**Лабораторная работа №2**

**Основы работы с технологией CUDA. Иерархия памяти в CUDA. Работа с разделяемой памятью**

**Цель:** изучить иерархию памяти в CUDA, работу с разделяемой памятью и основы оптимизации программы в CUDA.

При подготовке к лабораторной работе рекомендуется изучить материалы, предоставленные в списке литературы, а также прочие материалы по теме лабораторной работы, представленные в открытых источниках.

Далее следует краткий конспект теоретического материала для лабораторной работы, задания и требования к лабораторной работе, а также контрольные вопросы для самопроверки.

1. **Иерархия памяти в CUDA**

Классификация памяти в CUDA приведена в таблице 2.1. Помимо классификации по физическому расположению (в DRAM и на GPU) память разделяют по функциональному назначению.

Таблица 2.1 – Типы памяти в CUDA

|  |  |  |  |
| --- | --- | --- | --- |
| Тип памяти | Доступ | Уровень выделения | Скорость работы |
| Регистровая | R/W | Per-thread | Высокая(on-chip) |
| **Локальная** | **R/W** | **Per-thread** | **Низкая (DRAM)** |
| Разделяемая | R/W | Per-block | Высокая(on-chip) |
| **Глобальная** | **R/W** | **Per-grid** | **Низкая (DRAM)** |
| Константная | R/O | Per-grid | Высокая (L1 cache) |
| Текстурная | R/O | Per-grid | Высокая (L1 cache) |

Каждый потоковый мультипроцессор содержит от 8192 32-битных регистров. Количество регистров зависит от версии графической карты. Регистры распределяются между нитями блока на этапе компиляции. Каждая нить монопольно использует некоторое количество регистров, которые доступны на чтение и на запись. Нить не имеет доступ к регистрам других нитей.

Регистровая память обладает максимальной скоростью доступа. Если имеющихся регистров не достаточно, то для размещения локальных данных используется локальная память, разположенная в DRAM.

Разделяемая (shared) память расположена непосредственно в потоковом мультипроцессоре. Каждый потоковый мультипроцессор содержит 16 Кбайт разделяемой памяти. Каждый блок получает в свое распоряжение одно и то же количество разделяемой памяти. Объем памяти зависит от количества блоков.

Разделяемая память доступна всем нитям блока. Скорость доступа к разделяемой памяти совпадает со скоростью доступа к регистрам.

Глобальная память – это память DRAM, которая выделяется с помощью специальных функций на CPU. Она обладает высокой латентностью (400-800 тактов). Правильное использование глобальной памяти является одним из направлений оптимизации программы.

Константная и текстурная память расположены в DRAM, но в отличии от глабальной памяти кешируются, поэтому скорость доступа может быть значительно выше. Память доступна всем нитям сетки на чтение, запись в память может осуществлять CPU.

1. **Эффективная работа с глобальной памятью в CUDA. Паттерны доступа**

Глобальная память является основным местом для размещения и хранения большого объема данных для обработки ядрами. Глобальная память выделяется и освобождается CPU при помощи следующих вызовов:

|  |
| --- |
| cudaError\_t cudaMalloc ( void \*\* devPtr, size\_t size );  cudaError\_t cudaFree ( void \* devPtr ); |

Функция выделения глобальной памяти возвращает указатель на память, расположенную в GPU. Для доступа CPU к этой памяти необходимо использовать функции копирования:

|  |
| --- |
| cudaError\_t cudaMemcpy(void \* dst, const void \* src, size\_t size,  enum cudaMemcpyKind kind);  cudaError\_t cudaMemcpyAsync(void \* dst, const void \* src, size\_t size,  enum cudaMemcpyKind kind, cudaStream\_t stream);  cudaError\_t cudaMemcpy2D(void \* dst, size\_t dpitch, const void \* src, size\_t spitch,  size\_t width, size\_t height, enum cudaMemcpyKind kind, cudaStream\_t stream);  cudaError\_t cudaMemcpy2DAsync(void \* dst, size\_t dpitch, const void \* src, size\_t spitch,  size\_t width, size\_t height, enum cudaMemcpyKind kind, cudaStream\_t stream); |
|  |

где kind – направление копирования, может принимать одно из следующих значений: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice.

Типовой пример работы с глобальной памятью в CUDA приведен на рисунке 2.1.

|  |
| --- |
| float \* devPtr; // указатель на память в GPU  // выделение памяти на GPU  cudaMalloc ( (void \*\*) &devPtr, 256\*sizeof ( float );  // копирование данных с CPU на GPU  cudaMemcpy ( devPtr, hostPtr, 256\*sizeof ( float ), cudaMemcpyHostToDevice );  // обработка данных  ……….  // копирование результатов с GPU на CPU  cudaMemcpy ( hostPtr, devPtr, 256\*sizeof( float ), cudaMemcpyDeviceToHost );  // освобождение памяти устройства  cudaFree( devPtr ); |

Рисунок 2.1 – Типовой пример работы с глобальной памятью в CUDA

В качестве другого примера работы с глобальной памятью рассмотрим задачу транспонирования матрицы. Дана матрица A размером NxN, N кратна 16. Матрица расположена в глобальной памяти.

Поскольку матрица А двухмерна организуем нити в двухмерную сетку из блоков 16х16. Тогда для транспонирования матрицы можно использовать ядро, продемонстрированное на рисунке 2.2.

|  |
| --- |
| \_\_global\_\_ void transpose(float\* inData, float\* outData, int n)  {  unsigned int xIndex = blockDim.x\*blockIdx.x + threadIdx.x;  unsigned int yIndex = blockDim.y\*blockIdx.y + threadIdx.y;  unsigned int inIndex = xIndex + n\*yIndex;  unsigned int outIndex = yIndex + n\*xIndex;  outData[outIndex] = inData[inIndex];  } |

Рисунок 2.2 – Пример ядра для транспонирования матриц

Классическим примером работы с глобальной памятью является задача перемножения матриц. Даны матрицы A и B размером NxN, N кратна 16. Матрицы расположены в глобальной памяти. Произведение вычисляется с помощью следующей формулы:

.

Организуем нити в двухмерную сетку из блоков 16х16. Каждая нить будет выполнять элемент произведения. Ядро, продемонстрированное на рисунке 2.3, выполняет перемножение матриц с использованием глобальной памяти.

|  |
| --- |
| #define BLOCK\_SIZE 16  \_\_global\_\_ void kernel ( const float \* a, const float \* b, int n, float \* c )  {  int bx = blockIdx.x; // индексы блока  int by = blockIdx.y; //    int tx = threadIdx.x; // индексы нити внутри блока  int ty = threadIdx.y; //    float sum = 0.0f;    // смещение для a[i][0]  int ia = n \* BLOCK\_SIZE \* by + n \* ty;    // смещение для b[0][i]  int ib = BLOCK\_SIZE \* bx + tx;    // смещение для результата  int ic= n \* BLOCK\_SIZE \* by+ BLOCK\_SIZE \* bx;    // перемножаем и суммируем  for( int k = 0; k < n; k++ )  sum+= a [ia+ k] \* b [ib+ k\*n];  c [ic+ n \* ty+ tx] = sum; // запоминаем результат  } |

Рисунок 2.3 – Фрагмент программного кода, выполняющий умножение матриц

В данном примере для нахождения одного элемента произведения двух матриц необходимо прочесть 2N значений из глобальной памяти и выполнить 2N арифметических операций.

Оптимизация работы с памятью в CUDA основана на следующих принципах:

* использование выравнивания;
* максимальное использование разделяемой памяти;
* объединение нескольких запросов к глобальной памяти в один (coalescing);
* использование специальных паттернов доступа к памяти, гарантирующих эффективный доступ. Паттерны работают независимо в пределах каждого half-warp’а.

Объединение (coalescing) для GPU с Compute Capability 1.0/1.1 случается в следующих случаях:

* нити обращаются к 32-битовым словам, давая 64-байтовый блок, или к 64-битовым словам, давая 128-байтовый блок;
* все 16 слов лежат в пределах блока;
* *k*-ая нить half-warp’а обращается к *k*-му слову блока.

На рисунке 2.4 приведены паттерны обращения к глобальной памяти, приводящие к объединению запросов в одну транзакцию. Слева выполнены все условия, справа пропущены слова для частей нити, что позволяет заменить на фиктивные запросы к памяти.

****

Рисунок 2.4 – Паттерны обращения к глобальной памяти, приводящие к объединению запросов в одну транзакцию

На рисунке 2.5 слева для нитей 4 и 5 нарушен порядок обращения к словам, а справа нарушено условие выравнивания, что не приводит к объединению запросов.

****

Рисунок 2.5 – Примеры обращения к глобальной памяти, не приводящие к объединению запросов в одну транзакцию

1. **Работа с разделяемой памятью**

Каждый потоковый мультипроцессор содержит 16 Кбайт разделяемой памяти. Она поровну делится между всеми блоками сетки, исполняемыми на мультипроцессоре. Разделяемая память используется для передачи параметров при запуске ядра на выполнение, поэтому желательно избегать передачи большого объема входных параметров. Существует два способа управления выделением разделяемой памяти.

1. Явное задание размера массива выделяемой памяти (с использованием спецификатора \_\_shared\_\_):

|  |
| --- |
| \_\_global\_\_ void incKernal(float \* a)  {  //явно задаем выделение 256\*4 байтов на блок  \_\_shared\_\_ float buf[256];  // запись значения из глобальной памяти в разделяемую  buf[threadIdx.x] = a[blockIdx.x\*blockDim.x + threadIdx.x];  } |

1. Задание размера массива при запуске ядра:

|  |
| --- |
| \_\_global\_\_ void kernal(float \* a)  {  \_\_shared\_\_ float buf[]; //размер явно не указан  // запись значения из глобальной памяти в разделяемую  buf[threadIdx.x] = a[blockIdx.x\*blockDim.x + threadIdx.x];  ….  }  // запустить ядро и задать выделяемый объем разделяемой памяти  kernel<<<dim3(n/256), dim(256), k\*sizeof(float)>>> (a); |

1. **Эффективная работа с разделяемой памятью в CUDA. Паттерны доступа**

Рассмотрим эффективную работу с разделяемой памятью на примере перемножения двух квадратных матриц. Даны матрицы A и B размером NxN, N кратна 16. Матрицы расположены в глобальной памяти. Каждый блок будет вычислять одну 16х16 подматрицу С' искомого произведения.

|  |  |
| --- | --- |
|  |  |

Рисунок 2.6 – Схема перемножения двух матриц

При перемножении двух подматриц А' и B' наиболее эффективным вариантом решения задачи является размещение их в разделяемой памяти, однако размеры подматриц могут значительно превышать размер разделяемой памяти. Однако, если каждую из полос (А' и B') разбить на подматрицы 16х16 (см. рисунок 2.6), то матрица C' является суммой попарных произведений.

.

Вычисление матрицы C' выполняется всего за N/16 шагов. Ядро, выполняющее перемножение матриц с использованием разделяемой памяти, представлено на рисунке 2.6. В данном примере для нахождения одного элемента произведения двух матриц необходимо прочесть 2N/16 значений из глобальной памяти.

|  |
| --- |
| \_\_global\_\_ void kernel(const float \* a, const float \* b, int n, float \* c)  {  int bx = blockIdx.x; // индексы блока  int by = blockIdx.y; //    int tx = threadIdx.x; // индексы нити внутри блока  int ty = threadIdx.y; //    int aBegin = n \* BLOCK\_SIZE \* by;  int aEnd = aBegin + n -1;  int aStep = BLOCK\_SIZE;  int bBegin = bx \* BLOCK\_SIZE;  int bStep = BLOCK\_SIZE\*n;  float sum = 0.0f;  for( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )  {  \_\_shared\_\_ float as [BLOCK\_SIZE][BLOCK\_SIZE];  \_\_shared\_\_ float bs [BLOCK\_SIZE][BLOCK\_SIZE];  as [ty][tx] = a [ia + n \* ty + tx];  bs [ty][tx] = b [ib + n \* ty + tx];  \_\_syncthreads (); // Убедимся, что подматрицы полностью загружены  for( int k = 0; k < BLOCK\_SIZE; k++ )  sum += as [ty][k] \* bs [k][tx];  \_\_syncthreads(); // Убедимся, что подматрицы никому больше не нужны  }  c [n \* BLOCK\_SIZE \* by + BLOCK\_SIZE \* bx + n \* ty + tx] = sum;  } |

Рисунок 2.6 – Фрагмент программного кода, выполняющий умножение матриц с использованием разделяемой памяти

Для повышения пропускной способности разделяемая память разбита на 16 (CUDA 1.x) или 32 (CUDA 2.x/3.x) банков. Каждый банк работает независимо от других. Можно одновременно выполнить до 16 обращений к разделяемой памяти. Если идет несколько обращений к одному банку, то они выполняются последовательно. Номер банка можно определить согласно следующим выражениям:

Номер банка = (Адрес в байтах/4)%32 — для устройства версии 2.0.

Номер банка = (Адрес в байтах/4)%16 — для устройства версии 1.x.

Банки строятся из 32-битовых слов (рисунок 2.7). Подряд идущие 32-битовые слова попадают в подряд идущие банки. Конфликт доступа к банкам (bank conflict) возникает, если несколько нитей из одного half-warp’а (CUDA 1.x) или из одного warp’а (CUDA 2.x/3.x) обращаются к одному и тому же банку. Конфликта не происходит, если все 16 нитей обращаются к одному слову.

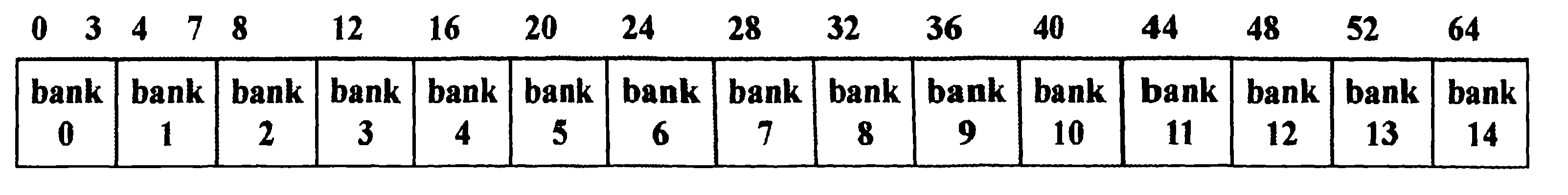


Рисунок 2.7 – Схема расположения банков

Типичные паттерны доступа к разделяемой памяти приведены на рисунке 2.8. На рисунке 2.9 приведены два паттерна доступа к разделяемой памяти, приводящие к возникновению конфликтов по банкам памяти. Паттерн доступа слева приводит к появлению 8 конфликтов 2-ого порядка (доступ работы с разделяемой памятью снизится в два раза), паттерн справа приводит к появлению 4, 5 и 6-ого порядка (доступ работы с разделяемой памятью снизится в шесть раз).



Рисунок 2.8 – Паттерны обращения к разделяемой памяти, не приводящие к появлению конфликта банков



Рисунок 2.9 – Паттерны обращения к разделяемой памяти, приводящие к появлению конфликта банков

1. **Общий шаблон решения задач в CUDA**

Несмотря на существенную разницу в назначении алгоритмов, исполняемых на GPU, можно выделить набор общих инструкций, которых следует придерживаться при составлении программы для GPU.

1. Выделение набора инструкций исполняемых одним экземпляром ядра.

2. Выделение набора данных, общего для блока ядер и загрузка его в разделяемую память.

3. Если объемы данных на шаге 2 слишком велики или набор инструкций выделенных на шаге 1 предполагает слишком интенсивные вычисления задача декомпозируется на более простые задачи.

4. Производятся вычисления над подгруженными ранее данными.

5. Результаты вычисления копируются назад, в глобальную память.

Особое внимание следует уделять размерностям блоков и сетки. Так, если ядра содержат большое количество инструкций – стоит уменьшить размерность блоков. Это позволит более эффективно использовать регистровую память GPU.

**Лабораторные задания** (№ варианта = (№ студента в списке)%2)

**Задание 1.** Даны два вектора А и В из N натуральных (ненулевых) элементов (задаются случайно). Вектора расположены в глобальной памяти. Написать программу на Cи с использованием CUDA runtime API, выполняющую поэлементное умножение двух векторов на GPU так, чтобы продемонстрировать паттерны доступа к глобальной памяти, приводящие и не приводящие к объединению запросов в одну транзакцию.

Измерить время работы программ. Написать программу для верификации результатов. **Результаты занести в отчёт.**

**Задание 2.** Написать программу на Cи с использованием CUDA runtime API в соответствии с вариантом задания. Измерить время работы программы для различных значений параметров на CPU, GPU с использованием глобально памяти, GPU с использованием разделяемой памяти. Написать программу для верификации результатов. **Результаты занести в отчёт.**

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| **0** | Дана матрица А из NxN натуральных (ненулевых) элементов (задаются случайно). Матрица расположена в глобальной памяти.  Написать программу, выполняющую транспонирование матрицы в двумерной сетке. |
| **1** | Даны матрицы А и В из NxN натуральных (ненулевых) элементов (задаются случайно). Матрицы расположены в глобальной памяти.  Написать программу, выполняющую перемножение двух матриц в двумерной сетке. |

**Контрольные вопросы**

1. Какие типы памяти поддерживаются в CUDA?
2. В чем заключается эффективность работы с глобальной и разделяемой памятью в CUDA?
3. Какие паттерны доступа к глобальной памяти существуют?
4. Какие паттерны доступа к разделяемой памяти существуют?
5. В чем особенность схемы расположения банков в разделяемой памяти?
6. Каким образом можно определить объем разделяемой памяти?
7. Как определить номер банка памяти?
8. Основные принципы оптимизации работы с памятью в CUDA.
9. Общий шаблон решения задач в CUDA.

**Требования к сдаче работы**

1. При домашней подготовке изучить теоретический материал по тематике лабораторной работы, представленный в списке литературы ниже, выполнить представленные примеры, занести в отчёт результаты выполнения.
2. Продемонстрировать выполнение лабораторных заданий.
3. Ответить на контрольные вопросы.
4. Показать преподавателю отчет.

**Литература**

1. <http://www.nvidia.ru/object/cuda-parallel-computing-ru.html>
2. А.В. Боресков, А.А. Харламов. Основы работы с технологией Cuda. – М: ДМК Пресс, 2010. – 232 с.