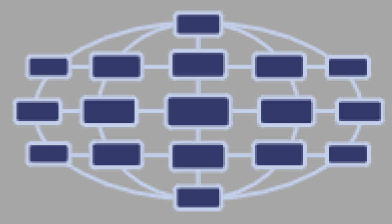


Новосибирский государственный университет
Факультет информационных технологий
Кафедра параллельных вычислений

Эффективное программирование современных микропроцессоров и мультипроцессоров

Использование компиляторов для оптимизации программ

Преподаватели:
Киреев С.Е.
Калгин К.В.

Компилятор

- Компилятор – это программа, которая преобразует текст программы на языке программирования в машинный код.

- Примеры компиляторов
  – GNU Compiler Collection: gcc, g++, gfortran, …
  – Intel C/C++ Compiler: icc, icpc, ifort
  – LLVM/Clang: clang
  – Microsoft C/C++ Compiler: cl.exe
  – x86 Open64 Compiler Suite: opencc, openCC, openf90, …
  – …

- Как правильно воспользоваться компилятором?
  – Дать компилятору информацию об алгоритме
  – Дать компилятору информацию об оборудовании
  – Сообщить компилятору, что от него требуется

КЛЮЧИ КОМПИЛЯТОРА ДЛЯ ОПТИМИЗАЦИИ КОДА

Ключи компилятора для оптимизации кода

- Ключи, задающие уровень оптимизации – набор конкретных настроек
  - -O0, -O1, -O2, -O3, -O, -Osize, -Ofast, -fast, …
- Ключи, задающие целевую архитектуру, расширения системы команд
  - -march=native, -mtune=skylake, -mavx2, -fma, …
- Ключи, задающие правила использования вещественной арифметики
  - -ffast-math, -fast-transcendentals, -fp-model fast=2
- Ключи, указывающие, что программа соответствует правилам «strict aliasing»
  - -ansi-alias, -fstrict-aliasing
- Ключи для оптимизации на основе сбора статистики
  - -fprofile-generate, -fprofile-use

Что такое Strict Aliasing и почему нас должно это волновать? (habr.com): [Часть 1], [Часть 2]

Ключи компилятора для оптимизации кода

- Ключи, задающие целевую архитектуру, расширения системы команд

  -march=native, -mtune=skylake, -mavx2, -fma, …

Ключи компилятора для оптимизации кода

- Ключи, задающие целевую архитектуру, расширения системы команд

  -march=native, -mtune=skylake, -mavx2, -fma, …

- Как получить эту информацию?
  - Linux: cat /proc/cpuinfo, lscpu, …
  - Windows: CPU-Z
  - Google…

kireev@mineral: ~

```
kireev@mineral:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
stepping        : 9
microcode       : 0x84
cpu MHz         : 900.199
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 cl
flush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pn
i pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowpre
fetch cpuid_fault invpcid_single pti retpoline intel_pt rsb_ctxsw spec_ctrl tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdsee
d adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify
hwp_act_window hwp_epp
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 7200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
```

Пример

kireev@mineral: ~

```
kireev@mineral:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
stepping        : 9
microcode       : 0x84
cpu MHz         : 900.199
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 cl
flush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pn
i pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowpre
fetch cpuid_fault invpcid_single pti retpoline intel_pt rsb_ctxsw spec_ctrl tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdsee
d adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify
hwp_act_window hwp_epp
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 7200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
```

Пример

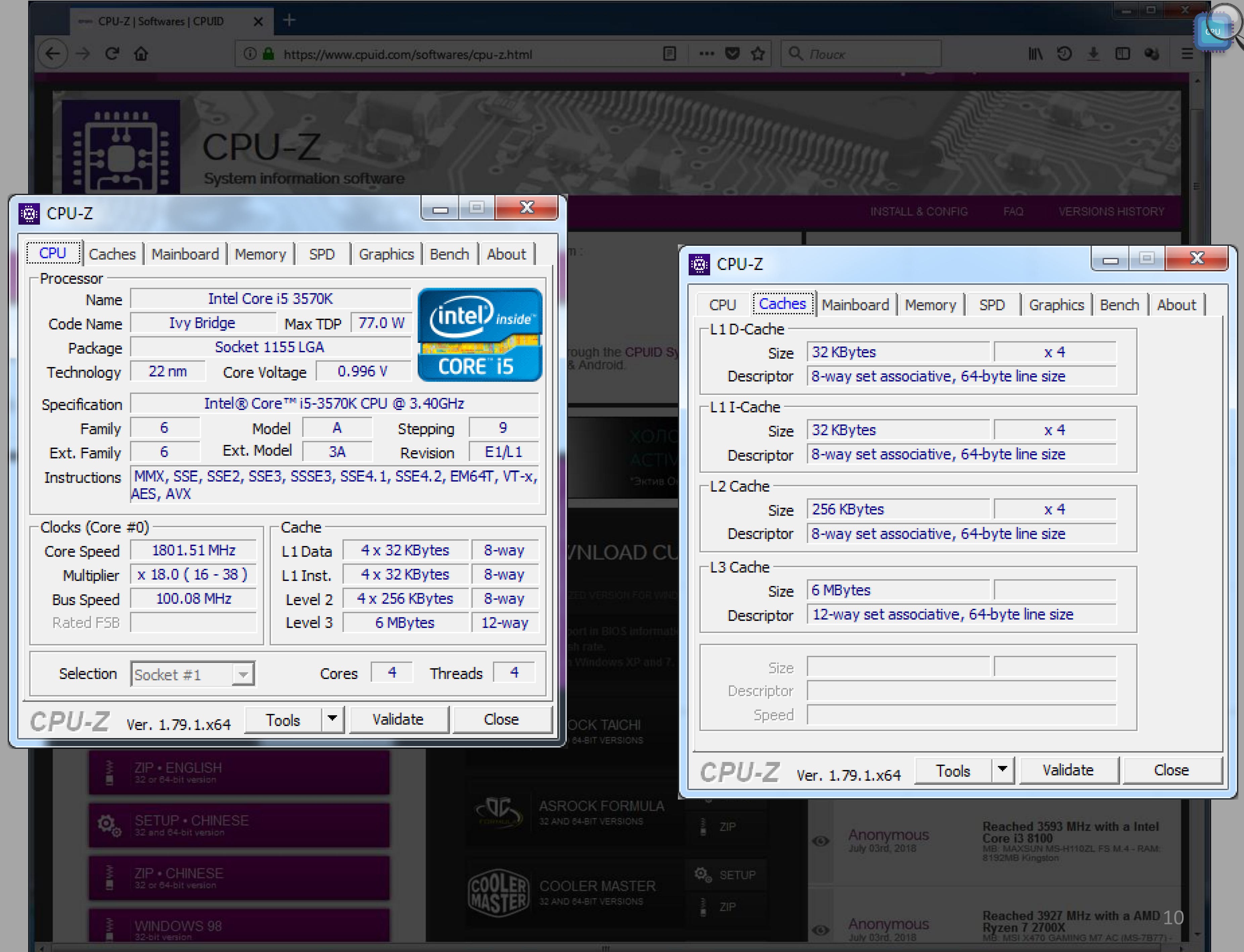
```
kireev@mineral:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      :
stepping        :
microcode       :
cpu MHz         : 900.199
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 cl
flush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pn
i pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowpre
fetch cpuid_fault invpcid_single pti retpoline intel_pt rsb_ctxsw spec_ctrl tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdsee
d adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify
hwp_act_window hwp_epp
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 7200.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
```

Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz

https://en.wikipedia.org/wiki/List_of_Intel_Core_i7_microprocessors

Пример


CPU-Z | Softwares | CPUID
https://www.cpuid.com/softwares/cpu-z.html
Поиск
CPU-Z
System information software
INSTALL & CONFIG
FAQ
VERSIONS HISTORY
CPU-Z
CPU
Caches
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
Intel Core i5 3570K
Code Name
Ivy Bridge
Max TDP
77.0 W
Package
Socket 1155 LGA
Technology
22 nm
Core Voltage
0.996 V
intel inside
CORE i5
Specification
Intel® Core™ i5-3570K CPU @ 3.40GHz
Family
6
Model
A
Stepping
9
Ext. Family
6
Ext. Model
3A
Revision
E1/L1
Instructions
MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX
Clocks (Core #0)
Core Speed
1801.51 MHz
Multiplier
x 18.0 (16 - 38)
Bus Speed
100.08 MHz
Rated FSB
Cache
L1 Data
4 x 32 KBytes
8-way
L1 Inst.
4 x 32 KBytes
8-way
Level 2
4 x 256 KBytes
8-way
Level 3
6 MBytes
12-way
Selection
Socket #1
Cores
4
Threads
4
CPU-Z
Ver. 1.79.1.x64
Tools
Validate
Close
CPU-Z
CPU
Caches
Mainboard
Memory
SPD
Graphics
Bench
About
L1 D-Cache
Size
32 KBytes
x 4
Descriptor
8-way set associative, 64-byte line size
L1 I-Cache
Size
32 KBytes
x 4
Descriptor
8-way set associative, 64-byte line size
L2 Cache
Size
256 KBytes
x 4
Descriptor
8-way set associative, 64-byte line size
L3 Cache
Size
6 MBytes
Descriptor
12-way set associative, 64-byte line size
Size
Descriptor
Speed
CPU-Z
Ver. 1.79.1.x64
Tools
Validate
Close
ZIP • ENGLISH
32 or 64-bit version
SETUP • CHINESE
32 and 64-bit version
ZIP • CHINESE
32 or 64-bit version
WINDOWS 98
32-bit version
ASROCK FORMULA
32 AND 64-BIT VERSIONS
COOLER MASTER
32 AND 64-BIT VERSIONS
ZIP
SETUP
Anonymous
July 03rd, 2018
Reached 3593 MHz with a Intel Core i3 8100
Anonymous
July 03rd, 2018
Reached 3927 MHz with a AMD Ryzen 7 2700X

Ключи компилятора для оптимизации кода

Указание набора команд в GCC (Intel Compiler)

- -march=…, -x… – использовать инструкции для заданной архитектуры. Программа может не работать на других архитектурах.
  - i386, x86-64, pentium-mmx, pentium4, nehalem, sandybridge, broadwell, skylake, knl, icelake-client, k6, athlon, opteron, barcelona, znver2, btver2, winchip2, c3, nano, geode, …
  - native, -xHost – определить архитектуру при компиляции

Ключи компилятора для оптимизации кода

Указание набора команд в GCC (Intel Compiler)

- -march=…, -x… – использовать инструкции для заданной архитектуры. Программа может не работать на других архитектурах.
  - i386, x86-64, pentium-mmx, pentium4, nehalem, sandybridge, broadwell, skylake, knl, icelake-client, k6, athlon, opteron, barcelona, znver2, btver2, winchip2, c3, nano, geode, …
  - native, -xHost – определить архитектуру при компиляции

- -mauto-arch=…, -ax… – генерировать дополнительную версию кода под заданную архитектуру, если это даст выигрыш в производительности

Ключи компилятора для оптимизации кода

Указание набора команд в GCC (Intel Compiler)

- -march=…, -x… – использовать инструкции для заданной архитектуры. Программа может не работать на других архитектурах.
  - i386, x86-64, pentium-mmx, pentium4, nehalem, sandybridge, broadwell, skylake, knl, icelake-client, k6, athlon, opteron, barcelona, znver2, btver2, winchip2, c3, nano, geode, …
  - native, -xHost – определить архитектуру при компиляции

- -mauto-arch=…, -ax… – генерировать дополнительную версию кода под заданную архитектуру, если это даст выигрыш в производительности

- -mmmx, -m3dnow, -msse, -msse2, -msse3, -msse4, -mavx, -mavx2, -mfma, … – разрешить компилятору использовать заданное расширение системы команд.

Ключи компилятора для оптимизации кода

Указание набора команд в GCC (Intel Compiler)

- -march=…, -x… – использовать инструкции для заданной архитектуры. Программа может не работать на других архитектурах.
  - i386, x86-64, pentium-mmx, pentium4, nehalem, sandybridge, broadwell, skylake, knl, icelake-client, k6, athlon, opteron, barcelona, znver2, btver2, winchip2, c3, nano, geode, …
  - native, -xHost – определить архитектуру при компиляции

- -mauto-arch=…, -ax… – генерировать дополнительную версию кода под заданную архитектуру, если это даст выигрыш в производительности

- -mmmx, -m3dnow, -msse, -msse2, -msse3, -msse4, -mavx, -mavx2, -mfma, … – разрешить компилятору использовать заданное расширение системы команд.

- -mtune=… – оптимизировать под заданную архитектуру, но сохранить совместимость с другими архитектурами.
  - Варианты как у -march.
  - native – определить архитектуру при компиляции
  - generic – оптимизировать для «общего случая»
  - intel – оптимизировать под «современные процессоры Intel» (см. man gcc).

Ключи компилятора для оптимизации кода

Управление вещественной арифметикой в GCC и Intel Compiler

- -mfpmath=… – реализовать вещественную арифметику заданным способом.
  - 387 – использовать сопроцессор (80-битное представление вещественных чисел в регистрах)
  - sse – использовать расширение SSE (32 и 64-битное представление вещественных чисел в регистрах)
  - sse+387 – использовать оба устройства одновременно (экспериментальная опция)

Ключи компилятора для оптимизации кода

Управление вещественной арифметикой в GCC и Intel Compiler

- -mfpmath=… – реализовать вещественную арифметику заданным способом.
  - 387 – использовать сопроцессор (80-битное представление вещественных чисел в регистрах)
  - sse – использовать расширение SSE (32 и 64-битное представление вещественных чисел в регистрах)
  - sse+387 – использовать оба устройства одновременно (экспериментальная опция)

- -ffast-math, -fast-transcendentals, -fp-model fast=2 – использование более быстрой и менее точной арифметики

- -fp-model … – управление семантикой вещественных вычислений
  - precise, fast=1, fast=2, consistent, strict, source, double, extended

Ключи компилятора для оптимизации кода

Оптимизация на основе сбора статистики

1. Компиляция программы с добавлением кода для сбора статистики
   – $ g++ -O1 -fprofile-generate -o prog prog.c
   – Создаётся файл: prog

2. Запуск программы много раз на типичных данных – сбор статистики
   – $ ./prog
   – Создаётся файл: prog.gcda

3. Компиляция программы с оптимизацией с учётом статистики
   – $ g++ -O1 -fprofile-use -o prog_opt prog.c
   – Создается файл: prog_opt

Ключи компилятора для

Оп

1. …бора статистики

2. …сбор статистики

3. …татистики
   - $ g++ -O1 -fprofile-use -o prog_opt prog.c
   - Создается файл: prog_opt

Пример (хороший)

```
$time ./cache
real     0m40.409s
user     0m40.323s
sys      0m0.016s
$time ./cache_opt
real     0m28.768s
user     0m28.710s
sys      0m0.008s
```

Ключи компилятора для

Оп

1.

2.

3.

– $ g++ -O1 -f

– Создается с

Пример (хороший)

```
$time ./cache
real      0m40.409s
user
sys
$time ./
real
user
sys
```

Пример (плохой)

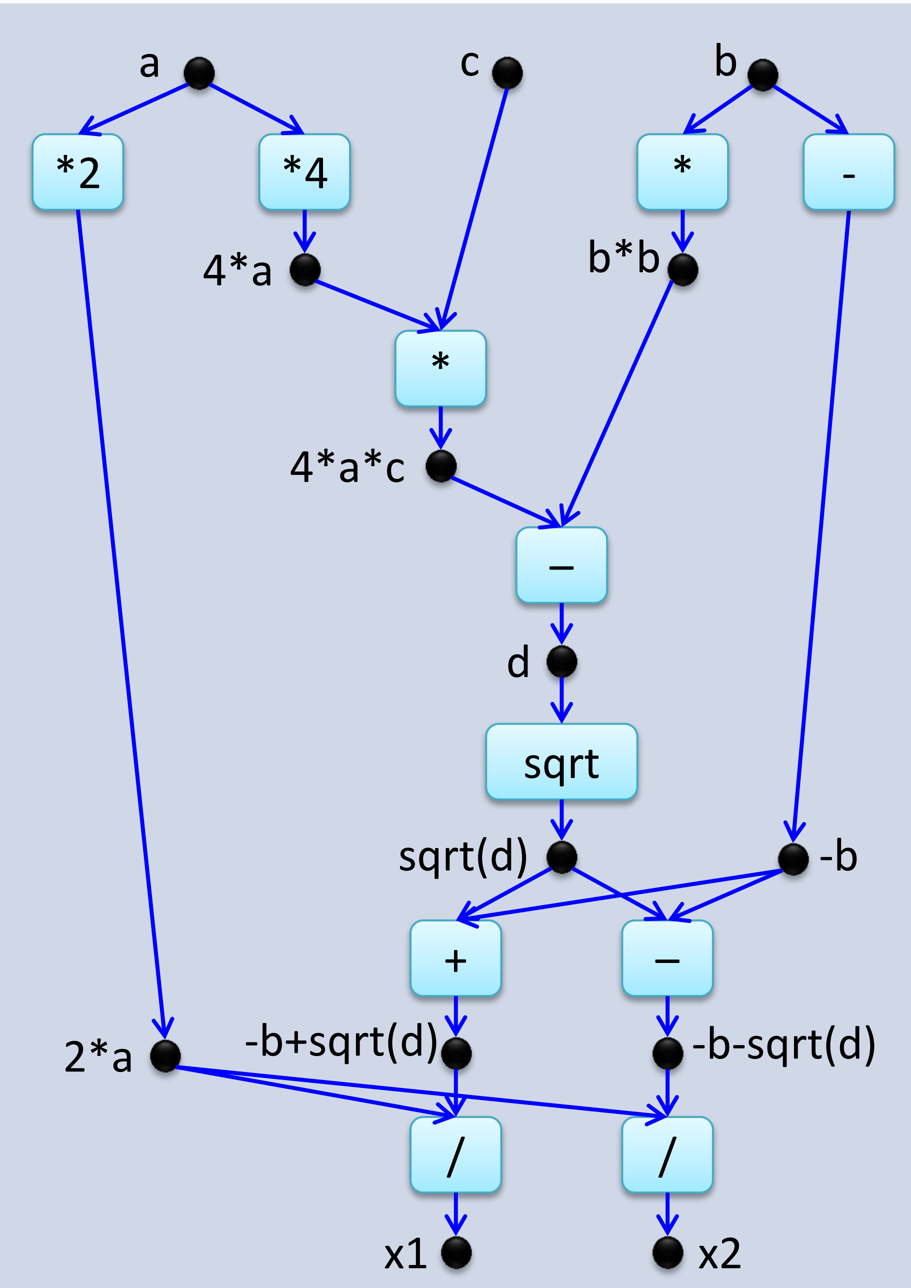
```
$time ./wave
real      0m12.442s
user      0m12.439s
sys       0m0.004s
$time ./wave_opt
real      0m15.714s
user      0m15.706s
sys       0m0.008s
```

ПРИНЦИПЫ НАПИСАНИЯ КОДА ДЛЯ ОПТИМИЗАЦИИ КОМПИЛЯТОРОМ

Принципы написания кода для оптимизации компилятором

- **Дать компилятору как можно больше информации об алгоритме**
  - Использование принципов структурного программирования для описания алгоритма
    - Писать хорошо структурированный код
    - Не использовать goto
    - Стараться не использовать break, continue
    - …
  - Распознавание времени жизни значений
    - Где значение вырабатывается и где потребляется, области видимости переменных
    - Какие переменные (области памяти) меняют значения, а какие нет (const, constexpr)
    - …
  - Независимость вычислений
    - Независимые итерации циклов (#pragma GCC ivdep)
    - Чистые функции (pure functions)
    - Наложение объектов в памяти (ключевое слово restrict, правила strict aliasing)
    - …
  - Использование стандартных библиотек
    - Компилятор знает «смысл» операций
  - …

Алгоритм	Программа




```
float a,b,c;
float d,x1,x2;

d = b*b - 4*a*c;
x1 = (-b + sqrt(d)) / (2*a);
x2 = (-b - sqrt(d)) / (2*a);
```

Принципы написания кода для оптимизации компилятором

- **Дать компилятору как можно больше информации об алгоритме**

- **Перевести как можно больше вычислений на этап компиляции**
  - Константные выражения
  - constexpr-функции
  - template<>
  - …

Принципы написания кода для оптимизации компилятором

- **Дать компилятору как можно больше информации об алгоритме**
- **Перевести как можно больше вычислений на этап компиляции**
- **Указывать компилятору реализацию конкретных элементов кода**
  - inline
  - register
  - выравнивание
  - …

ПРИМЕРЫ ТЕХНИК ОПТИМИЗАЦИИ, ПРИМЕНЯЕМЫХ КОМПИЛЯТОРАМИ

Типы оптимизаций компилятора по сфере применения

- Peelhole-оптимизация
  - Рассматривается несколько соседних инструкций
- Локальная оптимизация
  - Рассматривается один базовый блок
- Внутрипроцедурная оптимизация
  - Рассматривается одна подпрограмма
- Оптимизация циклов
  - Рассматривается один или несколько вложенных циклов
- Межпроцедурная оптимизация
  - Рассматривается весь исходный код программы

Подстановка функций
Function inlining

- Подстановка тела функции вместо вызова
  - Применяется для небольших функций, вызываемых из небольшого числа мест.
- Пример:

`float square (float a) { return a * a; }`
`float parabola (float x){`
` return square(x) + 1.0f;`
`}`
	`float parabola (float x) {`
	`  return x * x + 1.0f;`
	`}`

- Преимущества:
  - Устраняются накладные расходы на вызов и возврат из подпрограммы, на передачу параметров.
  - Код кэшируется более эффективно, т.к. становится последовательным.
  - Размер кода уменьшается, если вызов функции только один.
  - Появляется возможность других оптимизаций.
- Недостатки
  - Может увеличиться размер кода

Подстановка функций
Function inlining

- Встроенная функция может быть вызвана из другого модуля. Компилятор вынужден сделать невстроенную копию встроенной функции. Эта невстроенная функция является мертвым кодом, если она никем не вызывается. Такие функции повышают фрагментацию кода, в результате чего кэш команд работает менее эффективно.

- Способы устранения невстроенной копии функции:
  - Объявить функцию как static
    - Делает возможными другие оптимизации
  - Некоторые компиляторы имеют специальную опцию, которая позволяет линковщику удалить все функции, к которым нет обращений.

Свертка и распространение констант Constant folding and constant propagation

- Выражение, содержащее только константы, заменяется вычисленным результатом

  a = b + 2.0/3.0; → a = b + 0.666666666666667;

  - Рекомендуется такие выражения заключать в скобки

    b*2.0/3.0 → (b*2.0)/3.0

    b*(2.0/3.0) → b*0.666666666666667

  - Константа может быть продвинута через серию выражений

```
float parabola (float x) { return x * x + 1.0f; }
float a, b;
a = parabola (2.0f);
b = a + 1.0f;
```

⇒

```
float a, b;
a = 5.0f;
b = 6.0f;
```

  - Свертка и распространение констант не могут быть выполнены, если выражение содержит функцию, которая не может быть подставлена или вычислена во время компиляции.

Устранение указателя
Pointer elimination

- Указатель или ссылка могут быть устранены, если компилятору известно, куда они указывают.

`void Plus2 (int * p) { *p = *p + 2; }`
`int a;`
`Plus2 (&a);` ⇨
	`a += 2;`

Устранение общих подвыражений Common subexpression elimination

- Если одно и то же подвыражение встречается более одного раза, тогда компилятор может вычислить его один раз.

<table>
<tr>
<td>

```
int a, b, c;

b = (a+1) * (a+1);
c = (a+1) / 4;
```

</td>
<td>

```
int a, b, c, temp;
temp = a+1;
b = temp * temp;
c = temp / 4;
```

</td>
</tr>
</table>

Регистровые переменные Register variables

- Самые часто используемые переменные компилятор размещает на регистрах (для всей области жизни переменной или для её части)

- Типичные кандидаты на размещение на регистрах:
    - Промежуточные значения в выражениях, счётчики циклов, параметры функций, указатели, ссылки, указатель this, общие подвыражения, индукционные переменные.

- Переменная размещается в памяти, если
  - Определяется её адрес (есть указатель или ссылка на неё)
  - Она объявлена как глобальная или статическая

- Переменная не может быть временно размещена на регистре, если
  - Она была объявлена как volatile

Анализ времени жизни переменной Live range analysis

- Оптимизирующий компилятор может использовать один и тот же регистр для нескольких переменных, если их диапазоны использования не пересекаются, или если их значения гарантированно совпадают.

```
int SomeFunction (int a, int x[])
{
  int b, c;
  x[0] = a;
  b = a + 1;
  x[1] = b;
  c = b + 1;
  return c;
}
```



```
int SomeFunction (int a, int x[])
{
  int b, c;
  x[0] = a;
  b = a + 1;
  x[1] = b;
  c = a + 2;
  return c;
}
```


- Для объектов в памяти эта техника не используется – для каждого объекта выделяется своя область памяти.

Объединение одинаковых ветвей Join identical branches

- Код может стать более компактным благодаря объединению одинаковых частей кода.

```double x, y, z;```	```double x, y;```

```
double x, y, z;
bool b;
if (b) {
  y = sin(x);
  z = y + 1.;
}
else {
  y = cos(x);
  z = y + 1.;
}
```

⇨

```
double x, y;
bool b;
if (b) {
  y = sin(x);
}
else {
  y = cos(x);
}
z = y + 1.;
```

Устранение переходов Eliminate jumps

- Переходы могут быть устранены путем копирования кода, на который они ведут.

`int SomeFunction (int a, bool b) {`
`  if (b) { a = a * 2; }`
`  else   { a = a * 3; }`
`  return a + 1;`
`}`
	`int SomeFunction (int a, bool b) {`
	`  if (b) {`
	`    a = a * 2;`
	`    return a + 1;`
	`  }`
	`  else {`
	`    a = a * 3;`
	`    return a + 1;`
	`  }`
	`}`

- Переход может быть устранен, если условие является всегда истинным или ложным.

`if (true) { a = b; }` `else      { a = c; }`	`a = b;`

Устранение переходов
Eliminate jumps

- Переход также может быть устранен, если значение условного выражения известно из предыдущего перехода.

```
int SomeFunction (int a, bool b) {

  if (b) { a = a * 2; }
  else   { a = a * 3; }

  if (b) { return a + 1; }
  else   { return a - 1;}
}
```

⇒

```
int SomeFunction (int a, bool b) {
  if (b) {
    a = a * 2;
    return a + 1;
  }
  else {
    a = a * 3;
    return a - 1;
  }
}
```

Раскрутка цикла
Loop unrolling

- Уменьшение числа итераций с увеличением тела цикла
  - Плюсы
    - Уменьшаются накладные расходы на организацию цикла,
    - Появляются возможности для дополнительных оптимизаций
  - Минусы
    - Тело цикла может стать слишком большим, занять слишком много места в кэше команд и не поместиться в цикловой буфер.

- Циклы с малым числом итераций могут быть раскрыты полностью.

```
int i, a[2];
for (i = 0; i < 2; i++) a[i] = i+1;
```

⇨

```
int a[2];
a[0] = 1; a[1] = 2;
```

Вынос инвариантного кода за цикл Loop invariant code motion

- Вычисление может быть вынесено из цикла, если оно не зависит от счетчика цикла.

```
int i, a[100], b;

for (i = 0; i < 100; i++) {
  a[i] = b * b + 1;
}
```

```
int i, a[100], b, temp;
temp = b * b + 1;
for (i = 0; i < 100; i++) {
  a[i] = temp;
}
```

Индуктивные переменные Induction variables

- Выражение, являющееся линейной функцией от счетчика цикла, может быть вычислено путем прибавления константы к предыдущему значению.

```
int i, a[100];
for (i = 0; i < 100; i++) {
  a[i] = i * 9 + 3;
}
```

→

```
int i, a[100], temp;
temp = 3;
for (i = 0; i < 100; i++) {
  a[i] = temp;
  temp += 9;
}
```

- Индуктивные переменные часто используются для вычисления адресов элементов массива.

```
struct S1 { double a; double b; };
S1 list[100]; int i;
for (i = 0; i < 100; i++) {
  list[i].a = 1.0;
  list[i].b = 2.0;
}
```

→

```
struct S1 { double a; double b; };
S1 list[100], *temp;
for(temp=&list[0];temp<&list[100];temp++){
  temp->a = 1.0;
  temp->b = 2.0;
}
```

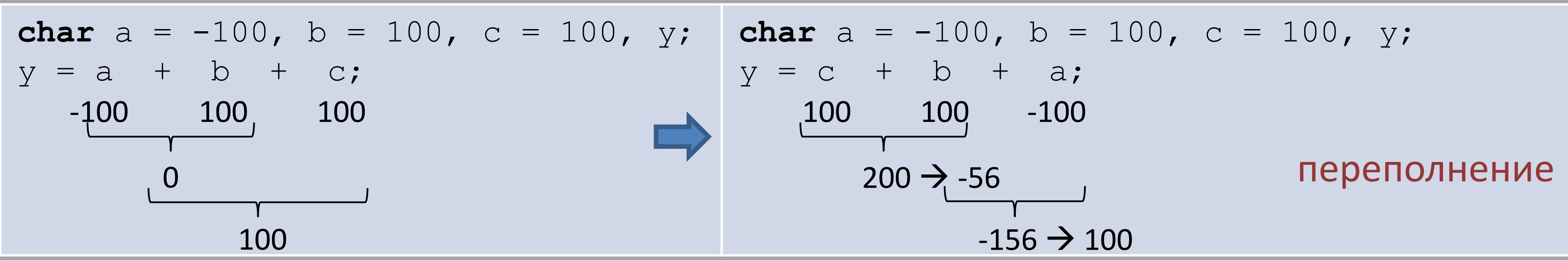
Планирование Scheduling

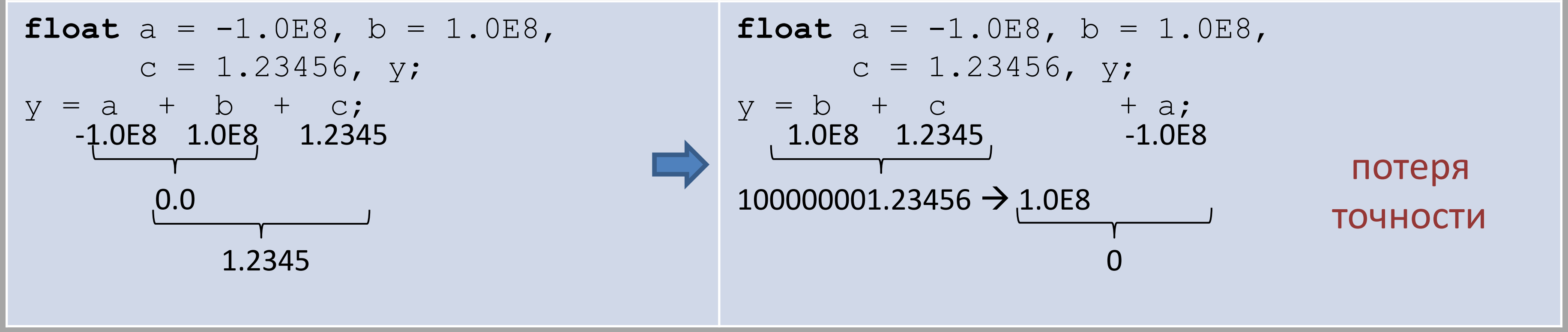
- Компилятор может переупорядочить команды с целью параллельного исполнения.

float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
	float a, b, c, d, e, f, x, y;
	ab = a + b;
	de = d + e;
	x = ab + c;
	y = de + f;

Алгебраические преобразования Algebraic reductions

- Большинство компиляторов могут упрощать простые алгебраические выражения, используя фундаментальные законы алгебры.
- Компиляторы лучше упрощают целочисленные выражения, чем вещественные, из-за опасности потери точности.





Алгебраические преобразования Algebraic reductions

- Компиляторы меняют порядок вещественных операндов только при указании специальной опции
    - gcc -ffast-math
    - icc -fp-model fast[=1|2]
- Некоторые «очевидные» преобразования вещественных выражений компиляторы не делают:
    - Не создаёт индуктивных переменных
    - 0.0 / a = 0.0
    - …
- Нельзя полагаться на то, что компилятор сделает какие-либо арифметические преобразования вещественного кода, и можно рассчитывать только на самые простые преобразования целочисленного кода. Более безопасно делать упрощения вручную.

Девиртуализация Devirtualization

- Оптимизирующий компилятор может пропустить просмотр таблицы виртуальных функций, если он знает, какая версия виртуальной функции необходима.

```
class C0 {
  public:
    virtual void f();
};
class C1 : public C0 {
  public:
    virtual void f();
};
void g() {
  C1 obj1;
  C0 * p = & obj1;
  p->f(); // Virtual call to C1::f
}
```

Чистые функции
Pure functions

- Чистая функция – это функция, которая не имеет побочных эффектов, и ее возвращаемое значение зависит только от значений аргументов. Это близко соответствует математическому понятию «функция».
- Множественные вызовы чистой функции с одними и теми же аргументами гарантированно дадут один и тот же результат. Компилятор может вынести общие подвыражения, которые содержат вызов чистой функции, и он может вынести за цикл инвариантный код, содержащий вызов чистой функции.
- Некоторые компиляторы знают, что стандартные функции (sqrt, pow, log, …) являются чистыми. Компиляторы gcc и icc позволяют объявить чистыми пользовательские функции:

```
__attribute__((pure)) double Func1(double);

__attribute__((const)) double Func2(double);

double Func3(double x) {
  return Func1(x) * Func1(x) + Func2(x) * Func2(x);
}
```

Реально будет два вызова функций: `Func1(x)` и `Func2(x)`

ЗАВИСИМОСТИ ПО ДАННЫМ (ПО РЕСУРСАМ)

Зависимости по данным

- Зависимости препятствуют оптимизации:
  - Зависимые команды нельзя переупорядочить или выполнить одновременно (при векторизации и распараллеливании компилятором, при выполнении в суперскалярном процессоре)

Зависимости по данным

- Виды зависимостей по данным (по ресурсам)
  - Flow (True) dependence – Read After Write
    - X := …
    - … := f(X)
  - Anti dependence – Write After Read
    - … := f(X)
    - X := …
  - Output dependence – Write After Write
    - X := …
    - X := …
  - Input dependence – Read After Read
    - … := f(X)
    - … := g(X)

Зависимости по данным

- Виды зависимостей по данным (по ресурсам)
  - Flow (True) dependence – Read After Write
    - X := …
    - … := f(X)

    истинная зависимость – нельзя устранить
  - Anti dependence – Write After Read
    - … := f(X)
    - X := …

    можно устранить
  - Output dependence – Write After Write
    - X := …
    - X := …

    можно устранить
  - Input dependence – Read After Read
    - … := f(X)
    - … := g(X)

    не нужно устранять

Зависимости по данным

- Виды зависимостей в цикле
  - Loop-independent – зависимость внутри одной итерации цикла

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] * b[i];
}
```

  - Loop-carried – зависимость между различными итерациями цикла

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] + c[i+1];
}
```

Зависимости по данным

- Способы устранения Anti и Output dependences
  - Variable Renaming
  - Scalar Expansion
  - Node Splitting

Зависимости по данным

- Устранение Anti и Output dependences: **Variable Renaming**

A = X + B
X = Y + 1
C = X + B
X = Z + B
D = X + 1

Зависимость между всеми операциями,
так как используется общая переменная.

Зависимости по данным

- Устранение Anti и Output dependences: **Variable Renaming**

A = X + B	**A = X + B**
X = Y + 1	**X1 = Y + 1**
C = X + B	**C = X1 + B**
X = Z + B	**X2 = Z + B**
D = X + 1	**D = X2 + 1**

Независимые группы операций

- Компилятор и процессор это распознают и применяют.

Зависимости по данным

- Устранение Anti и Output dependences: **Scalar Expansion**

```
for (i=0; i<N; i++) {
  x = a[i] + 1;
  b[i] = x * x;
}
```

Зависимость между всеми итерациями,
так как используется общая переменная.

Зависимости по данным

- Устранение Anti и Output dependences: **Scalar Expansion**

```
for (i=0; i<N; i++) {
  x = a[i] + 1;
  b[i] = x * x;
}
```

```
for (i=0; i<N; i++) {
  x[i] = a[i] + 1;
  b[i] = x[i] * x[i];
}
```

Итерации независимы, так как своя переменная на каждой итерации.

Зависимости по данным

- Устранение Anti и Output dependences: **Scalar Expansion**

```
for (i=0; i<N; i++) {
  x = a[i] + 1;
  b[i] = x * x;
}
```

```
for (i=0; i<N; i++) {
  x[i] = a[i] + 1;
  b[i] = x[i] * x[i];
}
```

```
for (i=0; i<N; i++) {
  float x = a[i] + 1;
  b[i] = x * x;
}
```

- Компилятор это распознаёт и применяет.

Зависимости по данным

- Устранение Anti и Output dependences: **Node Splitting**

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = (a[i] + a[i+1])/2;
}
```

Последовательность операций фиксирована, так как последующие итерации затирают входные данные предыдущих.

Зависимости по данным

- Устранение Anti и Output dependences:
  Node Splitting

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = (a[i] + a[i+1])/2;
}
```

```
for (i=0; i<N; i++) {
  temp[i] = a[i+1];
  a[i] = b[i] + 1;
  c[i] = (a[i] + temp[i])/2;
}
```

Можно выполнять каждую операцию сразу для нескольких итераций (например, при векторизации)

Зависимости по данным

- Устранение Anti и Output dependences: **Node Splitting**

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = (a[i] + a[i+1])/2;
}
```

```
for (i=0; i<N; i++) {
  temp[i] = a[i+1];
  a[i] = b[i] + 1;
  c[i] = (a[i] + temp[i])/2;
}
```

```
for (i=0; i<N; i++) {
  float temp = a[i+1];
  a[i] = b[i] + 1;
  c[i] = (a[i] + temp)/2;
}
```

СРАВНЕНИЕ КОМПИЛЯТОРОВ

Сравнение компиляторов

- Сравнение видов оптимизации, которые делают компиляторы (2019)
  - «Optimizing software in C++» by Agner Fog (раздел 8.2)
  - Компиляторы:
    - Gnu C++ v. 7.4.0 (2019, Cygwin64).
    - Clang C++ v. 5.0.1 (2019, Cygwin64).
    - Microsoft C++ Compiler v. 19.21.27702 (Visual Studio 2019).
    - Intel C++ Compiler v. 19.0.4.245for Intel64, 2019.
  - Результат: наиболее полный список оптимизаций у GNU, Clang

- Сравнение производительности получаемого кода (11.11.2017)
  - «A Performance-Based Comparison of C/C++ Compilers» by Colfax Research
  - Компиляторы:
    - Intel C++ Compiler 18.0.0 (Intel C++ compiler)
    - GNU Compiler Collection 7.2.0 (G++)
    - PGI® C++ Compiler 17.4 (PGC++)
    - Clang 5.0.0 (Clang)
    - Zapcc Compiler 1.0.1 (Zapcc)
    - AMD Optimizing C/C++ Compiler 1.0 (AOCC)
  - Результат: наиболее быстрый код у Intel C++ Compiler

ПРЕПЯТСТВИЯ ДЛЯ ОПТИМИЗАЦИИ КОМПИЛЯТОРОМ

Невозможность оптимизировать между модулями

- Компилятор не имеет информации о функциях, расположенных в других модулях.

<table>
<tr><td>

module1.cpp

```
int Func1(int x) {
  return x*x + 1;
}
```

module2.cpp

```
int Func2() {
  int a = Func1(2);
  ...
}
```

</td><td>Компилятор не может вычислить значение переменной `a` на этапе компиляции</td></tr>
</table>

- Некоторые компиляторы умеют выполнять оптимизацию между модулями при одновременной компиляции нескольких файлов
  - icc -ipo

Перекрытие указателей Pointer aliasing

- При доступе к переменой через указатель или ссылку, компилятор может быть не способен полностью распознать возможность того, что переменная, на которую было указание, идентична некоторой другой переменной в коде.

<table>
<tr>
<td>

```
void Func1 (int a[], int * p) {
  int i;
  for (i = 0; i < 100; i++) {
    a[i] = *p + 2;
  }
}
void Func2() {
  int list[100];
  Func1(list, &list[8]);
}
```

</td>
<td>Компилятор не имеет права считать *p+2 инвариантом цикла</td>
</tr>
</table>

- Как сказать компилятору, что пересечения указателей нет:
  - Специальная опция (для указателей разных типов):
    - gcc -fstrict-aliasing - включается при -O2, -O3, -Os
    - icc -fno-alias / -fansi-alias
  - Ключевое слово __restrict / __restrict__ (если компилятор поддерживает)

Виртуальные функции и указатели на функции

- Если компилятор не может предсказать, какая версия виртуальной функции будет вызвана, или на какую функцию указывает указатель, то он не может сделать подстановку функции или иным способом оптимизировать через вызов функции.

ПРЕПЯТСТВИЯ ДЛЯ ОПТИМИЗАЦИИ ПРОЦЕССОРОМ

Длинные цепочки зависимостей

- Современные процессоры могут выполнять инструкции параллельно вне порядка. Длинные цепочки зависимых команд не позволяют это делать. Нужно их избегать, особенно с долгими операциями, особенно в циклах.

Литература

- Agner Fog, Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms http://www.agner.org/optimize/optimizing_cpp.pdf