Программное обеспечение высокопроизводительных вычислительных систем

ЛАБОРАТОРНАЯ РАБОТА №3 (для А-10)

«Программирование графических процессоров средствами NVIDIA CUDA. Введение в использование GPU в качестве сопроцессора для ускорения вычислений»

Автор: Филатов А.В.

Графические процессоры используются в вычислительной технике уже давно. До недавнего времени (да и сейчас), основным их назначением был вывод графической информации на экран монитора вычислительной системы. Графические процессоры (*GPU*) включались в состав специальных устройств – видеоадаптеров, и, по мере своего развития, сначала помогали основному процессору (*CPU*), а позже взяли на себя основную заботу по подготовке и выводу графики на экран. Развитие функций *GPU* привело к такому их усовершенствованию, что они смогли выполнять многие универсальные вычислительные функции. Если сначала, функции *GPU* задавались жёстко на уровне аппаратуры, то потом разработчики графических процессоров позволили эти функции (так называемые шейдеры) программировать. Некоторые продвинутые программисты воспользовались этим, чтобы использовать графические адаптеры с *GPU* в качестве вычислительных сопроцессоров к основному *CPU*. Это оказалось непростым делом и требовало от программиста хороших знаний по программированию и функционированию *GPU*, поэтому оставалось делом узкого круга специалистов. Ситуация изменилась, когда известный мировой разработчик графических процессоров – компания *NVIDIA* создала в 2006 г. свой аппаратно-программный комплекс CUDA (*Compute Unified Device Architecture*). Этот аппаратно-программный комплекс позволяет упростить использование графических процессоров, разработанных компанией *NVIDIA*, в качестве вычислительных сопроцессоров и сводит их программирование к относительно простому расширению «обычных» программ на языках *C* и *C++* написанных для *CPU*.

Поскольку, *GPU* изначально создавались для графических видеоадаптеров, то и технология *CUDA* была сначала внедрена на видеоадаптерах *NVIDIA* серии *GeForce* (начиная с *GeForce 8800 GTX*). Позже, компания *NVIDIA* стала создавать уже специальные вычислительные

ускорители на *GPU*, такие как *Tesla*. Как видеоадаптеры *GeForce*, так и ускорители *Tesla* могут быть установлены в вычислительную систему с интерфейсом *PCI-Express* (в перспективе рассматривается использование других, возможно даже создание более быстрых специализированных) в количестве от одного и более, и, путем применения *CUDA*, использованы для вычислительных задач. На данный момент в мире получают широкое распространение кластерные вычислительные системы, узлы которых снабжаются ускорителями на базе *GPU*. К таким кластерным системам, в частности относится суперкомпьютер “Ломоносов” установленный в НИВЦ МГУ.

На рис. 1. схематически показаны: настольный персональный компьютер с видеоадаптером *NVIDIA* серии *GeForce* на шине *PCI-Express* (на его месте можно указать также и ноутбук со встроенным *GeForce*) и сервер, с несколькими ускорителями NVIDIA серии Tesla, также на шине *PCI-Express*.

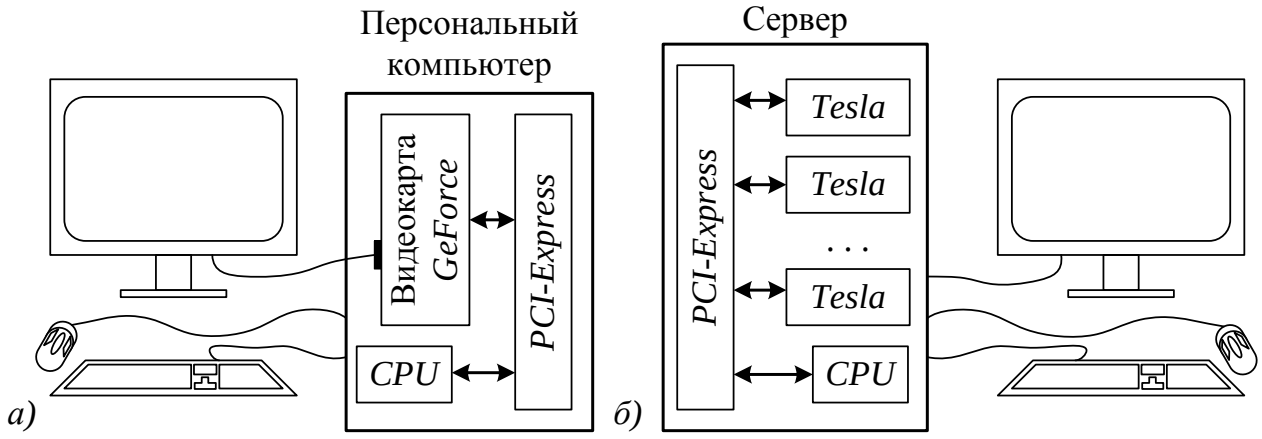

Рис. 1. Ускорители на *GPU* (*GeForce* и *Tesla*) в персональном компьютере *а)* и сервере *б)*

На рис. 2. схематически показано обобщенное внутреннее устройство одного ускорителя на *GPU*. Этим ускорителем может быть и видеоадаптер и специализированный ускоритель. У каждого конкретного устройства будет свой набор и количество составных частей. Важным отличием специализированного ускорителя от просто видеоадаптера является наличие у него дополнительных средств контроля, а также устройств для операций двойной точности. Ускоритель имеет в своём составе несколько (на момент разработки описания от 2 до 30 и более) потоковых мультипроцессоров *SM* (*Streaming Multiprocessor*). Каждый мультипроцессор состоит из элементарных ядер (*Core*) – потоковых процессоров *SP* (*Streaming Processor*) работающих под общим управлением по типу *SIMD*. Потоковые мультипроцессоры группируются (на данный момент по 2-3 штуки) в текстурные процессорные

кластеры *TPC* (*Texture Processor Cluster*) и имеют в рамках него общую текстурную (используется для хранения графических текстур) намять. На рис. 2. содержимое одного из *TPC* детализировано, а в нём детализированы два из трёх имеющихся в нем *SM*. Каждому из потоковых процессоров *SP* аппаратный планировщик выделяет (в момент вычислений) в его распоряжение часть регистрового массива *RM*. При решении задачи, в каждый момент времени, на каждом *SP* выполняется один поток вычислений – *thread*, однако в отличие от классических потоков, потоки всех *SP* входящих в *SM* выполняются под общим управлением единого для всего *SM* устройства управления (то есть по принципу *SIMD*, как было отмечено ранее).

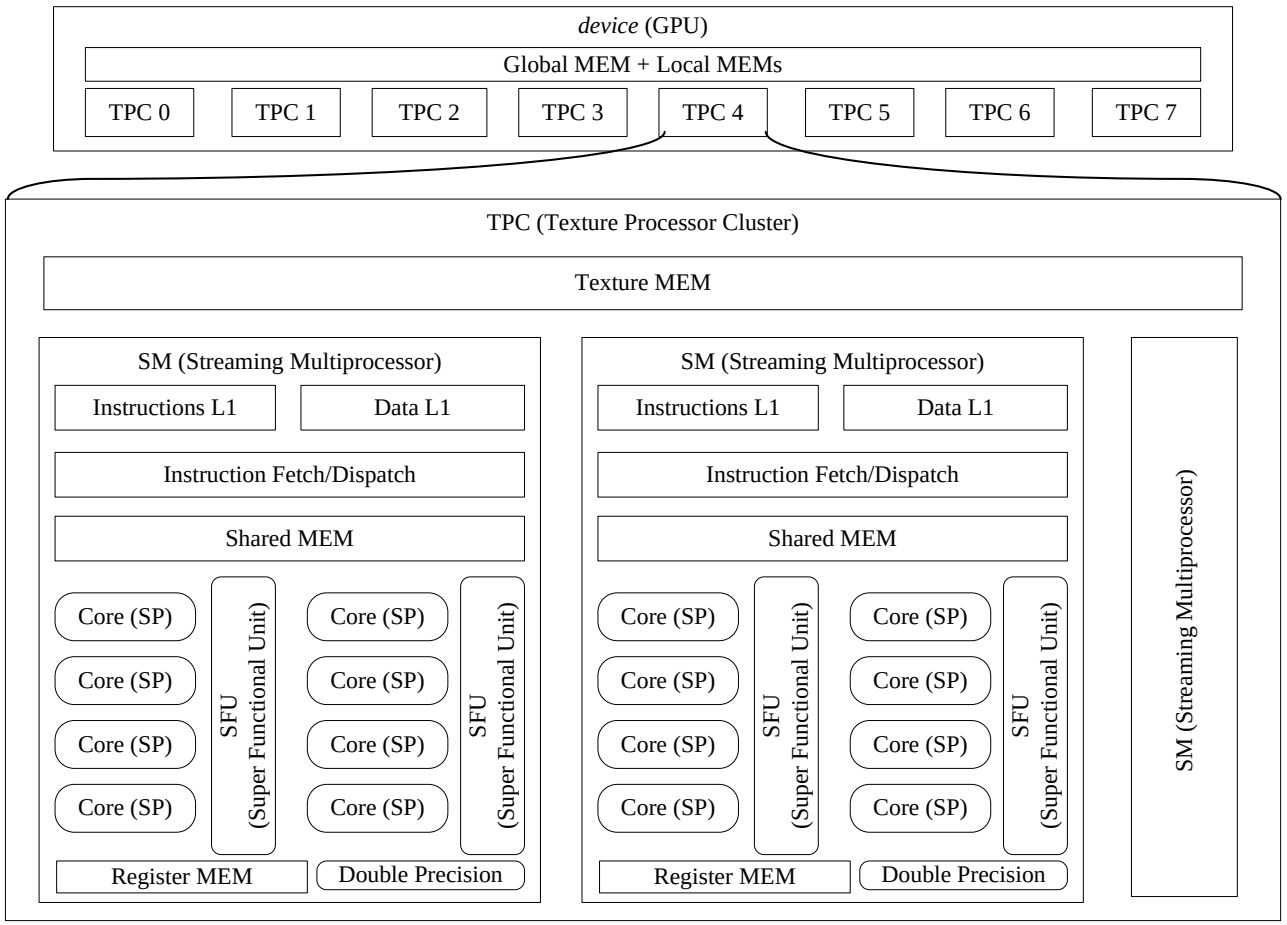

Рис. 2. Внутреннее устройство ускорителя на *GPU*.

На каждый SM может быть назначено любое количество потоков. В случае, если их количество больше числа *SP* в *SM*, то тогда все потоки будут поделены на варпы. Варп (*warp*) – это совокупность потоков, единовременно выполняемых в рамках одного потокового мультипроцессора *SM*. На момент создания данного описания, типичный размер варпа равен 32. Кроме SP, в SM ещё имеются блоки специальных функций (*SFU*).

Выполняющиеся в рамках *SM* потоки, могут взаимодействовать друг с другом посредством небольшой (ёмкостью, измеряемой килобайтами, например в 48 Кб) быстродействующей (поскольку она интегрирована на кристалл вместе с *SM*) общей памяти *Shared MEM*. Также у *SM* имеются КЭШи *L1* и дополнительные устройства для вычислений с двойной точностью. Вне *SM* имеются ёмкие, но довольно медленные устройства глобальной *Global MEM* (общей для всего ускорителя *GPU*) и локальной *Local MEM* (для каждого мультипроцессора) памяти. На рис. 2 не указана имеющаяся у каждого *SM* константная память, но о ней коротко будет сказано ниже.

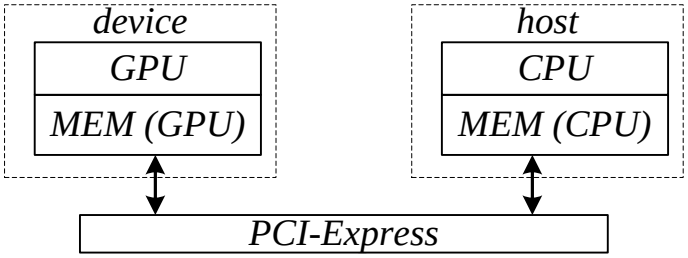

Рис. 3. Связь устройств *host* и *device*

На рис. 3. показана связь между основным процессором вычислительной системы *CPU* со своей памятью и ускорителем на *GPU* с его памятью. В терминах *CUDA* одно устройство (с *CPU*) называется *host*, а другое (с *GPU*)– *device*. Первым важным моментом является то, что программа, написанная в стиле *CUDA*, будет выполняться и на *host* и на *device*. Отсюда следует второй важный момент, что часть исходного текста программы должна быть скомпилирована для *host* стандартным компилятором *CPU*, а другая часть исходного текста программы для *device* должна быть скомпилирована компилятором *NVIDIA* для *GPU*. Такая компиляция производится автоматически специальными инструментариями *NVIDIA CUDA Toolkit*, разрабатываемыми под конкретные аппаратные и программные платформы. Третьим важным моментом на сегодня, является программная (на уровне пользователя) взаимная недоступность памяти устройств *host* и *device*. Для этого необходимо средствами *CUDA* осуществлять явное перемещение данных из памяти одного устройства в память другого.

На рис. 4. Приведен схематический пример программы на языке *C* в стиле *CUDA*. В тексте программы имеются две функции: *main* и *FUN_KERNEL*. Функция *main* является стандартной, будет скомпилирована под *CPU* и запущена для выполнения на *host*. Функция *FUN_KERNEL* (предположим мы так её решили назвать) будет скомпилирована под *GPU*, и запущена на *device* после того как её специальным образом вызовет функция *main*. Функции,

предназначенные для запуска на *device* из *host*-а, в терминах *CUDA* называются ядрами. То, что эта функция предназначена для выполнения на *device*, указывает специальный префикс *__global__*, который обозначает, что функция с её данными будет размещена в глобальной памяти ускорителя *GPU*.

```
#include<stdio.h>

//Функция-ядро, которая будет выполняться на device (GPU)
__global__ void FUN_KERNEL (формальные параметры функции-ядра)
{
        //Тело функции-ядра, Которое будет по умолчанию выполняться всеми запущенными
          потоками на GPU
}

// Функция main, которая будет выполняться на host (CPU)
int main ( int argc, char *  argv [] )
{
        ПОДГОТОВКА ДАНЫХ НА host

        //Выделение памяти на GPU
        cudaMalloc ( указатель памяти device, размер);
        // Копирование данных в память device из памяти host
        cudaMemcpy ( указатель памяти device, указатель памяти host, размер данных,
                        cudaMemcpyHostToDevice);

        // ЗАПУСК ЯДРА НА device
        FUN_KERNEL<<< параметры исполнения >>> (параметры функции-ядра);

        //Копирование данных в память host из памяти device
        cudaMemcpy ( указатель памяти CPU, указатель памяти GPU, размер данных,
                        cudaMemcpyDeviceToHost );

        ИСПОЛЬЗОВАНИЕ РЕЗУЛЬТАТОВ РАБОТЫ device НА host

        cudaFree   (указатель памяти device); //Очистка памяти на устройстве
        return 0;
}
```

Рис. 4. Схематический пример *CUDA*-программы

В функции main могут выполняться любые действия. Предположим, что результатом этих действий являются некоторые данные, которые надо обработать на *GPU*. Для этого с помощью функции *cudaMalloc* необходимо на *device* выделить память заданного размера и ассоциировать её с указателем. Потом, с помощью функции *cudaMemcpy* осуществляется копирование данных из памяти *host* в память *device* (указатели ставятся во второй и первый

параметры функции соответственно), а также размер данных и ключевое слово *cudaMemcpyHostToDevice*, задающее направление копирования.

Ключевым моментом является вызов ядра. Сначала пишется имя функции-ядра, потом в тройных угловых скобках параметры запуска (о них позже) и далее в круглых скобках фактические параметры самой функции. На время выполнения ядра, функция *main* на *host* приостанавливает свою работу.

После исполнения ядра, результаты работы копируются из памяти *device* в память *host*. Для этого снова используется функция *cudaMemcpy*, но с ключевым словом *cudaMemcpyDeviceToHost* в четвёртом параметре. Далее функция *main* может использовать результаты полученные с *GPU*.

Каким образом ядро выполняется на *GPU*? Как уже упоминалось, ядро выполняется множеством потоков. Вся совокупность потоков, запускаемых для выполнения ядра, образует так называемую сетку – *grid*. Все потоки *grid*-а могут быть разбиты на блоки. Эта часто бывает полезно сделать, поскольку потоки одного блока назначаются на один мультипроцессор *SM*, и, следовательно, у них есть возможность пользоваться для взаимодействия общей (s*hared*) памятью мультипроцессора и кое-чем другим.

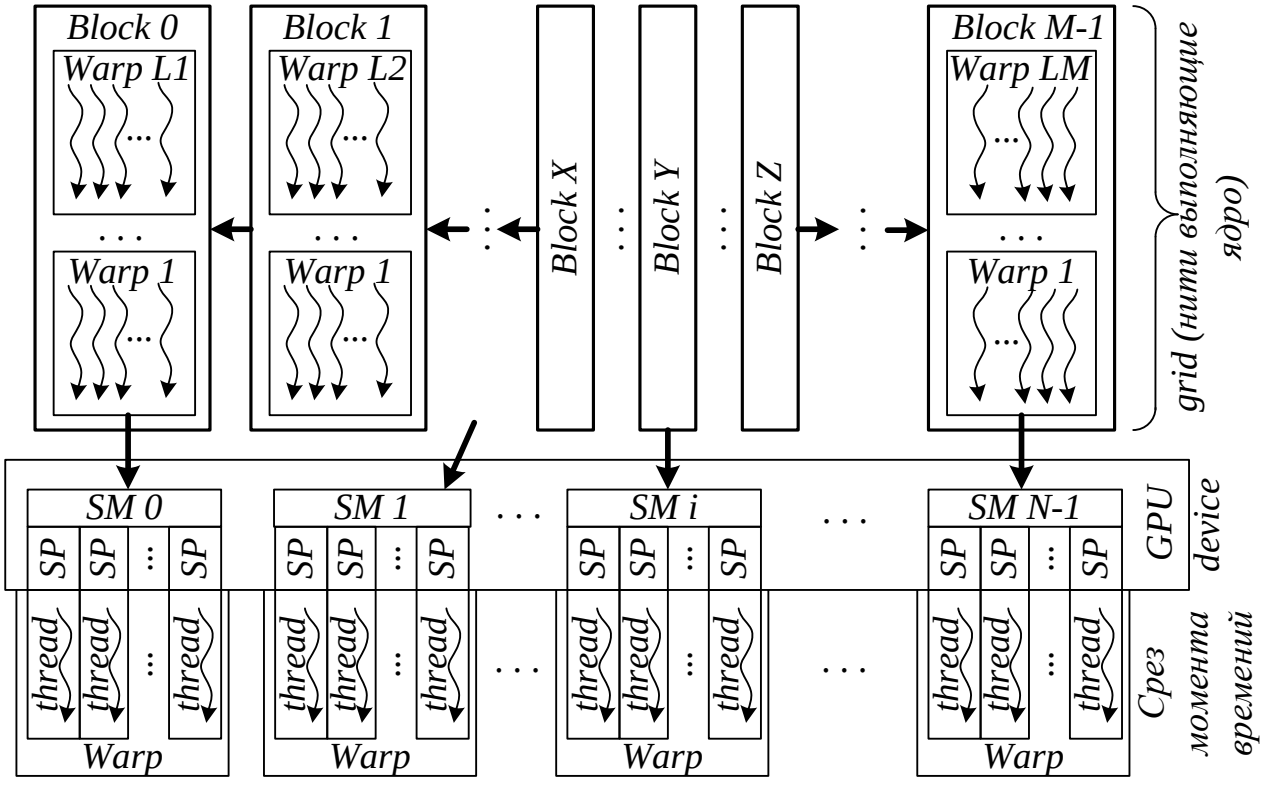

Рис. 5. Выполнение ядра на *GPU*

На рис. 5. схематически представлено выполнение ядра на *GPU*. Для выполнения некоторого ядра запускается *M* блоков, в каждом блоке

запускается некоторое заранее заданное количество потоков. Потоки каждого блока выполняются независимо от потоков других блоков. Потоки же одного блока выполняются по схеме *SIMD*. Как было отмечено, потоки выполняются на мультипроцессорах варпами. В результате, одновременно на *GPU* может максимально выполняться то количество нитей, сколько максимально может быть в варпе, помноженное на количество единовременно задействованных мультипроцессоров *SM*.

На рис. 6. приведён пример настоящей программы. Эта программа демонстрирует подсчёт суммы двух векторов.

```
#include<stdio.h>
#define N 4096

__global__ void FUN_KERNEL ( float * A, float * B, int S)
{
  int k;
  int idx_thread = blockIdx.x * blockDim.x + threadIdx.x;

  for (k=0;k<S;k++) A[idx_thread*S+k]=A[idx_thread*S+k]+B[idx_thread*S+k];
}

int main ( int argc, char *  argv [] )
{
      int         i, blocks, blocksize, steps;
      float       vect1[N],vect2[N];
      float       * devA, *devB;

  for (i=0; i<N; i++) { vect1[i]=i; vect2[i]=i;}

  cudaMalloc ( (void**)&devA, N * sizeof ( float ) ) ;
  cudaMalloc ( (void**)&devB, N * sizeof ( float ) ) ;

  blocks=4; blocksize=64;
  steps=(int)N/(blocks*blocksize);

  cudaMemcpy ( devA, vect1, N * sizeof ( float ), cudaMemcpyHostToDevice);
  cudaMemcpy ( devB, vect2, N*sizeof ( float ), cudaMemcpyHostToDevice);

  FUN_KERNEL<<<blocks,blocksize>>> ( devA, devB, steps);

  cudaMemcpy ( vect1, devA, N * sizeof ( float ), cudaMemcpyDeviceToHost );
  cudaFree   ( devA  );

  for (i = 0; i < N; i++)  printf("vect1[%d] = %.5f\n", i, vect1[i]);
  return 0;
}
```

Рис. 6. Пример программы сложения двух векторов

На устройстве *host*, в функции *main* осуществляется формирование содержимого векторов *vect1* и *vect2*. С помощью функций *cudaMalloc* на устройстве *device* выделяется память для обоих векторов, после чего *host*-у возвращаются указатели на эти области (*devA* и *devB*). Следует иметь в виду, что части программы, выполняемые на *host*-е, не могут (об этом уже говорилось) адресовать память на *device*-е и наоборот, части программы, выполняемые на *device*-е, не могут адресовать память на *host*-е. Поэтому данные должны быть скопированы средствами *CUDA* (функция *cudaMemcpy*) в память *device*. По окончании вычислений, с помощью этой же функции, результаты копируются в память *host*.

При вызове ядра (*FUN_KERNEL*) были установлены следующие параметры исполнения: количество блоков потоков (переменная *blocks*=4) и количество потоков в каждом блоке (переменная *blocksize*=64). В качестве параметров, функции-ядру передаются указатели на вектора в памяти *device* (переменные *devA* и *devB*) и количество элементов векторов, которые должен обработать один поток (значение переменной *steps*).

Сама функция-ядро *FUN_KERNEL* будет выполняться всеми запущенными на *device* потоками по принципу *SPMD*. Для каждого потока будут созданы свои экземпляры переменных, объявленных внутри функции (*k* и *idx_kernel*). Складываемые вектора будут находиться в глобальной памяти *device*, и к ним будут иметь доступ все потоки. Указатели на эти вектора, функции *FUN_KERNEL* передаются через параметры *A* и *B* при её вызове. Через параметр *S* передаётся количество элементов, которое должен обработать каждый поток.

Для определения номера потока в задаче, в ядре используются следующие собственные структурные переменные среды *CUDA*: *blockIdx.x* – номер блока в задаче; *blockDim.x* – размер блока (т.е. сколько в нем потоков); *threadIdx.x* – номер потока в текущем блоке. Блоки в задаче могут быть многомерными. Пока мы используем только одномерный вариант, поэтому берём значения лишь по измерению *x*.

На рис. 7. приведён пример функции *main*, в которой использованы средства *CUDA* для замера времени. С помощью функций *cudaEventRecord* фиксируются события начала (*start*) и конца (*stop*) процедуры вычислений. Предварительно события необходимо инициализировать функцией *cudaEventCreate*. К сожалению, *host* не всегда дожидается окончания функционирования *device*, поэтому для правильной фиксации события необходимо произвести синхронизацию функцией *cudaEventSynchronize*. Определить время между событиями позволяет функция *cudaEventElapsedTime*. Эта функция сохраняет время в переменной *elapsedTime* (тип *float*).

```
int main ( int argc, char *  argv [] )
{
      int i,blocks,blocksize,steps;
                  float vect1[N],vect2[N];
                  FILE *f;
                  float * devA, *devB, elapsedTime;

                  cudaEvent_t start, stop; //Идентификаторы событий

 cudaEventCreate(&start);  //Инициализация события start
 cudaEventCreate(&stop);  //Инициализация события stop

 for (i=0; i<N; i++) { vect1[i]=i; vect2[i]=i;}

 cudaMalloc ( (void**)&devA, N * sizeof ( float ) ) ;
 cudaMalloc ( (void**)&devB, N * sizeof ( float ) ) ;

 blocks=16; blocksize=128;
 steps=(int)N/(blocks*blocksize);

 cudaEventRecord(start,0); // Фиксация события start
 cudaMemcpy ( devA, vect1, N * sizeof ( float ), cudaMemcpyHostToDevice);
 cudaMemcpy ( devB, vect2, N*sizeof ( float ), cudaMemcpyHostToDevice);

 FUN_KERNEL<<<blocks,blocksize>>> ( devA, devB, steps);

 cudaMemcpy ( vect1, devA, N * sizeof ( float ), cudaMemcpyDeviceToHost );
 cudaFree   ( devA  );

 cudaEventRecord(stop,0);  // Фиксация события stop
 cudaEventSynchronize(stop); // Синхронизация host и device по событию stop

 // Определение времени (в миллисекундах)  между событиями start и stop
 cudaEventElapsedTime(&elapsedTime,start,stop);

 printf("time1 = %f\n", elapsedTime); //Вывод времени
 for (i = 0; i < N; i++)  printf("vect1[%d] = %.5f\n", i, vect1[i]);

 return 0;
}
```

Рис. 7. Пример функции *main* с замером времени

Теперь немного о памяти *GPU*.

Всего в *GPU* имеется шесть видов памяти:

- регистровая (*register*), это быстродействующая, программно недоступная память;

- константная (*constant*), это быстродействующая память, запись в которую можно осуществлять только с *host*. Значения данных, загруженных в

константную память, потоки на *device* могут использовать, но не могут изменять.

- общая (*shared*), это небольшая быстродействующая внутрикристальная память у каждого *SM*, доступная для записи и чтения всем потоковым процессорам *SP* мультипроцессора *SM*;

- текстурная (*texture*), это общая для мультипроцессоров *SM* одного *TPC* память. Она предназначена для хранения текстур графического процессора, но при вычислениях может быть использована как особый вид константной памяти;

- глобальная (*global*), это общедоступная для всего *GPU* память. Её ёмкость может измеряться гигабайтами (именно это значение указывается в качестве ёмкости видеопамяти). Время доступа к ней достаточно велико, поэтому при вычислениях она часто используется как некий базовый накопитель, откуда инструкции и данные считываются в кэши и другие модули памяти, где и производится реальная работа, а потом в ней сохраняется результат;

- локальная (*local*), это внешняя, достаточного ёмкая, но медленная память каждого отдельного мультипроцессора, она в частности содержит данные локальных функций.

ЗАДАНИЕ

1. Используя специальную программу, определить параметры имеющегося у Вас ускорителя *GPU* (*device*).
2. Разработать, опираясь на примеры рис. 6 и рис. 7, программу на *CUDA*, реализующую вычисления из таблицы 1 (вариант задания берётся на основе номера из журнала лабораторных работ).
3. Разработать аналогичный п.2. вариант программы только для *CPU*, предусматривающий чисто последовательную обработку. Для замера времени выполнения вычислений использовать средства *CUDA* как в п.2.
4. Выполнить программу п.3. два раза для размеров векторов 16384 и 65536 элементов. Время выполнения вычислений занести в протокол. Будем называть эти времена – временами последовательного выполнения на *host*.
5. Выполнить программу п.2. два раза для размеров векторов 16384 и 65536 элементов. Для выполнения ядра запустить один блок с одним потоком. Времена выполнения вычислений занести в протокол как времена последовательного выполнения на *device*.

6. Провести для каждого из двух размеров (16384 и 65536 элементов) векторов серию из 35 запусков программы п.2., запуская для выполнения ядра 1, 2, 4, 8 и 16 блоков, с 1 (кроме случая, когда один блок), 4, 32, 64, 256, 2048 нитями в блоке. Все времена занести в протокол.
7. Составить отчёт, содержащий:
   - параметры ускорителя (*device*), который был использован для экспериментов;
   - вариант задания на ЛР;
   - исходные тексты программ п.2 и п.3.;
   - для каждого из двух размеров векторов привести по две таблицы. Таблицу с коэффициентами ускорения многопоточных (параллельных) вариантов исполнения вычислений к последовательному на *device*. Таблицу с коэффициентами ускорения многопоточных (параллельных) вариантов исполнения вычислений к последовательному на *host*. Для формирования каждой из четырёх таблиц выбрать любые 8 наиболее представительных (по Вашему мнению) результатов п.6;
   - На основе таблиц коэффициентов ускорений построить графики.

Примечание: Для ускорения работы, ядро в функции *main* можно запускать несколько раз с разными параметрами исполнения.

ЛИТЕРАТУРА

1. Сандерс Дж., Кэндрот Э., Технология CUDA в примерах: Введение в программирование графических процессоров: Пер. с англ. Слинкина А.А., научный редактор Боресков А.В. – М.: ДМК Пресс, 2011. – 232с.: ил.
2. Боресков А.В., Харламов А.А., Основы работы с технологией CUDA. – М.: ДМК Пресс, 2011. – 232 с.: ил.

Варианты задания

№ бригады	Задание
1	$\mathbf{Y}_i=\mathbf{A}_i\mathbf{B}_i+\mathbf{A}^m{}_i\mathbf{C}_i/\mathbf{B}_i$
2	$\mathbf{Y}_i=(\mathbf{A}_i+\mathbf{C}_i)/(\mathbf{B}^m{}_i-\mathbf{D}_i)$
3	$\mathbf{Y}_i=\mathbf{A}_i(\mathbf{B}_i\mathbf{C}_i+13)/S1^m$
4	$\mathbf{Y}_i=\mathbf{C}_i/(\mathbf{A}_i+\mathbf{D}_i)+S1\mathbf{B}^m{}_i$
5	$\mathbf{Y}_i=(S1^m-3\mathbf{B}_i)/(\mathbf{A}_i+S2)$
6	$\mathbf{Y}_i=\mathbf{B}_i\mathbf{D}_i/(\mathbf{D}_i+\mathbf{E}_i)^m+\mathbf{A}_i\mathbf{C}_i$
7	$\mathbf{Y}_i=(\mathbf{C}^m{}_i-S2)/(\mathbf{A}_i+\mathbf{D}_i\mathbf{D}_i)+S1$
8	$\mathbf{Y}_i=\mathbf{A}_i(\mathbf{B}_i\mathbf{C}_i+34*\mathbf{A}^m{}_i)/S1^m$

Примечания:

1. **A**, **B**, **C**, **D**, **E** и **Y** – векторы, S1 и S2 – скаляры;
2. m – степень (***2*** для А-7-08, ***3*** для А-8-08, ***4*** для А-9-08)