For real-time applications such as games it is important that the performance of individual frames does not diverge too much from the average performance. To get real-time data on application performance it is sometimes beneficial to add a custom profiler to an application. Many game engines provide this.

Profiling data in CryEngine.

The End

Up next in #1: Low Level Optimization, in which we try to measure the cost of a single line of code.

This material is part of the Optimization & Vectorization course of the MGT master program at the Utrecht University in the Netherlands. More information at:

http://www.cs.uu.nl/docs/vakken/mov.

Utrecht University