Wykład – kompilacja (cz. 2)

```
$  g++ plik_zrodlowy.cpp -o program  -O0
```
Brak optymalizacji

```
$  g++ plik_zrodlowy.cpp -o program  -O1
```
Częściowa optymalizacja

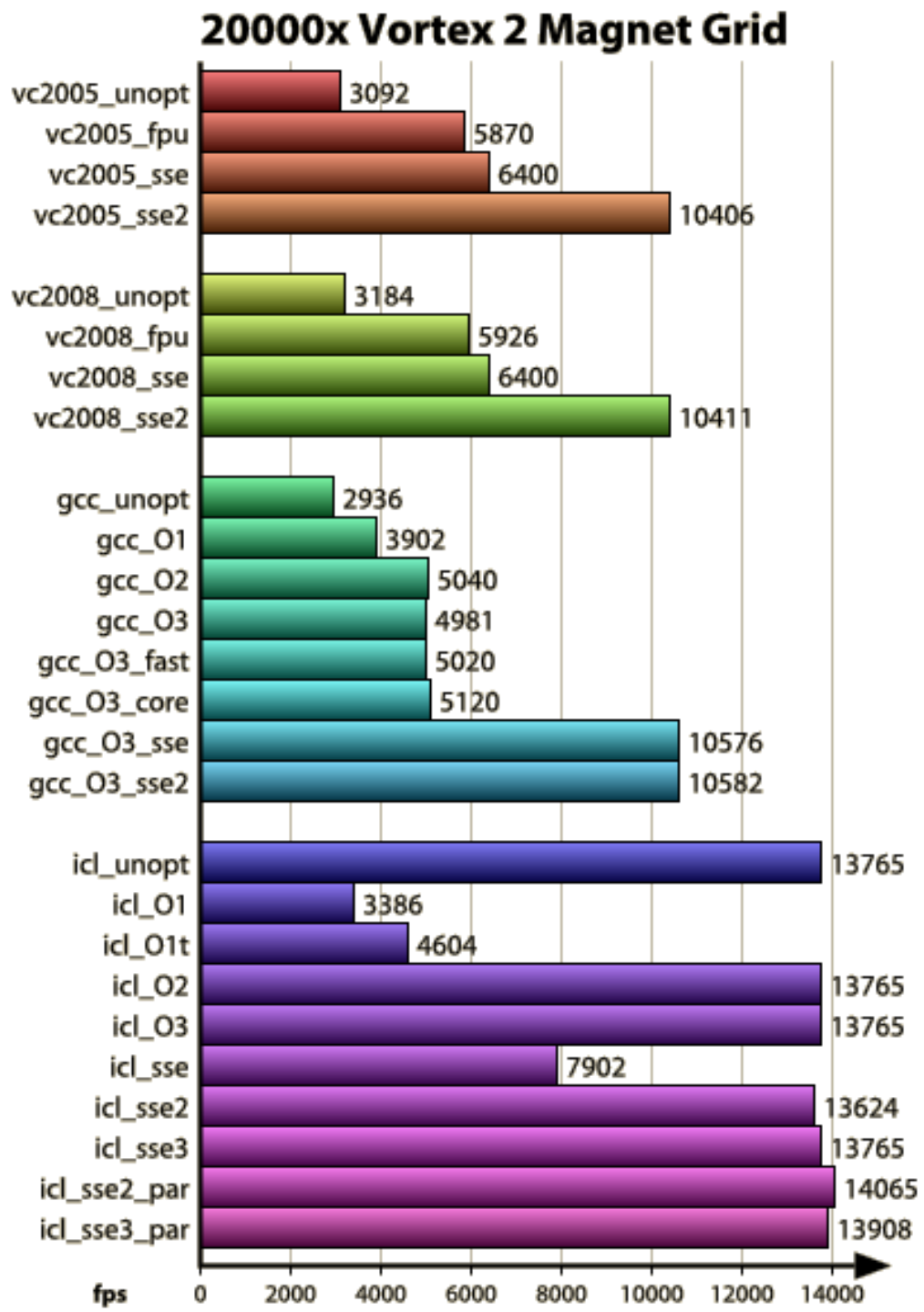
```
$  g++ plik_zrodlowy.cpp -o program  -O2
```
Pełna optymalizacja

```
$  g++ plik_zrodlowy.cpp -o program  -O3
```
Agresywna optymalizacja

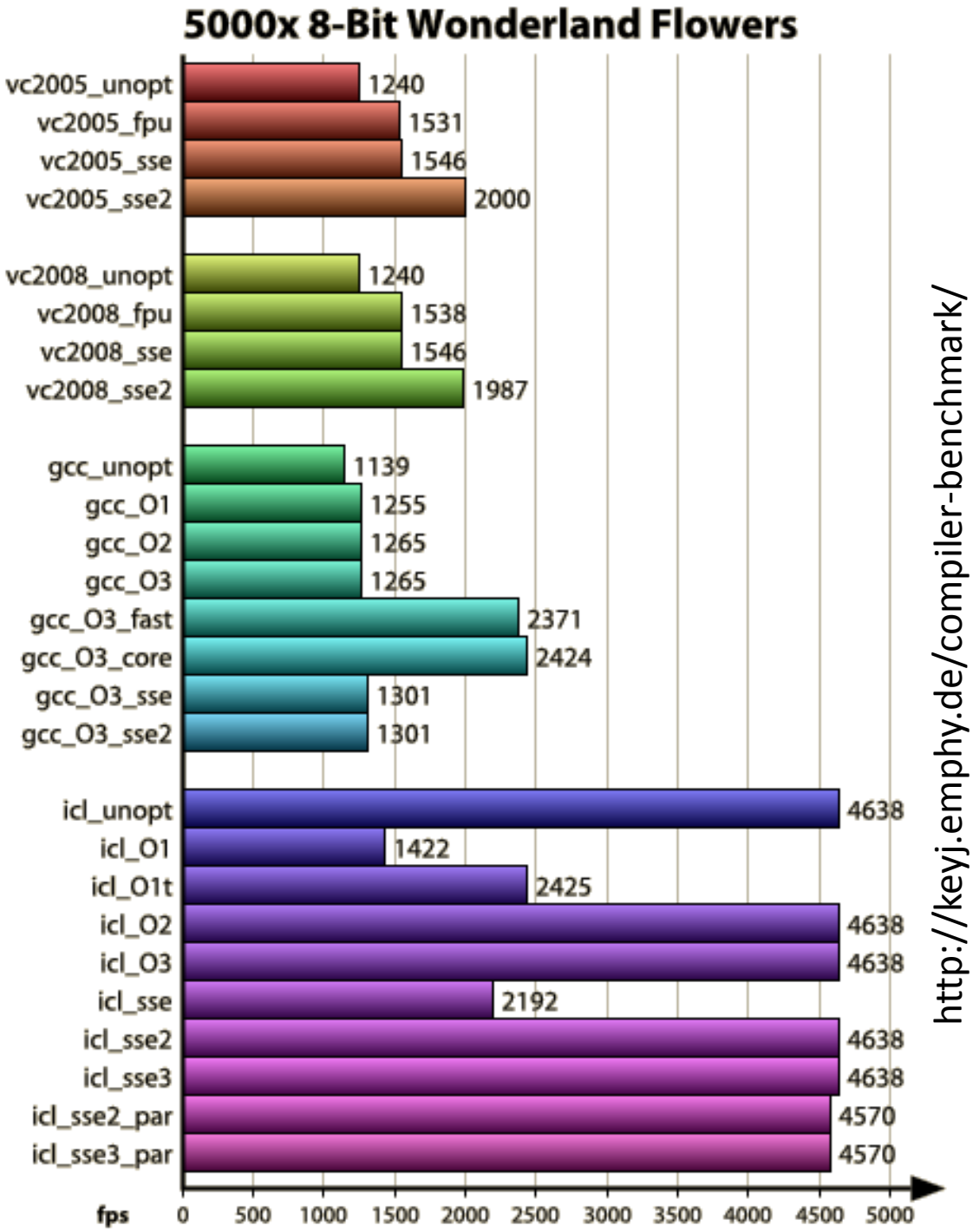
```
$  g++ plik_zrodlowy.cpp -o program  -Os
```
Optymalizacja rozmiaru

```
$  g++ plik_zrodlowy.cpp -o program  -Ofast
```
Lepiej unikać ☺

`-march=native` - optymalizacja pod konkretny procesor



http://keyj.emphy.de/compiler-benchmark/


5000x 8-Bit Wonderland Flowers
vc2005_unopt
vc2005_fpu
vc2005_sse
vc2005_sse2
vc2008_unopt
vc2008_fpu
vc2008_sse
vc2008_sse2
gcc_unopt
gcc_O1
gcc_O2
gcc_O3
gcc_O3_fast
gcc_O3_core
gcc_O3_sse
gcc_O3_sse2
icl_unopt
icl_O1
icl_O1t
icl_O2
icl_O3
icl_sse
icl_sse2
icl_sse3
icl_sse2_par
icl_sse3_par
1240
1531
1546
2000
1240
1538
1546
1987
1139
1255
1265
1265
2371
2424
1301
1301
4638
1422
2425
4638
4638
2192
4638
4638
4570
4570
fps
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
http://keyj.emphy.de/compiler-benchmark/

Program **make** i jego pliki opisowe

Program **make** służy do automatycznego wykonywania sekwencji zaspecyfikowanych zadań, a ogół "zrobienia" czegoś ze składników. Wygląda to jak duplikowanie skryptów csh, sh, itp., ale są dwie istotne różnice:

1. Zadanie z sekwencji jest realizowane jeżeli przynajmniej jeden z jego składników jest nowszy, niż plik będący wynikiem jego realizacji.
2. Automatycznie sprawdzane są i wykonywane w razie potrzeby wszystkie zadania, których pliki wynikowe są składnikami danego zadania.

Składnia polecenia:

make [-f *plik_opisowy*] [*opcje*] [*definicje*] [*zadanie*]

Standardowymi plikami opisowymi są Makefile lub makefile; jeden z nich musi być w aktualnym katalogu, jeżeli nie podajemy explicite pliku opisowego opcją –f.

Struktura pliku opisowego

Komentarze jak w skryptach unixowych; może być w dowolnym miejscu.
#
Najpierw definiujemy zmienne, jeżeli ich potrzebujemy. Do zdefiniowanych
zmiennych odwołujemy się jak w unixie przez ${*zmienna*} lub $(*zmienna*)
#
zmienna1=wartość1
…
zmiennaN=wartośćN

Teraz definiujemy zadania do wykonania. Uwaga! (TAB) oznacza znak tabulacji.
#
zadanie1:*lista_składników_zadania* (dependencies)
(TAB) *instrukcja1*
…
(TAB) *instrukcjaN1*
…
zadanieM:*lista_składników_zadania*
(TAB) *sposób wykonania*
…
(TAB) instrukcjaNM

Każda linia może być kontynuowana w następnej; w takim przypadku musi się ona
kończyć znakiem “\” a linia kontynuacji musi się rozpoczynać znakiem tabulacji.

Przykładowy plik Makefile

```
CC = gcc

all: pierwszy.o drugi.o trzeci.o czwarty.o
  $(CC) pierwszy.o drugi.o trzeci.o czwarty.o -o test

pierwszy.o: pierwszy.c pierwszy.h
  $(CC) pierwszy.c -c -o pierwszy.o

drugi.o: drugi.c drugi.h trzeci.h czwarty.h
  $(CC) drugi.c -c -o drugi.o

trzeci.o: trzeci.c trzeci.h
  $(CC) trzeci.c -c -o trzeci.o

czwarty.o: czwarty.c
  $(CC) czwarty.c -c -o czwarty.o
```

Jeszcze poziom wyżej:

- qmake – program biblioteki Qt
- cmake – program do generowania
- maven, ant, gradle – głównie dla Javy
- autoconf/automake

- Przenośny (Linux, Windows, Mac)
- Wygodne zarządzanie zależnościami
- Prosty w stosowaniu
- Bardzo dobrze się skaluje

- Przydatne narzędzia: CPack/CTest/CDash

CMake: stosowanie

1. Tworzymy plik `CMakeLists.txt`
2. Wywołujemy polecenie `cmake` a potem `make`

Ad. 1 Przykłady:

```
PROJECT( helloworld )
SET( hello_SRCS hello.cpp )
ADD_EXECUTABLE( hello ${hello_SRCS} )
```

```
PROJECT( mylibrary )
SET( mylib_SRCS library.cpp )
ADD_LIBRARY( my SHARED ${mylib_SRCS} )
```

Dalsze przykłady:

```
PROJECT(clockapp)
ADD_SUBDIRECTORY(libwakeup)
ADD_SUBDIRECTORY(clock)
```

```
PROJECT( pfrac )

FIND_PACKAGE( Qt4 REQUIRED )
INCLUDE( ${QT_USE_FILE} )

SET( pfrac_SRCS main.cpp client.h client.cpp )
SET( pfrac_MOC_HEADERS client.h )

QT4_ADD_RESOURCES( pfrac_SRCS ${PROJECT_SOURCE_DIR}/pfrac.qrc )
QT4_WRAP_CPP( pfrac_MOC_SRCS ${pfrac_MOC_HEADERS} )

ADD_EXECUTABLE( pfrac ${pfrac_SRCS} ${pfrac_MOC_SRCS}
TARGET_LINK_LIBRARIES( pfrac ${QT_LIBRARIES} )
```

Debugger, np. gdb

Pomaga śledzić wykonanie programu

- Krok po kroku (*step*, *next*)
- *Breakpoint*
- *Watchpoint*

- Na podstawie odpowiednich komentarzy generuje dokumentację
- Standardowo HTML i/lub LaTeX
- Szeroka gama wspieranych języków

Znaczące komentarze

```
/**
 * ... text ...
 */
```

```
/*!
 * ... text ...
 */
```

```
///
/// ... text ...
///
```

```
/*****************************************//**
 *   ... text
 *****************************************/
```

i inne...

Dokumentujemy

- Pliki
- Klasy
- Funkcje (metody)
  - Parametry
  - Zwracane wyniki
- Inne