НОЦ "ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ"
APPLIED PARALLEL COMPUTING E&R CENTER

Лекция 2

Иерархия памяти CUDA. Эффективное использование разделяемой памяти.

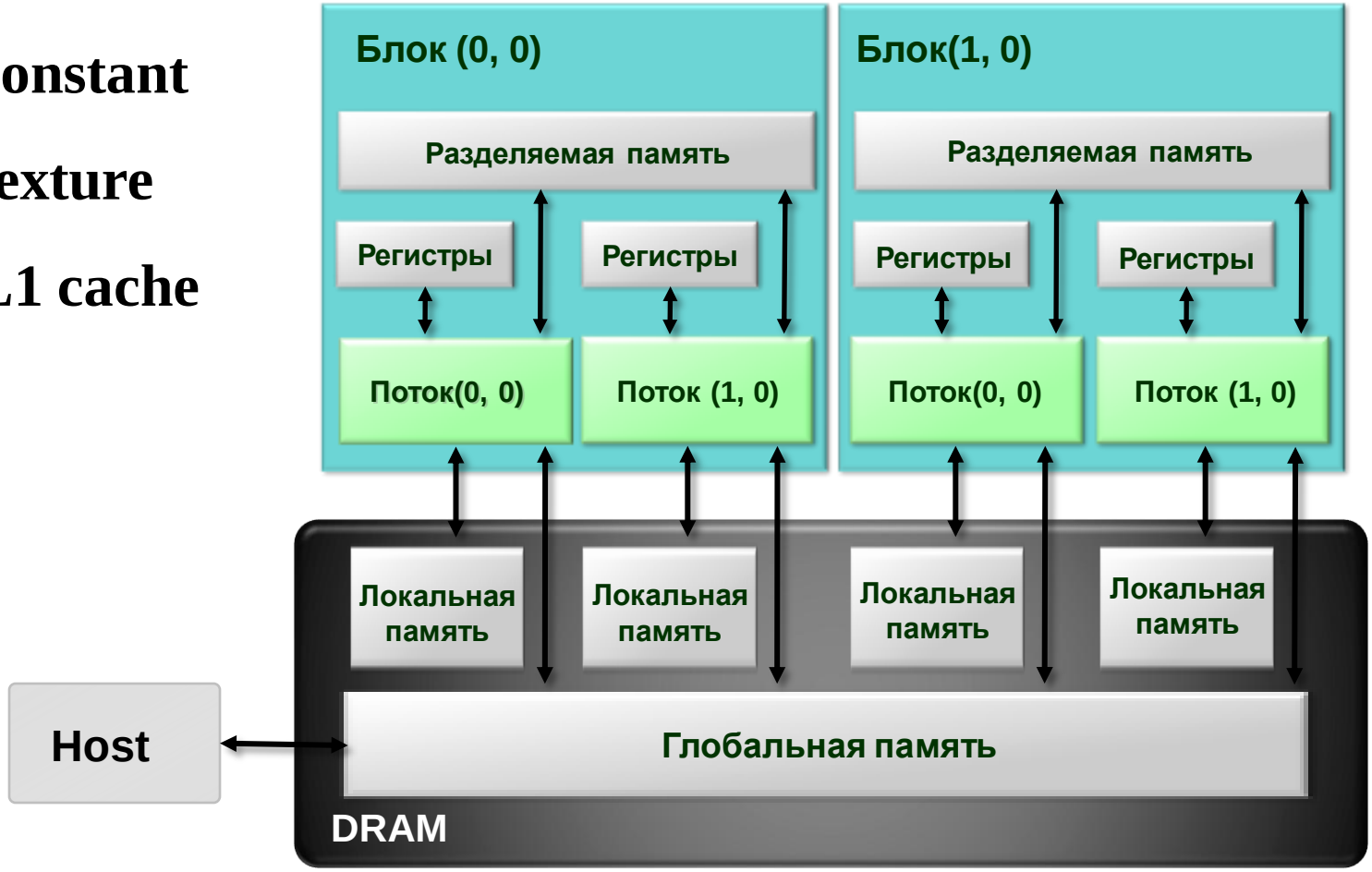
Перепёлкин Е.Е.

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	**R/W**	**Per-grid**	**Низкая (DRAM)**
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	R/O	Per-grid	Высокая(L1 cache)

Пространство памяти CUDA

- constant
- texture
- L1 cache



Типы памяти в CUDA

- **Самая быстрая – *shared* (on-chip) и регистры**
- **Самая медленная – глобальная (DRAM)**
- **Для ряда случаев можно использовать кэшируемую константную и текстурную память**
- **Доступ к памяти в CUDA идет отдельно для**
  - **каждой половины warp'а (*half-warp*) Tesla 10**
  - **warp'а (Tesla 20)**

Работа с памятью в CUDA

- Основа оптимизации – оптимизация работы с памятью
- Максимальное использование shared-памяти
- Использование специальных паттернов доступа к памяти, гарантирующих эффективный доступ
- Паттерны работают независимо в пределах каждого half-warp'а / warp'а

Работа с глобальной памятью в CUDA

```
cudaError_t cudaMalloc  ( void ** devPtr, size_t size );

cudaError_t cudaFree ( void * devPtr );


cudaError_t cudaMemcpy ( void * dst, const void * src,
size_t count, enum cudaMemcpyKind kind );


cudaError_t cudaMemcpyAsync ( void * dst,
            const void * src, size_t count,
            enum cudaMemcpyKind kind,
            cudaStream_t stream );


cudaError_t cudaMemset ( void * devPtr, int value,
                         size_t count );
```

Работа с глобальной памятью в CUDA

Пример работы с глобальной памятью

```
float * devPtr;              // pointer device memory
                             // allocate device memory
cudaMalloc ( (void **) &devPtr, 256*sizeof ( float );

                             // copy data from host to device memory
cudaMemcpy ( devPtr, hostPtr, 256*sizeof ( float ),
                      cudaMemcpyHostToDevice );

                             // process data

                             // copy results from device to host
cudaMemcpy ( hostPtr, devPtr, 256*sizeof( float ),
               cudaMemcpyDeviceToHost );

                             // free device memory
cudaFree    ( devPtr );
```

CUDA Compute Capability

- Возможности GPU обозначаются при помощи ***Compute Capability***, например 1.1
- Старшая цифра соответствует архитектуре
- Младшая – небольшим архитектурным изменениям
- Можно получить из полей *major* и *minor* структуры **cudaDeviceProp**

Compute Capability

GPU	Compute Capability
Tesla S2070/C2070/2090	2.0/2.1
Tesla S1070/C1060	1.3
GeForce GTX 260	1.3
GeForce 9800 GX2	1.1
GeForce 9800 GTX	1.1
GeForce 8800 GT	1.1
GeForce 8800 GTX	1.0

RTM **Appendix A.1** CUDA Programming Guide

Compute Capability

- **Compute Caps. – доступная версия CUDA**
  - Разные возможности HW
  - Пример:
    - в 1.1 добавлены атомарные операции в global memory
    - в 1.2 добавлены атомарные операции в shared memory
    - в 1.3 добавлены вычисления в double
    - в 2.0 добавлены управление кэшем и др. операции
- **Узнать доступный Compute Caps. можно через cudaGetDeviceProperties ()**
  - См. CUDAHelloWorld
- **Сегодня Compute Caps:**
  - Влияет на правила работы с глобальной памятью

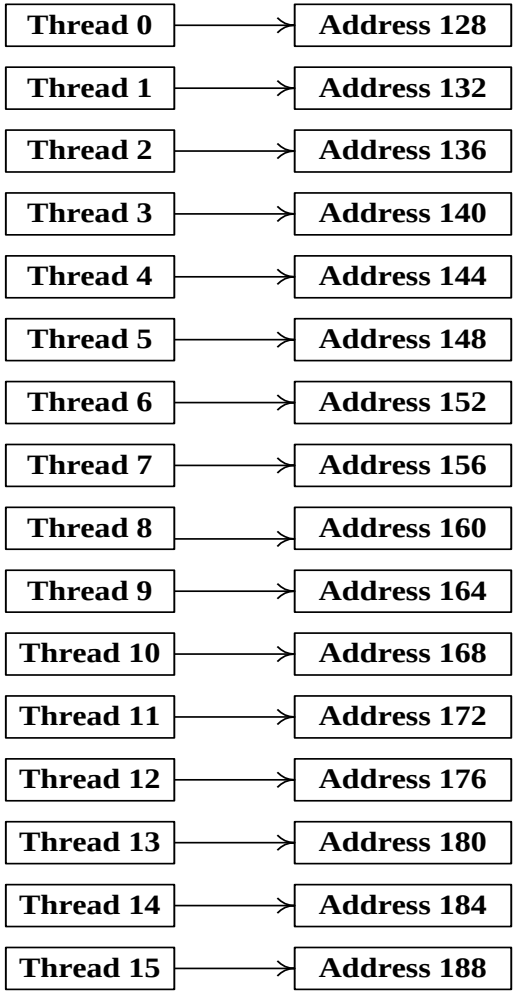
Объединение запросов

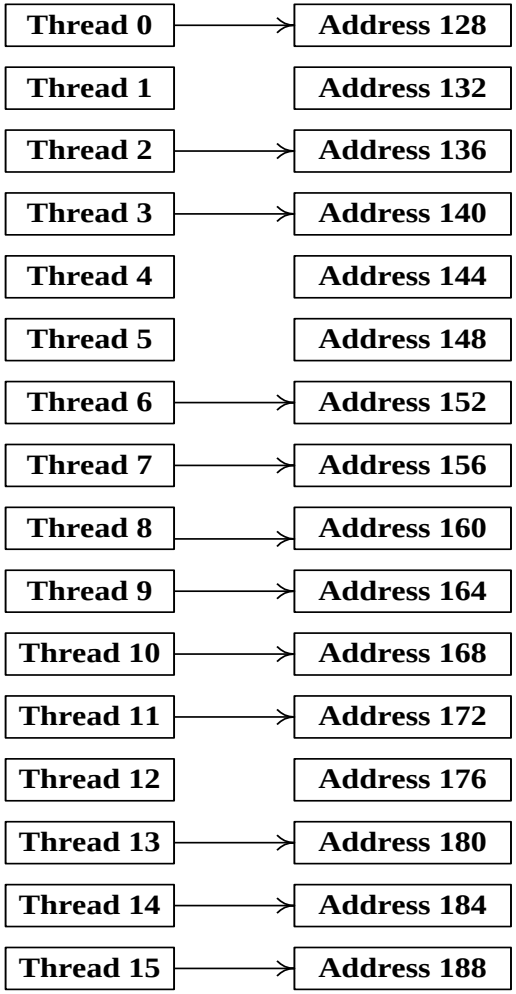
- GPU умеет объединять ряд запросов к глобальной памяти в один блок (транзакцию)
- Независимо происходит для каждого *half-warp*'а / *warp*'а (CC 1.x / 2.x)

GPU с CC 1.0/1.1

- Нити обращаются к
  - 32-битовым словам, давая 64-байтовый блок
  - 64-битовым словам, давая 128-байтовый блок
- Все 16 слов лежат в пределах блока
- *k*-ая нить *half-warp*'а обращается к *k*-му слову блока
- Блок выровнен

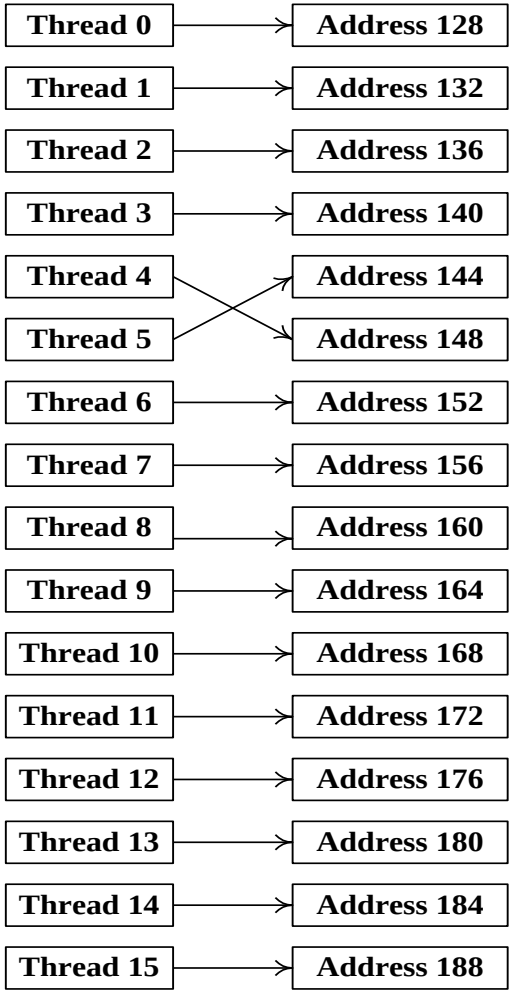
GPU с CC 1.0/1.1

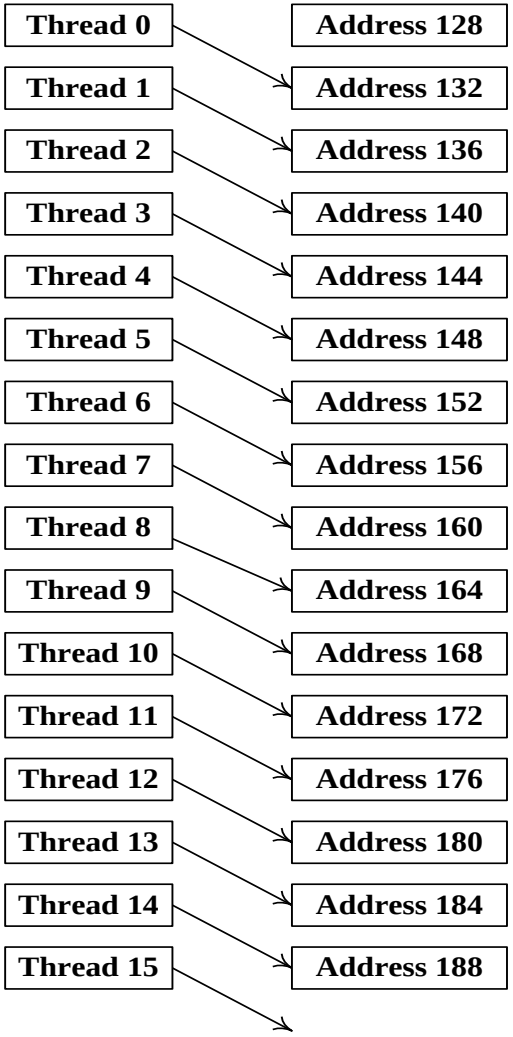




Coalescing

GPU с CC 1.0/1.1





Not Coalescing

GPU с CC 1.2/1.3

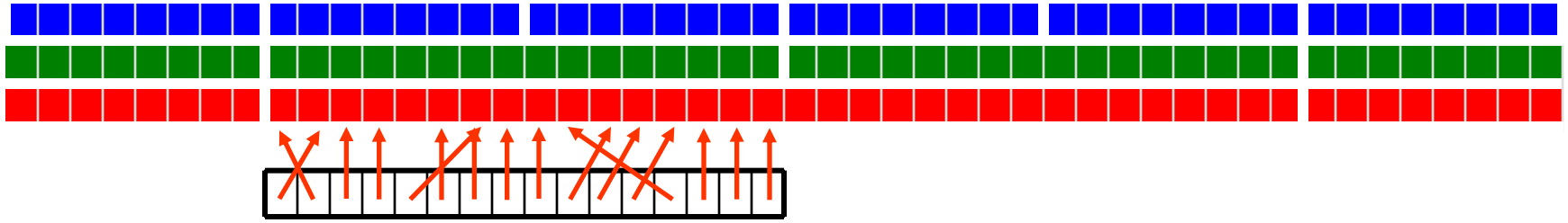
- Нити обращаются к
  - 8-битовым словам, дающим один 32-байтовый сегмент
  - 16-битовым словам, дающим один 64-байтовый сегмент
  - 32-битовым словам, дающим один 128-байтовый сегмент
- Получающийся сегмент выровнен по своему размеру

Coalescing

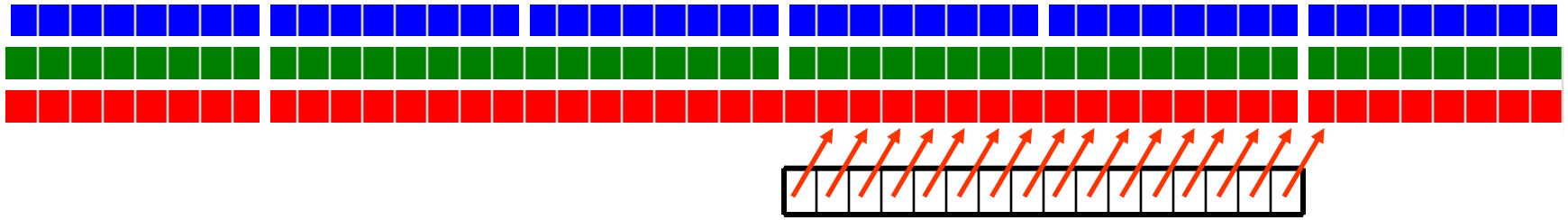
- Если хотя бы одно условие не выполнено
  - 1.0/1.1 – 16 отдельных транзакций
  - 1.2/1.3 – объединяет их в блоки (2,3,…) и для каждого блока проводится отдельная транзакция
- Для 1.2/1.3 порядок в котором нити обращаются к словам внутри блока не имеет значения (в отличии от 1.0/1.1)

Объединение для GPU с CC 1.2/1.3

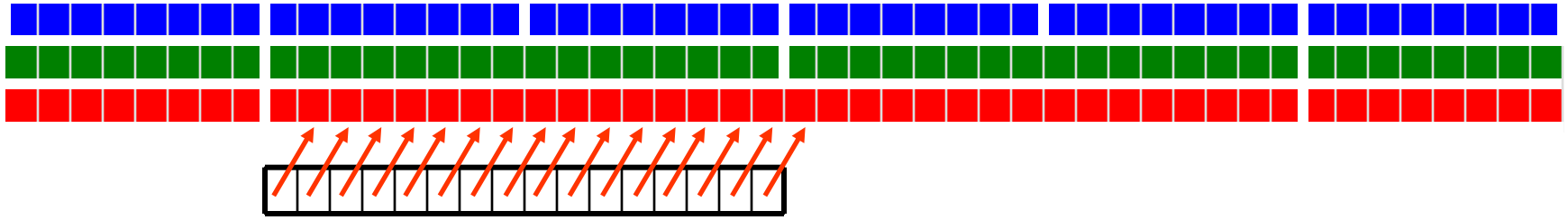
1 транзакция - 64B



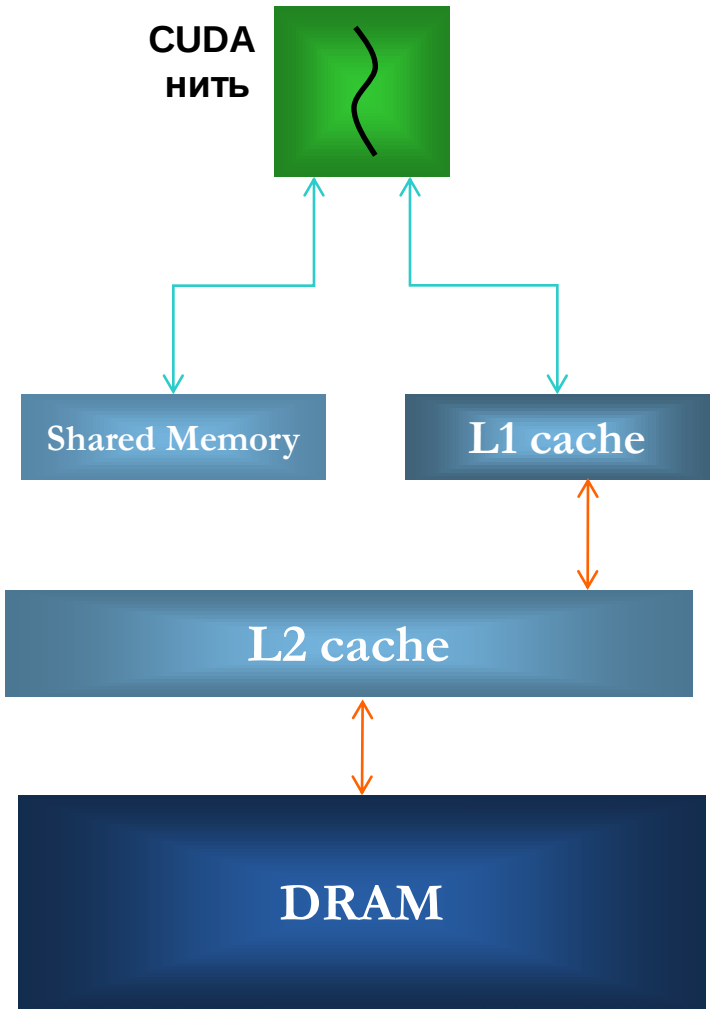
2 транзакции - 64B и 32B



1 транзакция - 128B



Fermi – Подсистема памяти



- Настраиваемый L1 кэш для каждого SM
  - 16КБ SMEM, 48КБ L1
  - 48КБ SMEM, 16КБ L1
- Общий L2 кэш для всех SM
  - 768КБ
- Атомарные операции
  - 20x быстрее чем на Tesla
- ECC, коррекция ошибок
  - Single-Error Detect
  - Double-Error Correct

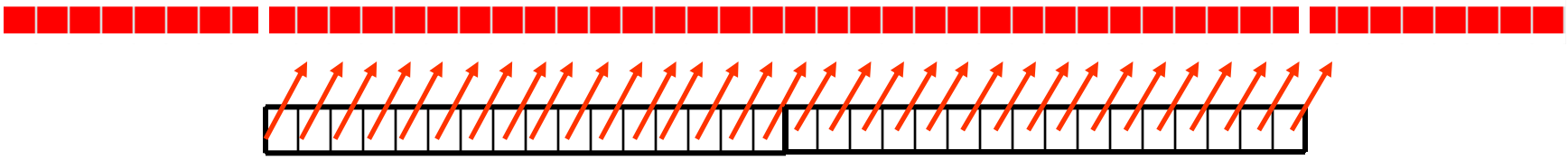
Coalescing CC 2.0

- **Флаги компиляции**
  - **использовать L1 и L2: -Xptxas –dlcm=ca**
  - **использовать L2: -Xptxas –dlcm=cg**
- **Кэш линия 128 байт и выравнивание по 128 байт в глобальной памяти**
- **Объединения происходит на уровне warp’ов**

Coalescing CC 2.0

- Объединение запросов в память для 32 нитей
- L1 включен – всегда идут запросы по 128B с кэшированием в L1

2 транзакции - 2 x 128B

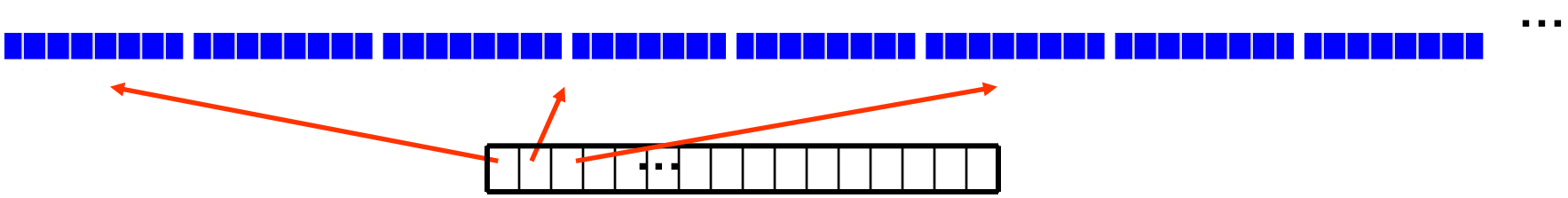


следующий варп скорей всего только 1 транзакция, так как попадаем в L1

Coalescing CC 2.0

- L1 выключен – всегда идут запросы по 32B
- Лучше для разреженного доступа к памяти

32 транзакции по 32B, вместо 32 x 128B



Coalescing

- **Можно добиться заметного увеличения скорости работы с памятью**
- **Лучше использовать не массив структур, а набор массивов отдельных компонент – это позволяет использовать *coalescing***

Использование отдельных массивов

```
struct vec3
{
  float x, y, z;
};
vec3 * a;

float x = a [threadIdx.x].x;
float y = a [threadIdx.x].y;
float z = a [threadIdx.x].z;
```

Не можем использовать *coalescing* при чтении данных

```
float * ax, * ay, * az;

float x = ax [threadIdx];
float y = ay [threadIdx];
float z = az [threadIdx];
```

Поскольку нити одновременно обращаются к последовательно лежащим словам памяти, то будет происходить *coalescing*

Решение СЛАУ

$$Ax=f,$$

A – матрица размера N*N,

f – вектор размера N

- Традиционные методы ориентированы на последовательное вычисление элементов и нам не подходят
- Есть еще итеративные методы

Итеративные методы

- **Эффективны когда**
  - Матрица A сильна разрежена
  - Параллельные вычисления
- **В обоих случаях цена (по времени) одной итерации O(N)**

Метод простых итераций

$$Ax = f$$

$$x_k = \frac{1}{a_{kk}}\left(f_k - \sum_{i \neq k} a_{ki} x_i \right), \quad 1 \le k \le n$$

$$\sum_{i \neq k} \left| \frac{a_{ki}}{a_{kk}} \right| < 1, \quad \textit{достаточное условие сходимости}$$

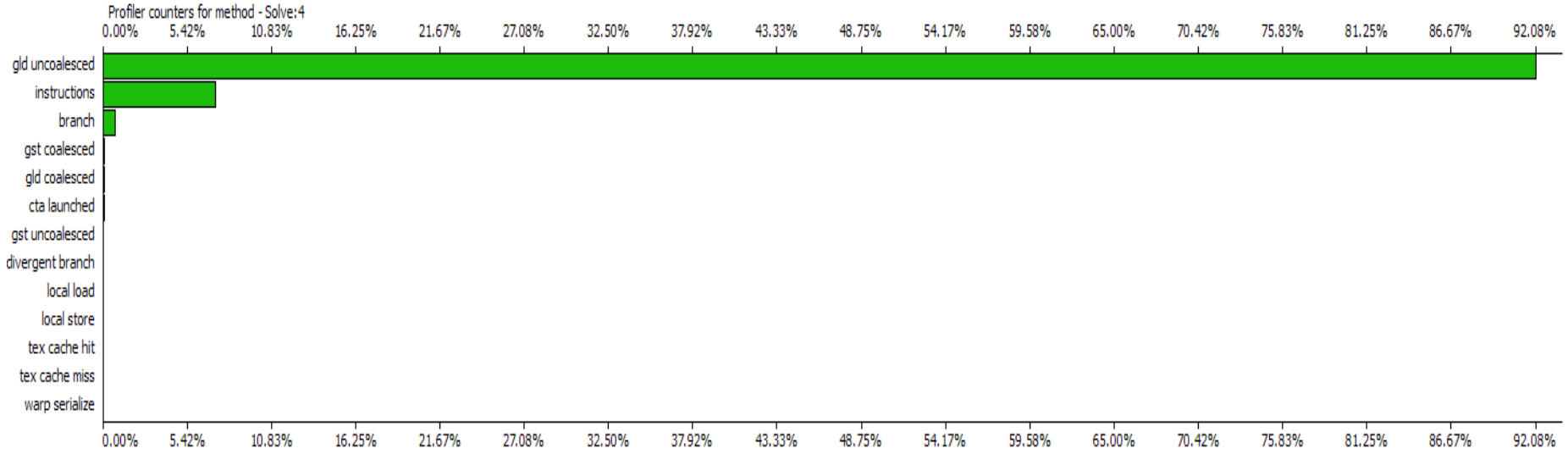
$$x_k^{s+1} = x_k^s + \frac{1}{a_{kk}}\left(f_k - \sum_{i=1}^{n} a_{ki} x_i^s \right), \quad 1 \le k \le n$$

Код на CUDA

coalescing ?

```
//
// one iteration
//
__global__ void kernel ( float * a, float * f,
                         float * x0, float * x1, int n )
{
  int   idx = blockIdx.x * blockDim.x + threadId.x;
  int   ia  = n * idx;
  float sum = 0.0f;

  for ( int i = 0; i < n; i++ ) sum += a [ia + i] * x0 [i];
  alpha = 1.0f / a [ia + idx];
  x1 [idx] = x0 [idx] + alpha * (f[idx] - sum);
}
```

Profiler СЛАУ. Вариант 1

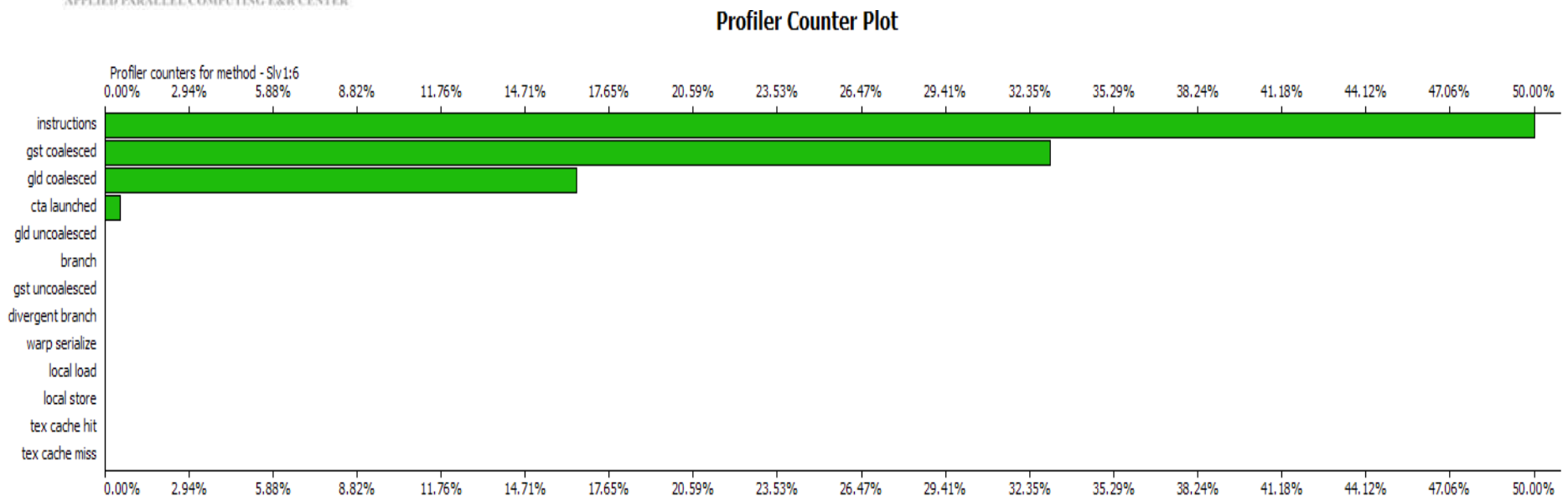


- Почти 90% uncoalesced доступ в память
- Инструкции всего 8%
- Проблема в считывании матрицы A и вектора X.

Решение СЛАУ. Вариант 2

```
// device
__global__ void Slv1 ( float *dA, float dx0, float *dSum, float *dAA, int
    N, int j)
{ int t   = blockIdx.x * blockDim.x + threadIdx.x;

  if(j==0) dSum[t]=0.f;
  float AA=dA[t+j*N];
  if(j==t) dAA[t]=AA;
  dSum[t]+=AA*dx0;
}
__global__ void Slv2 ( float *dA, float *dX0, float *dX1, float *dSum,
    float *dF,
                       float *dAA, int N)
{ int t   = blockIdx.x * blockDim.x + threadIdx.x;
  dX1[t]=dX0[t]+(dF[t]-dSum[t])/dAA[t];
}

// Host
  for(j=0;j<N;j++) Slv1<<<N_blocks,N_thread>>>(dA,hX0[j],dSum,dAA,N,j);
  cudaThreadSynchronize();
  cutilCheckMsg("Faild execution failed");

  Slv2<<<N_blocks,N_thread>>>(dA,dX0,dX1,dSum,dF,dAA,N);
  cudaThreadSynchronize();
  cutilCheckMsg("Faild execution failed");
```

Profiler СЛАУ. Вариант 2



- Весь доступ в память – coalesced !
- Инструкции 50%
- Однако, можно получить дополнительное ускорение за счет текстурной памяти

Типы памяти в CUDA

Тип памяти	**Доступ**	**Уровень выделения**	**Скорость работы**
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	**R/W**	**Per-block**	**Высокая(on-chip)**
Глобальная	R/W	Per-grid	Низкая (DRAM)
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	R/O	Per-grid	Высокая(L1 cache)

shared-память в CUDA

- Самая быстрая (on-chip)
- Сейчас всего 16/48 Кбайт на один мультипроцессор
- Совместно используется всеми нитями блока
- Отдельное обращение для каждой половины warp'а (*half-warp*)
- Как правило, требует явной синхронизации

Типичный паттерн использования

1. Загрузить необходимые данные в shared-память (из глобальной)
2. __syncthreads ()
3. Выполнить вычисления над загруженными данными
4. __syncthreads ()
5. Записать результат в глобальную память

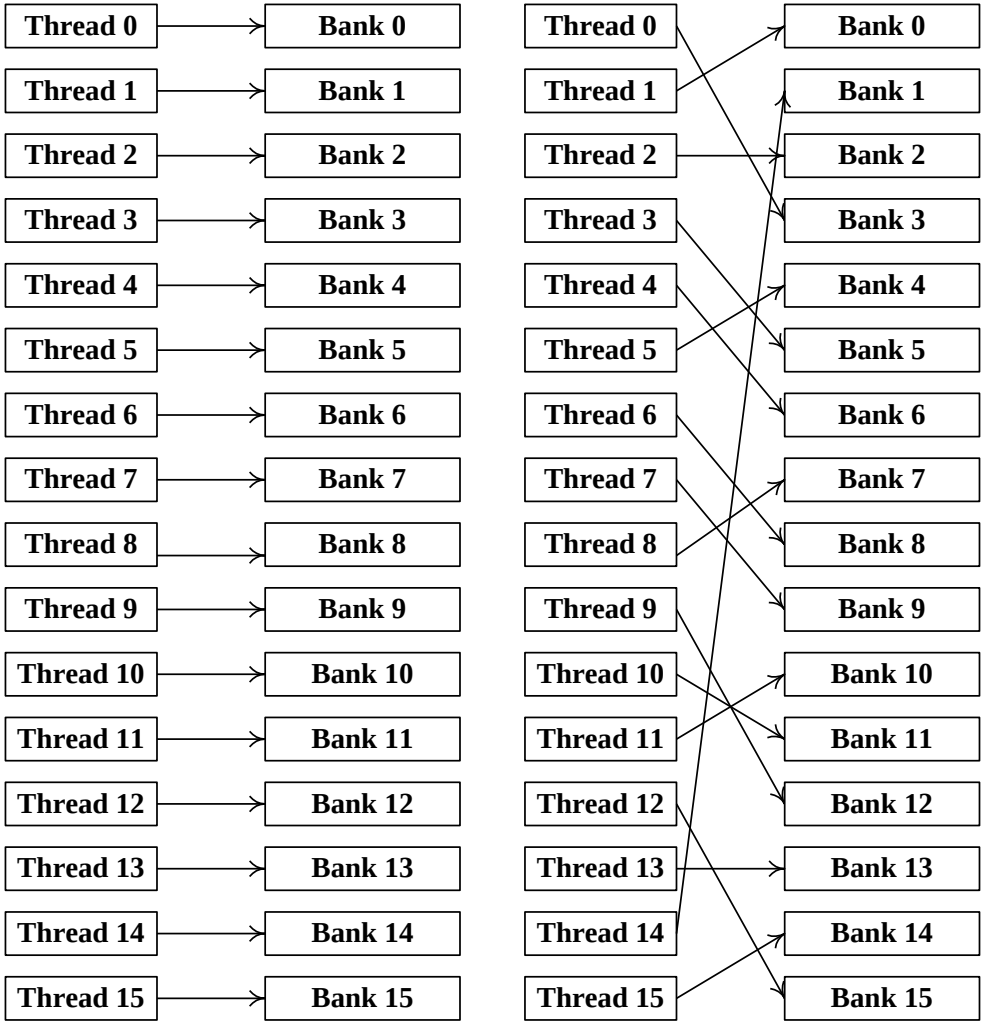
Эффективная работа с shared-памятью

- Для повышения пропускной способности вся shared-память разбита на 16 банков (Tesla 10) и на 32 банка (Tesla 20)
- Каждый банк работает независимо от других
- Можно одновременно выполнить до 16 обращений к shared-памяти
- Если идет несколько обращений к одному банку, то они выполняются по очереди

Эффективная работа с shared-памятью

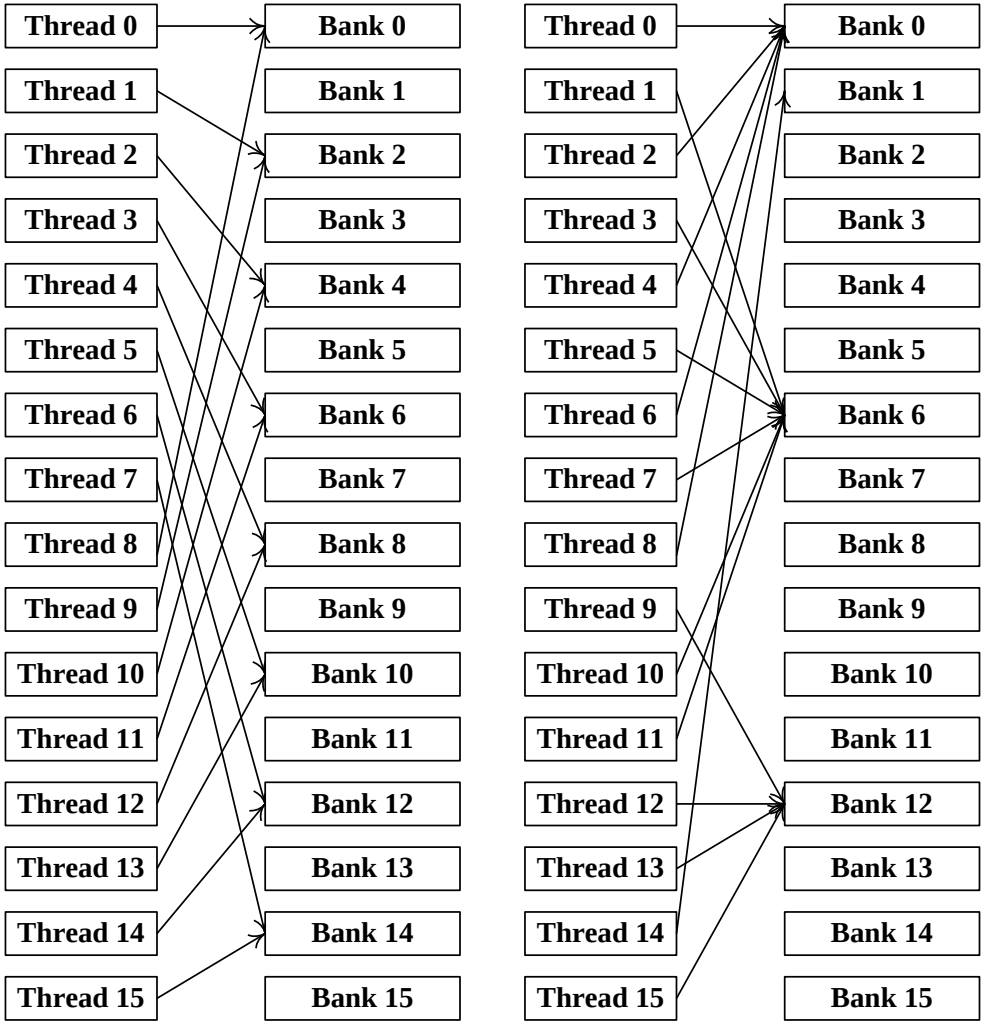
- Банки строятся из 32-битовых слов
- Подряд идущие 32-битовые слова попадают в подряд идущие банки
- ***Bank conflict*** – несколько нитей из одного half-warp'а обращаются к одному и тому же банку
- Конфликта не происходит если все 16 нитей обращаются к одному слову (*broadcast*)

Бесконфликтные паттерны доступа



Tesla 10

Паттерны с конфликтами банков



- Слева – конфликт второго порядка – вдвое меньшая скорость
- Несколько конфликтов, до 6-го порядка

Доступ к массиву элементов

```
__shared__ float a [N];

float x = a [base + threadIdx.x];
```

Нет конфликтов

```
__shared__ short a [N];

short x = a [base + threadIdx.x];
```

Конфликты 2-го порядка

```
__shared__ char a [N];

char x = a [base + threadIdx.x];
```

Конфликты 4-го порядка

```
__shared__ double a [N];

double x = a [base + threadIdx.x];
```

Конфликты 2-го порядка для CC 1.x

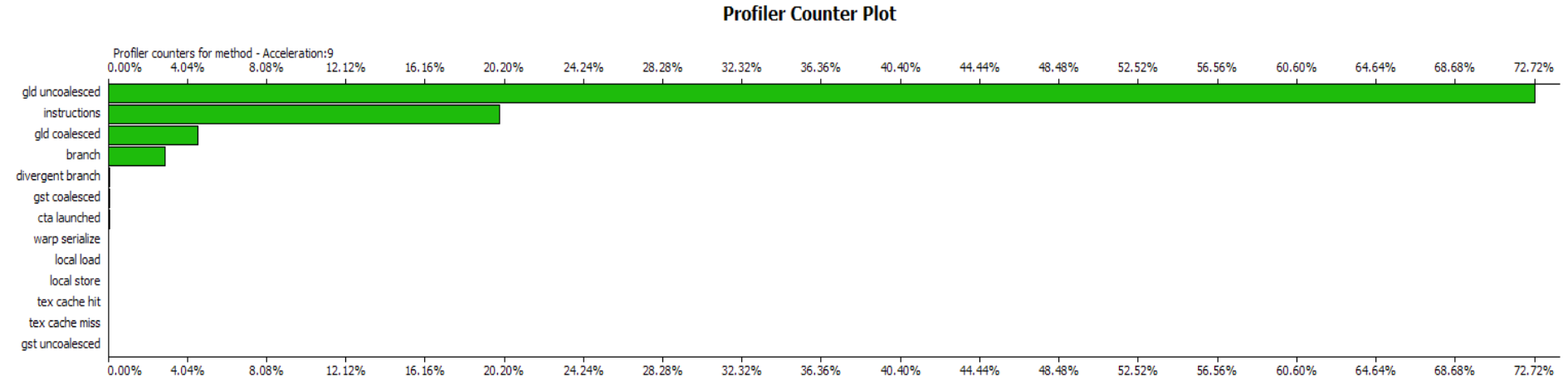
Нет конфликтов для CC 2.x

Задача N-тел. GPU вариант 1

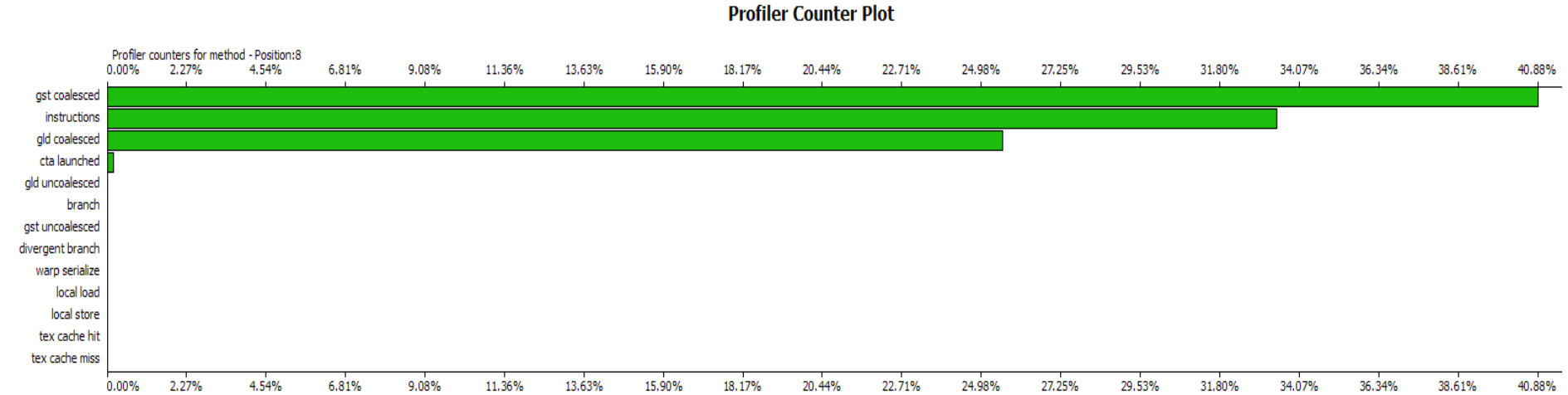
```
__global__ void Acceleration_GPU ( float *X, float *Y,
                                   float *AX, float *AY,  int N )
{int id = threadIdx.x + blockIdx.x*blockDim.x;
 float ax = 0.f;
 float ay = 0.f;
 float xx, yy, rr;

 for ( int j = 0; j < N;  j ++ )
 { if ( j ! = id )
    { xx = X [ j ] - X [ id ]; yy  = Y [ j ] - Y [ id ];
      rr = sqrtf ( xx * xx + yy * yy );
      if ( rr > 0.01f ) { rr  = 10.f / (rr * rr * rr );
                          ax + = xx * rr; ay + = yy * rr; }
    }
 }
 AX [ id ] = ax; AY [ id ] = ay;
}
```

Задача N-тел. Вариант 1

Kernel Acceleration



Kernel Position

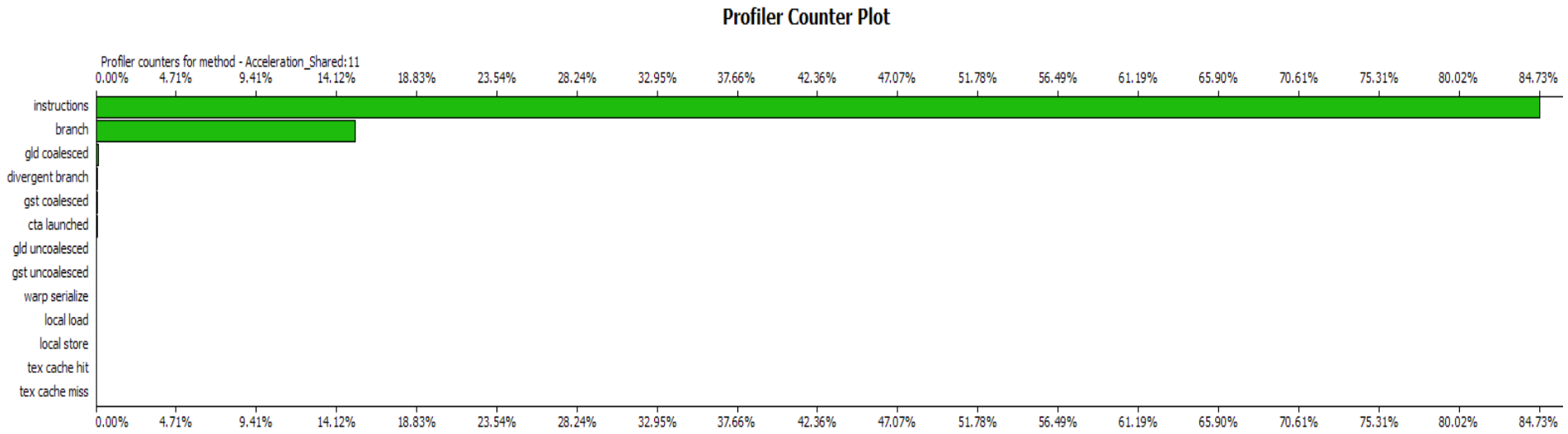


Задача N-тел. Вариант 2

```
__global__ void Acceleration_Shared ( float *X, float *Y, float *AX, float *AY, int N,
                                      int N_block )
{int id = threadIdx.x + blockIdx.x*blockDim.x;
 float ax = 0.f;  float ay = 0.f;
 float xx, yy, rr;
 float xxx = X [ id ];  float yyy = Y [ id ];
 __shared__ float Xs [ 256 ];  __shared__ float Ys [ 256 ];

 for ( int i=0; i< N_block; i++)
 { Xs [ threadIdx.x ] = X [ threadIdx.x + i * blockDim.x ];
   Ys [ threadIdx.x ] = Y [ threadIdx.x + i * blockDim.x ];
   __syncthreads();
   for ( int j=0; j < blockDim.x; j++)
   {if ( ( j + i * blockDim.x )!= id )
    {xx = Xs [ j ] - xxx; yy = Ys [ j ] - yyy; rr = sqrtf ( xx * xx + yy * yy );
        if ( rr > 0.01f ) { rr = 10.f / (rr * rr * rr ); ax + = xx * rr; ay + = yy * rr; }
    }
   } //j
   __syncthreads();
 } //i
 AX [ id ] = ax; AY [ id ] = ay;
}
```

Задача N-тел. Вариант 2

Kernel Acceleration_Shared

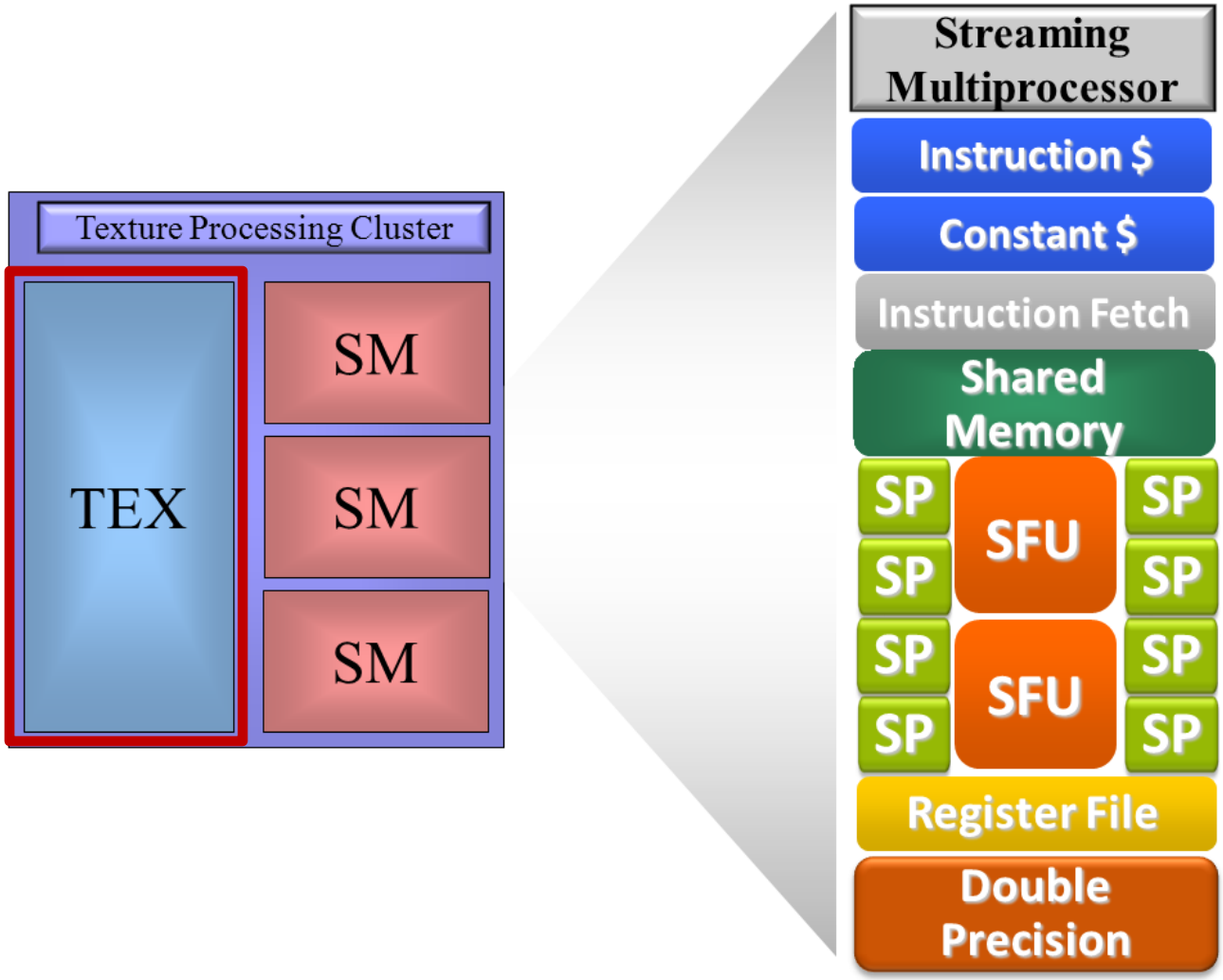


84% инструкции вместо 20% в 1 варианте.

Типы памяти в CUDA

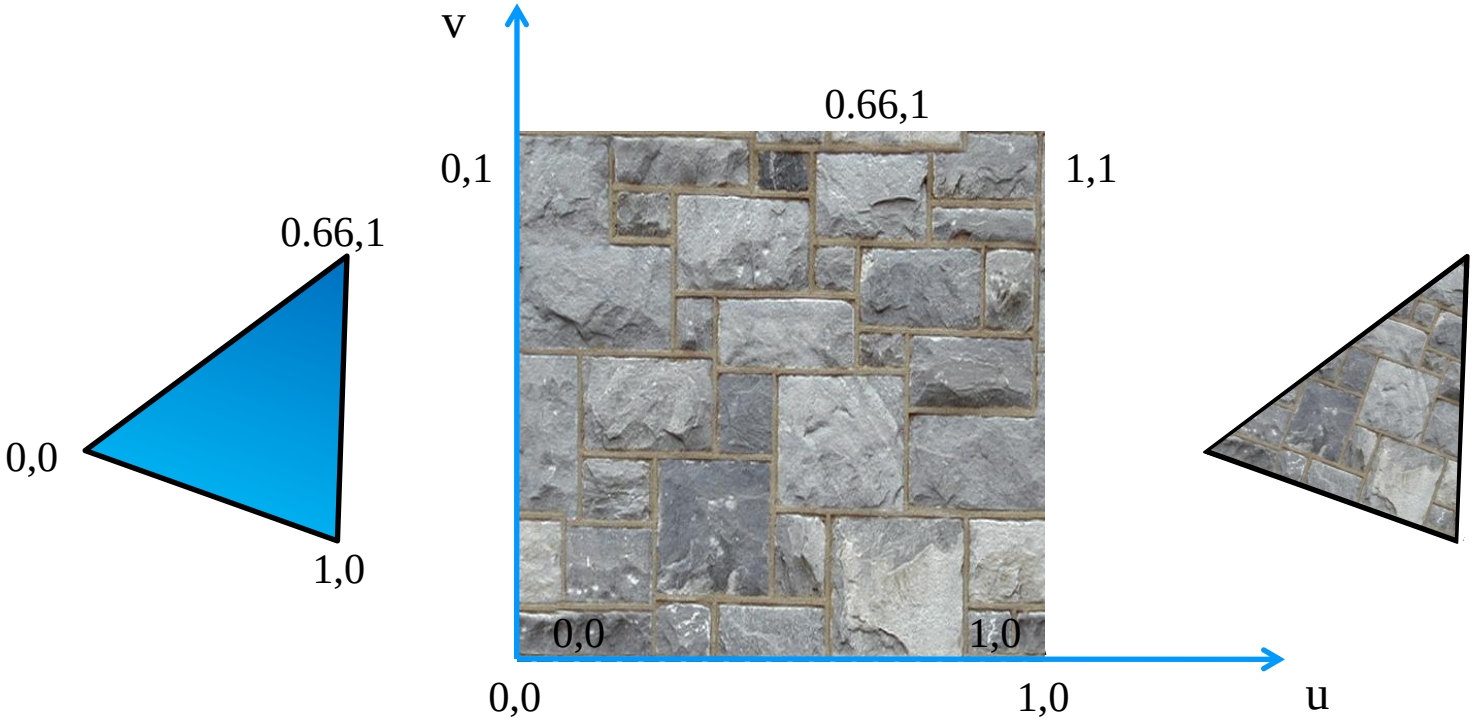
Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	R/W	Per-grid	Низкая (DRAM)
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	**R/O**	**Per-grid**	**[-]Низкая(DRAM)** **[+]L1 cache**

Мультипроцессор Tesla 10



Texture в 3D

- **В CUDA есть доступ к fixed-function HW: Texture Unit**



Texture HW

- **Латентность больше, чем у прямого обращения в память**
  - Дополнительные стадии в конвеере:
    - Преобразование адресов
    - Фильтрация
    - Преобразование данных
- **Но зато есть кэш**
  - Разумно использовать, если:
    - Объем данных не влезает в shared память
    - Паттерн доступа хаотичный
    - Данные переиспользуются разными потоками

Texture в CUDA (cudaArray)

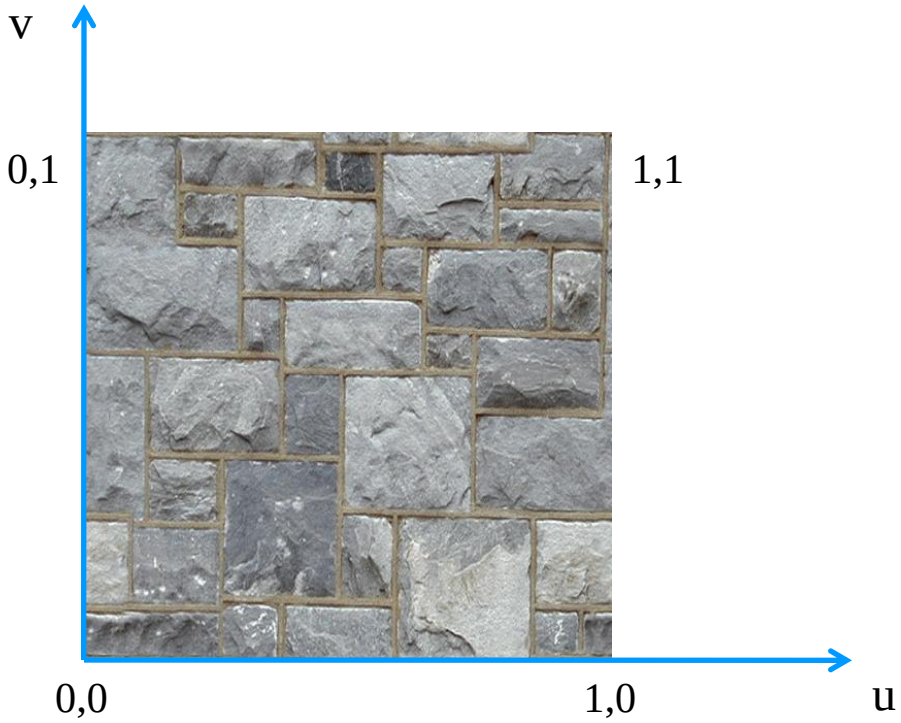
- **Особый контейнер памяти: cudaArray**
- **Черный ящик для приложения**
- **Позволяет организовывать данные в 1D/ 2D/3D массивы данных вида:**
  - 1/2/4 компонентные векторы
  - 8/16/32 bit signed/unsigned integers
  - 32 bit float
  - 16 bit float (driver API)
- **Доступ по семейству функций tex1D()/tex2D()/tex3D()**

Texture в CUDA (cudaArray)

- **Особенности текстур:**
  - **Обращение к 1D / 2D / 3D массивам данных по:**
    - Целочисленным индексам
    - Нормализованным координатам
  - **Преобразование адресов на границах**
    - Clamp
    - Wrap
  - **Фильтрация данных**
    - Point
    - Linear
  - **Преобразование данных**
    - Данные могут храниться в формате **uchar4**
    - Возвращаемое значение – **float4**

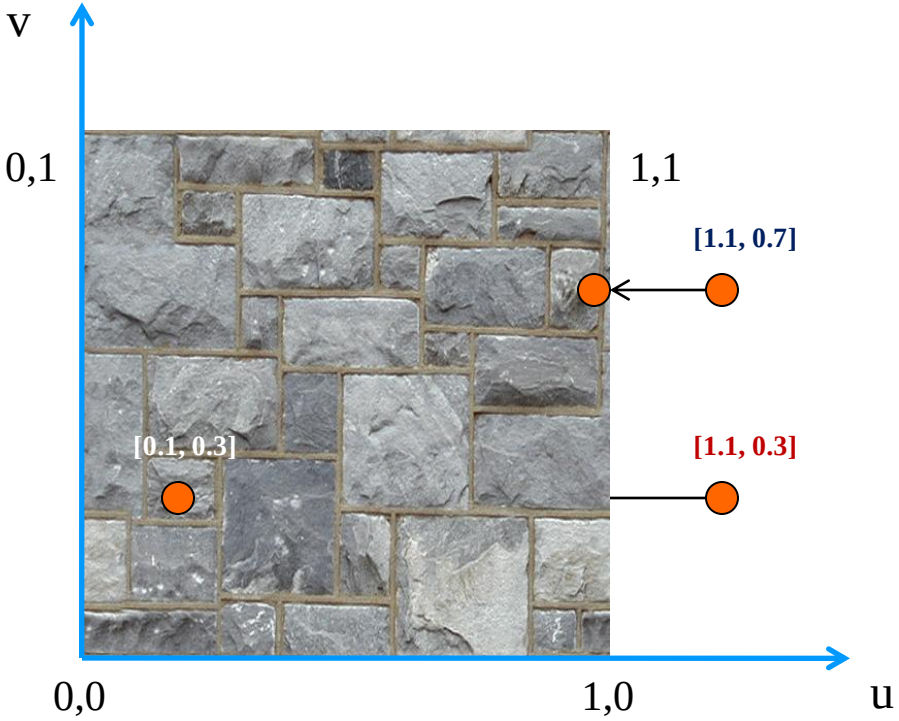
Texture в CUDA

- **Нормализация координат:**
  - Обращение по координатам, которые лежат в диапазоне [0,1]



Texture в CUDA

- **Преобразование координат:**
  - Координаты, которые не лежат в диапазоне [0,1] (или [w, h])



Clamp

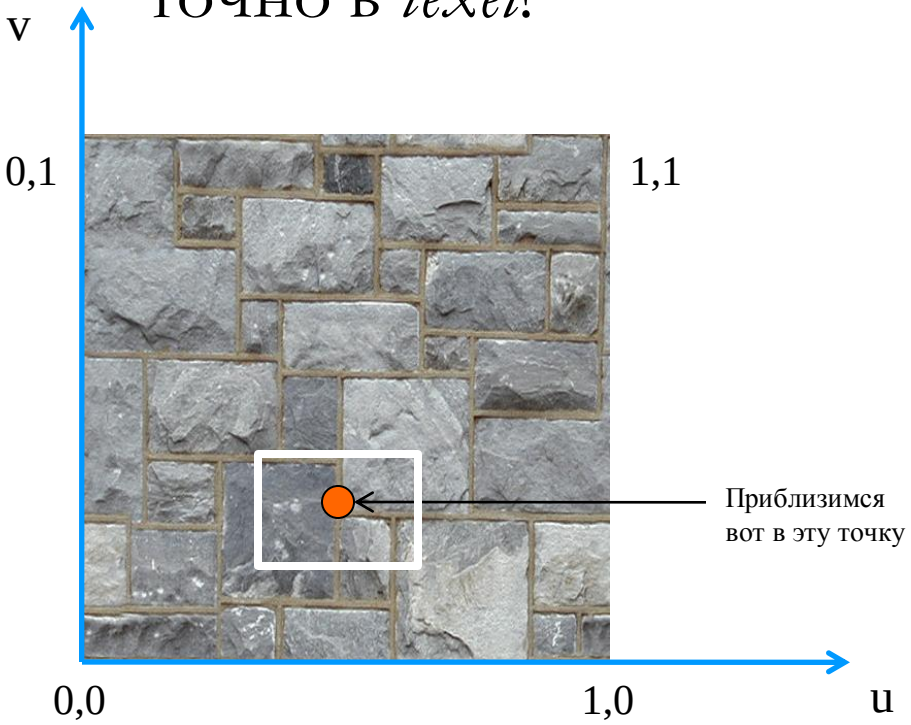
-Координата «обрубается» по допустимым границам

Wrap

- Координата «заворачивается» в допустимый диапазон

Texture в CUDA

- **Фильтрация:**
  - Если вы используете float координаты, что должно произойти если координата не попадает точно в *texel*?



Point

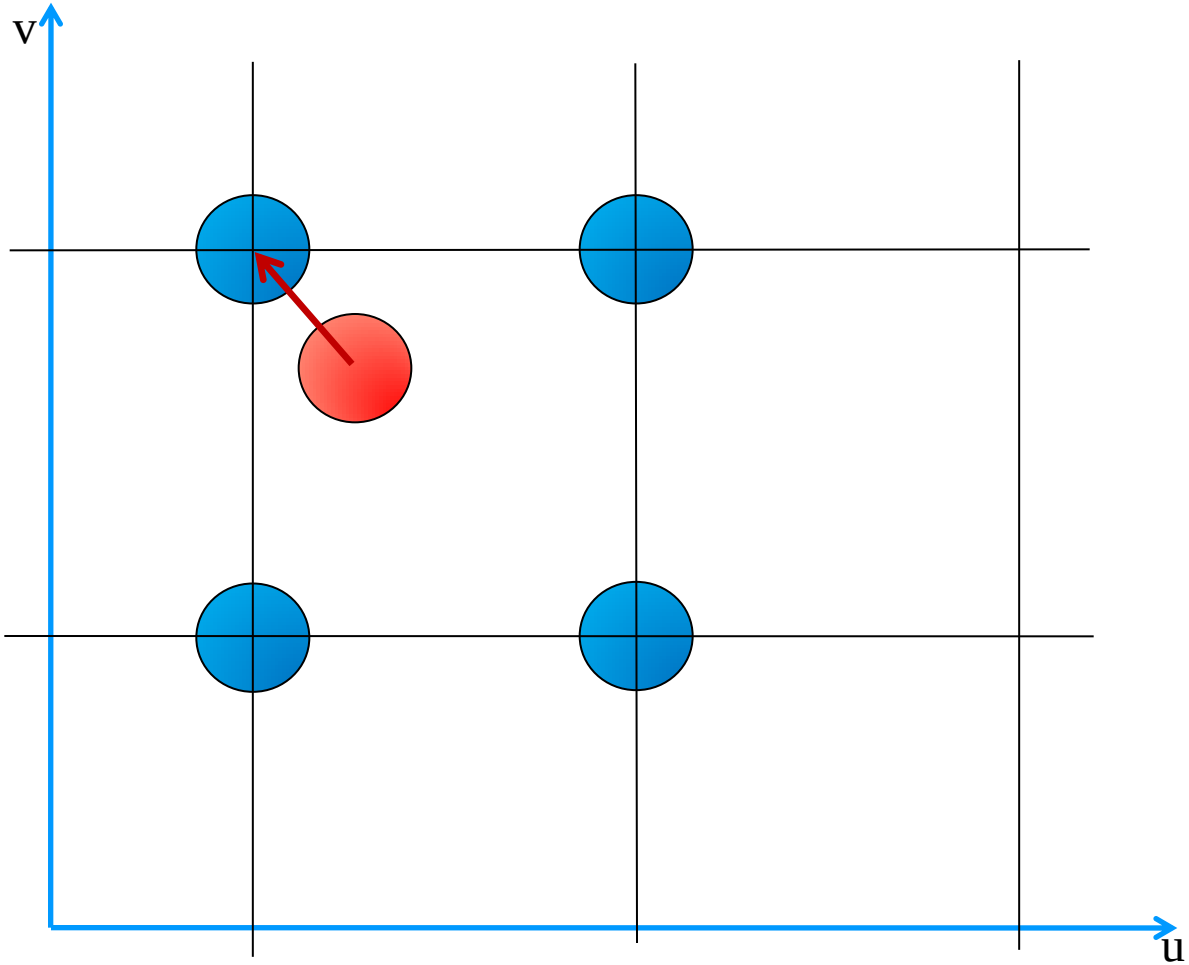
-Берется ближайший texel

Linear

- Билинейная фильтрация

Texture в CUDA

- **Фильтрация**

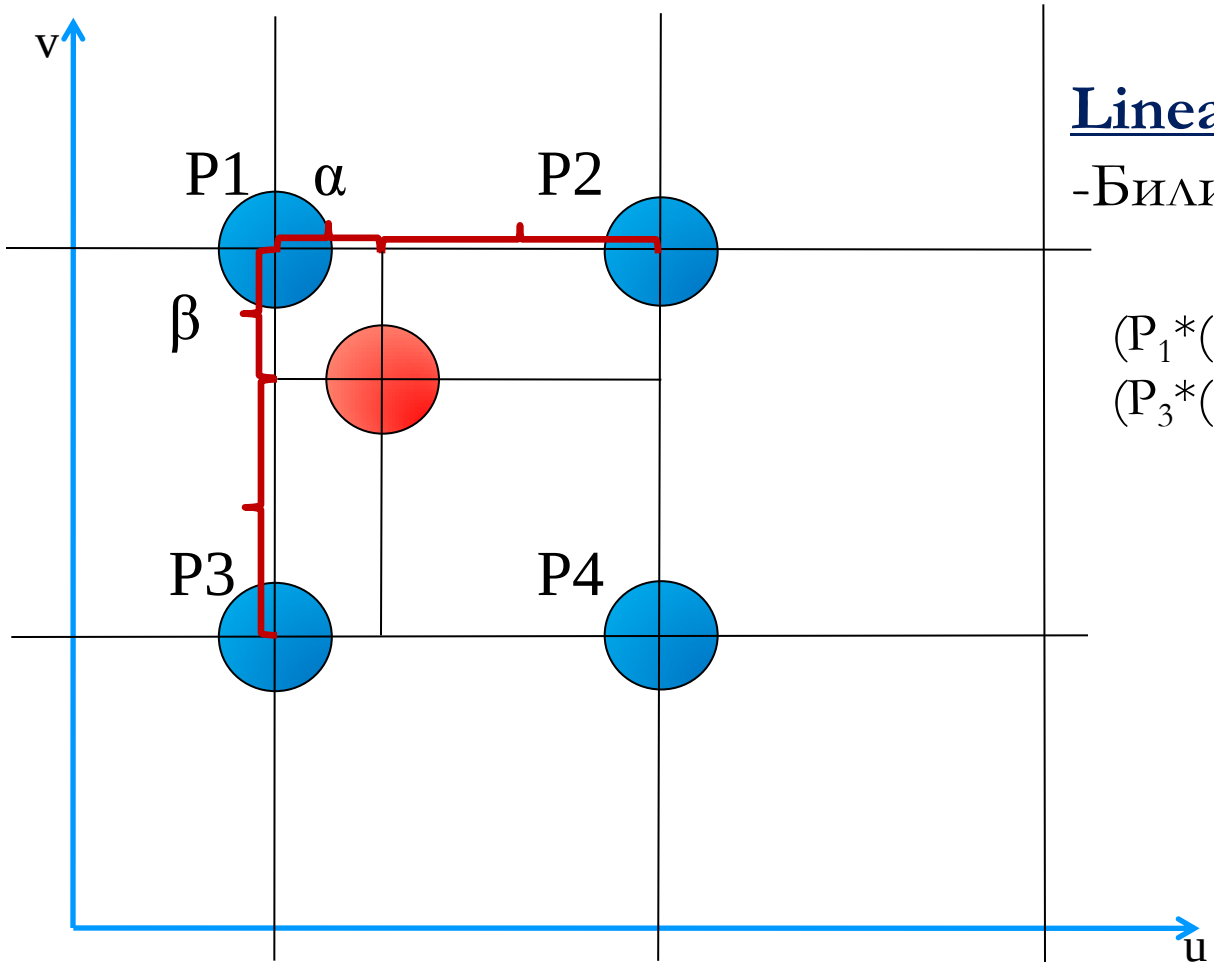


Point

-Берется ближайший texel

Texture в CUDA

- **Фильтрация**



Linear

-Билинейная фильтрация

$(P_1*(1-\alpha)+P_2*(\alpha)) * (1-\beta) +$
$(P_3*(1-\alpha)+P_4*(\alpha)) * \beta$

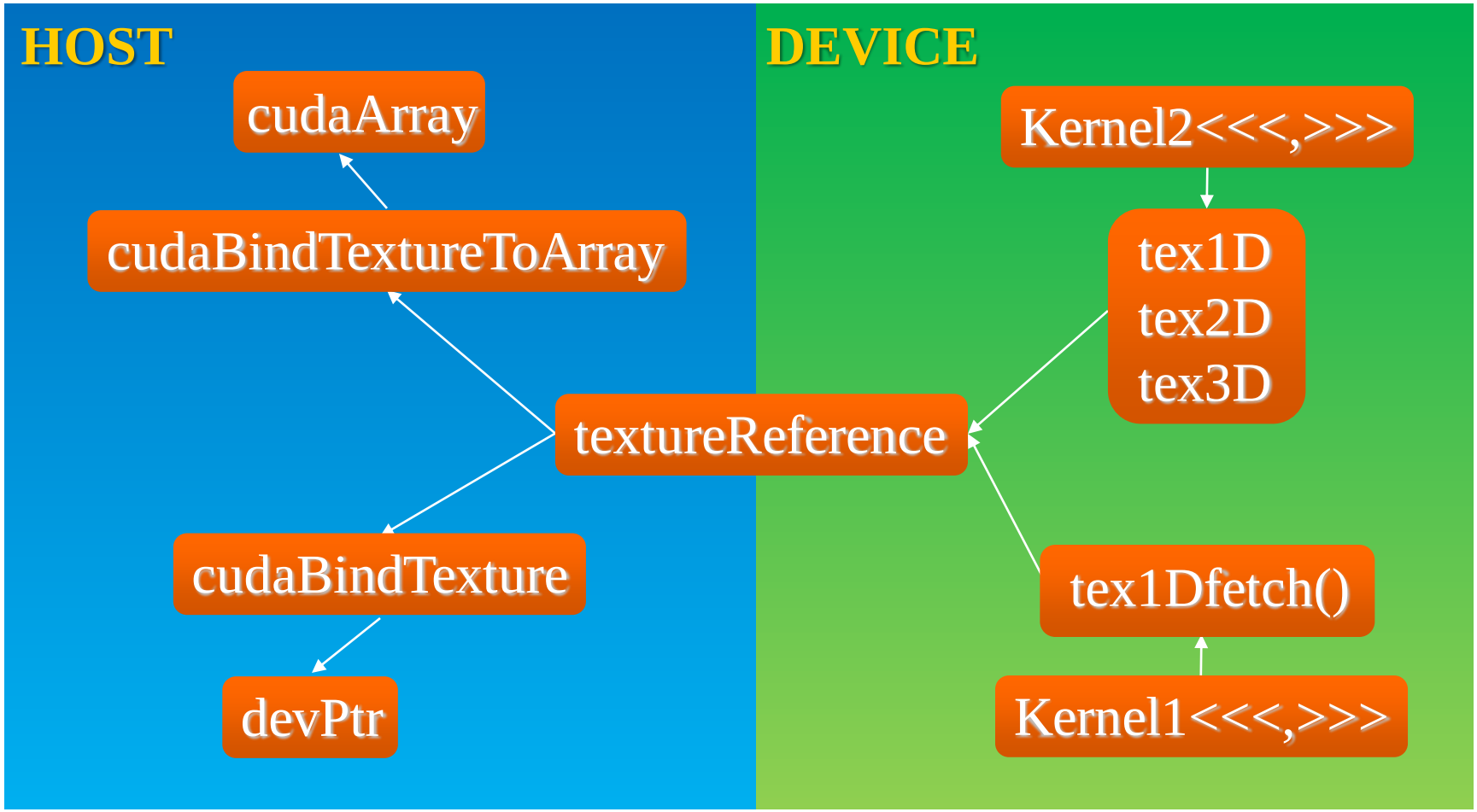
Texture в CUDA

- **Преобразование данных:**
  - **cudaReadModeNormalizedFloat :**
    - Исходный массив содержит данные в *integer*, возвращаемое значение во *floating point* представлении (доступный диапазон значений отображается в интервал [0, 1] или [-1,1])
    - Например, элемент unsigned 8-bit со значением 0xff перейдет в 1
  - **cudaReadModeElementType**
    - Возвращаемое значение то же, что и во внутреннем представлении

Texture в CUDA (linear)

- **Можно использовать обычную *линейную* память**
- **Ограничения:**
  - Только для одномерных массивов
  - Нет фильтрации
  - Доступ по целочисленным координатам
  - Обращение по адресу вне допустимого диапазона возвращает ноль

Texture в CUDA



Texture в CUDA (linear)

```
texture<float, 1, cudaReadModeElementType> g_TexRef;

__global__ void kernel1 ( float * data )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    data [idx] = tex1Dfetch(g_TexRef, idx);
}
```

Texture в CUDA (linear)

```
int main(int argc, char ** argv)
{
    float *phA = NULL, *phB = NULL, *pdA = NULL, *pdB = NULL;
    // -- memory allocation

    for (int idx = 0; idx < nThreads * nBlocks; idx++)
         phA[idx] = sinf(idx * 2.0f * PI / (nThreads * nBlocks) );

    CUDA_SAFE_CALL( cudaMemcpy ( pdA, phA, nMemSizeInBytes,
                    cudaMemcpyHostToDevice ) );

    CUDA_SAFE_CALL( cudaBindTexture(0, g_TexRef, pdA, nMemSizeInBytes) );

    dim3 threads = dim3( nThreads );
    dim3 blocks  = dim3( nBlocks );

    kernel1 <<<blocks, threads>>> ( pdB );
    CUDA_SAFE_CALL( cudaThreadSynchronize() );

    CUDA_SAFE_CALL( cudaMemcpy ( phB, pdB, nMemSizeInBytes,
                    cudaMemcpyDeviceToHost ) );

    // -- results check & memory release
    return 0;
}
```

Texture в CUDA (cudaArray)

```
texture<float, 2, cudaReadModeElementType> g_TexRef;

__global__ void kernel ( float * data )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    data [idx + blockIdx.y * gridDim.x * blockDim.x] =
                    tex2D(g_TexRef, idx, blockIdx.y);
}
```

Texture в CUDA (cudaArray)

```
int main ( int argc, char *  argv [] )
{
    float *phA = NULL, *phB = NULL, *pdB = NULL;      // linear memory pointers
    cudaArray * paA = NULL;                                  // device  cudaArray pointer
    // -- memory allocation

    cudaChannelFormatDesc cfDesc = cudaCreateChannelDesc(32, 0, 0, 0,
                                                         cudaChannelFormatKindFloat);
    cudaMallocArray(&paA, &cfDesc, nBlocksX * nThreads, nBlocksY);

    for (int idx = 0; idx < nThreads * nBlocksX; idx++) {
        phA[idx]                          = sinf(idx * 2.0f * PI / (nThreads * nBlocksX) );
        phA[idx + nThreads * nBlocksX] = cosf(idx * 2.0f * PI / (nThreads * nBlocksX) ); }

    cudaMemcpyToArray ( paA, 0, 0, phA, nMemSizeInBytes, cudaMemcpyHostToDevice );
    cudaBindTextureToArray(g_TexRef, paA);

    dim3 threads = dim3( nThreads );
    dim3 blocks  = dim3( nBlocksX, nBlocksY );

    kernel<<<blocks, threads>>> ( pdB );
    cudaThreadSynchronize();

    cudaMemcpy ( phB, pdB, nMemSizeInBytes, cudaMemcpyDeviceToHost );

    // -- results check & memory release
    return 0;
}
```

2D Интерполяция

```
// 2D float texture
texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;
// Simple transformation kernel
__global__ void transformKernel(float* output, int width, int height,
float theta)
{
// Calculate normalized texture coordinates
unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
float u = x / (float)width;
float v = y / (float)height;
// Transform coordinates
u -= 0.5f;
v -= 0.5f;
float tu = u * cosf(theta) – v * sinf(theta) + 0.5f;
float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;
// Read from texture and write to global memory
output[y * width + x] = tex2D(texRef, tu, tv);
}
```

2D Интерполяция

```
int main()
{
// Allocate CUDA array in device memory
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
// Copy to device memory some data located at address h_data
// in host memory
cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);
// Set texture parameters
texRef.addressMode[0] = cudaAddressModeWrap;
texRef.addressMode[1] = cudaAddressModeWrap;
texRef.filterMode = cudaFilterModeLinear;
texRef.normalized = true;
// Bind the array to the texture reference
cudaBindTextureToArray(texRef, cuArray, channelDesc);
// Allocate result of transformation in device memory
float* output;
cudaMalloc(&output, width * height * sizeof(float));
// Invoke kernel
dim3 dimBlock(16, 16);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x, (height + dimBlock.y - 1) / dimBlock.y);
transformKernel<<<dimGrid, dimBlock>>>(output, width, height, angle);
// Free device memory
cudaFreeArray(cuArray);
cudaFree(output);
}
```

Решение СЛАУ. Вариант 3

```
texture<float, 1, cudaReadModeElementType> ARef;
texture<float, 1, cudaReadModeElementType> FRef;
texture<float, 1, cudaReadModeElementType> X0Ref;
texture<float, 1, cudaReadModeElementType> X1Ref;

__global__ void Solve ( float *dA, float *dF, float *dX0, float
   *dX1, int N)
{
  float sum = 0.0f;
  float aa, AA;
  int t   = blockIdx.x * blockDim.x + threadIdx.x;
  int j;
  float x0 = tex1Dfetch( X0Ref, t );
  for ( j = 0; j < N; j++ ) {AA = tex1Dfetch( ARef, t+j*N );
                                  sum+=AA*tex1Dfetch( X0Ref, j );
                            if( j == t ) aa = AA;
                    }
  dX1[t] = x0 + ( tex1Dfetch( FRef, t) - sum ) / aa;
}
```