Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

ЛАБОРАТОРНАЯ РАБОТА №5
По дисциплине: «Программирование графических процессоров»

Выполнили:
Студенты 3 курса группы ИП-211
Назаров Е.С
Капустин Т.Е
Фролова А.Е
Проверил:
Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2025

1 Задание.

Общая информация
- Устройство: GeForce GTX 1050
- Размер данных: N = 3, K = 4 (1<<2), общее количество элементов = 12 (но в выводе видно 24 элемента, что указывает на возможное расхождение)
- Размер блока: 256 потоков
- Количество блоков: рассчитывается динамически

1. Время выполнения ядер на GPU

Для измерения времени выполнения можно использовать команду:

nvprof --print-gpu-trace ./lab5

2. Пропускная способность при работе с глобальной памятью

2.1. Загрузка из глобальной памяти (gld_throughput)

Ядро copyKernel:
- Пропускная способность: 49.542 MB/s
- Количество транзакций: 6

Ядро gInit:
- Пропускная способность: 0 B/s (ядро не читает из глобальной памяти, только записывает)
- Количество транзакций: 0

2.2. Запись в глобальную память (gst_throughput)

Ядро copyKernel:
- Пропускная способность: 44.016 MB/s
- Количество транзакций: 3

Ядро gInit:
- Пропускная способность: 63.578 MB/s
- Количество транзакций: 6

3. Работа с локальной памятью

Для обоих ядер (copyKernel и gInit):
- Пропускная способность при загрузке из локальной памяти: 0 B/s
- Транзакции при загрузке из локальной памяти: 0
- Пропускная способность при записи в локальную память: 0 B/s
- Транзакции при записи в локальную память: 0

Это указывает на то, что в коде не используется локальная память (__shared__ или локальные переменные с большим потреблением).

4. Выводы

1. Ядро **gInit** выполняет только запись в глобальную память (инициализация массивов), что подтверждается нулевой пропускной способностью при чтении и значительной при записи.

2. Ядро **copyKernel** выполняет как чтение (49.542 MB/s), так и запись (44.016 MB/s) в глобальную память, что соответствует его функции копирования с переупорядочиванием данных.

3. Оптимизация работы с памятью:
- Низкие значения пропускной способности указывают на неэффективное использование памяти
- Можно улучшить производительность, оптимизировав доступ к памяти (объединение запросов, использование разделяемой памяти)
- Размер данных очень мал (всего 12-24 элемента), что делает измерения нерепрезентативными для реальных сценариев

4. Отсутствие работы с локальной памятью указывает на потенциал для оптимизации через использование разделяемой памяти (__shared__) для уменьшения количества обращений к глобальной памяти.

2 Задание.

"Эмуляция недостатка регистров и анализ использования локальной памяти в CUDA"

1. Цель работы
Исследовать влияние большого количества локальных переменных в CUDA-ядре на:
- использование регистров
- использование локальной памяти
- общую производительность ядра

2. Описание программы
Программа копирует и обрабатывает массив из 1024×1024 элементов типа float с помощью CUDA-ядра.

Ключевые особенности:
- Глобальные массивы размером 1024×1024 элементов
- Ядро использует 64 локальные переменные (reg0-reg63)
- Каждая переменная инициализируется на основе элемента массива
- Все переменные участвуют в вычислении результата

3. Результаты профилирования (nvprof)

Метрика	Значение	Интерпретация
Achieved Occupancy	0.834	Высокая загрузка SM (83.4%)
Local Load Transactions	0	Нет чтения из локальной памяти
Local Store Transactions	0	Нет записи в локальную память
Register Per Thread	64	Используется 64 регистра на поток

4. Анализ результатов

1. Использование регистров:
   - Ядро использует 64 регистра на поток
   - Это значительное количество, но ниже лимита (255 для большинства архитектур)
   - Компилятор смог разместить все переменные в регистрах

2. Локальная память:
   - Отсутствие операций с локальной памятью (все значения = 0)
   - Это означает, что несмотря на большое количество переменных, компилятор не был вынужден использовать локальную память

3. Эффективность:
   - Высокий achieved occupancy (0.834) показывает хорошую утилизацию вычислительных ресурсов
   - Отсутствие операций с локальной памятью способствует высокой производительности

5. Выводы

1. При использовании 64 локальных переменных:
   - Все переменные размещаются в регистрах
- Не происходит вытеснения в локальную память
- Сохраняется высокая эффективность выполнения

2. Для создания реального давления на регистры:
   - Необходимо увеличить количество переменных (>100)
   - Или использовать более сложные вычисления
   - Можно явно указать компилятору использовать локальную память с помощью `__local__`

3. Оптимизационные возможности:
   - Уменьшение количества регистров может увеличить occupancy
   - Для сложных ядер может потребоваться баланс между использованием регистров и локальной памяти

6. Приложение: ключевые команды профилирования

```
# Базовое профилирование
nvprof ./program

# Анализ регистров и локальной памяти
nvprof --metrics
achieved_occupancy,register_per_thread,local_load_transactions,local_store_transactions
./program

# Детальный анализ
nvprof --print-gpu-trace --metrics all ./program
```

7. Заключение

Работа демонстрирует, что современные компиляторы CUDA эффективно управляют регистрами, и простое объявление большого количества переменных не всегда приводит к их вытеснению в локальную память. Для реального исследования ограничений архитектуры требуются более сложные сценарии работы с переменными.

Приложение 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define N 3
#define K (1<<2)

global void gInit(float *a, float *b){
	int i = threadIdx.x + blockIdx.x*blockDim.x;
	if(i < N * K){
		a[i] = (float)i;
		b[i] = 0.0f;
	}
}

global void copyKernel(float *a, float *b) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N * K) {
      int groupIdx = idx / K;
      int offset = idx % K;
      b[offset * N + groupIdx] = a[idx];
   }
}

int main(){

	size_t size = K * N * sizeof(float);

	float *A = (float*)malloc(size);
	float *B = (float*)malloc(size);

	float *d_A, *d_B;

	int threadsPerBlock = 256;
	int blocksPerGrid = (N * K + threadsPerBlock - 1) / threadsPerBlock;


	cudaMalloc((void**)&d_A, N * K * sizeof(float));
	cudaMalloc((void**)&d_B, N * K * sizeof(float));

	gInit<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B);
	cudaDeviceSynchronize();

	copyKernel<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B);
	cudaDeviceSynchronize();

	cudaMemcpy(B, d_B, N * K * sizeof(float), cudaMemcpyDeviceToHost);
```

```
	cudaMemcpy(A, d_A, N * K * sizeof(float), cudaMemcpyDeviceToHost);

	printf("First 10 elements of the original and copied arrays:\n");
	for (int i = 0; i < N*K; i++) {
		printf("a[%d] = %f, b[%d] = %f\n", i, A[i], i, B[i]);
	}

	free(A);
	free(B);
	cudaFree(d_A);
	cudaFree(d_B);

	return 0;
}
```

Приложение 2.

```
#include <stdio.h>
#include <stdlib.h>

#define N 1024
#define K 1024

__global__ void copyKernelWithRegs(float *a, float *b, int N, int K) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Большое количество локальных переменных для создания давления на регистры
    float reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7;
    float reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15;
    float reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23;
    float reg24, reg25, reg26, reg27, reg28, reg29, reg30, reg31;
    float reg32, reg33, reg34, reg35, reg36, reg37, reg38, reg39;
    float reg40, reg41, reg42, reg43, reg44, reg45, reg46, reg47;
    float reg48, reg49, reg50, reg51, reg52, reg53, reg54, reg55;
    float reg56, reg57, reg58, reg59, reg60, reg61, reg62, reg63;

    if (idx < N * K) {
        // Инициализация локальных переменных
        reg0 = a[idx];
        reg1 = a[idx] + 1.0f; reg2 = a[idx] + 2.0f; reg3 = a[idx] + 3.0f;
        reg4 = a[idx] + 4.0f; reg5 = a[idx] + 5.0f; reg6 = a[idx] + 6.0f;
        reg7 = a[idx] + 7.0f; reg8 = a[idx] + 8.0f; reg9 = a[idx] + 9.0f;
        reg10 = a[idx] + 10.0f; reg11 = a[idx] + 11.0f; reg12 = a[idx] + 12.0f;
        reg13 = a[idx] + 13.0f; reg14 = a[idx] + 14.0f; reg15 = a[idx] + 15.0f;
        reg16 = a[idx] + 16.0f; reg17 = a[idx] + 17.0f; reg18 = a[idx] + 18.0f;
        reg19 = a[idx] + 19.0f; reg20 = a[idx] + 20.0f; reg21 = a[idx] + 21.0f;
        reg22 = a[idx] + 22.0f; reg23 = a[idx] + 23.0f; reg24 = a[idx] + 24.0f;
        reg25 = a[idx] + 25.0f; reg26 = a[idx] + 26.0f; reg27 = a[idx] + 27.0f;
        reg28 = a[idx] + 28.0f; reg29 = a[idx] + 29.0f; reg30 = a[idx] + 30.0f;
        reg31 = a[idx] + 31.0f; reg32 = a[idx] + 32.0f; reg33 = a[idx] + 33.0f;
        reg34 = a[idx] + 34.0f; reg35 = a[idx] + 35.0f; reg36 = a[idx] + 36.0f;
        reg37 = a[idx] + 37.0f; reg38 = a[idx] + 38.0f; reg39 = a[idx] + 39.0f;
        reg40 = a[idx] + 40.0f; reg41 = a[idx] + 41.0f; reg42 = a[idx] + 42.0f;
        reg43 = a[idx] + 43.0f; reg44 = a[idx] + 44.0f; reg45 = a[idx] + 45.0f;
        reg46 = a[idx] + 46.0f; reg47 = a[idx] + 47.0f; reg48 = a[idx] + 48.0f;
        reg49 = a[idx] + 49.0f; reg50 = a[idx] + 50.0f; reg51 = a[idx] + 51.0f;
        reg52 = a[idx] + 52.0f; reg53 = a[idx] + 53.0f; reg54 = a[idx] + 54.0f;
        reg55 = a[idx] + 55.0f; reg56 = a[idx] + 56.0f; reg57 = a[idx] + 57.0f;
        reg58 = a[idx] + 58.0f; reg59 = a[idx] + 59.0f; reg60 = a[idx] + 60.0f;
        reg61 = a[idx] + 61.0f; reg62 = a[idx] + 62.0f; reg63 = a[idx] + 63.0f;

        // Использование переменных для предотвращения оптимизации
        float sum = reg0 + reg1 + reg2 + reg3 + reg4 + reg5 + reg6 + reg7 +
                reg8 + reg9 + reg10 + reg11 + reg12 + reg13 + reg14 + reg15 +
```

```
                reg16 + reg17 + reg18 + reg19 + reg20 + reg21 + reg22 + reg23 +
                reg24 + reg25 + reg26 + reg27 + reg28 + reg29 + reg30 + reg31 +
                reg32 + reg33 + reg34 + reg35 + reg36 + reg37 + reg38 + reg39 +
                reg40 + reg41 + reg42 + reg43 + reg44 + reg45 + reg46 + reg47 +
                reg48 + reg49 + reg50 + reg51 + reg52 + reg53 + reg54 + reg55 +
                reg56 + reg57 + reg58 + reg59 + reg60 + reg61 + reg62 + reg63;

        b[idx] = sum;
    }
}

int main() {
    float *a, *b;
    float *d_a, *d_b;

    // Выделение памяти на хосте
    a = (float *)malloc(N * K * sizeof(float));
    b = (float *)malloc(N * K * sizeof(float));

    // Инициализация данных
    for (int i = 0; i < N * K; i++) {
        a[i] = (float)i;
    }

    // Выделение памяти на устройстве
    cudaMalloc((void **)&d_a, N * K * sizeof(float));
    cudaMalloc((void **)&d_b, N * K * sizeof(float));

    // Копирование данных на устройство
    cudaMemcpy(d_a, a, N * K * sizeof(float), cudaMemcpyHostToDevice);



    // Настройка параметров запуска ядра
    int threadsPerBlock = 256;
    int blocksPerGrid = (N * K + threadsPerBlock - 1) / threadsPerBlock;

    // Запуск ядра
    copyKernelWithRegs<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, N, K);

    // Проверка ошибок
    cudaDeviceSynchronize();
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess) {
        printf("CUDA error: %s\n", cudaGetErrorString(error));
    }

    // Копирование результатов обратно на хост
```

```
    cudaMemcpy(b, d_b, N * K * sizeof(float), cudaMemcpyDeviceToHost);

    // Освобождение памяти
    free(a);
    free(b);
    cudaFree(d_a);
    cudaFree(d_b);

    return 0;
}
```