Лабораторная работа №2
Работа с памятью в NVIDIA CUDA при обработке векторов и матриц

Основной процесс приложения CUDA работает на универсальном процессоре (host), он запускает несколько копий процессов *kernel* на видеокарте. Код для CPU делает следующее: инициализирует GPU, распределяет память на видеокарте и системе, копирует константы в память видеокарты, запускает несколько копий процессов *kernel* на видеокарте, копирует полученный результат из видеопамяти, освобождает память и завершает работу.

В качестве примера для понимания приведем CPU код для сложения векторов, представленный в CUDA (рис. 2.1).

```
//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) );
```

Рис. 2.1. Код для сложения векторов.

Функции, исполняемые видеочипом, имеют следующие ограничения: отсутствует рекурсия, нет статических переменных внутри функций и переменного числа аргументов. Поддерживается два вида управления памятью: линейная память с доступом по 32-битным указателям, и CUDA-массивы с доступом только через функции текстурной выборки.

Программы на CUDA могут взаимодействовать с графическими API: для рендеринга данных, сгенерированных в программе, для считывания результатов рендеринга и их обработки средствами CUDA (например, при реализации фильтров постобработки). Для этого ресурсы графических API могут быть отображены (с получением адреса ресурса) в пространство глобальной памяти CUDA. Поддерживаются следующие типы ресурсов графических API: Buffer Objects (PBO / VBO) в OpenGL, вершинные буферы и текстуры (2D, 3D и кубические карты) Direct3D9.

Стадии компиляции CUDA-приложения представлены на рис. 2.2.

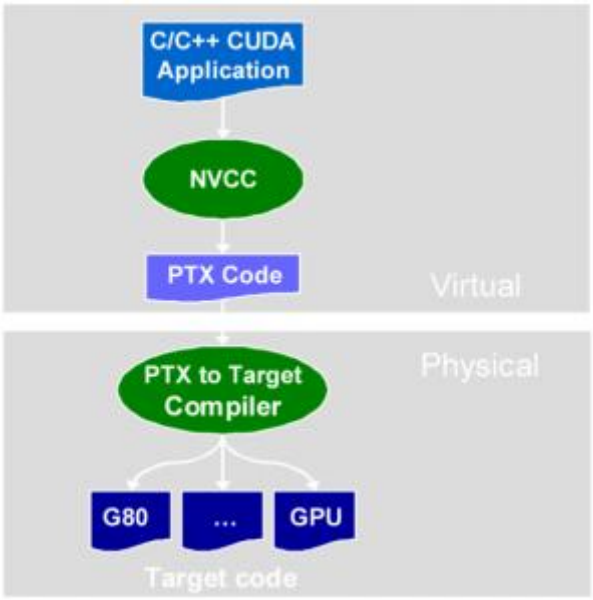

Рис. 2.2. Стадии компиляции CUDA-приложения.

Файлы исходного кода на CUDA C компилируются при помощи программы NVCC, которая является оболочкой над другими инструментами, и вызывает их: cudacc, g++, cl и др. NVCC генерирует: код для центрального процессора, который компилируется вместе с остальными частями приложения, написанными на чистом Си, и объектный код PTX для видеочипа. Исполнимые файлы с кодом на CUDA в обязательном порядке требуют наличия библиотек CUDA runtime library (cudart) и CUDA core library (cuda).

В состав CUDA входят *runtime*-библиотеки (*RTL, Run Time Library*):

- общая часть, предоставляющая встроенные векторные типы и подмножества вызовов RTL, поддерживаемые на CPU и GPU;
- CPU-компонента, для управления одним или несколькими GPU;
- GPU-компонента, предоставляющая специфические функции для GPU.

Вычислительная модель GPU

Рассмотрим вычислительную модель GPU более подробно. Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) — одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков

(thread blocks). Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме.

1. Верхний уровень ядра GPU состоит из блоков, которые группируются в сетку или грид (grid) размерностью N1 * N2 * N3.
2. Размерность сетки блоков можно узнать с помощь функции cudaGetDeviceProperties, в полученной структуре за это отвечает поле maxGridSize. Например, если размерность сетки указана как 65535*65535*1, то есть сетка блоков – двумерная (что удовлетворяет Compute Capability v.1.1).
3. Блок состоит из нитей (threads), которые являются непосредственными исполнителями вычислений. Нити в блоке сформированы в виде трехмерного массива, размерность которого так же можно узнать с помощью функции cudaGetDeviceProperties, за это отвечает поле maxThreadsDim.

При использовании GPU можно задействовать сетку необходимого размера и сконфигурировать блоки под нужды конкретной задачи.

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 будут 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 будут вызываться функции. Всего в CUDA 3 таких спецификатора:

- **__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 требуется выделить при расчете на GPU. Синтаксис запуска ядра имеет вид:

myKernelFunc<<<gridSize, blockSize, sharedMemSize, cudaStream>>>
(float* param1,float* param2),

где

- **gridSize** – размерность сетки блоков (dim3), выделенной 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,
- myKernelFunc – функция ядра (спецификатор __global__).

Некоторые переменные при вызове ядра можно опускать, например sharedMemSize и cudaStream.

Так же стоит упомянуть о встроенных переменных:

- **gridDim** – размерность сетки, имеет тип dim3. Позволяет узнать размер сетки, выделенной 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'а, имеет тип int.

Переменные gridDim и blockDim мы передаем при запуске ядра GPU, но в ядре они могут быть read only.

CUDA host API

Перед тем, как приступить к непосредственному использованию CUDA для вычислений, необходимо ознакомиться с так называемым CUDA host API, который является связующим звеном между CPU и GPU. CUDA host API в свою очередь можно разделить на низкоуровневое API под названием CUDA driver API, который предоставляет доступ к драйверу пользовательского режима CUDA, и высокоуровневое API – CUDA runtime API. В приводимых примерах используется CUDA runtime API.

В CUDA runtime API входят следующие группы функций:

- **Device Management** – включает функции для общего управления GPU (получение информации 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 событиями (events).
- **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.

Пример 1 программирования на CUDA

Нить – непосредственный исполнитель вычислений. Каким же тогда образом происходит распараллеливание вычислений между нитями? Рассмотрим работу отдельно взятого блока.

Задача. Требуется вычислить сумму двух векторов размерностью N элементов.

Известны максимальные размеры блока: 512*512*64 нитей. Так как вектор одномерный, то пока ограничимся использованием x-измерения нашего блока, то есть задействуем только одну полосу нитей из блока.

x-размерность блока равна 512, то есть, можно сложить за один раз векторы, длина которых N <= 512 элементов. Впрочем, при более массивных вычислениях, можно использовать большее число блоков и многомерные массивы.

В самой программе необходимо выполнить следующие этапы:

- Получить данные для расчетов.
- Скопировать эти данные в GPU память.
- Произвести вычисление в GPU через функцию ядра.
- Скопировать вычисленные данные из GPU памяти в ОЗУ.
- Посмотреть результаты.
- Высвободить используемые ресурсы.

Переходим непосредственно к написанию кода. Напишем функцию ядра, которая будет осуществлять сложение векторов:

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

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

Распараллеливание будет выполнено автоматически при запуске ядра. В этой функции так же используется встроенная переменная threadIdx (её поле x), которая позволяет задать соответствие между расчетом элемента вектора и нитью в блоке. Выполняем расчет каждого элемента вектора в отдельной нити.

```
      #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);
    …
}
```

Для выделения памяти на видеокарте используется функция **cudaMalloc**, которая имеет прототип:

cudaError_t cudaMalloc(void** devPtr, size_t count), где

devPtr – указатель, в который записывается адрес выделенной памяти,

count – размер выделяемой памяти в байтах.

Функция возвращает значение **cudaSuccess** – при удачном выделении памяти, **cudaErrorMemoryAllocation** – при ошибке выделения памяти.

Для копирования данных в память видеокарты используется cudaMemcpy, которая имеет 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).

Функция возвращает одно из следующих значений:

1. **cudaSuccess** – при удачном копировании
2. **cudaErrorInvalidValue** – неверные параметры аргумента (например, размер копируемого значения отрицателен)

3. **cudaErrorInvalidDevicePointe**r – неверный указатель памяти в видеокарте
4. c**udaErrorInvalidMemcpyDirection** – неверное направление (например, перепутан источник и место назначения копирования).

Переходим к непосредственному вызову ядра для вычисления на GPU.

```
…
dim3 gridSize = dim3(1, 1, 1);    //Размер используемой сетки
dim3 blockSize = dim3(SIZE, 1, 1); //Размер используемого блока

//Выполняем вызов функции ядра
addVector<<<gridSize, blockSize>>>(devVec1, devVec2, devVec3);
…
```

В нашем случае определять размер сетки и блока необязательно, так как используем всего один блок и одно измерение в блоке, поэтому код, приведенный выше, можно записать в виде:

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

Осталось только скопировать результат вычислений 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, события (events). Поэтому, перед копированием результатов на хост выполняем синхронизацию нитей GPU через event.

Код, который необходимо выполнить после вызова ядра:

```
    //Выполняем вызов функции ядра
addVector<<<blocks, threads>>>(devVec1, devVec2, devVec3);

cudaEvent_t syncEvent;                  //Дескриптор события
cudaEventCreate(&syncEvent);            //Создаем event
cudaEventRecord(syncEvent, 0);          //Записываем event
cudaEventSynchronize(syncEvent);        //Синхронизируем event

//Теперь получаем результат расчета
cudaMemcpy(vec3, devVec3, sizeof(float) * SIZE, cudaMemcpyDeviceToHost);
```

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 дескриптора события.

Функция возвращает одно из следующих значений:

1. **cudaSuccess** – в случае успеха
2. **cudaErrorInitializationError** – ошибка инициализации
3. **cudaErrorPriorLaunchFailure** – ошибка при предыдущем асинхронном запуске функции
4. **cudaErrorInvalidValue** – неверное значение
5. **cudaErrorMemoryAllocation** – ошибка выделения памяти

Запись события осуществляется с помощью функции **cudaEventRecord**, прототип которой имеет вид:

cudaError_t cudaEventRecord(cudaEvent_t event, CUstream stream),

где **event** – дескриптор записываемого события, **stream** – номер потока, в котором происходит запись (в нашем случае это основной нулевой поток).

Функция возвращает одно из следующих значений:

1. **cudaSuccess** – в случае успеха
2. **cudaErrorInvalidValue** – неверное значение
3. **cudaErrorInitializationError** – ошибка инициализации
4. **cudaErrorPriorLaunchFailure** – ошибка при предыдущем асинхронном запуске функции
5. **cudaErrorInvalidResourceHandle** – неверный дескриптор события.

Синхронизация события выполняется функцией cudaEventSynchronize(), которая ожидает окончания работы всех нитей GPU и прохождения заданного события, и только потом передает управление вызывающей программе. Прототип функции имеет вид:

cudaError_t cudaEventSynchronize(cudaEvent_t event),

где **event** – дескриптор события, прохождение которого ожидается.

Функция возвращает одно из следующих значений:

1. **cudaSuccess** – в случае успеха
2. **cudaErrorInitializationError** – ошибка инициализации
3. **cudaErrorPriorLaunchFailure** – ошибка при предыдущем асинхронном запуске функции
4. **cudaErrorInvalidValue** – неверное значение
5. **cudaErrorInvalidResourceHandle** – неверный дескриптор события.

В заключение выводим результат на экран и освобождаем выделенные ресурсы.

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

// Освобождаем ресурсы
cudaEventDestroy(syncEvent);
```

```
  cudaFree(devVec1);
  cudaFree(devVec2);
  cudaFree(devVec3);

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

Пример 2 программирования на CUDA

В качестве примера рассмотрим задачу умножения квадратной матрицы на вектор. Исходные данные:

A[n][n] – матрица размерности n x n;
b[n] – вектор, состоящий из n элементов.

Результат: c[n] – вектор из n элементов.
//Последовательный алгоритм умножения матрицы на вектор

```
for (i=0; i<n; i++)
{
      c[i]=0;
      for (j=0; j<n; j++)
      {
         c[i]+=A[i][j]*b[j];
      }
}
```

Теперь рассмотрим решение этой задачи на видеокарте. Следующий код иллюстрирует пример вызова функции CUDA:

```
// инициализация CUDA
if(!InitCUDA())   { return 0; }

int Size = 1000;
// обычные массивы в оперативной памяти
float *h_a,*h_b,*h_c;
h_a = new float[Size*Size];
h_b = new float[Size];
h_c = new float[Size];

for (int i=0; i<Size; i++)     // инициализация массивов a и b
{   for (int k=0; k<Size; k++)
   {         h_a[i*Size+k]=1;
   }
  h_b[i]=2;
}
```

```
// указатели на массивы в видеопамяти
float *d_a,*d_b,*d_c;

// выделение видеопамяти
cudaMalloc((void **)&d_a, sizeof(float)*Size*Size);
cudaMalloc((void **)&d_b, sizeof(float)*Size);
cudaMalloc((void **)&d_c, sizeof(float)*Size);



        // копирование из оперативной памяти в видеопамять
CUDA_SAFE_CALL(cudaMemcpy(d_a, h_a, sizeof(float)*Size*Size,
                        cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL(cudaMemcpy(d_b, h_b, sizeof(float)*Size,
                        cudaMemcpyHostToDevice) );

dim3 grid((Size+255)/256, 1, 1);                // установка количества блоков
dim3 threads(256, 1, 1);                        // установка количества потоков в блоке
MatrVectMul<<< grid, threads >>> (d_c, d_a, d_b,Size);          // вызов функции

// копирование из видеопамяти в оперативную память
CUDA_SAFE_CALL(cudaMemcpy(h_c, d_c, sizeof(float)*Size,
                        cudaMemcpyDeviceToHost) );

// освобождение памяти
CUDA_SAFE_CALL(cudaFree(d_a));
CUDA_SAFE_CALL(cudaFree(d_b));
CUDA_SAFE_CALL(cudaFree(d_c));
```

В принципе, структуру любой программы с использованием CUDA можно представить аналогично рассмотренному выше примеру. Таким образом, можно предложить следующую последовательность действий:

1. инициализация CUDA
2. выделение видеопамяти для хранения данных программы
3. копирование необходимых для работы функции данных из оперативной памяти в видеопамять
4. вызов функции CUDA
5. копирование возвращаемых данных из видеопамяти в оперативную память
6. освобождение видеопамяти

Пример функции, исполнимой на видеокарте

```
extern "C" __global__ void MatrVectMul(float *d_c, float *d_a,      float *d_b, int Size)
{       int i = blockIdx.x*blockDim.x+threadIdx.x;
        int k;

        d_c[i]=0;
        for (k=0; k<Size; k++)
          {    d_c[i]+=d_a[i*Size+k]*d_b[k];
          }
}
```

Здесь:

- threadIdx.x – идентификатор потока в блоке по координате x,
- blockIdx.x – идентификатор блока в гриде по координате x,
- blockDim.x – количество потоков в одном блоке.

Пока же следует запомнить, что таким образом получается уникальный идентификатор потока (в данном случае *i*), который можно использовать в программе, работая со всеми потоками как с одномерным массивом.

Важно помнить, что функция, предназначенная для исполнения на видеокарте, не должна обращать к оперативной памяти. Подобное обращение приведет к ошибке. Если необходимо работать с каким-либо объектом в оперативной памяти, предварительно его надо скопировать в видеопамять, и обращаться из функции CUDA к этой копии.

Среди основных особенностей CUDA следует отметить отсутствие поддержки двойной точности (типа double). Также для функций CUDA установлено максимальное время исполнения, отсутствует рекурсия, нельзя объявить функцию с переменным числом аргументов.

Функция, работающая на видеокарте, должна выполняться не более 1 секунды. Иначе, функция будет не завершена, и программа завершится с ошибкой.

Для синхронизации потоков в блоке существует функция __syncthreads(), которая ждет, пока все запущенные потоки отработают до этой точки. Функция __syncthreads() необходима, когда данные, обрабатываемые одним потоком, затем используются другими потоками.

Замер времени в CUDA

Для измерения времени выполнения отдельных частей программы и анализа скорости выполнения того или иного участка кода удобно использовать встроенные в CUDA функции по замеру времени.

```
unsigned int timer;
cutCreateTimer(&timer);          // создание таймера
cutStartTimer(timer);            // запуск таймера

… // код, время исполнения которого необходимо замерить

cutStopTimer( timer);            // остановка таймера
printf("time: %f (ms)\n", cutGetTimerValue(timer));          // получение времени
cutDeleteTimer( timer);          // удаление таймера
```

Создание проекта CUDA

После установки CUDA SDK, CUDA Toolkit и VS-интегратора в меню создания проекта появится новый пункт. Чтобы создать проект CUDA надо выбрать закладку Visual C++, CUDA, CUDAWinApp. Далее можно выбрать тип проекта (по умолчанию создается консольное) и другие параметры. В итоге будет создан проект, который выводит на экран строку «Hello CUDA!». Чтобы приложение могло запуститься, в текущем каталоге должен присутствовать файл cutil32D.dll для работы в режиме Debug и файл cutil32.dll для работы в режиме Release (например, для проекта с названием lab2 необходимо положить эти

файлы в директорию lab2/lab2). Причём, путь к файлу проекта не должен содержать русских букв.

Задание

Необходимо написать программу согласно варианту, при этом реализовать 2 функции: одну для выполнения на процессоре, вторую для выполнения на видеокарте. Затем сравнить результаты (возвращаемые значения) и скорость работы, задав большой размер матрицы (Size=1000). Массивы должны быть типа float или int.

Варианты заданий

1. Нахождение скалярного произведения векторов (протестировать для Size=1000000).
2. Нахождение суммы для каждой строки матрицы.
3. Вычисление среднего арифметического строк (столбцов) матрицы.
4. Умножение матрицы на множитель.
5. Сложение матриц.
6. Нахождение суммы квадратов элементов строк матрицы для каждой строки матрицы.
7. Вычисление матричного выражения A=B+k*C, где A, B, C - матрицы, k - скалярный множитель.
8. Умножение матрицы на матрицу.
9. Вычитание (поэлементное) матриц.
10. Транспонирование матрицы *.

Контрольные вопросы

1. Что подразумевается под терминами kernel, host и device?
2. Объясните понятия grid, thread block, thread.
3. На каких видеокартах доступна технология CUDA?
4. Как вы считаете, будут ли перечисленные выше задания выполняться быстрее для Size=5 на видеокарте, чем на процессоре и почему?