Параллельные вычисления на основе технологии OpenCL

Лекция 1

Силаков Роман Дмитриевич

Краткое содержание лекции

- Что и зачем изучаем?
- «Hello World!» для GPU
- Архитектура и модель программирования
  - Пример: умножение матриц
- Иерархия памяти
  - Пример: ускоряем умножение матриц
- Простые алгоритмы и паттерны
  - Паттерны: Map, Gather, Scatter
  - Редукция
  - Свертка
  - Гистограмма

Вычисления на GPU

GPGPU - G*eneral-purpose graphics processing units.*

Идея: Использовать GPU для общих вычислений, а не только для компьютерной графики.

GPU vs. CPU

	GeForce RTX 3090	**AMD** Ryzen **Threadripper 3990X**
Число ядер	10496	64
Производительность, GFLOPS	≈35000	≈3700
Архитектура (грубо)	SIMD*	SISD*

*SIMD - single instruction, multiple data
SISD - single instruction, single data

Области применения

- Машинное обучение
- Математические пакеты
- Компьютерная графика
- Физические движки
- Обработка видео
- Математическое моделирование

Еще примеры:
http://www.nvidia.ru/object/gpu-applications-ru.html

OpenCL

OpenCL(**Open C**omputing **L**anguage) – стандарт для написания программ, связанных с параллельными вычислениями

- Для разработки используются язык программирования, который базируется на стандарте C99 (C++14 для OpenCL >= 2.1)
- Поддерживается большинством современных CPU/GPU

CUDA

CUDA (*Compute Unified Device Architecture*) - архитектура параллельных вычислений от компании NVIDIA.

- Для разработки используются языки программирования CUDA C/C++, Fortran и др.
- Поддерживается только видеокартами NVidia

Сложение двух массивов, CPU

```
void main()
{
  static const int n = 1024;
  int a[n], b[n], c[n];
  fill_arrays(a, b); //заполнение массивов

  for (int i = 0; i < n; ++i)
    c[i] = a[i] + b[i];
}
```

Сложение двух массивов, GPU

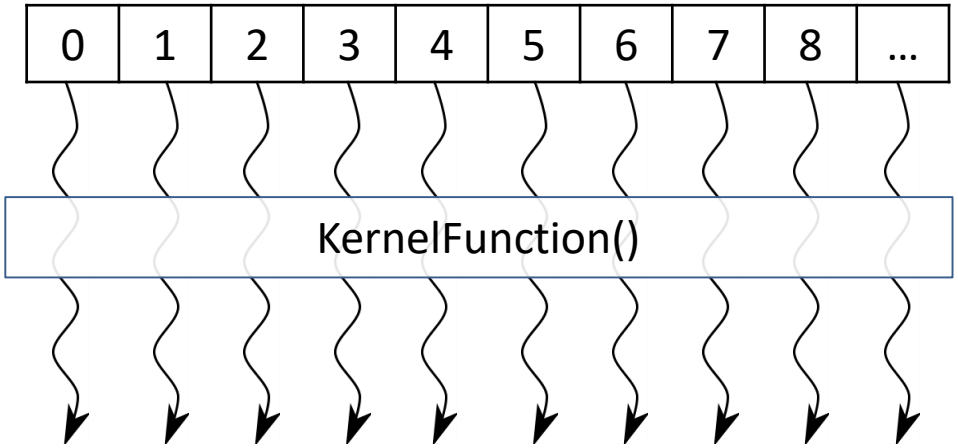
См. пример vector_add

Архитектура и модель программирования

Work items (Потоки)

Для выполнения параллельных вычислений запускается ядро вычислений (kernel) состоящее из множества потоков (threads)

- Все потоки выполняют один и тот же код
- Каждый поток имеет уникальный идентификатор



Проблема: Как организовать взаимодействие потоков?

- Взаимодействие потоков внутри одного массива потоков не масштабируемо

Решение: Разобьём массив потоков на блоки (work group).

- Взаимодействие между небольшими группами потоков масштабируемо

Взаимодействие потоков в блоке

- Локальная память (local memory) – быстрая память доступная всем потокам одного блока
- Команда барьера barrier()
  - Любой поток блока дойдя до барьера будет ждать пока остальные потоки не дойдут до этого же барьера

```
__kernel void foo()
{
int id = get_local_id();
load_data_to_local_memory(id);
barrier(CLK_LOCAL_MEM_FENCE);
use_data_from_local_memory();
barrier(CLK_LOCAL_MEM_FENCE);

}
```

Листинг 1. Пример использования команды barrier

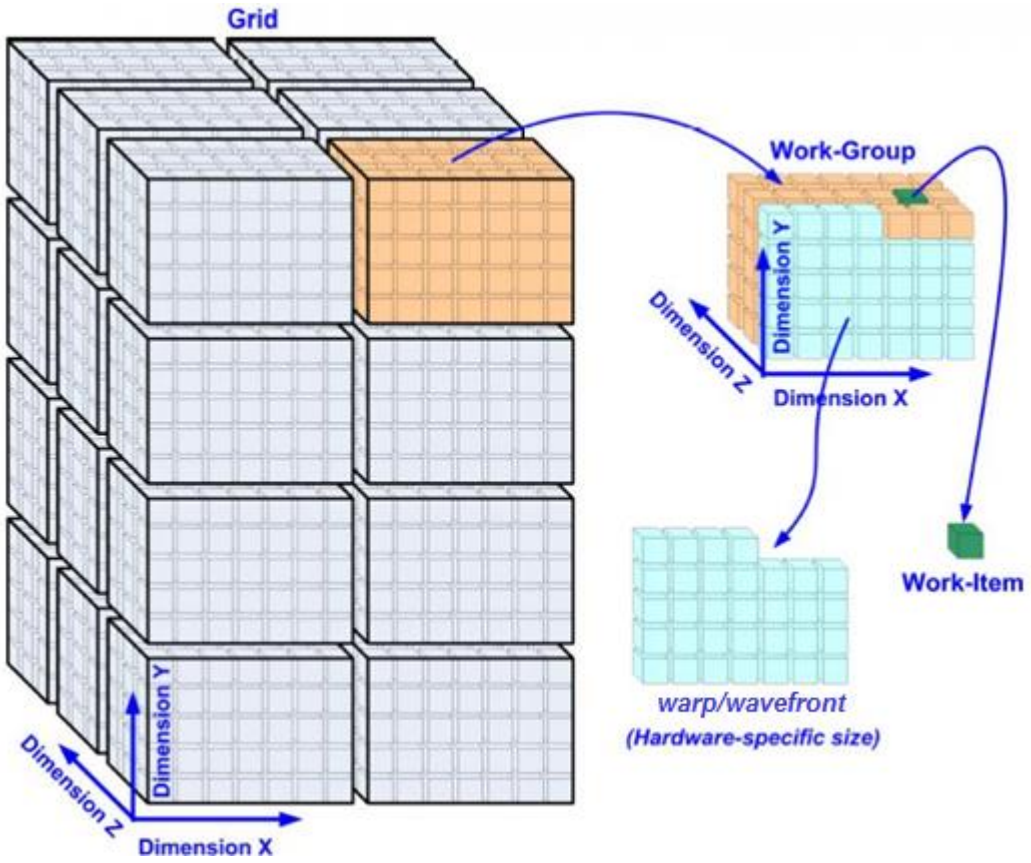
```
__kernel void foo()
{
if (condition)
  barrier1(CLK_LOCAL_MEM_FENCE);
else
  barrier2(CLK_LOCAL_MEM_FENCE);
}
```

Листинг 2. Классическая ошибка в использовании barrier

Вычислительная сетка (ND-Range) (1)

Вычислительная сетка – многомерный массив потоков разбитый на блоки (work-groups)

- Все блоки в сетке имеют одинаковые размеры
- Взаимодействие потоков происходит внутри блоков



Вычислительная сетка (ND-Range) (2)

- Размерность сетки ограничена (3 для OpenCL 2.0)
- Размер сетки ограничен
- Размер блока ограничен
- Для каждого потока доступны
  - Индекс потока в сетке: **get_global_id(uint dimension_idx)**
  - Индекс потока в блоке: **get_local_id(uint dimension_idx)**
  - Размер блока: **get_local_size**(**uint dimension_idx**)
  - Индекс блока: **get_group_id(uint dimension_idx)**
  - Количество блоков: **get_num_groups(uint dimension_idx)**
  - Количество потоков: **get_global_size(uint dimension_idx)**

Количество потоков в сетке и размер блока задают конфигурацию ядра.

Пример: умножение матриц

Ищем $C = A \cdot B$. Будем умножать матрицы размером $\mathrm{N} \times \mathrm{N}$.

Запустим ядро вычислений для сетки размером $\mathrm{N} \times \mathrm{N}$.

Поток с индексом (i, j) будет считать $C[i][j]$

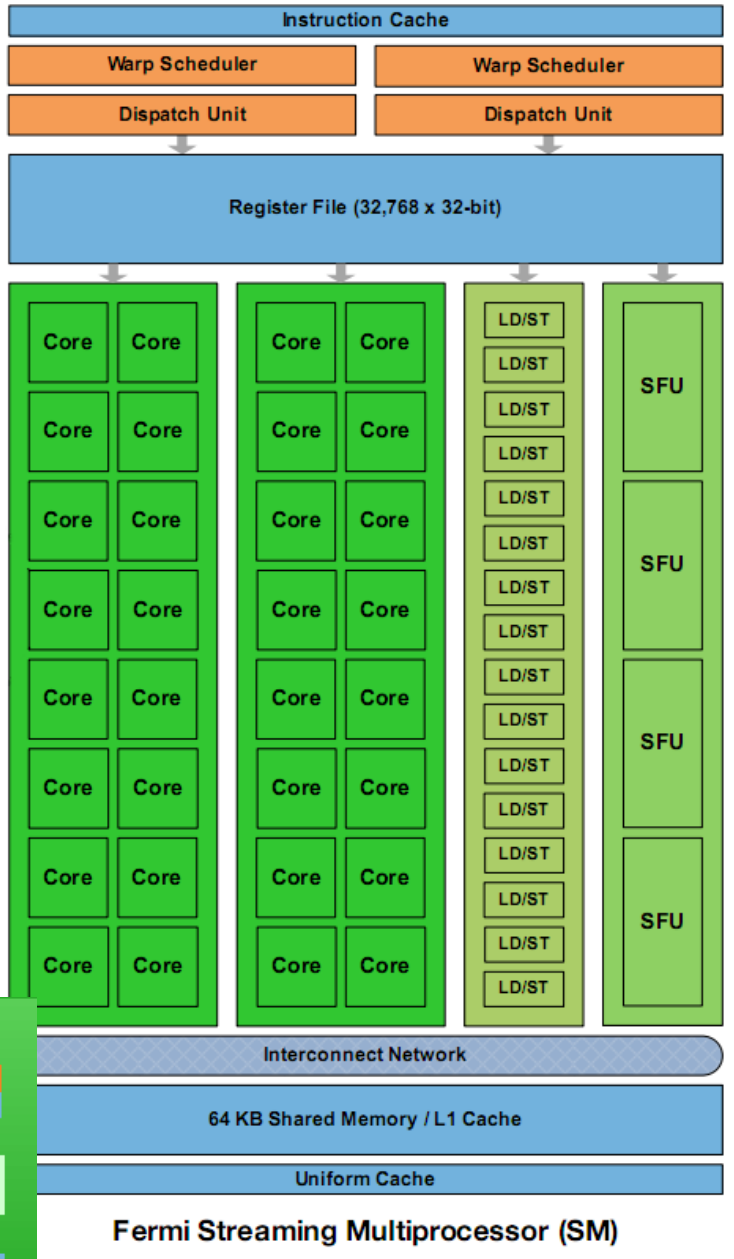
Исходный код - см. пример matrix_mult

Архитектура (NVidia Fermi)

- 16 потоковых мультипроцессоров (Streaming Multiprocessor, SM)

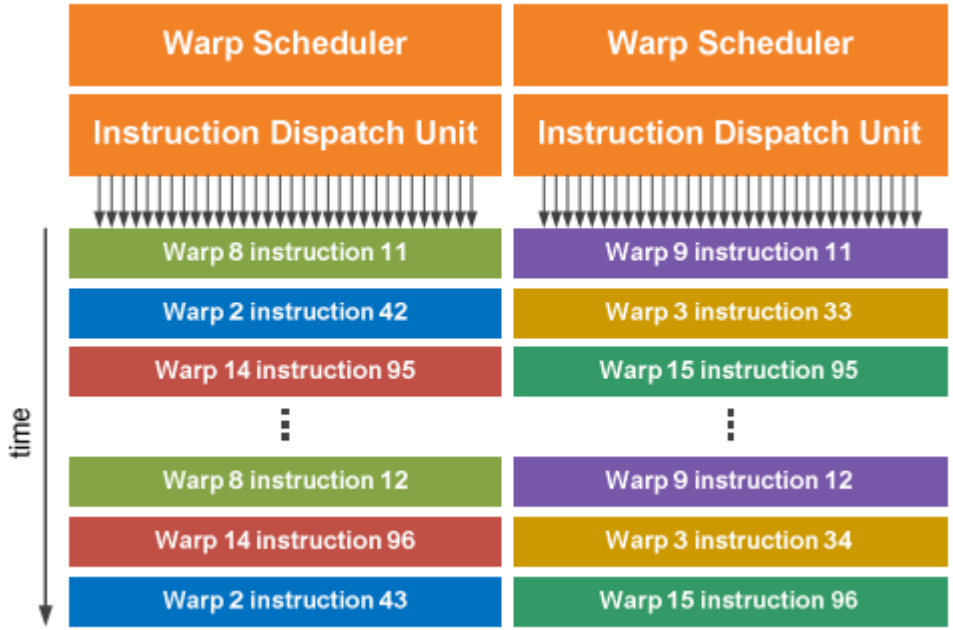
Особенности SM:

- Содержит 32 CUDA ядер, 64 KB разделяемой памяти / L1 кэша, 16 Load/Store units (LD/ST), 4 Special Function Units (SFUs)
- Выполняет вычислительные блоки, блок целиком выполняется на одном SM, одновременно не более 8 блоков и не более 1536 потоков
- При выполнение блок разбивается на warp-ы (warp – 32 подряд идущих потока)



Планировщик warp-ов (Fermi)

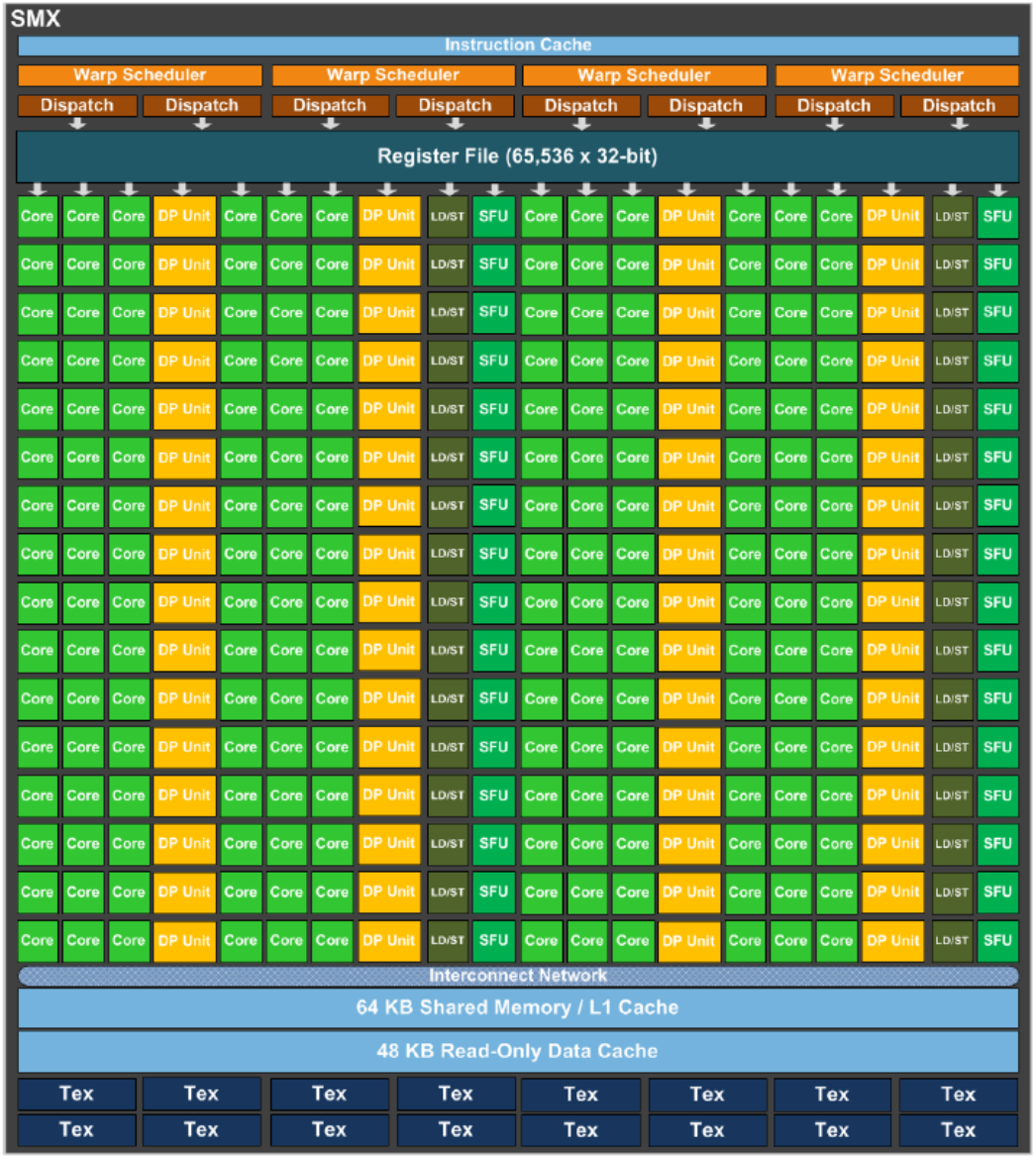
- Каждый SM содержит два планировщика warp-ов
- Инструкция warp-а назначается группе из 16 ядер, 16 LD/ST юнитов или 4 SFUs (в зависимости от типа инструкции)
- Разделение на warp-ы позволяет скрывать задержки
  - Пока один warp работает с памятью, будут выполнятся инструкции других warp-ов



Планировщик warp-ов

Архитектура Nvidia Kepler

Появился DP Unit - ядро для работы с 64 битными числами с плавающей запятой.



Выбор размера блока

Перемножаем две матрицы размером 1024x1024.

Вопрос: Каким выбрать размер блока?

Решение: Необходимо максимизировать число потоков на SM

Характеристики GPU:

- Максимальное число блоков на SM: 8
- Максимальное число потоков в одном блоке: 1024
- Максимальное число поток в одном SM: 1536

Варианты ответа:

1. 8x8
2. 16x16
3. 32x32
4. 64x64

Архитектура AMD Graphics Core Next

Вместо понятия warp используется понятие wavefront. Wavefront состоит из 64 потоков.

Подробнее, например, по ссылке:

http://developer.amd.com/wordpress/media/2013/06/2620_final.pdf

Архитектура AMD RDNA

- https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

Перешли на 32 потока. wave32

Иерархия памяти

Типы памяти

- Приватная и регистровая

```
int i;
float3 arr[4];
```

- Локальная

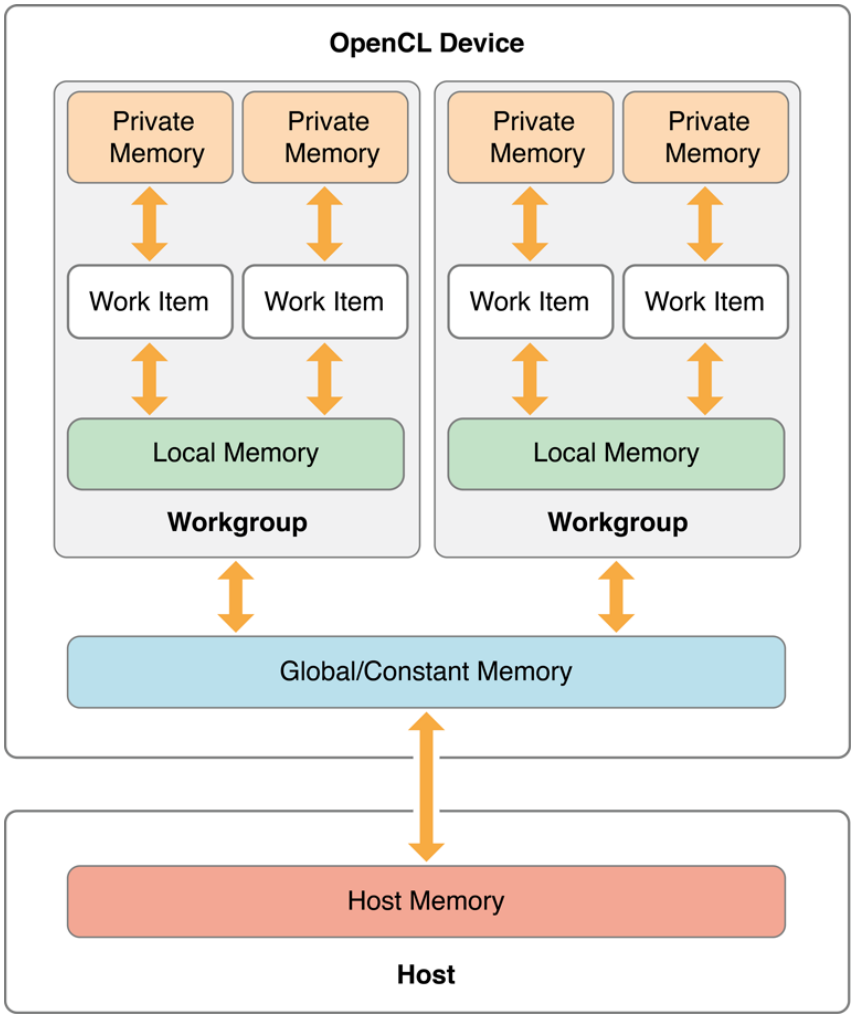
```
__local float s[128];
```

- Глобальная

```
__global int *p;
```

- Константная

```
__constant int i = 5;
```



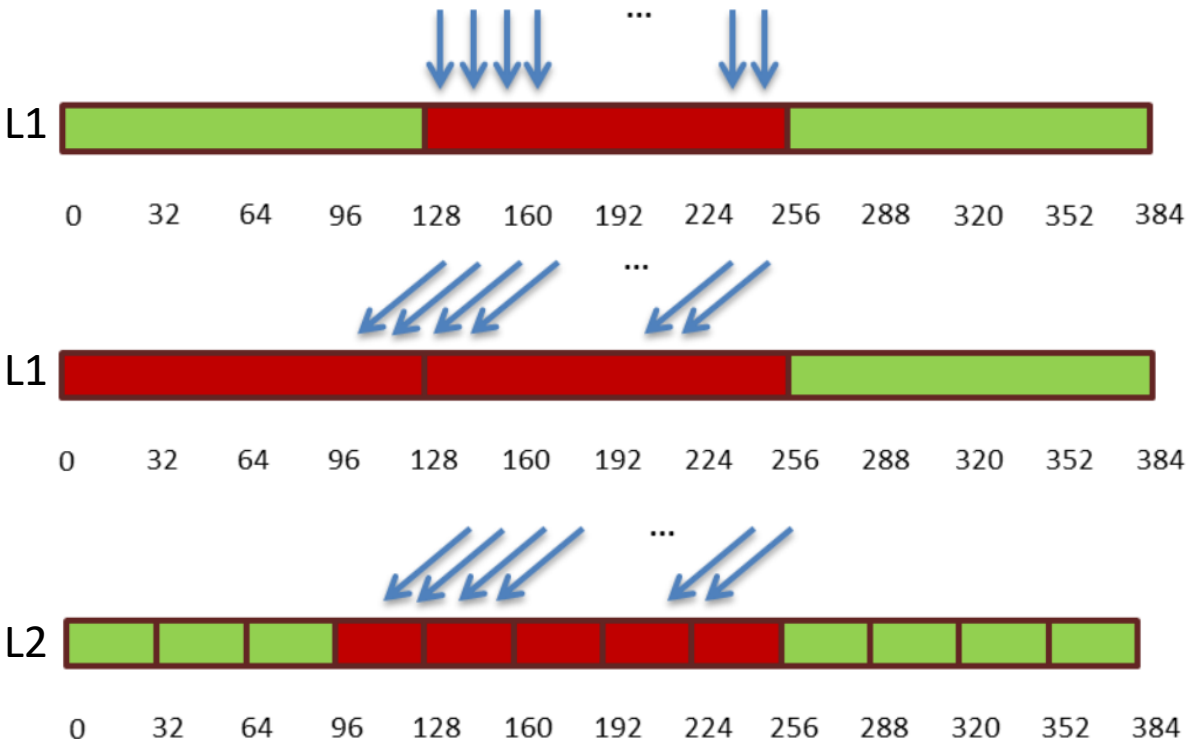
Сравнение типов памяти (на основе NVIDIA GeForce 580 GTX)

Тип	Размер	Задержка	Пропускная способность	Применение
Регистровая	Ограниченный размер (оценка сверху: 32768 * 4 / 1536 ~= 85 байт на поток)	~1 такт	Очень высокая (~ 2-3 регистра за такт = 8-12B / такт)	
Локальная	Ограниченный размер (48KB на мультипроцессор)	~5 тактов	Высокая (~ 4B / такт)	Кэш, контроллируемый программистом
Глобальная	Большой (гигабайты)	~500 тактов	Низкая (~ 4B / 10 тактов)	
Константная	Ограниченный размер (64KB, кэш мультипроцессора – 8KB)	~5 тактов (при попадании в кэш)	Высокая (~ 4B / такт, при попадании в кэш)	Broadcast данных

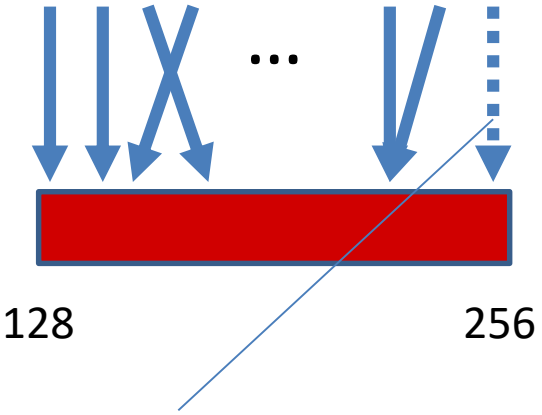
Такт = 1 / частота процессора

Объединение запросов к глобальной памяти (coalescing)

- Размер линии кэша L1 – 128 байт, L2 – 32 байта
- Линии отображены в участки глобальной памяти, выровненные по 128 байт
- Доступ к памяти обслуживается транзакциями по 128 байт (для L1)
- Правило объединения: запросы к памяти потоков в warp’е объединятся в транзакции, число которых равно кол-ву обновлений линий кэша, необходимых для данного запроса



Так тоже можно:



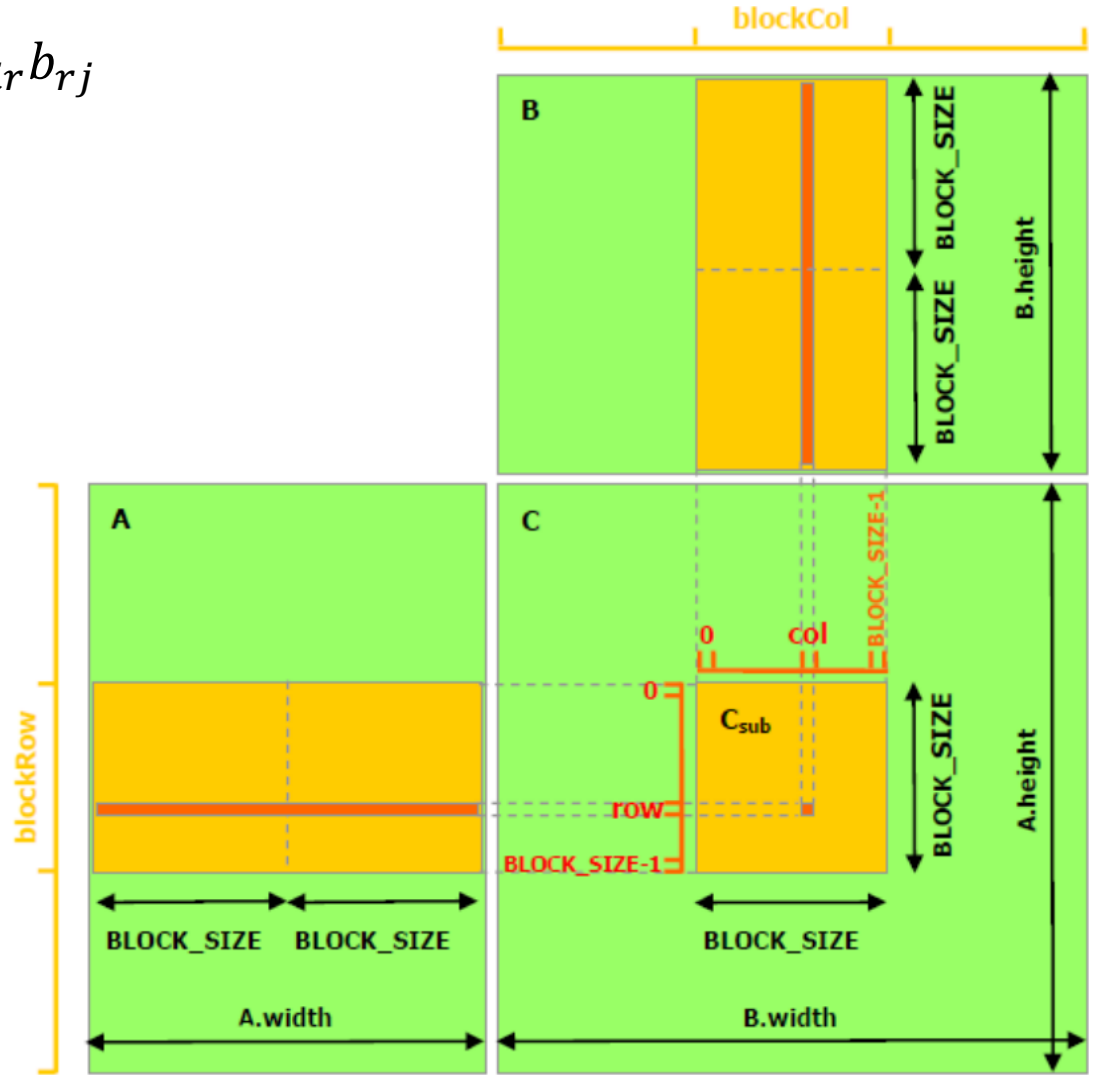
Нет запроса

Пример: умножение матриц с использованием локальной памяти (1)

Ищем $C = A \cdot B; c_{ij} = \sum_{r=1}^{A.width} a_{ir} b_{rj}$

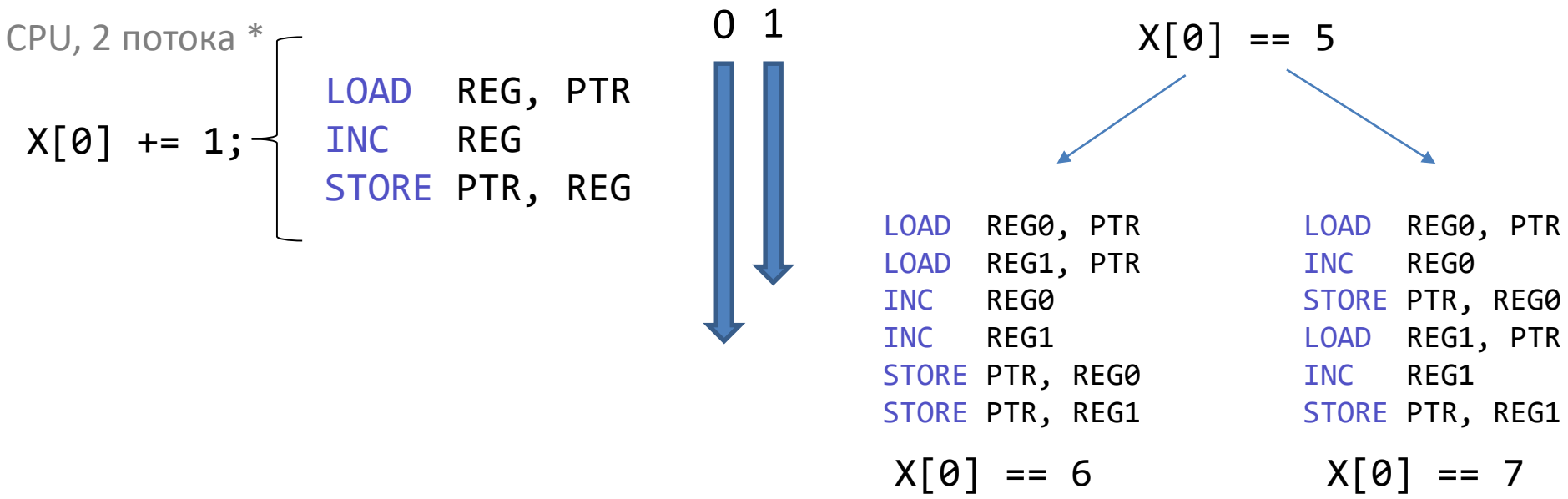
1. Разобьем матрицу C на блоки (тайлы)
2. Заметим, что одни и те же значения из A или B используются много раз при вычислении блока C_{sub}
3. Разобьем необходимые элементы в A и B на блоки и будем вычислять C_{sub} последовательно
4. Разместим элементы из A и B в разделяемой памяти

Исходный код - см. пример matrix_mult



Атомарные операции (1)

- Зачем? Для операций Read-Modify-Write



Примеры:

```
int atomic_add( volatile __global int * address, int val );
```

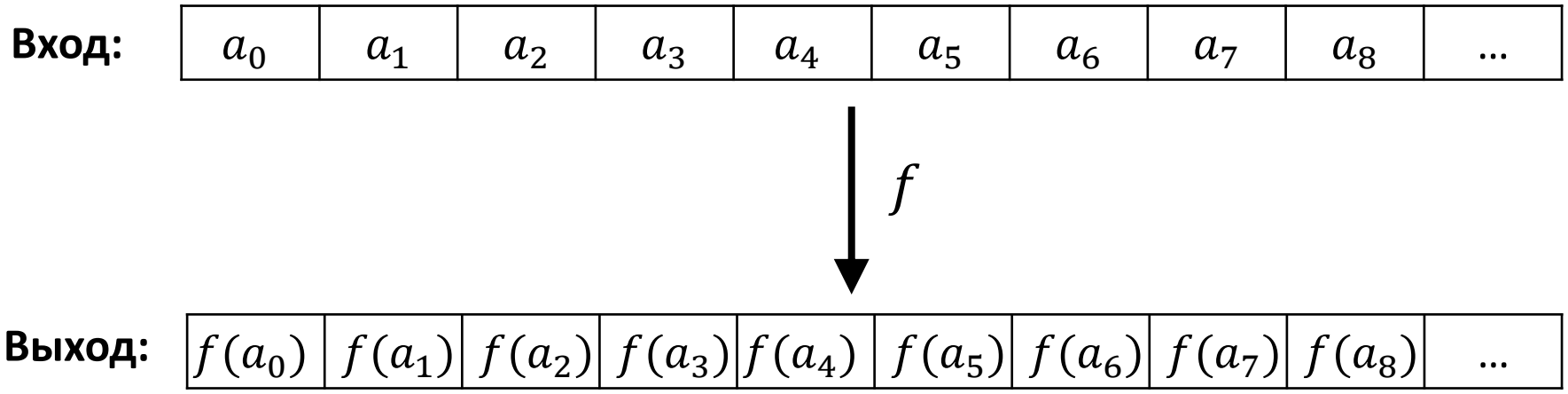
А также Sub, Xchg, Min, Max, Inc, Dec, Cmpxchg, And, Or, Xor

- Атомарные операции медленные (доступ к памяти сериализуется)

Простые алгоритмы и паттерны

Map

Операция **Map** - применение оператора $f()$ к элементам массива. Типичная реализация на GPU – один поток обрабатывает один элемента массива.

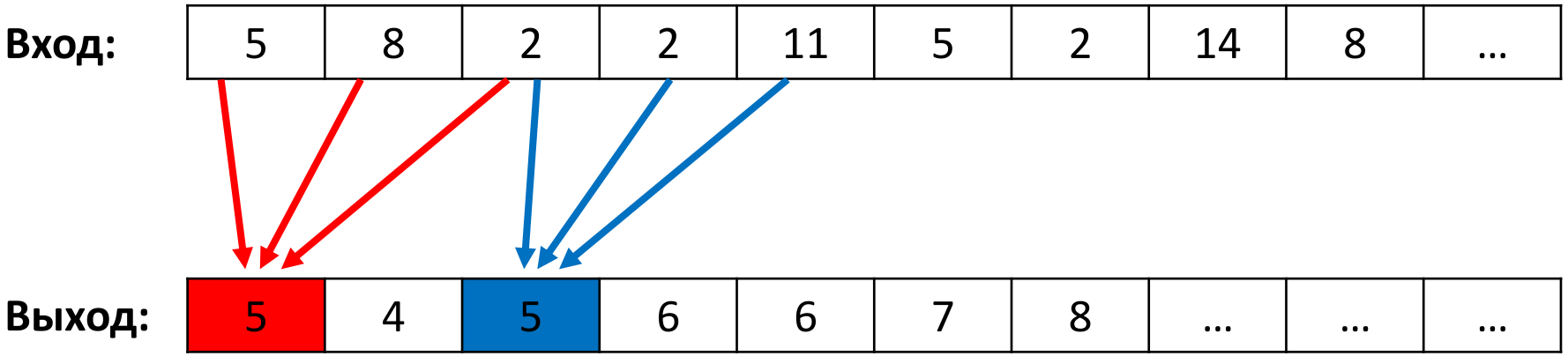


Gather

Операция **Gather** - «сбор» данных из нескольких элементов массива.

Один поток обращается к нескольким элементам массива, обрабатывает полученные данные и записывает результат.

Пример – арифметическое среднее трех подряд идущих элементов:



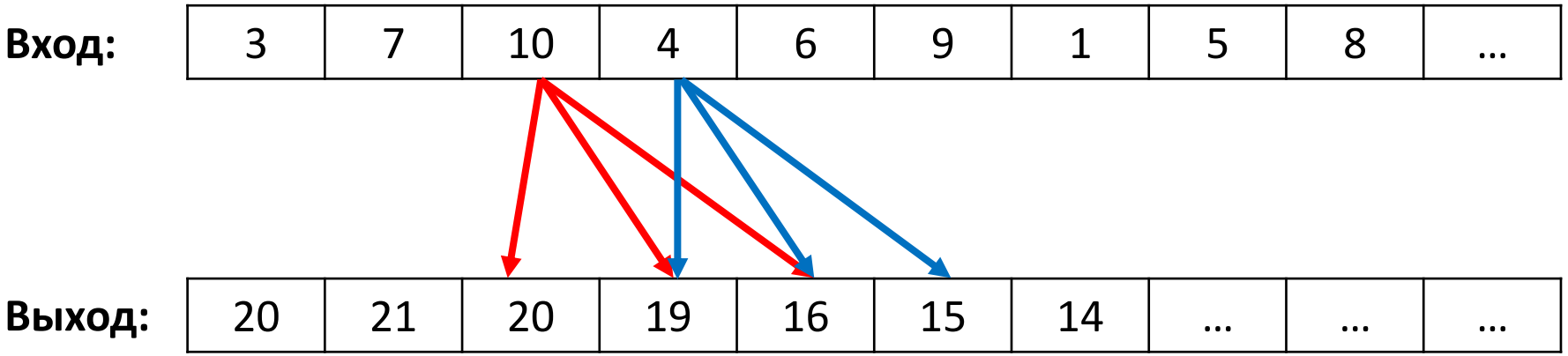
Scatter

Операция **Gather** – «разброс» данных массива.

Один поток считывает данные из элемента массива и записывает результат в несколько различных элементов.

Основная идея – поток сам считает куда записывать данные.

Пример – сумма трех подряд идущих элементов:



Редукция

Задача: Применить оператор редукции к массиву.

Оператор редукции: бинарный и ассоциативный

- Сложение, умножение
- Максимум, Минимум
- Логическое и/или

Пример:

Дано: Массив A=(10, 20, 5, 15). Оператор «+». Результат: 50

```
int reduce( int * A, int A_size )
{
   int res = 0;
   for (int i = 0; i < A_size; ++i)
      res += A[i];
   return res;
}
```

Листинг 1. Последовательная редукция

Параллельная редукция, идея

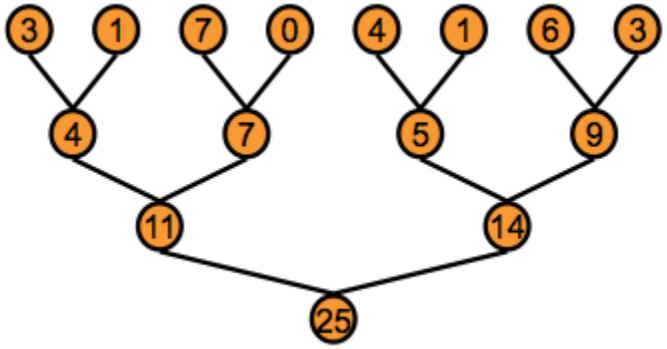


Рис. 1. Идея параллельной редукции

На вход поступает N элементов.

Step complexity: $O(logN)$

Work complexity: $O(N)$

Исходный код - см. пример reduce.

Параллельная редукция, синхронизация

Проблема: Отсутствует механизм синхронизации между блоками.

Решение: Последовательный запуск нескольких ядер вычисления.

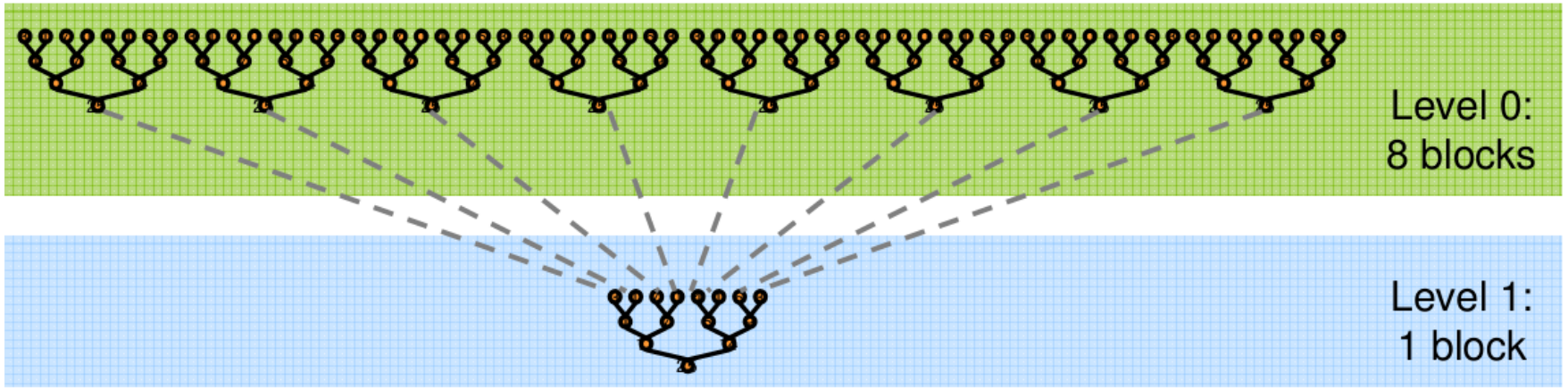


Рис. 1. Редукция, синхронизация между блоками с помощью запуска нескольких ядер

Свертка

Рассмотрим одномерный случай:

Вход: Массив A из N элементов; Маска M из L элементов.

Выход: Массив B из N элементов. $B[i] = \sum_{j=-L/2}^{L/2} A[i+j] \cdot M[j]$.

Примечание: $A[k] = 0$, если $k < 0$ или $k \geq N$.

Пример:

Пусть A=(4, 2, 3, 5, 6, 8); M=(2, -2, 1).

Тогда B=(-6, 7, 3, 2, 6, -4).

B[0] = 0·2 + 4·(-2) + 2·1=-6

B[1] = 4·2 + 2·(-2) + 3·1= 7

…

Исходный код: см. пример convolution

Гистограмма

Вход: Массив A из N элементов; число корзин M; функция $calcbin$ считает номер корзины для элементов массива A: $\forall\ i\ calcbin(A[i]) \rightarrow [0; M-1]$.

Выход: Массив B из M элементов. $B[i]$ – число элементов массива A попадающих в i -ую корзину.

Пример:

Пусть A=(5, 6, 2, 4, 5, 9, 0, 10); M=2; calcbin(x) = x % 2.

Тогда B[0] = 5; B[1]=3.

В 0-ую корзину попадают элементы 6, 2, 4, 0, 10.

В 1-ую корзину попадают элементы 5, 5, 9.

```
void calc_histogram_cpu(int * A, int N, int * B)
{
   for (size_t i = 0; i < N; ++i)
      B[calcbin(A[i])]++;
}
```

Листинг 1. Последовательное построение гистограммы

Гистограмма, наивный подход

```
__kernel void gpu_histogram_naive(__global int * input,
                                  __global int * output)
{
    size_t idx = get_global_id(0);
    int bin = calc_bin(input[idx]);
    output[bin]++;
}
```

Важно: Данная программа работает неправильно!

Гистограмма, простой алгоритм

```
__kernel void gpu_histogram_atomic(__global int * input,
                                   __global int * output)
{
    size_t idx = get_global_id(0);
    int bin = calc_bin(input[idx]);
    atomic_inc(&output[bin]);
}
```

Проблема: atomic_inc это медленно. Количество параллельно выполняемых потоков для одного compute unit-а не превысит количество корзин.

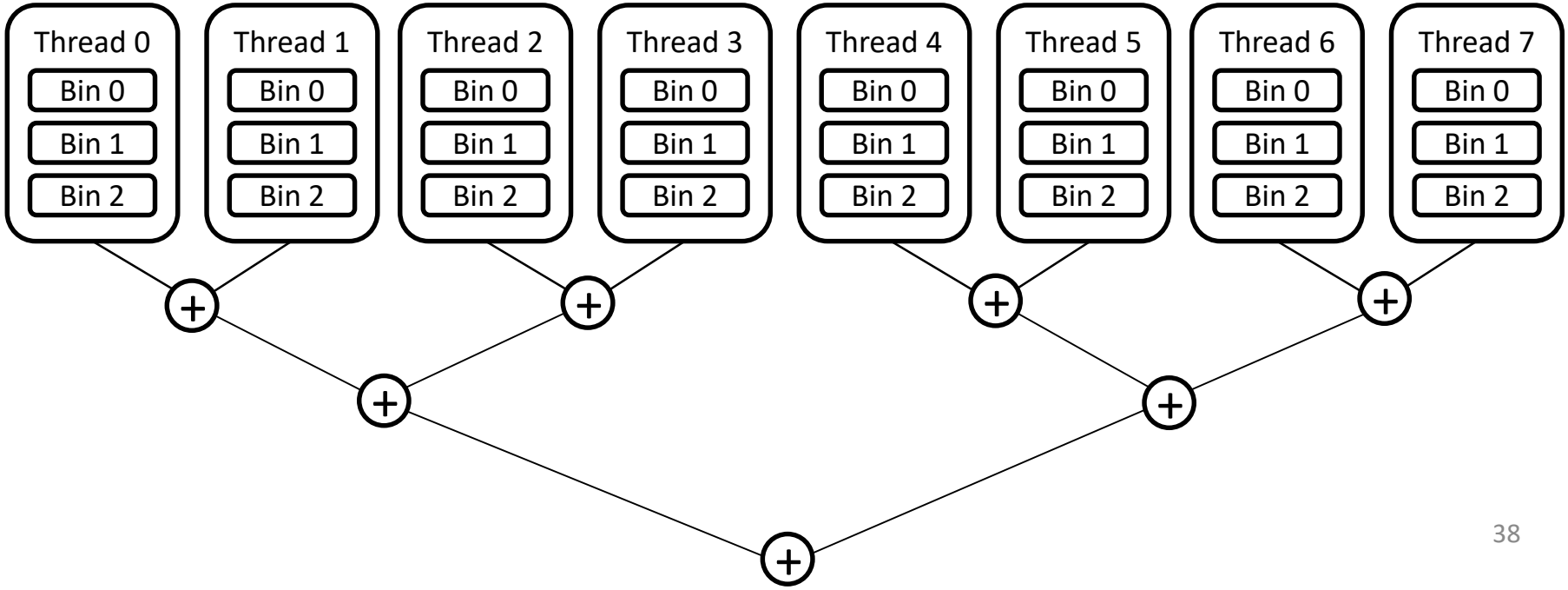
Гистограмма, алгоритм без атомарных операций

Каждый поток посчитает свои локальные корзины. Итоговый результат получим с помощью редукции.

Пример: Элементов N=128; Корзин M=3. Запустим 8 потоков.

Каждый поток возьмет по 128/8=16 элементов и посчитает для них локальную гистограмму. При подсчете локальных корзин можно не использовать атомарные операции.

Затем запустим редукцию для посчитанных локальных корзин:



Литература

Книги

- Heterogeneous Computing with OpenCL, Second Edition. Benedict Gaster
- OpenCL in Action: How to Accelerate Graphics and Computations. Matthew Scarpino

Интернет-курсы (на основе технологии CUDA):

- **www.coursera.org** Heterogeneous Parallel Programming
- **www.udacity.com** Introduction to Parallel Programming

Спецификация

- https://www.khronos.org/opencl/