Лабораторная работа № 8
Оптимизация работы с глобальной памятью для видеокарт с поддержкой технологии CUDA

Цель работы: научиться оптимизировать работу с глобальной памятью путем использования разделяемой памяти на примере задачи умножения квадратных матриц.

Как уже было отмечено ранее, работа с памятью является одним из краеугольных камней при разработке эффективных программ с использованием технологии CUDA. Существует ряд алгоритмов, требующих интенсивных обменов с памятью (одним из них является рассматриваемый в данной работе алгоритм умножения матриц). Для них реальная производительность видеокарты лимитируется не пиковой производительностью вычислительных блоков, а пропускной способностью памяти, поэтому повышение скорости обработки возможно путем ряда оптимизаций.

Глобальная память (англ. global memory) современных видеокарт имеет объем порядка нескольких гигабайт, однако характеризуется достаточно большой задержкой доступа (несколько сотен тактов), обращения к ней разрешены для всех потоков без исключения. При обработке информации можно использовать разделяемую память (англ. shared memory), время обращения к которой составляет единицы тактов и обычно несущественно по сравнению с временем вычислений. Однако разделяемая память характеризуется малым объемом (на данный момент 48 КБ для видеокарт семейств 4xx и старше) и доступна только для потоков одного блока. Если обработку исходного большого блока данных можно разбить на малые порции, не превышающие объема разделяемой памяти, то скорость обработки может быть существенно увеличена.

Для размещения переменной в глобальной памяти используется модификатор доступа `__device__`, либо функцию `cudaMalloc()`, для размещения в разделяемой памяти – `__shared__`. Пример:

```
  __shared__  float AA[S][S]; // Массив в разделяемой памяти
__device__ float A[N][N];  // Массив в глобальной памяти
```

Для копирования данных между глобальной и разделяемой памятью не требуется дополнительных действий со стороны программиста:

```
AA[i][j] = A[i0+i][j0+j];
```

При загрузке временных данных в буфер, расположенный в разделяемой памяти, конфигурация запуска CUDA-ядра обычно выбирается такой, чтобы каждый поток блока копировал одно (реже – несколько) значений. Пример копирования части одномерного массива в разделяемую память из глобальной приведен ниже:

```
const int N = 1024;          // Размер массива
const int S = 512;           // Размер порции данных в разделяемой памяти

__device__ float a[N];       // Массив в глобальной памяти

__global__ void CopyKernel(float *src)
{
       int x = threadIdx.x;
       int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
        __shared__ float dst[S];      // Массив в разделяемой памяти

        dst[x] = src[i];

        __syncthreads();

        // Обработка загруженной порции данных...
}

void CopyToShared()
{
        CopyKernel<<<dim3(N/S), dim3(S)>>>(a);
}
```

В приведенном примере исходный массив `a` размером $N = 1024$ элемента, расположенный в глобальной памяти, копируется в разделяемую память порциями по $S = 512$ элементов. При этом необходимо тщательно следить за вычислением индексов копируемых элементов. После завершения копирования необходима обязательная барьерная синхронизация потоков с использованием функции

```
__syncthreads();
```

чтобы убедиться в том, что перед началом обработки скопирована вся порция данных, а не только 32 элемента, скопированных потоками текущего WARP'а.

Если обращения в глобальную память из различных потоков идут по смежным адресам, они, при соблюдении ряда дополнительных условий, могут быть сгруппированы (англ. coalescing), что существенно (до 16 раз) повышает пропускную способность памяти.

Алгоритм умножения квадратных матриц размера $N \times N$ укрупненно можно представить в виде следующей последовательности шагов:

1. Копирование исходных матриц из оперативной памяти в глобальную память видеокарты.
2. Выполнение CUDA-ядра, реализующего умножение.
3. Копирование результирующей матрицы из глобальной памяти видеокарты в оперативную память.

В простейшем случае при выполнении умножения «в лоб» без использования разделяемой памяти каждый поток реализует вычисление одного элемента результирующей матрицы $C = AB$ по формуле

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

Значения индексов *i* и *j* определяются исходя из конфигурации запуска (число потоков в блоке, число потоков в сетке). Рекомендуемая конфигурация запуска:

```
MatrMulKernel<<<dim3(N/S, N/S), dim3(S, S)>>>();
```

При подобном умножении в глобальную память производится $2N^3$ обращений на чтение. Снизить число обращений к глобальной памяти можно при помощи буферизации *j*-й строки матрицы *A* или столбца матрицы *B* во временной переменной *t*, хранящейся в разделяемой памяти видеокарты. При этом умножение может быть реализовано по формулам

$$c_{ij} = \sum_{k=1}^{N} a_{ik} t_k$$

или

$$c_{ij} = \sum_{k=1}^{N} t_k b_{kj}$$

в зависимости от выбранного способа. Конфигурацию запуска потоков в блоке рекомендуется выбрать такой, чтобы в каждом потоке производилось вычисление одного элемента результирующей матрицы *C*, а сами элементы были сгруппированы в строку или столбец в зависимости от способа:

```
MatrMulBuffKernel<<<dim3(N), dim3(N)>>>();
```

При этом следует помнить, что для современных видеокарт максимальное число потоков в блоке ограничено значением 1024, что может быть ограничением на размер обрабатываемых матриц и должно быть учтено в коде CUDA-ядра.

При подобной буферизации производится $N^3 + N^2$ обращений к глобальной памяти на чтение и N^3 обращений к разделяемой памяти. Дополнительно снизить число обращений к глобальной памяти можно с использованием блочного подхода, при котором результат рассчитывается блоками размером $S \times S$ элементов. Каждый подобный блок C'_{xy} результирующей матрицы $C = \begin{Vmatrix} C'_{11} & C'_{12} & \dots \\ C'_{21} & C'_{22} & \dots \\ \dots & \dots & \dots \end{Vmatrix}$ обрабатывается группой потоков, входящих в состав блока, что соответствует конфигурации запуска

```
MatrMulBlockKernel<<<dim3(N/S, N/S), dim3(S, S)>>>();
```

При этом потоки блока поочередно загружают в разделяемую память блоки A'_{xz} и B'_{zy} матриц *A* и *B* (каждый блок также имеет размер $S \times S$ элементов). По окончании загрузки подматриц каждый поток блока считает свой элемент блока C'_{xy}

$$c'_{ij} = c_{x+i,\,y+j} = \sum_{k=1}^{S} a'_{ik} b'_{kj} = \sum_{k=1}^{S} a_{x+i,\,k} b_{k,\,y+j},$$

обращаясь к закэшированным элементам блоков A'_{xz} и B'_{zy}. Затем производится загрузка следующих блоков для $z = z + 1$. По завершении обработки всех подматриц A'_{xz} и B'_{zy}, $z = \overline{1, \frac{N}{S}}$ результирующие значения элементов c'_{ij} записываются в глобальную память (каждый поток блока считает одно подобное значение).

Данный подход требует $2N^2$ обращений к глобальной памяти на чтение и $2N^3$ обращений к разделяемой памяти.

При подобной обработке в зависимости от способа определения индексов *i*, *j*, *x* и *y* в коде потоков (зависят от координат потока в блоке и блока в сетке) могут быть нарушены условия coalescing-обращений к глобальной памяти, на что необходимо обратить внимание при разработке кода CUDA-ядра. При необходимости можно поменять индексы местами, контролируя корректность получаемого результата и общее время выполнения.

Для использования потенциала параллелизма на уровне команд (англ. Instruction Level Parallelism, ILP) код циклов умножения матриц можно раскрутить на некоторое количество итераций (2, 4, 8, …), контролируя корректность результата и время выполнения умножения.

Задание.

1. Разработать CUDA-ядра для вариантов умножения матриц «в лоб» (без оптимизации), с кэширование строки, столбца и для блочного умножения.
2. Разработать функции, реализующие копирование исходных матриц в глобальную память видеокарты, вызов соответствующего CUDA-ядра и обратное копирование результата.
3. Реализовать CPU-аналог умножения матриц (можно взять из предыдущей работы).
4. Организовать измерение времени выполнения умножения матриц для всех реализованных способов.
5. Реализовать подпрограмму, проверяющую корректность выполнения умножения путем сопоставления двух результирующих матриц. Убедиться в совпадении результатов для всех способов реализации умножения в независимости от выполненных оптимизаций. С использованием функции `cudaGetLastError()` убедиться в отсутствии ошибок.
6. Измерить время выполнения разработанных CUDA-ядер, выигрыш по сравнению с реализацией на процессоре и производительность при умножении матриц различного размера (для простоты размер N взять равным степени двойки). При подсчете производительности считать, что в наиболее вложенном цикле выполняется 2 операции с плавающей точкой (умножение и сложение), общее число операций с плавающей точкой составояет $2N^3$, а производительность $P = \frac{2N^3}{t}$, где t – время выполнения действия. Построить графики зависимости:
   - времени выполнения умножения и производительности от размера блока для матриц различного размера для каждого из алгоритмов;
   - времени выполнения и производительности от степени раскрутки цикла для выбранных значений N и S для каждого из алгоритмов;
   - времени выполнения и производительности для алгоритмов с кэшированием строки и столбца в зависимости от конфигурации запуска (числа потоков в блоке).
7. Сформулировать выводы и рекомендации о наилучшем способе умножения матриц с использованием GPU и параметрах запуска его CUDA-ядра.

Содержание отчета.

1. Титульный лист
2. Цель работы
3. Задание
4. Листинги программы
5. Результаты измерения времени выполнения, производительности и выигрыша в скорости. Соответствующий графический материал.
6. Выводы