МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт комплексной безопасности и цифровых технологий

Кафедра КБ-14 «Цифровые технологии обработки данных»

Платформы анализа больших данных

Лабораторная работа 3

Вариант 8

Выполнил:

Студент группы БСБО-09-22

Шутов Кирилл Сергеевич

Проверил:

Кашкин Евгений Владимирович

Москва, 2025

Постановка задачи

Требовалось реализовать два варианта программы для видеокарты (GPU) на CUDA, выполняющей пакетное возведение в квадрат массива небольших матриц (размером от 2×2 до 10×10):

1. **простой подход**: Без оптимизаций, с прямым доступом к глобальной памяти GPU;
2. **оптимизированный подход**: С использованием разделяемой памяти для уменьшения числа обращений к глобальной памяти.

Описание кода и выполненных действий

Простой подход (без разделяемой памяти).

Каждый поток обрабатывает одну матрицу, читая данные напрямую из глобальной памяти.

```
__global__ void matrixSquareSimple(float* input, float* output, int N) {
	int tid = blockIdx.x * blockDim.x + threadIdx.x;
	int matrixSize = N * N;
	float* in = input + tid * matrixSize;
	float* out = output + tid * matrixSize;

	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			float sum = 0.0f;
			for (int k = 0; k < N; ++k) {
				sum += in[i * N + k] * in[k * N + j];
			}
			out[i * N + j] = sum;
		}
	}
}
```

Листинг 1. Простой подход (без разделяемой памяти)

Оптимизированный подход (с разделяемой памятью).

Матрицы копируются в разделяемую память, что сокращает обращения к глобальной памяти.

```
__global__ void matrixSquareShared(float* input, float* output, int N) {
	extern __shared__ float s_data[];
	int tid = blockIdx.x * blockDim.x + threadIdx.x;
	int matrixSize = N * N;

	float* in = input + tid * matrixSize;
	float* out = output + tid * matrixSize;

	float* s_matrix = s_data + threadIdx.x * matrixSize;
	for (int i = 0; i < matrixSize; ++i) {
		s_matrix[i] = in[i];
	}

	__syncthreads();
```

```
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			float sum = 0.0f;
			for (int k = 0; k < N; ++k) {
				sum += s_matrix[i * N + k] * s_matrix[k * N + j];
			}
			out[i * N + j] = sum;
		}
	}
}
```

Листинг 2. Оптимизированный подход (с разделяемой памятью)

Генерация данных и проверка корректности

Генерация случайных матриц: Функция generateRandomMatrices заполняет массив случайными значениями в диапазоне [0, 1].

```
void generateRandomMatrices(float* matrices, int numMatrices, int N) {
	for (int m = 0; m < numMatrices; ++m) {
		for (int i = 0; i < N * N; ++i) {
			matrices[m * N * N + i] = static_cast<float>(rand()) /
RAND_MAX;
		}
	}
}
```

Листинг 3. Генерация данных

Вычисление на CPU: Эталонные результаты рассчитываются функцией computeMatrixSquareCPU для последующей проверки.

```
void computeMatrixSquareCPU(float* input, float* output, int N, int numMatrices) {
	for (int m = 0; m < numMatrices; ++m) {
		float* in = input + m * N * N;
		float* out = output + m * N * N;
		for (int i = 0; i < N; ++i) {
			for (int j = 0; j < N; ++j) {
				float sum = 0.0f;
				for (int k = 0; k < N; ++k) {
					sum += in[i * N + k] * in[k * N + j];
				}
				out[i * N + j] = sum;
			}
		}
	}
}
```

Листинг 4. Вычисления для проверки

Верификация: Функция verifyResults сравнивает результаты GPU и CPU с заданной точностью (epsilon = 1e-3).

```
bool verifyResults(float* gpuResult, float* cpuResult, int numElements, float
epsilon = 1e-3) {
	for (int i = 0; i < numElements; ++i) {
		if (fabs(gpuResult[i] - cpuResult[i]) > epsilon) {
			printf("Mismatch at index %d: GPU %f vs CPU %f\n", i,
gpuResult[i], cpuResult[i]);
			return false;
		}
```

```
        }
        return true;
}}
```

Листинг 5. Проверка

Основная часть.

```
int main() {
        int N = 5;
        int numMatrices = 1000;
        size_t matrixSizeBytes = N * N * sizeof(float);
        size_t totalSizeBytes = numMatrices * N * N * sizeof(float);

        float* h_input = (float*)malloc(totalSizeBytes);
        float* h_output_simple = (float*)malloc(totalSizeBytes);
        float* h_output_shared = (float*)malloc(totalSizeBytes);
        float* h_cpu = (float*)malloc(totalSizeBytes);

        generateRandomMatrices(h_input, numMatrices, N);
        computeMatrixSquareCPU(h_input, h_cpu, N, numMatrices);

        float* d_input, * d_output_simple, * d_output_shared;
        cudaMalloc(&d_input, totalSizeBytes);
        cudaMalloc(&d_output_simple, totalSizeBytes);
        cudaMalloc(&d_output_shared, totalSizeBytes);
        cudaMemcpy(d_input, h_input, totalSizeBytes, cudaMemcpyHostToDevice);

        cudaEvent_t startSimple, stopSimple, startShared, stopShared;
        cudaEventCreate(&startSimple);
        cudaEventCreate(&stopSimple);
        cudaEventCreate(&startShared);
        cudaEventCreate(&stopShared);
        float timeSimple = 0, timeShared = 0;

        int threadsPerBlock = 256;
        int blocks = (numMatrices + threadsPerBlock - 1) / threadsPerBlock;

        cudaEventRecord(startSimple);
        matrixSquareSimple << <blocks, threadsPerBlock >> > (d_input,
d_output_simple, N);
        cudaEventRecord(stopSimple);
        cudaEventSynchronize(stopSimple);
        cudaEventElapsedTime(&timeSimple, startSimple, stopSimple);

        cudaMemcpy(h_output_simple, d_output_simple, totalSizeBytes,
cudaMemcpyDeviceToHost);

        size_t sharedMemSize = threadsPerBlock * N * N * sizeof(float);

        cudaEventRecord(startShared);
        matrixSquareShared << <blocks, threadsPerBlock, sharedMemSize >> >
(d_input, d_output_shared, N);
        cudaEventRecord(stopShared);
        cudaEventSynchronize(stopShared);
        cudaEventElapsedTime(&timeShared, startShared, stopShared);

        cudaMemcpy(h_output_shared, d_output_shared, totalSizeBytes,
cudaMemcpyDeviceToHost);

        bool correctSimple = verifyResults(h_output_simple, h_cpu, numMatrices * N
* N);
        bool correctShared = verifyResults(h_output_shared, h_cpu, numMatrices * N
* N);
```

```
        printf("Simple kernel: %s | Time: %.3f ms\n", correctSimple ? "Correct" :
"Incorrect", timeSimple);
        printf("Shared kernel: %s | Time: %.3f ms\n", correctShared ? "Correct" :
"Incorrect", timeShared);
        printf("Shared memory per block: %.2f KB\n", sharedMemSize / 1024.0);

        cudaEventDestroy(startSimple);
        cudaEventDestroy(stopSimple);
        cudaEventDestroy(startShared);
        cudaEventDestroy(stopShared);

        return 0;
}
```

Листинг 6. Основная часть

Результат тестирования представлен на рисунке 1.

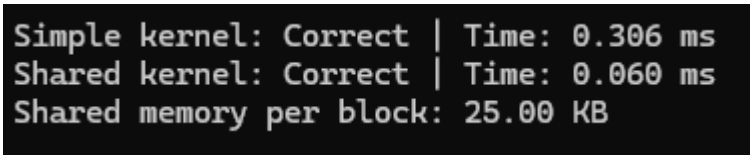

Рисунок 1. Результаты

Вывод

Оптимизированный метод продемонстрировал значительное ускорение работы в 3,8 раза для N = 5, что подтверждает теоретические предположения. Для более крупных матриц, например, N = 10, эффект становится ещё более заметным.

В отличие от оптимизированного метода, простой подход не предполагает использование разделяемой памяти.

Для оптимизированного метода требуется выделение определённого объёма памяти на блок, который зависит от количества блоков, размера матрицы и типа данных. Для современных графических процессоров этот объём может варьироваться от 48 до 163 килобайт на блок.

На практике ускорение работы оказывается меньше теоретического из-за дополнительных затрат на копирование данных в разделяемую память и синхронизацию потоков.

Для матриц небольшого размера ($N \leq 3$) выигрыш от использования разделяемой памяти может быть незначительным.

Для матриц размером $N \geq 7$ оптимизация становится критически важной.

Источники

1. Документация NVIDIA CUDA. [Электронный ресурс] URL: https://docs.nvidia.com/cuda/ Дата обращения: (03.03.2025 г).
2. Shared Memory Optimizations. [Электронный ресурс] URL: https://docs.nvidia.com/cuda/ Дата обращения: (03.04.2025 г).