Введение

CUDA (Compute Unified Device Architecture)**Ошибка! Источник ссылки не найден.** – программно-аппаратная архитектура массового параллелизма для графических процессоров NVIDIA©. Данная архитектура обладает одной из наиболее успешных и простых моделей массового параллелизма, необходимого для эффективного программирования GPU.

Изначально GPU не имели универсальной архитектуры и использовались только для рендеринга 3D сцен. По мере роста сложности создаваемых сцен и улучшения их качества потребовалось внести урезанные программные возможности по трансформации геометрии и текстур. Эти мини программы получили название вершинные и пиксельные шейдеры. Начиная с версии 2.0, которые вошли в состав Microsoft© DirectX 9.0, была введена поддержка циклов. Именно на этих шейдерах впервые были реализованы универсальные алгоритмы, такие как, перемножение матриц, вычисления DCT и др. Это послужило началом создания новой области применения графических процессоров – GPGPU (General Purpose Computation on GPU). Из-за высокой сложности программирования шейдеров и недостаточного развития средств разработки, развитие GPGPU шло достаточно медленно.

Следующим этапом развития GPGPU стало объединение вершинных и пиксельных шейдеров. Именно с этого момента GPU стали универсальными вычислительными устройствами.

Осознав высокий потенциал GPU в области универсальных вычислений, компания NVIDIA© в 2008 году представила первую версию технологии CUDA. Эта модель не является исторически первой архитектурой для универсальных вычислений на GPU. Чуть раньше компания ATI/AMD© представила архитектуру, разработанную совместно с университетом Стенфорда, которая называлась BrookGPU[2]. Несмотря на некоторые революционные идеи, заложенные в ней, она обладала меньшей универсальностью и не имела явных средств для работы с разделяемой памятью. Из-за достаточно сырой программной реализации компилятора и средств разработки данная модель не получила широкого распространения.

В 2010 году представлена технология параллельного программирования OpenCL (Open Computational Language)**Ошибка! Источник ссылки не найден.**. Технология является открытым стандартом и поддерживаемая рядом крупных компаний (Apple©, Intel©, AMD©). С точки зрения вычислительной модели и способа выражения параллелизма эта технология идентична CUDA.

Из-за того, что OpenCL развивается достаточно медленно и его поддерживает множество компаний, первоначально все новые особенности GPU отражаются в CUDA, а уже потом в OpenCL. В настоящее время компания NVIDIA© является лидером на рынке GPGPU, ее сопроцессоры устанавливаются в большинство современных суперкомпьютеров.

1. Программная модель CUDA

Классический процессор, построенный по фон-неймановской архитектуре, не подразумевает наличие параллелизма: есть один поток команд, все команды исполняются последовательно и обрабатывают один поток данных. Степень параллелизма у такого процессора равна 1 (количество одновременно обрабатываемых элементов данных). Для данной модели рост производительности возможен только за счет роста тактовой частоты. Современные процессоры умеют использовать параллелизм на уровне инструкций за счет суперскалярного и внеочередного исполнения. Но это ведет к росту сложности управляющей логики, которая начинает доминировать над исполняющими устройствами. Вся микроархитектура центрального процессора ориентирована на уменьшение латентности, т.к. для скорейшего исполнения единственного потока команд нужно как можно быстрее исполнить цепочку зависимых инструкций на критическом пути.

Архитектуры массового параллелизма ориентированы на общую производительность системы (пропускную способность), когда однотипные действия выполняется над массивом независимых элементов. GPU изначально являлись архитектурой массового параллелизма, так как задача построения 3D сцены как раз и является задачей с однотипной обработкой массива независимых элементов (вершин или пикселей). Для таких архитектур важным является скорейшее завершение всей задачи, нежели обработка одного элемента, т.к. все элементы множества обладают равным приоритетом. Степень параллелизма таких задач определяется количеством элементов в массиве. Для задач построения 3D сцен степень параллелизма может достигать порядка $10^6 – 10^7$.

Латентность для устройств с архитектурой массового параллелизма особой роли не играет, так как наличие большого количества независимых элементов позволяет замаскировать вычисления одних элементов другими.

1.1. Нить, блок, сетка блоков

В упрощенном виде вычислительный блок графического процессора представляет собой совокупность потоковых мультипроцессоров (streaming multiprocessor, SMX), управляемые с помощью GigaThread Engine. Каждый SMX состоит из множества вычислительных CUDA-ядер (рисунок 2.1).

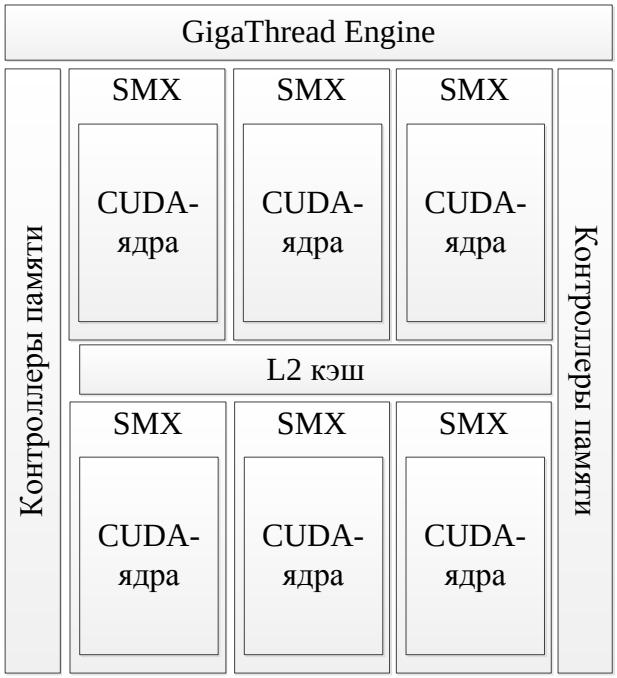

Рис. 1.1. Упрощенная структура графического процессора

Такая архитектура легла в основу базовых понятий в CUDA – *нити (thread), блока (block)* и *решетки блоков (grid).*

Нить (thread) – базовая абстракция, представляющая собой легковесный поток, отвечающий за исполнение инструкций. Именно нить соответствует одному исполняющему ядру CUDA мультипроцессора. Максимальное число нитей, которое можно определить, - 2048 (для архитектуры NVIDIA© Kepler). Легковесность нити заключается в том, что время, необходимое на его создание, разрушение и переключение между нитями, ничтожно мало. Для упрощения аппаратной схемы каждые 32 CUDA ядра соединены с одним счетчиком команд, поэтому данные блоки ядер всегда выполняют одну и ту же инструкцию. Данные 32 нити называются *варпом (warp).*

Множество нитей объединяется в *блок (block)*, который проецируется на мультипроцессор. Это означает, что все нити блока всегда будут выполняться на выделенном для него мультипроцессоре.

Решетка блоков (grid) представляет собой самый высокий уровень абстракции и, как правило, представляет собой всю задачу, которую требуется решить.

Каждая из абстракций является 3-мерной. Для ее определения в NVIDIA© CUDA определена специальная структура `dim3`. Инициализация значений структуры осуществляется следующим образом:

```
dim3 blocks (16, 16); // эквивалентно blocks ( 16, 16, 1 )
dim3 grid (256); // эквивалентно grid ( 256, 1, 1 )
```

Для того чтобы определить текущие координаты в решаемой задаче введены следующие константы:

```
// размерность решетки блоков при запуске ядра
dim3 gridDim;
// координаты текущего блока внутри решетки блоков
uint3 blockIdx;
// размерность блока при запуске ядра
dim3 blockDim;
// координаты текущей нити внутри блока
uint3 threadIdx;
```

Базовая задача, которую программист должен решить, - правильное разбиение решаемой задачи между нитями и блоками для обеспечения эффективной нагрузки всех имеющихся вычислительных ресурсов.

Например, графический процессор с архитектурой NVIDIA© Kepler содержит до 15 мультипроцессоров, каждый из которых содержит 192 CUDA-ядра. Всего получается до 2880 CUDA ядер. Для утилизации данной вычислительной мощности требуется как минимум 15 блоков (так как каждый блок проецируется на один мультипроцессор) и по 192 нити в каждом блоке. Данная конфигурация, с одной стороны, сможет сопоставить на каждое CUDA ядро фрагмент решаемой задачи, а, с другой стороны, не является эффективной. Главным преимуществом CUDA является легковесность нитей и ничтожно малое время переключения между ними. Встроенный в графический процессор планировщик следит за загруженностью ядер и, если нити варпа перешли в режим ожидания результатов выполнения инструкции, планировщик может переключиться на выполнение других нитей данного блока, что позволяет снизить простаивание системы и в итоге время выполнения всего кода. Поэтому при проектировании конфигурации нитей и блоков рекомендуется подбирать конфигурацию таким образом, чтобы она была заведомо большей, чем имеющиеся вычислительные ресурсы.

1.2. Ядро, конфигурирование и запуск ядра

Ядро (kernel) – функция, описывающая последовательность операций, выполняемых каждой нитью параллельно.

Ядро, выполняющее поэлементное сложение двух матриц, описывается следующим образом:

```
__global__ void MatAdd(float *A, float *B, float *C) {
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      C[idx] = A[idx] + B[idx];
}
```

Для определения устройства исполнения функции (ядра) в CUDA вводятся следующие ключевые слова:

- `__host__` – функция может быть вызвана и выполнена только на стороне хоста (на центральном процессоре). Если не указывается спецификатор, то функция считается объявленной как `__host__`.
- `__global__` – ядро, предназначенное для выполнения на устройстве (графическом процессоре) и может быть вызвано с хоста. На устройствах с Compute Capability 3.5 (способ описания версии архитектуры графического процессора и поддерживаемого CUDA API) и новее допускается вызов с устройства. Ядра данного типа не возвращают никаких данных (тип `void`);
- `__device__` – ядро выполняется на устройстве, вызывается из кода, выполняемого на устройстве;

Вызов ядра `MatAdd` выполняется следующим образом:

```
int main(){
...
      const int N = 1024 * 1024;
      dim3 threadsPerBlock(1024);
      dim3 numBlocks(N / threadsPerBlock.x);
      MatAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C);
...
}
```

Здесь считается, что матрицы `d_A`, `d_B` и `d_C` имеют размер `N` элементов. Специальное расширение языка C `<<<...>>>` определяет конфигурацию запуска нитей. Первый параметр задает количество блоков по каждому из 3-х из-

мерений (тип `dim3`), а второй – количество нитей в блоке, так же по каждому измерению.

Все запуски ядер CUDA являются асинхронными: CPU запрашивает разрешение на запуск ядра путем записи в специальный буфер команд без проверки выполнен код или нет, после этого продолжает выполнять код в соответствии с алгоритмом для хоста.

1.2.1. Выделение памяти, копирование на GPU

Хост и устройство имеют разные типы памяти. Ядро на момент написания методического пособия может взаимодействовать только с памятью устройства, поэтому перед запуском ядра требуется выделить память на устройстве и скопировать туда данные из памяти хоста. После завершения вычислений результат копируется обратно, а выделенная память должна быть освобождена.

Память на устройстве может быть выделена как *линейная память* или как *CUDA массив* (CUDA Array) – специальная разметка памяти, оптимизированная для работы с текстурами. Более подробно массивы описываются в **Ошибка! Источник ссылки не найден.**.

Линейная память выделяется с помощью функции `cudaMalloc()` и освобождается с помощью `cudaFree()`. Передача данных между памятью хоста и девайса чаще всего выполняется с помощью `cudaMemcpy()`.

Для рассмотренного ранее ядра `MatAdd` выделение памяти производится следующим образом:

```
int main() {
      const int N = 1024 * 1024;
 size_t size = N * sizeof(float);

      // выделение памяти h_A и h_B на хосте
      float* h_A = (float*)malloc(size);
      float* h_B = (float*)malloc(size);
      float* h_C = (float*)malloc(size);

      // Инициализация входных массивов
      ...

      // Выделение векторов в памяти GPU
      float* d_A, *d_B, *d_C;
      cudaMalloc(&d_A, size);
      cudaMalloc(&d_B, size);
      cudaMalloc(&d_C, size);
```

```
        // Копирование векторов из памяти хоста
        //    в память устройства
        cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

        // Вызов ядра
        dim3 threadsPerBlock(1024);
        dim3 numBlocks(N / threadsPerBlock.x);
        MatAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C);

        // Копирование результата с устройства на хост
        cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

        // Освобождение памяти устройства
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);

        free(h_A);
        free(h_B);
        free(h_C);
    }
```

Линейная память может быть выделена так же функциями `cudaMallocPitch()` и `cudaMalloc3D()`. Эти функции рекомендуется использовать для выделения памяти под 1D и 3D массивы соответственно, при этом начало каждой строки или слоя размещается по адресу кратному 128, что необходимо для быстрого доступа к глобальной памяти. Соседние строки или слои могут размещаться не последовательно (за последним байтом одной строки не гарантируется, что следует первый байт следующей), а с некоторым смещением, которое называется pitch (или stride для 3D). Пример использования функции `cudaMallocPitch()` будет показан в разделе 4.

Для копирования данных в/из 2D памяти устройства используется функция

```
cudaMemcpy2D(
    void* dst,// адрес приемника данных
    size_t dpitch,// фактическая ширина одной строки приемника
    const void* src,// адрес источника данных
    size_t spitch,// фактическая ширина одной строки источника
    size_t width,        // ширина копируемых данных в байтах
    size_t height,       // высота матрицы копируемых данных
    cudaMemcpyKind kind  // направление копирования
```

```
)
```

Параметр `cudaMemcpyKind` используется во всех функциях передачи данных и может иметь следующие значения:

```
cudaMemcpyHostToHost = 0        // Host -> Host
cudaMemcpyHostToDevice = 1      // Host -> Device
cudaMemcpyDeviceToHost = 2      // Device -> Host
cudaMemcpyDeviceToDevice = 3    // Device -> Device
cudaMemcpyDefault = 4           // По-умолчанию
```

1.3. Конфигурирование проекта (nvcc, -arch, -code, -gencode)

Код программы, написанный с использованием CUDA, должен сохраняться в файлы с расширением `.cu`, заголовочные файлы – с расширением `.cuh`.

Исходный код CUDA программы компилируется с помощью утилиты *nvcc*, распространяемой вместе с CUDA Toolkit. Данная утилита является консольной и позволяет управлять процессом компиляции с помощью параметров командной строки.

Исходный код, компилируемый с помощью nvcc, как правило, содержит код, выполняемый на CPU, и код, выполняемый на GPU. Процесс компиляции выглядит следующим образом:

- код, предназначенный для CPU, компилируется с использованием компилятора C/C++, используемого по-умолчанию;
- компилируется код, предназначенный для GPU, в ассемблерную форму (специальный PTX код) и/или бинарную форму (cubin объект). При этом с помощью опций компилятора возможно настроить версию Compute Capability, для которой осуществляется генерация PTX-кода, и версию архитектуры при генерации бинарного кода;
- полученный объектный файл интегрируется в исполняемый (либо в библиотеку) и добавляется код управления данным объектом.

Для определения версии Compute Capability, т.е. для определения версии CUDA API и PTX-кода, используется параметр compute_XX, где XX – версия Compute Capability (допустимые значения – compute_10, compute_11, compute_12, compute_13, compute_20, compute_30, compute_32, compute_35, compute_50). Для определения версии архитектуры, для которой создается бинарный файл, используется параметр sm_XX, где XX – версия архитектуры (допустимые значения – sm_10, sm_11, sm_12, sm_13, sm_20, sm_21, sm_30, sm_32, sm_35, sm_50).

Для управления процессом компиляции используются следующие параметры:

- -arch – определяет архитектуру графического процессора, для которой осуществляется создание PTX-кода. Значение по-умолчанию – compute_10;
- -code – определяет архитектуру графического процессора, для которой будет создан PTX и/или бинарный код. Значение по-умолчанию – compute_10,sm_10;
- -gencode – объединяет опции -arch и -code в качестве одного параметра.

При использовании параметра sm_XX не гарантируется совместимость между различными версиями архитектур графического процессора, т.е. для бинарного кода, скомпилированного для архитектуры sm_30, не гарантируется, что он будет исполнен на графическом процессоре с архитектурой sm_20. В тоже время указание, например, compute_20 добавит PTX-код с поддержкой CUDA API и PTX для архитектуры 2.0 в исполняемый файл. При этом поведение драйвера можно описать следующим образом:

- бинарная версия кода для текущей архитектуры есть в исполняемом файле?
  - да – исполняем бинарную версию кода;
  - нет – проверяем PTX-код:
    - есть PTX код и версия API в PTX-коде меньше или равна поддерживаемой текущей архитектурой?
      - да – компилируем код с помощью драйвера и исполняем;
      - нет – прерываем работу с генерацией ошибки.
    - PTX-код отсутствует – прерываем работу с генерацией ошибки.

Примечание [D1]: Оставить так? Или сделать блок-схему

Также компилятором поддерживается множественное задание различных архитектур (например, для версий 1.0, 2.0, 3.0). При этом компилятором будет созданы версии кода для каждой из архитектур и поведение драйвера в результате усложняется, т.к. требуется из всего множества версий найти наиболее подходящую.

Пример использования опции -gencode:

```
nvcc x.cu
      -gencode arch=compute_20,code=sm_20
      -gencode arch=compute_30,code=\'compute_30,sm_30\'
```

В коде программы можно определить участок кода для различных архитектур, используя макрос `__CUDA_ARCH__`. Данный макрос будет работать только в ядре. Например, при компиляции с параметром `arch=compute_20`, значение `__CUDA_ARCH__` равно `200`.

При использовании CUDA Toolkit 6.0 и новее не рекомендуется использование архитектуры версии 1.0, т.к. данная версия считается устаревшей.

1.4. Использование событий в CUDA

При работе с технологией CUDA вместо стандартных методов измерения времени (функции `GetTickCount`, `time` и другие) рекомендуется использовать CUDA-события (events). При этом будут задействованы счетчики, расположенные на графическом процессоре, что обеспечит более качественное измерение времени при работе с CUDA.

Следующий пример демонстрирует создание, использование и уничтожение событий для замера времени выполнения передачи данных и работы ядра:

```
cudaEvent_t start, stop;
// создание события для точки старта
cudaEventCreate(&start);
// создание события для точки завершения
cudaEventCreate(&stop);

// точка начала замера времени
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
      cudaMemcpyAsync(inputDev,
                      inputHost, size,
                      cudaMemcpyHostToDevice, 0);
      MyKernel<<<100, 512>>> (outputDev, inputDev, size);
      cudaMemcpyAsync(outputHost,
                      outputDev, size,
                      cudaMemcpyDeviceToHost, 0);
}
// точка завершения замера времени
cudaEventRecord(stop, 0);
// ожидание завершения выполнения задач на GPU
cudaEventSynchronize(stop);
float elapsedTime;
// elapsedTime - затраченное время в миллисекундах
cudaEventElapsedTime(&elapsedTime, start, stop);
```

```
// уничтожение объектов событий
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

При исполнении кода функция `cudaEventCreate` создает событие и с помощью функции `cudaEventRecord` фиксируются интересуемые точки, между которыми планируется измерить время работы (переменные `start` и `stop`). Так как ядро вызывается асинхронно и используются асинхронные версии функций копирования, требуется осуществить синхронизацию для гарантии того, что ядро завершило свою работу. Для этого используется функция `cudaEventSynchronize`.

После синхронизации можно определить время между двумя событиями с помощью функции `cudaEventElapsedTime`, которая возвращает время в миллисекундах.

1.5. Задание

Реализовать CPU и GPU версии программ в соответствии с вариантом задания. Результат работы на GPU проверяется с помощью реализации на CPU, которая является эталонной. Для обеих версий измерить время работы алгоритма.

Варианты задания:

1. транспонирование матрицы;
2. сложение двух матриц в соответствии с вариантом, выданным преподавателем;
3. перемножение двух матриц;
4. выполнить обнуление элементов выше (ниже) главной диагонали матрицы.

2. Иерархия памяти в CUDA

Технология CUDA использует следующие типы памяти: регистры, локальная, глобальная, разделяемая, константная и текстурная память (рисунок 2.1). Однако разработчику недоступны к управлению регистры и локальная память. Все остальные типы он вправе использовать по своему усмотрению.

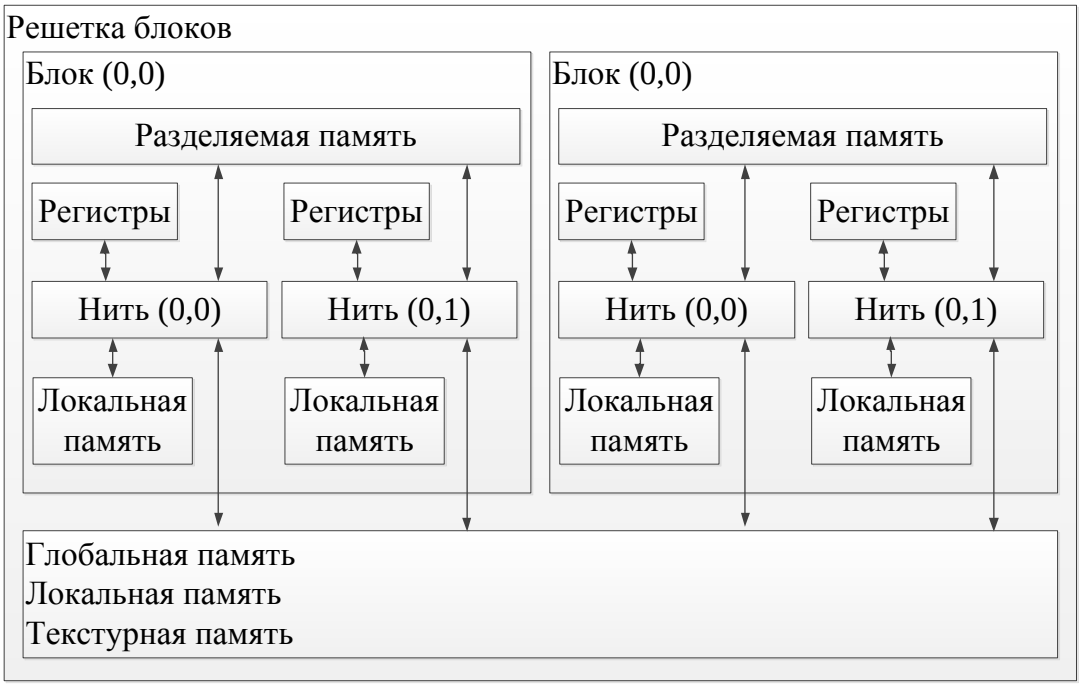

Рис. 2.1. Иерархия памяти CUDA

Характеристики каждого типа памяти представлены в таблице 2.1. Для локальной и глобальной типов памяти был добавлен кэш, начиная с Compute Capability 2.0 (обозначено символом «*»).

Таблица 2.1

Типы памяти CUDA

Тип памяти	Расположение	Кэш	Доступ	Видимость	Время жизни
Регистры	Мультипроцессор	Нет	R/W	Нить	Нить
Локальная	DRAM GPU	*	R/W	Нить	Нить
Разделяемая	Мультипроцессор	Нет	R/W	Блок	Блок

Глобальная	DRAM GPU	*	R/W	Сетка	
Константная	DRAM GPU	Есть	R	Сетка	
Текстурная	DRAM GPU	Есть	R	Сетка	

2.1. Глобальная память

Глобальная память (global memory) – тип памяти с самой высокой латентностью, из доступных на GPU. Переменные в данном типе памяти можно выделить с помощью спецификатора `__global__`, а так же динамически, с помощью функций из семейства `cudaMalloc`. Глобальная память в основном служит для хранения больших объемов данных, над которыми осуществляется обработка, и для сохранения результата работы. Данные перемещения осуществляются с использованием функций `cudaMemcpy`. В алгоритмах, требующих высокой производительности, количество операций с глобальной памятью необходимо свести к минимуму.

Пример сложения двух векторов из подраздела 1.2.1 демонстрирует использование глобальной памяти.

Для ускорения передачи данных между ОЗУ и памятью устройства используется механизм *объединения запросов (coalescing)*. Шаблон доступа к глобальной памяти отличается в зависимости от версии архитектуры графического процессора (Compute Capability), на котором осуществляется запуск. Самые жесткие требования накладывает версия 1.x.

Для Compute Capability 1.0 и 1.1 накладываются следующие ограничения для полуварпа (16 нитей из варпа):

- размер слова, к которому осуществляется доступ нитью, должен быть 4, 8 или 16 байтным. При этом если работа осуществляется с 4 либо 8 байтными словами, тогда все 16 слов должны находиться в одном 64-либо 128-байтном сегменте соответственно. При 16 байтных словах первых 8 слов должны лежать в одном 128-байтном сегменте, следующие 8 слов следующим за первым 128-байтном сегменте;
- нити должны обращаться последовательно, т. е. k-я нить полуварпа должна обращаться к k-ому слову.

Если данные условия выполняются, формируются 64-, 128- или 2 по 128 байт транзакции соответственно для 4, 8 и 16 байтных словах.

Для Compute Capability 1.2 и 1.3 шаблон доступа несколько усовершенствован (при этом доступ по-прежнему осуществляется полу-варпами):

- осуществляется поиск сегмента в памяти, содержащего адрес, запрашиваемый активной нитью с минимальным индексом thread ID. Размер сегмента зависит от размера слова, к которому осуществляется доступ:
    - 32 байта для 1-байтных слов;
    - 64 байта для 2-байтных слов;
    - 128 байт для 4-, 8- и 16-байтных слов;
- ищутся все активные нити, запрашиваемый адрес которых находится в одном сегменте;
- анализируется возможность понижения размера транзакции:
    - если размер транзакции 128 байт и только старшая или младшая его части загружаются, происходит уменьшение размера транзакции до 64 байт:
    - по аналогии для 64 байт возможно понижение размера транзакции до 32 байт;
- выполняется транзакция и обработанные нити отмечаются как не активные;
- алгоритм повторяется для всех нитей полуварпа.

Для Compute Capability 2.x шаблон доступа существенно упростился. В основном это связано с вводом кэша. При компиляции появляется дополнительный флаг, позволяющий задействовать L1 и L2 кэши (опция -Xptxas -dlcm=ca, включенная по-умолчанию) либо только L2 (опция -Xptxas -dlcm=cg).

Размер кэш-линии — 128 байт, на которые проецируется соответственно 128-байтный выравненный сегмент в глобальной памяти. При этом если задействуется L1 и L2 кэши, формируются 128-байтные транзакции, в случае, если только L2 кэш, тогда формируются только 32-байтные транзакции (в зависимости от местоположения запрашиваемых данных (например, в произвольном порядке) 32-байтные транзакции могут оказаться более эффективными).

В случае если размер слова, запрашиваемого нитью, больше 4 байт, запрашиваемый варпом, блок данных разделяется на 128-байтные запросы, обрабатываемые независимо. Транзакции по 128 байт будут формироваться до тех пор, пока все нити варпа не получат необходимые данные.

В отличие от Compute Capability 1.x шаблон доступа в архитектуре 2.x допускает доступ нитью к словам в произвольном порядке.

Для Compute Capability 3.x убрана возможность управления кешированием данных в L1, т.к. данный уровень кэша зарезервирован для работы с локальной памятью. При этом схема работы для L2 сохраняется такой же, как и для Compute Capability 2.x. Для Compute Capability 3.5 добавлена поддержка работы с

кэшем, доступным из ядра только на чтение. Более подробно этот режим рассмотрен в [1].

На рисунке 2.2 показан пример эффективного обращения в глобальную память, в результате такого чтения будет сформирована одна транзакция. При этом предполагается, что каждая нить обращается к 4-байтному слову. Следует учитывать, что некоторые нити могут не обращаться в свою ячейку памяти (например, нить 2 на рисунке 2.2), при этом транзакция нарушена не будет. Загружаемый блок данных выделен с помощью точек.

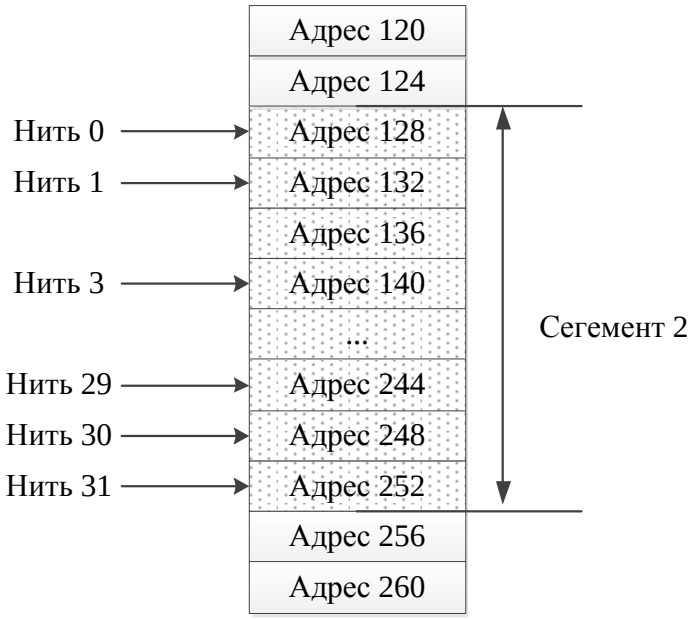

Рис. 2.2. Пример выравненного обращения к глобальной памяти

На рисунке 2.3 показан пример обращения к памяти со смещением. В результате такого обращения будут сформированы 2 транзакции и загружены два 128-байтных сегмента данных. Загружаемый блок данных выделен с помощью точек.

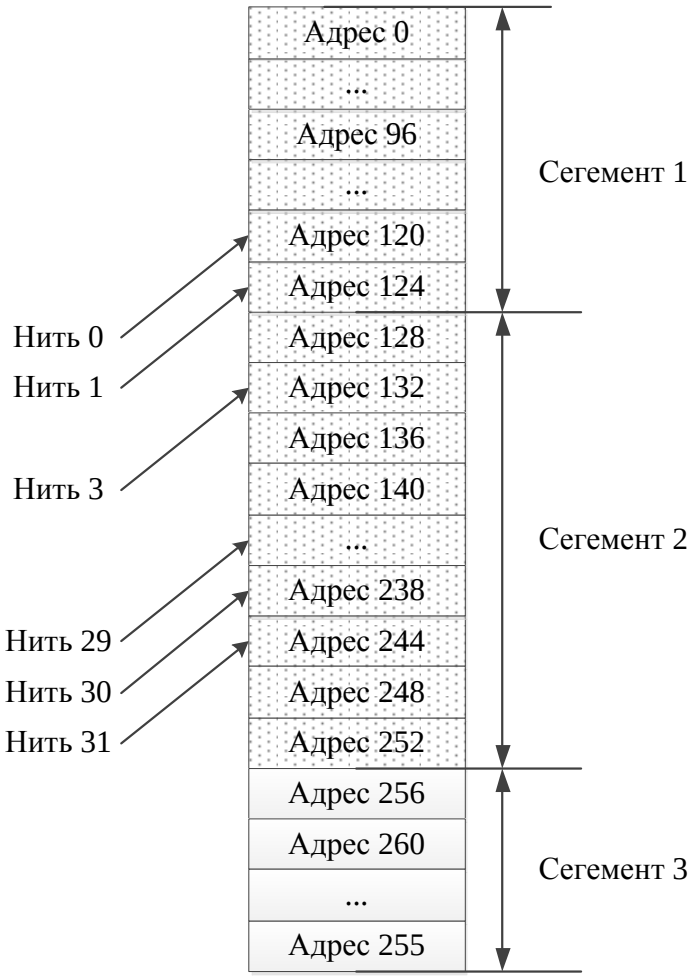

Рис. 2.3. Пример не выравненного обращения к глобальной памяти

В тоже время отключение L1 (рисунок 2.4) позволяет уменьшить объем данных, который будет передан при выполнении. При этом будут загружены данные, начиная с адреса 96. Загружаемый блок данных выделен с помощью точек.

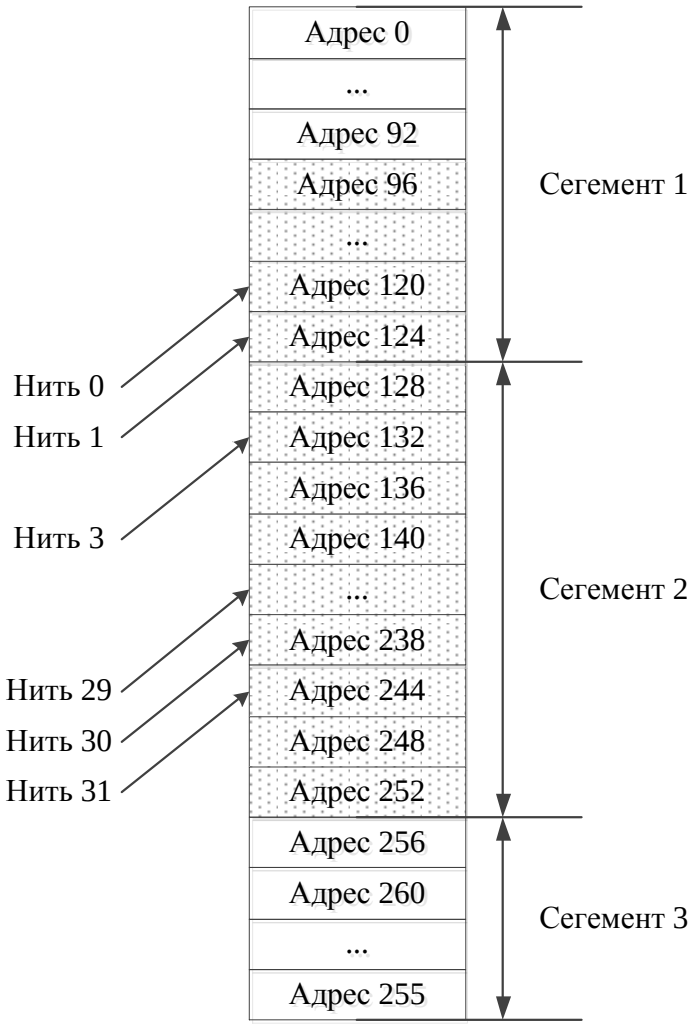

Рис. 2.4. Пример не выравненного обращения к глобальной памяти с отключенным кэшем L1

2.2. Разделяемая память

Разделяемая память (shared memory) относится к типу памяти с низкой латентностью. Данный тип памяти рекомендуется использовать для минимизации обращения к глобальной памяти, а так же для хранения локальных переменных функций. Адресация разделяемой памяти осуществляется между нитями одного блока, что может быть использовано для обмена данными между по-

Примечание [D2]: ????

токами в пределах одного блока. Для размещения данных в разделяемой памяти используется спецификатор `__shared__`.

Разделяемая память делится на блоки фиксированного размера, доступ к которым осуществляется одновременно. Таким образом, если нити обращаются к различным банкам, латентность доступа будет минимальна. Если хотя бы одна из нитей не получила данные из-за конфликта доступа к банку, осуществляется повторный доступ к памяти и данные догружаются. Данная процедура будет повторяться до тех пор, пока не будут разрешены все конфликты и загружены или сохранены все данные. В результате время доступа будет произведением порядка конфликта на латентность доступа к разделяемой памяти.

Рассмотрим следующий пример. Имеется массив на `N` элементов типа `int`. Требуется выполнить сложение трех подряд идущих элементов и сохранить результат в качестве одного элемента выходного массива (рисунок 2.5).

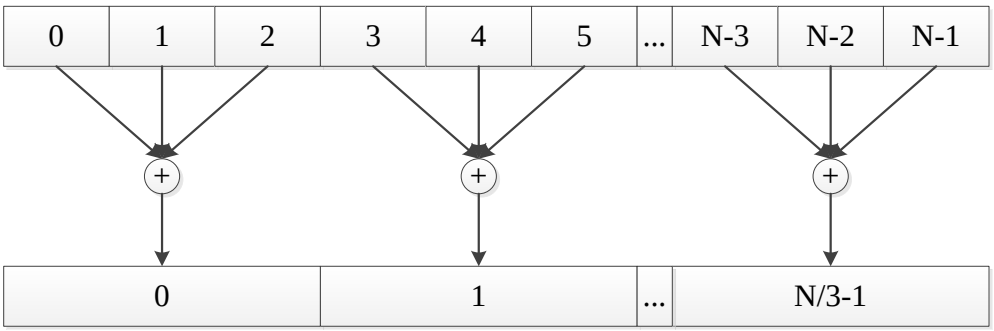

Рис. 2.5. Постановка задачи для примера

При формировании конфигурации исходим из следующих предположений:

- одна нить формирует один выходной элемент;
- число нитей в блоке – 512;
- число элементов N гарантированно делится на 512*3=1536.

Простейшее решение – каждая нить читает последовательно 3 элемента из глобальной памяти. Однако такое решение приводит к тому, что возникает конфликт третьего порядка при чтении из глобальной памяти и, как результат, нарушение условия формирования транзакций.

Для того чтобы устранить данный недостаток рекомендуется задействовать разделяемую память как промежуточный буфер. При этом запись в разделяемую память осуществляется блоками по 512 последовательных элементов. Пример эффективного решения данной задачи показан ниже:

```
__global__ void Sum(
           const int *input, const int *output, const int N
) {
      __shared__ smem[512 * 3];

      // копируем входные данные в разделяемую память
      smem[threadIdx.x] =
            input[blockIdx.x * 512 * 3 + threadIdx.x];
      smem[threadIdx.x +  512] =
            input[blockIdx.x * 512 * 3 + threadIdx.x + 512];
      smem[threadIdx.x + 1024] =
            input[blockIdx.x * 512 * 3 + threadIdx.x + 1024];

      // синхронизируем нити, чтобы гарантировать,
      // что в разделяемую память записаны все данные
      __syncthreads();

      // находим сумму и сохраняем результат работы
      output[blockIdx.x * 512 + threadIdx.x] =
                  smem[threadIdx.x * 3] +
                  smem[threadIdx.x * 3 + 1] +
                  smem[threadIdx.x * 3 + 2];
}
```

Функция `__syncthreads()` гарантирует, что все нити блока выполнили запись в разделяемую память. Разрешение конфликта 3-го порядка, возникающего при чтении данных из разделяемой памяти, является гораздо менее затратным по сравнению с конфликтом при чтении из глобальной памяти, поэтому в данном случае этим можно пренебречь.

Запись в глобальную память осуществляется с формированием транзакций, поэтому промежуточного сохранения в разделяемую память не требуется.

2.3. Константная память

Константная память (constant memory) является одной из самых быстрых из доступных на GPU. Отличительной особенностью данного типа памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название. Константная память обладает собственным кэшем, что обеспечивало выигрыш на платформах с Compute Capability 1.x, однако на более новых архитектурах из-за ввода кэшей L1 и L2 константная память частично потеряла свою актуальность.

Для размещения данных в константной памяти предусмотрен спецификатор `__constant__`. Если необходимо использовать массив в константной памяти, то его размер необходимо указать на этапе компиляции, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается. Для записи с хоста в константную память используется функция `cudaMemcpyToSymbol`.

2.4. Текстурная память

Текстурная память (texture memory) входит в состав текстурных блоков, используемых в графических задачах для формирования текстур. В текстурном блоке аппаратно реализована фильтрация текстурных координат, интерполяция, нормализация текстурных координат в случаях, когда они выходят за допустимые пределы. Наличие кэша делала эффективным использование данного типа памяти на архитектурах с Compute Capability 1.x, однако с появлением архитектуры Fermi значимость данного фактора потеряла актуальность, т.к. появился собственный кэш для глобальной памяти.

Текстурная память выделяется с помощью функции `cudaMallocArray` и освобождается с использованием `cudaFreeArray`:

```
cudaError_t cudaMallocArray ( cudaArray_t* array, const cudaChannelFormatDesc* desc, size_t width, size_t height = 0, unsigned int  flags = 0 );

cudaError_t cudaFreeArray ( cudaArray_t array );
```

Для того чтобы получить доступ к выделенной памяти в CUDA-ядре требуется ассоциировать указатель с текстурной ссылкой:

```
cudaError_t cudaBindTexture ( size_t* offset, const textureReference* texref, const void* devPtr, const cudaChannelFormatDesc* desc, size_t size = UINT_MAX );
```

Для получения доступа к данным из CUDA-ядра используются специальные функции `tex1D`, `tex2D`, `tex3D`.

В связи с практически полной потерей актуальности текстурной памяти в задачах вычислений общего назначения, подробно указанные функции не рассматриваются.

Примечание [D3]:

2.5. Регистровая память

Регистровая память (register) является самой быстрой из всех типов памяти. Определить общее количество регистров, доступных GPU, можно с помощью функции `cudaGetDeviceProperties`.

Количество регистров, доступных одной нити, определяется следующим образом:

$$RegsPerThread = \frac{MaxRegsCount}{ThreadsPerBlock \cdot BlocksPerGrid},$$

где $MaxRegsCount$ – общее количество регистров, доступных GPU; $ThreadsPerBlock$ – число нитей в блоке; $BlocksPerGrid$ – число блоков в решетке блоков.

Все регистры GPU 32 разрядные. При этом в технологии CUDA нет явных способов использования регистровой памяти, всю работу по размещению данных в регистрах берет на себя компилятор.

2.6. Локальная память

Локальная память (local memory) может быть использована компилятором, если все локальные переменные не могут быть размещены в регистровой памяти. По скоростным характеристикам локальная память значительно медленнее, чем регистровая. Для проверки, используется ли локальная память, рекомендуется при компиляции включить опцию PTXAS Output в свойствах проекта в Microsoft© Visual Studio (опция --ptxas-options=-v при компиляции через командную строку) и проанализировать окно вывода информации о сборке проекта:

Число для ___, отличное от нуля, говорит о том, что компилятор не смог использовать только регистры и задействовал локальную память. В данной ситуации рекомендуется проанализировать код и уменьшить число используемых переменных.

2.7. Разделяемая память как управляемый программный кэш, регистровый кэш

Примечание [D4]: Не слишком ли сложно???

Латентность доступа к глобальной памяти крайне высока, поэтому в правильно спроектированном алгоритме чтение и запись элемента данных должно

осуществляться только один раз, при этом должны формироваться транзакции. Поэтому, как правило, ядро начинается с загрузки данных из глобальной в разделяемую память и заканчивается сохранением результата работы из разделяемой в глобальную память. Латентность доступа к разделяемой памяти в сравнении с глобальной гораздо ниже, однако она не нулевая. Поэтому при использовании одной нитью значений из разделяемой памяти более одного раза может оказаться эффективной программная реализация дополнительного кэша, построенного на уровне регистров (локальных переменных). В данные переменные размещаются наиболее часто используемые данные и в дальнейшем могут использоваться для аккумулирования промежуточного результата.

Однако работа с массивами переменных в качестве кэша данных является задачей, требующей осторожности и внимательности. Если компилятор не сможет распределить переменные по доступным регистрам, будет задействована локальная память, что приведет к существенному замедлению работы ядра. Поэтому для контроля используемого числа регистров одной нитью и количества байт задействованной локальной памяти рекомендуется воспользоваться рекомендациями из подраздела 2.7.

2.8. Задание

1. Выполнить оптимизацию доступа к глобальной памяти для задания из подраздела 1.5.
2. Проанализировать возможность использования регистрового кэша для задания из подраздела 1.5.

3. Синхронизация данных

Архитектура CUDA предлагает 2 основных способа синхронизации данных:

- синхронизация устройства, когда графический процессор гарантированно завершает поставленные на выполнение задачи, а работа центрального процессора для данного алгоритма блокируется до окончания работы GPU. При этом синхронизация может происходить как в явном виде (функция `cudaDeviceSynchronize`), так и в не явном виде (например, при использовании не асинхронной версии функции копирования данных `cudaMemcpy`);
- синхронизация нитей внутри блока.

Межблочная синхронизация нитей на момент написания методического пособия не предусмотрена технологией и осуществляется самостоятельно разработчиком собственными силами.

Кроме описанных выше способов можно выделить дополнительный способ синхронизации, который при правильном проектировании алгоритма позволяет избавиться синхронизации нитей внутри блока в явном виде, – использование свойств варпа. Данный способ будет описан ниже.

3.1. Функции синхронизации нитей внутри блока

К функциям синхронизации нитей внутри блока относятся следующие 4 функции – `__syncthreads`, `__syncthreads_count`, `int __syncthreads_and`, `__syncthreads_or`. Описание функций сведено в таблицу 3.1.

Таблица 3.1

Функции синхронизации нитей внутри блока

Функция	Compute Capability	Возвращаемое значение
1	**2**	**3**
void __syncthreads()	1.0 и новее	–
int __syncthreads_count(int predicate)	2.0 и новее	Число нитей, для которых выражение `predicate` не равно 0
int __syncthreads_and(int predicate)	2.0 и новее	Не нулевое значение только в том случае, если

		значение `predicate` для всех нитей блока не равно 0
int __syncthreads_or(int predicate)	2.0 и новее	Не нулевое значение только в том случае, если значение `predicate` хотя бы для одной нити блока не равно 0

3.2. Варп

Понятие варпа (warp) в архитектуре CUDA является одним из ключевых наравне с такими понятиями как нить, блок и решетка блоков. *Варп* представляет собой блок из 32 нитей мультипроцессора GPU, которые выполняют одну и ту же инструкцию, но каждый над своим блоком данных, т.е. фактически варп определяет минимальное число нитей, которые будут работать в один момент времени. Тот факт, что 32 нити работают синхронно, при правильном проектировании алгоритма (с учетом разбиения блоков записи и чтения в/из глобальной памяти по 32 нити) позволяет избавиться от операции `__syncthreads()`, обеспечивающей барьерную синхронизацию внутри блока потоков.

Рассмотрим следующий пример. Допустим, имеется некоторый массив длиной `elementCount` (тип данных – `int`). Требуется написать ядро, которое вычисляет среднее арифметическое двух соседних элементов и сохраняет результат в целочисленном виде. Схема стандартного подхода представлена на рисунке 3.1, цифрами на рисунке обозначены индексы элементов, которые задействуются.

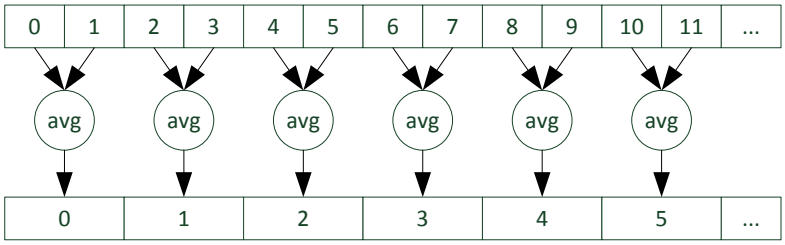

Рис. 3.1. Схема вычисления среднего арифметического

В представленном ниже алгоритме предполагается, что `elementCount` делится на 2. Код ядра, вычисляющего среднее арифметическое, показан ниже:

```
__global__ void average(
     int *inElements, int *outElements, unsigned elementCount
) {
        // разделяемая память, используемая для сохранения данных
        __shared__ int elementsBlock[512*2];

        // вычисляем смещение для текущего блока
        const int baseOffsetIn    = blockIdx.x * blockDim.x * 2;
        const int baseOffsetOut = blockIdx.x * blockDim.x;

        // сохраняем входной массив
        if ( ( baseOffsetIn + threadIdx.x ) < elementCount )
            elementsBlock[threadIdx.x] =
                        inElements[baseOffsetIn + threadIdx.x];
        if ((baseOffsetIn+threadIdx.x+blockDim.x) < elementCount )
            elementsBlock[threadIdx.x + blockDim.x] =
          inElements[baseOffsetIn + threadIdx.x + blockDim.x];

// синхронизация данных для гарантии того, что все элементы
// были загружены
        __syncthreads();

        // вычисляем среднее арифметическое и сохраняем данные
        if ( (baseOffsetOut + threadIdx.x) < (elementCount >> 1) )
            outElements[baseOffsetOut + threadIdx.x] =
               ( elementsBlock[threadIdx.x * 2] +
                 elementsBlock[threadIdx.x * 2 + 1] ) >> 1;
    }
```

Код запуска ядра выглядит следующим образом:

```
dim3 threads(512);
dim3 blocks( ( elementCount + 1023 ) / ( 2 * 512 ) );
average<<<blocks, threads>>>(
       inElements, outElements, elementCount
);
```

В конфигурации запуска ядра задается 512 нитей на блок (переменная `threads`) и динамически вычисляется число блоков (переменная `blocks`). При этом выражение `( elementCount + 1023 ) / ( 2 * 512 )` гарантирует правильное число блоков при условии, что остаток от деления `elementCount` на 1024 не равен нулю. Таким образом, весь массив разбивается на блоки по 1024 элемента в каждом, при этом каждая нить обрабатывает по 2 соседних элемен-

та. Для соблюдения условия формирования транзакций (см. подраздел 2.2) чтение 1024 элементов разбивается на 2 этапа – последовательно по 512 элементов с сохранением результата в разделяемую память (переменная `elementsBlock`).

Вызов функции `__syncthreads()` гарантирует, что все 1024 элемента были загружены и размещены в разделяемой памяти. Если убрать функцию синхронизации, существует не нулевая вероятность получить некорректный результат. После синхронизации осуществляется расчет среднего арифметического и сохранение результата в заданную позицию.

Если задействовать свойства варпа, можно реорганизовать ядро таким образом, чтобы функция `__syncthreads()` стала лишней. Преобразуем загрузку элементов следующим образом: вместо блоков 2 по 512 элементов, будем загружать блоками 2 по 32 элемента (в соответствии с размером варпа). То есть первых 32 нити загружает последовательно 64 элемента данных, необходимых для вычисления среднего арифметического, следующие 32 нити еще 64 элемента, которые необходимы, и так далее. Разбиение на блоки по 32 в соответствии со свойством синхронности нитей варпа гарантирует, что все необходимые данные будут доступны без явной синхронизации. Получившийся код ядра представлен ниже:

```
__global__ void average(int *inElements, int *outElements,
                        unsigned elementCount) {
    // разделяемая память, используемая для сохранения данных
    __shared__ int elementsBlock[512*2];

    // индекс нити в реорганизованной структуре
    const int threadId = threadIdx.x & 31;
    // индекс варпа
    const int warpId = threadIdx.x >> 5;

    // вычисляем глобальное смещение для текущего блока
    const int baseOffsetIn  = blockIdx.x * blockDim.x * 2;
    const int baseOffsetOut = blockIdx.x * blockDim.x;

    // вычисляем локальное смещение внутри блока
    const int localOffsetIn = warpId * 32 * 2 + threadId;
// в данном примере выходное локальное смещение
// совпадает с threadIdx.x,
// однако в общем случае это может быть не справедливо
    const int localOffsetOut = warpId * 32 + threadId;

    // сохраняем входной массив
```

```
    if ( ( baseOffsetIn + localOffsetIn ) < elementCount )
        elementsBlock[localOffsetIn] =
                 inElements[baseOffsetIn + localOffsetIn];
    if ( (baseOffsetIn + localOffsetIn + 32) < elementCount )
        elementsBlock[localOffsetIn + 32] =
            inElements[baseOffsetIn + localOffsetIn + 32];

    // вычисляем среднее арифметическое и сохраняем данные
    if ((baseOffsetOut + localOffsetOut) < (elementCount>>1))
        outElements[baseOffsetOut + localOffsetOut] =
           (elementsBlock[localOffsetOut*2] +
            elementsBlock[localOffsetOut*2+1])>>1;
}
```

3.3. Атомарные операции

Атомарные операции представляют собой набор функций, обеспечивающий гарантированное чтение – модификацию – сохранение одного 32- либо 64-битного слова в глобальной или разделяемой памяти. Сохранение при этом происходит по тому же адресу, из которого было произведено чтение данных. В отличие от стандартных операций загрузки, модификации и сохранения, в атомарных операциях гарантируется:

- полное выполнение операции. Если, например, при чтении будет сгенерировано исключение, операция модификации и сохранения выполнена не будет;
- отсутствие влияния других нитей на результат работы, т.е. другие нити будут вынуждены ожидать полного завершения выполнения данной операции, чтобы прочитать из нее данные.

При этом накладываются следующие ограничения:

- атомарные операции доступны только при исполнении на устройстве (обязательна метка `__device__` либо `__global__`);
- не применимы к памяти выделенной как Mapped page-locked memory.

На современных архитектурах графического процессора поддерживаются операции сложения, вычитания, операции над битами. Актуальный список атомарных операций и их описание, а также версию Compute Capability, начиная с которой данная операция поддерживается, можно найти в официальной документации [1].

3.4. NVIDIA© Visual Profiler

Профилировщик NVIDIA© Visual Profiler является основным средством для анализа производительности разработанного приложения с применением технологии CUDA. Данное приложение доступно для всех основных операционных системам (Windows, Linux, Max OS) и входит в стандартный набор NVIDIA© CUDA Toolkit.

Настройка приложения (настройка сессии в терминологии CUDA) для подготовки к анализу сводится лишь к указанию приложения, которое разработчик хочет протестировать, указанию рабочей директории и параметров командной строки при их наличии (рисунок 3.2). В примере на рисунке 3.2 осуществляется запуск приложения SampleTest.exe для тестирования и указана рабочая папка «K:\!Projects\».

Результат работы профилировщика с настройками по-умолчанию для тестового приложения (пример из раздела 3.2) показан на рисунке 3.3. Центральную часть экрана приложения занимает линия времени (timeline), на которой отображается последовательность запусков CUDA-функций и ядер с привязкой ко времени. Это позволяет быстро оценить, на какие действия тратится максимум времени, чтобы разработчик в дальнейшем мог оптимизировать это. Справа при наведении курсора мыши на ядро на линии времени появляются полученные свойства ядра. Данные свойства позволяют узнать полученную производительность ядра (теоретическую и практическую), число регистров, приходящихся на одну нить и другие параметры. Снизу экрана отображаются все функции-ядра, вызываемые в приложении и их краткая характеристика и, что самое важное, отображается реальное время работы ядра.

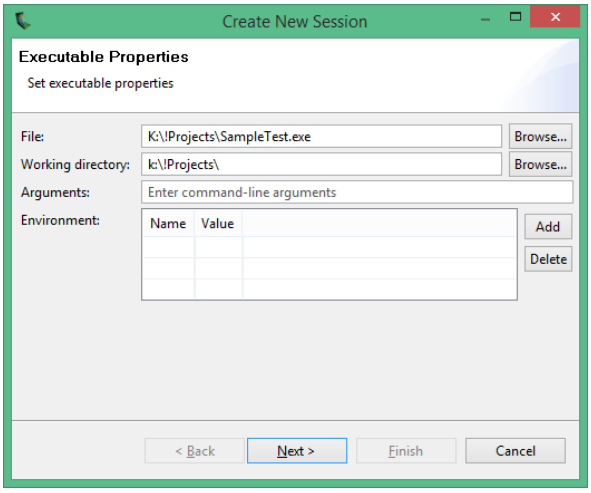

Рис. 3.2. Создание новой сессии в профилировщике

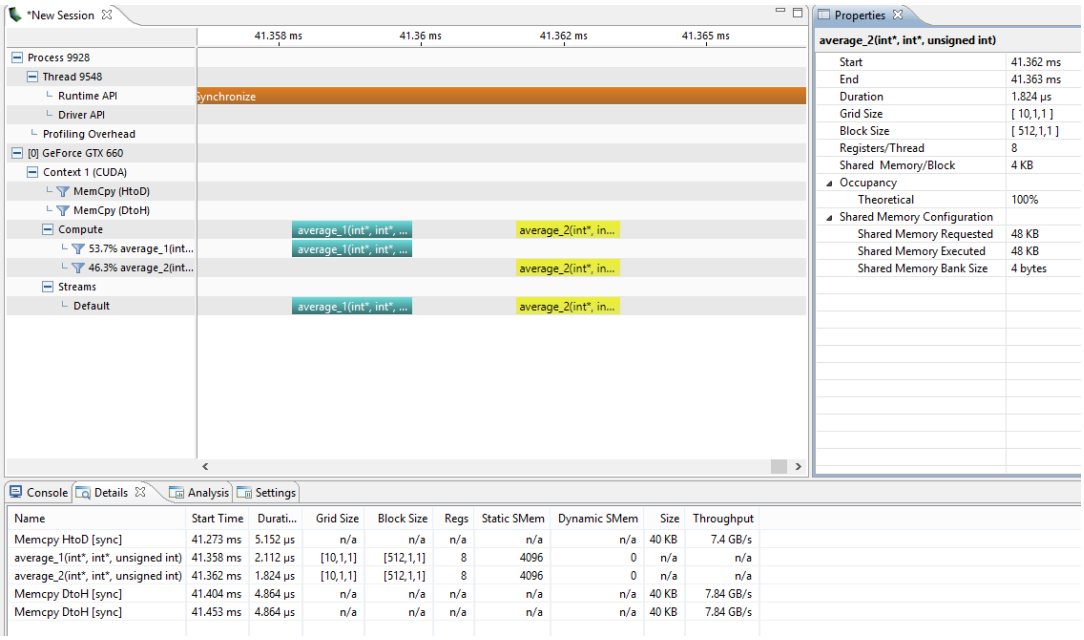

Рис. 3.3. Результат работы профилировщика с настройками по-умолчанию

После профилирования с настройками по-умолчанию через контекстное меню Run можно вызвать диалоговое окно, в котором указать характеристики, которые требуется измерить (рисунок 3.4).

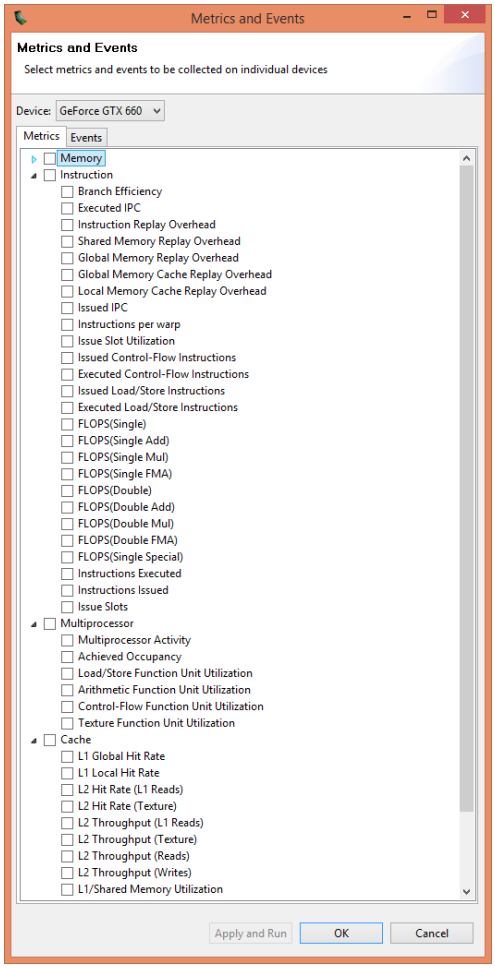

Рис. 3.4. Диалоговое окно выбора метрик и событий для анализа

Все метрики сгруппированы в следующие группы: память, инструкции, мультипроцессор, кэш, текстуры. Данные метрики позволяют оценить эффективность работы с глобальной памятью (например, путем оценки теоретической и эффективной пропускной способности шины), работу мультипроцессора и другие характеристики. Более подробное описание всех метрик можно найти в официальной документации [].

Важной составляющей профилировщика является встроенная система анализа производительности приложения (вкладка Analysis). При выборе ядра, которое требуется проанализировать, становится доступными следующие режимы анализа:

- «Kernel Performance Limiter» – определяет, является ли вычислительное устройство ограничителем производительности;

- «Kernel Latency» – определяет, насколько качественно осуществляется сокрытие латентности операций работы с памятью и арифметических операций;
- «Kernel Compute» – оценка утилизации мультипроцессоров ядра видеокарты;
- «Kernel Memory» – анализ эффективной пропускной способности памяти;
- «Memory Access Pattern» – анализ шаблона обращения к памяти;
- «Divergent Execution» – анализ наличия дивергенции потоков;
- «Data Movement and Concurrency» – оценка эффективности перекрытия вычислительной нагрузки с функциями копирования данных (cudaMemcpy);
- «Compute Utilization» – анализ вычислительной нагрузки;
- «Kernel Performance» – оценка производительности ядра.

После запуска анализа выводятся результаты работы. При этом в случае, если система обнаружила критические проблемы, выдается предупреждение об этом и рекомендации, как устранить данную проблему. На рисунке 3.5 представлен результат анализа ядра average после удаления из кода `__syncthreads()` в режиме «Kernel Latency».

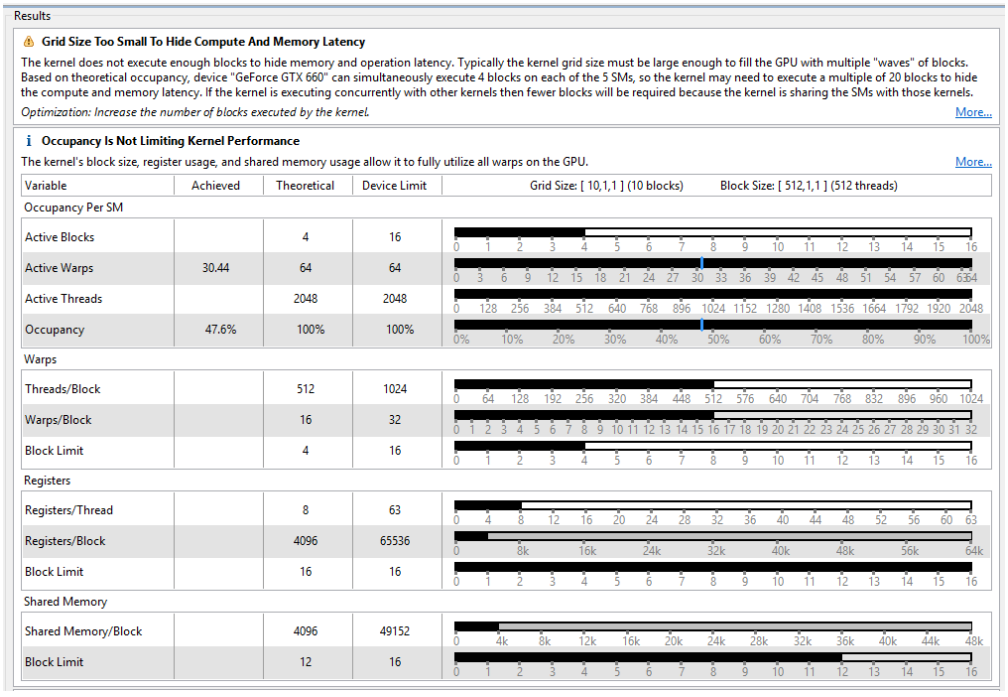

Рис. 3.5. Результат анализа в режиме «Kernel Latency»

Как видно из рисунка 3.5, используемая в тесте конфигурация запуска является не эффективной, т.к. не позволяет качественно нагрузить имеющиеся мультипроцессоры видеокарты. При этом система рекомендовала увеличить число блоков, используемых в CUDA-ядре.

3.5. Базовые алгоритмы с использованием CUDA

К базовым относятся следующие алгоритмы: редукция и операция вычисления префиксной суммы (scan).

Редукция – сумма значений всех элементов массива. Графически схема простейшего решения данной задачи представлена на рисунке 3.6. В качестве входного массива для алгоритма используется разделяемая память.

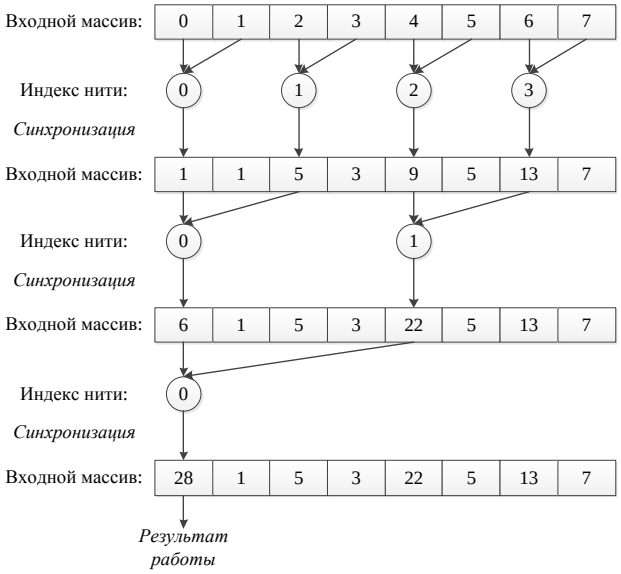

Рис. 3.6. Алгоритм редукции. Базовая версия

Код алгоритма редукции показан ниже:

```
__global__ void reduceBase(int* inputData, int* outputData) {
    extern __shared__ int smemData[];
    // загружаем данные в разделяемую память
    const unsigned tid = threadIdx.x;
    smemData[tid] =
            inputData[blockIdx.x * blockDim.x+ threadIdx.x];
```

```
    // синхронизируемся
    __syncthreads();

    // выполняем редукцию над элементами массива
    for( unsigned s = 1; s < blockDim.x; s <<= 1 ) {
        if( tid % ( s << 1 ) == 0 ) {
            smemData[tid] += smemData[tid + s];
        }
        // синхронизируемся, чтобы гарантировать
        // корректность входных данных
        // на следующей итерации цикла
        __syncthreads();
    }

    // сохраняем результат вычислений
    if ( tid == 0 )
        outputData[blockIdx.x] = smemData[0];
}
```

Анализ кода на производительность указывает на несколько ключевых проблем в ядре:

- дивергенция потоков при выполнении условия: `tid%(s<<1) == 0`;
- возможно нарушение доступа по банкам разделяемой памяти: `smemData[tid] += smemData[tid + s]`;
- большое число вызовов функции синхронизации `__syncthreads()`.

Для решения описанных проблем представим схему вычисления как показано на рисунке 3.7.

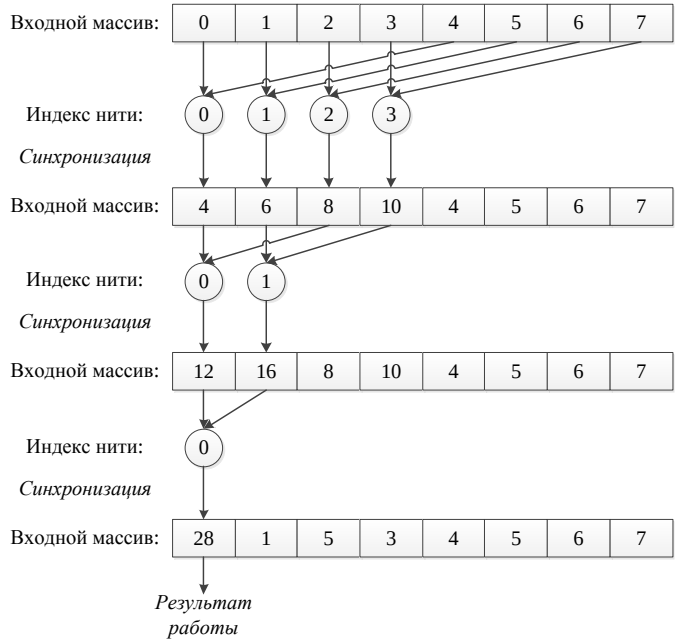

Рис. 3.7. Алгоритм редукции. Оптимизированная версия

Последовательное обращение нитей к памяти позволяет решить проблему доступа к банкам разделяемой памяти. Кроме того появляется возможность сразу провести первое суммирование при чтении данных из глобальной памяти, тем самым убрав одну синхронизацию. Для этого достаточно правильной настройки конфигурации и чтение данных из глобальной памяти должно осуществляться с шагом `blockDim.x`, т.е. `sum = inputData[threadIdx.x] + inputData[threadIdx.x + blockDim.x]`. Данную оптимизацию читателю предлагается реализовать самостоятельно. Более детальное описание алгоритма и сравнение производительности можно найти в книге [4].

Префиксной суммой называется массив следующего вида: $\{0, a_0, \{a_0 \circ a_1\}, \{a_0 \circ a_1 \circ a_2\}, ..., \{a_0 \circ a_1 \circ a_2 \circ ... \circ a_{n-2}\}\}$, где $\circ$ – любое арифметическое действие. Простейшая версия последовательной реализации алгоритма префиксной суммы сводится к следующему:

```
sum[0] = 0;
for (int i = 0; i < size; i++)
    sum[i] = sum[i – 1] ° a[i – 1];
```

В качестве примера в дальнейшем выбрано арифметическое действие – сумма. Параллельная реализация является не тривиальной задачей и сводится к

двум подзадачам – построение дерева сумм (рисунок 3.8) и на основе полученного дерева построение префиксных сумм (рисунок 3.9).

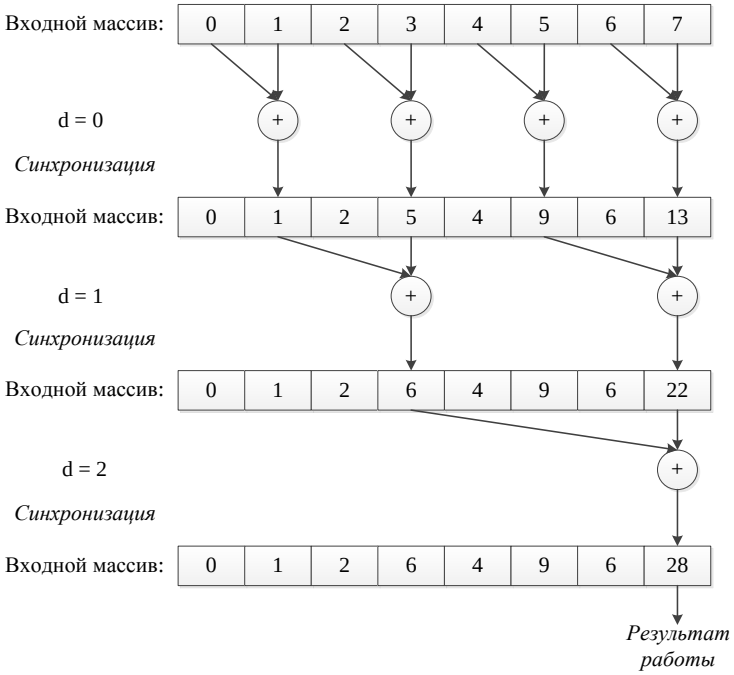

Рис. 3.8. Построение дерева сумм

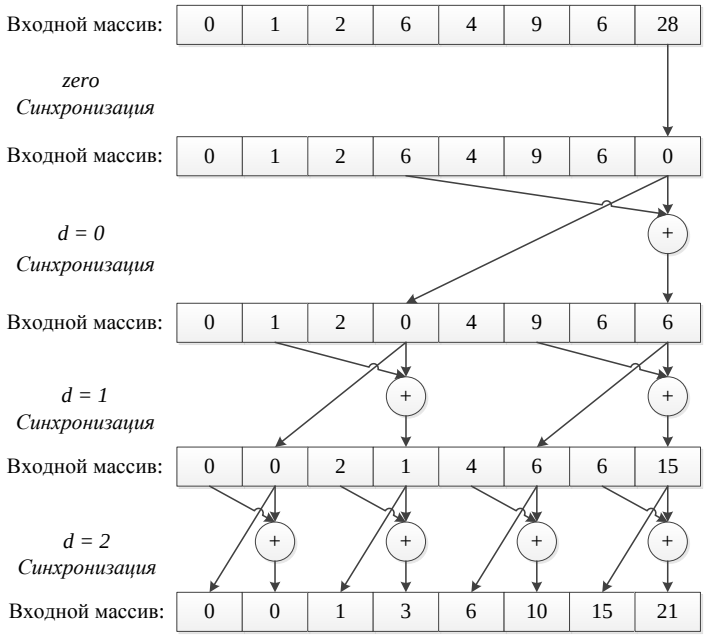

Рис. 3.9. Построение префиксных сумм по дереву сумм

Задачей первой части алгоритма является нахождение сумм элементов массива (по-аналогии с редукцией). После этого обнуляется последний элемент, содержащий сумму всех элементов. Далее начинается обработка пар, начиная с пары элементов, удаленных на половину длины блока и заканчивая парами соседних элементов. На каждом шаге один из элементов пары копируется на место второго, а на место первого записывается сумма исходных элементов. Код функции-ядра, реализующий данный алгоритм, представлен ниже:

```
const int blockSize = 256;

__global__ void scan( float *inData, float *outData, int size)
{
    __shared__ float temp[2 * blockSize];
    int tid = threadIdx.x;
    int offset = 1;

    // чтение данных
    temp[tid] = inData[tid];
    temp[tid + blockSize] = inData[tid + blockSize];

    // построение дерева сумм
    for (int d = n >> 1; d > 0; d >>= 1) {
        __syncthreads();

        if ( tid < d ) {
            int ai = offset * ( 2 * tid + 1 ) - 1;
            int bi = offset * ( 2 * tid + 2 ) - 1;

            temp[bi] += temp[ai];
        }

        offset <<= 1;
    }

    // обнуление одного элемента
    if ( tid == 0 ) {
        temp[n - 1] = 0;
    }

    // построение префиксных сумм по дереву сумм
    for ( int d = 1; d < n; d <<= 1 ) {
```

```
        offset >>= 1;
        __syncthreads();

        if ( tid < d ) {
            int ai = offset * ( 2 * tid + 1 ) - 1;
            int bi = offset * ( 2 * tid + 2 ) - 1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

    outData[2 * tid] = temp[2 * tid];
    outData[2 * tid + 1] = temp[2 * tid + 1];
}
```

Основной проблемой данного алгоритма являются постоянные конфликты доступа к банкам памяти вплоть до 16 порядка. Читателю предлагается самостоятельно подумать над тем, как устранить данную проблему.

3.6. Задание

1. Оценить с помощью NVIDIA Visual Profiler качество реализации алгоритма из подраздела 2.8.
2. Оптимизировать алгоритм префиксной суммы с учетом рекомендаций из подраздела 3.5.

4. Обработка изображения

Обработка графической информации является одним из тех направлений, для которых технология CUDA может обеспечить существенное ускорение.

Реализация алгоритмов в области обработки изображений связана с решением ряда существенных проблем, актуальных для текущих архитектур графического процессора:

- эффективное распределение нагрузки между мультипроцессорами и вычислительными ядрами;
- формирование транзакций при чтении и записи данных из/в глобальную память (подраздел 2.1);
- специфические особенности конкретных алгоритмов (например, проблема обработки краев при фильтрации изображения);

4.1. Проблема загрузки изображения

Одна из проблем, с которой сталкивается разработчик при реализации любого алгоритма обработки изображений, - формирование транзакций при доступе к глобальной памяти. При этом можно выделить две случая, которые могут привести к нарушению шаблона доступа:

- один пиксель представляет собой 3 байта данных (каналы RGB);
- отсутствие выравнивания в памяти для второй и последующих строк исходного изображения.

При чтении данных в ядре из глобальной памяти разработчик должен решить проблему эффективной загрузки в разделяемую память. Проблема заключается в том, что пиксель для большинства алгоритмов представляет собой 3-байтную последовательность данных. Простейший способ загрузки – побайтное чтение данных – приводит к нарушению шаблона доступа к памяти и в результате транзакции не формируются, т.к. требуется последовательное обращение нитей к 4-байтным данным (подробнее в подразделе 2.1). На современных архитектурах за счет наличия кэша частично данная проблема сглаживается, однако это все равно приводит к существенному замедлению работы алгоритма.

Эффективное решение данной проблемы сводится к одному из двух решений. Первое решение заключается в добавлении канала альфа к каждому пикселю изображения, однако, с одной стороны, это приводит к увеличению требуемой глобальной памяти, а с другой стороны – требуется реализация алгоритма конверсии из 3 байтного формата в 4 байтный и обратно, что увеличивает полное время работы программы.

Второй способ эффективного решения проблемы заключается в том, чтобы одна нить загружала не один пиксель, а сразу 4 (рисунок 4.1). Как видно из рисунка 4.1, для этого потребуется выполнить 3 загрузки данных типа `int`.

Рис. 4.1. Демонстрация загрузки и сохранения данных

Фрагмент кода, выполняющий загрузку в соответствии с рисунком 4.1, показан ниже:

```
__global__ void filter(
           unsigned *input, const unsigned inputWidth, …) {
      __shared__ unsigned smem[(BLOCK_DIM_X*3)*BLOCK_DIM_Y];
      const int baseX = (blockIdx.x*blockDim.x+threadIdx.x)*3;
      const int baseY = blockIdx.y * blockDim.y + threadIdx.y;

      smem[threadIdx.y * BLOCK_DIM_X * 3 + threadIdx.x] =
           input[baseY * inputWidth + baseX];
      smem[threadIdx.y*BLOCK_DIM_X*3+threadIdx.x+blockDim.x] =
           input[baseY * inputWidth + baseX + blockDim.x];
      smem[threadIdx.y*BLOCK_DIM_X*3+threadIdx.x+blockDim.x*2]
           = input[baseY*inputWidth+baseX + blockDim.x * 2];

      __syncthreads();
      …
}
```

В представленном фрагменте кода предполагается, что `BLOCK_DIM_X` и `BLOCK_DIM_Y` – число нитей по осям X и Y соответственно конфигурации ядра, `inputWidth` – ширина изображения в 4-байтных словах.

Проблема с негарантированным формированием транзакций при чтении второй и последующих строк заключается в том, что данные строки должны быть выравнены на границу 128 байт. В общем случае при произвольной ши-

рине изображения данное условие гарантируется только для первой строки, т.к. функция `cudaMalloc` осуществляет выравнивание стартового адреса.

Решением данной проблемы является искусственное увеличение ширины изображения до границы 128 байт при инициализации памяти устройства (рисунок 4.2). В рамках технологии CUDA предлагается 2 способа решения:

- ручное выравнивание и расширение изображения;
- использование CUDA API.

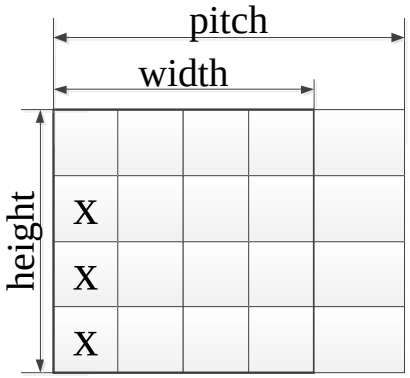

Рис. 4.2. Выравнивание строк изображения

Фрагмент кода при ручном выравнивании представлен ниже:

```
void cudaInit(
      char *inputImg,     // входное изображение
      const int width,    // ширина в байтах
      const int height,   // высота изображения
      …
) {
      …
      int *d_inputImg = NULL;
      // выравниванием ширину изображения
      const int pitch = ( ( width + 127 )  / 128 ) * 128;
      // выделяем память с учетом новой ширины
      cudaMalloc( (void **)&d_inputImg, pitch * height );
      // копируем данные с расширением на новую ширину
      for (int i = 0; i < height; i++) {
           cudaMemcpy(
                &d_inputImg[i * pitch], //адрес приемника
                &inputImg[i * width], // адрес источника
                 width, // число байт для копирования
```

```
                cudaMemcpyHostToDevice
            );
        }
        …
}
```

При использовании CUDA API достаточно воспользоваться функцией выделения памяти:

```
cudaError_t cudaMallocPitch (
void** devPtr,  // указатель на выделенную память
size_t* pitch,  // фактически сформированная ширина в байтах
size_t width,   // реальная ширина изображения в байтах
size_t height   // реальная высота изображения
)
```

В результате выполнения данной функции будет выделено как минимум `width * height` байт. При этом в переменной `pitch` будет возвращено фактически выделенная ширина изображения в байтах с учетом выравнивания.

Для расширения массива до pitch байт CUDA API предлагает использовать следующую функцию:

```
cudaError_t cudaMemcpy2D (
void* dst,      // приемник
size_t dpitch,  // фактическая ширина приемника в байтах
const void* src,     // источник
size_t spitch,  // фактическая ширина источника в байтах
size_t width,   // реальная ширина копируемых данных в байтах
size_t height,  // реальная высота копируемых данных
cudaMemcpyKind kind  // направление копирования
)
```

В результате будет выполнено копирование `height` строк по `width` байт каждая из `src` в `dst`. При этом смещение для строки `i` приемника будет определяться как `i * dpitch`, а для источника – `i * spitch`.

Фрагмент кода, демонстрирующий работу с CUDA API, показан ниже:

```
void cudaInitAPI(
      char *inputImg,     // входное изображение
      char *outputImg,    // выходное изображение
      const int width,    // ширина в байтах
      const int height,   // высота изображения
      …
```

```
) {
      …
      int *d_inputImg = NULL;
      int pitch = 0;
      cudaMalloc((void **)&d_inputImg, &pitch, width, height);
      cudaMemcpy2D(
           d_ inputImg, pitch,
           inputImg, width,
           width, height,
           cudaMemcpyHostToDevice
      );
      …
// можем работать с d_ inputImg, при этом ширина – pitch байт
      cudaMemcpy2D(
           outputImg, width,
           d_ inputImg, pitch,
           width, height,
           cudaMemcpyDeviceToHost
      );
      // в outputImg ширина изображения снова width байт
      …
}
```

4.2. Проблема границ. Разделение ядер на краевые и центральные

Еще одна проблема, с которой сталкивается разработчик, - обработка границы изображения. В особенности это касается правой и нижней частей изображения, т.к. требуется обработать возможный выход за пределы обрабатываемой области. Несмотря на то, что изображение расширяется до границы 128 байт, тратить ресурсы и обрабатывать участок, содержащий «мусор», появляющийся за счет расширения, не всегда имеет смысл. Поэтому в код ядра добавляется условный оператор, ограничивающий участок обработки.

Для центральных блоков данный оператор не имеет смысла, т.к. все нити блока «примут» одинаковое решение. В результате дивергенция потоков не сформируется, однако ресурсы на обработку условия затрачены будут. Для того чтобы этого избежать, ядро разбивается на две ветки:

- первая – обрабатывает центральные блоки изображения и исключает все проверочные условные операторы;
- вторая – обрабатывает только правый и нижний края изображения.

4.3. Разбиение на блоки

Формирование конфигурации исполнения ядра является важной задачей, т.к. даже идеально оптимизированное ядро при неправильной конфигурации способно существенно замедлить работу.

Исходя из практики разработки ядер для обработки цветных изображений, наиболее оптимальным числом нитей по оси X изображения для большинства задач является 32. Данное число нитей позволяет использовать свойства варпа для синхронизации нитей. Обрабатываемое число пикселей в этом случае – 128, что, с одной стороны, позволяет обрабатывать сравнительно небольшие по ширине изображения, а с другой стороны увеличить число блоков для более качественной нагрузки имеющихся потоковых мультипроцессоров (Streaming Multiprocessor).

Четких критериев для определения числа нитей по оси Y нет и зависит от решаемой задачи и вычислительной сложности ядра.

4.4. Конкатенация ядер

4.5. Задание

1. Свертка изображения с фильтром 3x3

5. Оптимизация доступа к памяти и технология CUDA Stream

5.1. Pinned память

Как уже отмечалось, операции копирования между ОЗУ и памятью видеокарты обладают существенной латентностью. Для оптимизации передачи данных рекомендуется обращение с формированием транзакций, как механизма, позволяющего максимально эффективно задействовать шину памяти.

Все современные операционные системы работают в режиме с виртуальной адресацией памяти. Для каждого запущенного процесса выделяется свое адресное пространство, в рамках которого процесс может осуществлять чтение и запись данных. Сторонний процесс не может получить доступ на чтение/запись в области памяти текущего процесса без специального разрешения со стороны ОС.

В большинстве операционных систем виртуальное адресное пространство делится на страницы по как минимум 4096 байт. Для доступа к физическому адресу данных в процессор встроена таблица PTE (Page Table Entry), осуществляющая трансляцию между виртуальным и физическим адресами. В случае если запрашиваемая страница не загружена, генерируется исключение, которое должна обработать ОС. Схема работы показана на рисунке 5.1.

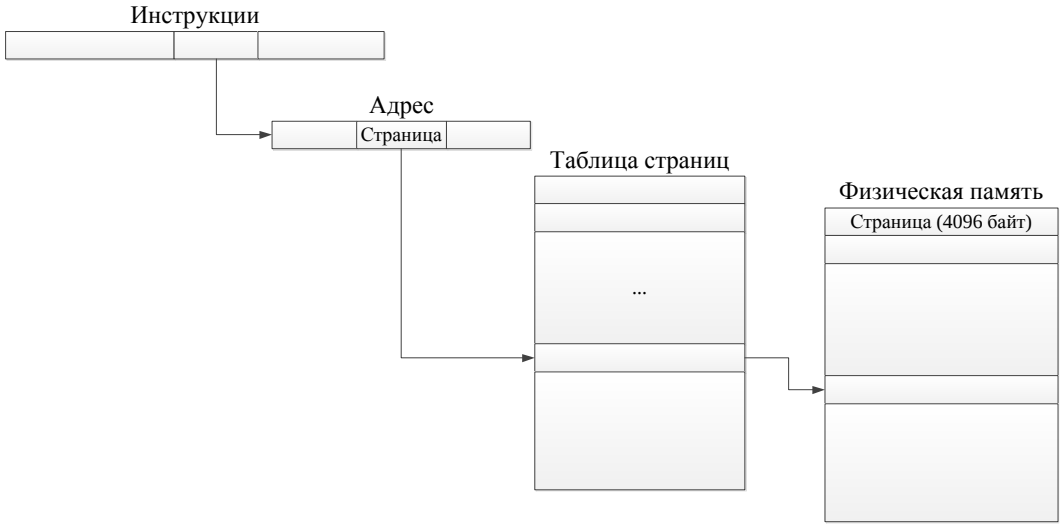

Рис. 5.1. Виртуальное адресное пространство

Для организации доступа к глобальной памяти на видеокарте используется упрощенный механизм виртуальной адресации – сохраняется безопасность и локальность в памяти для конкретной CUDA программы, однако убрана страничная адресация. Различия в механизме адресации для CPU и GPU привели к тому, что доступ к памяти является сравнительно сложной задачей:

- определяется номер страницы, содержащей необходимые данные в ОЗУ, и ее наличие в памяти;
- осуществляется копирование страницы в специальный pinned-буфер;
- выполняется синхронное копирование страницы в память GPU с использованием DMA;
- освобождаются запрошенные ресурсы.

Данный алгоритм повторяется для каждой последующей страницы.

Очевидно, что из-за большого числа операций приблизиться к пиковой пропускной способности шины PCI Express не представляется возможным.

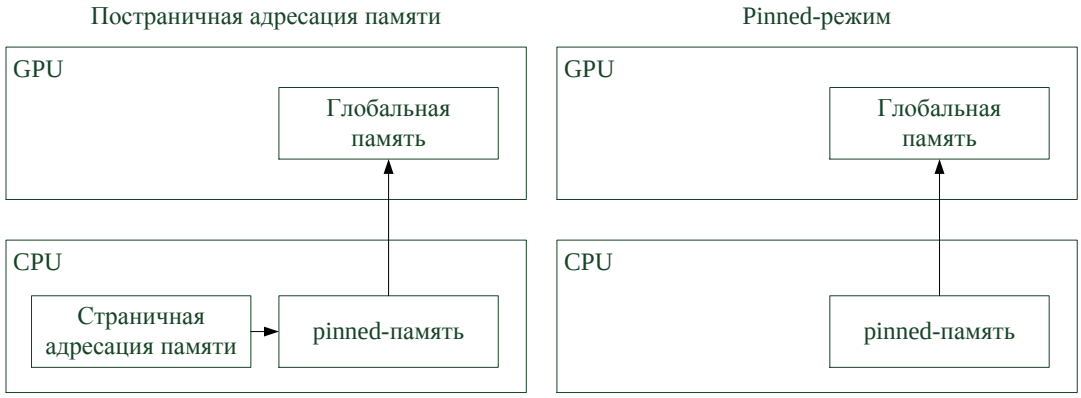

Рис. 5.2. Сравнение передачи данных в постраничном и pinned-режимах

Для решения данной проблемы в CUDA API предложен способ выделения памяти в ОЗУ непосредственно в pinned-буфере в обход постраничной адресации. С одной стороны, разработчик теряет в объеме доступно памяти ОЗУ и возможность кэшировать данные на диск, а с другой – существенно упрощает механизм копирования из ОЗУ в память GPU и приближается к пиковой пропускной способности PCI Express. Для выделения памяти в pinned-режиме достаточно выделить память в ОЗУ с применением следующей функции:

```
cudaError_t cudaMallocHost (
      void** ptr,    // указатель на выделяемую память в ОЗУ
      size_t size    // объем выделяемой памяти
```

```
) ;
```

Заранее выделенная системная память может быть также зарегистрирована в pinned-режиме с использованием CUDA API:

```
cudaError_t cudaHostRegister (
 void * ptr,     // указатель на память ОЗУ
 size_t size,    // размер регистрируемой памяти в байтах
 unsigned int flags
)
```

К параметрам управления (к флагам) относятся следующие допустимые значения:

- `cudaHostRegisterPortable` – зарегистрированная память переводится в pinned-режим для всех контекстов, а не только для текущего;
- `cudaHostRegisterMapped` – проецирует выделенную память в адресное пространство CUDA. Указатель на память в GPU может быть получен с помощью `cudaHostGetDevicePointer()`.

Однако следует учитывать, что передача данных может быть ускорена, но в целом система станет работать медленнее, особенно если задействовать в алгоритмах заранее выделенную системную память и осуществлять ее регистрацию с использованием `cudaHostRegister()`.

5.2. Унифицированная память (Unified Memory)

Данная технология была введена в CUDA 6.0 и поддерживается архитектурой Kepler и более новыми. Основная задача заключается в том, чтобы полностью скрыть от разработчика различия в CPU и GPU памяти. Рассмотрим следующий пример:

CPU

```
void sampleFunc(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);
  fread(data, 1, N, fp);

  testFunction(data, N);

  free(data);
```

CUDA 6

```
void sampleFunc(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);
  fread(data, 1, N, fp);

  testFunction<<<…>>>(data, N);

  cudaFree(data);
```

} }

Код слева – код для исполнения на CPU, код справа – для GPU. Основное отличие – в выделении памяти. Для этого используется функция `cudaMallocManaged()`.

By migrating data on demand between the CPU and GPU, Unified Memory can offer the performance of local data on the GPU, while providing the ease of use of globally shared data. The complexity of this functionality is kept under the covers of the CUDA driver and runtime, ensuring that application code is simpler to write. The point of migration is to achieve full bandwidth from each processor; the 250 GB/s of GDDR5 memory is vital to feeding the compute throughput of a Kepler GPU.

5.3. CUDA Stream

Достижение максимальной производительности возможно только при максимальной утилизации всех имеющихся ресурсов (в случае с технологией CUDA – CPU и GPU). Данный принцип лежит в основе управления нитями на GPU: как только нити варпа переходят в состояние ожидания получения результата, планировщик пытается найти варп, который может начать либо продолжить выполнение следующих операций, и переключается на него. В результате при правильном проектировании конфигурации системы возможно достигнуть идеального баланса производительности для GPU, когда задержки из-за ожидания выполнения операции сводятся к минимуму.

Кроме того асинхронный запуск ядра позволяет одновременно с расчетами задействовать ресурсы CPU. Однако данный подход не позволяет полностью перекрыть операции копирования с хоста на устройство и наоборот. Для решения данной проблемы NVIDIA© предлагает использовать CUDA Stream. CUDA Stream представляет собой очередь команд для GPU, для которого справедливы следующие правила:

- команды в рамках одного stream выполняются строго последовательно;
- команды из разных stream могут выполняться по мере освобождения необходимых ресурсов, т.е. возможно параллельное исполнение команд из разных stream.

Примечание [D5]: Как их назвать на русском?

По-умолчанию все команды выполняются в 0 stream’е, который создается автоматически системой. Стандартная последовательность команд (копирование с хоста на устройство, выполнение ядра, копирование результата с устройства на хост) при использовании только одного stream’а показана на рис.5.3. Используемый при этом код выглядит следующим образом:

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<<1, N>>>(d_a);
cpuFunction();
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Рис. 5.3. Исполнение при использовании stream'а по-умолчанию

Основная проблема в данной ситуации – простаивание GPU при копировании данных и простаивание шины при исполнении ядра.

Для использования различных stream'ов требуется реорганизовать код следующим образом:

- ядро `kernel` для каждого stream'а должно обрабатывать свой локальных участок данных полностью независимо по отношению к остальным stream'ам (например, при обработке изображений – каждый stream обрабатывает свой рисунок);
- функции копирования требуется заменить на их асинхронные версии;
- инициализировать необходимое число stream'ов.

Инициализация stream'а выглядит следующим образом (все представленные команды будут относиться к одному stream'у):

```
cudaStream_t stream1;
// создаем stream1
cudaStreamCreate(&stream1);
// асинхронное копирование на устройство
cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
// запуск ядра
kernelStream<<<1, N, 0, stream1>>>(d_a);
… // дальнейшая работа со stream1, CPU
// разрушаем определенный stream1
cudaStreamDestroy(stream1);
```

Для определения того, что использование CUDA Stream может обеспечить повышение производительности, на этапе инициализации рекомендуется выполнить следующие действия:

- проверить флаг `deviceOverlap` в структуре `cudaDeviceProp`. В соответствии с документации все устройства с Compute Capability 1.1 и новее данные флаг установлен в истинное значение;
- память на хосте должна быть определена как pinned.

Для синхронизации, например, со `stream1` введена функция `cudaStreamSynchronize(stream1)`, которая блокирует хост до тех пор, пока не будет полностью выполнены все команды для `stream1`. Для того чтобы узнать завершены команды stream'а или нет без блокировки хоста используется функция `cudaStreamQuery(stream)`.

Новая версия кода с использование `nStreams` stream'ов показана ниже:

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(
                  &d_a[offset],
                  &a[offset],
                  streamBytes,
                  cudaMemcpyHostToDevice,
                  stream[i]
  );
  kernel<<>>(d_a, offset);
  cudaMemcpyAsync(
                  &a[offset],
                  &d_a[offset],
                  streamBytes,
                  cudaMemcpyDeviceToHost,
                  stream[i]
  );
}
```

Рис. 5.4. Перекрытие операций с использование различных stream'ов

Для представленного кода возможно обеспечить перекрытие, показанное на рис. 5.4, начиная с архитектуры NVIDIA© Fermi и более новых. Это связано с аппаратной организацией системы очередей. Изначально в архитектуре NVIDIA© Tesla содержалось две очереди – одна для копирования, одна для исполнения ядер. Команды на исполнение поступают в очередь в порядке их следования по коду, в результате в очередь на копирование попадает вначале команда копирования с хоста на устройство для `stream[0]`, затем с устройства на хост для этого же stream'а и так далее для всех `nStream` stream'ов. Система выполняет команды последовательно, в результате очередь блокируется из-за необходимости копирования с устройства на хост, а для этого требуется выполнить ядро. В результате для архитектуры NVIDIA© Tesla данный код не даст никакого повышения производительности. Начиная с архитектуры NVIDIA© Fermi, число буферов копирования было расширено до двух устройств (одно для копирования с хоста на устройство, второе – в обратном направлении), что определяет эффективность работы алгоритма.

Примечание [D6]: Перед Fermi ??

Использование stream'ов будет заблокировано или прервано, если:

- переменная окружения CUDA_LAUNCH_BLOCKING = 1;
- выделение page-locked памяти;
- выделение и инициализация памяти на устройстве;
- любая команда из CUDA API, работающая с нулевым stream'ом (stream'ом по-умолчанию);
- переключение между режимами, управляющими L1кэшем и разделяемой памятью для Compute Capability 2.x и Compute Capability 3.x.

5.4. Задание

Оптимизировать алгоритм из задания раздела 4 для использования технологии CUDA Stream.

6. Список литературы

[1] CUDA C Programming Guide [Электронный ресурс]. – 2014. – Режим доступа: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_ Guide.pdf.

[2] BrookGPU [Электронный ресурс]. – 2014. – Режим доступа: http://graphics.stanford.edu/projects/brookgpu/.

[3] The open standard for parallel programming of heterogeneous systems[Электронный ресурс]. – 2014. – Режим доступа: http://www.khronos.org/opencl/.

[4] Параллельные вычисления на GPU. Архитектура и программная модель CUDA / А. Боресков [и др.]. – М. : Изд-во МГУ, 2012. – 336 с.

[5] Wilt, N. The CUDA Handbook: A Comprehensive Guide to GPU Programming / N. Wilt. – Addison Wesley, 2013. – 528 p.

[6] Cook, S. CUDA Programming (Applications of GPU Computing Series) / S. Cook. – Morgan Kaufmann, 2012. – 560 p.

[7] Farber, R. CUDA Application Design and Development / R. Farber. – Morgan Kaufmann, 2011. – 336 p.

[8] Kirk, D. Programming Massively Parallel Processors: A Hands-on Approach / D. Kirk, Wen-Mei Hwu ; 2nd edition. – Morgan Kaufmann, 2012. – 514 p.