Архитектура и программирование потоковых многоядерных процессоров для научных расчётов

Лекция 3. Типы параллелизма
Модель памяти CUDA

Возможные виды параллелизма

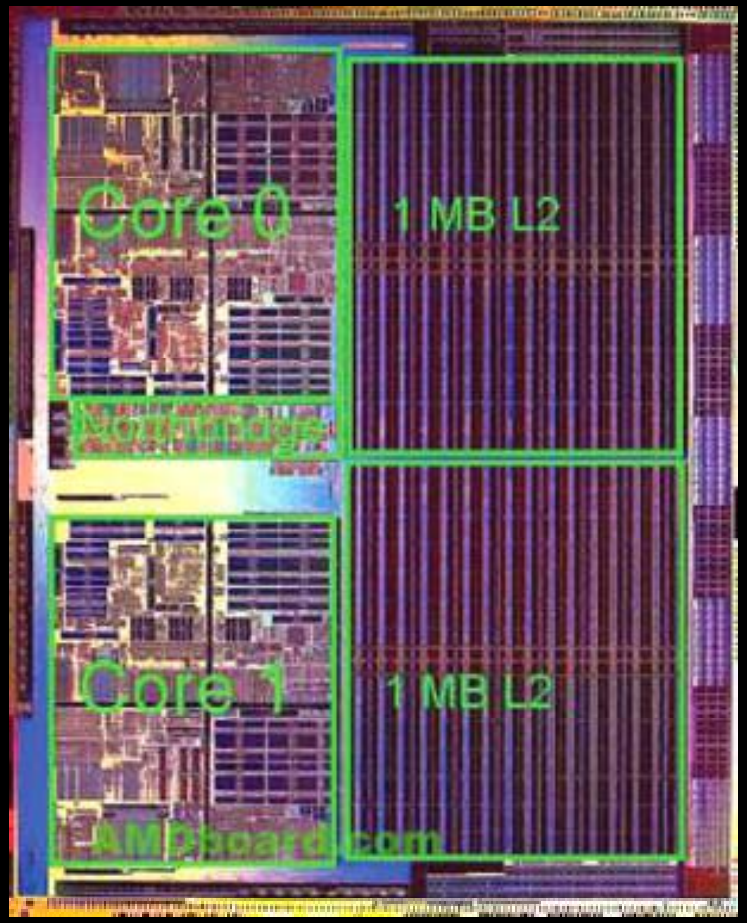
- Классификация Флина (SISD,SIMD, etc.) не вполне отражает вариации архитектур современных выч устройств
  - Сложно отнести к какому-то определённому классу
    - Физическая многоядерность VS Одно ядро над различными контекстами во времени
    - Различные виды конвейеров
      - Буферизованные / не буферизованные | синхронные / асинхронные
    - Разбиение на независимые пути выполнения на высоком и низком уровне
      - CPU – высокий уровень / GPU – низкий уровень
    - Параллелизм на различных данных, на независимых путях исполнения или на уровне инструкций
      - GPU – над данными / CPU – на путях исполнения
    - Векторные VS скалярные АЛУ

Параллелизм задач –возможность применения праллельных методов

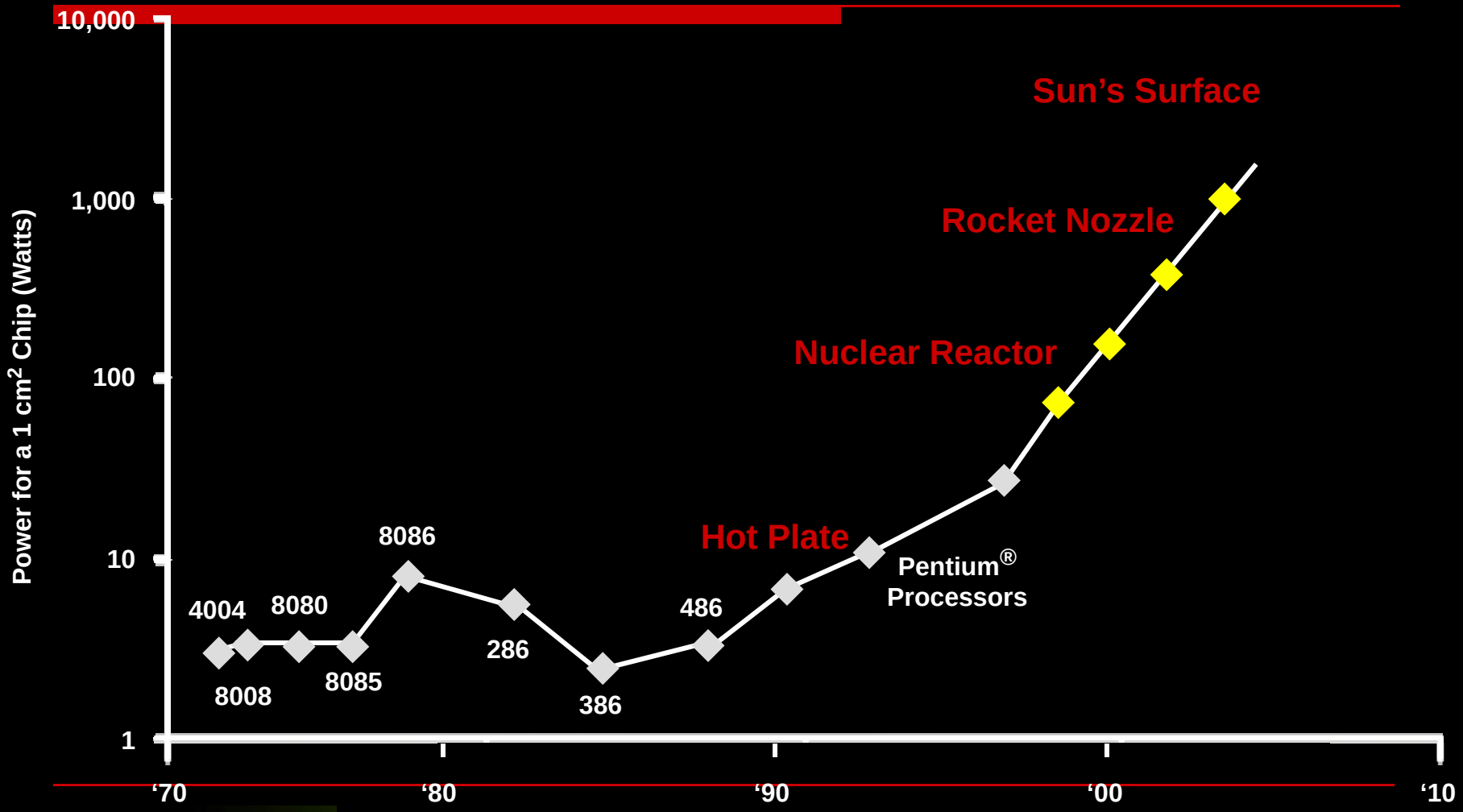
- **Если параллелизм отсутствует** в исходной задаче – параллелизм счётного устройства **не даст выигрыша**
- **Большие задачи** в подавляющем количестве случаев – **параллельны**
  - Способ мышления человека – составление сложных объектов и методов как комбинаций простых (**от простого к сложному**) – модель газа состоит из одинаковых частиц
  - Свойства физического мира (**принцип локальности взаимодействия**) - задачи динамики газов и жидкостей
  - Свойства математических объектов – перебор / независимые испытания
- Хорошее приложение это:
  - Много входов / много выходов
  - Существование параллельных путей ведения рассчётов
  - Ограниченность взаимодействия между отдельными путями
  - Большое кол-во вычислений на обращение к памяти

Традиции архитектуры CPU

- **Одноядерные процессоры** развивались ~ 40 лет
- Теперь количество ядер CPU **удваивается ежегодно**
- Традиционное ядро **не содержит поддержки параллелизма**
  - **Скалярная модель** исполнения программ
  - **Размещение тредов по ядрам** (баланс вычислительной нагрузки) – новинка для операционных систем
  - **Программист должен** примерно представлять конфигурацию PU
  - Нет универсального автоматического решения **(проблема кэширования)**



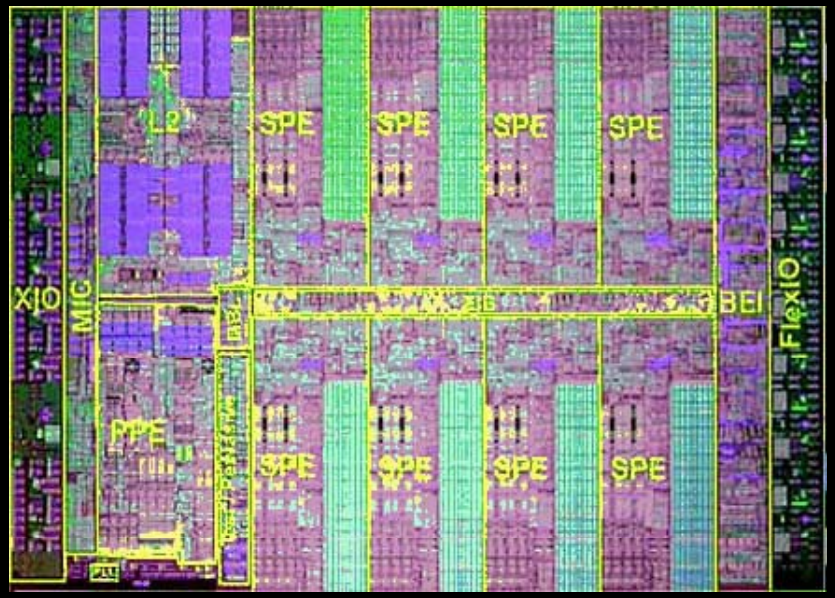
Почему нельзя далее наращивать скорость ядра?

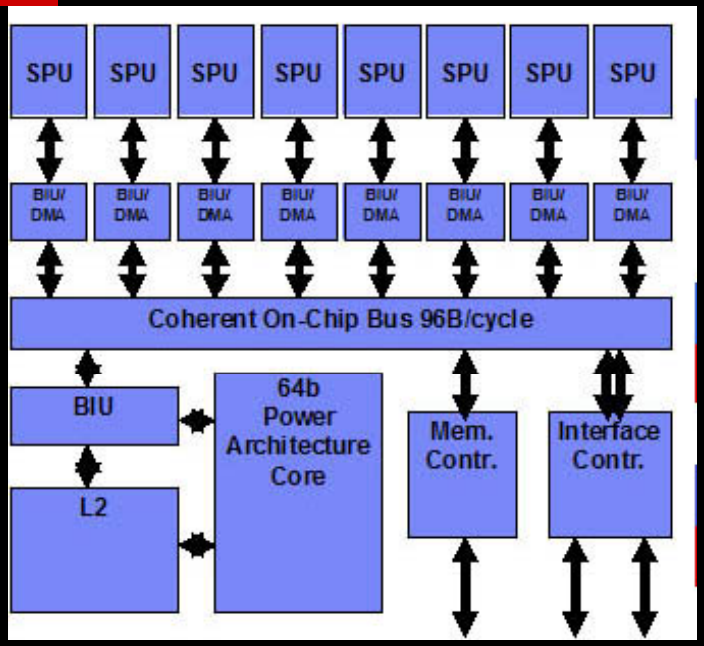


NVIDIA
World Leader in Visual Computing Technologies

Source: Pat Gelsinger, Intel, ISSCC 2001

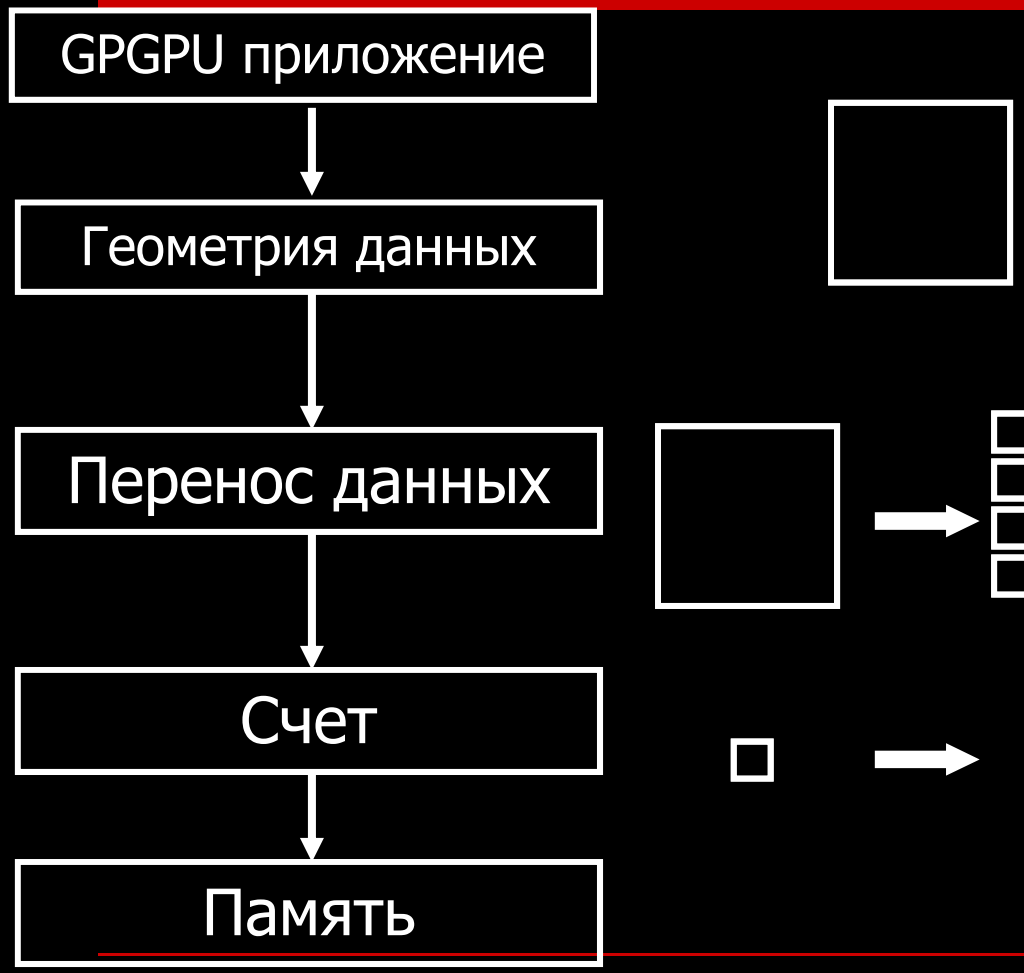
Пример архитектуры Cell процессора (IBM & Sony)





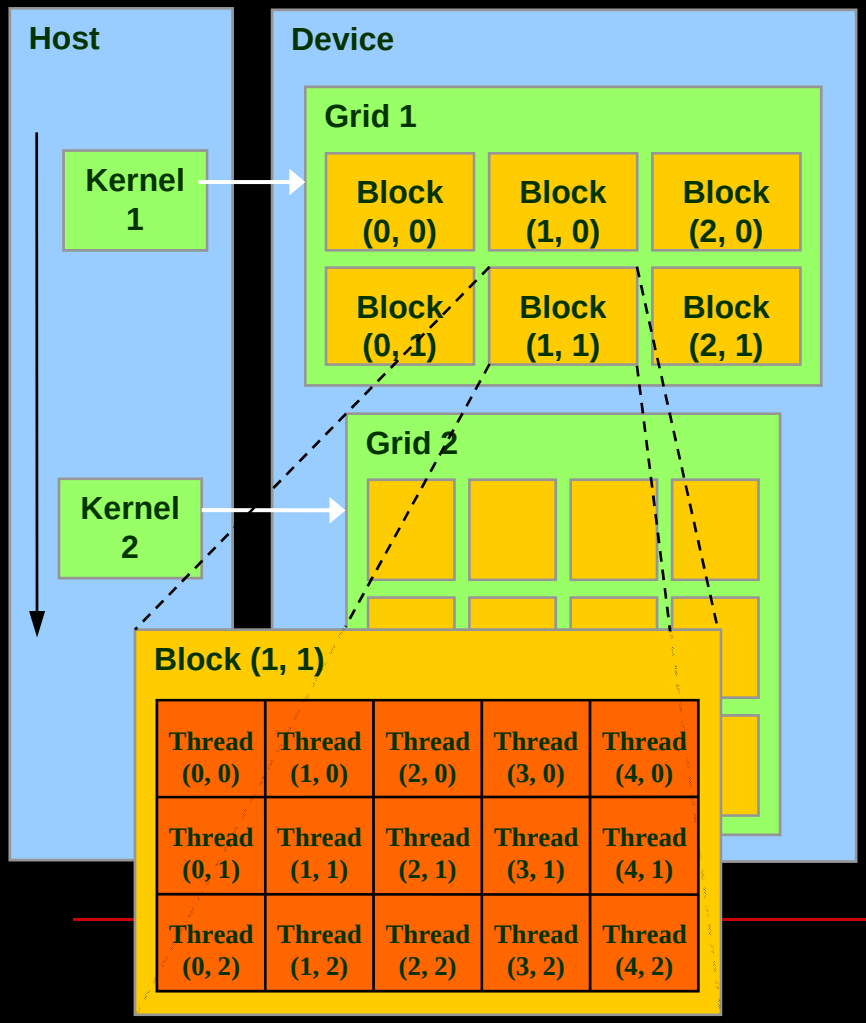
- **PPE** – Power Processor Element (распределяет нагрузку между отдельными вычислительными блоками)
- **SPU** – Synergistic Processing Unit (векторное ядро, цифровой сигнальный процессор)
- Сложен в программировании - предполагает **параллелизм высокого уровня**
- Программист пишет **код для каждого SPU отдельно**

Альтернатива – использование GPU



- Задачи схожие с графикой – позволяющие **параллелизм низкого уровня**
- **Одна программа** выполняется для **большого количества** частей задачи

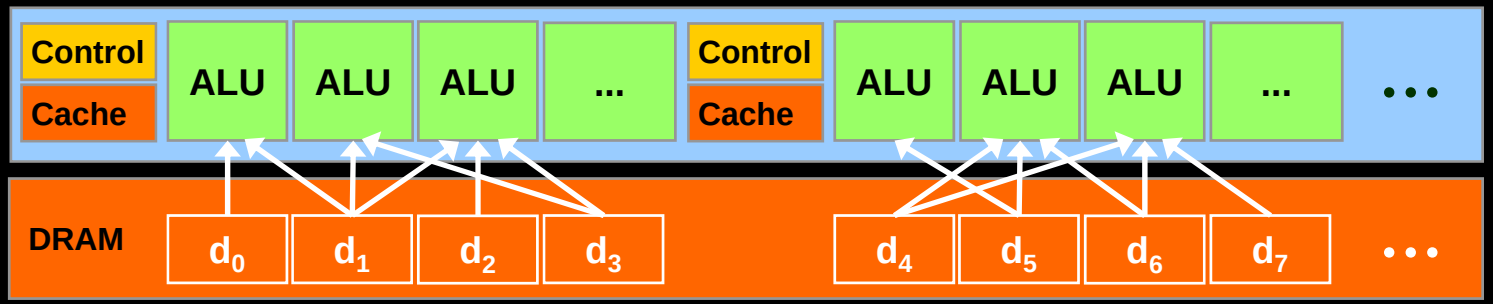
Вычислительная конфигурация GPU



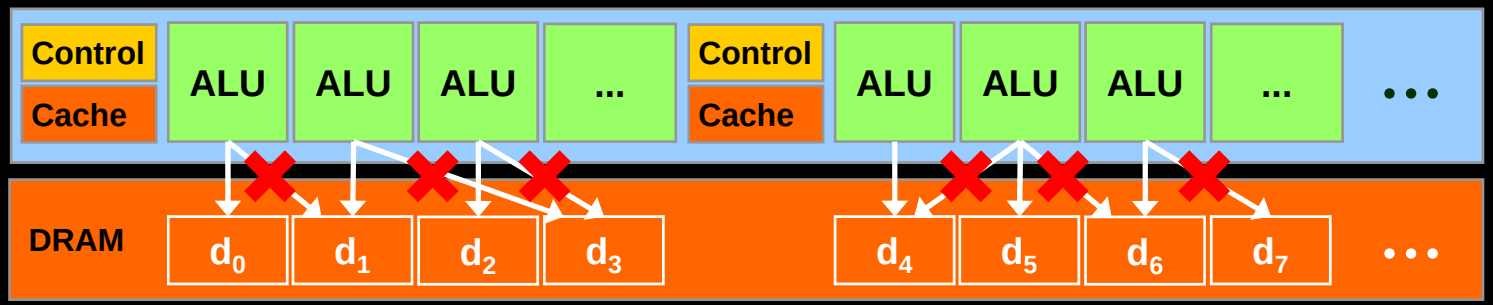
- Процессы объединяются в блоки (**blocks**), внутри которых они имеют общую память (**shared memory**) и синхронное исполнение
- Блоки объединяются в сетки (**grids**)
  - Нет возможности предсказать очерёдность запуска блоков в сетки
  - Между блоками нет и не может быть (см. выше) общей памяти

Gather и Scatter обмен данными в GPU

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

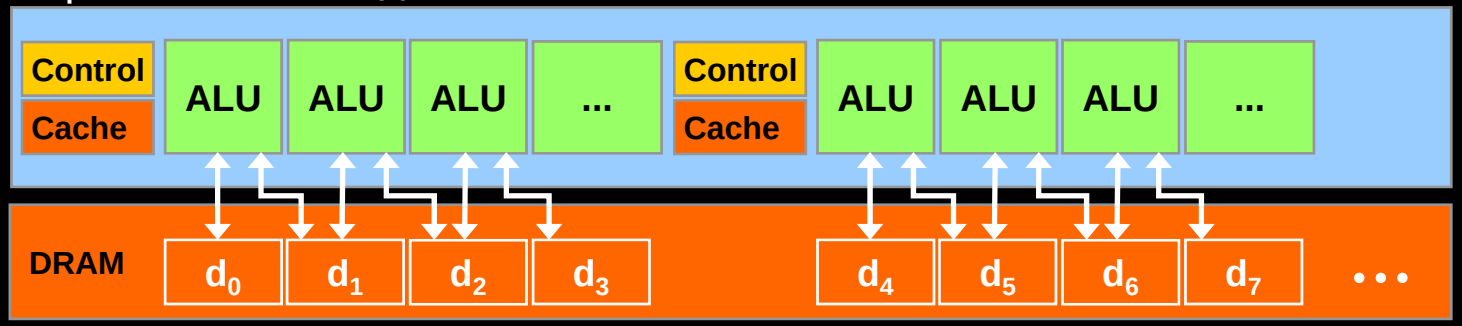


- Но **scatter** был невозможен / **CUDA позволяет делать scatter**

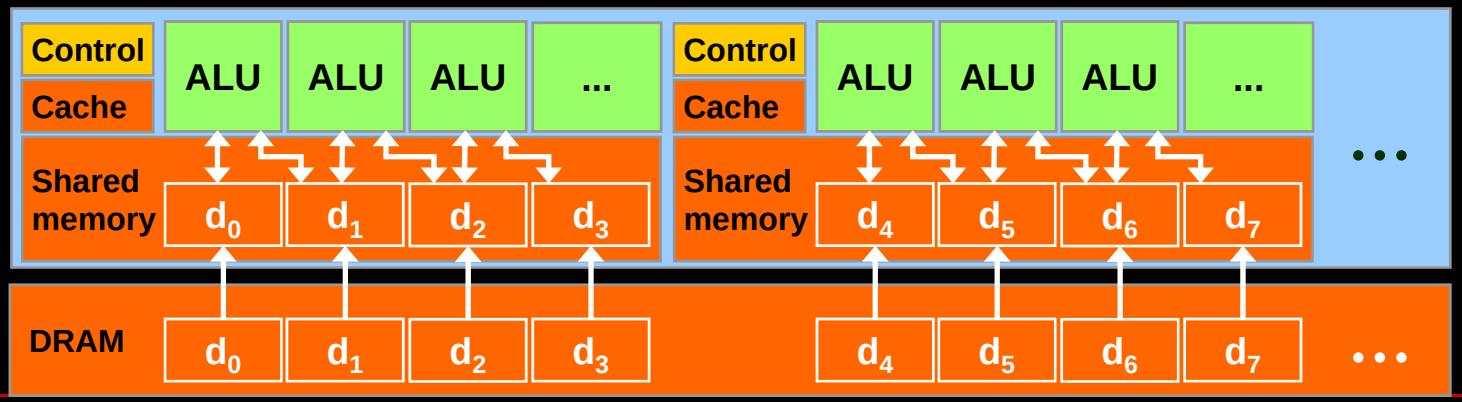


Использование общей памяти

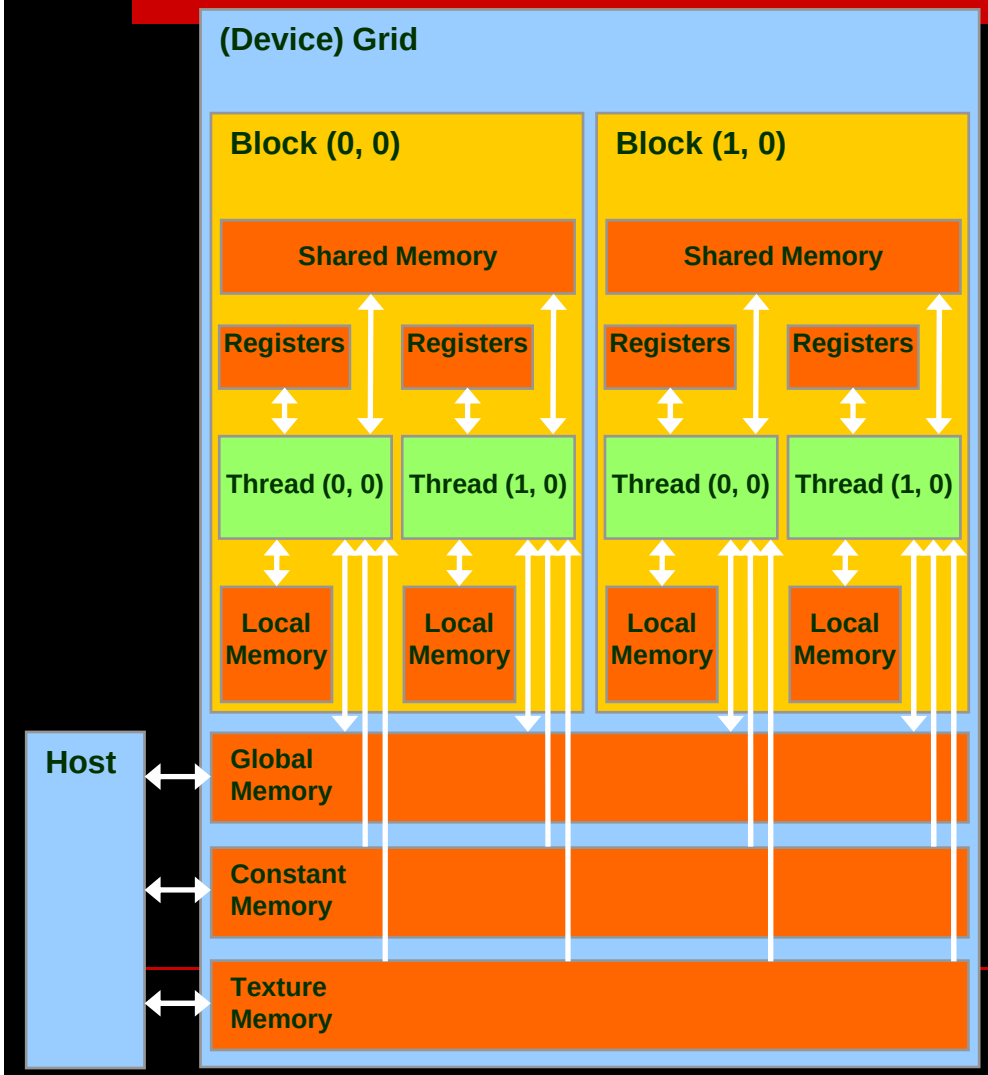
- Прямое чтение данных во многие АЛУ - неоптимально



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



Модель памяти GPU



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

Параметры GPU GeForce 8800GTX

- Programming model:
  - Максимум 512 процессов в блоке (512x512x64)
  - Максимальные размеры сетки (65535x65535)
  - Максимальный размер вычислительного ядра ~ 2 млн. инструкций

- HW architecture:
  - 16 мультипроцессоров (Streaming Multiprocessors - SM) / 128 потоковых процессоров (Streaming Processors)
  - Вплоть до 8 блоков исполняются одновременно на каждом SM
  - Вплоть до 24 варпов (**warps***) исполняются одновременно на каждом SM
  - Вплоть до 768 процессов исполняются одновременно на каждом SM
  - Количество регистров на SM - 8192
  - 16k общей памяти на SM / 16 банков
  - 64k памяти констант (кэшируется по 8k на SM)

  - 32-разрядная IEEE float арифметика

- *warp = часть блока, исполняемая на SM в SIMD виде

Процедура разработки программы

- **Global и Local память расположена на устройстве – DRAM – обращение к ней очень медленное**
- Общий подход к програмированию
  - 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
- Много-кретерийная оптимизация – нет простого алгоритма решения
  - **“Как сделать автомобили дешёвыми и безопасными”**

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**

C 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 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(this->Name,"CUDA has failed to de-allocate WM.");
    ```
    ```

C модификация – векторные типы

- Существуют следующие векторные типы:

- **char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4,**
- **short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4,**
- **int1, uint1, int2, uint2, int3, uint3, int4, uint4,**
- **long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4,**
- **float1, float2, float3, float4**

Структуры с полями x, y, z, w:

```
uint4 param;
int y = param.y;
```

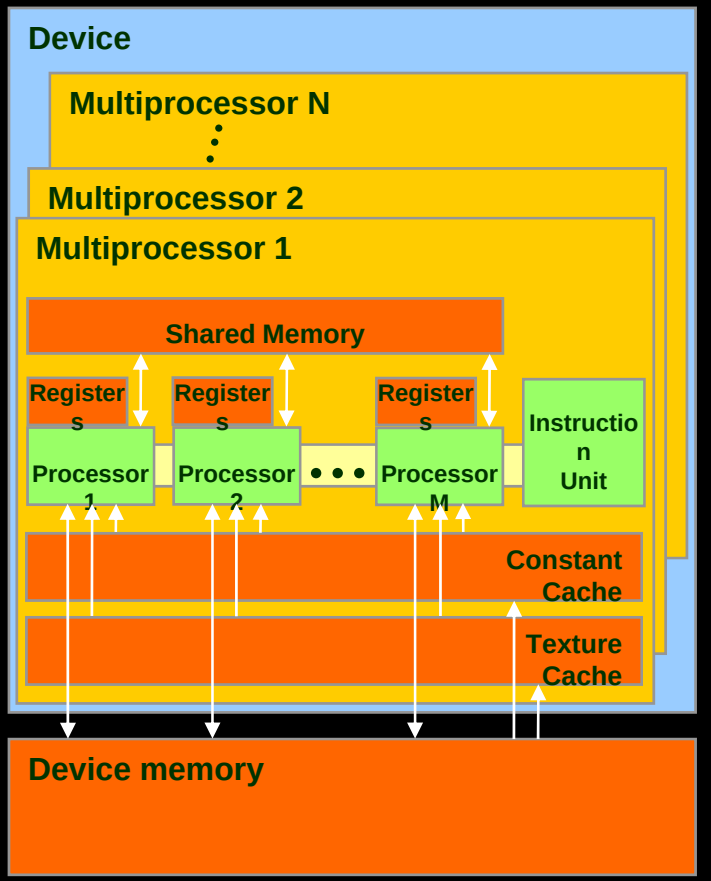
- **dim3**

  Основан на типе uint3

  Используется для указания параметров вычислительного ядра

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 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**

Когерентность общения с глобальной памятью (часть 1)

- Чтение 32- 64- 128- битных слов за 1 инструкцию
  - Тип размещаемых данных type должен удовлетворять условию **sizeof(type)** равен 4, 8, 16 байт
  - Нужно пользоваться модификаторами **__align__(8)** или **__align__(16)** для “плохих” типов (как например float3)

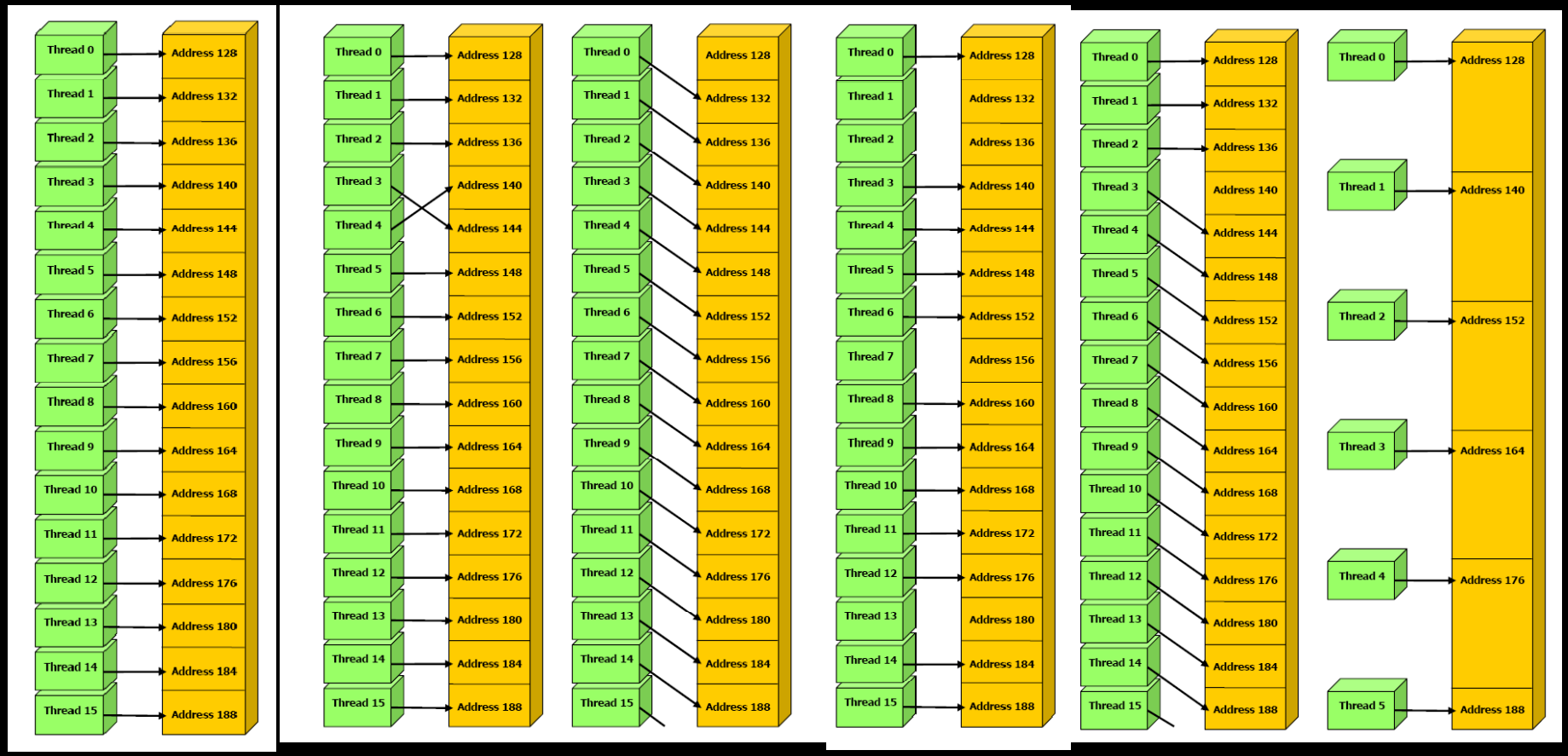
```
struct __align__(16) {
float a;
float b;
float c;
float d;
float e;
};
```

- Пример позволяет обойтись двумя 128-битными чтениями место 5 32-битных

Когерентность общения с глобальной памятью (часть 2)

- Каждый тред из **полуворпа** обращается к глобальной памяти одновременно
  - Необходимо, чтобы операция с памятью была оформлена в обращение к единой непрерывной области с адресом (где N – индекс в полуворпе)
    HalfWarpBaseAddress + N
  - **HalfWarpBaseAddress** типа **type*** и равна **16*sizeof(type)**
- Подобные требования рекомендуется выполнять **для целых ворпов** (для совместимости с будущими устройствами)

Примеры когерентных и некогерентных обращений



Итоги лекции

- **В результате лекции студенты должны :**
  - **Понимать возможности использования GPU для осуществления параллельных вычислений**
  - **Иметь понятие об организации разработки приложений**
    - **Цикл планирования приложения**
    - **Модель памяти GPU устройства**
    - **Модификации языка C используемые для обращения с элементами данных**
  - **Достаточные знания для начала самостоятельной работы**