Основы технологии CUDA

http://ru.wikipedia.org/wiki/CUDA http://docs.nvidia.com/cuda/index.html

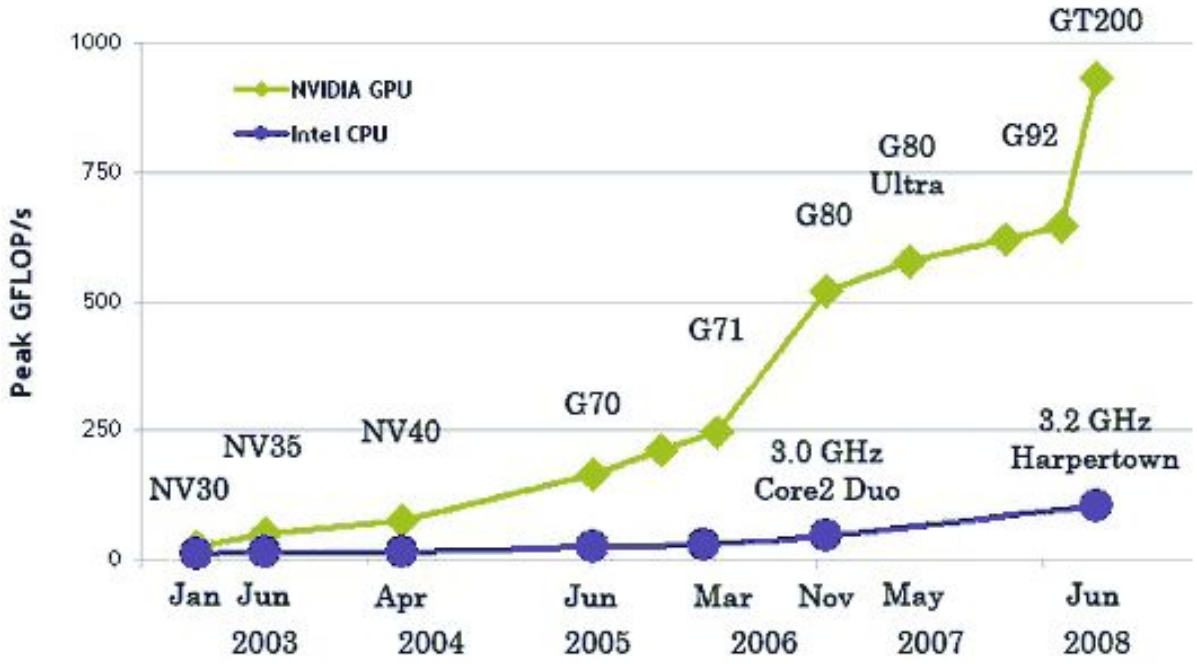
Compute Unified Device Architecture

Технология CUDA появилась в 2006 году и представляет из себя программно-аппаратный комплекс производства компании Nvidia, позволяющий эффективно писать программы под графические адаптеры. С 2006 года компания Nvidia обещает, что все графические адаптеры их производства независимо от серии будут иметь сходную архитектуру, которая полностью поддерживает программную часть технологии CUDA. Программная часть, в свою очередь, содержит в себе всё необходимое для разработки программы: расширения языка C, компилятор, API для работы с графическими адаптерами и набор библиотек.

CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования Си алгоритмы, выполнимые на графических процессорах NVIDIA.

Графический процессор организует аппаратную многопоточность, что позволяет задействовать все ресурсы графического процессора.

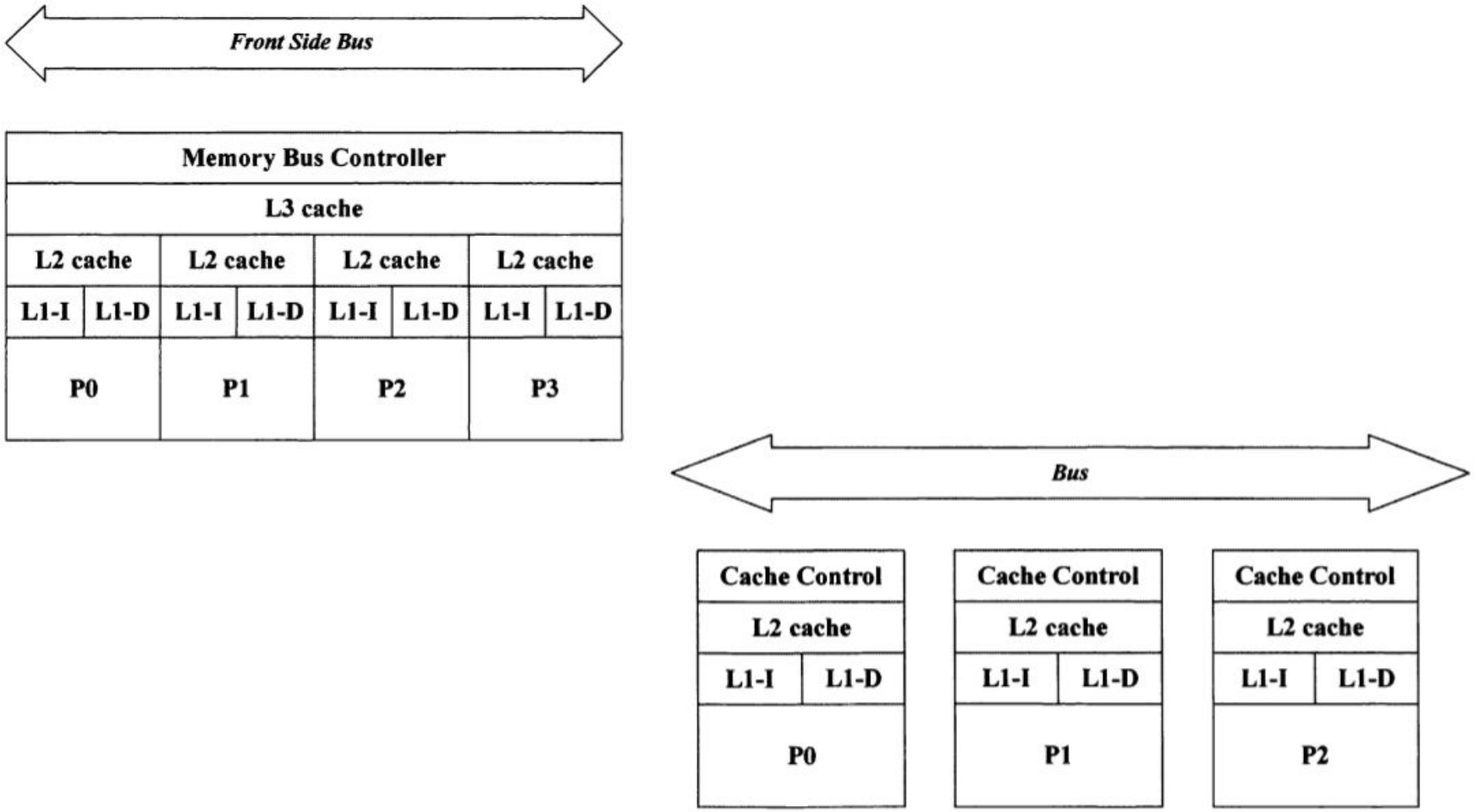
Основы технологии CUDA http://www.ixbt.com/video3/cuda-1.shtml



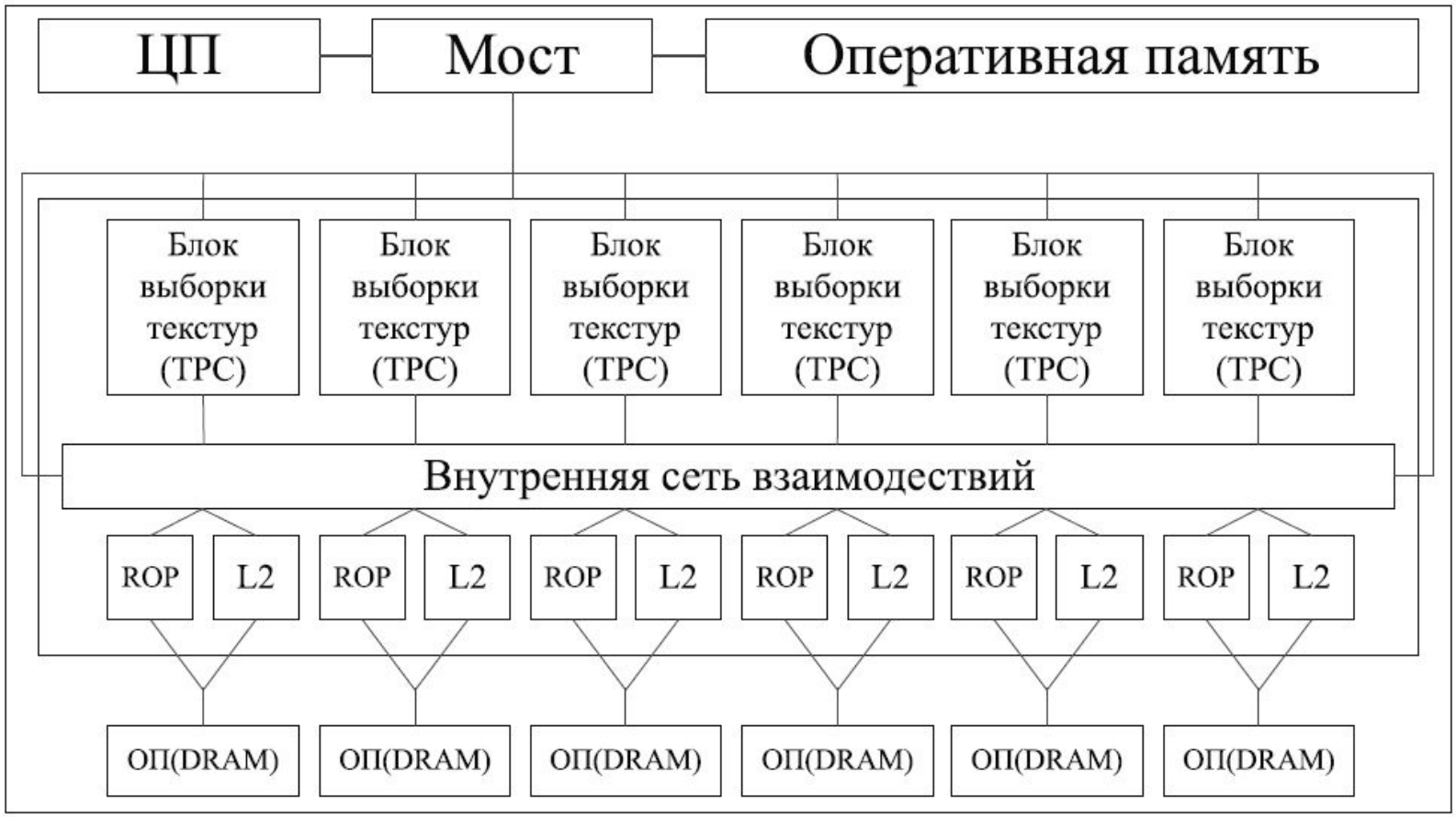
Код для сложения векторов, представленный в CUDA

```
//Размер вектора в элементах
const int N = 1048576;
//размер вектора в байтах
const int dataSize = N * sizeof(float);

//Выделение памяти CPU
float *h_A = (float *)malloc(dataSize);
float *h_B = (float *)malloc(dataSize);
float *h_C = (float *)malloc(dataSize);

//Выделение памяти GPU
float *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, dataSize));
cudaMalloc((void **)&d_B, dataSize));
cudaMalloc((void **)&d_C, dataSize));

//Инициализировать h_A[], h_B[]…

//Скопировать входные данные в GPU для обработки
cudaMemcpy(d_A, h_A, dataSize, cudaMemcpyHostToDevice) );
cudaMemcpy(d_B, h_B, dataSize, cudaMemcpyHostToDevice) );

//Запустить ядро из N / 256 блоков по 256 потоков
//Предполагая, что N кратно 256
vectorAdd<<<N / 256, 256>>>(d_C, d_A, d_B);

//Считать результаты GPU
cudaMemcpy(h_C, d_C, dataSize, cudaMemcpyDeviceToHost) );
```

Вычислительная схема CPU с несколькими ядрами SMP



CUDA
Вычислительная схема GPU



CUDA
Вычислительная схема GPU

SM (Streaming Multiprocessor) мультипроцессор;
SP (Streaming Processor) потоковый процессор

Основные термины.

•Хост(Host) - центральный процессор, управляющий выполнением программы.

• Устройство(Device)—видеоадаптер, выступающий в роли сопроцессора центрального процессора.

• Грид(Grid)—объединение блоков, которые выполняются наодном устройстве.

• Блок(Block)—объединение тредов, которое выполняется целиком на одномSM. Имеет

свой уникальный идентификатор внутри грида.

• Тред(Thread,поток)—единица выполнения программы. Имеетсвой уникальный идентификатор внутри блока.

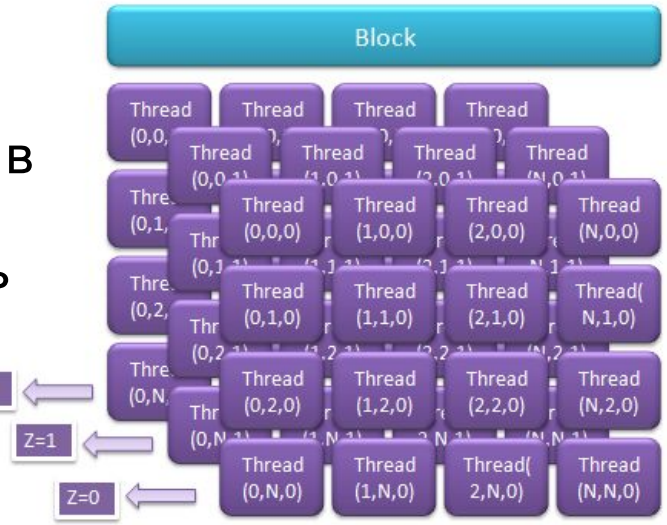
• Варп(Warp)—32 последовательно идущих треда, выполняется физически одновременно.

• Ядро(Kernel)—параллельная часть алгоритма, выполняется на гриде.

CUDA http://habrahabr.ru/post/54707/

Вычислительная модель GPU

- Верхний уровень ядра GPU состоит из блоков, которые группируются в сетку или грид (grid) размерностью N1 * N2 * N3. Размерность сетки блоков можно узнать с помощь функции cudaGetDeviceProperties, в полученной структуре за это отвечает поле maxGridSize.
- Любой блок в свою очередь состоит из нитей (threads), которые являются непосредственными исполнителями вычислений. Нити в блоке сформированы в виде трехмерного массива (рис. 2), размерность которого так же можно узнать с помощью функции cudaGetDeviceProperties, за это отвечает поле maxThreadsDim.



Память

Локальная —используется для хранения локальных переменных, когда регистров не хватает; скорость доступа низкая, так как расположена в микросхемах DRAM, находящихся вне кристалла. Выделяется отдельно для каждого треда.

Разделяемая —16КБ (или48КБнаFermi) на SM, используется для хранения массивов данных, используемых совместно всеми тредами в блоке. Расположена на кристалле GPU; имеет чуть меньшую скорость доступа, чем регистры (около 10 тактов). Выделяется на блок.

Глобальная —основная память видеокарты (на данный момент максимально 6Гб). Используется для хранения больших массивов данных. Расположена в микросхемах
DRAM и имеет медленную скорость доступа (около80тактов). Выделяется целиком на грид.

Константная —память, располагающаяся в микросхемах DRAM, снабжена специальным константным кэшем. Используется для передачи в ядро аргументов, превышаюших допустимые размеры для параметров ядра(для чипа Fermi—256 байт). Выделяется целиком на грид.

Текстурная —память, располагающаяся в микросхемах DRAM, кэшируется .Используется для хранения больших массивов данных. Выделяется целиком на грид.

CUDA и язык C:
Сама технология CUDA (компилятор nvcc.exe) вводит ряд дополнительных расширений для языка C, которые необходимы для написания кода для GPU:

1. Спецификаторы функций, которые показывают, как и откуда буду выполняться функции.
2. Спецификаторы переменных, которые служат для указания типа используемой памяти GPU.
3. Спецификаторы запуска ядра GPU.
4. Встроенные переменные для идентификации нитей, блоков и др. параметров при исполнении кода в ядре GPU .
5. Дополнительные типы переменных.

Спецификаторы функций определяют, как и откуда буду вызываться функции

__host__ — выполнятся на CPU, вызывается с CPU (в принципе его можно и не указывать).

__global__ — выполняется на GPU, вызывается с CPU.

__device__ — выполняется на GPU, вызывается с GPU.

Спецификаторы запуска ядра служат для описания количества блоков, нитей и памяти, которые вы хотите выделить при расчете на GPU.
myKernelFunc<<<gridSize, blockSize, sharedMemSize, cudaStream>>>(float* param1,float* param2), где

gridSize – размерность сетки блоков (dim3), выделенную для расчетов,

blockSize – размер блока (dim3), выделенного для расчетов,

sharedMemSize – размер дополнительной памяти, выделяемой при запуске ядра,

cudaStream – переменная cudaStream_t, задающая поток, в котором будет произведен вызов.

Встроенные переменные:

gridDim – размерность грида, имеет тип dim3. Позволяет узнать размер грида, выделенного при текущем вызове ядра.

blockDim – размерность блока, так же имеет тип dim3. Позволяет узнать размер блока, выделенного при текущем вызове ядра.

blockIdx – индекс текущего блока в вычислении на GPU, имеет тип uint3.

threadIdx – индекс текущей нити в вычислении на GPU, имеет тип uint3.

warpSize – размер warp’a, имеет тип int.

CUDA *API* (*application programming interface)12*

В CUDA есть два уровня API:низкоуровневый драйвер-API и высокоуровневый runtime-API. Runtime-API реализован через драйвер-API.

Runtime-API обладает меньшей гибкостью, но более удобен для написания программ. Оба API не требуют явной инициализации, и для использования дополнительных типов и других расширений языка C не требуется подключать дополнительные заголовочные файлы.

Все функции драйвер-API начинаются с приставки cu,все функции runtime-API начинаются с приставки cuda. Практически все функциии обоих API возвращают значение типа t_cudaError, которое принимает значение cudaSuccess в случае успеха.

CUDA host API, который является связующим звеном между CPU и GPU.

В CUDA runtime API входят следующие группы функций:

Device Management – включает функции для общего управления GPU (получение инфор-мации о возможностях GPU, переключение между GPU при работе SLI-режиме и т.д.).

Thread Management – управление нитями.

Stream Management – управление потоками.

Event Management – функция создания и управления event'ами.

Execution Control – функции запуска и исполнения ядра CUDA.

Memory Management – функции управлению памятью GPU.

Texture Reference Manager – работа с объектами текстур через CUDA.

OpenGL Interoperability – функции по взаимодействию с OpenGL API.

Direct3D 9 Interoperability – функции по взаимодействию с Direct3D 9 API.

Direct3D 10 Interoperability – функции по взаимодействию с Direct3D 10 API.

Error Handling – функции обработки ошибок.

Задача. Требуется вычислить сумму двух векторов размерностью N элементов.

Нам известна максимальные размеры нашего блока: 512*512*64 нитей. Так как вектор у нас одномерный, то пока ограничимся использованием x-измерения нашего блока, то есть задействуем только одну полосу нитей из блока. Заметим, что x-размерность блока 512, то есть, мы можем сложить за один раз векторы, длина которых N <= 512 элементов.

В самой программе необходимо выполнить следующие этапы:

1. Получить данные для расчетов.
2. Скопировать эти данные в GPU память.
3. Произвести вычисление в GPU через функцию ядра.
4. Скопировать вычисленные данные из GPU памяти в ОЗУ.
5. Посмотреть результаты.
6. Высвободить используемые ресурсы.

- Для выделения памяти на видеокарте используется функция **cudaMalloc**, которая имеет следующий прототип:
  cudaError_t cudaMalloc(void devPtr, size_t count),** где
  devPtr – указатель, в который записывается адрес выделенной памяти,
  count – размер выделяемой памяти в байтах.
  Возвращает:
  cudaSuccess – при удачном выделении памяти
  cudaErrorMemoryAllocation – при ошибке выделения памяти
- Для копирования данных в память видеокарты используется cudaMemcpy, которая имеет следующий прототип:
  cudaError_t cudaMemcpy(void* dst, const void* src ,size_t count, enum cudaMemcpyKind kind), где
  dst – указатель, содержащий адрес места-назначения копирования,
  src – указатель, содержащий адрес источника копирования,
  count – размер копируемого ресурса в байтах,
  cudaMemcpyKind – перечисление, указывающее направление копирования (может быть cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice).
  Возвращает:cudaSuccess – при удачном копировании
  cudaErrorInvalidValue – неверные параметры аргумента (например, размер копирования отрицателен)
  cudaErrorInvalidDevicePointer – неверный указатель памяти в видеокарте
  cudaErrorInvalidMemcpyDirection – неверное направление (например, перепутан источник и место-назначение копирования)

```
// Функция сложения двух векторов
__global__ void addVector(float* left, float* right, float* result)
{
  //Получаем id текущей нити.
  int idx = threadIdx.x;

  //Расчитываем результат.
  result[idx] = left[idx] + right[idx];
}
```

Программа

```
#define SIZE 512
__host__ int main()
{
  //Выделяем память под вектора
  float* vec1 = new float[SIZE];
  float* vec2 = new float[SIZE];
  float* vec3 = new float[SIZE];

  //Инициализируем значения векторов
  for (int i = 0; i < SIZE; i++)
  {
    vec1[i] = i;
    vec2[i] = i;
  }

  //Указатели на память видеокарте
  float* devVec1;
  float* devVec2;
  float* devVec3;
```

```
/Выделяем память для векторов на видеокарте
  cudaMalloc((void**)&devVec1, sizeof(float) * SIZE);
  cudaMalloc((void**)&devVec2, sizeof(float) * SIZE);
  cudaMalloc((void**)&devVec3, sizeof(float) * SIZE);
  //Копируем данные в память видеокарты
  cudaMemcpy(devVec1, vec1, sizeof(float) * SIZE,
cudaMemcpyHostToDevice);
  cudaMemcpy(devVec2, vec2, sizeof(float) * SIZE,
cudaMemcpyHostToDevice);
```

Непосредственный вызов ядра для вычисления на GPU.

```
dim3 gridSize = dim3(1, 1, 1);    //Размер используемого грида
dim3 blockSize = dim3(SIZE, 1, 1); //Размер используемого блока
//Выполняем вызов функции ядра
addVector<<<gridSize, blockSize>>>(devVec1, devVec2, devVec3);
```

Определять размер грида и блока необязательно, так как используем всего один блок и одно измерение в блоке, поэтому код выше можно записать:

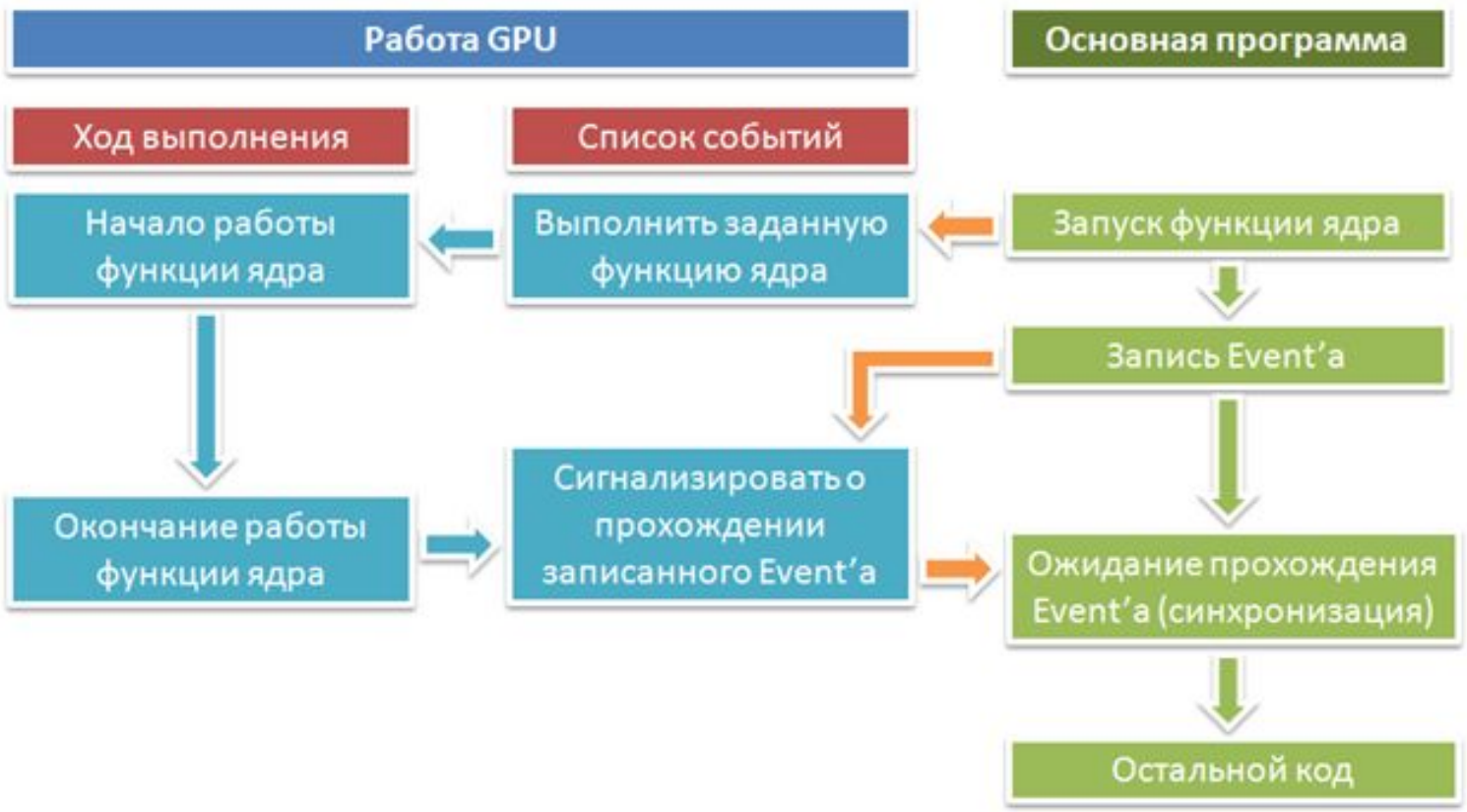
```
addVector<<<1, SIZE>>>(devVec1, devVec2, devVec3); …
```

Теперь нам остается скопировать результат расчета из видеопамяти в память хоста. Но у функций ядра при этом есть особенность – асинхронное исполнение, то есть, если после вызова ядра начал работать следующий участок кода, то это ещё не значит, что GPU выполнил расчеты. Для завершения работы заданной функции ядра необходимо использовать средства синхронизации, например event'ы. Поэтому, перед копированием результатов на хост выполняем синхронизацию нитей GPU через event.

Код после вызова ядра:

```
  //Выполняем вызов функции ядра
  addVector<<<blocks, threads>>>(devVec1, devVec2,
devVec3);
  //Хендл event'a
  cudaEvent_t syncEvent;
  cudaEventCreate(&syncEvent);    //Создаем event
  cudaEventRecord(syncEvent, 0);  //Записываем event
  cudaEventSynchronize(syncEvent);  //Синхронизируем
event
  //Только теперь получаем результат расчета
  cudaMemcpy(vec3, devVec3, sizeof(float) * SIZE,
cudaMemcpyDeviceToHost);
```

Синхронизация работы основной и GPU прграмм.



Рассмотрим более подробно функции из Event Managment API.
Event создается с помощью функции **cudaEventCreate**, прототип которой имеет вид:
cudaError_t cudaEventCreate(cudaEvent_t* event), где

***event** – указатель для записи хендла event'а.
Возвращает:

cudaSuccess – в случае успеха; cudaErrorInitializationError – ошибка инициализации

cudaErrorPriorLaunchFailure – ошибка при предыдущем асинхронном запуске функции

cudaErrorInvalidValue – неверное значение

cudaErrorMemoryAllocation – ошибка выделения памяти
Запись event'а осуществляется с помощью функции **cudaEventRecord**, прототип которой имеет вид:
cudaError_t cudaEventRecord(cudaEvent_t event, CUstream stream), где
event – хендл хаписываемого event'а,

stream – номер потока, в котором записываем (в нашем случае это основной нулевой по-ток).
Возвращает:
cudaSuccess – в случае успеха

cudaErrorInvalidValue – неверное значение

cudaErrorInitializationError – ошибка инициализации

cudaErrorPriorLaunchFailure – ошибка при предыдущем асинхронном запуске функции

cudaErrorInvalidResourceHandle – неверный хендл event'а

Синхронизация event'а выполняется функцией **cudaEventSynchronize**. Данная функция ожидает окончание работы всех нитей GPU и прохождение заданного event'а и только потом отдает управление вызывающей программе. Прототип функции имеет вид:
cudaError_t cudaEventSynchronize(cudaEvent_t event), где

event – хендл event'а, прохождение которого ожидается.
Возвращает: cudaSuccess – в случае успеха; cudaErrorInitializationError – ошибка инициализации;

cudaErrorPriorLaunchFailure – ошибка при предыдущем асинхронном запуске функции

cudaErrorInvalidValue – неверное значение; cudaErrorInvalidResourceHandle – неверный хендл event'а

Выводим результат на экран и чистим выделенные ресурсы.

```
//Результаты расчета
for (int i = 0; i < SIZE; i++)
{
  printf("Element #%i: %.1f\n", i , vec3[i]);
}
// Высвобождаем ресурсы
//
cudaEventDestroy(syncEvent);
cudaFree(devVec1);
cudaFree(devVec2);
cudaFree(devVec3);

delete[] vec1; vec1 = 0;
delete[] vec2; vec2 = 0;
delete[] vec3; vec3 = 0;
```

Создание CUDA проекта

http://habrahabr.ru/post/54330/

Необходимые элементы:

1. Видеокарта из серии nVidia GeForce 8xxx/9xxx или более современная
2. CUDA Toolkit v.2.1 (скачать можно здесь: www.nvidia.ru/object/cuda_get_ru.html)
3. CUDA SDK v.2.1 (скачать можно там же где Toolkit)
4. Visual Studio 2008
5. CUDA Visual Studio Wizard (скачать можно здесь:sourceforge.net/projects/cudavswizard/)

Листинг простой программы на CUDA, который выводит на экран информацию об аппаратных возможностях GPU

```
#include <stdio.h>
#include <cuda_runtime_api.h>

int main()
{  int deviceCount;
   cudaDeviceProp deviceProp;

//Сколько устройств CUDA установлено на PC.
   cudaGetDeviceCount(&deviceCount);

   printf("Device count: %d\n\n", deviceCount);
   for (int i = 0; i < deviceCount; i++)
   {
//Получаем информацию об устройстве
      cudaGetDeviceProperties(&deviceProp, i);
//Выводим иформацию об устройстве
```

Листинг простой программы на CUDA, который выводит на экран информацию об аппаратных возможностях GPU

```
   //Выводим иформацию об устройстве
   printf("Device name: %s\n", deviceProp.name);
   printf("Total global memory: %d\n", deviceProp.totalGlobalMem);
   printf("Shared memory per block: %d\n",
deviceProp.sharedMemPerBlock);
   printf("Registers per block: %d\n", deviceProp.regsPerBlock);
   printf("Warp size: %d\n", deviceProp.warpSize);
   printf("Memory pitch: %d\n", deviceProp.memPitch);

   printf("Max threads dimensions: x = %d, y = %d, z = %d\n",
     deviceProp.maxThreadsDim[0],
     deviceProp.maxThreadsDim[1],
     deviceProp.maxThreadsDim[2]);
   printf("Multiprocessor count: %d\n", deviceProp.multiProcessorCount);

   printf("Kernel execution timeout enabled: %s\n",
     deviceProp.kernelExecTimeoutEnabled ? "true" : "false"); }
  return 0;}
```