**ЛАБОРАТОРНАЯ РАБОТА №7**

**ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ**

**Цели работы:**

1. Изучение SIMD-расширений архитектуры x86/x86-64.
2. Изучение способов использования SIMD-расширений в программах на языке Си.
3. Получение навыков использования SIMD-расширений.

**Задача:** основной задачей лабораторной работы является сравнение времени работы при векторизации вычислений и при использовании высокопроизводительных библиотек с обычным кодом с различными флагами компиляции.

В приложении 1 приведен полный листинг компилируемой программы. Правильность вычислений была проверена на матрице размера 4х4.

Первый вариант программы не содержит в себе никаких дополнительных оптимизаций. Для ускорения работы алгоритма использовались лишь различные флаги компиляции.

Второй вариант программы использует встроенные SIMD-функции компилятора. Данные функции встраивают в программу SIMD-расширения. Они обеспечивают возможность использования команд SIMD-расширений с помощью привычного синтаксиса для вызова С-функций вместо использования ассемблерного кода и работы с регистрами процессора.

С помощью библиотеки xmmintrin.h была переписана функция перемножения матриц:

1. void MultiplyMatrices(float \*A, float \*B, float \*resMatrix, const int size) { // A \* B = resMatrix
2. for (int i = 0; i < size; i++) {
3. float \*rowInResMatrix = resMatrix + i \* size;
4. for (int j = 0; j < size; j += 8)
5. **\_mm256\_storeu\_ps(rowInResMatrix + j, \_mm256\_setzero\_ps());**
6. for (int k = 0; k < size; ++k) {
7. const float \*columnInB = B + k \* size;
8. \_\_m256 a = \_mm256\_set1\_ps(A[i \* size + k]);
9. for (int j = 0; j < size; j += 8)
10. **\_mm256\_storeu\_ps(rowInResMatrix + j, \_mm256\_fmadd\_ps(a, \_mm256\_loadu\_ps(columnInB + j), \_mm256\_loadu\_ps(rowInResMatrix + j)));**
11. }
12. }
13. }

И так же был написан вариант с использованием высокопроизводительной библиотеки BLAS. В нем также была изменена функция умножения матриц:

1. void MultiplyMatrices(float \*A, float \*B, float \*resMatrix, const int size) {
2. cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, size, size, size, 1.0, A, size, B, size, 0.0, resMatrix, size);
3. }

Для каждого варианта программ были проведены замеры времени. Результаты замеров представлены в таблице:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Main,  sec | Main  -O2, sec | SIMD-vect,  sec | SIMD-vect -O2,  sec | BLAS-opt -O2,  sec |
| 205 | 34 | 39 | 10 | 0.6 |

**Заключение:** на основе полученных результатов видно, что применение метода векторизации и использование высокопроизводительных библиотек в разы увеличивает быстродействие программы.

Приложение 1. Полный листинг компилируемой программы.

1. #include <iostream>
2. #include <chrono>
4. #define N 2048
5. **#define M 10**
7. using namespace std;
9. void FillMatrix(float \*matrix, int size) {
10. **for (int i = 0; i < size; i++) {**
11. for (int j = 0; j < size; j++) {
12. if (i == j) {
13. matrix[i \* size + j] = size - i;
14. }
15. **else {**
16. matrix[i \* size + j] = 1;
17. }
18. }
19. }
20. **}**
22. void PrintMatrix(float \*matrix, int size) {
23. for (int i = 0; i < size; i++) {
24. for (int j = 0; j < size; j++) {
25. **cout << matrix[i \* size + j] << "\t";**
26. }
27. cout << endl;
28. }
29. cout << endl;
30. **}**
32. void TransposeMatrix(float \*srcMatrix, float \*resMatrix, const int size) {
33. for (int i = 0; i < size; i++) {
34. for (int j = 0; j < size; j++) {
35. **resMatrix[i \* size + j] = srcMatrix[j \* size + i];**
36. }
37. }
38. }
40. **void DivMatrixByScalar(float \*srcMatrix, float \*resMatrix, const float scalar, const int size) {**
41. for (int i = 0; i < size; i++) {
42. for (int j = 0; j < size; j++) {
43. resMatrix[i \* size + j] = srcMatrix[i \* size + j] / scalar;
44. }
45. **}**
46. }
48. float MaxRow(float \*srcMatrix, const int size) {
49. float max0 = INT\_MIN;
50. **for (int i = 0; i < size; i++) {**
51. float rowSum = 0;
52. float \*rowPtr = srcMatrix + i \* size;
53. for (int j = 0; j < size; j++) {
54. rowSum += rowPtr[j];
55. **}**
56. max0 = max(rowSum, max0);
57. }
58. return max0;
59. }
61. float MaxColumn(float \*srcMatrix, const int size) {
62. float max0 = INT\_MIN;
63. for (int j = 0; j < size; j++) {
64. float columnSum = 0;
65. **for (int i = 0; i < size; i++) {**
66. columnSum += srcMatrix[i \* size + j];
67. }
68. max0 = max(columnSum, max0);
69. }
70. **return max0;**
71. }
73. void FillUnitMatrix(float \*srcMatrix, const int size) {
74. memset(srcMatrix, 0, sizeof(float) \* size \* size);
75. **for (int i = 0; i < size; i++) {**
76. srcMatrix[i \* size + i] = 1;
77. }
78. }
80. **void MultiplyMatrices(float \*A, float \*B, float \*resMatrix, const int size) {**
81. for (int i = 0; i < size; i++) {
82. float \*rowInResMatrix = resMatrix + i \* size;
83. for (int j = 0; j < size; j++)
84. rowInResMatrix[j] = 0;
85. **for (int k = 0; k < size; k++) {**
86. const float \*columnInB = B + k \* size;
87. const float elInA = A[i \* size + k];
88. for (int j = 0; j < size; j++) {
89. rowInResMatrix[j] += elInA \* columnInB[j];
90. **}**
91. }
92. }
93. }
95. **void SubstructMatrices(float \*minuendMatr, float \*subtrahendMatr, float \*resMatrix, const int n) {**
96. for (int i = 0; i < n; i++) {
97. for (int j = 0; j < n; j++) {
98. resMatrix[i \* n + j] = minuendMatr[i \* n + j] - subtrahendMatr[i \* n + j];
99. }
100. **}**
101. }
103. void AddMatrix(float \*addendumMatr1, float \*addendumMatr2, float \*sum, const int n) {
104. for (int i = 0; i < n; i++) {
105. **for (int j = 0; j < n; j++) {**
106. sum[i \* n + j] = addendumMatr1[i \* n + j] + addendumMatr2[i \* n + j];
107. }
108. }
109. }
111. void InverseMatrix(float \*srcMatrix, float \*resMatrix, const int size, const int iterNumb) {
112. float \*bMatrix = new float[size \* size];
113. float \*srcTransp = new float[size \* size];
114. float \*identityMatrix = new float[size \* size];
115. **float \*rMatrix = new float[size \* size];**
116. float \*tmpMatrix = new float[size \* size];
117. TransposeMatrix(srcMatrix, srcTransp, size);
118. DivMatrixByScalar(srcTransp, bMatrix, MaxRow(srcMatrix, size) \* MaxColumn(srcMatrix, size), size);
119. FillUnitMatrix(identityMatrix, size);
120. **MultiplyMatrices(bMatrix, srcMatrix, tmpMatrix, size);**
121. SubstructMatrices(identityMatrix, tmpMatrix, rMatrix, size);
122. memset(resMatrix, 0, sizeof(float) \* size \* size);
123. AddMatrix(resMatrix, identityMatrix, resMatrix, size);
124. AddMatrix(resMatrix, rMatrix, resMatrix, size);
125. **float \*rPow = tmpMatrix;**
126. tmpMatrix = srcTransp;
127. memcpy(rPow, rMatrix, sizeof(float) \* size \* size);
128. for (int i = 2; i < iterNumb; i++) {
129. MultiplyMatrices(rMatrix, rPow, tmpMatrix, size);
130. **memcpy(rPow, tmpMatrix, sizeof(float) \* size \* size);**
131. AddMatrix(resMatrix, rPow, resMatrix, size);
132. }
133. MultiplyMatrices(resMatrix, bMatrix, tmpMatrix, size);
134. memcpy(resMatrix, tmpMatrix, sizeof(float) \* size \* size);
135. **delete rMatrix;**
136. delete rPow;
137. delete bMatrix;
138. delete tmpMatrix;
139. delete identityMatrix;
140. **}**
142. int main() {
143. float \*matrix = new float[N \* N];
144. FillMatrix(matrix, N);
145. **float \*matrixInversed = new float[N \* N];**
147. chrono::high\_resolution\_clock::time\_point start = chrono::high\_resolution\_clock::now();
148. InverseMatrix(matrix, matrixInversed, N, M);
149. chrono::high\_resolution\_clock::time\_point end = chrono::high\_resolution\_clock::now();
150. **cout << "\ntime = " << chrono::duration\_cast<chrono::seconds> (end - start).count() << " sec" << endl;**
151. delete matrix;
152. delete matrixInversed;
153. return 0;
154. }