МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ
НЭТИ | Факультет прикладной
математики и информатики

Кафедра теоретической и прикладной информатики

Расчётно-графическая работа

по дисциплине «Языки программирования и методы трансляции»

МЕТОДЫ ОПТИМИЗАЦИИ КОДА. ОПТИМИЗАЦИЯ ЦИКЛОВ

Бригада 10 ИСАКИН ДАНИИЛ

Группа ПМ-13 ВОСТРЕЦОВА ЕКАТЕРИНА

Вариант 4

Преподаватель ДВОРЕЦКАЯ ВИКТОРИЯ КОНСТАНТИНОВНА

Новосибирск, 2024

Цель работы. Изучить теорию по оптимизации циклов.

Введение

В настоящее время компиляторы языков C/C++ предоставляют почти одинаковые возможности, поэтому оптимизация остается одним из важнейших отличий, по которому их можно определить. Компилятор переводит процедурное описание задач и эффективно отображает его в выделенное множество машинных команд процессора. Эффективность генерации низкоуровневого кода напрямую зависит на скорость выполнения программы и ее размер. Оптимизация кода должна выработать более экономичные потребляемые ресурсы кода.

Оптимизация циклов

Б*о*льшая часть времени исполнения программы приходится на циклы: это могут быть вычисления, прием и обработка информации и т.д. Правильное применение техник оптимизации циклов позволит увеличить скорость работы программы.

Время исполнения кода в циклах зависит от организации памяти, архитектуры процессора, в том числе, поддерживаемого набора инструкций, конвейеров, кэшей и опыта программиста.

Что такое цикл?

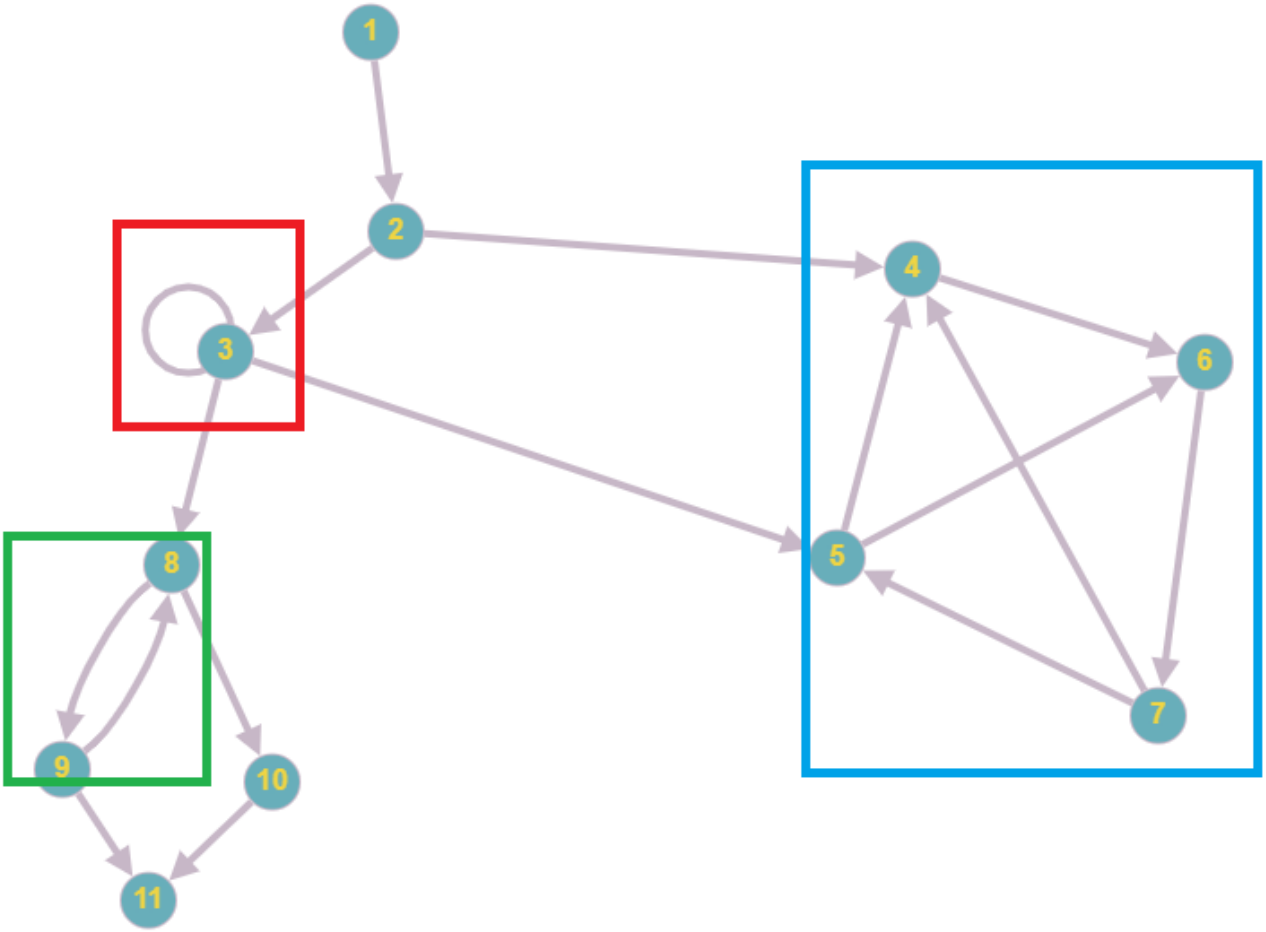
В теории компиляторов различают понятия cycle (цикл) и loop (петля). Cycle -- это любая последовательность блоков, такая, что начав с некоторого блока и двигаясь по рёбрам, вы можете вернуться в исходный блок. A loop -- это цикл, обладающий некоторыми дополнительными свойствами, это именно та конструкция, которая появляется, когда мы пишем цикл на языке программирования высокого уровня. Любой loop -- это cycle, но не любой cycle -- это loop.

При этом компиляторы (в большинстве своём) умеют оптимизировать только loop'ы, а циклы, не являющиеся loop'ами, чаще всего игнорируют или ведут себя с ними крайне осторожно. Такие конструкции языков программирования, как for, while, do-while, foreach и т.п. порождают именно loop.

Для начала нужно узнать что такое граф потока управления (CFG) и попытаться понять, как выглядят циклы в этом графе. Сначала пара общих определений:

Определение 1: Две вершины A и B *сильно связаны*, если в графе существует ориентированный путь как из A в B, так и из B в A.

Определение 2: Максимальный по включению связанный подграф называется *компонентой сильной связности*.



Компоненты сильной связности выделены цветными прямоугольниками

По графу видно, что вершины 8 и 9 - сильно связаны, потому что есть путь из 8 в 9 и наоборот. Вместе они образуют компоненту сильной связности. То же самое можно сказать о вершинах 4, 5, 6 и 7, которые попарно достижимы друг из друга и также формируют компоненту. Вершина 3 имеет ребро сама в себя, поэтому она также выделяется в отдельную компоненту сильной связности.

Компонента сильной связности - это обобщение понятие cycle. Например, внутри второй компоненты есть несколько cycle'ов, например, 4 → 6 → 7 и 5 → 6 → 7, но компонента - это именно максимальная по включению область, куда входят все они.

Определение 3: *Циклом* называется максимальный по включению сильно связный подграф, в котором одна из его вершин доминирует над всеми остальными. Эта вершина называется *головным блоком (header)* цикла, а все рёбра, идущие из данного цикла в головной блок, называются *обратными рёбрами (backedge)*. Если обратное ребро одно, то оно ещё может называться *latch*.

При этом цикл верхнего уровня - это всегда компонента сильной связности, но внутри неё могут быть и другие вложенные циклы. Они не являются компонентами сильной связности (потому что не максимальны по включению), и они доминируются другой вершиной.

Определение 4: Ребро, идущее из цикла за его пределы, называется *исходящим*. Блок цикла, откуда оно выходит - *выходящий блок (exiting block)*, а блок вне цикла, куда оно входит -- *блок выхода (exit block)*.

На рисунке выше красная и зелёная компонента являются циклами. В красном цикле блок 3 одновременно является головным и латчем (он доминирует сам себя и сам в себя идёт). В зелёной компоненте головным является блок 8, а латчем - блок 9. В синей же компоненте нет головного блока, потому что никакой из её блоков не доминирует все остальные. В этом легко убедиться: вы можете достичь вершин 6 и 7 как пройдя мимо вершины 4, так и пройдя мимо вершины 5. Ну или можете построить дерево доминирования и убедиться в этом ещё надёжнее.

Методы оптимизации циклов

Существует множество методов оптимизации циклов, рассмотрим лишь некоторые из них: развертывание циклов (loop unrolling), автоматическая векторизация (automatic vectorization) и выделение из цикла кода неизменяемых выражений (инвариантов) (loop-invariant code motion).

Рис. 1. Файл source1.c

```
#include <stdio.h> // scanf_s и printf
#include "Source2.h"
int square(int x) { return x*x; }
main() {
  int n = 5, m;
  scanf_s("%d", &m);
  printf("The square of %d is %d.", n, square(n));
  printf("The square of %d is %d.", m, square(m));
  printf("The cube of %d is %d.", n, cube(n));
  printf("The sum of %d is %d.", n, sum(n));
  printf("The sum of cubes of %d is %d.", n, sumOfCubes(n));
  printf("The %dth prime number is %d.", n, getPrime(n));

}
```

Если вы модифицируете код на **рис. 1** так, чтобы в sumOfCubes передавалась m вместо n, то компилятор не сможет определить значение параметра, поэтому он должен компилировать функцию для обработки любого аргумента. Полученная функция высоко оптимизирована и ее размер довольно велик, а значит, компилятор не станет подставлять ее.

Компиляция кода с ключом /O1 приводит к созданию ассемблерного кода, оптимизированного по размеру. В этом случае к функции sumOfCubes никакие оптимизации не применяются. Компиляция с ключом /O2 дает код, оптимизированный по скорости работы. Размер кода будет значительно больше, но сам код будет работать существенно быстрее, так как цикл в sumOfCubes развернут и векторизован. Важно понимать, что векторизация была бы невозможна без подстановки функции cube. Более того, без подстановки функций развертывание циклов было бы не столь эффективным. Упрощенное графическое представление создаваемого ассемблерного кода показано на **рис. 3**. Схема потока управления одинакова для архитектур x86 и x86-64.

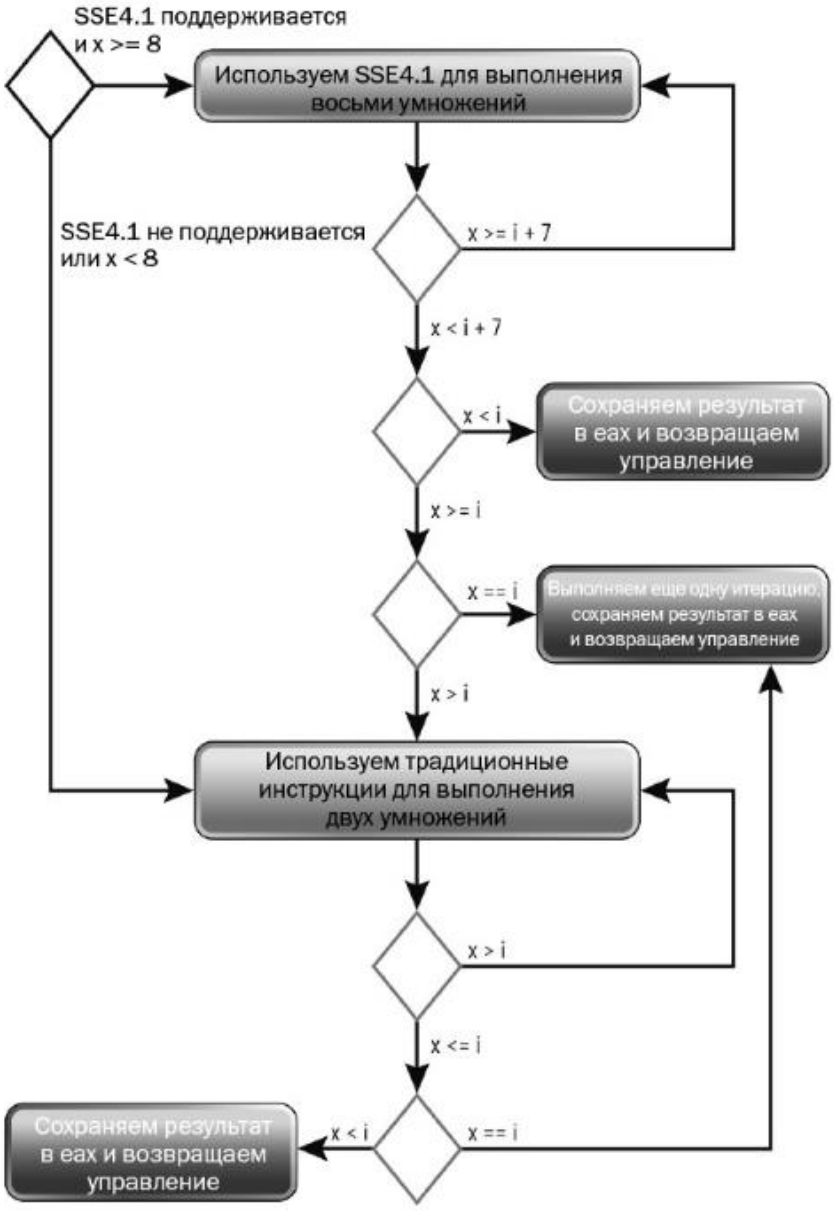


Рис. 3. Схема потока управления в sumOfCubes

SSE4.1 supported and x >= 8	SSE4.1 поддерживается и x >= 8

Use SSE4.1 to perform 8 multiplications	Используем SSE4.1 для выполнения восьми умножений
SSE4.1 not supported or x < 8	SSE4.1 не поддерживается или x < 8
Store result in eax and return	Сохраняем результат в eax и возвращаем управление
Iterate once more, store result in eax and return	Выполняем еще одну итерацию, сохраняем результат в eax и возвращаем управление
Use traditional instructions to perform 2 multiplications	Используем традиционные инструкции для выполнения двух умножений
Store result in eax and return	Сохраняем результат в eax и возвращаем управление

На **рис. 3** черный ромб обозначает точку входа, а прямоугольные блоки указывают точки выхода. Светло-серые ромбы представляют условия, которые выполняются как часть функции sumOfCubes в период выполнения. Если SSE4 поддерживается процессором и x больше или равен восьми, тогда SSE4-инструкции будут задействованы для выполнения четырех умножений одновременно. Процесс выполнения одной и той же операции над несколькими значениями одновременно называют векторизацией (vectorization). Кроме того, компилятор будет дважды развертывать цикл, т. е. тело цикла будет дважды повторяться на каждой итерации. В итоге на каждой итерации будут выполняться восемь умножений. Когда **X** станет меньше восьми, для выполнения остальных вычислений будут использоваться традиционные инструкции. Заметьте, что компилятор сгенерировал три точки выхода, содержащие отдельные эпилоги в функции вместо одного. Это уменьшает количество переходов.

Развертывание цикла — это процесс повторения его тела в пределах цикла, так чтобы более одной итерации цикла выполнялось в рамках одной итерации развернутого цикла. Причина, по которой это увеличивает производительность, заключается в том, что инструкции, управляющие циклом, выполняются реже. Еще важнее, что это, возможно, позволит компилятору выполнить много других оптимизаций, например векторизацию. Недостаток развертывания в том, что он увеличивает размер кода и нагрузку на регистры. Однако в зависимости от тела цикла развертывание может повысить скорость работы кода на десятки процентов.

Глядя на сгенерированный ассемблерный код, можно заметить, что этот код можно еще немного оптимизировать. Однако компилятор уже проделал большую работу и не станет тратить гораздо больше времени на анализ кода и применение малозначимых оптимизаций.

Функция someOfCubes — не единственная, чей цикл можно развернуть. Если модифицировать код таким образом, что в функцию sum будет передаваться m вместо n, компилятор не сможет оценить эту функцию и поэтому сгенерирует ее код. В этом случае цикл будет развернут дважды.

Последняя оптимизация — выделение из цикла кода инвариантов (loop-invariant code motion). Рассмотрим следующий фрагмент кода:

```
int sum(int x) {
  int result = 0;
  int count = 0;
  for (int i = 1; i <= x; ++i) {
    ++count;
    result += i;
  }
  printf("%d", count);
  return result;
}
```

Здесь единственное изменение — дополнительная переменная, которая увеличивается на 1 при каждой итерации, а затем ее значение выводится. Нетрудно увидеть, что этот код может быть оптимизирован переносом приращения переменной-счетчика за пределы цикла. То есть я могу просто присвоить **x** переменной-счетчику. Эту оптимизацию называют выделением из цикла кода инвариантов. Инвариантная к циклу часть ясно показывает, что этот метод работает, только когда код не зависит от какого-либо выражения в заголовке цикла.

Некоторые важные оптимизации могут быть выполнены лишь при анализе полной программы.

Параллелизм в циклах

Еще одной важной частью оптимизации является параллелизация циклов.

Можно разделить большое количество итераций в циклах между процессорами. Если количество вычислений, выполняемых в каждой итерации, примерно одинаковое, простое равномерное разделение итераций по процессорам обеспечит максимальную степень параллелизма. Исключительно простой пример 11.1 демонстрирует возможность использования преимуществ параллелизма на уровне цикла.

Пример 11.1.

```
for(i = 0; i < n; i++) {
```

2[1] = X[i] - Y[i];

2[i]

= Z[i] * Z{[i];

}

Цикл вычисляет квадраты разностей между элементами векторов X и У и сохраняет их в векторе 2. Цикл распараллеливаем, поскольку каждая его итерация обращается к различным множествам данных. Можно выполнить цикл на компьютере с M процессорами, присваивая каждому процессору уникальный идентификатор p = = 0, 1, ..., M 1 и выполняя на каждом процессоре один и тот же код

b

ceil(n/M);

for(i

z[i] z[i]

}

b*p; i < min(n,b*(p+1)); i++) {

=

X[1] - Y[i];

Z[i] * Z[i];

Мы равномерно разделяем итерации цикла между процессорами; p-й процес- сор получает для выполнения p-ю "полосу" итераций. Заметим, что количество итераций может не быть кратно M, так что мы не можем гарантировать, что последний процессор не достигнет конца исходного цикла за минимальное коли- чество операций.

Параллельный код, приведенный в примере 11.1, представляет собой SPMD-программу (Single Program Multiple Data - одна программа, много данных). Один и тот же код выполняется всеми процессорами, но для каждого процессора этот код параметризован уникальным идентификатором процессора, так что различные процессоры могут выполнять различные действия. Обычно один процессор, известный как ведущий (master), выполняет все последовательные части вычислений. При достижении параллелизованной части кода ведущий процессор активирует все ведомые (slave) процессоры. Все вместе процессоры выполняют параллелизованную часть кода, после чего

участвуют в барьерной синхронизации (barrier synchronization). Все операции, выполняемые процессором до входа в барьер синхронизации, гарантированно завершаются до того, как любому другому процессору будет позволено покинуть этот барьер и выполнить операцию, находящуюся за ним.

Если параллелизуются только небольшие циклы наподобие приведенного в примере 11.1, то получающийся в результате код имеет невысокую степень параллелизма и относительно малую его зернистость. Предпочтительно параллелизовать внешние циклы программы, поскольку это существенно повышает зернистость параллелизма. Рассмотрим, например, приложение двумерного быстрого преобразования Фурье (БПФ), работающего с массивом данных размером их п. Такая программа выполняет и БПФ каждой строки данных, затем БПФ столбцов. Предпочтительнее назначить каждому из п независимых БПФ свой процессор, чем использовать несколько процессоров для выполнения одно- го БПФ. Такой код легче написать, степень параллельности алгоритма окажется равной 100%, и код будет иметь хорошую локальность, поскольку во время вычисления БПФ взаимодействие процессоров не потребуется.

Источники:

1. https://learn.microsoft.com/ru-ru/archive/msdn-magazine/2015/february/compilers-what-every-programmer-should-know-about-compiler-optimizations
2. https://habr.com/ru/articles/742062/
3. file:///C:/Users/katya/Downloads/Ахо,_Сети,_Ульман._Компиляторы._Принципы,_технологии,_инструменты.2ed.2008.pdf