Московский государственный университет им. М.В. Ломоносова

Параллельное программирование для многопроцессорных систем с общей и распределенной памятью

Разработчики:
В.И. Лаева, e-mail: lvi@math.tsu.ru
А.А. Трунов, e-mail: trunov@math.tsu.ru
Томский государственный университет

Направление 010400.62
«Прикладная математика и информатика»

Проект комиссии Президента по модернизации и техническому развитию экономики России
«Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения»

Содержание курса

- Введение в параллельное программирование с использованием стандарта OpenMP. Параллельные области
- Параллельные циклы. Секции. Директивы master, single, workshare, threadprivate.
- Синхронизация в OpenMP
- Введение в параллельное программирование с использованием технологии CUDA
- Иерархия памяти ГПУ и работа с ней в CUDA
- Работа с разделяемой памятью. Синхронизация в CUDA

Содержание лекции

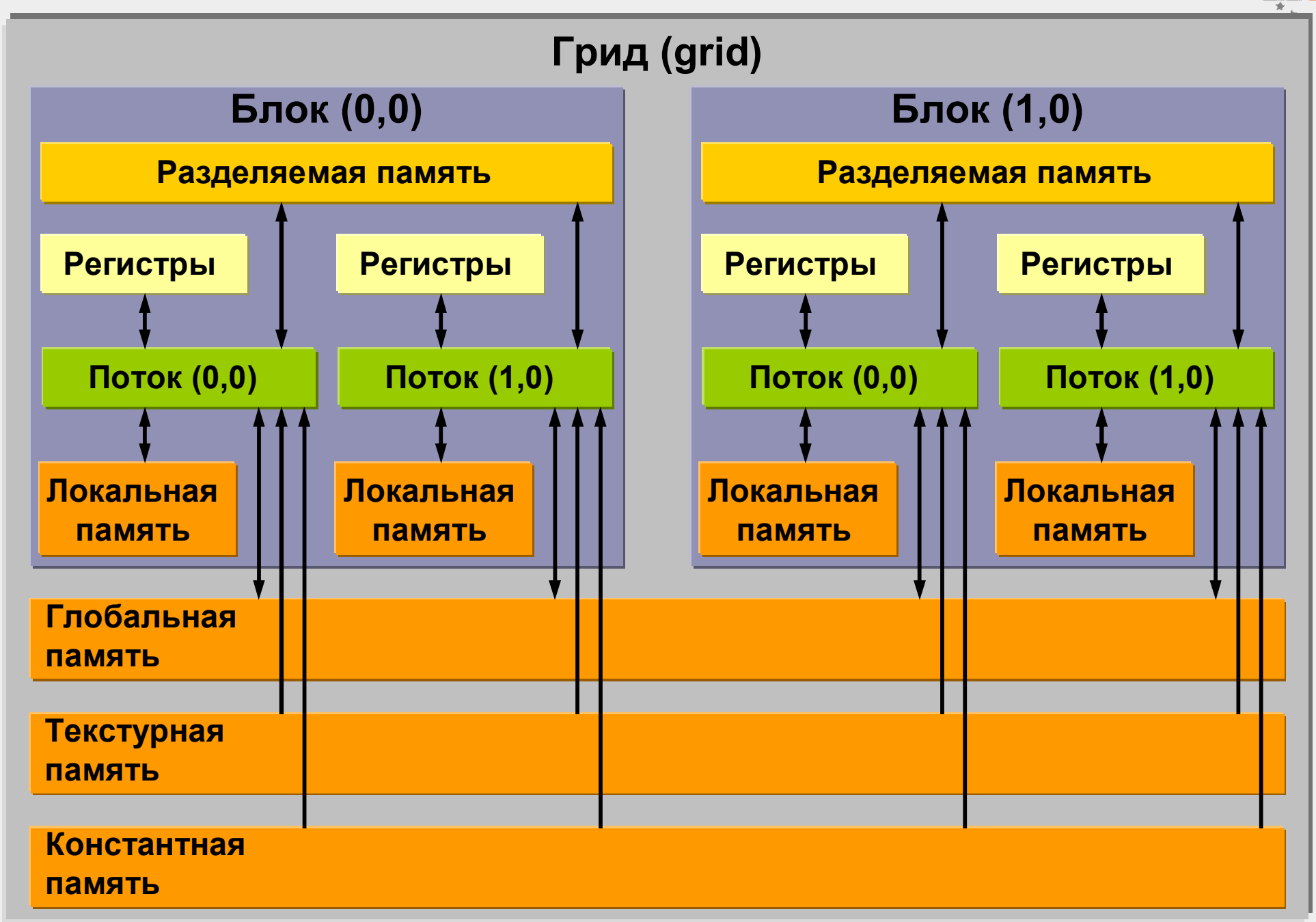
- Иерархия памяти.
- Работа с глобальной памятью: эффективный доступ.
- Работа с разделяемой памятью: кэширование.
- Работа с разделяемой памятью: конфликты доступа.
- Ветвления.

Иерархия памяти

- Каждый поток имеет доступ к регистрам
- Каждый поток имеет собственную локальную память (local memory), физически находящуюся в глобальной памяти ГПУ
- Каждый блок потоков имеет разделяемую память (shared memory), доступную потокам только в пределах блока. Размер ~ 10 кБ.
- Разделяемая память блока хранит данные только во время выполнения блока.

Иерархия памяти

- Все потоки имеют доступ к глобальной памяти (global memory). Размер ~ 1 - 10 ГБ.
- Константная память (constant memory). Только для чтения. Физически находится в глобальной памяти. Кэшируется.
- Текстурная память (texture memory). Только для чтения. Физически находится в глобальной памяти. Кэшируется.


Грид (grid)
Блок (0,0)
Разделяемая память
Регистры
Регистры
Поток (0,0)
Поток (1,0)
Локальная память
Локальная память
Блок (1,0)
Разделяемая память
Регистры
Регистры
Поток (0,0)
Поток (1,0)
Локальная память
Локальная память
Глобальная память
Текстурная память
Константная память

Спецификаторы переменных

Спецификатор	Размещение	Примечание
<нет>	регистры или локальная память	на усмотрение компилятора
`__device__`	глобальная память	
`__constant__`	константная память	
`__shared__`	разделяемая память	
`__restrict__`	-	ограниченные указатели

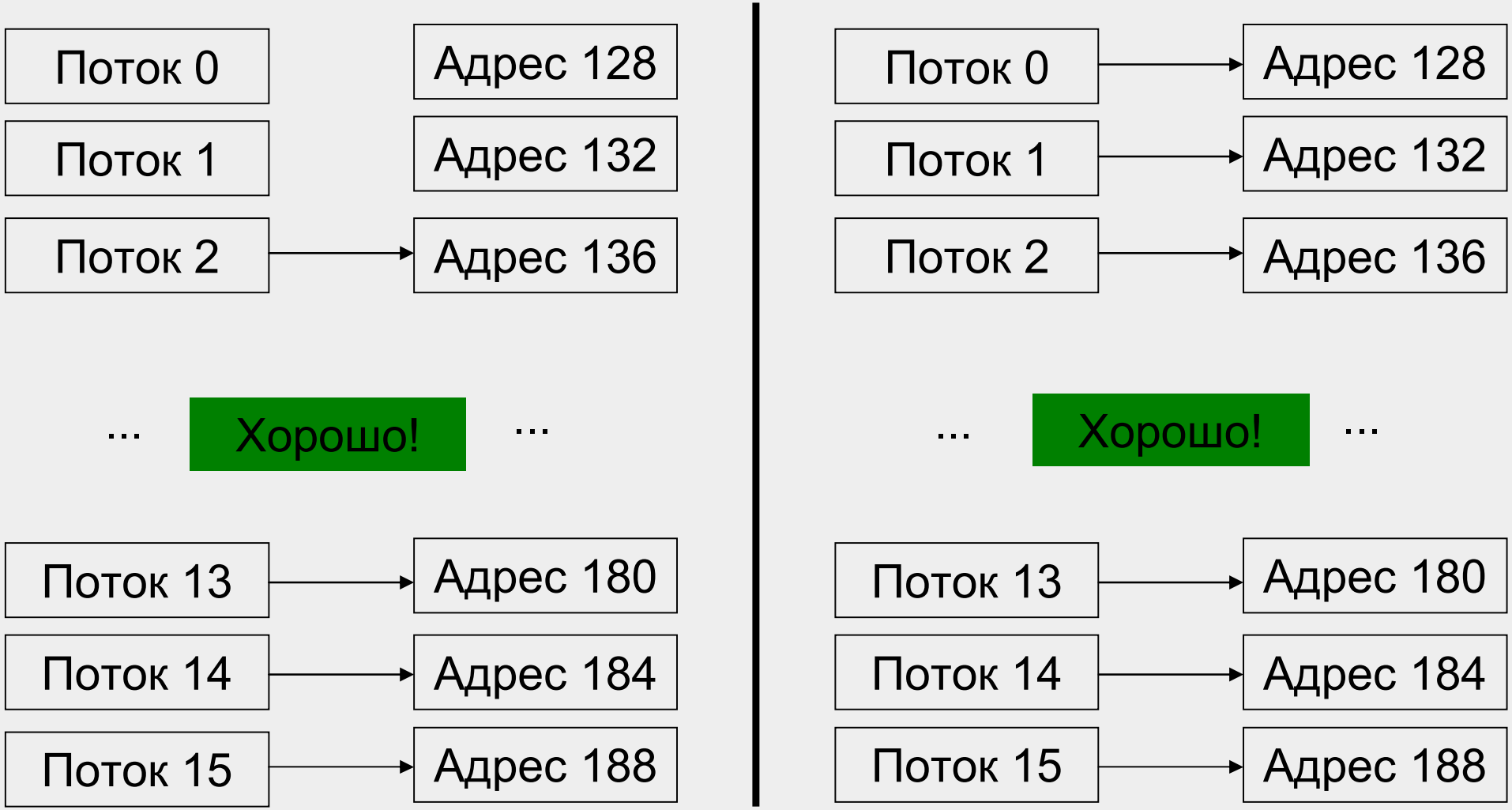
Работа с глобальной памятью: эффективный доступ

- Важно объединять запросы к памяти от нескольких потоков в один запрос.
- Границы запрашиваемых из глобальной памяти данных должны быть выровнены (aligned) по 32-, 64- или 128-байтным сегментам памяти.
- Для архитектур предыдущих поколений (compute capability 1.x) важным является согласованный (coalesced) доступ: номер потока должен определённым образом быть согласован со смещением адреса.

Работа с глобальной памятью: эффективный доступ

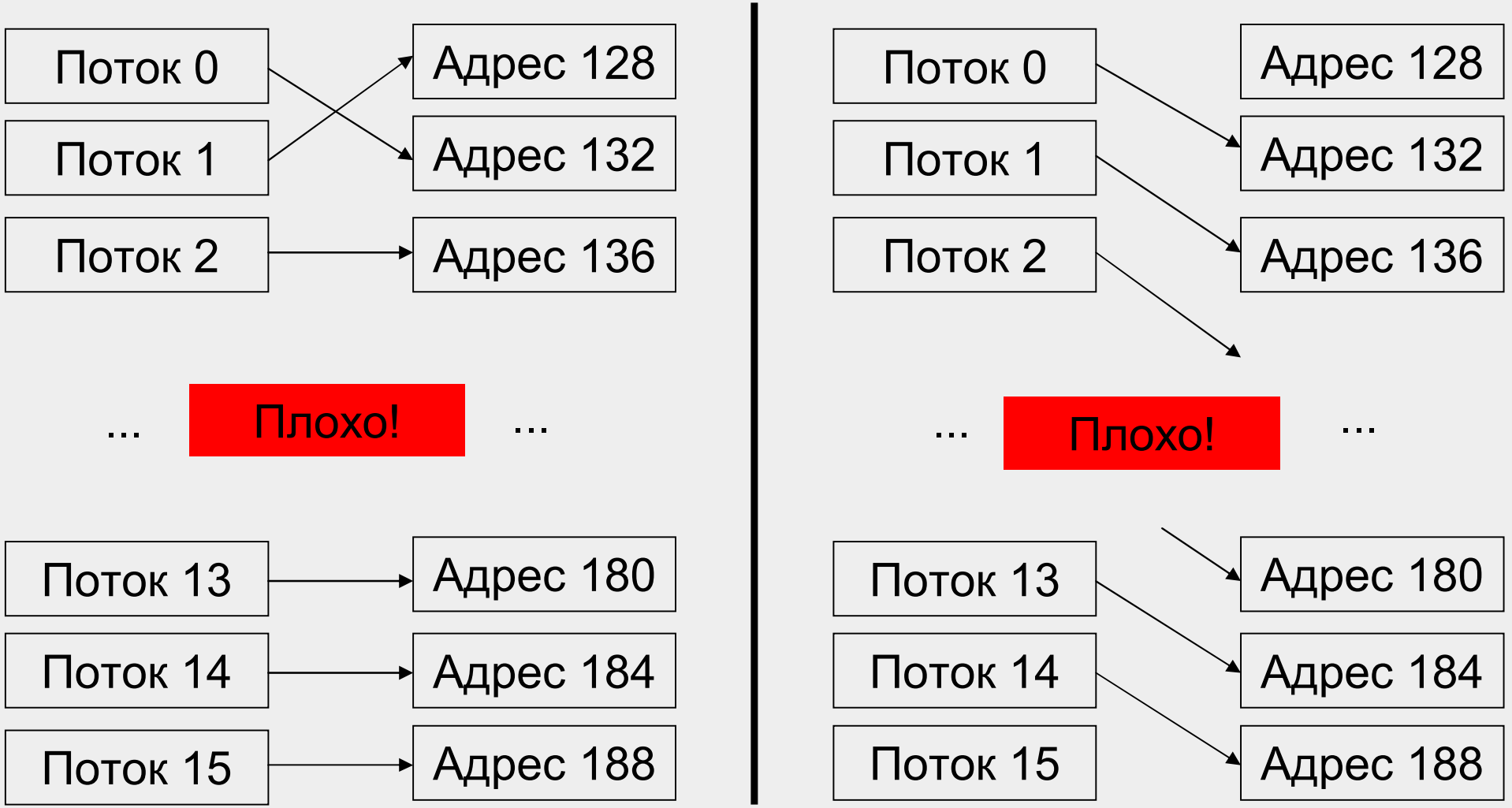
- Важным для быстродействия является принцип локальности данных: потоки в пределах полуварпа/варпа должны обращаться к смежным адресам в глобальной памяти.
- Один из простых шаблонов: последовательное обращение к смежным адресам памяти в пределах варпа (warp)
- Матрицы в языках C/C++ хранятся "по строкам"
- Матрицы в языке Fortran хранятся "по столбцам"

Работа с глобальной памятью: эффективный доступ



Compute capability 1.0 и 1.1

Работа с глобальной памятью: эффективный доступ



Compute capability 1.0 и 1.1

Работа с глобальной памятью: эффективный доступ

- согласованный (coalesced) доступ к элементам вектора

```
unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
```

- несогласованный доступ к элементам вектора

```
unsigned int i = (blockIdx.x * blockDim.x + threadIdx.x * 513) % N;
```

N - размер вектора

blockDim.x = 32

Работа с разделяемой памятью: кэширование

- ГПУ с версией архитектуры < 2.0 (compute capability < 2.0) не кэшируют запросы к глобальной памяти.
- Для достижения высокой производительности часто приходится реализовывать ручное кэширование (повторное использование данных).
- Для кэширования используется разделяемая память.

Работа с разделяемой памятью: кэширование

- Типовой шаблон работы с разделяемой памятью:
  - загрузить небольшую часть данных из глобальной памяти в разделяемую память блока;
  - синхронизировать потоки в пределах блока;
  - провести вычисления с данными из разделяемой памяти;
  - сохранить результат в глобальной памяти.

Все потоки в пределах блока копируют данные из ОЗУ в разделяемую память

Барьерная синхронизация в пределах блока потоков

Вычисления с использованием разделяемой памяти

Все потоки в пределах блока копируют данные из ОЗУ в разделяемую память

Работа с разделяемой памятью: конфликты доступа

- Эффективный доступ к разделяемой памяти также имеет свой шаблон.
- Разделяемая память разбита на 16/32 банков.
- Каждый из банков способен выполнить одно чтение или запись 32-битового слова.
- К каждому из этих банков потоки полуварпа могут обратиться параллельно.
- Если несколько потоков (но не все!) в пределах полуварпа обращаются к одному банку, то происходит конфликт доступа к банку.

Работа с разделяемой памятью: конфликты доступа

- В случае конфликта доступа обращение к банкам происходит последовательно, что снижает производительность.
- Шаблон: обращение по последовательным адресам разделяемой памяти приводит к обращению к разным банкам.

Матрично-векторное произведение: версия 1.0

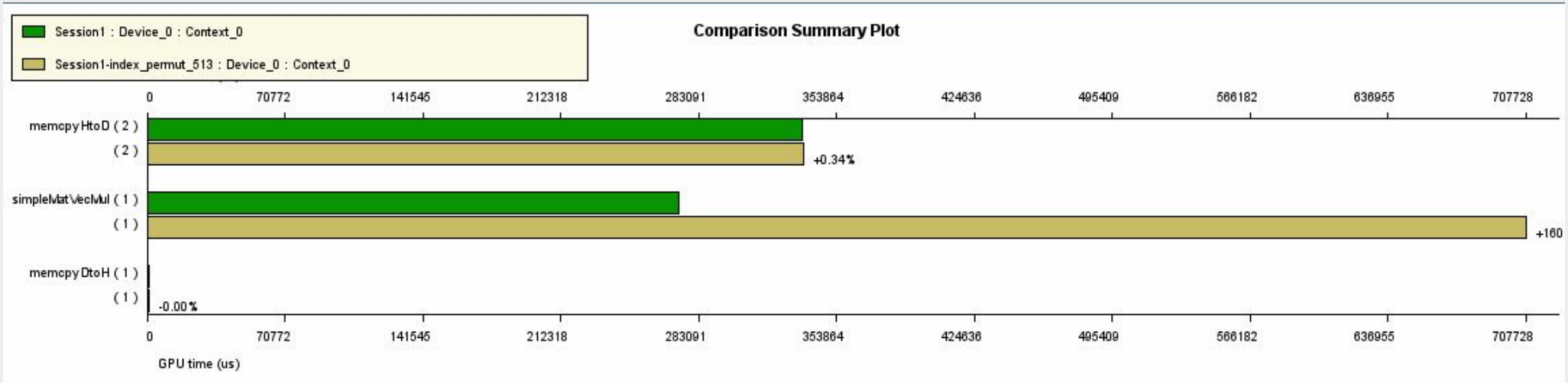
```
__global__ void simpleMatVecMul (float *A, float *x, float *y)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < N; i++)
      sum += A[row*N + i] * x[i];
    y[row] = sum;
}
```

Матрично-векторное произведение: версия 1.1 (non-coalesced, кэш-промахи)

```
__global__ void simpleMatVecMul (float *A, float *x, float *y)
{
    int row = (blockIdx.x * blockDim.x + 513 * threadIdx.x) % N;
    float sum = 0.0f;
    for (int i = 0; i < N; i++)
      sum += A[row*N + i] * x[i];
    y[row] = sum;
}
```

```
N = 16000
blockDim.x = 32
```

Матрично-векторное произведение: профилирование версий 1.0 и 1.1



Параметры запуска:

N = 16000

blockDim.x = 32

Матрично-векторное произведение: версия 2 (повторное использование вектора x)

```
__global__ void shared_x_MatVecMul (float *A, float *x, float *y)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    __shared__ float x_tile[BLOCK_SIZE];

    for (int i = 0; i < N/BLOCK_SIZE; i++) {
       x_tile[threadIdx.x] = x[i*BLOCK_SIZE + threadIdx.x];
       __syncthreads();
       for (int j = 0; j < BLOCK_SIZE; j++)
          sum += A[row*N + i*BLOCK_SIZE + j] * x_tile[j];
    }
    y[row] = sum;
}
```

Матрично-векторное произведение: версия 3

```
__global__ void shared_A_x_MatVecMul (float *A, float *x, float *y)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    __shared__ float x_tile[BLOCK_SIZE];
    __shared__ float A_tile[BLOCK_SIZE][BLOCK_SIZE];

    for (int i = 0; i < N/BLOCK_SIZE; i++) {
       x_tile[threadIdx.x] = x[i*BLOCK_SIZE + threadIdx.x];
       for (int k = 0; k < BLOCK_SIZE; k++)
          A_tile[k][threadIdx.x] = A[(blockIdx.x*blockDim.x + k)*N +
          i*BLOCK_SIZE + threadIdx.x];
       __syncthreads();
       for (int j = 0; j < BLOCK_SIZE; j++)
          sum += A_tile[threadIdx.x][j] * x_tile[j];
    }
    y[row] = sum;
}
```

Ветвления

- Все потоки в пределах варпа выполняют одну инструкцию (SIMT).
- При выполнении условных конструкций каждый поток в пределах варпа выполняет ветку if и ветку else, что уменьшает производительность.
- Если для данного потока инструкция не должна быть выполнена, то результаты его работы не сохраняются.

Ветвления

- Для нормальной работоспособности вышеприведённые примеры программ должны быть снабжены условными конструкциями, предупреждающими некорректные операции с памятью, если размер задачи не кратен кол-ву потоков в блоке.
- Модифицированные примеры теряют в производительности.

Вопросы для обсуждения

- Почему следует избегать многократной пересылки данных между ЦПУ и ГПУ?
- Для чего используется разделяемая память?
- Почему пример 2 (матрично-векторное произведение) на ГПУ с кэш-памятью не даст прироста производительности относительно версии 1.0?
- Для чего используется функция __syncthreads()?
- Для чего применяется копирование блока матрицы A в разделяемую память в примере 3, если каждый элемент матрицы используется только 1 раз?
- Что произойдёт, если блок матрицы A при копировании будет транспонироваться?
- Возможно ли использовать рассмотренные в лекции подходы для оптимизации программ ЦПУ?

Темы заданий для самостоятельной работы

- Параллельная реализация матрично-векторного произведения с повторным использованием данных при помощи разделяемой памяти
- Параллельная реализация матрично-матричного произведения с повторным использованием данных при помощи разделяемой памяти
- Параллельная реализация произведения матрицы с её транспонированной с повторным использованием данных при помощи разделяемой памяти и устранением конфликтов доступа к банкам разделяемой памяти

Литература

- http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
- http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
- http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html
- Сандерс Дж., Кэндрот Э. Технология CUDA в примерах. Введение в программирование графических процессоров. М.: ДМК Пресс, 2011 г., ISBN 978-5-94074-504-4, 978-0-13-138768-3
- Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. - 232 с.: ил. ISBN 978-5-94074-578-5

Содержание курса

- Введение в параллельное программирование с использованием стандарта OpenMP. Параллельные области
- Параллельные циклы. Секции. Директивы master, single, workshare, threadprivate.
- Синхронизация в OpenMP
- Введение в параллельное программирование с использованием технологии CUDA
- Иерархия памяти ГПУ и работа с ней в CUDA
- Работа с разделяемой памятью. Синхронизация в CUDA