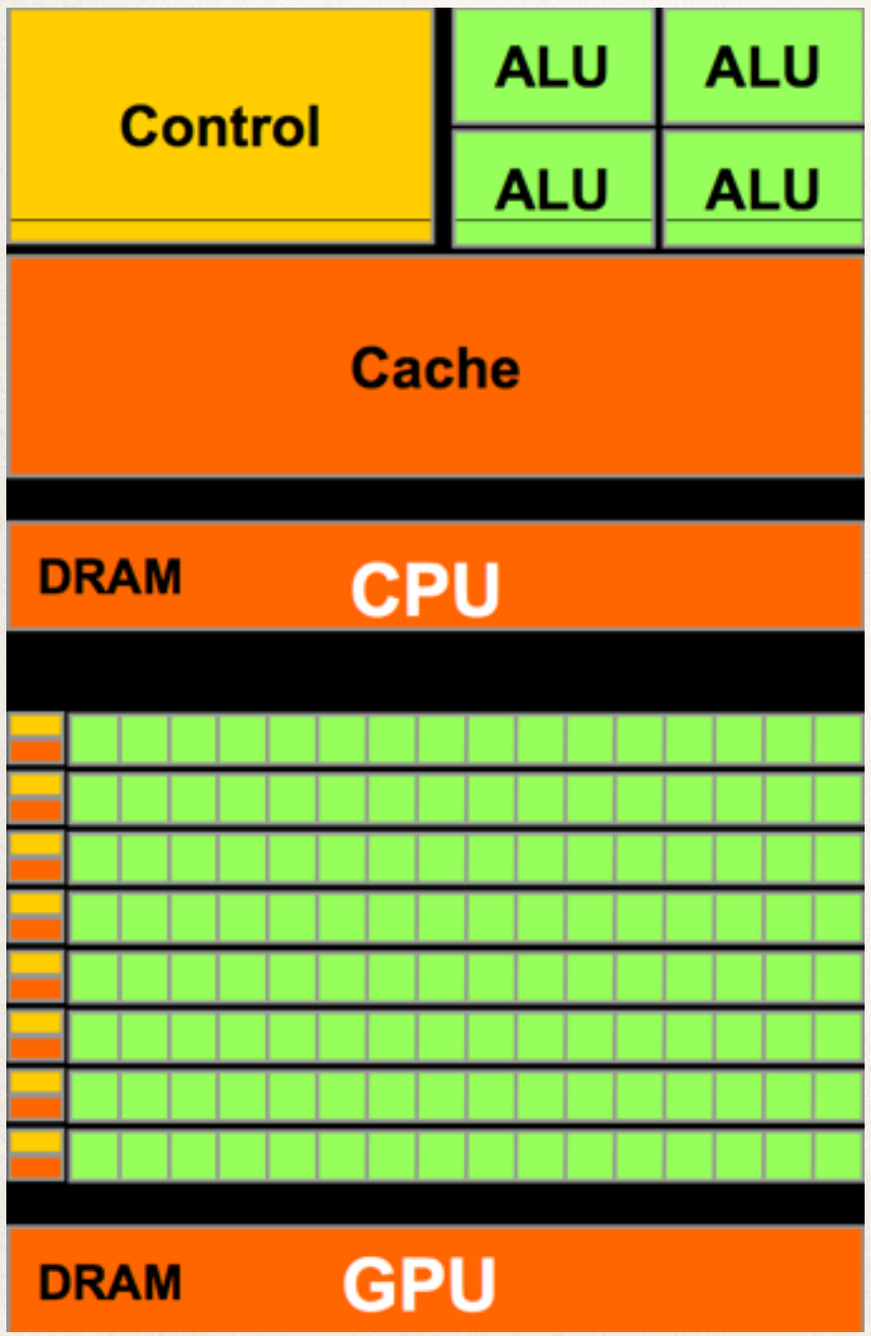
Параллельные вычисления на CUDA
Виды памяти GPGPU

Анна Субботина

7 Декабря 2015

Архитектура



- Вместо кэша и сложных АЛУ отдадим площадь кристала под упрощённые АЛУ, имеющие общую память на кристалле Cache
- +: Латентность доступа к памяти устанавливает однократную систематическую задержку между потоком исходных данных и результатов
- -: Программист обязан тщательно рассчитывать размещение алгоритма на исполняющих элементах / продумывать стратегии доступа к памяти

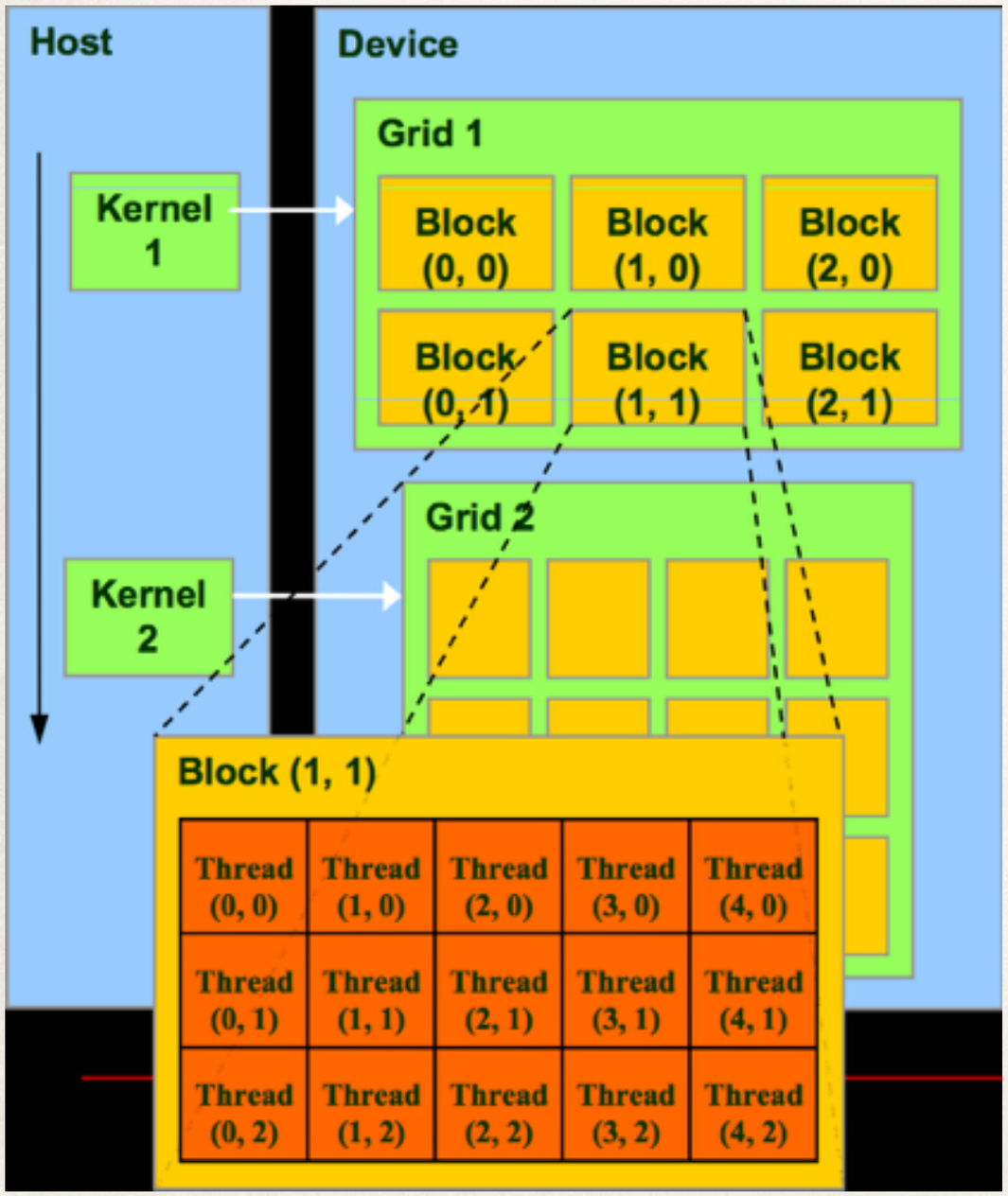
CUDA = Compute Unified Design Architecture

- Многократное ускорение в каждой конкретной задаче достигается в результате усилий программиста
- Существовали ранние попытки использования графических карт для научных расчётов
- Проблема – алгоритмы должны были быть реализованы на специальном “шейдерном” языке (shade language)
- Shade Language разработан и подходит для графики
- Для облегчения работы с алгоритмами общего назначения (GPGPU=General Purpose computing on Graphical Processing Units) компания NVIDIA выдвинула инициативу создания аппаратно/программной архитектуры общего назначения

CUDA

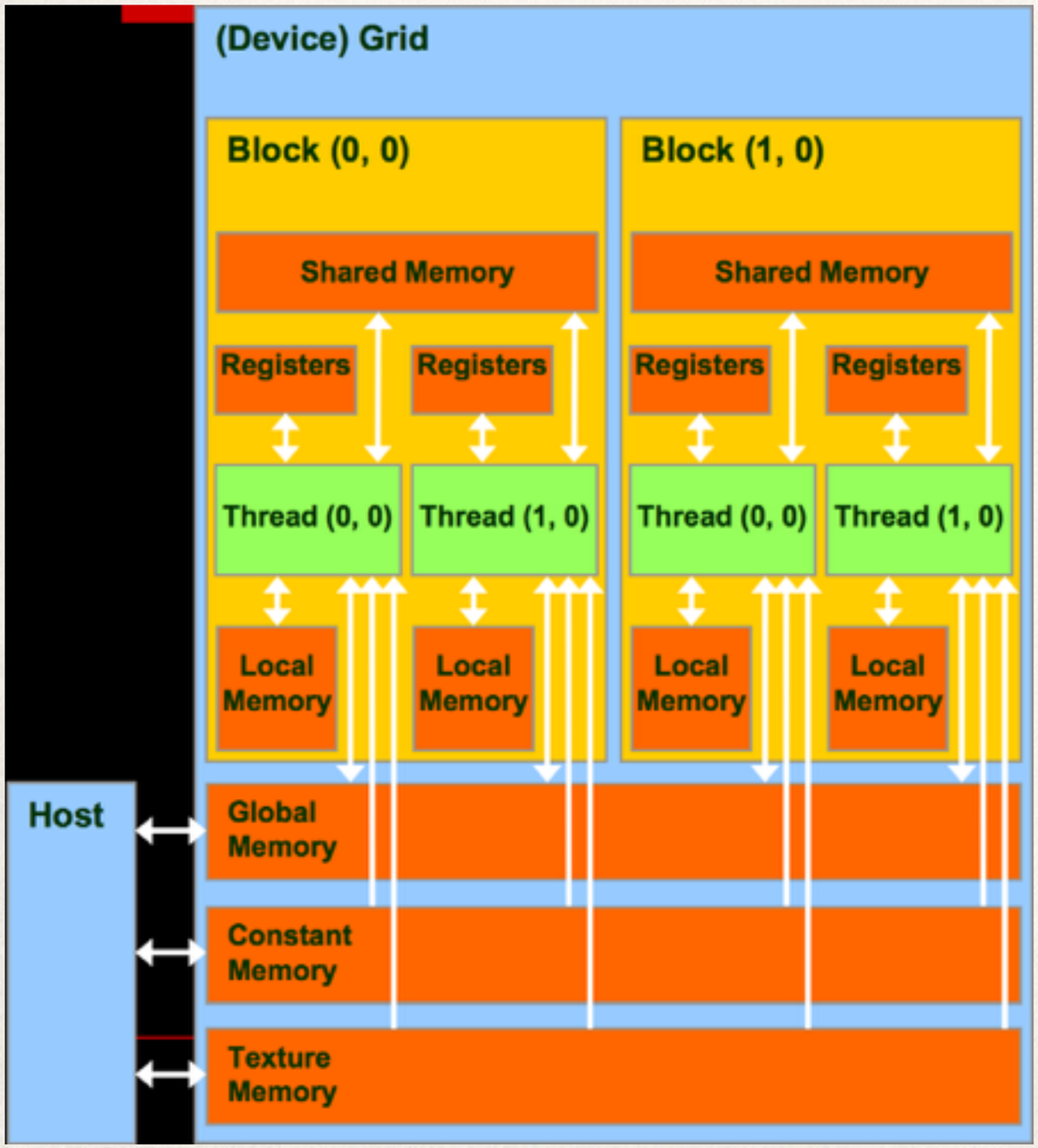
- GPU – сопроцессор для CPU (хоста)
- У GPU есть собственная память (device memory)
- GPU способен одновременно обрабатывать множество процессов (threads) данных одним и тем же алгоритмом
- Для осуществления расчётов при помощи GPU хост должен осуществить запуск вычислительного ядра (kernel), который определяет конфигурацию GPU в вычислениях и способ получения результатов (алгоритм)
- Процессы GPU (в отличие от CPU) очень просты и многочисленны (~ 1000 для полной загрузки GPU)

Вычислительная конфигурация GPU



- Процессы объединяются в блоки (blocks), внутри которых они имеют общую память (shared memory) и синхронное исполнение
- Блоки объединяются в сетки (grids)
  - Нет возможности предсказать очерёдность запуска блоков в сетки
  - Между блоками нет и не может быть общей памяти

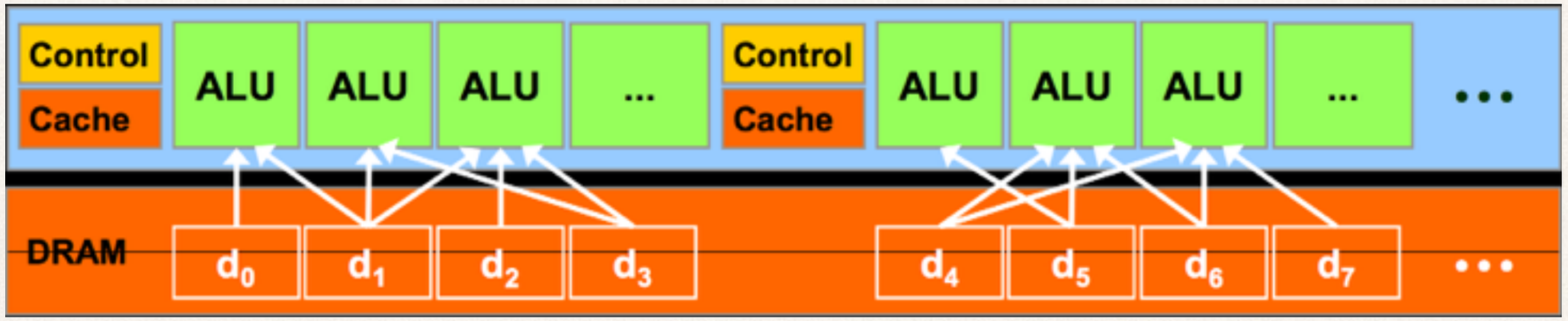
Модель памяти GPU



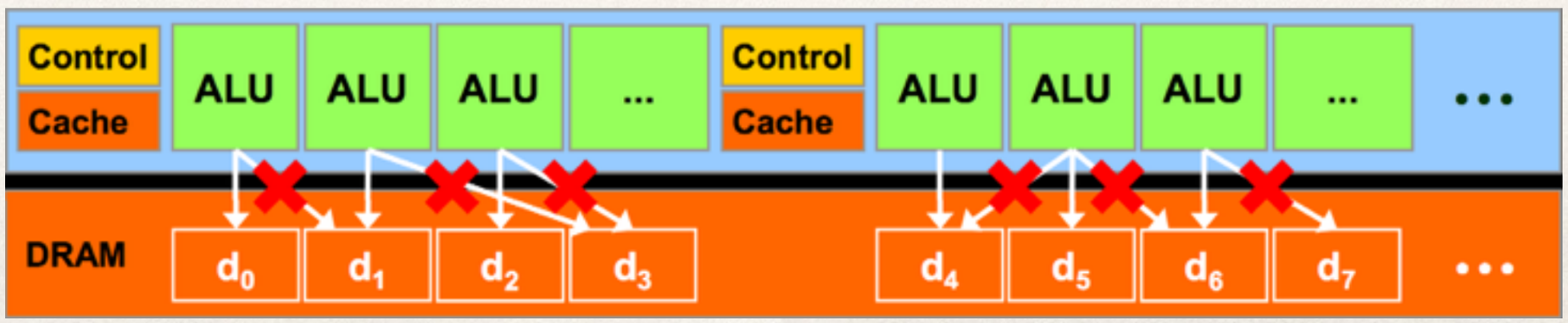
- GPU может читать: Constant Memory, Texture Memory
- GPU может читать/писать: Global Memory
- Каждый из процессов читать/писать: Local Memory, Registers
- Каждый из процессов внутри блока читать/писать: Shared memory, Gather/Scatter MA pattern
- Хост имеет возможность читать/писать: Global Memory, Constant Memory, Texture Memory

Gather и Scatter Scatter обмен данными данными в GPU

- Традиционно в GPU доступ к памяти осуществлялся для обработки пикселов. Gather модель

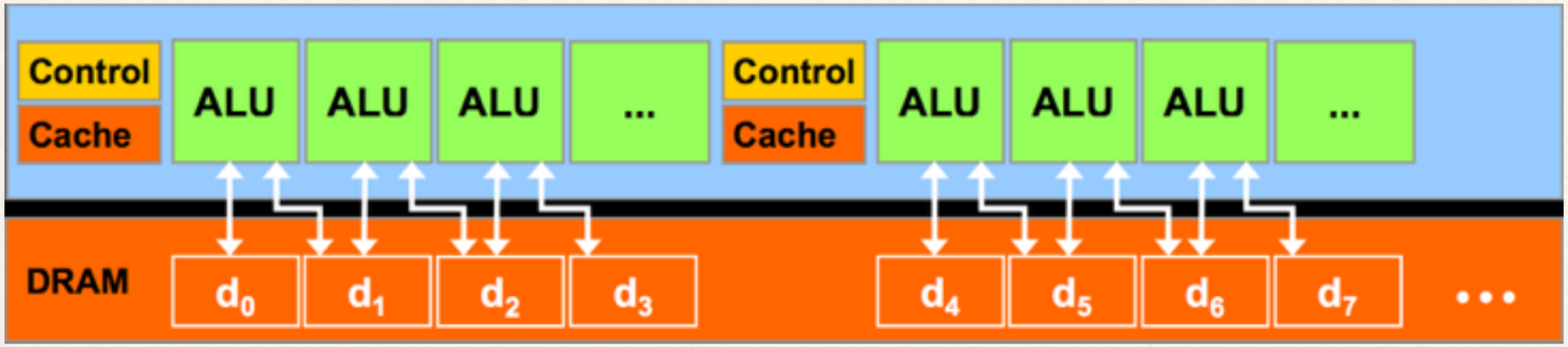


- Но scatter был невозможен / CUDA позволяет делать scatter

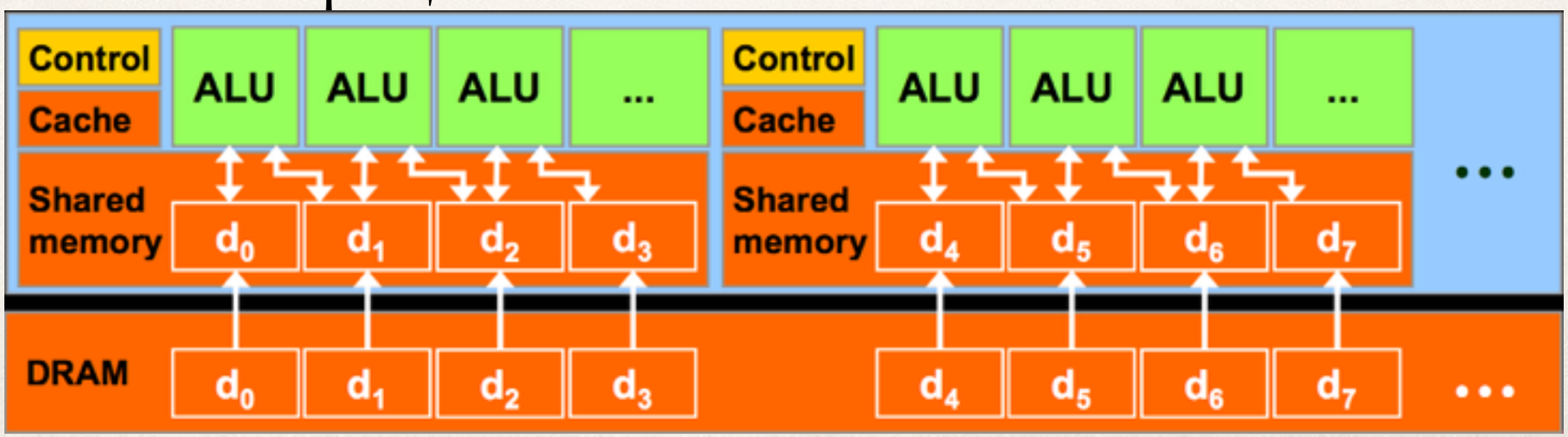


Использование общей памяти

- Прямое чтение данных во многие АЛУ - неоптимально



- Используется механизм Shared Memory: Каждый процесс знает какую часть даных загрузить и где искать данные загруженные остальными процессами



Процедура разработки программы

- Global и Local память расположена на устройстве – DRAM – обращение к ней очень медленное
- Общий подход к программированию
  - Разбить задачу на элементарные блоки данных, над которыми выполняется стандартный алгоритм обработки (единый для всех блоков)
  - Разбить за-/вы- гружаемые данные на элементарные непересекающиеся блоки, которые каждый из процессов прочтёт/запишет
  - Определить конфигурацию грида/блока, позволяющее
    - Оптимальное размещение промежуточных данных в регистрах и общей памяти
    - Оптимальную вычислительную загрузку потоковых процессоров
  - Определить график когерентного обращения к памяти процессами при загрузке данных

Размещение различных данных в различной памяти

- Constant и Texture память тоже расположена на устройстве (DRAM). Но эти типы кэшированы = быстрый доступ
- Распределение элементов данных по типам памяти
  - R/O no structure -> constant memory
  - R/O array structured -> texture memory
  - R/W shared within Block -> shared memory
  - R/W registers при переполнении автоматически отправляются в local memory
  - R/W inputs/results -> global memory

С модификация – объявление переменных

Модификатор	Память	Область	Срок жизни
__device__ __local__ int LocalVar;	local	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __device__ можно опустить если есть модификаторы __local__, __shared__ или __constant__
- Автоматические переменные, размещаются в регистрах (! Нужно следить за тем, чтобы массив регистров не переполнялся)
- Автоматические массивы – в local памяти
- Указатели используются только с областями в global памяти

Массивы в shared памяти

```
__global__ void CUDAforwardpropagateCALCv6(float *in,float *wm,float *bv,float
*lo,int insize,int outsize,int r,int wm_pitch){
extern __shared__ float sharray[];
float* laSH1=&sharray[0];
float* laSH2=&sharray[128];
**************************************************************************
CUDA_SAFE_CALL( cudaThreadSynchronize() );
CUDAforwardpropagateCALCv6<<<grid, threads,threads.x*sizeof(float)>>>(in,wm,…);
CUDA_SAFE_CALL( cudaThreadSynchronize() );
```

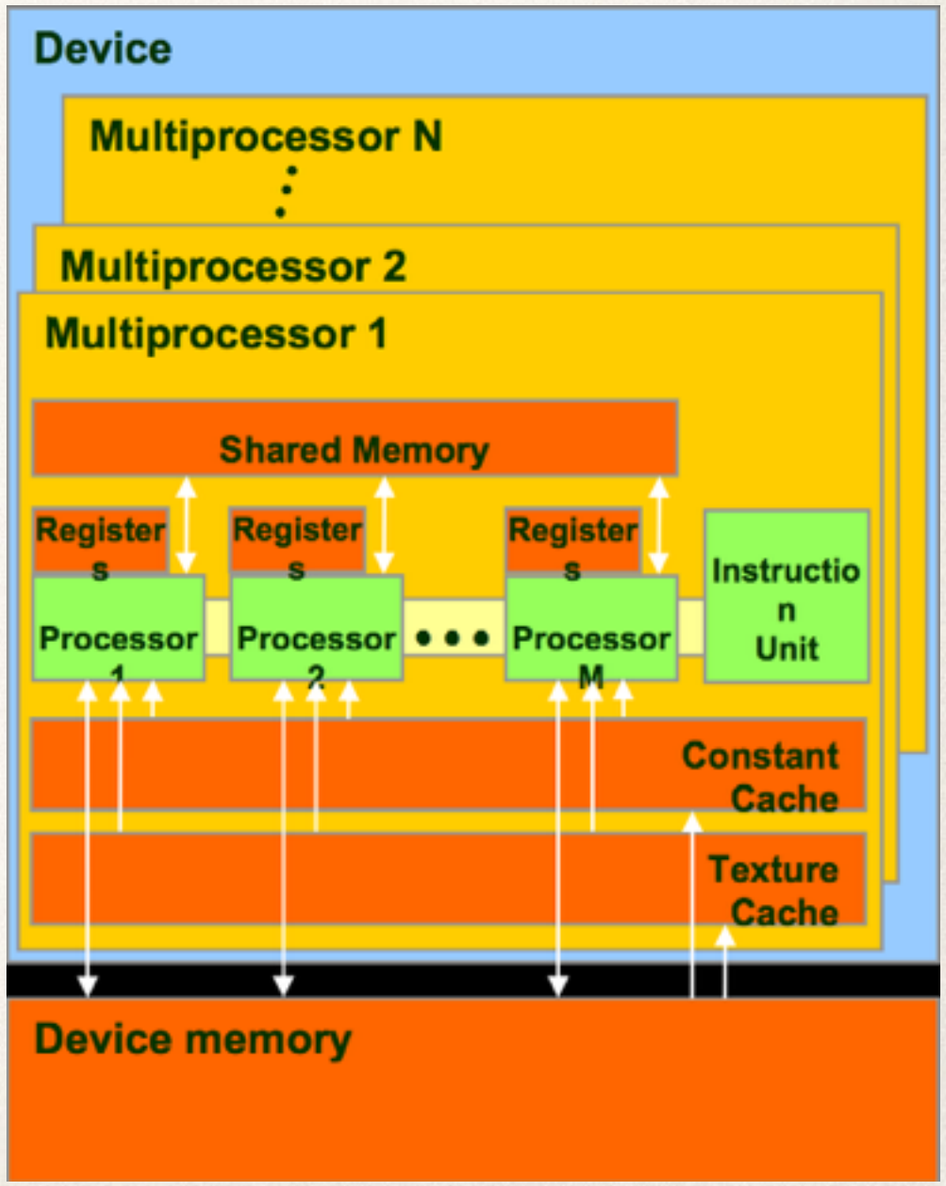
- Динамические массивы в shared памяти
- Всего лишь выделение выделение области области shared памяти
- Необходимо явно указывать смещения реальных объектов данных
- Помнить пользоваться syncthreads() при записи/чтении

Пересылка данных

- Копирование между хостом и устройством
  - cudaMemset
    - if(cudaMemset(ldCU,0,ns*sizeof(float))!=cudaSuccess)
    - throw Unsupported(this->Name,"CUDA has failed to zero-init LD.”);
  - cudaMemcpy
    - if(cudaMemcpy(bvCU,bvF32,ns*sizeof(FLT32),cudaMemcpyHostToDevice)!=cudaSuccess)
    - throw Unsupported(this->Name,"CUDA failed to upload BV.”);
- Освобождение памяти
  - cudaFree
    - if(cudaFree(wmCU)!=cudaSuccess)
    - throw Unsupported(Unsupported(this->Name,"CUDA has failed to de CUDA has failed to de-allocate WM allocate WM.");

Что такое ВОРП (WARP)?

- Device делает 1 grid в любой момент
- SM обрабатывает 1 или более blocks
- Каждый Block разделён на SIMD группы, внутри которых одни и те же инструкции выполняются реально одновременно над различными данными (warps) warp size=16/32
- Связывание в ворпы детерминировано в порядке нарастания threadID
- threadID=TIDX.x+TIDX.y*Dx +TIDX.z*Dx*Dy
- Важно! Полуворп – первая или вторая половина ворпа



Обращение с памятью памятью из ворпа

- НЕАТОМАРНЫЕ ИНСТРУКЦИИ (G80)
  - ЕСЛИ какая-либо инструкция исполняемая ворпом пишет в одно место в глобальной или общей памяти
  - ТО количество записей и их очерёдность недетерминированы
  - ОДНАКО по крайней мере одна запись состоится

- АТОМАРНЫЕ ИНСТРУКЦИИ (G92+)
  - ЕСЛИ какая-либо инструкция исполняемая ворпом пишет/читает/модифицирует одно место в глобальной памяти
  - ТО их очерёдность записей недетерминирована
  - ОДНАКО все записи состоятся последовательно

Атомарные функции

- Функции чтения/модификации/записи данных в глобальную память – основа построения алгоритмов стекового декодирования
- Работают только с целыми значениями (!)
- Гарантируют неизменность операнда в процессе операции
  - Арифметические функции: atomicAdd,atomicSub, atomicExch, atomicMax, atomicInc, atomicDec
    - int atomicAdd(int* address, int val);
  - Функция atomicCAS – Compare and store
    - int atomicCAS(int* address, int compare, int val);
  - Битовые функции atomicAnd, atomicOr, atomicXor
    - int atomicAnd(int* address, int val);

ЗАДАНИЕ

- На основе примера прямого перемножения матриц через глобальную память сделать перемножение матриц через локальную (shared) память.

Вопросы?