Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ

по лабораторной работе №3

на тему

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ GPU

Студенты: Н.Г. Альхимович

К.А. Скалозуб

Преподаватель: Т.С. Жук

Минск 2024

**1 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

**1.1 Цель работы**

Изучить особенности разработки алгоритмов на GPU.

**1.2 Исходные данные к работе**

Задача: реализовать заданный по варианту аогритм на CPU и GPU. Сравнить производительность двух типов архитектур.

Обязательные требования:

1. Реализация CPU и GPU:

- CPU - эталон (качество кода не существенно);

- GPU - (каждая оптимизация - отдельное ядро, основанное на предыдущей реализации).

2. Задание = 1 ядро, исключения:

- при работе со сверхбольшими размерами матриц.

3. Сравнение времени работы в едином формате:

- формат: микро-, миллисекунды и т.п.;

- измерение времени на GPU через события ​в CUDA.

4. Сравнение результатов работы:

- полное поэлементное сравнения массива;

- вывод фрагмента на экран.

Допускается использование двух массивов: с входными и выходными данными. Дополнительные массивы запрещены инициализация случайными числами.

Обязательно должно быть 3 измерения времени:

1. На CPU.

2. На GPU (без учета копирования данных).

3. На CPU (с учетом копирования данных).

Опциональные условия:

1. Использование технологии `OpenCL` допустимо по разрешению преподавателя.

2. Инициализация массива через `cuRAND`.

3. Проверка ошибок выполнения `cudaError\_t`.

4. Работа со сверхбольшими объемами данных.

5. Невозможно вычислить за один вызов ядра.

6. Объем матрицы / массива больше объема видеопамяти.

Задачи:

1. Кратко описать проведенные эксперименты;

2. Выдвинуть предположения до начала экспериментов;

3. Провести эксперименты c изменением параметров;

4. Описать полученные результаты.

**2 ВЫПОЛНЕНИЕ РАБОТЫ**

В данной работе была проведена реализация алгоритма трансформации матрицы на CPU и GPU с целью сравнения их производительности. Для этого была написана программа на C++, использующая CUDA для параллельного вычисления на графическом процессоре.

Эксперименты:

1. Инициализация матрицы: исходная матрица заполняется случайными целыми числами в диапазоне от 0 до 99. Размер матрицы составляет 4096 на 4096 элементов.

2. Трансформация матрицы на CPU: для проверки производительности CPU реализована функция transformMatrix, которая выполняет преобразование матрицы, перемещая элементы в соответствии с заданным алгоритмом.

3. Трансформация матрицы на GPU: для GPU была разработана функция transformMatrixGPU, которая использует ядро CUDA transformMatrixKernel для выполнения аналогичной операции. В процессе работы производилось измерение времени выполнения ядра и общего времени, включая копирование данных между хостом и устройством.

4. Сравнение результатов: после выполнения преобразований на CPU и GPU результаты сравнивались с помощью функции compareMatrices, которая проверяет, совпадают ли элементы двух матриц.

5. Измерение времени: время выполнения измерялось с помощью стандартной библиотеки <chrono> для CPU и событий CUDA для GPU. Проводились три измерения времени: время выполнения на CPU, время выполнения на GPU (без учета передачи данных), время выполнения на CPU с учетом передачи данных.

Предположения до начала экспериментов:

1. GPU будет значительно быстрее CPU при выполнении задачи трансформации матрицы, особенно с увеличением размера входных данных.

2. Оптимизация кода для GPU через использование блоков и темплейтов позволит добиться лучшей производительности.

3. Возможные проблемы с памятью могут возникнуть при работе с очень большими матрицами, превышающими объем видеопамяти.

Результаты:

1. Время выполнения на CPU составило, в среднем, 6.35 секунд.

2. Время выполнения ядра на GPU составило 0.15 секунд.

3. Общее время выполнения на GPU (включая копирование данных) составило 0.8 секунд.

Сравнение производительности показало, что реализация на GPU была быстрее реализации на CPU в среднем в 0,6 раз.

Кроме того, все результаты трансформации матриц совпадали, что подтверждает корректность реализации обоих алгоритмов.

**3 ВЫВОДЫ**

В результате проведенной работы была достигнута цель: изучены особенности разработки и реализации алгоритмов на GPU.

Основные выводы:

1. Производительность: GPU не продемонстрировал значительно более высокую производительность по сравнению с CPU.

2. Корректность алгоритмов: результаты выполнения алгоритмов на обоих типах архитектур подтвердили свою корректность, так как выданные матрицы совпадали.

3. Оптимизация: использование параллельных вычислений на GPU позволяет значительно ускорить выполнение задач, что делает эту технологию предпочтительной для работы с большими объемами данных.

4. Проблемы с памятью: при увеличении размеров матриц необходимо учитывать ограничения видеопамяти, что может потребовать дополнительных оптимизаций или использования методов, позволяющих работать с данными, превышающими ее объем.

Таким образом, работа продемонстрировала важность понимания архитектур и особенностей работы с различными типами процессоров для эффективного выполнения вычислительных задач.

**ПРИЛОЖЕНИЕ А**

*(обязательное)*

Содержимое файла LP3.cpp:

1: #include <iostream>

2: #include <vector>

3: #include <chrono>

4: #include <iomanip>

5: #include <random>

6:

7: using namespace std;

8:

9: vector<vector<int>> fillMatrix(int rows, int cols) {

10: vector<vector<int>> matrix(rows, vector<int>(cols));

11:

12: random\_device rd;

13: mt19937 gen(rd());

14: uniform\_int\_distribution<> range(0, 99);

15:

16: for (int i = 0; i < rows; i++) {

17: for (int j = 0; j < cols; j++) {

18: matrix[i][j] = range(gen);

19: }

20: }

21:

22: return matrix;

23: }

24:

25: void printMatrix(const vector<vector<int>>& matrix) {

26: for (const auto& row : matrix) {

27: for (int elem : row) {

28: cout << elem << " ";

29: }

30: cout << "\n";

31: }

32:

33: cout << "\n";

34: }

35:

36: vector<vector<int>> transformMatrix(const vector<vector<int>>& matrix, int N, int M, int block\_size, int window\_size) {

37: vector<vector<int>> result(N, vector<int>(M, 0));

38:

39: for(int i = 0; i < N; i += block\_size) {

40: for(int j = 0; j < M; j += block\_size) {

41:

42: for(int iw = 0; iw < window\_size; ++iw) {

43: for(int jw = 0; jw < window\_size; ++jw) {

44: result[i + iw \* 2][j + jw \* 2] = matrix[i + iw][j + jw];

45: result[i + iw \* 2][j + jw \* 2 + 1] = matrix[i + iw][j + jw + window\_size];

46: result[i + iw \* 2 + 1][j + jw \* 2] = matrix[i + iw + window\_size][j + jw];

47: result[i + iw \* 2 + 1][j + jw \* 2 + 1] = matrix[i + iw + window\_size][j + jw + window\_size];

48: }

49: }

50: }

51: }

52:

53: return result;

54: }

55:

56: // bool compareMatrices(const vector<vector<int>>& matrix1, const vector<vector<int>>& matrix2) {

57: // if (matrix1.size() != matrix2.size() || matrix1[0].size() != matrix2[0].size())

58: // return false;

59: //

60: // for (size\_t i = 0; i < matrix1.size(); i++) {

61: // for (size\_t j = 0; j < matrix1[0].size(); j++) {

62: // if (matrix1[i][j] != matrix2[i][j]) {

63: // return false;

64: // }

65: // }

66: // }

67: //

68: // return true;

69: // }

70:

71: int main()

72: {

73: int N = 8;

74: int M = 8;

75: int window\_size = 2;

76: int block\_size = N / 2;

77:

78: // auto matrix = fillMatrix(N, M);

79: std::vector<std::vector<int>> matrix = {

80: {1, 2, 3, 4, 5, 6, 7, 8},

81: {9, 10, 11, 12, 13, 14, 15, 16},

82: {17, 18, 19, 20, 21, 22, 23, 24},

83: {25, 26, 27, 28, 29, 30, 31, 32},

84: {33, 34, 35, 36, 37, 38, 39, 40},

85: {41, 42, 43, 44, 45, 46, 47, 48},

86: {49, 50, 51, 52, 53, 54, 55, 56},

87: {57, 58, 59, 60, 61, 62, 63, 64}

88: };

89: printMatrix(matrix);

90:

91: cout << "Transformation (CPU) started." << endl;

92:

93: auto start = chrono::high\_resolution\_clock::now();

94: auto result = transformMatrix(matrix, N, M, block\_size, window\_size);

95: auto end = chrono::high\_resolution\_clock::now();

96: chrono::duration<double> duration = end - start;

97: cout << "Duration: " << duration.count() << " sec." << endl << endl;

98: printMatrix(result);

99:

100: cout << "Transformation (GPU) started." << endl;

101:

102: start = chrono::high\_resolution\_clock::now();

103: // auto result\_gpu = transformMatrix(matrix, N, M, block\_size, window\_size);

104: end = chrono::high\_resolution\_clock::now();

105: chrono::duration<double> duration\_gpu = end - start;

106: cout << "Duration: " << duration\_gpu.count() << " sec." << endl << endl;

107:

108: // cout << "GPU realization is faster than CPU in " << duration.count() / duration\_blocked\_vectorised.count() << endl;

109:

110: // if (compareMatrices(result, result\_gpu)) {

111: // cout << endl << "Results are the same" << endl;

112: // cout << "First matrix element: " << result[0][0] << "." << endl;

113: // }

114: // else

115: // cout << "Results are NOT the same." << endl;

116:

117: cout << endl << "Testing results:" << endl;

118: cout << left << setw(8) << "M" << setw(8) << "N"

119: << setw(19) << "Time CPU (s)" << setw(19) << "Time GPU (s)"

120: << endl;

121: cout << string(76, '-') << endl;

122:

123: cout << left << setw(8) << M << setw(8) << N

124: << setw(19) << duration.count()

125: << setw(19) << duration\_gpu.count()

126: << endl;

127:

128: return 0;

129: }

Содержимое файла LP3.cu:

1: #include <iostream>

2: #include <vector>

3: #include <chrono>

4: #include <iomanip>

5: #include <random>

6: #include <cuda\_runtime.h>

7:

8: using namespace std;

9:

10: float duration\_gpu\_kernel = 0;

11: float duration\_gpu\_total = 0;

12:

13: vector<vector<int>> fillMatrix(int rows, int cols) {

14: vector<vector<int>> matrix(rows, vector<int>(cols));

15: random\_device rd;

16: mt19937 gen(rd());

17: uniform\_int\_distribution<> range(0, 99);

18:

19: for (int i = 0; i < rows; i++) {

20: for (int j = 0; j < cols; j++) {

21: matrix[i][j] = range(gen);

22: }

23: }

24:

25: return matrix;

26: }

27:

28: void printMatrix(const vector<vector<int>>& matrix) {

29: for (const auto& row : matrix) {

30: for (int elem : row) {

31: cout << elem << " ";

32: }

33: cout << "\n";

34: }

35: cout << "\n";

36: }

37:

38: void checkCudaError(cudaError\_t error, const char \*message) {

39: if (error != cudaSuccess) {

40: cerr << "Error: " << message << " (" << cudaGetErrorString(error) << ")" << endl;

41: exit(EXIT\_FAILURE);

42: }

43: }

44:

45: vector<vector<int>> transformMatrix(const vector<vector<int>> &matrix, int N, int M, int block\_size, int window\_size) {

46: vector<vector<int>> result(N, vector<int>(M, 0));

47:

48: for(int i = 0; i < N; i += block\_size) {

49: for(int j = 0; j < M; j += block\_size) {

50:

51: for(int iw = 0; iw < window\_size; ++iw) {

52: for(int jw = 0; jw < window\_size; ++jw) {

53: result[i + iw \* 2][j + jw \* 2] = matrix[i + iw][j + jw];

54: result[i + iw \* 2][j + jw \* 2 + 1] = matrix[i + iw][j + jw + window\_size];

55: result[i + iw \* 2 + 1][j + jw \* 2] = matrix[i + iw + window\_size][j + jw];

56: result[i + iw \* 2 + 1][j + jw \* 2 + 1] = matrix[i + iw + window\_size][j + jw + window\_size];

57: }

58: }

59: }

60: }

61:

62: return result;

63: }

64:

65: \_\_global\_\_ void transformMatrixKernel(int \*d\_matrix, int \*d\_result, int N, int M, int block\_size, int window\_size) {

66: int block\_row = blockIdx.y \* block\_size;

67: int block\_col = blockIdx.x \* block\_size;

68:

69: int iw = threadIdx.y;

70: int jw = threadIdx.x;

71:

72: if (iw < window\_size && jw < window\_size) {

73: int src\_idx = (block\_row + iw) \* M + (block\_col + jw);

74: int dst\_idx = (block\_row + iw \* 2) \* M + (block\_col + jw \* 2);

75:

76: d\_result[dst\_idx] = d\_matrix[src\_idx];

77: d\_result[dst\_idx + 1] = d\_matrix[src\_idx + window\_size];

78: d\_result[dst\_idx + M] = d\_matrix[src\_idx + window\_size \* M];

79: d\_result[dst\_idx + M + 1] = d\_matrix[src\_idx + window\_size \* M + window\_size];

80: }

81: }

82:

83: vector<vector<int>> transformMatrixGPU(const vector<vector<int>> &matrix, int N, int M, int block\_size, int window\_size) {

84: int \*h\_matrix = new int[N \* M];

85: int \*h\_result = new int[N \* M];

86:

87: for (int i = 0; i < N; ++i)

88: for (int j = 0; j < M; ++j)

89: h\_matrix[i \* M + j] = matrix[i][j];

90:

91: int \*d\_matrix;

92: int \*d\_result;

93:

94: checkCudaError(cudaMalloc(&d\_matrix, N \* M \* sizeof(int)), "cudaMalloc d\_matrix");

95: checkCudaError(cudaMalloc(&d\_result, N \* M \* sizeof(int)), "cudaMalloc d\_result");

96:

97: cudaEvent\_t start\_event, stop\_event, start\_total\_event, stop\_total\_event;

98: cudaEventCreate(&start\_event);

99: cudaEventCreate(&stop\_event);

100: cudaEventCreate(&start\_total\_event);

101: cudaEventCreate(&stop\_total\_event);

102:

103: // замер времени (с учетом копирования данных)

104: cudaEventRecord(start\_total\_event, nullptr);

105:

106: checkCudaError(cudaMemcpy(d\_matrix, h\_matrix, N \* M \* sizeof(int), cudaMemcpyHostToDevice), "cudaMemcpy h\_matrix to d\_matrix");

107: checkCudaError(cudaMemset(d\_result, 0, N \* M \* sizeof(int)), "cudaMemset d\_result");

108:

109: dim3 block(32, 32);

110: dim3 grid((M + block\_size - 1) / block\_size, (N + block\_size - 1) / block\_size);

111:

112: // замер времени (без учета копирования данных)

113: cudaEventRecord(start\_event, nullptr);

114: transformMatrixKernel<<<grid, block>>>(d\_matrix, d\_result, N, M, block\_size, window\_size);

115: checkCudaError(cudaGetLastError(), "Kernel launch");

116:

117: cudaEventRecord(stop\_event, nullptr);

118: cudaEventSynchronize(stop\_event);

119:

120: cudaEventElapsedTime(&duration\_gpu\_kernel, start\_event, stop\_event);

121:

122: checkCudaError(cudaMemcpy(h\_result, d\_result, N \* M \* sizeof(int), cudaMemcpyDeviceToHost), "cudaMemcpy d\_result to h\_result");

123:

124: cudaEventRecord(stop\_total\_event, nullptr);

125: cudaEventSynchronize(stop\_total\_event);

126:

127: cudaEventElapsedTime(&duration\_gpu\_total, start\_total\_event, stop\_total\_event);

128:

129: vector<vector<int>> result(N, vector<int>(M));

130: for (int i = 0; i < N; ++i)

131: for (int j = 0; j < M; ++j)

132: result[i][j] = h\_result[i \* M + j];

133:

134: checkCudaError(cudaFree(d\_matrix), "cudaFree d\_matrix");

135: checkCudaError(cudaFree(d\_result), "cudaFree d\_result");

136:

137: delete[] h\_matrix;

138: delete[] h\_result;

139:

140: cudaEventDestroy(start\_event);

141: cudaEventDestroy(stop\_event);

142: cudaEventDestroy(start\_total\_event);

143: cudaEventDestroy(stop\_total\_event);

144:

145: return result;

146: }

147:

148: bool compareMatrices(const vector<vector<int>>& matrix1, const vector<vector<int>>& matrix2) {

149: if (matrix1.size() != matrix2.size() || matrix1[0].size() != matrix2[0].size())

150: return false;

151:

152: for (size\_t i = 0; i < matrix1.size(); i++) {

153: for (size\_t j = 0; j < matrix1[0].size(); j++) {

154: if (matrix1[i][j] != matrix2[i][j]) {

155: return false;

156: }

157: }

158: }

159:

160: return true;

161: }

162:

163: int main() {

164: int N = 4096;

165: int M = 4096;

166: int window\_size = 2;

167: int block\_size = N / 2;

168:

169: vector<vector<int>> matrix = fillMatrix(N, M);

170: cout << "Original Matrix:\n";

171: // printMatrix(matrix);

172:

173: cout << "Transformation (CPU) started." << endl;

174: auto start = chrono::high\_resolution\_clock::now();

175: auto result\_cpu = transformMatrix(matrix, N, M, block\_size, window\_size);

176: auto end = chrono::high\_resolution\_clock::now();

177: chrono::duration<double> duration\_cpu = end - start;

178: cout << "Duration (CPU): " << duration\_cpu.count() << " sec." << endl;

179: // printMatrix(result\_cpu);

180:

181: cout << "Transformation (GPU) started." << endl;

182: auto result\_gpu = transformMatrixGPU(matrix, N, M, block\_size, window\_size);

183: cout << "Kernel execution time (GPU): " << duration\_gpu\_kernel / 1000.0 << " sec." << endl;

184: cout << "Total execution time (GPU including data transfer): " << duration\_gpu\_total / 1000.0 << " sec." << endl;

185: // printMatrix(result\_gpu);

186:

187: cout << "GPU realisation is faster than CPU by a factor of " << duration\_cpu.count() / (duration\_gpu\_total / 1000.0) << endl;

188:

189: if (compareMatrices(result\_cpu, result\_gpu)) {

190: cout << endl << "Results are the same" << endl;

191: cout << "First matrix element: " << result\_cpu[0][0] << "." << endl;

192: }

193: else {

194: cout << "Results are NOT the same." << endl;

195: }

196:

197: cout << endl << "Testing results:" << endl;

198: cout << left << setw(8) << "M" << setw(8) << "N"

199: << setw(19) << "Time CPU (s)" << setw(29) << "Time GPU kernel-only (s)"

200: << setw(19) << "Time GPU (s)" << endl;

201: cout << string(76, '-') << endl;

202:

203: cout << left << setw(8) << M << setw(8) << N

204: << setw(19) << duration\_cpu.count()

205: << setw(29) << duration\_gpu\_kernel / 1000.0

206: << setw(19) << duration\_gpu\_total / 1000.0 << endl;

207:

208: return 0;

209: }