Компиляторы Intel. Обзор возможностей. Автоматическая оптимизация

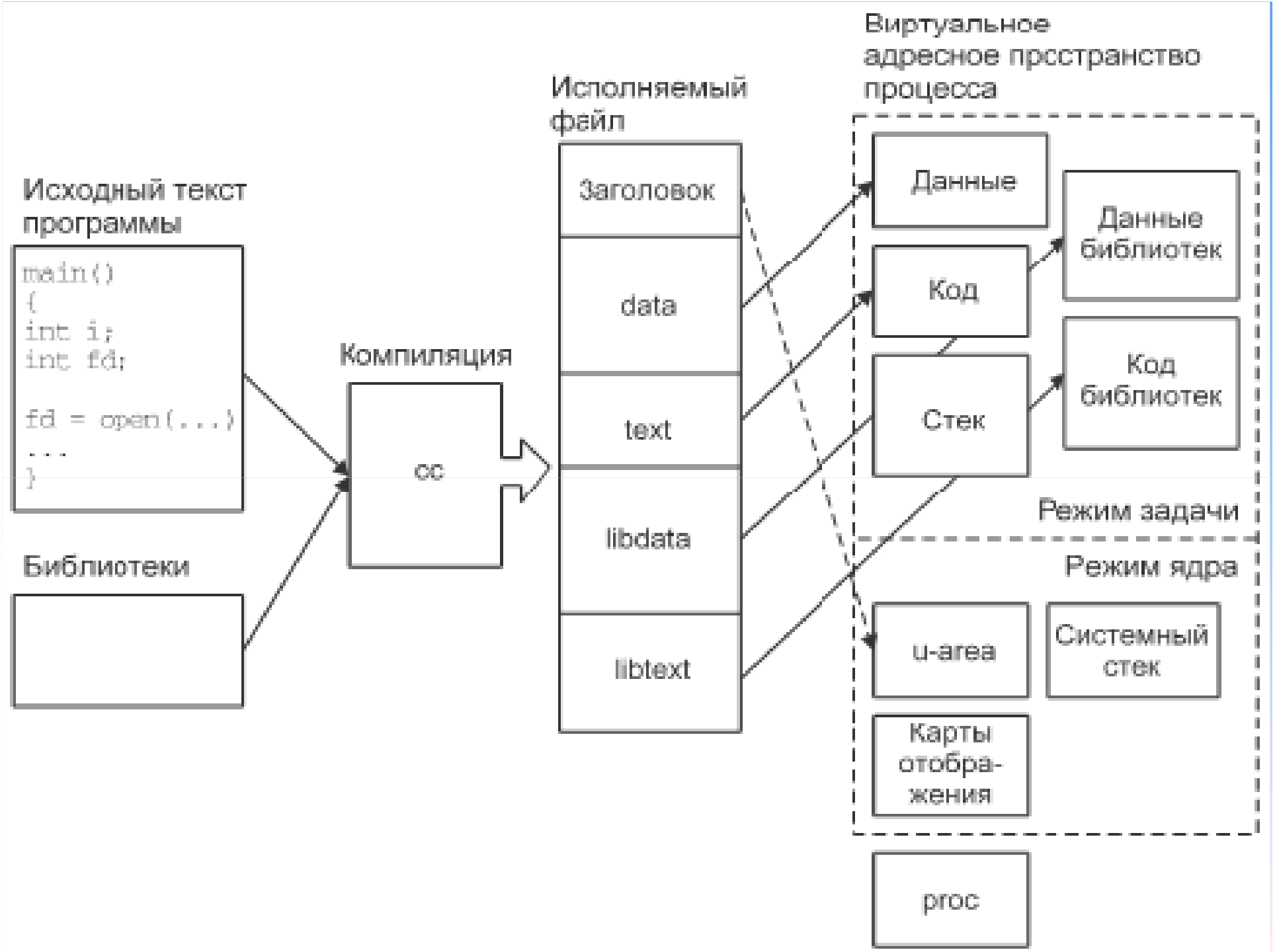
С.А.Немнюгин

Высокопроизводительные вычисления на GRID системах. Архангельск, 2012

Содержание

- Компиляция – первый шаг в создании процесса.
- Компиляторы Intel.
- Оптимизирующие преобразования.
- Отчёты об оптимизации.
- Оптимизация в примерах.

Компиляция – первый шаг в создании процесса


Виртуальное адресное прсстранство процесса
Исполняемый фейл
Исходный текст программы
main()
{
int i;
int fd;
fd = open(...)
...
}
Компиляция
cc
Библиотеки
Заголовок
data
text
libdata
libtext
Данные
Код
Стек
Данные библиотек
Код библиотек
Режим задачи
Режим ядра
u-area
Системный стек
Карты отобра-жения
proc

Компиляторы Intel

Intel® Composer XE включает компиляторы C/C++, Fortran, а также библиотеки.

Поддерживаются платформы Microsoft Windows и Linux.

Поддерживается работа с оптимизированными библиотеками: Intel®MKL, Intel®IPP.

Средства поддержки оптимизации.

Улучшенная поддержка векторизации (увеличенная разрядность векторных инструкций).

Интеграция в среды разработки Microsoft Visual Studio, Eclipse, XCode.

Совместимость с Microsoft Visual C, компиляторами GCC (Linux) и MacOS.

Поддержка Fortran 77 – 2003. Поддержка COARRAY и DO CONCURRENT из Fortran 2008.

Подробная диагностика.

Поддержка Intel® Cilk™ Plus

Intel® Cilk™ Plus – средство разработки параллельных программ, включающее небольшой набор ключевых слов, гиперобъекты, средства работы с массивами (расширенная индексная нотация), эффективную поддержку векторизации и другое.

- «Fork-join» модель параллельного программирования.
- Последовательная семантика, если игнорировать ключевые слова Cilk.
- Эффективная сбалансированная диспетчеризация на основе захвата работы.
- Гиперобъекты.
- Поддержка векторизации с помощью сечений массивов и элементных функций.

Запуск (режим командной строки):

`# icl` (MS Windows)

`# icc` (Linux)

Ключи в обоих случаях аналогичны.

Несовместимые ключи игнорируются.

Оптимизирующие преобразования

Использование возможностей автоматической оптимизации компилятора может дать <u>значительный</u> выигрыш в производительности

Какую оптимизацию может выполнить компилятор

Удаление общих подвыражений

Если одно и то же подвыражение встречается несколько раз, компилятор может вычислить его один раз, а все последующие включения заменяются вычисленным значением:

```
a = (b+35) * (b+35);
c = 12.0 * (b+35);
```

в результате удаления общего подвыражения превращается в следующий:

```
tmp = b + 35;
a = tmp * tmp;
c = 12.0 * tmp;
```

Развёртка цикла

Развёртка заключается в преобразовании цикла в линейную последовательность операций. Выигрыш в производительности достигается благодаря снижению накладных расходов на организацию цикла. Эффективным этот приём может быть в случае, когда тело цикла очень маленькое:

```
ifor (i = 0; i < 2; i++) a[i] = 2.0 * d[i];
```

После развёртки:

```
a[0] = 2.0 * d[0];
a[1] = 2.0 * d[1];
a[2] = 2.0 * d[2];
```

Удаление инвариантных выражений из тела цикла

Выражения, не зависящие от счётчика цикла (*инварианты цикла*), целесообразно выносить за пределы цикла, так как это позволяет уменьшить количество операций:

```
for (i = 0; i < 10; i++) {
a[i] = b * i + 0.35;
}
```

преобразуется следующим образом:

```
temp = b * 2.0 + 0.35;
for (i = 0; i < 10; i++) {
a[i] = temp;}
```

Подстановка функций

Это преобразование заключается в замене обращения к функции её кодом. Выигрыш в производительности может быть получен благодаря уменьшению накладных расходов на вызов функции и возвращение из неё, а также на передачу параметров.

Часто подстановка функции увеличивает эффективность использования кэш-памяти, так как код становится непрерывным.

Недостатком является увеличение размера программы, которое тем более заметно, чем больше размер функции и чем больше число обращений к ней.

Свёртка и распространение констант

Подвыражение, содержащее только константы, вычисляется, а результат его вычисления подставляется в код программы:

```
a = b + 2.0 / 3.0;
```

Компилятор заменит данный код следующим:

```
a = b + 0.6666666666666666667;
```

Первый вариант более удобен для программиста, поскольку улучшает читаемость кода.

Упростить свёртку констант компилятором можно, разместив константное выражение в скобках.

Исключение указателей

При автоматической оптимизации указатель или ссылка могут быть удалены, если известен их адресат:

```
void sub(int * p) {
*p = *p - 5;}
 int main()
{
int a;
 a = 10;
sub(&a);
```

В результате преобразования генерируется следующий код:

```
a = 10;
a -= 5;
```

Объединение ветвей

Этот вид оптимизации заключается в объединении одинаковых фрагментов кода, например фрагмент кода:

```
if (b) { y = x*x; z = y * 1.5; }
else {
y = sin(x); z = y * 1.5;
}
```

после оптимизации будет выглядеть следующим образом:

```
if (b) { y = x*x; }
else { y = sin(x); }
z = y * 1.5;
```

Алгебраическая редукция

Алгебраическая редукция заключается в упрощении алгебраических выражений. Компиляторы могут выполнять редукцию относительно простых выражений, хотя и с определёнными ограничениями.

```
a * b + a * c = a * (b + c)
A + a + a + a = a * 4
-(-a) = a
A - a = 0
(-a) * (-b) = a * b
a * 1 = a
a / a = 1
...
```

Следует обращать внимание на корректность программы после оптимизации с использованием алгебраической редукции.

Редукция логических выражений

```
!(!a) = a
(a&&b) || (a&&c) = a&&(b||c)
!a && !b = !(a || b)
a && !a = false, a || !a = true
a && true = a, a || false = a
a && false = false, a || true = true
a && a = a
(a&&b) || (!a&&c) = a ? b : c
```

Управление точностью вычислений

Примеры ключей управления точностью:

```
/Qfp:double         (53)
/Qfp:extended       (64)

/Qpc32|64|80
```

Исключение переходов

Объединение ветвей условного оператора

Использование регистровых переменных

Использование индуктивных переменных

Изменение последовательности выполнения операций

При выполнении арифметических операций с плавающей точкой изменение последовательности операндов может привести к изменению результата вычисления арифметического выражения. В примере:

```
float a = -1.0E8, b = 1.0E8, c = 1.23456, y;
y = a + b + c;
printf("%f\n", y);
 y = b + c + a;
printf("%f", y);
```

оба результата – разные.

…

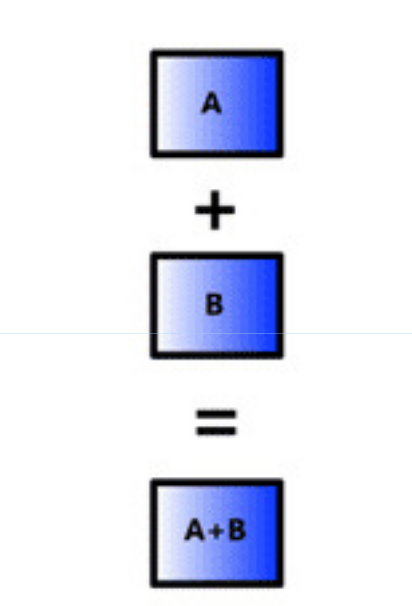
Другие виды автоматической оптимизации

- Оптимизация под архитектуру
- Распараллеливание
- Оптимизация с профилированием
- Межпроцедурная оптимизация

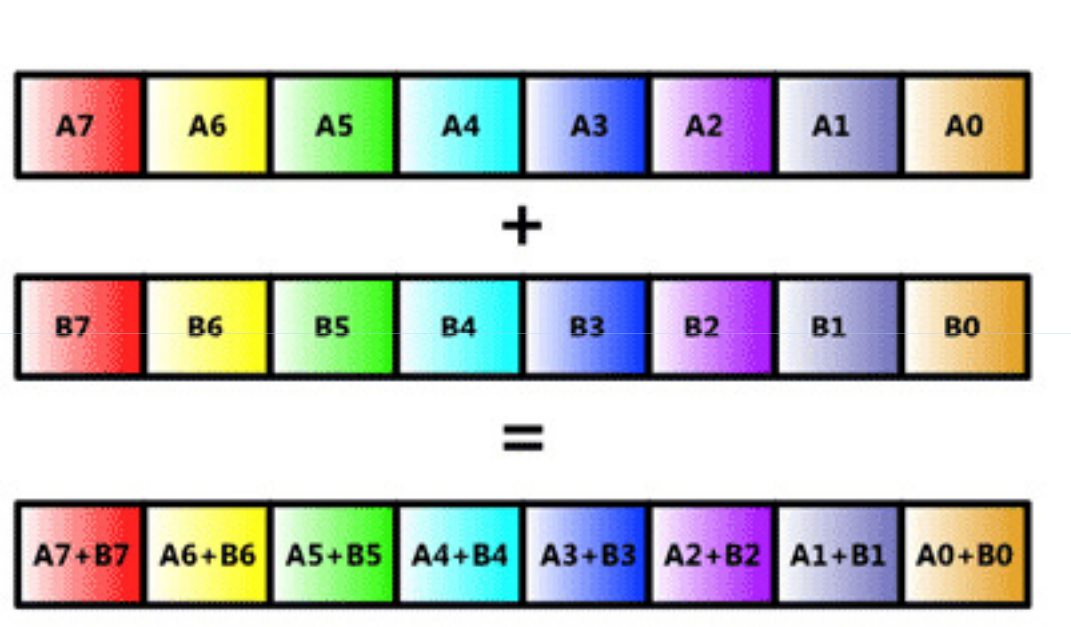
и т.д.

Векторизация

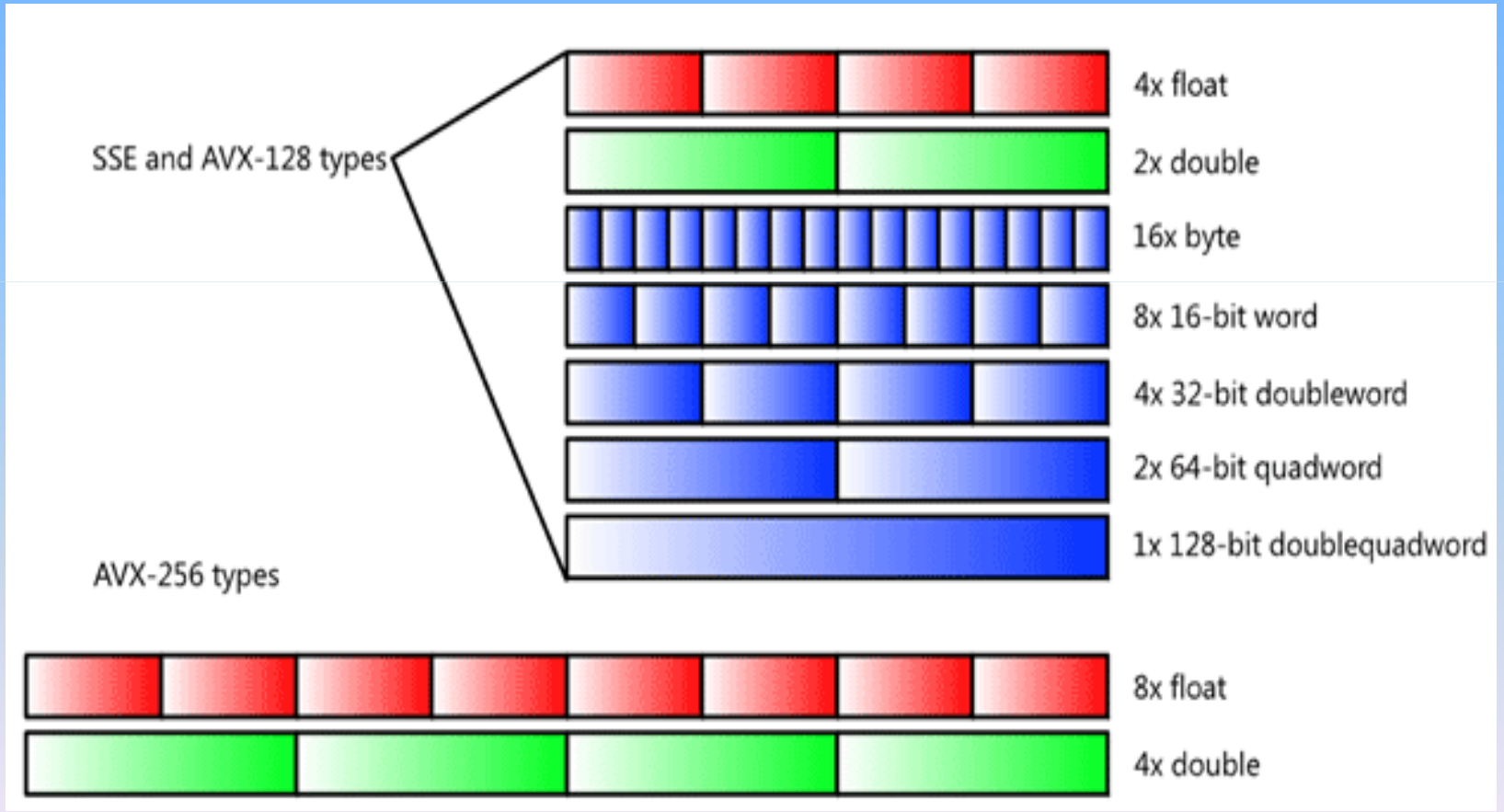
Скалярная операция



SIMD операция



Типы данных SSE и AVX



AVX – набор векторных команд

Intel® AVX (Advanced Vector Extensions) расширяет возможности 128-разрядных векторных регистров XMM (16 регистров), позволяя использовать 256-разрядные векторы.

256 разрядов используются в операциях с плавающей точкой.

В других операциях используются младшие 128 разрядов.

В AVX операнды и результат операции отделены друг от друга. Это позволяет оптимизировать использование регистров, генерировать более компактный код, уменьшить количество зависимостей т.д.

Поддержка векторных конструкций в Intel® Cilk™ Plus и Array Building Blocks.

Поддержка прагм и интринсиков (низкоуровневые возможности).

Поддержка сложных условных выражений.

Улучшенная поддержка смешения типов в одном цикле.

Поддержка операций с насыщением.

Отчёты об оптимизации

Отчёты об оптимизации

- /Qvec-report – генерация протокола векторизации (что векторизовано, что нет и почему).
- /Qpar-report - генерация протокола распараллеливания.
- /Qopt-report – генерация протокола оптимизации.

Профилирование на уровне циклов

Сбор статистики по циклам и функциям

- /Qprofile-loops:all - сбор статистики.
- LoopProfileViewer - утилита для просмотра отчётов.
- GAP - Guided Auto Parallelism (направляемая автопараллелизация).
- /Qguide (-guide) - запуск анализа.
- Не порождает параллельный код, но даёт рекомендации.

Оптимизация в примерах

/Od (Windows), -O0 (Linux)

- Отключение оптимизации
- Подразумевается в режиме Debug в MS Visual Studio

/O1 (Windows), -O1 (Linux)

- Глобальная оптимизация
- Не увеличивает размер кода

/O2 (Windows), -O2 (Linux)

- Увеличивает размер кода
- Оптимизация времени выполнения
- Опция по умолчанию
- Подстановки в коде
- Развертывание циклов
- Векторизация

/O3 (Windows), -O3 (Linux)

- Высокоуровневая оптимизация.
- /O2 + более агрессивные методы.
- Улучшенная векторизация.
- Более полный учёт свойств циклов и массивов.
- Оптимизация циклов: разделение циклов (loop distribution), перестановка циклов (loop interchange), слияние циклов (loop fusion), развёртка циклов (loop unrolling).
- Подстановка кода в ветвлениях.
- Оптимизация под размер кэша.
- Предвыборка и предсказания ветвления.
- Возможна большая эффективность в приложениях, включающих обработку больших массивов.

Пример

Matrices.cpp

- icl /FeMatricesOd /Od Matrices.cpp
- icl /FeMatricesO1 /O1 Matrices.cpp
- icl /FeMatricesO2 /O2 Matrices.cpp
- icl /FeMatricesO3 /O3 Matrices.cpp

```
#include <ctime>
#include <iostream>
#include "stdlib.h"
#include "conio.h"
#include "math.h“
using namespace std;
const int SIZE = 1000;

void matrixMultiply(double ** matrA, double ** matrB, double ** matrC, int matrSize);
double sinCosMultiply(double i, double j);

int main()
{
	time_t t1, t2;
	double trace(0);
	double ** matrixA = new double*[SIZE];
	double ** matrixB = new double*[SIZE];
	cout << "We will construct the square matrix " << SIZE << "x" << SIZE << endl;
	cout << "of pseudo-random values" << endl;
	cout << "and multiply it by itself. Then we will find the trace of the matrix." << endl;

	for(int i = 0; i < SIZE; i++)
	{
		matrixA[i] = new double[SIZE];
		matrixB[i] = new double[SIZE];
	}
```

```
t1 = clock();

for(int i=0; i<SIZE; i++)
{
        for(int j=0; j<SIZE; j++)
        {
                matrixA[i][j]=sinCosMultiply(rand()%10/3, rand()%10/3);
        }
}
matrixMultiply(matrixA, matrixA, matrixB, SIZE);
for(int i=0; i<SIZE; i++)
{
        for(int j=0; j<SIZE; j++)
        {
                if(i==j) trace += matrixB[i][j];
        }
}
t2 = clock();
cout << "\nTrace is " << trace << endl;
cout << "\nInitial clock ticks value is " << t1 << endl;
cout << "Final clock ticks value is " << t2 << endl;
cout << "Difference in clock ticks is " << difftime(t2,t1) << endl;
cout << "Clock ticks per second value is " << CLOCKS_PER_SEC << endl;
cout << "\nActual time of calculations is " << ((t2-t1)/CLOCKS_PER_SEC) << " sec" << endl;
```

```
	for(int i = 0; i < SIZE; i++)
	{
		delete [] matrixA[i];
		delete [] matrixB[i];
	}
	delete [] matrixA;
	delete [] matrixB;
	return 0;
}
void matrixMultiply(double ** matrA, double ** matrB, double ** matrC, int matrSize)
{
	for(int i=0; i<matrSize; i++)
	{
		for(int j=0; j<matrSize; j++)
		{
			matrC[i][j] = 0;
			for(int k=0; k<matrSize; k++)
			{
				matrC[i][j] += matrA[i][k]*matrB[k][j];
			}
		}
}	}
double sinCosMultiply(double i, double j)
{
	return sin(i)*cos(j);
}
```

Результаты

Intel Core 2 Duo T3700 2.00 ГГц, 2 Гб RAM

Среднее значение времени выполнения:

- MatricesOd . . . 21.76 с.
- MatricesO1 . . . 14.05 с.
- MatricesO2 . . . 9.63 с.
- MatricesO3 . . . 9.57 с.

Оптимизация под архитектуру

/Qax (Windows), -ax (Linux)

- Оптимизация под архитектуру Intel
- QxHost
- QxAVX
- QxSSE2, QxSSE3, QxSSE3_ATOM, QxSSE4.1, QxSSE4.2, QxSSSE3

Пример

Primes.cpp

- icl /FePOd /Od Primes.cpp
- icl /FePx /QaxSSSE3 Primes.cpp

Intel Core 2 Duo T3700 2.00 ГГц, 2 Гб ОЗУ

Среднее значение времени выполнения:

- POd . . . . . 29.8 с.
- Px . . . . . 4.5 с.

```
#include "time.h"
#include <iostream>
#include "stdlib.h"
#include "conio.h"
#include "math.h“
using namespace std;
const int NUM_OF_TESTS = 2000000;
const int RANGE = 1000;
bool isComposite(const int y);
int main()
{
	time_t t1, t2;
	int counter(0);
	t1 = clock();
	for	(int i = 0; i < NUM_OF_TESTS; i++)
		if(isComposite(rand()%RANGE))
			++counter;
	t2 = clock();

	cout << "For " << NUM_OF_TESTS << " pseudo-random values" << endl;
	cout << "within the range 0.." << RANGE << endl;
	cout << ((NUM_OF_TESTS-counter)*100.0)/NUM_OF_TESTS << "% were prime numbers." << endl;
```

```
	cout << "\nInitial clock ticks value is " << t1 << endl;
	cout << "Final clock ticks value is " << t2 << endl;
	cout << "Difference in clock ticks is " << difftime(t2,t1) << endl;
	cout << "Clock ticks per second value is " << CLOCKS_PER_SEC << endl;
	cout << "\nActual time of calculations is " << ((t2-t1)/CLOCKS_PER_SEC) << " sec" << endl;

	return 0;
}

bool isComposite(const int y)
{
	bool result = false;
	for (int i = 2; i <= ceil(y/2.0); i++)
	{
		if (y%i == 0) result = true;
	}
	return result;
}
```

Результаты

Intel Core 2 Duo T3700 2.00 ГГц, 2 Гб ОЗУ

Среднее значение времени выполнения:

- POd . . . . . 29.8 с.
- Px . . . . . 4.5 с.

Автоматическое распараллеливание

/Qparallel (Windows), -parallel (Linux)

- Автоматическое распараллеливание.
- Определяются те части кода, которые можно распараллелить.
- Выполняется анализ зависимостей.
- Разделение данных для параллельной обработки.
- Работа с циклами.

Пример

FermatsCubes.cpp

- icl /FeFCOd /Od FermatsCubes.cpp
- icl /FeFCPar /Qparallel FermatsCubes.cpp

Сравнить результаты

```
#include "time.h"
#include <iostream>
#include "stdlib.h"
#include "conio.h"
#include "math.h“
using namespace std;
const int RANGE = 2000;
long sumOfCubes(const int x, const int y);
int main()
{
	time_t t1, t2;
	bool isDisproven = false;
	cout << "Checking for mispredictions of Fermat's Great Theorem" << endl;
	cout << "for cubes in range 3.." << RANGE << endl;
	t1 = clock();
	for (int i = 3; i < RANGE; i++)
	{
		for(int k = 1; k < i; k++)
		{
			for(int j = 1; j < i; j++)
			{
				if (long(i*i*i) == sumOfCubes(k,j))
				{
					cout << k << " " << j << " " << i << endl;
					isDisproven = true;
				}}}}
```

```
        t2 = clock();
        cout << "\nAre theorem's predictions correct? +" << !(isDisproven) << endl;
        cout << "\nInitial clock ticks value is " << t1 << endl;
        cout << "Final clock ticks value is " << t2 << endl;
        cout << "Difference in clock ticks is " << difftime(t2,t1) << endl;
        cout << "Clock ticks per second value is " << CLOCKS_PER_SEC << endl;
        cout << "\nActual time of calculations is " << ((t2-t1)/CLOCKS_PER_SEC) << " sec" << endl;
        return 0;
}

long sumOfCubes(const int x, const int y)
{
        return (x*x*x + y*y*y);
}
```

Результаты

Среднее значение времени выполнения:

- FCOd . . . . . 25.95 с.
- FCPar . . . . . 3.78 с.

Оптимизация с профилированием

/Qprof-gen (Windows), -prof-gen (Linux)
/Qprof-use (Windows), -prof-use (Linux)

- Инструментовка.
- Сбор информации.
- Компиляция с учетом проанализированных данных.

Пример

Branches.cpp

- icl /FeBOd /Od Branches.cpp
- icl /FeBPrg /Qprof-gen Branches.cpp
- BPrg.exe
- icl /FeBProf /Qprof-use Branches.cpp

```
#include "time.h"
#include <iostream>
#include "stdlib.h"
#include "conio.h"
#include "math.h“
using namespace std;
bool slow_func(void);
bool quick_func(void);
int main()
{
	time_t t1, t2;
	t1 = clock();
	if(slow_func() && quick_func()) cout << "You can't see this...";
	t2 = clock();
	cout << "\nInitial clock ticks value is " << t1 << endl;
	cout << "Final clock ticks value is " << t2 << endl;
	cout << "Difference in clock ticks is " << difftime(t2,t1) << endl;
	cout << "Clock ticks per second value is " << CLOCKS_PER_SEC << endl;
	cout << "\nActual time of calculations is " << ((t2-t1)/CLOCKS_PER_SEC) << " sec" << endl;

	return 0;
}
```

```
bool slow_func(void)
{
	double a(0);
	for(int i = 0; i < 50000000; i++)
		a = pow((sin(a/2.0)+cos(a/2.0))*(sin(i/2.0)+cos(i/2.0)), 2);
	for(int i = 0; i < 100; i++)
		a += i;
	return !a;
}

bool quick_func(void)
{
		return false;
}
```

Результаты

Среднее значение времени выполнения:

- BOd . . . . . 13.171 с.
- BProf . . . . . 0.0 с.

Межпроцедурная оптимизация

/Qip (Windows), -ip (:inux)

- Анализ вызываемых функций приложения.
- Оптимизация многочисленных «маленьких» функций, особенно в циклах.
- Встраивание (inlining, подстановка кода) – уменьшение накладных расходов, создание возможностей для других видов оптимизации.
- Удаление неиспользуемого кода.
- Замена виртуальных вызовов статическими.
- Замена параметра функции константой.
- Эффективный анализ зависимостей для оптимизации циклов, векторизации и распараллеливания.
- Размер бинарного файла и время компиляции увеличиваются.

Пример

SquareRoots.cpp

- icl /FeSqOd /Od SquareRoots.cpp
- icl /FeSqip /Qip SquareRoots.cpp

```
#include "time.h"
#include <iostream>
#include "stdlib.h"
#include "conio.h"
#include "math.h“
using namespace std;
const int N = 10000; //number of steps in grid used for numerical method
const int LIM = 50000; //we want to find the sum of square roots of 1..LIM
double sqrootNewton(double x);
double returnHalf(double value);
double multiply(double value1, double value2);
int main()
{
	time_t t1, t2;
	double numRes(0), trueRes(0);
	t1 = clock();
	for (int i = 0; i < LIM; i++)
	{
		numRes += sqrootNewton(i+1);
		trueRes += sqrt((i+1)/1.0);
	}
	t2 = clock();
	cout << "Numerical result via Newton's method is " << numRes << endl;
	cout << "PC arithmetical result is " << trueRes << endl;
```

```
	cout << "\nInitial clock ticks value is " << t1 << endl;
	cout << "Final clock ticks value is " << t2 << endl;
	cout << "Difference in clock ticks is " << difftime(t2,t1) << endl;
	cout << "Clock ticks per second value is " << CLOCKS_PER_SEC << endl;
	cout << "\nActual time of calculations is " << ((t2-t1)/CLOCKS_PER_SEC) << " sec" << endl;
	_getch();
	return 0;
}
double sqrootNewton(double x) // Newtonian method applied for square root
{
	double sq = 1.0;
	for (int i = 0; i < N; i++)
		sq = returnHalf((sq + multiply(x, 1/sq)));
	return sq;
}
double returnHalf(double value)
{
	return value*0.5;
}
double multiply(double value1, double value2)
{
	return value1*value2;
}
```

Результаты

Среднее значение времени выполнения:

- SqOd . . . . . 35.69 с.
- Sqip . . . . . 9.59 с.

При автоматической оптимизации что-то может измениться в поведении программы. Почему?

```
bool func()
{
    int a(1000), b(0);

    for(int i = 0; i < 10000; i++)
    b = (a-i)/(a-i)*b;

    return false;
}
```

nemnyugin@parserplus.com