**Лабораторная работа № 4.**

**Оптимизация доступа к памяти.**

1.На языке С/С++/C# реализовать функцию DGEMM BLAS последовательное умножение двух квадратных матриц с элементами типа double.

Обеспечить возможность задавать размерности матриц в качестве аргумента командной строки при запуске программы. Инициализировать начальные значения матриц случайными числами.

2.Провести серию испытаний и построить график зависимости времени выполнения программы от объёма входных данных.Например, для квадратных матриц с числом строк/столбцов 1000, 2000, 3000, ... 10000.

3.Оценить предельные размеры матриц, которые можно перемножить на вашем вычислительном устройстве.

4.Реализовать дополнительную функциюDGEMM\_opt\_1,в которой выполняется оптимизация доступа к памяти, за счет построчного перебора элементов обеих матриц.

5.\* Реализовать дополнительную функцию DGEMM\_opt\_2, в которой выполняется оптимизация доступа к памяти, за счет блочного перебора элементов матриц. Обеспечить возможность задавать блока, в качестве аргумента функции.

6.\*\* Реализовать дополнительную функцию DGEMM\_opt\_3, в которой выполняется оптимизация доступа к памяти, за счет векторизации кода.

7.Оценить ускорение умножения для матриц фиксированного размера, например, 1000х1000, 2000х2000, 5000х5000, 10000х10000.\* Для блочного умножения матриц определить размер блока, при котором достигается максимальное ускорение.

8.С помощью профилировщика для исходной программы и каждого способа оптимизации доступа к памяти оценить количество промахов при работе к КЭШ памятью (cache-misses).

9.Подготовить отчет отражающий суть, этапы и результаты проделанной работы.

Оглавление

[1. BLAS (Basic Linear Algebra Subroutines) 2](#_Toc57896385)

[2. Измерение времени выполнения кода 14](#_Toc57896386)

[**Возможные ошибки** 16](#_Toc57896387)

[**Примечания** 16](#_Toc57896388)

[3. Определение предельных размеров матриц, которые возможно перемножить на устройстве. 18](#_Toc57896389)

[4. Оптимизация 18](#_Toc57896390)

[5.Функция DGEMM\_opt\_2 20](#_Toc57896391)

[6.Функция DGEMM\_opt\_3 22](#_Toc57896392)

[7.Оценка ускорения умножения матриц фиксированного размера 23](#_Toc57896393)

[8. Профилирование – сбор характеристик работы программы 26](#_Toc57896394)

# 1. BLAS (Basic Linear Algebra Subroutines)

BLAS (англ. Basic Linear Algebra Subprograms) - это базовые подпрограммы линейной алгебры, которые предписывают набор низкоуровневых подпрограмм для выполнения общих операций линейной алгебры, таких как сложение векторов , скалярное умножение , скалярные произведения , линейные комбинации и умножение матриц . Это стандартные низкоуровневые процедуры для библиотек линейной алгебры; подпрограммы имеют привязки как для C («интерфейс CBLAS»), так и для Fortran («интерфейс BLAS»). Хотя спецификация BLAS является общей, реализации BLAS часто оптимизированы для скорости на конкретном компьютере, поэтому их использование может принести существенный выигрыш в производительности. Реализации BLAS будут использовать преимущества специального оборудования с плавающей запятой, такого как векторные регистры или инструкции SIMD.

Она возникла как библиотека Fortran в 1979 году, а ее интерфейс был стандартизирован Форумом BLAS Technical (BLAST), последний отчет которого можно найти на веб-сайте netlib.

Большинство библиотек, предлагающих процедуры линейной алгебры, соответствуют интерфейсу BLAS, что позволяет пользователям библиотек разрабатывать программы, безразличные к используемой библиотеке BLAS. Примеры библиотек BLAS включают: AMD Core Math Library (ACML), Arm Performance Libraries, ATLAS , Intel Math Kernel Library (MKL) и OpenBLAS . ACML больше не поддерживается его производителем. ATLAS - это портативная библиотека, которая автоматически оптимизируется для произвольной архитектуры. MKL - это бесплатная и проприетарная библиотека поставщика, оптимизированная для x86 и x86-64 с упором на производительность процессоров Intel. OpenBLAS - это библиотека с открытым исходным кодом, оптимизированная вручную для многих популярных архитектур. В Linpack ориентиры в значительной мере полагаются на рутину BLAS gemm для его измерения производительности.

Многие численные программные приложения используют BLAS-совместимые библиотеки для выполнения вычислений линейной алгебры, включая Armadillo , LAPACK, LINPACK , GNU Octave , Mathematica , MATLAB , NumPy , R и Julia

Ниже приведен пример процедуры xGEMM из библиотеки BLAS в программе на C.

Процедура xGEMM вычисляет следующее выражение: С = αAB + βC, где:

A- матрица M\*K

B- матрица K\*N

C -матрица M\*N

α, β - коэффициенты.

Первая буква в названии функции определяет тип данных элементов матрицы:

S - вещественный с одинарной точностью (например, SGEMM),

D - вещественный с двойной точностью (например, DGEMM),

С - комплексный с одинарной точностью (например, CGEMM),

Z - комплексный с двойной точностью (например, ZGEMM).

Для использования процедур из библиотеки BLAS, необходимо, чтобы массивы в памяти хранились одним непрерывным блоком. Например, массив A из следующего примера нельзя передавать в качестве параметра в BLAS-процедуру:

|  |
| --- |
| float \*\*A=(float\*\*)malloc(M\*sizeof(float\*));  for (i=0;i<M;i++) A[i]=(float\*)malloc(K\*sizeof(float)); |

В нем не гарантировано, что строки будут следовать одна за другой. Следующий массив B использовать можно:

|  |
| --- |
| float \*\*B=(float\*\*)malloc(M\*sizeof(float\*));  float \*B1=(float\*)malloc(M\*K\*sizeof(float))  for (i=0;i<M;i++) B[i]=&B1[i\*K] |

Прототип функции SGEMM в интерфейсе языка C выглядит следующим образом:

|  |
| --- |
| void cblas\_sgemm (const enum CBLAS\_ORDER Order,  const enum CBLAS\_TRANSPOSE TransA,  const enum CBLAS\_TRANSPOSE TransB,  const int M, const int N,  const int K, const float alpha, const float \*A,  const int lda, const float \*B, const int ldb,  const float beta, float \*C, const int ldc); |

где: Order - определяет порядок следования элементов:

CblasRowMajor - матрицы хранятся по строкам (стандартно в C),

CblasColMajor - матрицы хранятся по столбцам;

TransA, TransB - определяет предварительные операции над матрицами A и B:

CblasNoTrans - ничего не делать,

CblasTrans - транспонировать,

CblasConjTrans - вычислить сопряженную матрицу;

M, N, K - размеры матриц;

alpha, beta - коэффициенты;

lda, ldb, ldc - число элементов в ведущей размерности матрицы (строке или столбце). Для массивов языка Си - число элементов в строке:

lda = K

ldb = N

ldc = M

Для использования процедур из библиотеки BLAS, необходимо, чтобы массивы в памяти хранились одним непрерывным блоком. Например, массив A из следующего примера нельзя передавать в качестве параметра в BLAS-процедуру:

|  |
| --- |
| float \*\*A=(float\*\*)malloc(M\*sizeof(float\*));  for (i=0;i<M;i++) A[i]=(float\*)malloc(K\*sizeof(float)); |

В нем не гарантировано, что строки будут следовать одна за другой.

Следующий массив B использовать можно:

|  |
| --- |
| float \*\*B=(float\*\*)malloc(M\*sizeof(float\*));  float \*B1=(float\*)malloc(M\*K\*sizeof(float))  for (i=0;i<M;i++) B[i]=&B1[i\*K]; |

Двумерный массив — это массив массивов, поэтому он хранится в памяти следующим образом:

|  |
| --- |
| char v[2][3] = {{1,3,5},{5,10,2}};  Content: | 1 | 3 | 5 | 5 | 10 | 2  Address: v v+1 v+2 v+3 v+4 v+5 |

Чтобы получить доступ к v [x] [y], компилятор переписывает его как: \*(v + y \* M + x) (где М — второе заданное измерение)

Например, чтобы получить доступ к v [1] [1], компилятор переписывает его как \*(v + 1\*3 + 1) => \*(v + 4)

Помните, что это не такой же как указатель на указатель (char \*\*).

Указатель на указатель не является массивом: он содержит и адрес ячейки памяти, которая содержит другой адрес.

Чтобы получить доступ к члену двумерного массива, используя указатель на указатель, вот что сделано:

|  |
| --- |
| char \*\*p;  /\* Initialize it \*/  char c = p[3][5]; |

1. Перейти по адресу, указанному в содержании p;
2. Добавьте смещение к этому адресу (3 в нашем случае);
3. Перейдите по этому адресу и получите его содержимое (наш новый адрес).
4. Добавьте второе смещение к этому новому адресу (5 в нашем случае).
5. Получить содержание этого адреса.

В то время как для доступа к члену через традиционный 2-мерный массив, это шаги:

|  |
| --- |
| char p[10][10];  char c = p[3][5]; |

1. Получить адрес pи суммируем первое смещение (3), умноженное на размерность строки (10).
2. Добавьте второе смещение (5) к результату.
3. Получить содержание этого адреса.

Библиотеку ATLAS (реализацию BLAS) можно скачать отсюда: atlas3.6.0\_Linux\_P4SSE2.tgz. Файл можно скачать программой wget и распаковать в домашний каталог (появится каталог Linux\_P4SSE2).

При компиляции программы необходимо будет указать пути к каталогам include и lib и прилинковать необходимые библиотеки libcblas.a и libatlas.a

**Пример команды компиляции:**

gcc -I $HOME/Linux\_P4SSE2/include -L $HOME/Linux\_P4SSE2/lib -O3 prog.c -lcblas -latlas

**Базовые подпрограммы линейной алгебры имеют следующие реализации:**

**refblas**

Официальная эталонная реализация из netlib. Доступны версии на C и Fortran 77.

**Accelerate**

Фреймворк от Apple для Mac OS X, включающий оптимизированные версии BLAS и LAPACK для процессоров PowerPC и Intel Core.

**ACML**

Основная математическая библиотека AMD, поддерживающая процессоры AMD Athlon и Opteron под Linux и Windows.

**ATLAS**

Самооптимизирующийся программный пакет линейной алгебры (англ. Automatically Tuned Linear Algebra Software), реализация интерфейса BLAS с открытым исходным кодом для C и Fortran 77.

**CUDA SDK**

NVIDIA CUDA SDK включает функциональность BLAS (cuBLAS) для написания программ на C для видеокарт серии GeForce 8, GeForce 200, GeForce 300 (Fermi).

**ESSL**

Библиотека инженерных и научных подпрограмм (англ. Engineering and Scientific Subroutine Library) от IBM, поддерживающая архитектуру PowerPC под AIX и Linux.

**libflame**

Реализация библиотеки линейной алгебры, включающей BLAS, проектом FLAME.

**Goto BLAS**

Реализация Кадзусигэ Гото.

**HP MLIB**

Математическая библиотека от HP, поддерживающая архитектуры IA-64, PA-RISC, x86 и Opteron под HP-UX и Linux.

**Intel MKL**

Основная математическая библиотека Intel (англ. Intel Math Kernel Library), поддерживающая процессоры Intel под Linux, Windows и Mac OS X.

**MathKeisan**

Математическая библиотека от NEC, поддерживающая архитектуру NEC SX под SUPER-UX, и Itanium под Linux.

**PDLIB/SX**

Математическая библиотека, находящаяся в общественном достоянии (англ. Public Domain Mathematical Library), от NEC для системы NEC SX-4.

**SCSL**

Программная библиотека для научных вычислений (англ. Scientific Computing Software Library) от SGI содержит реализации BLAS и LAPACK для рабочих станций SGI Irix.

**Sun Performance Linaray**

Sun Performance Library содержит оптимизированные BLAS и LAPACK для архитектур SPARC и AMD64 под Solaris 8, 9, и 10.

**uBLAS**

Библиотека шаблонных классов C++, обеспечивающая функциональность BLAS. Часть библиотеки Boost. В отличие от других реализаций uBLAS фокусируется больше на правильности алгоритмов, используя продвинутые возможности C++, чем на высокой производительности.

**GSL**

Научная библиотека GNU (англ. GNU Scientific Library) содержит кроссплатформенную неоптимизированную реализацию на C, которая распространяется под GNU GPL

Подобную функцию можно реализовать собственноручно в упрощенном виде, так как весь ее функционал нам не требуется:

|  |
| --- |
| void DGEMM\_BLAS(double\*\* X, double\*\* Y, double\*\* R, int N) {  /\*цикл последовательного умножения элементов строки первой матрицы на элементы столбца второй матрицы\*/  for (int i = 0; i < N; i++)  for (int j = 0; j < N; j++) {  for (int k = 0; k < N; k++)  R[i][j] += X[i][k] \* Y[k][j];  }  } |

Функция DGEMM\_BLAS, поэлементно умножающая матрицы (оптимизации нет). Гистограмма работы функции DGEMM\_BLAS:

Рисунок 1 Гистограмма результатов работы DGEMM\_BLAS

**Заполнение массива значениями:**

При решении учебных задач, обычно предлагается заполнить массив значениями либо введёнными с клавиатуры, либо случайными значениями из определённого диапазона. Начнём со второго случая, как более простого.

**Заполнение массива случайными числами:**

Для начала необходим генератор случайных чисел. Ниже приведён код одной из простейших реализаций:

|  |
| --- |
| #include <cstdlib>  using namespace std;  // функция генерации случайного числа из диапазона от range\_min до range\_max включительно  int rrand(int range\_min, int range\_max) {  return rand() % (range\_max - range\_min + 1) + range\_min;  } |

Однако без дополнительных телодвижений стандартная функция rand() будет при каждом запуске программы генерировать одинаковую последовательность случайных чисел (кстати, это очень удобно при отладке!). Для того, что бы при каждом запуске программы получать уникальную последовательность случайных чисел, функцию rand() надо «разогнать» начальным случайным значением. Это делается с помощью функций srand() и time().

Заполнение массива значениями, естественно, делаем в цикле. Помним, что элементы массива в C/C++ нумеруются с 0. Следовательно последний элемент массива имеет индекс на единицу меньший, чем размер массива.

В примере показано заполнение глобального автоматического массива из 10 элементов типа int случайными значения из диапазона от −100 до 100 включительно:

|  |
| --- |
| #include <cstdlib>  #include <time.h>  using namespace std;  // функция генерации случайного числа из диапазона от range\_min до range\_max включительно  int rrand(int range\_min, int range\_max) {  return rand() % (range\_max - range\_min + 1) + range\_min;  }  const unsigned int ARRSIZE = 10;  const int ABSLIMIT = 100;  int ary[ARRSIZE];  int main(void) {  srand(static\_cast<unsigned int>(time(NULL)));  // инициализация массива случайными значениями в диапазоне от -ABSLIMIT до ABSLIMIT  for (unsigned int i = 0; i < ARRSIZE; i++) {  ary[i] = rrand(-ABSLIMIT, ABSLIMIT);  }  return 0;  } |

**Хранение массивов в памяти**

Очень часто возникают задачи обработки массивов данных, размерность которых заранее неизвестна. В этом случае возможно использование одного из двух подходов:

1. выделение памяти под статический массив, содержащий максимально возможное число элементов, однако в этом случае память расходуется не рационально;

2. динамическое выделение памяти для хранение массива данных.

Для использования функций динамического выделения памяти необходимо описать указатель, представляющий собой начальный адрес хранения элементов массива.

int \*p; // указатель на тип int

Начальный адрес статического массива определяется компилятором в момент его объявления и не может быть изменен.

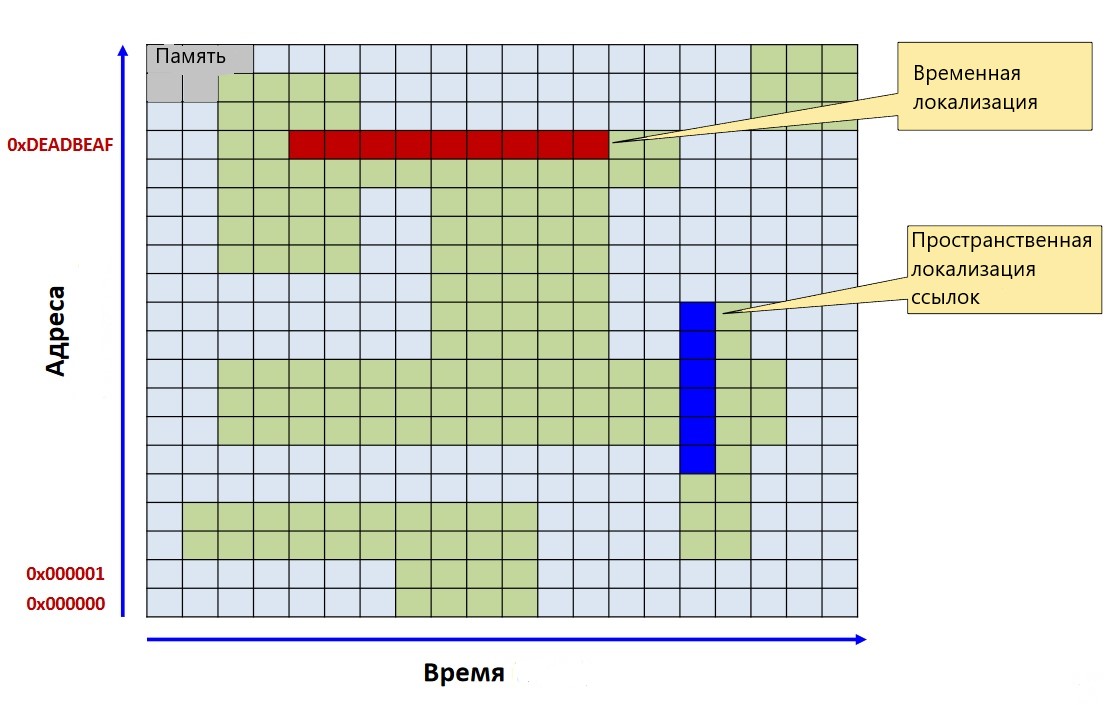
Для динамического массива начальный адрес присваивается объявленному указателю на массив в процессе выполнения программы.

**Локальность ссылок** – свойство программ повторно (часто) обращаться к одним и тем же адресам в памяти (данным, инструкциям)

**Формы локальности ссылок:**

Временная локализация – повторное обращение к одному и тому же адресу через короткий промежуток времени (например, в цикле).

Пространственная локализация ссылок– свойство программ повторно обращаться через короткий промежуток времени к адресам близкорасположенным в памяти друг к другу



**Стандартные функции динамического выделения памяти**

Функции динамического выделения памяти находят в оперативной памяти непрерывный участок требуемой длины и возвращают начальный адрес этого участка.

Функции динамического распределения памяти:

void\* malloc(РазмерМассиваВБайтах);

void\* calloc(ЧислоЭлементов, РазмерЭлементаВБайтах);

Для использования функций динамического распределения памяти необходимо подключение библиотеки <malloc.h>:

#include <malloc.h>

Поскольку обе представленные функции в качестве возвращаемого значения имеют указатель на пустой тип void, требуется явное приведение типа возвращаемого значения.

Для определения размера массива в байтах, используемого в качестве аргумента функции malloc() требуется количество элементов умножить на размер одного элемента. Поскольку элементами массива могут быть как данные простых типов, так и составных типов (например, структуры), для точного определения размера элемента в общем случае рекомендуется использование функции

int sizeof(тип);

которая определяет количество байт, занимаемое элементом указанного типа.

Память, динамически выделенная с использованием функций calloc(), malloc(), может быть освобождена с использованием функции

free(указатель);

«Правилом хорошего тона» в программировании является освобождение динамически выделенной памяти в случае отсутствия ее дальнейшего использования. Однако если динамически выделенная память не освобождается явным образом, она будет освобождена по завершении выполнения программы.

Динамическое выделение памяти для одномерных массивов

Форма обращения к элементам массива с помощью указателей имеет следующий вид:

int a[10], \*p; // описываем статический массив и указатель

int b;

p = a; // присваиваем указателю начальный адрес массива

... // ввод элементов массива

b = \*p; // b = a[0];

b = \*(p+i) // b = a[i];

Динамическое выделение памяти для двумерных массивов

Пусть требуется разместить в динамической памяти матрицу, содержащую n строк и m столбцов. Двумерная матрица будет располагаться в оперативной памяти в форме ленты, состоящей из элементов строк. При этом индекс любого элемента двумерной матрицы можно получить по формуле

index = i\*m+j;

где i - номер текущей строки; j - номер текущего столбца.

Обращение к элементу с использованием указателя будет выглядеть как

\*(p+i\*m+j),

где

p - указатель на массив,

m - количество столбцов,

i - индекс строки,

j - индекс столбца.

Возможен также другой способ динамического выделения памяти под двумерный массив - с использованием массива указателей. Для этого необходимо:

· выделить блок оперативной памяти под массив указателей;

· выделить блоки оперативной памяти под одномерные массивы, представляющие собой строки искомой матрицы;

· записать адреса строк в массив указателей.

При таком способе выделения памяти компилятору явно указано количество строк и количество столбцов в массиве.

С помощью динамического выделения памяти под указатели строк можно размещать свободные массивы. Свободным называется двухмерный массив (матрица), размер строк которого может быть различным. Преимущество использования свободного массива заключается в том, что не требуется отводить память компьютера с запасом для размещения строки максимально возможной длины. Фактически свободный массив представляет собой одномерный массив указателей на одномерные массивы данных.

Для размещения в оперативной памяти матрицы со строками разной длины необходимо ввести дополнительный массив m, в котором будут храниться размеры строк.

**Перераспределение памяти**

Если размер выделяемой памяти нельзя задать заранее, например при вводе последовательности значений до определенной команды, то для увеличения размера массива при вводе следующего значения необходимо выполнить следующие действия:

Выделить блок памяти размерности n+1 (на 1 больше текущего размера массива)

Скопировать все значения, хранящиеся в массиве во вновь выделенную область памяти

Освободить память, выделенную ранее для хранения массива

Переместить указатель начала массива на начало вновь выделенной области памяти

Дополнить массив последним введенным значением

Все перечисленные выше действия (кроме последнего) выполняет функция

void\* realloc (void\* ptr, size\_t size);

ptr - указатель на блок ранее выделенной памяти функциями malloc(), calloc() или realloc() для перемещения в новое место. Если этот параметр равен NULL, то выделяется новый блок, и функция возвращает на него указатель.

size - новый размер, в байтах, выделяемого блока памяти. Если size = 0, ранее выделенная память освобождается и функция возвращает нулевой указатель, ptr устанавливается в NULL.

Размер блока памяти, на который ссылается параметр ptr изменяется на size байтов. Блок памяти может уменьшаться или увеличиваться в размере. Содержимое блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Но отбрасываются те данные, которые выходят за рамки нового блока. Если новый блок памяти больше старого, то содержимое вновь выделенной памяти будет неопределенным.

# 2. Измерение времени выполнения кода

Измерение времени выполнения кода (GNU/Linux), типы функций:

* int gettimeofday(struct timeval \*tv, struct timezone \*tz)
* int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp)
* clock\_t clock(void)
* time\_t time(time\_t \*t)
* CPU Time Stamp Counter (TSC, 64-bit MSR register)

Функция **gettimeofday**()

|  |
| --- |
| #include <sys/time.h>  int main()  {  struct timeval t1, t2;  double t;  gettimeofday(&t1, NULL);  /\* какой-либо код \*/  gettimeofday(&t2, NULL);  t = (t2.tv\_sec \* 1E6 + t2.tv\_usec –  t1.tv\_sec \* 1E6 - t1.tv\_usec) \* 1E-6;  printf("Elapsed time: %.6f sec.\n", t);  return 0;  } |

Функции **gettimeofday**() и **settimeofday**() позволяют получить и задать время, а также часовой пояс. Аргумент tv имеет тип struct timeval (определён в *<*sys/time.h*>*):

|  |
| --- |
| struct timeval {  time\_t tv\_sec; /\* секунды \*/  suseconds\_t tv\_usec; /\* микросекунды \*/  }; |

и задаёт количество секунд и микросекунд, прошедших с начала эпохи. Аргумент tz имеет тип struct timezone:

|  |
| --- |
| struct timezone {  int tz\_minuteswest; /\* количество минут западнее Гринвича \*/  int tz\_dsttime; /\* тип корректировки DST \*/  }; |

Если *tv* или *tz* имеют значение NULL, то соответствующая структура не задана и не будет возвращена (однако при компиляции будут выдаваться предупреждения, если *tv* равно NULL).

Пользоваться структурой timezone не рекомендуется; значением аргумента tz в большинстве случаев должен быть NULL (см. Примечания далее).

В Linux существует специфическое понятие «временной сдвиг» (warp clock), связанное с системным вызовом **settimeofday**(), который образуется при самом первом вызове (после загрузки), если аргумент *tz* не равен NULL, аргумент tv равен NULL и поле tz\_minuteswest не равно нулю (в этом случае значение поля tz\_dsttime должно быть равно нулю). В этом случае предполагается, что время аппаратных часов (CMOS clock) местное и к нему должен быть добавлен этот параметр для того, чтобы получилось время UTC. Но, как мы и говорили, использовать этот метод не рекомендуется.

При успешном выполнении **gettimeofday**() и **settimeofday**() возвращают 0, или -1 в случае ошибки (errno устанавливается в соответствующее значение).

**Возможные ошибки**

**EFAULT**

Одно из значений в *tv* или *tz* находится вне доступного адресного пространства.

**EINVAL**

Неправильный часовой пояс (или что-то ещё).

**EPERM**

У вызывающего процесса недостаточно прав для вызова **settimeofday**(); в Linux для этого требуется мандат **CAP\_SYS\_TIME**.

**Примечания**

На время, возвращаемое **gettimeofday**(), влияют скачки в системном времени (например, если системный администратор вручную изменил системное время). Если вам требуются однообразно увеличивающееся время, смотрите **clock\_gettime**.

Для работы со структурой timeval существуют макросы, описанные в **timeradd**.

Обычно, поля struct timeval имеют тип long. Далее рассмотрим методику измерения времени выполнения программы.

**Методика измерения времени выполнения кода**

1. Готовим систему к проведению измерений

настраиваем аппаратные подсистемы (настройки BIOS)

параметры операционной системы и процесса, в котором будут осуществляться измерения

2. Выполняем разогревочный вызов измеряемого кода (Warmup)

Регистрируем время выполнения первого вызова и не учитываем его в общей статистике

(при первом вызове может осуществляться отложенная инициализация и пр.)

3. Выполняем многократные запуски и собираем статистику о времени выполнения

Каждый запуск должен осуществляться в одних и тех же условиях

(входные массивы заполнены одними и теми же данными,

входные данные отсутствуют/присутствуют в кеш-памяти процессора, …)

Выполняем измерения пока:

✓ относительная стандартная ошибка среднего времени выполнения (RSE) больше 5%

[опционально]

✓ число выполненных измерений меньше максимально допустимого

4. Проводим статистическую обработку результатов измерений

Находим и отбрасываем промахи измерений (выбросы, outliers):

например, 25% минимальных и максимальных значений результатов измерений [опционально]

Вычисляем оценку математического ожидания времени выполнения (mean)

(медиану [опционально])

Вычисляем несмещенную оценку дисперсии времени выполнения

(unbiased sample variance - Var)

Вычисляем стандартное отклонение (corrected sample standard deviation - StdDev)

Вычисляем стандартную ошибку среднего времени выполнения

(standard error of the mean - StdErr)

Вычисляем относительную стандартную ошибку среднего времени выполнения

(relative standard error of the mean - RSE)

Строим доверительные интервалы (confidence interval)

**Элементарная обработка результатов измерений**

■ Математическое ожидание времени выполнения (Mean)

■ Несмещенная оценка дисперсии времени выполнения (Unbiased sample variance - Var)

■ Стандартное отклонение (Corrected sample standard deviation - StdDev)|

■ Стандартная ошибка среднего времени выполнения (Standard error of the mean - StdErr)

■ Относительная стандартная ошибка среднего времени выполнения

(relative standard error of the mean - RSE)

■ RSE показывает на сколько близко вычисленное среднее время выполнения

к истинному среднему времени выполнения (среднему генеральной совокупности)

o На практике хорошая точность RSE <= 5%

■ Оценка погрешности при большом числе измерений

o С вероятностью 0.997 время выполнения лежит в интервале

o С вероятностью 0.955 время выполнения лежит в интервале

o С вероятностью 0.683 время выполнения лежит в интервале

■ Оценка погрешности при малом числе измерений (n < 10)

o Задаем требуемый уровень α доверительной вероятности (0.95; 0.99), из таблицы берем значение коэффициента Стьюдента.

# 3. Определение предельных размеров матриц, которые возможно перемножить на устройстве.

Определить предельный размер матриц, которые можно перемножить на нашем устройстве возможно двумя способами:

1. Это определение огромной разницы во времени вычисления для смежных размеров матриц(например если про размере 500х500 - 5 секунд, а при 1000х1000 - 50 секунд, то оптимален 500х500, либо вычисляем время для 700х700 и смотрим далее на результаты) .
2. Это определение оптимального размера оперативной памяти устройства, в который поместятся исходные и результирующая матрицы(приблизительно 80% от общей ОП).

# 4. Оптимизация

Оптимизация – модификация системы для улучшения её эффективности. Система может быть одиночной компьютерной программой, цифровым устройством, набором компьютеров или даже целой сетью, такой как Интернет.

**Цели оптимизации программного обеспечения:**

* Минимизация времени выполнения
* Минимизация объема потребляемой памяти
* Минимизация энергопотребления процессором/вычислительным узлом

**Этапы оптимизации кода**

1)Алгоритмическая оптимизация кода

2)Распараллеливание – Thread Level Parallelism (OpenMP/Pthreads, CUDA/OpenCL/OpenACC, MPI)

3)Оптимизация доступа к памяти (кеш-памяти, ОЗУ, дисковой, сетевым хранилищам)

4)Векторизация кода – Data Parallelism (SSE, AVX, AltiVec, ARM SIMD)

5)Оптимизация ветвлений, оптимизация загрузки конвейеров – Instruction Level Parallelism

**Оптимизация кода**

1. Осуществляется профилирование кода (проблемных участков) и выявляются “горячие точки” (hot spots) – участки программы, на которые приходится больше всего времени выполнения
2. Горячие точки сортируются в порядке не убывания времени выполнения – формируется приоритетный список участков программы для последующей оптимизации
3. Каждая горячая точка оптимизируется
4. После оптимизации каждого “хот спота” проводится тестирование программы – проверяется совпадение результатов её работы до модификации и после внесения изменений.

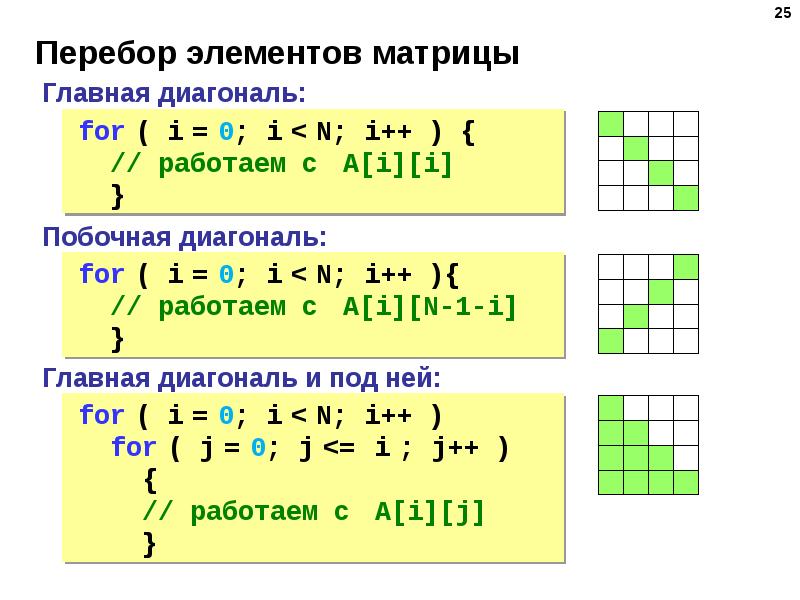
|  |
| --- |
| void DGEMM\_opt\_1(double\* X, double\* Y, double\* R, int N){  /\*в цикле происходит умножение путем построчного перебора элементов квадратных матриц, заданных в виде одномерных массивов\*/  for (int i=0;i<N;i++)  for (int j=0;j<N;j++)  for (int k=0;k<N;k++)  R[i\*N+k]+=X[i\*N+j]\*Y[j\*N+k];  } |

Оптимизация доступа к памяти за счёт построчного перебора элементов матриц в цикле (Функция DGEMM\_opt\_1)

Один из способов улучшения доступа к памяти при реализации алгоритма умножения матриц – применение блочных алгоритмов. При использовании блочных алгоритмов повышается локальность доступа к памяти. Как правило, при повышении локальности доступа к памяти повышается эффективность ее использования за счет уменьшения количества кеш-промахов. Пример гистограммы результатов работы программы DGEMM\_opt\_1:

Рисунок 2 Гистограмма результатов работы функции DGEMM\_opt\_1

# 5.Функция DGEMM\_opt\_2



Оптимизация доступа к памяти за счёт блочного перебора элементов матриц в цикле (Функция DGEMM\_opt\_2)

|  |
| --- |
| void DGEMM\_opt\_2(int size, double \*A, double \*B, double \*C, int size\_block) { int i, j, k, ik, jk, kk;  for(j = 0; j < size; j++) {  for(i = 0; i < size; i++) {  C[j \* size + i] = 0;  }  }  /\*матрицы разбиты на блоки\*/  /\*цикл построчного умножения элементов квадратных матриц, заданных в виде одномерных массивов вложен в цикл, перебирающий блоки матриц последовательно\*/  for(jk = 0; jk < size; jk+= size\_block)  for(kk = 0; kk < size; kk+= size\_block)  for(ik = 0; ik < size; ik+= size\_block)  for(j = 0; j < size\_block; j++ )  for(k = 0; k < size\_block; k++ )  #pragma simd  for(i = 0; i < size\_block; i++ )  C[(jk + j) \* size + (ik + i)] += A[(jk + j) \* size + (kk + k)] \* B[(kk + k) \* size + (ik + i)]; } |

Одним из основных способов использования векторных инструкций в программах на С/C++, является *векторизация* циклов компилятором. Под векторизацией *цикла* понимается одновременное, векторное, *исполнение* нескольких итераций *цикла* с использованием векторных инструкций. Естественно, такое *исполнение* возможно не для всех циклов. *Компилятор* осуществляет проверку возможности и целесообразности векторизации и, в случае выполнения обоих условий, генерирует код с использованием векторных инструкций. С помощью специальных средств языка и директив компилятора программист имеет возможность предоставить дополнительную информацию или дать определенные гарантии, которые могут оказать влияние на решение компилятора о векторизации *цикла*.

Итерации *цикла* являются независимыми и *векторизация* *цикла* потенциально возможна (если массивы не накладываются друг на друга). Проверяем, генерирует ли *компилятор* код с использованием векторных инструкций. Для этого создадим функцию main, в которой выделяем *память* для массивов и вызываем данную функцию. Для того, чтобы предотвратить ее встраивание, реализуем функцию vectorization\_simple в отдельном файле и используем #pragma noinline. Данные усилия прилагаются для имитации реальной ситуации, когда *компилятор* не будет встраивать сложные функции и, следовательно, не будет обладать дополнительной информацией.

Для проверки того, произведена ли *векторизация* и, при отрицательном ответе, вывода причин, в компиляторе есть возможность вывода отчета о векторизации. Данный отчет выводится при компиляции с ключом -vec-report[n] , где вместо [n] необходимо подставить число, определяющее степень подробности отчета (чем больше число, тем более подробный генерируется отчет).

Для каждого *цикла* указывается *имя файла* с исходным кодом и номер строки в скобках. Пример гистограммы результатов выполнения программы DGEMM\_opt\_2:

Рисунок 3 Пример гистограммы результатов выполнения DGEMM\_opt\_2

# 6.Функция DGEMM\_opt\_3

|  |
| --- |
| void DGEMM\_opt\_3(int n, double \*A, double \*B, double \*C){  int i, j, k;  for(j = 0; j < n; j++ )  for(i = 0; i < n; i++ )  C[j \* n + i] = 0;  for(j = 0; j < n; j++ )  for(k = 0; k < n; k++ )  //цикл идентичен DGEMM\_opt\_1  // делает векторизацию кода  #pragma simd  for(i = 0; i < n; i++ )  C[j \* n + i] += A[j \* n + k] \* B[k \* n + i];  } |

Векторизация кода перемножения матриц (Функция DGEMM\_opt\_3)

Важно иметь ввиду, что *компилятор* не имеет *права* произвести некорректную векторизацию *цикла*, которая привела бы к несовпадающим с не векторизованной версией результатам (не считая допустимого изменения порядка операций). Таким образом, при векторизации делаются лишь преобразования, для которых доказана *эквивалентность*. В связи с этим *компилятор* вынужден быть очень консервативным и, в отсутствие дополнительной информации, любая возможная зависимость является препятствием для векторизации. В рассматриваемой функции у компилятора нет информации о том, как соотносятся указатели. Например, если b[1] и a[0] расположены по одному и тому же адресу в памяти, то при векторном исполнении итераций 0 и 1 результат будет некорректен (отличаться от результата не векторизованного исполнения *цикла*).

В то же время программист может обладать информацией о том, что массивы никогда не пересекаются, что и происходит в рассматриваемом примере. В таком случае обнаруженная компилятором зависимость потенциально возможна, но на самом деле никогда не осуществляется. Существует несколько возможностей предоставить компилятору дополнительную информацию или гарантии отсутствия зависимости и, таким образом, способствовать векторизации.

Одним из способов предоставления гарантии того, что указатели не пересекаются, является использование ключевого слова restrict. Для компиляции с поддержкой ключевого слова restrict необходимо добавить ключ –restrict. Далее приведён пример гистограммы результатов DGEMM\_opt\_3:

Рисунок 4 Пример гистограммы результатов выполнения программы DGEMM\_opt\_3

# 7.Оценка ускорения умножения матриц фиксированного размера

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. Когда размер матриц n оказывается больше, чем число процессоров p, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк и столбцов перемножаемых матриц. В этом случае исходная матрица A разбивается на ряд горизонтальных полос, а матрица B представляется в виде набора вертикальных (для первого алгоритма) или горизонтальных (для второго алгоритма) полос. Размер полос при этом следует выбрать равным k=n/p (в предположении, что n кратно p ), что позволит по-прежнему обеспечить равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

Для распределения подзадач между процессорами может быть использован любой способ, обеспечивающий эффективное представление кольцевой структуры информационного взаимодействия подзадач. Для этого достаточно, например, чтобы подзадачи, являющиеся соседними в кольцевой топологии, располагались на процессорах, между которыми имеются прямые линии передачи данных.

Общая трудоемкость последовательного алгоритма является пропорциональной n3. Для параллельного алгоритма на каждой итерации каждый процессор выполняет умножение имеющихся на процессоре полос матрицы А и матрицы В (размер полос равен n/p, и, как результат, общее количество выполняемых при этом умножении операций равно n3/p2 ). Поскольку число итераций алгоритма совпадает с количеством процессоров, сложность параллельного алгоритма без учета затрат на передачу данных может быть определена при помощи выражения

С учетом этой оценки показатели ускорения и эффективности данного параллельного алгоритма матричного умножения принимают вид:

и

Таким образом, общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Для уточнения полученных соотношений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение *операций передачи данных* между процессорами.

С учетом числа и длительности выполняемых операций время выполнения вычислений параллельного алгоритма может быть оценено следующим образом:

(здесь  есть время выполнения одной элементарной скалярной операции).

Для оценки коммуникационной сложности параллельных вычислений будем предполагать, что все *операции передачи данных* между процессорами в ходе одной итерации алгоритма могут быть выполнены параллельно. Объем передаваемых данных между процессорами определяется размером полос и составляет n/p строк или столбцов длины n. Общее количество параллельных операций передачи сообщений на единицу меньше числа итераций алгоритма (на последней итерации передача данных не является обязательной). Тем самым, оценка трудоемкости выполняемых *операций передачи данных* может быть определена как

где  – латентность,  – пропускная способность сети передачи данных, а w есть размер элемента матрицы в байтах.

С учетом полученных соотношений общее время выполнения параллельного алгоритма матричного умножения определяется следующим выражением:

Чтобы оценить разницу времени выполнения опытным путем, «обложим» функции, предложенные в пункте 4, таймером chrono::steady\_clock:

|  |
| --- |
| void DGEMM\_BLAS(double\*\* X, double\*\* Y, double\*\* R, int N, double& time) {  auto start = chrono::steady\_clock::now();  for (int i = 0; i < N; i++)  for (int j = 0; j < N; j++) {  for (int k = 0; k < N; k++)  R[i][j] += X[i][k] \* Y[k][j];  }  auto end = chrono::steady\_clock::now();  chrono::duration<double> elapsed\_seconds = end - start;  time = elapsed\_seconds.count();  } |

|  |
| --- |
| void DGEMM\_opt\_1(double\*\* X, double\*\* Y, int N, double& time1)  {  auto start = chrono::steady\_clock::now();  for (int i=0;i<N;i++)  for (int j=0;j<N;j++)  for (int k=0;k<N;k++)  R[i][k]+=X[i][j]\*Y[j][k];  auto end = chrono::steady\_clock::now();  chrono::duration<double> elapsed\_seconds = end - start;  time1 = elapsed\_seconds.count();  } |

|  |
| --- |
| void DGEMM\_opt\_2(int size, double \*A, double \*B, double \*C, int size\_block, double& time1) {  int i, j, k, ik, jk, kk;  for(j = 0; j < size; j++) {  for(i = 0; i < size; i++) {  C[j \* size + i] = 0;  }  }  auto start = chrono::steady\_clock::now();  for(jk = 0; jk < size; jk+= size\_block)  for(kk = 0; kk < size; kk+= size\_block)  for(ik = 0; ik < size; ik+= size\_block)  for(j = 0; j < size\_block; j++ )  for(k = 0; k < size\_block; k++ )  #pragma simd  for(i = 0; i < size\_block; i++ )  C[(jk + j) \* size + (ik + i)] += A[(jk + j) \* size + (kk + k)] \* B[(kk + k) \* size + (ik + i)];  auto end = chrono::steady\_clock::now();  chrono::duration<double> elapsed\_seconds = end - start;  time1 = elapsed\_seconds.count(); } |

|  |
| --- |
| void DGEMM\_opt\_3(int n, double \*A, double \*B, double \*C, double& time1){  int i, j, k;  auto start = chrono::steady\_clock::now();  for(j = 0; j < n; j++ )  for(i = 0; i < n; i++ )  C[j \* n + i] = 0;  for(j = 0; j < n; j++ )  for(k = 0; k < n; k++ )  // делает векторизацию кода  #pragma simd  for(i = 0; i < n; i++ )  C[j \* n + i] += A[j \* n + k] \* B[k \* n + i];  auto end = chrono::steady\_clock::now();  chrono::duration<double> elapsed\_seconds = end - start;  time1 = elapsed\_seconds.count();  } |

# 8. Профилирование – сбор характеристик работы программы

Цели профилирования – Оценка эффективности работы программы или вычислительной системы – Определение критических участков программы (hotspots) или компонентов вычислительной системы

Полный код программы:

|  |
| --- |
| //[#include](https://vk.com/im?sel=207782921&st=%23include) <QCoreApplication> [#include](https://vk.com/im?sel=207782921&st=%23include) <iostream> [#include](https://vk.com/im?sel=207782921&st=%23include) <cstdio> [#include](https://vk.com/im?sel=207782921&st=%23include) <chrono> [#include](https://vk.com/im?sel=207782921&st=%23include) <time.h> using namespace std;  void DGEMM\_BLAS(double\*\* X, double\*\* Y, double\*\* R, int size, double& time) { auto start = chrono::steady\_clock::now(); for (int i = 0; i < size; i++)  for (int j = 0; j < size; j++)  for (int k = 0; k < size; k++)  R[i][j] += X[i][k] \* Y[k][j];  auto end = chrono::steady\_clock::now(); chrono::duration<double> elapsed\_seconds = end - start; time = elapsed\_seconds.count(); }  void DGEMM\_opt\_1(double\*\* X, double\*\* Y, double\*\* R, int N, double& time1) { auto start = chrono::steady\_clock::now(); for (int i = 0; i < N; i++)  for (int j = 0; j < N; j++)  for (int k = 0; k < N; k++)  R[i][k] += X[i][j] \* Y[j][k]; auto end = chrono::steady\_clock::now(); chrono::duration<double> elapsed\_seconds = end - start; time1 = elapsed\_seconds.count(); }  void DGEMM\_opt\_2(int size, double\* A, double\* B, double\* C, int size\_block, double& time1) {  int i, j, k, ik, jk, kk;  for (j = 0; j < size; j++)  {  for (i = 0; i < size; i++)  {  C[j \* size + i] = 0; } } auto start = chrono::steady\_clock::now();  for (jk = 0; jk < size; jk += size\_block)  for (kk = 0; kk < size; kk += size\_block)  for (ik = 0; ik < size; ik += size\_block)  for (j = 0; j < size\_block; j++)  for (k = 0; k < size\_block; k++) //[#pragma](https://vk.com/im?sel=207782921&st=%23pragma) simd  for (i = 0; i < size\_block; i++)  C[(jk + j) \* size + (ik + i)] += A[(jk + j) \* size + (kk + k)] \* B[(kk + k) \* size + (ik + i)];  auto end = chrono::steady\_clock::now(); chrono::duration<double> elapsed\_seconds = end - start; time1 = elapsed\_seconds.count(); }  void DGEMM\_opt\_3(int n, double\* A, double\* B, double\* C, double& time1) { int i, j, k; auto start = chrono::steady\_clock::now();  for (j = 0; j < n; j++)  for (i = 0; i < n; i++)  C[j \* n + i] = 0;  for (j = 0; j < n; j++)  for (k = 0; k < n; k++) // делает векторизацию кода  //[#pragma](https://vk.com/im?sel=207782921&st=%23pragma) simd  for (i = 0; i < n; i++)  C[j \* n + i] += A[j \* n + k] \* B[k \* n + i]; auto end = chrono::steady\_clock::now(); chrono::duration<double> elapsed\_seconds = end - start; time1 = elapsed\_seconds.count(); }  //[#define](https://vk.com/im?sel=207782921&st=%23define)  int main(int argc, char\* argv[]) { // QCoreApplication a(argc, argv);  srand(time(nullptr));  double\*\* X, \*\* Y, \*\* R; double\* A, \* B, \* C; int size = 500; double someTime;  int i, j, k, l;  for (k = 0; k < 3; k++) {  X = new double\* [size];  Y = new double\* [size];  R = new double\* [size];  A = new double[size \* size];  B = new double[size \* size];  C = new double[size \* size];  for (i = 0; i < size; i++)  {  X[i] = new double[size];  Y[i] = new double[size];  R[i] = new double[size];  for (j = 0; j < size; j++) {  X[i][j] = rand() % 101 - 50;  Y[i][j] = rand() % 101 - 50;  R[i][j] = 0;  A[i \* size + j] = X[i][j];  B[i \* size + j] = Y[i][j];  C[i \* size + j] = 0; } } /\* DGEMM\_BLAS(X, Y, R, size, someTime); cout « "DGEMM\_BLAS " « size « " " « someTime « endl;   DGEMM\_opt\_1(X, Y, R, size, someTime); cout « "DGEMM\_opt\_1 " « size « " " « someTime « endl; \*/  int blockSize[5] = { 4, 20, 50, 100, 250 }; double blockTime[5];  int id = 0;  for (l = 0; l < 5; l++) {   DGEMM\_opt\_2(size, A, B, C, blockSize[l], someTime);  cout « "DGEMM\_opt\_2 " « size « " " « someTime « " Block size: " « blockSize[l] « endl;  blockTime[l] = someTime; }  for(l = 0; l < 5; l++) {  if(blockTime[l] < blockTime[id])  id = l; } cout « "Оптимальный Block size для матрицы размерности " « size « ": " « blockSize[id] « endl;  /\* DGEMM\_opt\_3(size, A, B, C, someTime); cout « "DGEMM\_opt\_3 " « size « " " « someTime « endl; \*/ // среднее время / номер испытания / ускорение // проверка времени BLAS opt 1-3 // вызов с двумерной для ускорения CSV  for (i = 0; i < size; i++) {  delete[]X[i];  delete[]Y[i];  delete[]R[i]; }  delete[]X;  delete[]Y;  delete[]R;   delete[]A;  delete[]B;  delete[]C;  size += 500 + 500 \* k; }  // return a.exec(); } |

Программа выводит результат времени вычисления в консоль

**Результаты профилирования:**

Характеристики работы программы

Путь исполнения (покрытие кода, дерево вызовов подпрограмм, количество вызовов подпрограмм, …)

Время работы участка программы (функции, оператора языка, машинной команды, …)

Количество событий, произошедших в системе (исполненных команд, промахов кэша, неправильно предсказанных переходов, …)

Распределение времени и событий по коду программы

Потребление ресурсов системы (процессор, память, диск, сеть, …)

Анализ характеристик

Обнаружение критических участков программы (hotspots, critical path)

Оценка достигнутой производительности (хорошо/плохо), причины плохой производительности, варианты улучшения

Способы профилирования

Инструментирование кода – внесение в программу дополнительного

кода для сбора нужной информации

– Сильное влияние на характеристики программы

Обработка событий – код профилировщика вызывается при

срабатывании определенных событий

Отслеживание высокоуровневых событий (call/ret/new/delete/exception/…)

Статистическое профилирование (sampling) – остановка по

системным/аппаратным прерываниям и накопление статистики на

точках останова

+ Малое влияние на характеристики программы

– Результаты профилирования приблизительны

Эмуляция выполнения программы

+ Повторяемость результата

– Значительно большее время работы программы

– Не учитывается «реальная» ситуация

**gprof**

Компиляция программы

$g++ -O1 -g -pg -o cache cache.c

Создаётся файл: cache

Запуск программы (много раз)

$./cache

Создается файл: gmon.out

Получение профиля программы и графа вызовов

$gprof ./cache >cache.profile.txt

Создается файл: cache.profile.txt

Получение аннотированного листинга

$gprof -A cache >cache.source.txt

Создается файл: cache.source.txt

**gcov – тест покрытия кода**

Компиляция программы

$g++ -O1 --coverage -o cache cache.c

Создаются файлы: cache, cache.gcno

Запуск программы (много раз)

$./cache

Создаётся файл: cache.gcda

Получение аннотированного листинга

$gcov -b cache.c

Создаётся файл: cache.c.gcov

**Оптимизация в gcc с помощью профилирования**

Компиляция программы

$g++ -O1 -fprofile-generate -o cache cache.c

Создаётся файл: cache

Запуск программы (много раз)

$./cache

Создаётся файл: cache.gcda

Компиляция программы с оптимизацией

$ g++ -O1 -fprofile-use -o cache\_opt cache.c

Создается файл: cache\_opt

**OProfile**

Получение списка аппаратных счетчиков

$ophelp

Компиляция программы

$g++ -O1 -g -o cache cache.c

Запуск программы и профилирование

$operf -e CPU\_CLK\_UNHALTED:100000:0:0:1 ./cache

Создаётся каталог: ./oprofile\_data

Получение информации об исполнении программы

$opreport ./cache - профиль по модулям

$opreport -l ./cache - профиль по функциям

$opreport -с ./cache - граф вызовов

$opannotate -s ./cache - аннотированный исходный код

$opannotate -a ./cache - аннотированный ассемблерный код

$opgprof - генерация gmon.out для gprof