###### **МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

###### **ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

###### **НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

###### **Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ** **О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ № 2**

Изучение оптимизирующего компилятора

студента 2 курса, группы 23201

Сорокина Матвея Павловича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.С. Матвеев

Новосибирск 2024

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ** 3](#_Toc174976371)

[**ЗАДАНИЕ** 3](#_Toc174976372)

[**ОПИСАНИЕ РАБОТЫ** 4](#_Toc174976373)

[**ЗАКЛЮЧЕНИЕ** 4](#_Toc174976374)

[**ПРИЛОЖЕНИЯ** 5](#_Toc174976375)

# **ЦЕЛЬ**

1. Изучение основных функций оптимизирующего компилятора, некоторых примеров оптимизирующих преобразований и уровней оптимизации.
2. Получение базовых навыков работы с компилятором GCC.
3. Анализ влияния различных уровней оптимизации компилятора GCC на время выполнения программы.

# **ЗАДАНИЕ**

1. Написать программу на языке C или C++, которая реализует выбранный алгоритм из задания.
2. Проверить правильность работы программы на нескольких тестовых наборах входных данных.
3. Выбрать значение параметра N таким, чтобы время работы программы было порядка 30-60 секунд.
4. Программу скомпилировать компилятором GCC с уровнями оптимизации **-O0, -O1, -O2, -O3, -Os, -Ofast, -Og** под архитектуру процессора x86.
5. Для каждого из семи вариантов компиляции измерить время работы программы при нескольких значениях N.
6. Составить отчет по лабораторной работе.

# **ОПИСАНИЕ РАБОТЫ**

1. Реализовано задание 2 - алгоритм вычисления числа Пи метом Монте-Карло на языке *C++*.

Также предоставлен *bash-скрипт* ***compile\_and\_run.sh***, компилирующий и запускающий программу с конкретными знаениями N, записывающий время вычисления числа Пи с различными уровнями оптимизации компилятора *GCC* в файл *report.csv* (см. Приложение 2), после чего запускает ***create\_table.sh***, визуализирующий информацию из *report.csv* в терминале (см. Приложение 3).

Время замерялось перед началом и после окончания работы функции, вычисляющей число Пи. Разность этих двух значений дает общее время выполнения функции. Для проверки точности измерений, код программы запускается несколько раз.

1. Для измерения времени функции в разделе «СПОСОБЫ ПОЛУЧЕНИЯ ПОКАЗАНИЙ НЕКОТОРЫХ ТАЙМЕРОВ» были выбраны пункты **4.2** и **4.3**
2. *GCC* предоставляет несколько уровней для оптимизации кода, каждый из которых влияет на производительность и размер скомпилированного приложения. Уровень оптимизации - множество используемых оптимизирующих преобразований. В *GCC* есть следующие уровни оптимизации:

* ***-O0***: почти все оптимизации отключены. Компиляция выполняется быстрее, чем на любом другом уровне оптимизации. однако производительность скомпилированной программы может быть значительно ниже.

Данный уровень предпочтителен если вам нужно производить отладку программы или изучать ассемблерный листинг сгенерированного кода, так как получаемый листинг проще в понимании по сравнению с листингами для других уровней.

* ***-O1***: включены оптимизации для уменьшения размера бинарного исполняемого файла и такие оптимизации, уменьшающие время работы программы, которые не сильно замедляют работу компилятора.

Компилятор убирает мертвый код, выполняет простые преобразования выражений, удаляет ненужные операции.

Оптимизации при использовании данного уровня:

* Удаление мертвого кода: *-fdce; -ftree-dce*
* Удаление мёртвых записей: *-fdse*

Пример:

|  |
| --- |
| int main() {  int a = 5;  a = 10; // Мёртвая запись, так как значение переменной 'a' не используется далее  return 0; } |

* Инлайнинг функций, вызываемых один раз: *-finline-functions-called-once*

Пример:

|  |
| --- |
| int func(int x)  {return x \* x; }  int main() {  int result = func(5);  return 0; } |

* Разворачивание циклов: *-funroll-loops*
* ***-O2***: включены практически все доступные оптимизации, кроме тех, что ускоряют вычисления за счет увеличения размера кода.

Включены все оптимизации уровня ***-O1.*** Также используются некоторые дополнительные. Ниже представлены некоторые из них:

* Условное удаление мёртвого кода: *-ftree-builtin-call-dce*

Пример:

|  |
| --- |
| int example() {  double x = sqrt(-1); // sqrt может установить errno  return 0; } |

* Инлайнинг функций: *-finline-functions; -finline-small-functions*
* Косвенный инлайнинг: *-findirect-inlining*
* Перепрыгивание переходов: *-fcrossjumping*
* ***-O3***: включены все оптимизации из предыдущего уровня, добавлены оптимизации времени работы программы, которые могут приводить к увеличению размера бинарного исполняемого файла (агрессивные оптимизации).

Включены все оптимизации уровня ***-O2.*** Также используются следующие дополнительные:

|  |
| --- |
| -fgcse-after-reload  -fipa-cp-clone  -floop-interchange  -floop-unroll-and-jam  -fpeel-loops  -fpredictive-commoning  -fsplit-loops  -fsplit-paths  -ftree-loop-distribution  -ftree-partial-pre  -funswitch-loops  -fvect-cost-model=dynamic  -fversion-loops-for-strides |

* ***-Os***: оптимизация размера программ.

Включены все оптимизации уровня ***-O2,*** ноне включены флаги, увеличивающие размер кода:

|  |
| --- |
| -falign-functions  -falign-jumps  -falign-labels  -falign-loops  -fprefetch-loop-arrays  -freorder-blocks-algorithm=stc |

Описание вышеперечисленных флагов компиляции:

*-falign-functions:* выравнивает начала функций в коде на определённые границы (например, на границу в 16 байт) с целью улучшения производительности программы.

*-falign-jumps:* выравнивание переходов (jumps) в машинном коде на определенные границы. (*jmp, call, ret*)

*-falign-labels:* выравнивания инструкций на определенные границы памяти.

Также включает в себя *-finline-functions*, заставляя компилятор оптимизировать код в пользу уменьшения его размера, а не скорости выполнения.

* ***-Ofast***: включает все оптимизации уровня -***O3***, а также ряд других, таких как использование более быстрых и менее точных математических функций.

Пример: флаг *-ffast-math* используется для агрессивных оптимизаций арифметики с плавающей точкой, которые могут улучшить производительность, но при этом могут нарушить точность и соответствие стандартам IEEE для арифметики с плавающей точкой.

При использовании данного флага включаются следующие параметры:

* *-fno-math-errno:* отключает установку глобальной переменной errno после выполнения математических функций.
* *-funsafe-math-optimizations*: включает небезопасные оптимизации математических операций.
* *-ffinite-math-only:* математические функции работают только с конечными числами.
* *-fno-rounding-math:* отключает поддержку округления чисел.
* *-fno-signaling-nans*: Отключает поддержку сигнализирующих NaN.
* *-fcx-limited-range:* оптимизация операций с плавающей точкой, предполагая, что числа в вычислениях находятся в ограниченном диапазоне.
* *-fexcess-precision=fast:* использование избыточной точности в вычислениях с плавающей запятой. fast означает, что избыточная точность используется только там, где это может улучшить скорость работы программы.
* ***-Og***: производит все оптимизации, которые сохраняют возможность просмотра стека вызовов, фрагментов исходного текста программы, относящихся к разным уровням этого стека, и возможность приостановки программы для каждой строки исходного текста, содержащей операторы.

Позволяет компилятору выполнять оптимизации, которые не мешают отладке, сохраняя при этом полезную информацию для отладчиков.

Включены все оптимизации уровня ***-O1***, за исключением тех, которые могут мешать отладке:

|  |
| --- |
| -fbranch-count-reg  -fdelayed-branch  -fdse  -fif-conversion  -fif-conversion2  -finline-functions-called-once  -fmove-loop-invariants  -fmove-loop-stores  -fssa-phiopt  -ftree-bit-ccp  -ftree-dse  -ftree-pta  -ftree-sra |

1. В данной лабораторной работе вместо вычисления среднего значения времени использовалось минимальное. Этому способствует ряд причин:
   1. **Среднее время может искажать реальную картину**: среднее время включает как минимальные, так и случайные задержки, вызванные внешними факторами. Это делает среднее значение менее надежным, так как оно отражает не только работу программы, но и случайные сбои в системе.
   2. **Минимальное время лучше для оптимизаций**: когда речь идет об оптимизации, минимальное время выполнения указывает на потенциальную производительность программы. К этому значению следует стремиться, оптимизируя код и минимизируя зависимости от внешних факторов.

# **ЗАКЛЮЧЕНИЕ**

В ходе данной лабораторной работы мы познакомились с различными методами измерения работы программ и научились пользоваться ими на практике.

# **ПРИЛОЖЕНИЯ**

**Приложение 1:Исходный код программы *PiCalculation.cpp с библиотечной функцией times***

|  |
| --- |
| #include <iostream>  #include <time.h>  #include <cstdlib> // for atoi and atof  long double PiCalculation(long long n) {      long long m = 0;      srand(time(NULL));      // srand(static\_cast<unsigned int>(time(0))); // устанавливаем значение системных часов в качестве стартового числа      for (long long i = 0; i < n; i++) {          // double x = static\_cast<double>(rand()) / RAND\_MAX // диапозон [0;1]          // RAND\_MAX \* 2.0 // увеличиваем диапозон до [0; 2]          // RAND\_MAX \* 2.0 - 1.0 // сдвигаем диапозон до [-1; 1]          double x = static\_cast<double>(rand()) / RAND\_MAX \* 2.0 - 1.0;          double y = static\_cast<double>(rand()) / RAND\_MAX \* 2.0 - 1.0;          if (x \* x + y \* y <= 1.0) {              m++;          }      }      return 4.0 \* m / n;  }  int main(int argc, char \*argv[]) {      if (argc != 2) {          std::cerr << "Invalid number of arguments" << std::endl;          return 0;      }      struct timespec start, end;      long long n = atoll(argv[1]);      std::cout << "N = " << n << std::endl;      int runs = 5;      double min\_time = 1e3;      for (long long i = 0; i < runs; i++) {          clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);          double pi = PiCalculation(n);          clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);          double taken\_time = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;          std::cout << "pi = " << pi << std::endl;          std::cout << "Run time: " << taken\_time << "\n" << std::endl;          min\_time = std::min(min\_time, taken\_time);      }      std::cout << "Min time: " << min\_time << " seconds" << std::endl;      return 0;  } |

**Приложение 2: bash-скрипт *compile\_and\_run.sh* для компиляции и запуска программы, записи результата в файл *report.csv***

|  |
| --- |
| #!/bin/bash  input="PiCalculation.cpp"  optimization\_keys=("-O0" "-O1" "-O2" "-O3" "-Os" "-Ofast" "-Og")  n="750000000"  echo "Optimization level, N value, Time taken (seconds)" > report.csv  for i in "${optimization\_keys[@]}"; do      g++ $input $i -std=c++11      output=$(./a.out $n)      # каждая итерация      echo "$output" | grep "Run time:" | awk -v opt="$i" -v N="$n" '{print opt ", " N ", " $3}' >> report.csv      # минимальное время, не среднее!      min\_time=$(echo "$output" | grep "Min time:" | awk '{print $3}')      echo "$i, $n, $min\_time" >> report.csv  done  echo "Successfully generated report.csv"  ./create\_table.sh |

**Приложение 3: bash-скрипт *create\_table.sh* для вывода в табличном формате данных из файла *report.csv***

|  |
| --- |
| #!/bin/bash  if [ ! -f report.csv ]; then      echo "Файл report.csv не найден!"      exit 1  fi  # заголовок таблицы  echo -e "Оптимизация\t N\t\t Время выполнения (сек)"  prev=""  # -F',': запятая в качестве разделителя столбцов  # NR>1: пропускаем первую строку  awk -F',' 'NR>1 {      if (prev != $1 && NR > 2) {          print ""; # перевод на новую строку      }      printf "%s\t\t%s\t%s\n", $1, $2, $3;      prev = $1;  }' report.csv |

**Приложение 4: Таблица с результатами измерений**

|  |  |  |
| --- | --- | --- |
| Оптимизация | Значение N | Время выполнения (с) |
| -O0 | 750000000 | 16.55 |
| -O1 | 750000000 | 15.97 |
| -O2 | 750000000 | 15.87 |
| -O3 | 750000000 | 15.89 |
| -Os | 750000000 | 15.74 |
| -Ofast | 750000000 | 15.61 |
| -Og | 750000000 | 16.05 |

**Приложение 5: График зависимости времени от оптимизации компиляции**

A graph with a line going up

Description automatically generated

**Приложение 6: Скрипт для визуализации графика зависимости времени от зависимости компиляции на языке Python**

|  |
| --- |
| import matplotlib.pyplot as plt  n\_value = 750000000  times\_o0 = 16.55  times\_o1 = 15.97  times\_o2 = 15.87  times\_o3 = 15.89  times\_os = 15.74  times\_ofast = 15.61  times\_og = 16.05  # уровни оптимизации  optimization\_levels = ['-O0', '-O1', '-O2', '-O3', '-Os', '-Ofast', '-Og']  times = [times\_o0, times\_o1, times\_o2, times\_o3, times\_os, times\_ofast, times\_og]  plt.figure(figsize=(10, 5))  # график  plt.plot(optimization\_levels, times, 'o-', color='b', label=f'N = {n\_value}')  plt.xlabel('Optimization Levels')  plt.ylabel('Time taken (seconds)')  plt.legend()  plt.grid(True)  plt.show()  # plt.savefig('plot.png') |