



UPPSALA  
UNIVERSITET

# Imperativ och objektorienterad programmeringsmetodik

Föreläsning 9, HT 2020

Fredrik Nysjö

Automatisering & optimering

## Vem är jag?

- 2005-2010: MSc i datavetenskap, UU
- 2010-2015: Forskningsingenjör, CBA/UU
- 2015-2020: Doktorand i digital bildanalys, UU
- Just nu: Lärare (adjunkt) och forskningsingenjör
- Forskningsintressen: Medicinsk bildanalys, maskininlärning, datorgrafik & visualisering, haptik





UPPSALA  
UNIVERSITET

# Automatisering



UPPSALA  
UNIVERSITET

# Skalet (shell)

# Skalskript för att bygga program

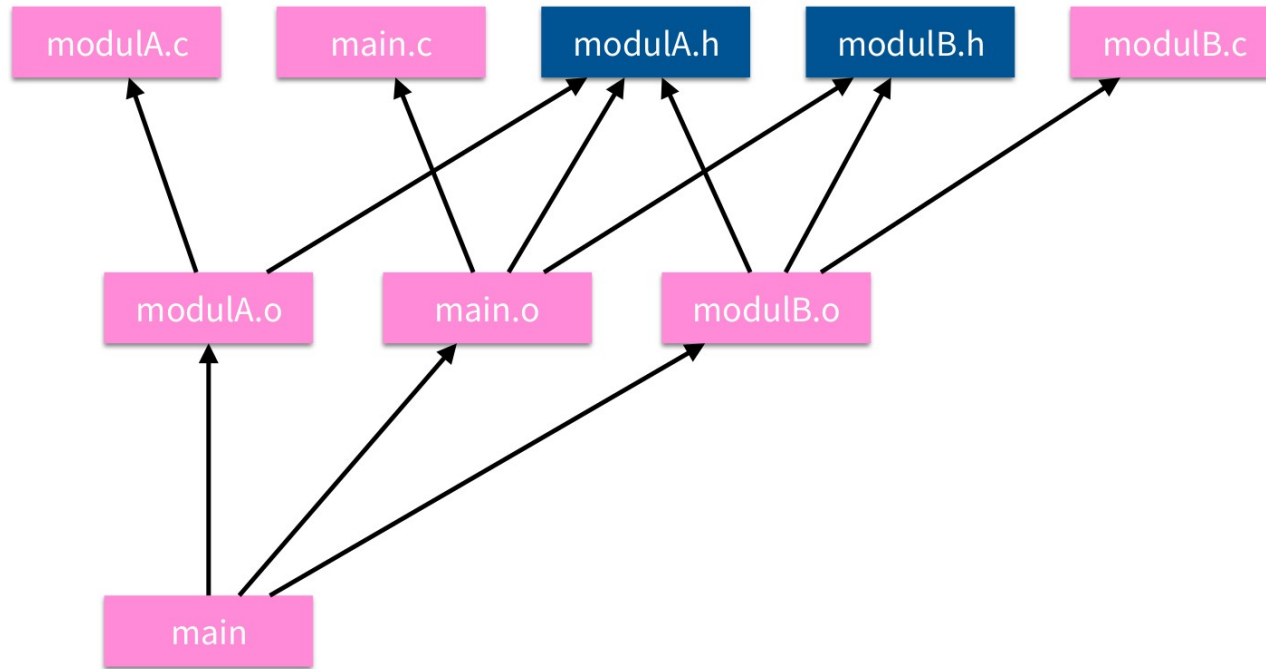
- Körs som vilket program som helst
- Glöm inte att göra `chmod +x` så att filen går att köra

Vilket program (shell) skall tolka filen?

```
#!/bin/bash  
emacs stack.c  
gcc -g -o stack_test stack.c stack_test.c -lcunit  
gdb stack_test
```

work

# Hur håller vi reda sådana här beroenden?





UPPSALA  
UNIVERSITET

# Byggverktyg (Make)

- Se kort introduktion på <http://wrigstad.com/ioopm/makefiles.html>

# Makefiler

mål

beroenden (ofta målfiler i andra mål)

```
main: main.c sub.c sub.h  
gcc -g main.c sub.c -o main
```

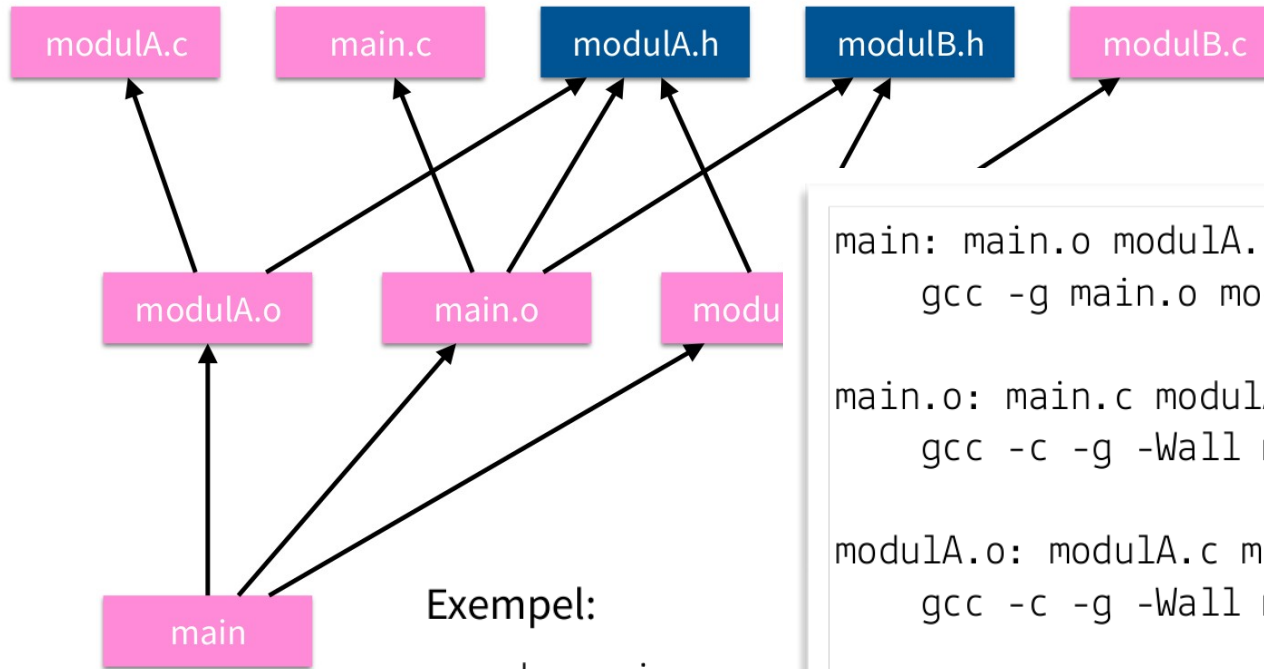
OBS! TAB-tecken här.

***Inte*** blanktecken!

kommando som utförs om målfilen inte finns eller är äldre än någon av de filer den är beroende av



# Makefiler



Exempel:

```
make main.o  
make main  
make
```

```
main: modulA.o modulB.o  
    gcc -g main.o modulA.o modulB.o -o main  
  
main.o: main.c modulA.h modulB.h  
    gcc -c -g -Wall main.c  
  
modulA.o: modulA.c modulA.h  
    gcc -c -g -Wall modulA.c  
  
modulB.o: modulB.c modulA.h modulB.h  
    gcc -c -g -Wall modulB.c
```

Makefile

# Makefiler (variabler och wildcards)

Det matchande namnet

Alla beroenden som var nyare än målet

Matchar filnamn

Exempel:

Namn på målet

```

main: myprog

C_COMPILER = gcc
C_OPTIONS  = -Wall -pedantic -g
C_LINK_OPTIONS = -lm
CUNIT_LINK = -lcuni

%.o: %.c
    $(C_COMPILER) $(C_OPTIONS) $* -o $@

myprog: file1.o file2.c
    $(C_COMPILER) $(C_LINK_OPTIONS) $? -o $@

test1: mytests1.o file1.o
    $(C_COMPILER) $(C_LINK_OPTIONS) $(CUNIT_LINK) $? -o $@

clean:
    rm -f *.o myprog
  
```

## Alternativ till make (CMake)

- När vanliga makefiler inte riktigt räcker till:
  - Stora kodbaser med många filer och externa beroenden/bibliotek som också behöver byggas eller länkas in
  - Vi vill kunna bygga koden på flera plattformar (Windows, Linux, etc.)
- CMake är ett verktyg som genererar makefiler (eller byggfiler i andra format) och hjälper till att hålla reda på beroenden

# CMake – Litet exempel

```
cmake_minimum_required(VERSION 3.0.0)

# Specify project name
project(my_program)

# Specify build type
set(CMAKE_BUILD_TYPE Debug)

# Add source directories
aux_source_directory("${CMAKE_CURRENT_SOURCE_DIR}/src" PROJECT_SRCS)

# Add include directories
include_directories("${CMAKE_CURRENT_SOURCE_DIR}/src")

# Create build files for executable
add_executable(${PROJECT_NAME} ${PROJECT_SRCS})

# Link against libraries (-lm and -lcunit)
target_link_libraries(${PROJECT_NAME} m cunit)

# Install executable
install(TARGETS ${PROJECT_NAME} DESTINATION bin)
```

CMakeLists.txt



UPPSALA  
UNIVERSITET

# CMake – Större exempel (från kursen 1TD389)



UPPSALA  
UNIVERSITET

# Optimering


# Hur mäta prestanda?


- Några sätt:
  - Verktuget `time` (mäter körningstid för hela programmet)
  - Manuellt lägga in timers i koden
  - Profileringsverktyg

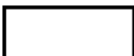
# Körningstid?



real (wall clock) time

 = **user time** (*time executing instructions in the user process*)

 = **system time** (*time executing instructions in kernel on behalf of user process*)

 = **some other user's time** (*time executing instructions in different user's process*)

 +  +  = **real (wall clock) time**

*We will use the word "time" to refer to user time.*

 **cumulative user time**



## Profileringsverktyg (gprof)

- Vi kan profilera ett program med verktyget `gprof`:

```
$ gcc -pg myprog.c -o myprog
```

Skapas vid körningen av `myprog`

```
$ ./myprog
```

```
$ gprof gmon.out myprog > profile.txt
```

```
$ more profile.txt
```

- Filen `profile.txt` kommer innehålla en profil som beskriver tiden spenderad i olika delar av programmet

# Profilering i gprof – Exempel

```
bool intersect_sphere(Ray ray, Sphere sphere, float *tmin, float *tmax, Vec3 *normal)
{
  11 lines: const Vec3 oc = sub(ray.origin, sphere.center);-----
}

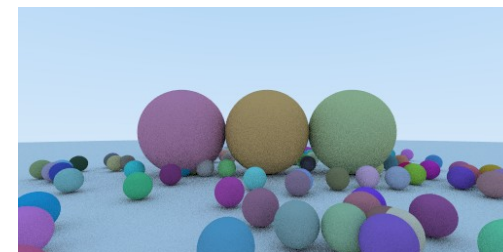
Hit hit_closest(Ray ray, const Scene *scene)
{
  10 lines: assert(scene);-----
}

Vec3 trace_scene(Ray ray_eye, const Scene *scene)
{
  assert(scene);

  Hit hits[RAY_DEPTH_MAX];
  uint32_t hit_count = 0;

  Ray ray = ray_eye;
  for (uint32_t i = 0; i < RAY_DEPTH_MAX; ++i) {
    Hit hit = hit_closest(ray, scene);
    if (hit.t >= RAY_TMIN_MAX) break;

    hits[hit_count++] = hit;
    ray.origin = add(ray.origin, smul(ray.dir, hit.t));
    ray.dir = add(normalize(hit.normal), rnd_ball());
    ray.origin = add(ray.origin, smul(ray.dir, RAY_EPSILON));
  }
}
```



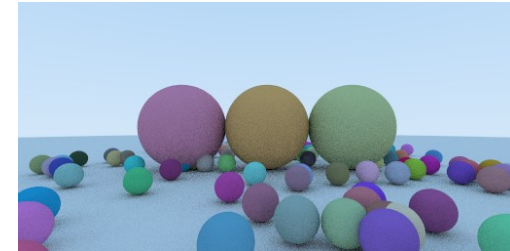
Del av programmet `rtow_main.c` som beräknar bilden till höger

# Profiling i gprof (gcc -pg)

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.22% of 4.54 seconds

index	% time	self	children	called	name
[1]	100.0	0.04	4.50		<spontaneous>
		0.04	4.46	524288/524288	main [1]
		0.00	0.00	1/1	trace_scene [2]
		0.00	0.00	1/1	create_image2d [17]
		0.00	0.00	1/1	setup_scene [19]
		0.00	0.00	1/1	write_to_ppm [20]
		0.00	0.00	1/1	destroy_image2d [18]
-----					
[2]	99.1	0.04	4.46	524288/524288	main [1]
		0.04	4.46	524288	trace_scene [2]
		0.50	3.64	914210/914210	hit_closest [3]
		0.06	0.09	439528/439528	lookup_material [7]
		0.02	0.04	474682/474682	sky [10]
		0.05	0.00	1318584/2461125	add [9]
		0.02	0.01	439528/439528	rnd_ball [13]
		0.02	0.00	1318584/4289545	smul [11]
		0.01	0.01	439528/1353738	normalize [12]
		0.00	0.00	439528/439528	mul [15]
-----					
[3]	91.2	0.50	3.64	914210/914210	trace_scene [2]
		0.50	3.64	914210	hit_closest [3]
		1.29	2.36	117018880/117018880	intersect_sphere [4]
-----					
[4]	80.2	1.29	2.36	117018880/117018880	hit_closest [3]
		1.29	2.36	117018880	intersect_sphere [4]
		1.24	0.00	117686739/117686739	sub [5]
		1.09	0.00	351056640/351056640	dot [6]
		0.02	0.00	667859/2461125	add [9]
		0.01	0.00	667859/4289545	smul [11]

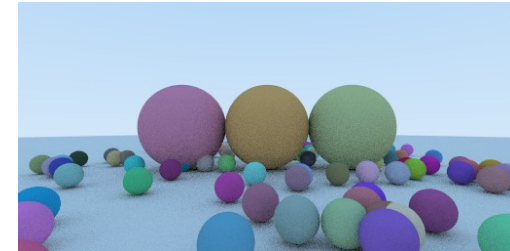


# Profiling i gprof (gcc -pg)

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.22% of 4.54 seconds

index	% time	self	children	called	name
[1]	100.0	0.04	4.50		<spontaneous>
		0.04	4.46	524288/524288	main [1]
		0.00	0.00	1/1	trace_scene [2]
		0.00	0.00	1/1	create_image2d [17]
		0.00	0.00	1/1	setup_scene [19]
		0.00	0.00	1/1	write_to_ppm [20]
		0.00	0.00	1/1	destroy_image2d [18]
-----					
[2]	99.1	0.04	4.46	524288/524288	main [1]
		0.04	4.46	524288	trace_scene [2]
		0.50	3.64	914210/914210	hit_closest [3]
		0.06	0.09	439528/439528	lookup_material [7]
		0.02	0.04	474682/474682	sky [10]
		0.05	0.00	1318584/2461125	add [9]
		0.02	0.01	439528/439528	rnd_ball [13]
		0.02	0.00	1318584/4289545	smul [11]
		0.01	0.01	439528/1353738	normalize [12]
		0.00	0.00	439528/439528	mul [15]
-----					
[3]	91.2	0.50	3.64	914210/914210	trace_scene [2]
		0.50	3.64	914210	hit_closest [3]
		1.29	2.36	117018880/117018880	intersect_sphere [4]
-----					
[4]	80.2	1.29	2.36	117018880/117018880	hit_closest [3]
		1.29	2.36	117018880	intersect_sphere [4]
		1.24	0.00	117686739/117686739	sub [5]
		1.09	0.00	351056640/351056640	dot [6]
		0.02	0.00	667859/2461125	add [9]
		0.01	0.00	667859/4289545	smul [11]



# Optimering

- Skriv om kod så att den gör samma sak som tidigare, fast snabbare :)
- Lågnivå-språk som C låter oss skriva “maskinnära” kod och kontrollera när minne allokeras, vilket möjliggör många slags optimeringar
- Andra egenskaper hos ett program vi också kan optimera för:
  - Minnesanvändning
  - Kodstorlek (programstorlek)

## När bör man optimera sin kod?

- I regel så sent i utvecklingen som möjligt, när du vet vilka delar i koden som utgör “hotspots” och verkligen behöver optimeras\*
- Merparten av koden i ett program har ingen inverkan på prestandan!
- Läsbarhet och algoritmiska optimeringar är oftast att föredra
- Mät alltid innan du börjar optimera!

\*Kompilatorn kan även göra vissa optimeringar automatiskt, vilket vi strax kommer se

## Inlining av kod

- Kopiera funktionskroppen dit en funktion anropas
- Undviker overheaden av ett funktionsanrop (pusha saker på stacken, etc.)
- Oftast inte en optimering vi vill göra för hand!

```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; ++x) {  
        A[y][x] = add(B[y][x], C[y][x]);  
    }  
}
```



```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; ++x) {  
        A[y][x] = B[y][x] + C[y][x];  
    }  
}
```

# Loop-utrullning

- Veckla ut en loop för att minska overheaden av att räkna upp loopvariabeln

```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; ++x) {  
        A[y][x] = B[y][x] + C[y][x];  
    }  
}
```



```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; x += 4) {  
        A[y][x+0] = B[y][x+0] + C[y][x+0];  
        A[y][x+1] = B[y][x+1] + C[y][x+1];  
        A[y][x+2] = B[y][x+2] + C[y][x+2];  
        A[y][x+3] = B[y][x+3] + C[y][x+3];  
    }  
}
```



# Vektorisering

- Utnyttja vektorinstruktionerna (SIMD-instruktionerna) på CPU:n för att processa flera värden på samma gång
- Manuell vektorisering resulterar oftast i mer eller mindre oläslig kod...

```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; x += 4) {  
        A[y][x+0] = B[y][x+0] + C[y][x+0];  
        A[y][x+1] = B[y][x+1] + C[y][x+1];  
        A[y][x+2] = B[y][x+2] + C[y][x+2];  
        A[y][x+3] = B[y][x+3] + C[y][x+3];  
    }  
}
```



```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; x += 4) {  
        __m128 b = _mm_load_ps(&B[y][x]);  
        __m128 c = _mm_load_ps(&C[y][x]);  
        __m128 a = _mm_add_ps(b, c);  
        _mm_store_ps(&A[y][x], a);  
    }  
}
```

## Läsbarhet vs prestanda?

- En avvägning ibland, men oftast är läsbarhet mycket viktigare i längden
- Vi behöver dock sällan optimera koden på det här viset för hand...

```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; ++x) {  
        A[y][x] = add(B[y][x], C[y][x]);  
    }  
}
```

VS

```
for (int y = 0; y < h; ++y) {  
    for (int x = 0; x < w; x += 4) {  
        __m128 b = _mm_load_ps(&B[y][x]);  
        __m128 c = _mm_load_ps(&C[y][x]);  
        __m128 a = _mm_add_ps(b, c);  
        _mm_store_ps(&A[y][x], a);  
    }  
}
```

?

# Automatisk optimering via kompilatorn

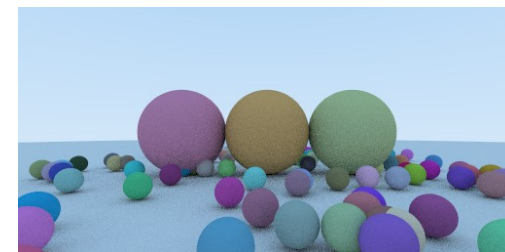
- Optimeringsnivåer i GCC:
  - O0 ingen optimering (standard)
  - O2 normal nivå (de flesta “säkra” optimeringar aktiverade)
  - O3 aggressiv nivå (loop-utrullning, autovektorisering, mfl.)
  - Os prioriterar kodstorlek över prestanda
- Några fler nyttiga kompilatorflaggor:
  - march och –mtune (låter kompilatorn optimera för en viss CPU-arkitektur)
  - ffast-math (tillåter att t.ex. flyttalsdivisioner ersätts med multiplikation)
  - fomit-frame-pointer

# Ingen optimering (gcc -O0) (4.54 sekunder)

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
28.32	1.29	1.29	117018880	0.01	0.03	intersect_sphere
27.33	2.53	1.24	117686739	0.01	0.01	sub
23.91	3.61	1.09	351056640	0.00	0.00	dot
11.02	4.11	0.50	914210	0.55	4.53	hit_closest
1.98	4.20	0.09	1318584	0.07	0.07	fract
1.87	4.29	0.09	2461125	0.03	0.03	add
1.21	4.34	0.06	439528	0.13	0.33	lookup_material
1.10	4.39	0.05	4289545	0.01	0.01	smul
0.88	4.43	0.04	524288	0.08	8.59	trace_scene
0.88	4.47	0.04				main
0.66	4.50	0.03	1353738	0.02	0.03	normalize
0.44	4.52	0.02	474682	0.04	0.13	sky
0.44	4.54	0.02	439528	0.05	0.08	rnd_ball
0.00	4.54	0.00	474682	0.00	0.06	mix
0.00	4.54	0.00	439528	0.00	0.00	mul
0.00	4.54	0.00	393216	0.00	0.00	clamp
0.00	4.54	0.00	1	0.00	0.00	create_image2d
0.00	4.54	0.00	1	0.00	0.00	destroy_image2d
0.00	4.54	0.00	1	0.00	0.00	setup_scene
0.00	4.54	0.00	1	0.00	0.00	write_to_ppm

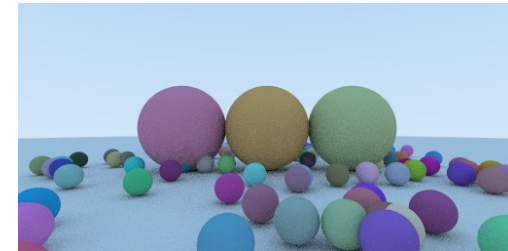


# Automatisk optimering (gcc -O2) (2.1 sekunder)

Flat profile:

Each sample counts as 0.01 seconds.

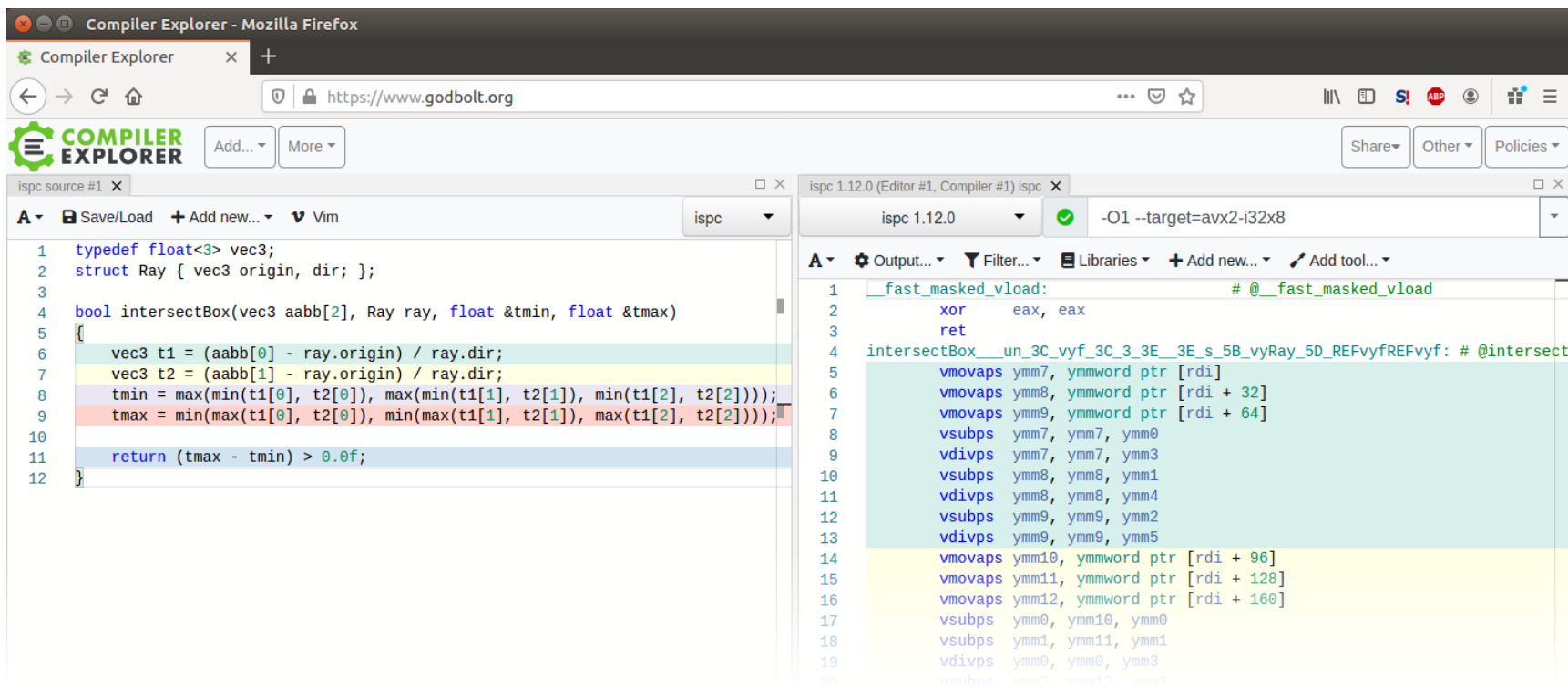
%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
77.19	1.62	1.62	117018880	0.01	0.01	intersect_sphere
12.87	1.89	0.27	914210	0.30	2.07	hit_closest
5.72	2.01	0.12	439528	0.27	0.27	lookup_material
1.91	2.05	0.04				trace_scene
0.95	2.07	0.02				write_to_ppm
0.48	2.08	0.01	1353738	0.01	0.01	normalize
0.48	2.09	0.01	474682	0.02	0.03	sky
0.48	2.10	0.01	439528	0.02	0.02	rnd_ball



(Var har sub, mul, mfl. tagit vägen?)

# Godbolt

- Webbaserat verktyg för att inspektera instruktionerna som olika kompilatorer och kompilatorflaggor genererar!



The screenshot shows the Compiler Explorer interface in Mozilla Firefox. The browser address bar shows <https://www.godbolt.org>. The interface includes a 'COMPILER EXPLORER' logo, navigation buttons, and a 'Compiler Explorer' tab.

The left pane shows the source code for a C++ program named 'ispc'. The code defines a `vec3` type and a `Ray` struct, then implements a `intersectBox` function. The function calculates the intersection of a ray with a box by finding the minimum and maximum values of the intersection parameters along each axis.

```

1  typedef float<3> vec3;
2  struct Ray { vec3 origin, dir; };
3
4  bool intersectBox(vec3 aabb[2], Ray ray, float &tmin, float &tmax)
5  {
6      vec3 t1 = (aabb[0] - ray.origin) / ray.dir;
7      vec3 t2 = (aabb[1] - ray.origin) / ray.dir;
8      tmin = max(min(t1[0], t2[0]), max(min(t1[1], t2[1]), min(t1[2], t2[2])));
9      tmax = min(max(t1[0], t2[0]), min(max(t1[1], t2[1]), max(t1[2], t2[2])));
10
11     return (tmax - tmin) > 0.0f;
12 }

```

The right pane shows the assembly output for the `intersectBox` function, compiled with `ispc 1.12.0` using the `-O1 --target=avx2-i32x8` flags. The assembly includes instructions for loading vector registers, performing arithmetic operations (xor, ret, vsubps, vdivps), and storing the results back to memory.

```

1  __fast_masked_vload:                                     # @__fast_masked_vload
2      xor     eax, eax
3      ret
4  intersectBox___un_3C_vyf_3C_3_3E_3E_s_5B_vyRay_5D_REFvyfREFvyf: # @intersect
5      vmovaps ymm7, ymmword ptr [rdi]
6      vmovaps ymm8, ymmword ptr [rdi + 32]
7      vmovaps ymm9, ymmword ptr [rdi + 64]
8      vsubps  ymm7, ymm7, ymm0
9      vdivps  ymm7, ymm7, ymm3
10     vsubps  ymm8, ymm8, ymm1
11     vdivps  ymm8, ymm8, ymm4
12     vsubps  ymm9, ymm9, ymm2
13     vdivps  ymm9, ymm9, ymm5
14     vmovaps ymm10, ymmword ptr [rdi + 96]
15     vmovaps ymm11, ymmword ptr [rdi + 128]
16     vmovaps ymm12, ymmword ptr [rdi + 160]
17     vsubps  ymm0, ymm10, ymm0
18     vsubps  ymm1, ymm11, ymm1
19     vdivps  ymm0, ymm0, ymm3
20     vsubps  ymm2, ymm12, ymm2

```



UPPSALA  
UNIVERSITET

# Parallellisering

- Ett annat sätt att få koden att köra snabbare (på flera processorkärnor eller trådar)
- Går ofta att kombinera med vanlig optimering
- OBS! All kod är inte parallelliserbar!