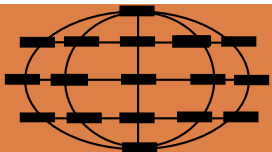


Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Новосибирский национальный исследовательский государственный
университет»
Факультет информационных технологий

Программная архитектура CUDA

Новосибирск, 2013



Константин Калгин

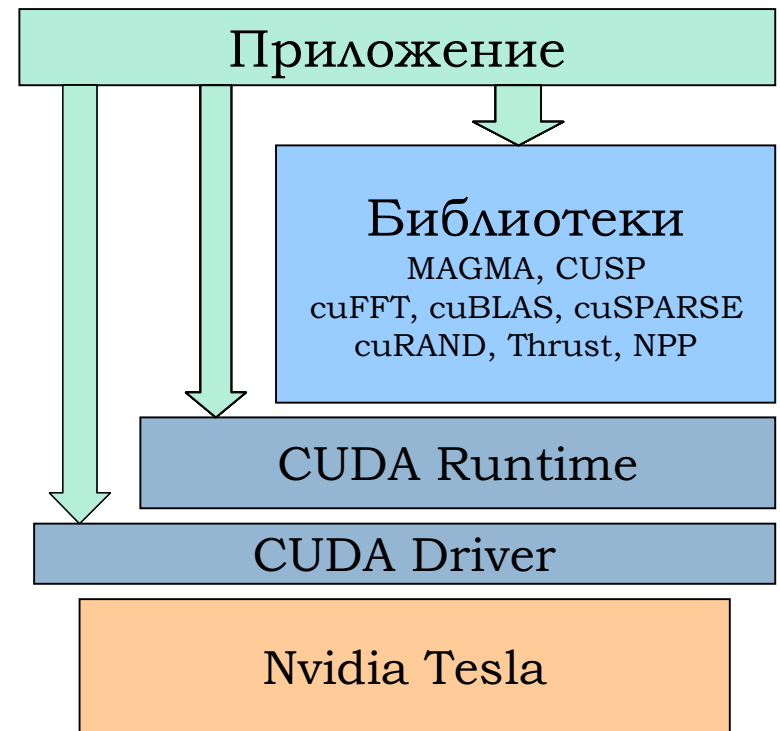
kalgin@ssd.sccc.ru

Использование Nvidia Tesla

CUDA – **C**ompute **U**nified **D**evice **A**rchitecture

Программно-аппаратная архитектура для GPU

- Расширение языка C/C++
- Компилятор nvcc
- Компиляция GPU-функций run-time
- Профилировщик
- Отладчик
- Поддерживаются: Windows Linux MacOS



Компилятор NVCC

```
__global__ void device_sum( int *a, int *b){  
    int id = threadIdx.x + blockIdx.x * blockDim.x  
    a[ i ] += b[ i ];  
}
```

```
void host_sum(int *a, int *b){  
    for( int i=0; i<N; i++) a[i] += b[ i ];  
}
```

```
int main(){
```

```
    init();
```

```
    sum<<<1024,256>>>device_sum(dev_a,dev_b);
```

```
    sum( a,b);
```

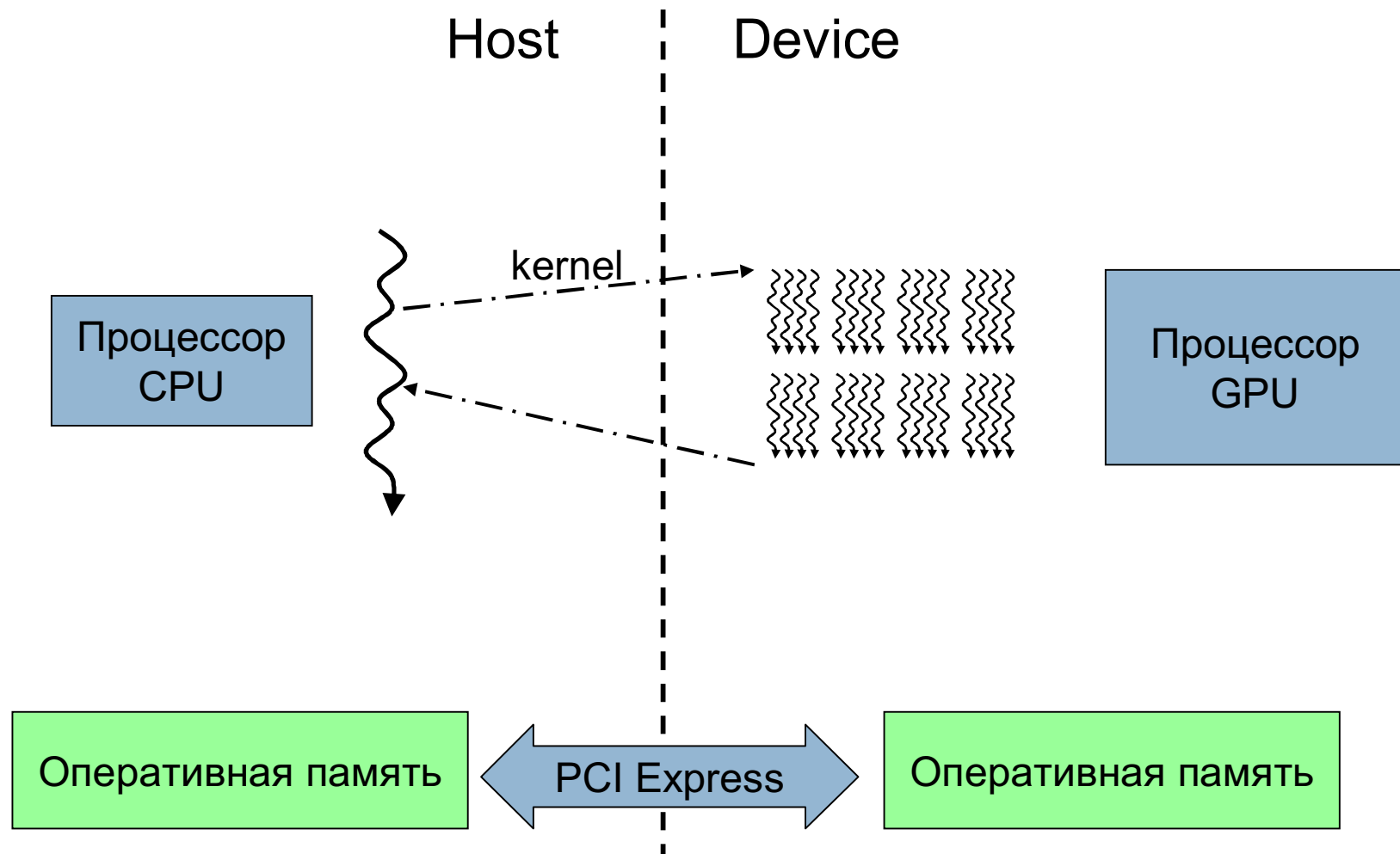
```
}
```

GCC
GNU C Compiler

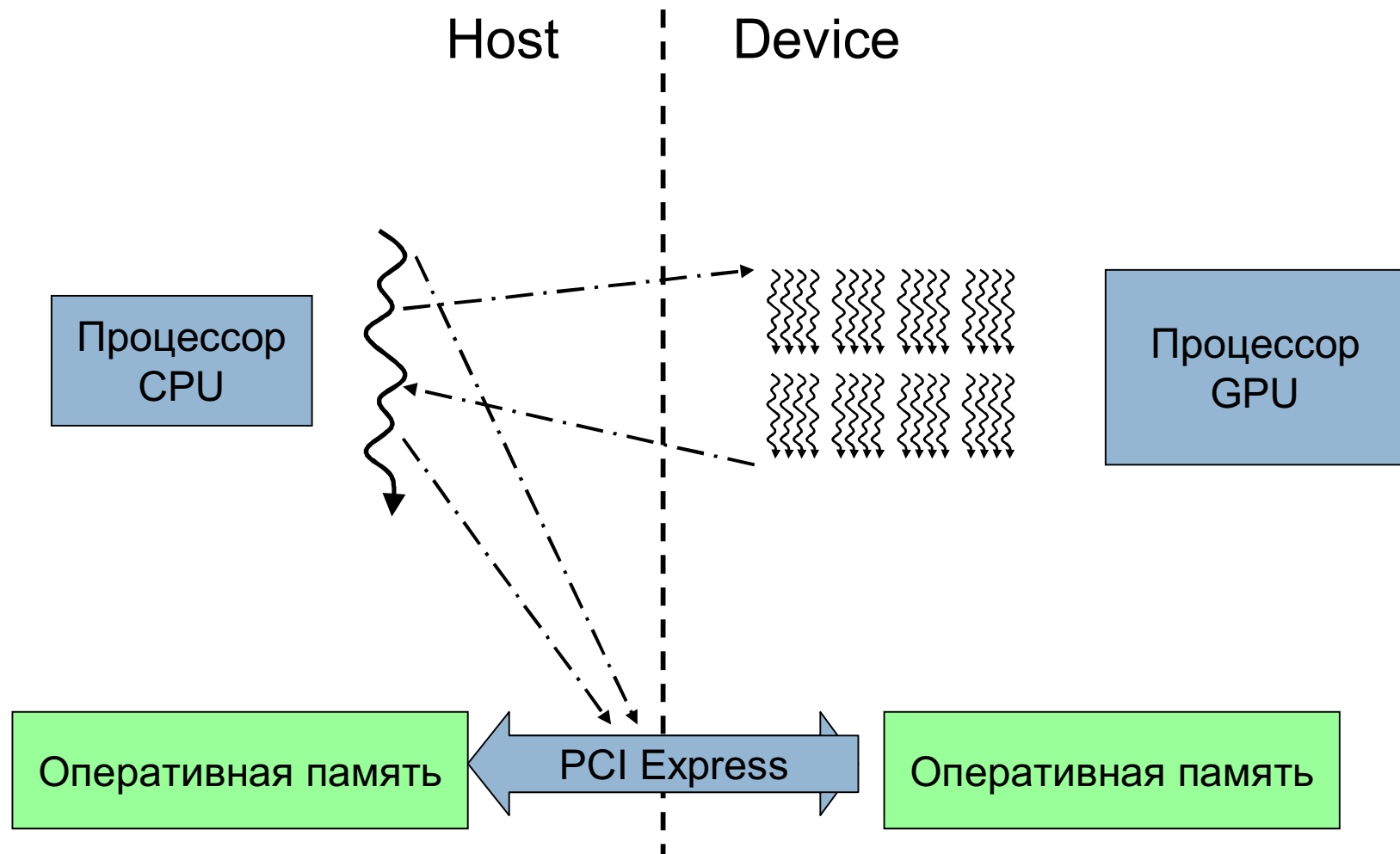
NVCC
Nvidia CUDA Compiler

Программа

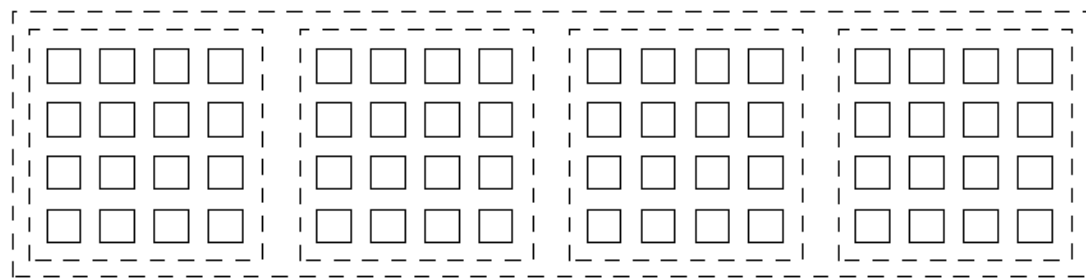
Исполнение программы



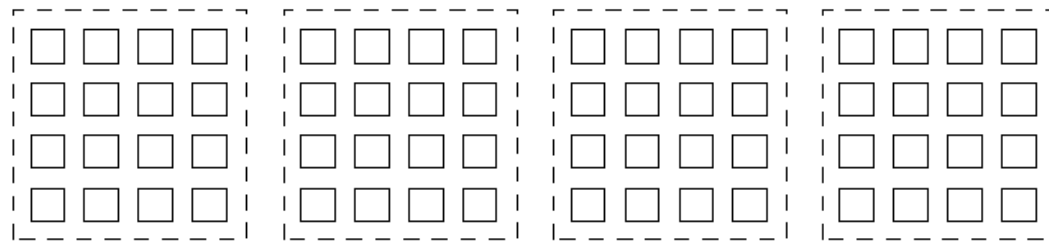
Исполнение программы



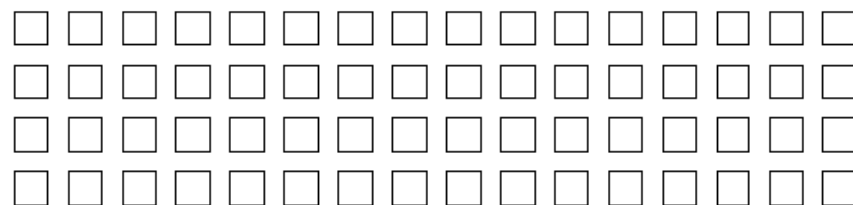
Иерархия потоков



Массив блоков потоков



Блоки потоков



Потоки

Иерархия потоков



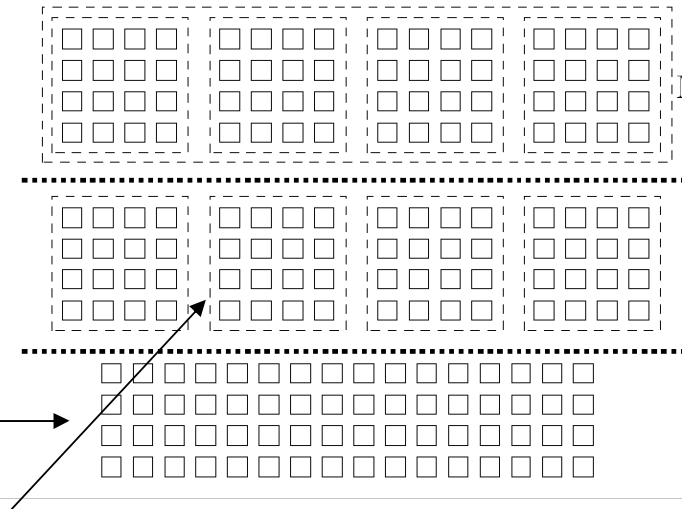
При запуске программы на графическом ускорителе порождается **множество потоков**.

Все потоки поделены на группы одного размера -- **блоки потоков**. Максимальный размер блока потоков на современных графических ускорителях равен 1024.

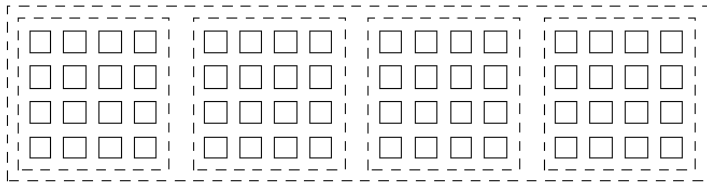
У каждого потока и блока потоков имеются свои уникальные идентификаторы, называемые координатами.

Тем самым, для разных потоков аргументы исполняемых инструкций и их последовательность могут различаться, поскольку могут зависеть от координат потока и блока потоков.

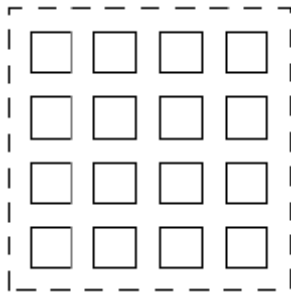
Множества координат потоков и блоков потоков образуют **одно-, дву- или трехмерные массивы**. Размеры блока потоков и массива блоков потоков задаются при запуске ядра.



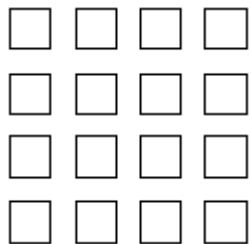
Иерархия потоков



Сетка < 65536 блоков
+ порождается при запуске ядра




Блок потоков < 1024 потока
+ разделяемая память < 48Kb
+ барьерные синхронизации
+ координаты блока потоков blockIdx



Поток
+ регистры
+ координаты потока threadIdx

Варп = 32 потока
+ согласованный доступ в память
+ синхронное исполнение инструкций

- 
- Во время исполнения ядра потоки одного блока могут синхронизироваться между собой посредством **барьеров**, а **потоки разных блоков исполняются независимо.**
 - Кроме возможности барьерной синхронизации потоки одного блока могут взаимодействовать посредством **разделяемой памяти.**
 - Потоки разных блоков могут взаимодействовать лишь через **глобальную память**, аналог оперативной памяти в компьютере.
 - Кроме разделяемой и глобальной есть **константная и текстурная памяти.**
 - На аппаратном уровне глобальная, константная, текстурная и разделяемая памяти оптимизированы под соответствующие варианты использования.

Область видимости с GPU

Область видимости и возможные операции с различными типами памяти для потоков на графическом ускорителе:

Тип памяти	Область видимости	Возможные операции
Регистры	поток	чтение/запись
Разделяемая	блок потоков	чтение/запись
Глобальная	все потоки	чтение/запись
Константная	все потоки	чтение
Текстурная	все потоки	чтение

Область видимости с CPU

Видимость и возможные операции с различными типами памяти для программы, исполняемой на центральном процессоре:

Тип памяти	Видимость	Возможные операции
Регистры	—	—
Разделяемая	—	—
Глобальная	+	чтение/запись
Константная	+	чтение/запись
Текстурная	+	чтение/запись

Координаты потока и размер сетки

Переменные, доступные каждому потоку во время исполнения ядра, содержащие соответствующие координаты и размеры

<code>threadIdx</code>	– координаты потока в блоке потоков
<code>blockIdx</code>	– координаты блока потоков в сетке
<code>blockDim</code>	– размеры блока потоков
<code>gridDim</code>	– размеры сетки блоков потоков

Доп. типы данных



Структуры данных

```
struct int4{ int x, y, z, w; }
```

char1, char2, char3, char4

int1, int2, int3, int4

float1, float2, float3, float4

uint1, uint2, uint3, uint4

Конструкторы :

```
make_int4( a, b, 2.0, a+b*2 );
```

Добавлены для эффективной организации массивов
(с выравниванием)

Пример: сложение векторов



- Выбрать Device
- Выделить память на Device
- Скопировать данные с Host на Device
- Запустить ядро (kernel) на Device
- Дождаться завершения работы ядра
- Скопировать данные с Device на Host

Пример: сумма векторов

- **Выбрать Device**

```
cudaSetDevice(0);
```

- **Выделить память на Device**

```
int N = 1024;
```

```
int *a, *b;
```

```
int *dev_a, *dev_b;
```

```
cudaMalloc( &dev_a, sizeof(int)*N);
```

```
cudaMalloc( &dev_b, sizeof(int)*N);
```

- **Скопировать данные с Host на Device**

```
a = (int*)malloc( sizeof(int) * N );
```

```
b = (int*)malloc( sizeof(int) * N );
```

```
for( int i=0; i<N; i++){ a[ i ] = i; b[ i ] = N-i; }
```

```
cudaMemcpy( dev_a, a, sizeof(int)*N, cudaMemcpyHostToDevice );
```

```
cudaMemcpy( dev_b, b, sizeof(int)*N, cudaMemcpyHostToDevice );
```

Пример: сумма векторов

- Запустить ядро (kernel) на Device

```
__global__ void sum( int * a, int * b, int N ){  
    int i = threadIdx.x + blockIdx.x * blockDim.x ;  
    if( i >= N ) return;  
    a[ i ] += b[ i ];  
}  
  
__main__ : sum<<< (N+255)/256, 256 >>>( dev_a, dev_b, N );
```

- Дождаться завершения работы ядра

```
cudaDeviceSynchronize();
```

- Скопировать данные с Device на Host

```
cudaMemcpy( a, dev_a, sizeof(int)*N, cudaMemcpyDeviceToHost );
```


Пример: сумма векторов

```
#include <cuda.h>

__global__ void sum( int * a, int * b, int N ){
    int i = threadIdx.x + blockIdx.x * blockDim.x ;
    if( i >= N ) return;
    a[ i ] += b[ i ];
}

int main(){
    cudaSetDevice(0);
    int N = 1024;
    int *a, *b;    int *dev_a, *dev_b;
    cudaMalloc( &dev_a, sizeof(int)*N);    cudaMalloc( &dev_b, sizeof(int)*N);
    a = (int*)malloc( sizeof(int) * N );    b = (int*)malloc( sizeof(int) * N );
    for( int i=0; i<N; i++){ a[ i ] = i; b[ i ] = N-i; }
    cudaMemcpy( dev_a, a, sizeof(int)*N, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, sizeof(int)*N, cudaMemcpyHostToDevice );
    sum<<< (N+255)/256, 256 >>>( dev_a, dev_b, N );
    cudaDeviceSynchronize();
    cudaMemcpy( a, dev_a, sizeof(int)*N, cudaMemcpyDeviceToHost );
}
```

Сумма векторов: текстурный кэш

```
texture<float,1,cudaReadModeElementType> tex_a, tex_b;
```

Объявление 1D текстуры

```
__global__ void sum(float *c){  
    int n = blockDim.x * blockIdx.x + threadIdx.x;  
    c[n] = tex1D( tex_a, n ) + tex1D( tex_b, n );  
}
```

Обращение через tex-кэш

```
float a[N], b[N], c[N];  
int main(){  
    float *dev_a, *dev_b, *dev_c;  
    cudaSetDevice(0);  
    cudaMalloc( ..., sizeof(float)*N );  
    cudaMemcpy( ..., cudaMemcpyHostToDevice );  
    cudaBindTexture(0, tex_a, dev_a, tex_a.channelDesc, N);  
    sum<<< N/64, 64 >>>( dev_c );  
    cudaMemcpy( ..., cudaMemcpyDeviceToHost );  
}
```

«Привязывание» области памяти к текстуре

Сумма матриц: глобальная память

```
__global__ void sum(float *a, float *b, float *c){  
    int nx = blockDim.x * blockIdx.x + threadIdx.x;  
    int ny = blockDim.y * blockIdx.y + threadIdx.y;  
    c[ny*N+nx] = a[ny*N+nx] + b[ny*N+nx];  
}
```

Преобразование 2D координат
в линейный номер

```
float a[N][M], b[N][M], c[N][M];  
int main(){  
    cudaSetDevice ...  
    cudaMalloc((void**)&dev_a, sizeof(float)*N*M);  
    cudaMemcpy ...  
    dim3 gdim={N/16,M/16,1}, bdim={16,16,1};  
    sum<<< gdim, bdim >>>( dev_a, dev_b, dev_c );  
    cudaMemcpy(..., cudaMemcpyDeviceToHost );  
}
```

2D сетка блоков потоков

Сумма матрица: текстурный кэш 2D

```
texture<float,2,cudaReadModeElementType> tex_a,tex_b;  
float a[N][M], b[N][M], c[N][M];    size_t pitch;
```

Объявление 2D
текстуры

```
__global__ void sum(float *c){  
    int nx = blockDim.x * blockIdx.x + threadIdx.x;  
    int ny = blockDim.y * blockIdx.y + threadIdx.y;  
    c[ny*N+nx] = tex2D(tex_a,nx,ny )  
                + tex2D(tex_b,nx,ny);  
}
```

Обращение к «пикселю»
2D текстуры

```
cudaMallocPitch( (void**)&dev_a, &pitch, sizeof(float)*N, M );  
cudaMemcpy ...  
cudaBindTexture2D( 0, tex_a, dev_a, tex_a.channelDesc, N, M, pitch );  
sum<<< dim3(N/16,M/16,1), dim3(16,16,1) >>>( dev_c );  
cudaMemcpy( ... , cudaMemcpyDeviceToHost );
```

Выделение памяти
с выравниванием строк

Привязка

Пример: транспонирование



- Выделить память на Device
- Скопировать данные с Host на Device
- Запустить ядро (kernel) на Device
 - ▣ Использовать разделяемую память!
- Дождаться завершения работы ядра
- Скопировать данные с Device на Host

Пример: транспонирование

- **Выбрать Device**

```
cudaSetDevice(0);
```

- **Выделить память на Device**

```
int N = 1024;
```

```
int *a, *b;
```

```
int *dev_a, *dev_b;
```

```
cudaMalloc( &dev_a, sizeof(int)*N*N);
```

```
cudaMalloc( &dev_b, sizeof(int)*N*N);
```

- **Скопировать данные с Host на Device**

```
a = (int*)malloc( sizeof(int) *N*N );
```

```
b = (int*)malloc( sizeof(int) *N*N );
```

```
for( int i=0; i<N*N; i++){ a[ i ] = i; b[ i ] = N-i; }
```

```
cudaMemcpy( dev_a, a, sizeof(int)*N,*N cudaMemcpyHostToDevice );
```

```
cudaMemcpy( dev_b, b, sizeof(int)*N*N, cudaMemcpyHostToDevice );
```

Пример: транспонирование

```
#define BS 16
```

```
dim3 gdim = dim3((N+BS-1)/BS, (N+BS-1)/BS, 1);
```

```
dim3 bdim = dim3(BS,BS,1);
```

```
main : sum<<< gdim, bdim >>>( dev_a, dev_b, N );
```

```
__global__ void transpose( int * a, int * b, int N ){  
    int i = threadIdx.x + blockIdx.x * blockDim.x ;  
    int j = threadIdx.y + blockIdx.y * blockDim.y ;  
    if( i >= N || j >=N) return;  
    a[ i*N + j ] = b[ j*N + i ];  
}
```

Сокращения

Далее в листингах используются следующие сокращения

<code>threadIdx</code>	<code>tid</code>
<code>blockIdx</code>	<code>bid</code>
<code>blockDim</code>	<code>bdim</code>
<code>gridDim</code>	<code>gdim</code>

В начале каждого листинга будут опущены следующие строки (см. след. слайд)

Преамбула к листингам

```
1  #include <cuda.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #define tid  threadIdx
5  #define bid  blockIdx
6  #define bdim blockDim
7  #define gdim gridDim
8  #define cudaCheck \
9  {\
10      cudaError_t err = cudaGetLastError();\
11      if ( err != cudaSuccess ){\  
12          printf("cudaError=%s\nin%s'%d\n",\  
13              cudaGetErrorString( err ),\  
14              __FILE__, __LINE__ );\  
15          exit(0);\
16      }\
17 }
```

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.