Лабораторная работа 4

АВП, БГУИР 01/03/2023

1 Введение

1.1 Метод Монте-Карло(ММК) для подсчета площади

В случае, если для подсчета величины или моделирования процесса использование аналитических методов не представляется возможным или вычислительно сложным, используется ММК. Применительно к задаче подсчета площади(и ее обобщения на объем) он выглядит так:

$$S = \frac{S_0}{N} \sum^{N} \begin{cases} 0 & \text{, если P(x,y) вне искомой фигуры} \\ 1 & \text{, если P(x,y) внутри искомой фигуры} \end{cases}, \tag{1}$$

где N - число итераций в ММК, $P(x, y)$ - точка, выбранная с помощью случайного равномерного распределния из S_0, S_0 - площадь ограничивающего прямоугольника.

1.2 Реализация ММК для окружности на CPU

Для примера рассмотрим применение ММК для подсчета площади круга радиуса 7 с центром в начале координат(см. Рис. 1)

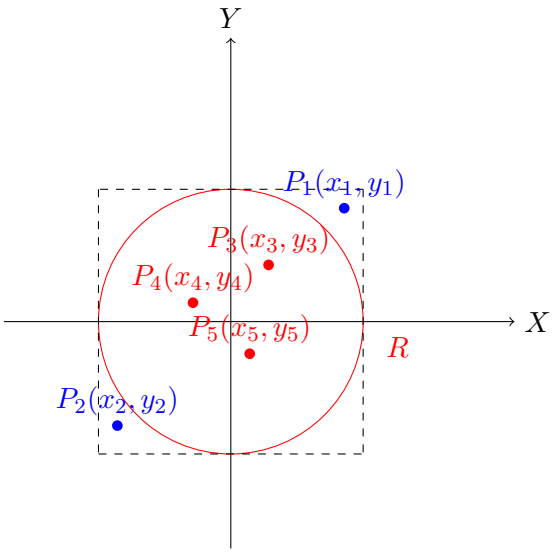


Рис. 1: Площадь круга.

Для процессора реализация в виде псевдокода может иметь следующий вид:

Algorithm 1 Псевдокод ММК для CPU

$S_0 \leftarrow 14 * 14$ ▷ Сторона ограничивающего прямоугольника в два раза больше радиуса
$R \leftarrow 7$ ▷ Радиус круга
$M \leftarrow 0$
$N_0 \leftarrow 10000000$ ▷ Чем больше, тем точнее
$N \leftarrow N_0$
while $N \neq 0$ do
  $x \leftarrow randomfrom[-7; 7]$
  $y \leftarrow randomfrom[-7; 7]$
  if $(x^2 + y^2 \leq R^2)$ then
    $M \leftarrow M + 1$
  end if
  $N \leftarrow N - 1$
end while
$S \leftarrow S_0 * \frac{M}{N_0}$

Для достижения большей точности необходимо увеличивать число N, что ведет к очевидному минусу данной реализации - увеличивающегося времени. Одним из решений является перенос вычислений и их параллелизация на GPU.

2 Технология CUDA

2.1 Реализация ММК для окружности на GPU

Основной единицей выполнения является kernel - ядро. Ядро выполняется паралелльно сразу на нескольких исполнителях. Исполнители организованы в виде логической сетки(grid) с блоками (block) и потоками(thread) внутри. Один kernel выполняется сразу на 32(warp) потоках, что позволяет ускорить вычисления. Применительно к этой лабораторной работе ядро будет выглядеть так:

```
__global__ void computeArea(size_t *accumulator, size_t numThreads,
                            size_t numAttemps, double R) {
  size_t threadId = blockIdx.x * blockDim.x + threadIdx.x
  // Check of thread boundary condition
  if (threadId >= numThreads) {
    return;
  }
  // Initialization of random numbers generator
  curandState localState;
  curand_init(1234, threadId, 0, &localState);

  size_t numInside = 0;
  for (size_t i = 0; i < numAttemps; ++i) {
    float x = curand_uniform(&localState);
    float y = curand_uniform(&localState);
    x = (x - 0.5) * 2 * R;
```

```
    y = (y - 0.5) * 2 * R;
    if (x * x + y * y <= R * R) {
      ++numInside;
    }
  }
  // Store of computation results
  accumulator[threadId] = numInside;
}

int main(...) {
    ...
    CUDA_CALL(cudaMalloc((void **)&deviceNumPointsInside, deviceMemorySize)); // Allocation of memory on GPU side
    CUDA_CALL(cudaMemset(deviceNumPointsInside, 0, deviceMemorySize));        // Initialization of memory

    const dim3 blockDimensions{BLOCK_SIZE};     // Configuration of block size
    const dim3 gridDimensions{numBlocks};       // Configuration of grid
    computeArea<<<gridDimensions, blockDimensions>>>(
        deviceNumPointsInside, numThreads, ATTEMPTS_PER_THREAD, R);           // Invokation of GPU code
    CUDA_CALL(cudaDeviceSynchronize());         // Synchronization of execution

    std::vector<size_t> hostPointsInside;
    hostPointsInside.resize(numThreads);
    CUDA_CALL(cudaMemcpy(hostPointsInside.data(), deviceNumPointsInside,
                         deviceMemorySize,
                         cudaMemcpyKind::cudaMemcpyDeviceToHost));            // Transfer of execution results to CPU
    CUDA_CALL(cudaFree(deviceNumPointsInside));
    const auto totalPointsInside =                                            // Aggregation of results across threads
    std::accumulate(hostPointsInside.begin(), hostPointsInside.end(), 0ll);
    ...
}
```

Листинг 1: Реализация на GPU

Полный исходный код примера можно будет найти в sample_circle.cu и собрать с помощью *nvcc sample_circle.cu −O2 −o mmk*

2.2 Аггрегация параллельных результатов на GPU

В предыдущей реализации часть вычислений все еще проводилась на CPU, в частности суммирование результатов выполнения каждого GPU потока. Причиной этого является параллельное выполнение всех потоков, что может привести к гонкам и неправильному результату. Одним из решений будет использование атомных операций не приводящих к гонкам. *atomicAdd* - операция атомарного сложения. Во время ее выполнения она не позволяет другим потокам оперировать над целевой ячейкой памяти, тем самым исключая возможность гонок.

```
using AccumulatorType = unsigned int;
__global__ void computeArea(size_t *AccumulatorType, size_t numThreads,
                            size_t numAttemps, double R) {
    ...
    atomicAdd(accumulator, numInside);
}

int main(...) {
    ...
    AccumulatorType *deviceNumPointsInside = nullptr;
    size_t deviceMemorySize = sizeof(AccumulatorType);
    ...
    AccumulatorType totalPointsInside;
    CUDA_CALL(cudaMemcpy(&totalPointsInside, deviceNumPointsInside,
                         deviceMemorySize,
                         cudaMemcpyKind::cudaMemcpyDeviceToHost));
    ...
}
```

Листинг 2: Реализация с использованием атомарных операций

Другим методом является реализация паттерна *Parallelreduce*. В его основе лежит идея о том, что если разбить операцию на серию непересекающихся подобных операций, то можно избежать гонок и проблем с синхронизацией. Схематически это изображено на рисунке 2.

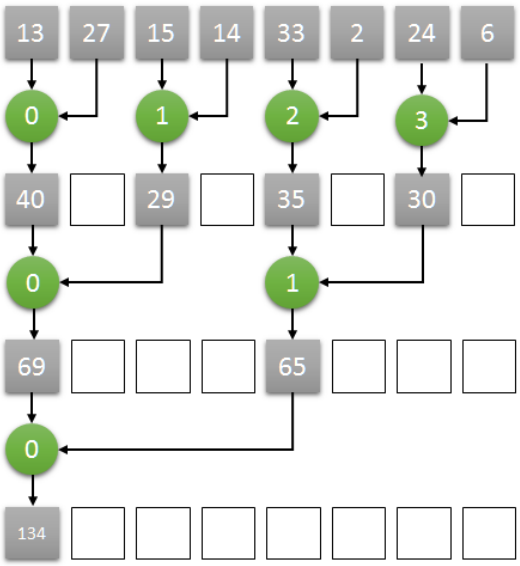


Рис. 2: Parallel reduce.

Для хранения промежуточных результатов необходима память. Для этого можно использовать глобальную, но для достижения большей производительности можно использовать разделяемую(*shared*) память. Разделяемая память общая для потоков одного блока и при этом гораздо быстрее глобальной. К минусам можно отнести необходимость синхронизации с помощью примитива барьерной синхронизации - $\_\_syncthreads()$. Вызов данной функции блокирует выполнение потоков до тех пор, пока все они не вызовут данную функ-

цию (см. Рис. 3).

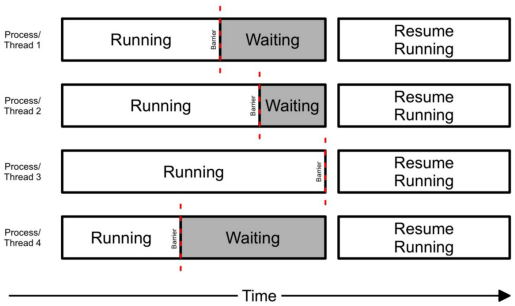


Рис. 3: Барьерная синхронизация

```
using AccumulatorType = unsigned int;
__global__ void computeArea(size_t *AccumulatorType, size_t numThreads,
                            size_t numAttemps, double R) {
    ...
  extern  __shared__ AccumulatorType sharedAccum[];
  sharedAccum[threadIdx.x] = numInside; // Load local result to shared memory
  __syncthreads();  // Wait for all threads

  size_t stride = blockDim.x / 2;
  while (stride > 0) {
    if (threadIdx.x >= stride) {
      return;
    }
    sharedAccum[threadIdx.x] += sharedAccum[threadIdx.x + stride];
    stride /= 2;
    __syncthreads();
  }
  const auto result = sharedAccum[threadIdx.x]; // In the end only thread with
    threadIdx.x remains and stores result of computation
}

int main(...) {
    ...
    size_t sharedMemorySize = BLOCK_SIZE * sizeof(AccumulatorType);
  computeArea<<<gridDimensions, blockDimensions, sharedMemorySize>>>(
      deviceNumPointsInside, numThreads, ATTEMPTS_PER_THREAD, R); // Invokation
    with dynamic size of shared memory
    ...
}
```

Листинг 3: Реализация parallel reduce

3 Задание

3.1 Условие

Для выполнения потребуется наличие видеокарты. Если в бригаде есть видеокарта NVIDIA - работа выполняется на ней с помощью технологии CUDA. Для видеокарт от AMD, Intel, Apple - OpenCL. Язык выполнения - C/C++.

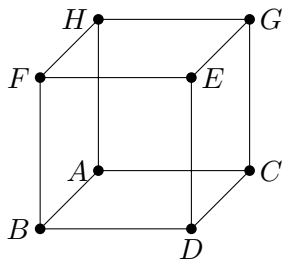
1. С помощью выбранной технологии вывести название используемого устройства(GPU), его производителя, количество доступной памяти(глобальной и разделяемой), compute capability.
2. Написать программу высчитывающую объем фигуры по методу ММК. Фигура выбирается согласно варианту.
3. Написать программу высчитывающую объем той же фигуры используя ММК, GPU и соответствующую технологию. Метод аггрегации вычислений одного потока выбрать соответсвенно варианту:

    - Для вариантов 1, 4: аггрегация на уровне блоков - `parallel reduce`, на уровне сетки - `parallel reduce`.
    - Для вариантов 2, 5: аггрегация на уровне блоков - `parallel reduce`, на уровне сетки - `atomicAdd`.
    - Для вариантов 3, 6: аггрегация на уровне блоков - `atomicAdd`, на уровне сетки - `parallel reduce`.

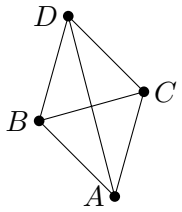
4. Используя число N не менее 100 000 измерить время выполнения реализации на CPU и GPU. Подобрать число N так, чтобы GPU реализация выполнялась не менее 1с, а реализация на CPU не более 10с.
5. Вывести на экран вычисленный объем с помощью ММК на CPU, объем полученный с помощью ММК на GPU, а также время выполнения программ из пункта 2 и 3. Также посчитать и сравнить ответ с полученный аналитическим путем(a.k.a. посчитанный руками).
6. Необязательное задание(на доп. балл): реализовать пункт 3 с использованием тензорных ядер.

3.2 Варианты

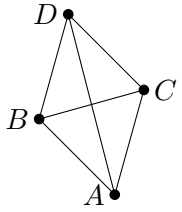
1. Для точек $A(-1,-1,-1)$, $B(-1,1,-1)$, $C(1,-1,-1)$, $E(1,1,1)$, $F(-1,1,1)$, $G(1,-1,1)$ посчитать объем фигуры ограниченной плоскостями $\triangle ABC$, $\triangle ABF$, $\triangle ACG$, $\triangle FEG$, $\triangle EGC$, $\triangle EFD$



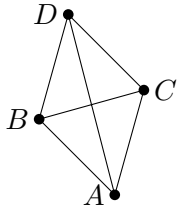
2. Для точек $A(0, 0, -1)$, $B(-1, 0, 0)$, $C(0, -1, 0)$, $D(1, 1, 1)$ посчитать объем фигуры ограниченной плоскостями $\triangle ABC$, $\triangle ABD$, $\triangle BCD$, $\triangle CAD$



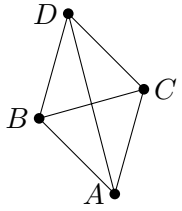
3. Для точек $A(0, 0, -0.7)$, $B(-1.5, 0, 0)$, $C(0, -0.2, 0)$, $D(0.3, 0.4, 0.5)$ посчитать объем фигуры ограниченной плоскостями $\triangle ABC$, $\triangle ABD$, $\triangle BCD$, $\triangle CAD$



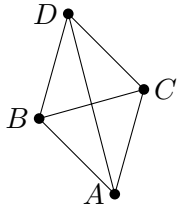
4. Для точек $A(0.4, 0.5, -0.05)$, $B(0.2, -1, -0.2)$, $C(-2, -0.3, -0.5)$, $D(-0.1, 0.05, 0.3)$ посчитать объем фигуры ограниченной плоскостями $\triangle ABC$, $\triangle ABD$, $\triangle BCD$, $\triangle CAD$



5. Для точек $A(0.6, 0.7, -0.1)$, $B(0.1, -1.3, -0.27)$, $C(-2.1, 0.4, -0.5)$, $D(0.3, 0.1, 0.4)$ посчитать объем фигуры ограниченной плоскостями $\triangle ABC$, $\triangle ABD$, $\triangle BCD$, $\triangle CAD$



6. Для точек $A(0.2, 0.2, -0.13)$, $B(0.1, -0.83, -0.12)$, $C(-1.4, 0.2, -0.8)$, $D(-0.2, -0.1, 0.7)$ посчитать объем фигуры ограниченной плоскостями $\triangle ABC$, $\triangle ABD$, $\triangle BCD$, $\triangle CAD$



3.3 Требования и допущения

- Точка $O(0, 0, 0)$ всегда лежит внутри фигуры;
- Фигура всегда является `выпуклым` многогранником;

- В коде программы явно указаны входные точки и плоскости;
- При изменении координат точек программа должна вести себя корректно(при соблюдении предыдущих условий);
- Для вычислений использовать тип данных как минимум `float`. Для доп. задания можно использовать `half`;
- Разница в точности для CPU и GPU реализации не больше 0.001. Разница между ММК и аналитическим - 0.1;
- Для GPU должна использоваться разделяемая память;
- Для CPU измерение должно проводиться с помощью $high - precision\, clock$, для GPU - с помощью событий;
- Реализация на GPU должна использовать размер блока в пределах от 32 до максимально допустимого вашей GPU. Программа должна поддерживать смену размера блока и по возможности вести себя валидно.

3.4 Контрольные вопросы

- Какая у Вас видеокарта? Какой у нее compute capability? Сколько в ней CUDA ядер/Compute Units/X^e ядер?
- Какие ограничения на Grid/Block у Вашей видеокарты? Какую форму и размерность они имеют?
- Какова иерархия памяти в программной модели CUDA? Какие типы памяти Вы использовали в лабораторной работе, а какие нет?
- Что такое ядро? Где оно выполняется? Какие параметры необходимы для ее запуска?
- Какой максимальный размер блока для вашей реализации? Какие значения в Вашей реализации он может принимать? Что на это влияет?
- Для чего нужен $cudaDeviceSynchronize()$?

3.5 Доп. ресурсы

- CUDA C++ Programming Guide
- cuRand overview
- NVIDIA CUDA Installation Guide for Linux - лучше ставить через менеджер пакетов
- CUDA Installation Guide for Microsoft Windows