Повышение эффективности работы последовательного алгоритма

Марчевский И.К., Попов А.Ю.

МГТУ им. Н.Э. Баумана

Способы повышения эффективности работы программы

- Помимо использования параллелизма и применения высокопроизводительных вычислительных средств и технологий (OpenMP, MPI, CUDA) следует помнить, что большой ресурс для повышения эффективности работы алгоритма лежит *в оптимизации его последовательной версии*;
- однако важно придерживаться золотой середины, поскольку зачастую наиболее эффективно реализованный последовательный алгоритм может очень плохо распараллеливаться, а наиболее эффективно — наоборот, алгоритм, дающий не самую высокую производительность в последовательном режиме;
- ускорение s (speed-up) при применении p процессоров/узлов/ядер вычисляется по формуле

$$s = \frac{t_1}{t_p}, \tag{1}$$

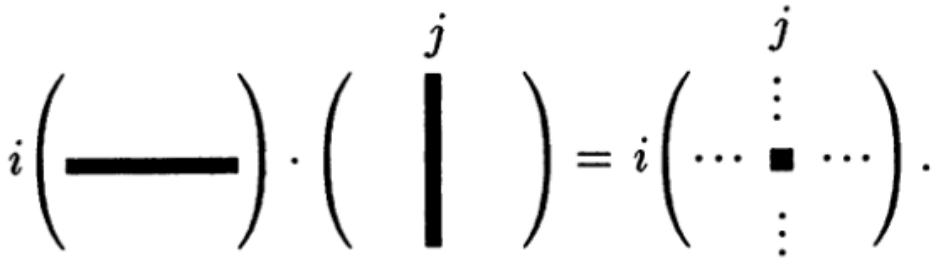
где t_p — время работы в параллельном режиме, а t_1 — в последовательном, причем либо того же алгоритма, либо (что более справедливо) наиболее эффективного последовательного, если такая информация есть.

Тестовая задача — умножение матриц

Матрицы квадратные: $A = (a_{ij}), B = (b_{ij}), i, j = \overline{1, n}$. Произведение матриц A и B — матрица $C = (c_{ij}), i, j = \overline{1, n}$ с элементами

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{2}$$

В простейшем варианте умножение выполняется по правилу «строка на столбец»:



Общее количество операций: $2n^3$ (n^3 умножений + n^3 сложений чисел с плавающей точкой — по аналогии с оценкой в LINPACK все арифметические операции учитываются равным образом).

Алгоритм и результаты работы

- Матрицы хранятся в виде одномерных массивов длины n^2, тип данных — double;
- матрицы содержат элементы $a_{ij} = \sin(i + j)$, $b_{ij} = \cos(i - j)$;
- вычислительное ядро алгоритма:

```
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j)
for (int k = 0; k < n; ++k)
C[i * n + j] += A[i * n + k] * B[k * n + j];
```

Пример

компилятор Microsoft Visual C++ Compiler 2022, процессор Intel Core i7-10700K (3.8 GHz), 32 GB оперативной памяти, $n = 1024$.
Время работы ≈ 18.224 с., производительность 1 ядра ≈ 0.118 GFlops.

Результаты работы и доработка алгоритма

1. Причина — режим “Debug”. Время работы в режиме “Release” — 1.740 с., производительность 1 ядра ≈ 1.234 GFlops.
2. Матрицы размера $n = 2048$. Время работы в режиме “Release” — 57.912 с., производительность 1 ядра ≈ 0.297 GFlops, ухудшение в 33.3 раза.
3. Причина — попадание данных в кэш при $n = 1024$. Для указанного процессора размер кэша уровня L3 — 16 MB; размер 1 матрицы при $n = 1024$ равен 8 MB, при $n = 1024$ — 32 MB.

Доработка алгоритма

Без изменения результата вычислений действия могут выполняться при различном порядке циклов:
$i, j, k \to j, k, i \to k, i, j \to j, i, k \to k, j, i \to i, k, j$.

Результаты работы для 6 вариантов циклов

Компилятор Microsoft Visual C++ 2022, $n = 1024$

Последовательность циклов	Время, с.	Производительность 1 ядра, GFlops
i, j, k	2.434	0.882
j, k, i	6.783	0.317
k, i, j	0.532	4.037
i, k, j	0.533	4.029
k, j, i	2.588	0.830
j, i, k	6.870	0.313

Наилучший и наихудший варианты отличаются по времени выполнения в $12,9$ раз.

Причина — в более эффективных вариантах в самом внутреннем цикле (выполняется наибольшее количество раз) происходит выборка данных из кэша подряд по строкам (данные считываются процессором из кэша, куда загружаются из оперативной памяти, но не по отдельному значению, а целым набором — машинными словами).

Результаты работы для 6 вариантов циклов

Наименее эффективный вариант

```
for (int j = 0; j < n; ++j)
for (int k = 0; k < n; ++k)
for (int i = 0; i < n; ++i)
C[i * n + j] += A[i * n + k] * B[k * n + j];
```

Самый внутренний цикл выполняется наибольшее ($\sim 10^6$) число раз, при этом для выборки элементов используются k и j — частые кэш-промахи.

Наиболее эффективный вариант

```
for (int k = 0; k < n; ++k)
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j)
C[i * n + j] += A[i * n + k] * B[k * n + j];
```

В самом внутреннем цикле элемент A_{ik} остается постоянным, элементы B_{kj} выбираются по строке — попадание в кэш. Работа с элементами C_{ij} также ведется по строке, эффективно используется кэш-память.

Результаты работы для 6 циклов

Компиляторы Intel C++ Compiler Classic 19.2 (1) и Intel C++ Compiler 2023 (2), $n = 1024$, стандартные настройки (оптимизация `O2`)

Посл-ность циклов	Время, с.		Произ-ть 1 ядра, GFlops	
	1	2	1	2
i, j, k	0.283	2.128	7.588	1.009
j, k, i	0.266	6.754	8.073	0.318
k, i, j	0.257	0.314	8.356	6.839
i, k, j	0.256	0.285	8.389	7.535
j, i, k	0.257	2.161	8.356	0.994
k, j, i	0.258	6.748	8.324	0.318

Вывод: современные эффективные компиляторы при создании машинного кода в некоторых случаях выполняют оптимизацию самостоятельно и организуют циклы таким образом, чтобы достигалась наибольшая производительность.

С дополнительными настройками (оптимизация `O3`, `Favor fast code`, Intel Processor-Specific Optimization — `Intel(R) AVX2`) наилучший результат — 0.067с., 32.052 GFlops/0.185с., 11.608 GFlops.

Использование двумерных массивов

Матрицы могут храниться в виде двумерных массивов.
Инициализация:

```
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j){
A[i][j] = sin(i + j); B[i][j] = cos(i - j); C[i][j] = 0.0;
}
```

Время расчета для $n = 1024$:

Хранение	Одномерное		Двумерное	
Компилятор	MS VC++	Intel	MS VC++	Intel
i, j, k	2.434	2.128	1.496	1.491
j, k, i	6.783	6.754	7.646	7.789
k, i, j	0.532	0.314	0.528	0.252
i, k, j	0.533	0.285	0.510	0.238
k, j, i	2.588	2.161	1.426	1.547
j, i, k	6.870	6.748	7.499	7.629

Умножение на транспонированную матрицу

С вычислительной точки зрения эффективнее умножать матрицу A на транспонированную B: $C = A \cdot B^{\mathrm{T}}$. При этом отдельной операции транспонирования B не выполняется, оно учитывается с помощью индексов:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk}. \tag{3}$$

Если необходимо умножить A на B, то B можно предварительно транспонировать (вычислительно не очень трудоемкая операция), а затем перемножить по формуле (3).

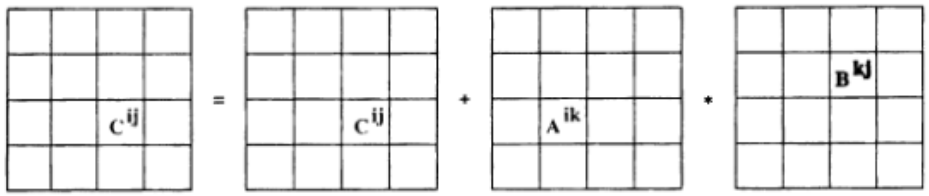
Результаты для компилятора Microsoft Visual C++ 2022, $n = 1024$:

Последовательность циклов	i, j, k	j, k, i	k, i, j	i, k, j	j, i, k	k, j, i
Время, с.	0.942	6.787	1.582	1.444	0.958	6.801

Лучший результат для компилятора Intel: 0.183с., 11.748 GFlops.

Блочное умножение матриц I

Более эффективным является алгоритм блочного умножения матриц, когда матрицы разбиваются на N блоков размера $s \times s$, которые умножаются с использованием кэш-памяти:



Блок C^{ij} вычисляется с использованием блоков матриц A и B, которые перемножаются между собой стандартным образом:

$$C^{ij} = \sum_{k=1}^{N} A^{ik} \cdot B^{kj}, \qquad i, j = \overline{1, N}, \tag{4}$$

$$A^{ik} \cdot B^{kj} = \sum_{r=1}^{s} (A^{ik})_{pr} \cdot (B^{kj})_{rs}, \qquad p, q = \overline{1, s}. \tag{5}$$

Блочное умножение матриц II

Оптимальный размер блока зависит от архитектуры и характеристик процессора.
Время расчета для $n = 2048$:

Размер блока	1	4	16	32	64	128
MS VC++	5.487	14.921	3.125	2.140	1.714	1.810
Intel	2.994	6.746	2.572	1.595	1.292	1.160

Вариант блочного умножения — прямое (без циклов) перемножение элементов для блоков с использованием статических переменных. Результат лучше по сравнению с исходным вариантом для того же размера блока за счет использования более быстрой (регистровой) памяти и отказа от массивов.

Время расчета для блока размера 4×4: MS VC++ — 2.361c., Intel C++ Compiler — 1.548c.

Алгоритм Штрассена I

Алгоритм блочного умножения матриц, использующий рекурсию — процедура вызывается для блоков размера $n/2$, а при достижении некоторого размера блока матрицы перемножаются напрямую. Матрицы A, B и C представляются в виде

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Алгоритм позволяет перемножить матрицы размера 2×2 за 7 умножений и 18 сложений. Тем самым, общая асимптотическая сложность алгоритма $n^{\log_2 7} \approx n^{2.81}$. Имеются другие алгоритмы, включая следующие:

- Pan, 1978 ($n \approx 2.796$);
- Schönhage, 1981 ($n \approx 2.522$);
- Coppersmith, Winograd, 1981 ($n \approx 2.496$), Coppersmith, Winograd, 1990 ($n \approx 2.3755$);
- Williams, 2013 ($n \approx 2.3729$).

Алгоритм Штрассена II

Однако на практике они дают выигрыш только на очень больших матрицах ($\sim 10^9$ или даже $\sim 10^{12}$ элементов).

Блоки матрицы C вычисляются по формулам

$$C_{11} = P_1 + P_2 - P_4 + P_6, \qquad C_{21} = P_6 + P_7,$$
$$C_{12} = P_4 + P_5, \qquad C_{22} = P_2 - P_3 + P_5 - P_7,$$

где

$$P_1 = (A_{12} - A_{22})(B_{21} + B_{22}), \qquad P_5 = A_{11}(B_{12} - B_{22}),$$
$$P_2 = (A_{11} + A_{22})(B_{11} + B_{22}), \qquad P_6 = A_{22}(B_{21} - B_{11}),$$
$$P_3 = (A_{11} - A_{21})(B_{11} + B_{12}), \qquad P_7 = (A_{21} + A_{22})B_{11},$$
$$P_4 = (A_{11} + A_{12})B_{22}.$$

Матрицы P_i — половинного размера.

Время расчета для $n = 2048$: MS VC++ — 1.386с., Intel C++ Compiler — 1.074с.

Использование библиотеки MKL I

`Math Kernel Library` — библиотека Intel, содержащая оптимизированные процедуры для выполнения научных вычислений.

- Содержит BLAS (Basic Linear Algebra Subprograms, базовые подпрограммы линейной алгебры), LAPACK (Linear Algebra PACKage, библиотека решения задач линейной алгебры), быстрое преобразование Фурье, решатели разреженных СЛАУ и др.;
- первая версия выпущена в 2003 году;
- является частью пакета Intel oneAPI Base Toolkit и распространяется свободно по лицензии Intel Simplified Software License;
- работает с компилятором Intel C++ Compiler.

Использование:

1. `#include "mkl.h"`
2. Настройки проекта – Intel Libraries for oneAPI – Intel® oneAPI Math Kernel Library (oneMKL) – Use oneMKL – Sequential (последовательный режим) либо Parallel (многопоточный режим).

Использование библиотеки MKL II

Умножение матриц:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, n, n,
                n, alpha, A, n, B, n, beta, C, n);
```

- `cblas` — интерфейс на языке Си к библиотеке BLAS;
- `d` — данные типа `double`;
- `gemm` — умножение матриц общего вида (General Matrix Multiplication);
- умножение по формуле $C := \alpha AB + \beta C$ (`alpha` $= 1.0$, `beta` $= 0.0$);
- `CblasRowMajor` — главный (внешний) цикл — по строкам, аналогично хранению массивов в C++;
- `CblasNoTrans`, `CblasTrans` — умножение исходной либо транспонированной матрицы;
- `n,n,n` — размеры матриц (в общем случае возможно умножение прямоугольных).

Использование библиотеки MKL III

Для последующей эффективной работы библиотека MKL требует предварительного выполнения функции для подготовки структур в памяти, настройки параметров и т.д. — достаточно на данных небольшого размера.

Время расчета и ускорение (8-ядерный процессор):

Режим	**2048**	**4096**	**8192**
Sequential	0.260	2.067	16.257
Parallel	0.044	0.380	3.283
Ускорение	5.91	5.44	4.95