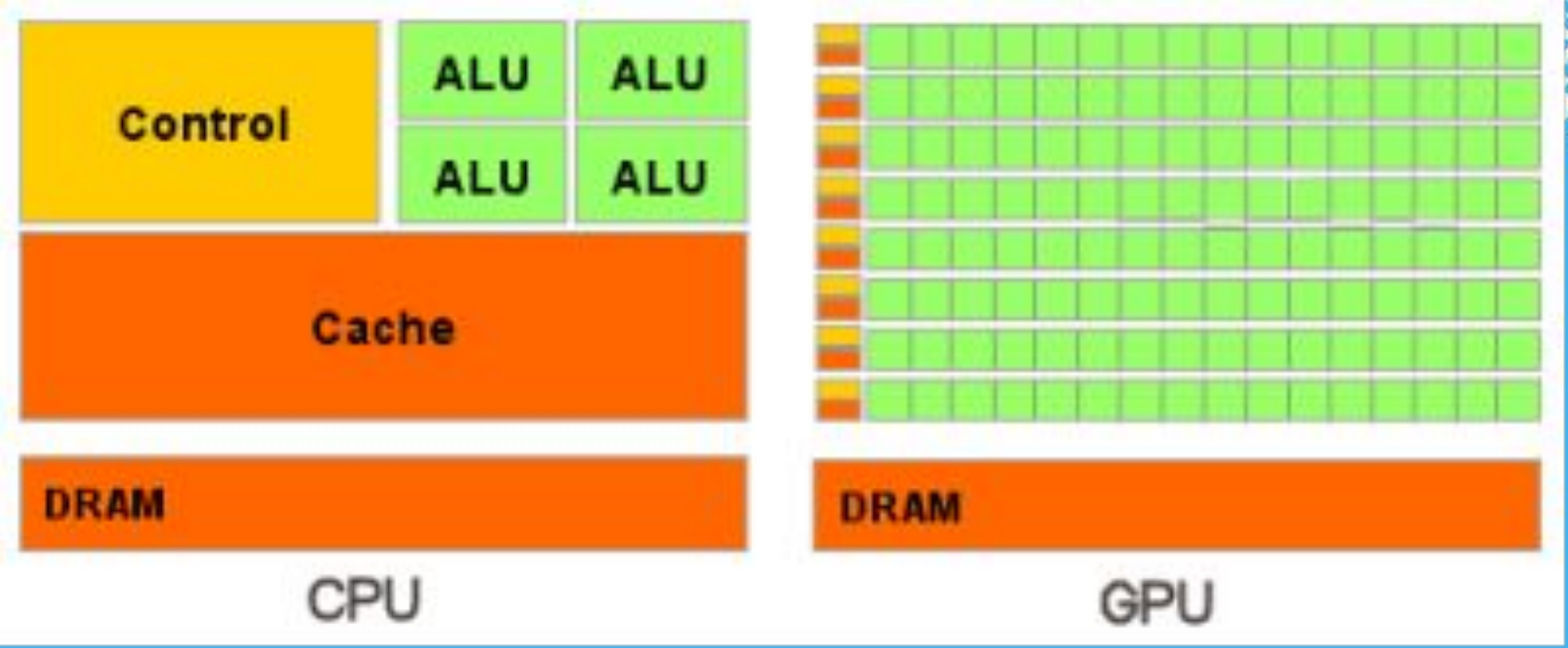
Основы технологии CUDA.

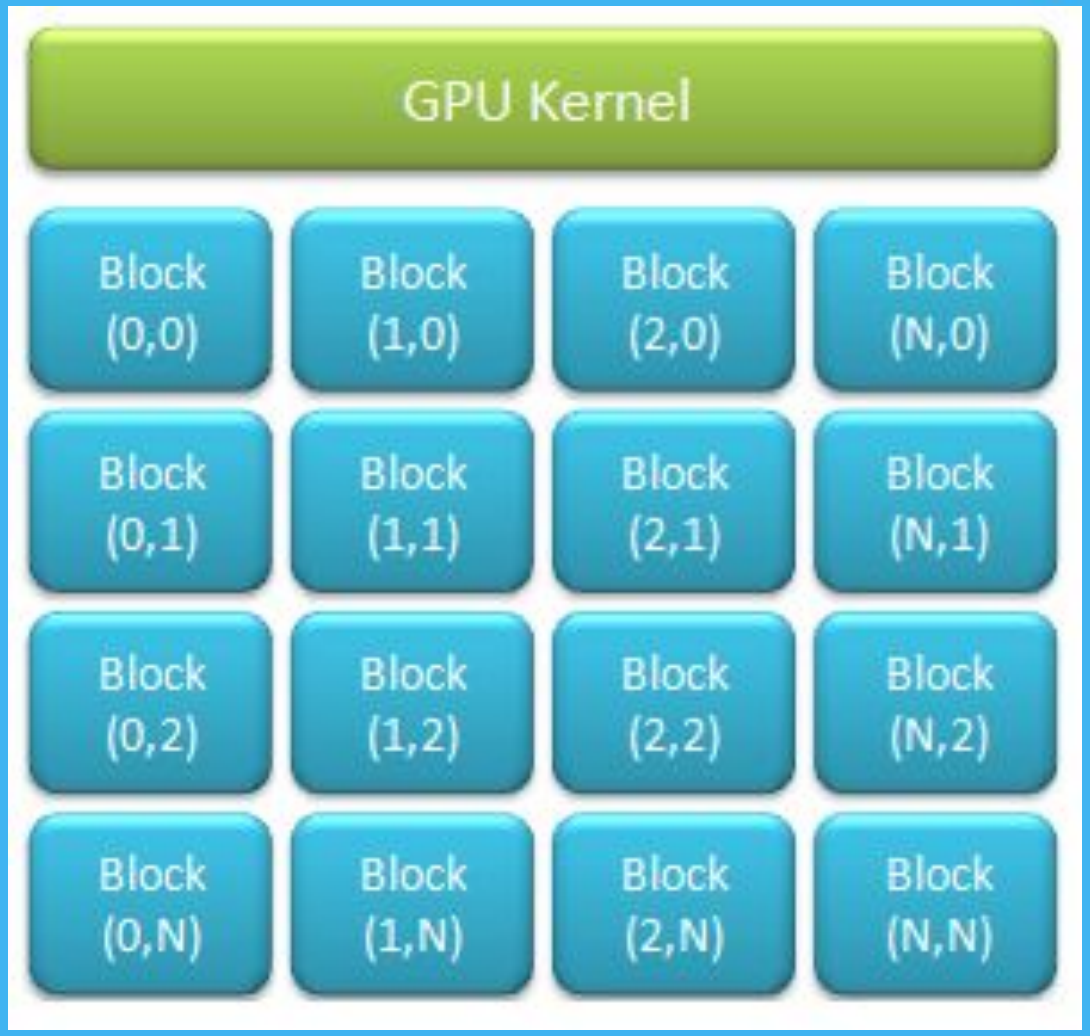
Работа с памятью

Сравнение архитектур CPU и GPU



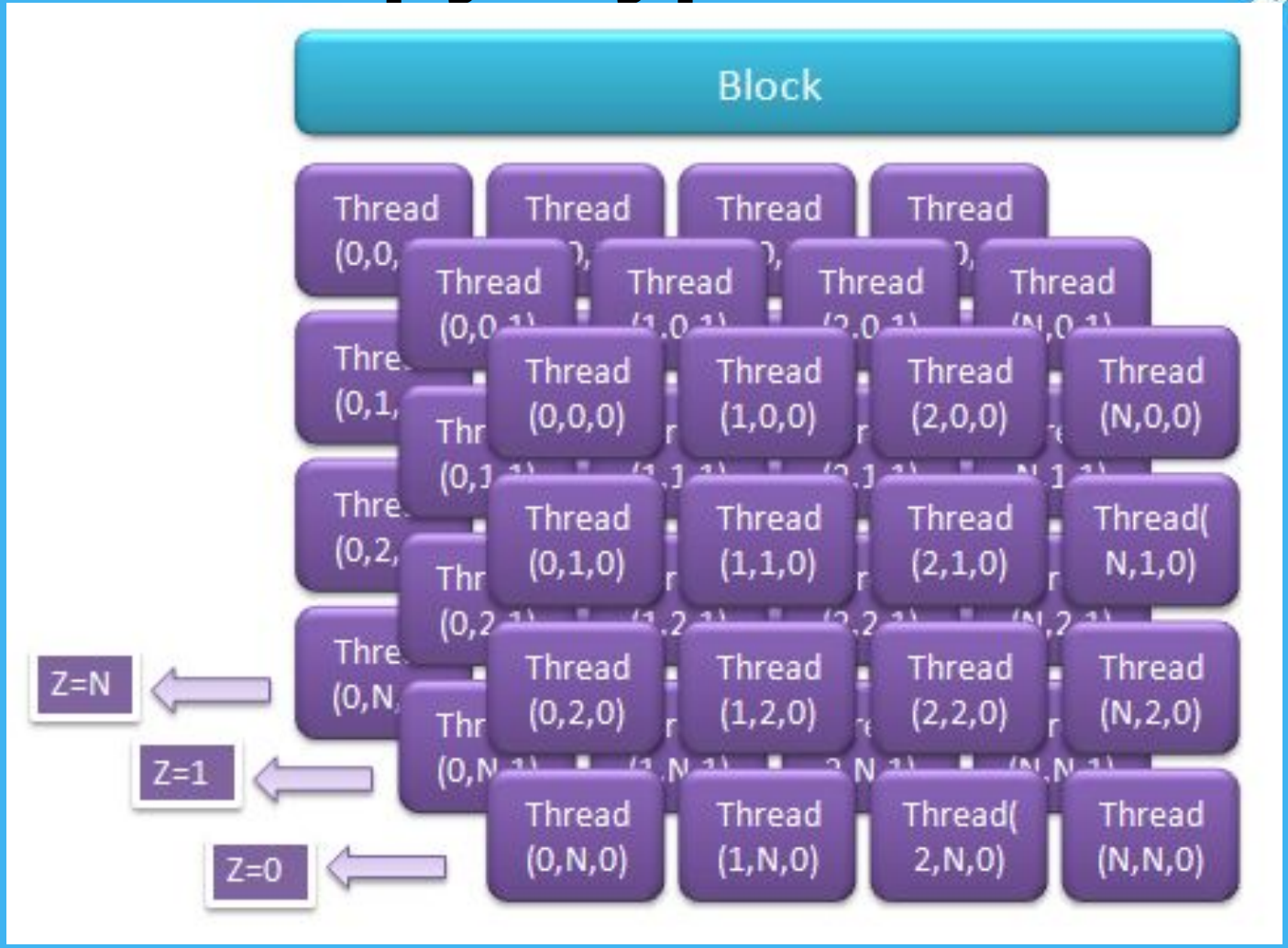
Параллельная обработка данных распределяет элементы данных на параллельно обрабатываемых потоках. GPU особенно хорошо подходит для решения проблем, которые могут быть выражены как вычислений данными параллельно - та же программа выполняется на многих элементов данных параллельно - с высокой интенсивностью - арифметическое отношение арифметических операций к операциям с памятью.

Вычислительная модель GPU



- Двумерная блочная структура

Структура блоков



Трехмерная структура блоков

Подключаемые библиотеки

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <windows.h>
#include <stdio.h>
```

Оценка затраченного на вычисления времени

```
cudaEvent_t start, stop;
float gpuTime;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );
 ...
cudaEventRecord( stop, 0 ) ;
 cudaEventSynchronize( stop ) );
cudaEventElapsedTime( &gpuTime, start, stop ) ;
printf("time spent executing by the GPU: %.2f millseconds\n",
                                    gpuTime );
cudaEventDestroy( start) ;
cudaEventDestroy( stop) ;
```

cudaEventCreate() - создание временных меток start, stop ;

cudaEventRecord() - фиксация времени старта ;

cudaEventRecord( ) - фиксация времени завершения.

cudaEventSynchronize() - синхронизация асинхронных процессов;

cudaEventElapsedTime() - вычисление разницы во времени.

О компоновке нитей и блоков

```
#define DGX 8
#define DGY 32
#define DBX 8
#define DBY 8
#define DBZ 8
#define N (DBX*DBY*DBZ*DGX*DGY)
__global__ void kern( float *a )
{ int bs = blockDim.x*blockDim.y*blockDim.z;
int idx = threadIdx.x + threadIdx.y*blockDim.x
            +  threadIdx.z* (blockDim.x*blockDim.y)
            + blockIdx.x*bs + blockIdx.y*bs*gridDim.x ;
 if(idx > N-1) return;
a[idx] -= 0.5f;
}
```

Размер блока 8x8x8, что как раз равно 512 тредов на один блок, размер грида 8x32 блока, таким образом общее количество параллельных процессов 131072=8x8x8x8x32.

При этом, адресация выделенной памяти -- линейная и сложный номер треда пересчитывается в индекс ячейки масива памяти.

Отладка программ

Функции **из CUDA runtime API** могут возвращать различные коды ошибок. Можно использовать следующий макрос для отлова ошибок:

```
#define CUDA_DEBUG
#ifdef CUDA_DEBUG
#define CUDA_CHECK_ERROR(err)
if (err != cudaSuccess) {
printf("Cuda error: %s\n", cudaGetErrorString(err));
printf("Error in file: %s, line: %i\n", __FILE__, __LINE__);
}                \
#else
#define CUDA_CHECK_ERROR(err)
#endif
```

Если определена переменная среды CUDA_DEBUG, происходит проверка кода ошибки и выводиться информация о файле и строке, где она произошла. Эту переменную можно включить при компиляции под отладку и отключить при компиляции под релиз.

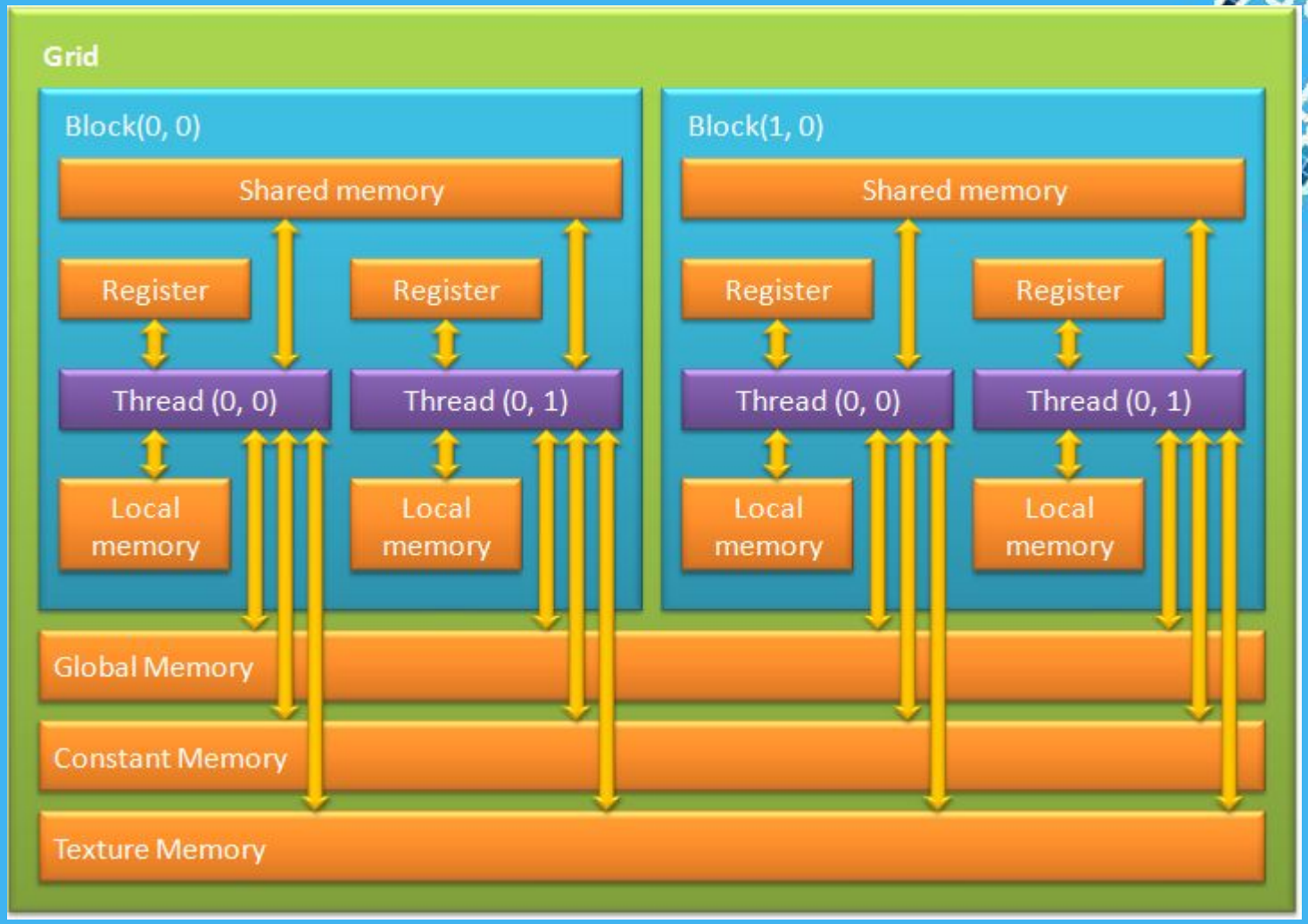
Типы памяти

Типы памяти

1. **Регистры.**
2. **Локальная память.**
3. **Глобальная память.**
4. **Разделяемая память.**
5. **Константная память.**
6. **Текстурная память**

Тип памяти	Доступ	Уровень выделения	Скорость работы
регистры	R/W	per-thread	высокая (on chip)
local	R/W	per-thread	низкая (DRAM)
shared	R/W	per-block	высокая (on-chip)
global	R/W	per-grid	низкая(DRAM)
constant	R/O	per-grid	высокая(on chip L1 cache)
texture	R/O	per-grid	высокая(on chip L1 cache)

Организация памяти устройства



Регистровая память (register)

Является **самой быстрой** из всех видов. Определить количество регистров доступных GPU можно с помощью уже функции cudaGetDeviceProperties.

Рассчитать количество регистров, доступных одной нити GPU - для этого необходимо разделить общее число регистров на произведение количества нитей в блоке и количества блоков в гриде.

Все регистры GPU 32 разрядные.

В CUDA **нет явных способов использования регистровой памяти**. Это определяет компилятор.

Расчет количества регистров, доступных одной нити GPU

Вызов функции
cudaGetDeviceProperties

```
Номер устройства: 0
Имя устройства: GeForce GT 645M
Объем глобальной памяти: 2048 Мбайт
Объем shared-памяти в блоке : 49152
Объем регистровой памяти: 65536
Размер warp'а: 32
Размер шага памяти: 2147483647
Макс количество потоков в блоке: 1024
Максимальная размерность потока: x = 1024, y = 1024, z = 64
Максимальный размер сетки: x = 2147483647, y = 65535, z = 65535
Тактовая частота: 780000 кГц
Общий объем константной памяти: 65536
Вычислительная мощность: 3.0
Величина текстурного выравнивания : 512
Количество процессоров: 2
Для продолжения нажмите любую клавишу . . .
```

При вызове функций ядра
myKernelFunc<<< **gridSize**, **blockSize**, sharedMemSize, cudaStream >>>(float* param1,float * param2),

$$N_p = \frac{65536}{\mathbf{gridSize} * \mathbf{blockSize}} = \frac{65536}{64*1} = 1024$$

$$N_p = \frac{65536}{\mathbf{gridSize} * \mathbf{blockSize}} = \frac{65536}{64*1} = 1024$$

Локальная память

Локальная память (local memory) **может быть использована компилятором** при большом количестве локальных переменных в какой-либо функции. По скоростным характеристикам локальная память **значительно медленнее**, чем регистровая. В документации от nVidia рекомендуется использовать локальную память только в самых необходимых случаях.

Явных средств, позволяющих блокировать использование локальной памяти, не предусмотрено, поэтому при падении производительности стоит тщательно проанализировать код и исключить лишние локальные переменные.

Глобальная память

Глобальная память (global memory) – самый медленный тип памяти, из доступных GPU. Глобальные переменные можно выделить с помощью спецификатора **__global__**, а так же динамически, с помощью функций из семейства **cudMallocXXX.** Глобальная память в основном служит для хранения больших объемов данных, поступивших на device с host'a, данное перемещение осуществляется с использованием функций **cudaMemcpyXXX**.

В алгоритмах, требующих высокой производительности, количество операций с глобальной памятью необходимо свести к минимуму.

Разделяемая память

Разделяемая память (shared memory) относиться **к быстрому типу памяти**. Разделяемую память рекомендуется использовать для минимизации обращение к глобальной памяти, а так же для хранения локальных переменных функций.

Адресация разделяемой памяти между нитями потока одинакова в пределах одного блока, что может быть использовано для обмена данными между потоками в пределах одного блока.

Для размещения данных в разделяемой памяти используется спецификатор **__shared__.**

Константная память

Константная память (constant memory) является **достаточно быстрой** из доступных GPU. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название.

Для размещения данных в константной памяти предусмотрен спецификатор **__constant__**. Если необходимо использовать массив в константной памяти, то его размер необходимо указать заранее, так как **динамическое выделение** в отличие от глобальной памяти в константной **не поддерживается**.

Для записи с хоста в константную память используется функция **cudaMemcpyToSymbol**, и для копирования с device'а на хост **cudaMemcpyFromSymbol**, как видно этот подход несколько отличается от подхода при работе с глобальной памятью.

Текстурная память

Предназначена главным образом для работы с текстурами. Она **оптимизирована** под выборку 2D данных и имеет следующие возможности:

- **быстрая выборка** значений фиксированного размера из одномерного или двухмерного массива;
- **нормализованная адресация** числами типа float в интервале [0,1).
- **аппаратная линейная** или билинейная интерполяция соседних значений в случае нормализованной адресации;
- **аппаратная обработка** выхода за границу массива с использованием двух режимов: clamp и wrap.

Размер текстурной памяти ограничивается только максимальным размером памяти, которую может выделить устройство. Но так же из текстурной памяти можно читать данные только встроенных в nvcc типов, имеющих размер 1, 2, 4, 8 или 16 байт, и только с помощью специальных функций — tex1D, tex2D или tex1Dfetch, tex2Dfetch. Другими словами, нельзя сделать указатель на текстурную память и переименовать его произвольным образом.

Пример использования различных типов памяти

При операции транспонирования матрицы

Транспонирование матрицы на CPU

```
// inputMatrix - указатель на исходную матрицу
// outputMatrix - указатель на матрицу результат
// width - ширина исходной матрицы (она же высота матрицы-результата)
// height - высота исходной матрицы (она же ширина матрицы-результата)
__host__ void transposeMatrixCPU(float* inputMatrix, float*
outputMatrix, int width, int height)
{
  for (int y = 0; y < height; y++)
  {
    for (int x = 0; x < width; x++)
    {
     outputMatrix[x * height + y] = inputMatrix[y * width + x];
    }
  }
}
```

Использование только глобальной памяти.

```
__global__ void transposeMatrixSlow(float* inputMatrix, float* outputMatrix,
int width, int height)
{
  int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {
    int inputIdx = xIndex + width * yIndex;
    int outputIdx = yIndex + height * xIndex;
    outputMatrix[outputIdx] = inputMatrix[inputIdx];
  }
}
```

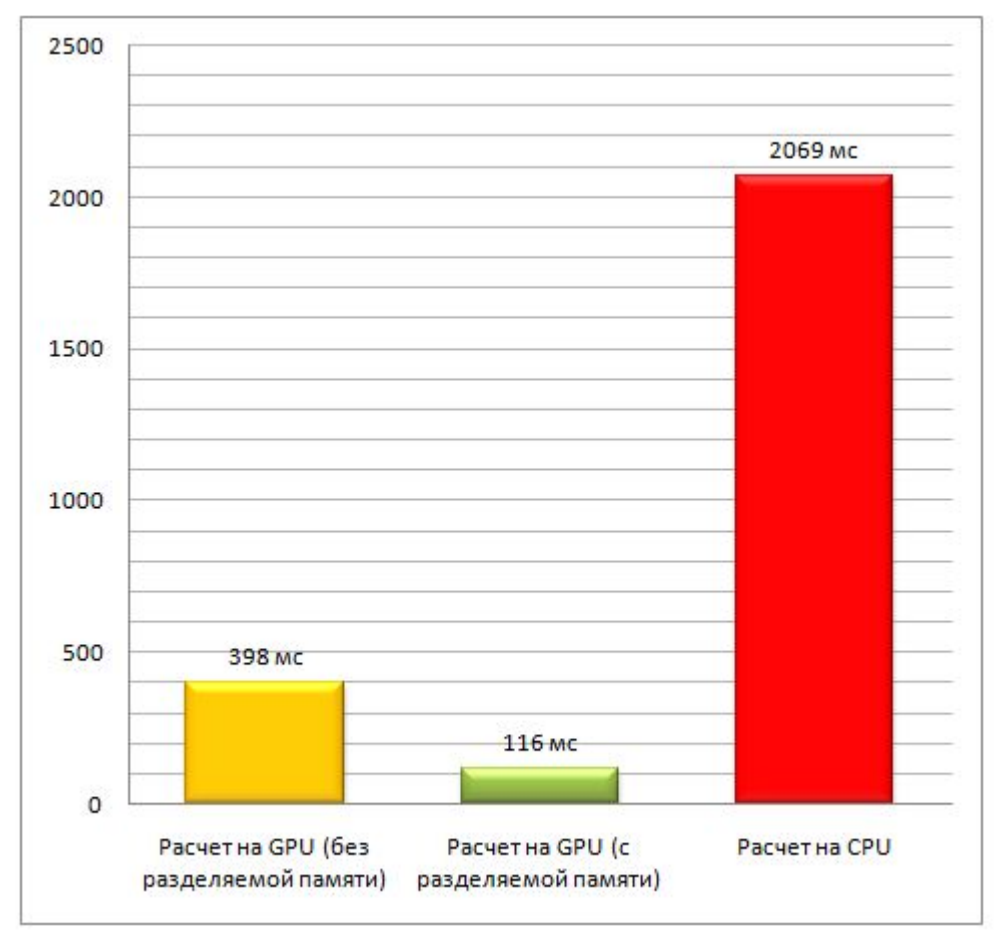
Использование константной памяти.

```
#define N 100
__constant__  float  devInputMatrix[N];
__global__ void transposeMatrixSlow(float* inputMatrix, float* outputMatrix,
int width, int height)
{   int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {     int inputIdx = xIndex + width * yIndex;
        int outputIdx = yIndex + height * xIndex;
        outputMatrix[outputIdx] = inputMatrix[inputIdx];  }  }
void host_function() {
   float  devInputMatrix[N];
    ...
cudaMemcpy(devInputMatrix, inputMatrix, byteSize,
                                    cudaMemcpyHostToDevice);
}
```

Использование разделяемой памяти

```
#define BLOCK_DIM 16
__global__ void transposeMatrixFast(float* inputMatrix, float* outputMatrix, int width, int height)
{  __shared__ float temp[BLOCK_DIM][BLOCK_DIM];
  int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {
    int idx = yIndex * width + xIndex;
   temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];
  }
  __syncthreads();
  xIndex = blockIdx.y * blockDim.y + threadIdx.x;
  yIndex = blockIdx.x * blockDim.x + threadIdx.y;
  if ((xIndex < height) && (yIndex < width))
  {
     int idx = yIndex * height + xIndex;
outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];
  }
}
```

Результаты вычислений



матрица размерностью 2048 * 1536 = 3145728 элементов и 20 итераций в нагрузочных циклах