




Казанский  
федеральный  
университет

ВЫСШАЯ ШКОЛА  
информационных технологий  
и информационных систем

# Оптимизация алгоритмов CUDA

# Параллельные алгоритмы

- ▶ Производительность разных версий параллельного алгоритма на CUDA может отличаться на порядки
  - ▶ Оптимизация алгоритма – ускорение работы с памятью, применение векторизации, развертка циклов и т.д.
  - ▶ Рассмотрим методы оптимизации алгоритма на примере алгоритма параллельной редукции
- 

# Редукция

- ▶ Дан массив  $a[n]$ , и операция  $op$
- ▶ Редукция  $A = a[0] op a[1] op \dots op a[n-1]$
- ▶ Рассмотрим редукцию массива данных по сумме
- ▶ Последовательная реализация редукции тривиальна

```
for(int i = 0; i < n; ++i) sum += a[i];
```


# Параллельная редукция

- ▶ Метод «разделяй и властвуй»
- ▶ Исходный массив делится на части и каждую часть обрабатывает свой блок
- ▶ Независимое нахождение частичных сумм
- ▶ Если массив очень большой используется «сканирующее окно»

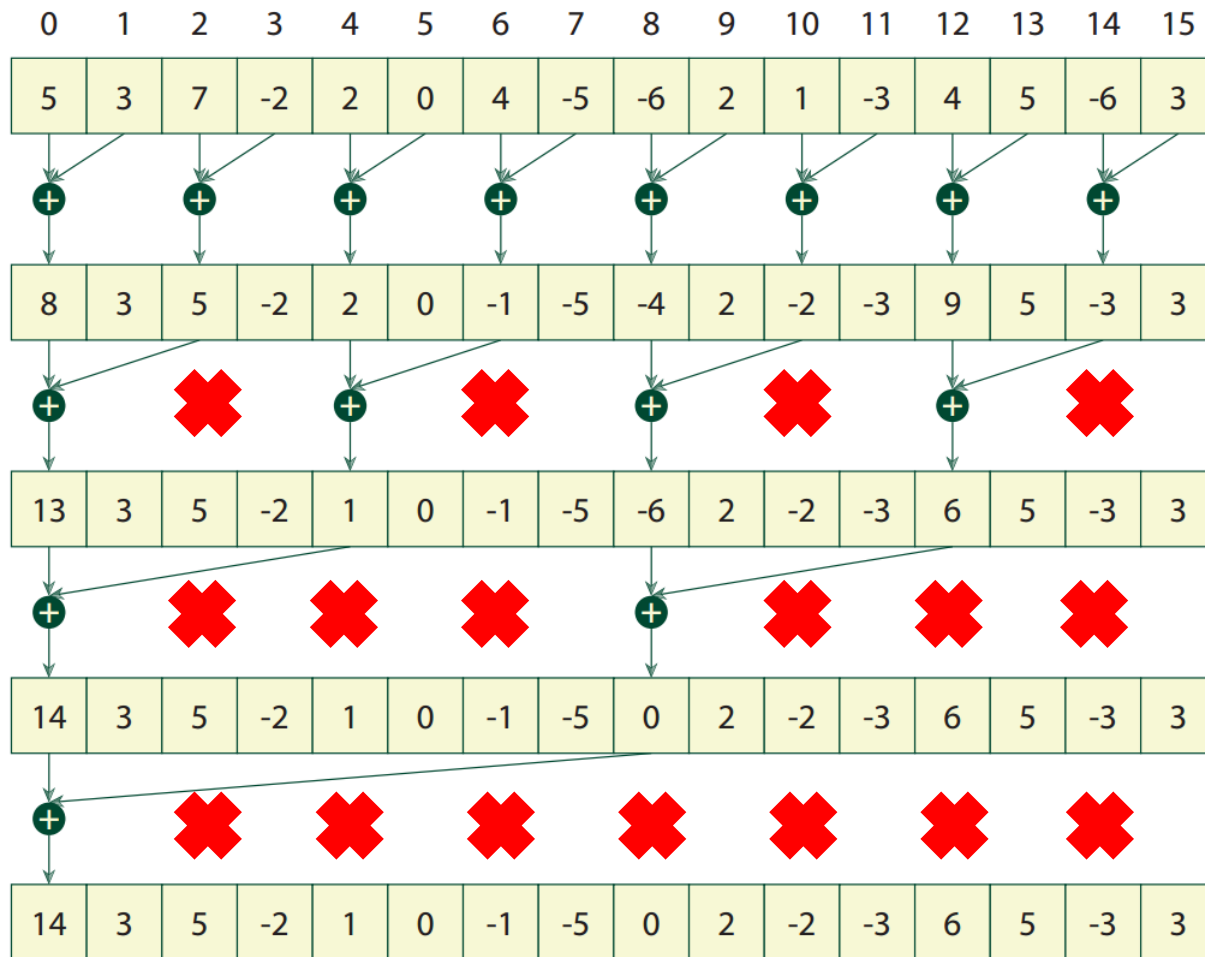
# Параллельная редукция

- ▶ Внутри каждого блока также введем параллелизацию по отдельным нитям
- ▶ Сначала суммируются попарно соседние элементы, потом их частичные суммы, и т.д.
- ▶ На каждом этапе суммирования число шагов уменьшается 2
- ▶ Количество шагов =  $O(\log_2 n)$

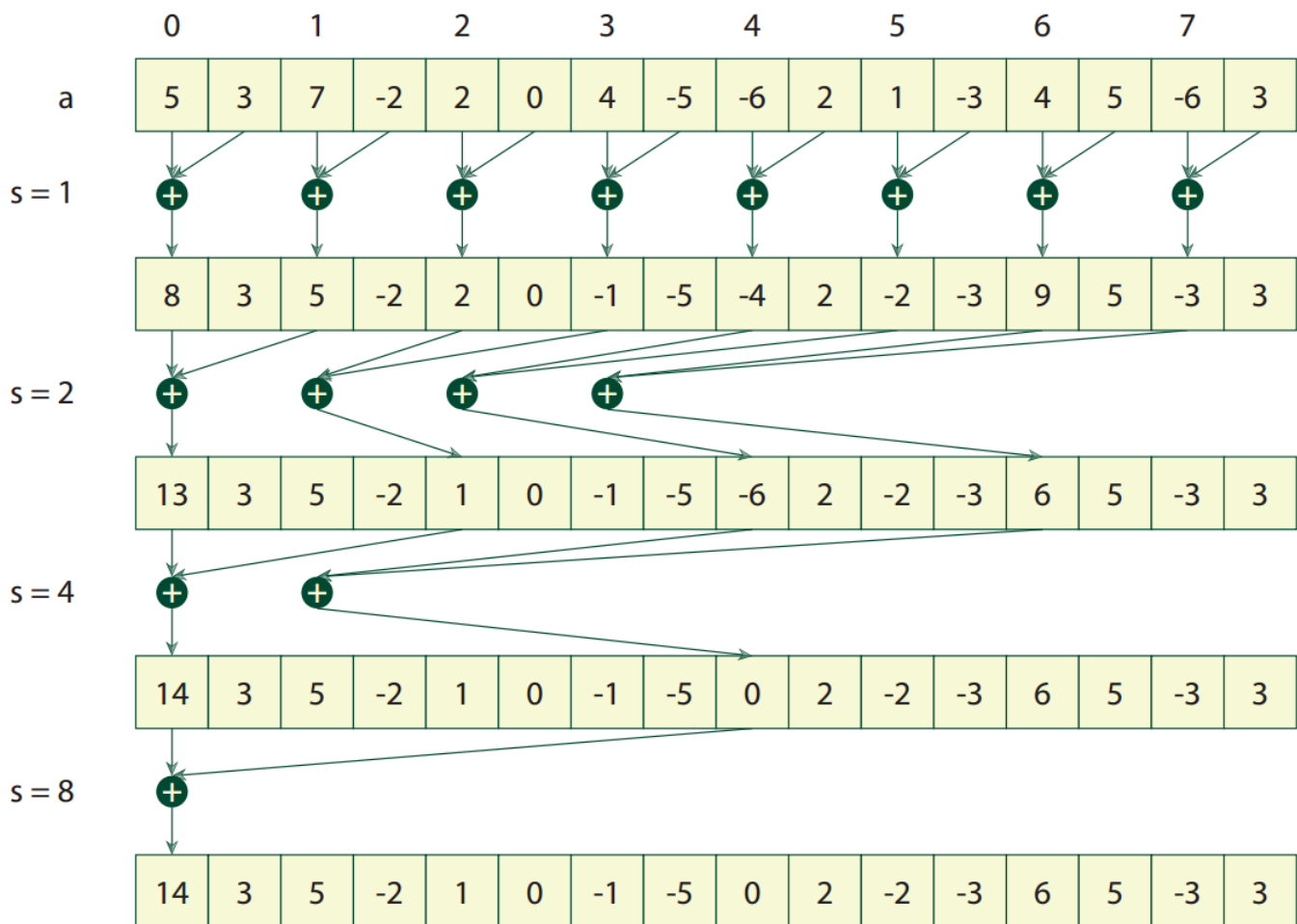
# Наивная реализация

- ▶ В случае если *ор* – простая операция, производительность ограничена быстродействием памяти
  - ▶ Оптимизация – создать буфер в разделяемой памяти для части массива обрабатываемой блоком
  - ▶ Недостаток – ветвление нитей
  - ▶ Выход – перераспределение операций
- 

# Ветвление нитей



# Ветвление нитей

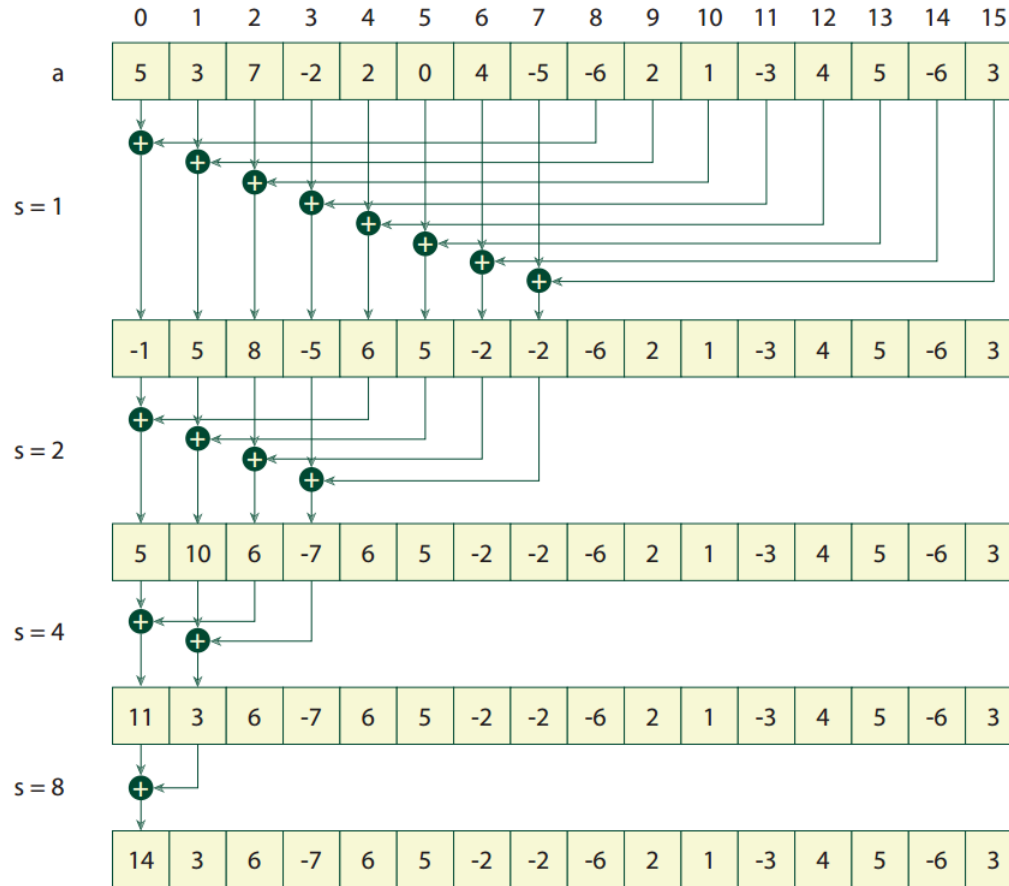




# Конфликт банков

- ▶ При  $s = 1$  конфликт банков 2-ого порядка
  - threadIdx.0 и threadIdx.16 обращаются к 0-му банку
  - threadIdx.1 и threadIdx.17 обращаются к 3-му банку
  - etc
- ▶ При  $s = 2$  конфликт 3-ого порядка,  $s = 3$  4-ого, etc.
- ▶ Решение – суммирование наиболее удаленных элементов

# Конфликт банков



# Укрупнение блоков

- ▶ На первом шаге работает половина нитей блока
- ▶ Уменьшив число блоков вдвое, а в каждом блоке будем обрабатывать в 2 раза больше элементов

# Shuffle

- ▶ Начиная с архитектуры Kepler нити могут обмениваться данными регистров без разделяемой памяти
- ▶ Специальные shuffle-инструкции
  - `__shfl()`
  - `__shfl_down()`
  - `__shfl_up()`
  - `__shfl_xor()`

# Shuffle

```
int __shfl_down(int var, unsigned int delta, int width=warpSize);
```

- ▶ Нить добавляет к своему адресу значение delta и считывает по этому адресу значение var
- ▶ width={2, 4, 8, 16, 32}
- ▶ Если адрес вышел за пределы width возвращается значение var самой нити

# Shuffle


- Поддерживаются только 4-байтовые типы данных

```
__device__ inline double __shfl_down(double var, unsigned int  
srcLane, int width=32)  
{  
    int2 a = *reinterpret_cast<int2*>(&var);  
    a.x = __shfl_down(a.x, srcLane, width);  
    a.y = __shfl_down(a.y, srcLane, width);  
    return *reinterpret_cast<double*>(&a);  
}
```

# Shuffle

- ▶ Вариант редукции с использованием shuffle:
  - В каждом блоке варпы вычисляют промежуточные суммы через `__shfl_down()`
  - Суммы помещаются в массив в разделяемой памяти, массив суммируется первым варпом
  - Частичные суммы каждого блока суммируются повторным вызовом ядра
- ▶ Модификация – каждый варп добавляет свою сумму к итоговой через `atomicAdd()`
- ▶ Модификация варианта 2 для блоков

# Векторизация

- ▶ Для улучшения паттерна доступа к памяти можно использовать векторизацию
  - ▶ Векторизация является практически «серебряной пулей»
  - ▶ Обработка данных как векторов длиной 64/128 бит
  - ▶ Используются встроенные векторные типы данных `int2`, `int4`, `float2`, `float4`
- 



# Векторизация

- ▶ Указатели на исходные данные преобразуются к векторным типам
- ▶ Разыменованное обращение к таким указателям вызывает векторные инструкции для выровненных данных
- ▶ Использование векторизации увеличивает нагрузку на регистры

# Векторизация

```
__global__ void device_copy_scalar_kernel(int* d_in, int* d_out, int N){  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int i = idx; i < N; i += blockDim.x * gridDim.x)  
        d_out[i] = d_in[i];  
}
```



```
__global__ void device_copy_vector4_kernel(int* d_in, int* d_out, int N) {  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    for(int i = idx; i < N/4; i += blockDim.x * gridDim.x)  
        reinterpret_cast<int4*>(d_out)[i] =  
            reinterpret_cast<int4*>(d_in)[i];  
}
```

# Векторизация

- ▶ Вариант редукции с векторизацией и использованием cooperative groups(CG)
  - Каждая нить пробегает по массиву складывая по 4 элемента за раз
  - Каждый блок получает частичную сумму, потом атомарно складывает результат с итоговой суммой

# Векторизация

- ▶ Оптимизация: шаблонизированное создание CG с размером в качестве параметра
- ▶ В этом случае возможна развертка цикла в функции частичных сумм для группы – размер группы известен на этапе компиляции



Казанский  
федеральный  
университет

ВЫСШАЯ ШКОЛА  
информационных технологий  
и информационных систем

# Вопросы

ekhramch@kpfu.ru