Иерархия памяти CUDA. Работа с текстурной памятью

Текстурная память – это вид памяти, предназначенной только для чтения, позволяющая повысить производительность и сократить трафик между процессором и памятью при определенных способах доступа. Изначально текстурная память предназначалась для традиционных задач обработки графики, но ее с успехом можно применять и в некоторых вычислительных приложениях GPU.

Текстурные кэши предназначены для графических приложений, в которых доступ к памяти характеризуется высокой ***пространственной локальностью***. В вычислительном приложении общего назначения это означает, что нить с большой вероятностью будет обращаться к адресам, расположенным "рядом" с адресами, к которым обращаются близкие нити (рис. 7.1).

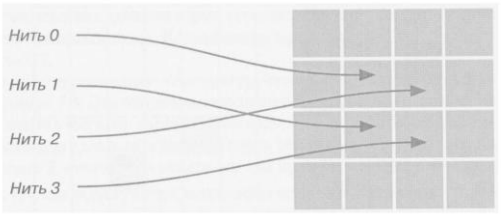

Рис. 7.1. Соответствие между нитями и двумерной областью памяти.

С точки зрения арифметики, четыре показанных на рисунке адреса не являются соседними, поэтому не будут кэшироваться вместе при использовании стандартной схемы кэширования, применяемой в CPU. Но текстурные кэши GPU специально разработаны для ускорения доступа с таких ситуациях, поэтому применение текстурной памяти вместо глобальной может дать выигрыш.

Моделирование теплообмена

Физические модели – одни из самых сложных вычислительных задач. В них надо находить компромисс между точностью и вычислительной сложностью. Но многие физические модели легко распараллеливаются, что позволяет увеличивать точность без существенного увеличения сложности при использовании методов параллельного программирования.

Простая модель теплообмена

Чтобы продемонстрировать ситуацию, в которой можно эффективно задействовать текстурную память, построим двумерную модель теплообмена. Пусть имеется прямоугольная комната, которую мы разобьем на одинаковые ячейки. Внутри получившейся сетки случайным образом расположены "нагреватели", имеющие разную температуру (рис. 7.2).

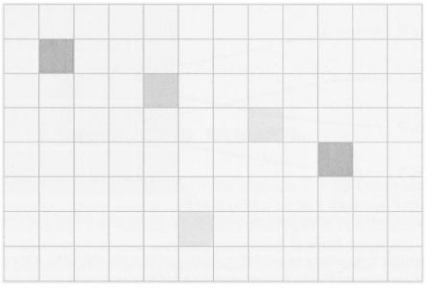

Рис. 7.2. Комната с разнотемпературными "нагревателями".

Имея такую сетку и зная расположение нагревателей, построим модель, которая позволит узнать, как изменяется температура в каждой ячейке со временем. Считаем для простоты, что

ячейки с нагревателями сохраняют постоянную температуру. На каждом шаге тепло "переносится" из каждой ячейки в соседние (рис. 7.3).

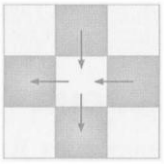

Рис. 7.3. Перенос тепла из теплых ячеек в холодные.

В предлагаемой модели новая температура ячейки вычисляется как сумма разностей между ее текущей температурой и температурами ее соседей:

$$T_{new} = T_{old} + \sum_{\text{по соседям}} k \cdot (T_{neighbor} - T_{old}) \qquad (7.1)$$

здесь k описывает скорость переноса тепла. При больших значениях k система быстрее перейдет в равновесное состояние. Т.к. мы рассматриваем только четырех соседей, а k и T_{old} постоянны, то формулу можно переписать в виде:

$$T_{new} = T_{old} + k \cdot (T_{top} + T_{bottom} + T_{left} + T_{right} - 4 \cdot T_{old}) \qquad (7.2)$$

К реальной физике распространения тепла эта формула отношения не имеет, но в качестве иллюстрации для работы GPU вполне подходит.

Обновление температур

В целом процедура обновления состояния выглядит следующим образом:

1. имея сетку входных температур, скопировать в нее ячейки с нагревателями с помощью функции copy_const_kernel();
2. имея сетку входных температур, вычислить выходные температуры по преобразованной формуле (функция blend_kernel());
3. обменять местами выходной и входной буферы, подготовив все для следующего шага моделирования, когда выходные данные текущего шага становятся входными данными для следующего шага.

Перед началом моделирования считаем, что мы уже сгенерировали сетку констант. В большинстве ячеек это нули, но в некоторых ячейках значения отличны от нуля – это нагреватели. Буфер констант не будет изменяться в процессе моделирования и считывается на каждом шаге.

Моделирование построено так, что новый шаг начинается с выходной сетки, построенной на предыдущем шаге. Затем копируем температуры ячеек с нагревателями, затирая ранее вычисленные значения. Это делается потому, что температуру нагревателей мы считаем постоянной. Копирование сетки констант во входную сетку выполняется следующим ядром.

```
__global__ void copy_const_kernel(float *iPtr, const float *cPtr)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = thredIdx.x + blockIdx.x * blockDim.x;
  int y = thredIdx.y + blockIdx.y * blockDim.y;

  int offset = x + y * blockDim.x * gridDim.x;

 if (cPtr[offset] != 0) iPtr[offset] = cPtr[offset];
}
```

В первых двух строках пара индексов thredIdx и blockIdx преобразуется в координаты x и y. В третьей строке вычисляется смещение от начала буферов констант и входных данных. В четвертой строке производится копирование температур нагревателей из массива cPtr[] во входную сетку, находящуюся в массиве iPtr[]. Температура копируется только в том случае, если она не равна 0, т.е. в данной точке есть нагреватель.

Шаг 2 алгоритма самый трудоемкий. Чтобы произвести обновление, можно выделить для обсчета каждой ячейки отдельную нить. Она будет считывать температуры "своей" ячейки и ее соседей и вычислять новое значение.

```
__global__ void  blend_kernel(float *outSrc, const float * inSrc)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = thredIdx.x + blockIdx.x * blockDim.x;
  int y = thredIdx.y + blockIdx.y * blockDim.y;

  int offset = x + y * blockDim.x * gridDim.x;

  int left = offset – 1;
  int right = offset + 1;
  if (x == 0) left++;
  if (x == DIM - 1) right--;

  int top = offset – DIM;
  int bottom = offset + DIM;

  if (y == 0) top += DIM;
  if (y == DIM - 1) bottom -= DIM;

  outSrc[offset] = inSrc[offset] + SPEED*( inSrc[top] + inSrc[bottom] + inSrc[left] +
                    inSrc[right] - inSrc[offset]*4);
}
```

Индексы thredIdx и blockIdx преобразуются в x, y и offset, но затем вместо цвета пикселя нить вычисляет температуру ячейки.

Далее вычисляются смещения всех четырех соседних ячеек, чтобы затем прочесть значения температуры в них. Эти значения понадобятся для вычисления новой температуры в текущей ячейке. Единственная сложность – граничные ячейки, для которых индексы следует подкорректировать, чтобы не выйти за границы сетки. И последний оператор выполняет вычисления по формуле (7.2).

Анимация моделирования

В оставшейся части программы мы инициализируем сетку, а затем отображаем анимированную карту распределения тепла. Рассмотрим этот код.

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_anim.h"

#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED 0.25f

// глобальные данные, необходимые функции обновления
```

```
struct DataBlock
{   unsigned char       *output_bitmap;
    float               *dev_inSrc;
    float               *dev_outSrc;
    float               *dev_constSrc;
    CPUAnimBitmap       *bitmap;
    cudaEvent_t         start, stop;
    float               totalTime;
    float               frames;
};

void anim_gpu (DataBlock *d, int ticks)
{ HANDLE_ERROR (cudaEventRecord (d->start,0));
  dim3 blocks(DIM/16, DIM/16);
  dim3 threads(16,16);
  CPUAnimBitmap * bitmap = d-> bitmap;
  for (int i=0; i<90; i++)
    {  copy_const_kernel<<<blocks, threads>>>( d->dev_inSrc, d-> dev_constSrc );
       blend_kernel<<<blocks, threads>>>( d-> dev_outSrc, d-> dev_inSrc );
       swap ( d-> dev_inSrc, d-> dev_outSrc );
    }
    float_to_color<<<blocks, threads>>>( d->output_bitmap, d->dev_inSrc );
  HANDLE_ERROR (cudaMemcpy (bitmap->get_ptr(), d->output_bitmap,
                            bitmap->image_size(), cudaMemcpyDeviceToHost));
  HANDLE_ERROR (cudaEventRecord (d->stop,0));
  HANDLE_ERROR (cudaEventSynchronize (d->stop,0));
  float elapsedTime;
  HANDLE_ERROR (cudaEventElapsedTime (&elapsedTime, d->start, d->stop));
  d->totalTime += elapsedTime;
  ++d->frames;
  printf("Среднее время на один кадр: %3.1f ms\n", d->totalTime / d->frames);
}


void anim_exit(DataBlock *d)
{  cudaFree (d->dev_inSrc);
   cudaFree (d->dev_outSrc);
   cudaFree (d->dev_constSrc);
   HANDLE_ERROR (cudaEventDestroy (d->start));
   HANDLE_ERROR (cudaEventDestroy (d->stop));
}
```

Каркас анимации вызывает функцию anim_gpu() в каждом кадре. В качестве аргументов ей передается указатель на структуру DataBlock и количество уже прошедших тактов анимации. Используются блоки из 256 нитей, организованные в сетку 16x16. На каждой итерации цикла for в функции anim_gpu() вычисляется один временной шаг моделирования, описанный в разделе "Обновление температур". Поскольку структура DataBlock содержит буфер постоянных температур нагревателей, а также результат последнего шага моделирования, то она инкапсулирует все состояние анимации, следовательно anim_gpu() вообще не нуждается в переменной ticks.

В одном кадре выполняется 90 шагов моделирования – это величина, которая была экспериментально выбрана в качестве компромисса между необходимостью выгружать растр на каждом шаге и вычислением слишком большого числа шагов в одном кадре. Если нужно наблюдение за результатом каждого шага моделирования, то функцию можно изменить так, чтобы в каждом кадре вычислялся ровно один шаг.

После вычисления 90 шагов моделирования с момента окончания предыдущего кадра функция anim_gpu() готова скопировать растровое изображение текущего кадра назад в память CPU. Поскольку цикл for оставил входной и выходной буферы переставленными, следующему ядру передается входной буфер, который в действительности содержит результат, получившийся после 90-го шага моделирования. Температуры преобразуются в цвета с помощью ядра float_to_color(), а затем выполняется копирование результирующего изображения в память CPU, вызывая функцию cudaMemcpy() с параметром cudaMemcpyDeviceToHost. Затем, чтобы подготовиться к следующей последовательности шагов моделирования, обмениваем входной буфер с выходным, т.к. последний будет выступать в роли входного на следующем шаге моделирования.

```
int main(void)
{ DataBlock data;
  CPUAnimBitmap bitmap(DIM, DIM, &data);
  data.bitmap = & bitmap;
  data.totalTime = 0;
  data.frames = 0;
  HANDLE_ERROR (cudaEventCreate (&data.start));
  HANDLE_ERROR (cudaEventCreate (&data.stop));
  HANDLE_ERROR (cudaMalloc ((void**)&data.output_bitmap, bitmap.image_size()));

        // предполагаем, что размер float равен 4 байтам (т.е. rgba)
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_inSrc, bitmap.image_size()));
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_outSrc, bitmap.image_size()));
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_constSrc, bitmap.image_size()));
  float *temp = (float*)malloc(bitmap.image_size());
  for (int i = 0; i<DIM*DIM; i++)
     { temp[i] = 0;
       int x = i % DIM;
       int y = i / DIM;
       if ((x>300) && (x<600) && (y>310) && (y<601))
                temp[i] = MAX_TEMP;
     }

  temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP) / 2;
  temp[DIM*700+100] = MIN_TEMP;
  temp[DIM*300+300] = MIN_TEMP;
  temp[DIM*200+700] = MIN_TEMP;
  for (int y = 800; y < 900; y++)
    {  for (int x = 400; x < 500; x++)
          temp[x+y*DIM] = MIN_TEMP;
    }
  HANDLE_ERROR (cudaMemcpy (data.dev_constSrc, temp, bitmap.image_size(),
                            cudaMemcpyHostToDevice));
  for (int y = 800; y < DIM; y++)
    {  for (int x = 0; x < 200; x++)
          temp[x+y*DIM] = MAX_TEMP;
    }
  HANDLE_ERROR (cudaMemcpy (data.dev_inSrc, temp, bitmap.image_size(),
                            cudaMemcpyHostToDevice));
  free(temp);
  bitmap.anim_and_exit ( (void (*)(void*, int)) anim_gpu,
                           (void (*)(void*)) anim_exit);
}
```

На рис. 7.4 показано, как может выглядеть картина теплообмена. На рисунке видны нагреватели в виде точек размером в пиксель, которые нарушают непрерывность распределения температур.

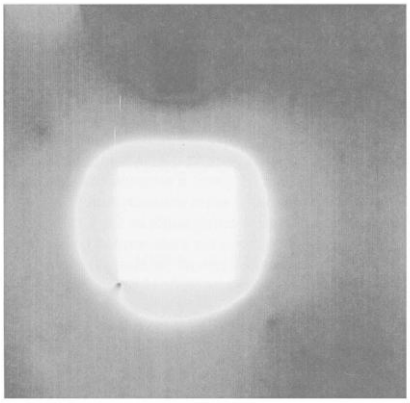

Рис. 7.4. Снимок экрана, полученный в процессе моделирования теплообмена.

Применение текстурной памяти

Доступ к памяти в алгоритме пересчета температур на каждом шаге характеризуется *пространственной локальностью*. Для ускорения работы в этом случае используется текстурная память. Но прежде, чем воспользоваться ей, нужно понять механизм ее работы.

Сначала необходимо объявить входные данные как ссылки на текстуры. Мы используем текстуры с плавающей точкой, потому что именно в таком виде представлены данные о температуре.

```
// эти данные находятся в памяти GPU
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
```

Следующая особенность заключается в том, что после выделения памяти GPU для всех трех буферов мы должны "привязать" эти ссылки к буферам в памяти с помощью функции cudaBindTexture() (в версии CUDA 3.2 и выше механизм работы механизм работы с текстурной памятью был несколько изменен). Тем самым исполняющей среде сообщаются две вещи:

- что указанный буфер будет использоваться в качестве текстуры;
- что указанная ссылка будет использоваться в качестве "имени" текстуры.

В программе моделирования теплообмена после трех операций выделения памяти "привязываем" ко всем трем областям ранее объявленные ссылки (texConstSrc, texIn, texOut):

```
HANDLE_ERROR (cudaMalloc ((void**)&data.dev_inSrc, image_size));
HANDLE_ERROR (cudaMalloc ((void**)&data.dev_outSrc, image_size));
HANDLE_ERROR (cudaMalloc ((void**)&data.dev_constSrc, image_size));

HANDLE_ERROR (cudaBindTexture (NULL, texConstSrc, data.dev_constSrc, image_size));
HANDLE_ERROR (cudaBindTexture (NULL, texIn, data.dev_inSrc, image_size));
HANDLE_ERROR (cudaBindTexture (NULL, texOut, data.dev_outSrc, image_size));
```

Теперь текстуры полностью настроены, и можно запустить ядро. Но при чтении данных из текстур внутри ядра необходимо вызывать специальные функции, которые сообщают GPU, что

запросы должны проходить через текстурный блок, а не через глобальную память. Поэтому для чтения из буферов обычные квадратные скобки уже не годятся; нужно так модифицировать blend_kernel(), чтобы для чтения из памяти вызывалась функция text1Dfetch().

Между глобальной и текстурной памятью имеется еще одно различие, вынуждающее сделать дополнительное изменение. Хотя text1Dfetch() выглядит как вызов функции, на самом деле ее код генерирует компилятор. А поскольку ссылки на текстуры должны быть объявлены глобально в области видимости файла, то нельзя больше передавать входной и выходной буферы в виде параметров blend_kernel(), потому что еще на этапе компиляции компилятор должен знать, к каким текстурам будет обращаться text1Dfetch(). Вместо того, чтобы передавать указатели на входной и выходной буферы, будем передавать blend_kernel() булевский флаг dstOut, который показывает, какой буфер является входным, какой – выходным. Ниже изменения в коде blend_kernel() выделены полужирным шрифтом:

```
__global__ void  blend_kernel(float *dst, bool dstOut)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = thredIdx.x + blockIdx.x * blockDim.x;
  int y = thredIdx.y + blockIdx.y * blockDim.y;

  int offset = x + y * blockDim.x * gridDim.x;

  int left = offset – 1;
  int right = offset + 1;
  if (x == 0) left++;
  if (x == DIM - 1) right--;

  int top = offset – DIM;
  int bottom = offset + DIM;

  if (y == 0) top += DIM;
  if (y == DIM - 1) bottom -= DIM;

  float t, l, c, r, b;
  if (dstOut)
    { t = text1Dfetch(texIn, top);
      l = text1Dfetch(texIn, left);
      c = text1Dfetch(texIn, offset);
      r = text1Dfetch(texIn, right);
      b = text1Dfetch(texIn, bottom);
    }
      else
         { t = text1Dfetch(texOut, top);
           l = text1Dfetch(texOut, left);
           c = text1Dfetch(texOut, offset);
           r = text1Dfetch(texOut, right);
           b = text1Dfetch(texOut, bottom);
         }

  dst[offset] = c + SPEED*( t + b + r + l – 4*c);
}
```

Ядро copy_const_kernel() читает из буфера, в котором хранятся позиции и температуры нагревателей, поэтому необходимо проделать аналогичное изменение, чтобы чтение производилось из текстурной, а не из глобальной памяти:

```
__global__ void copy_const_kernel(float *iPtr)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = thredIdx.x + blockIdx.x * blockDim.x;
  int y = thredIdx.y + blockIdx.y * blockDim.y;

  int offset = x + y * blockDim.x * gridDim.x;

  float c = text1Dfetch(texConstSrc, offset);

  if (c != 0) iPtr[offset] = c;
}
```

Так как функция blend_kernel() теперь принимает флаг, описывающий назначение буферов, то необходимо внести соответствующее исправление в функцию anim_gpu(). Вместо того, чтобы менять местами сами буферы, меняем значение флага на противоположное – dstOut = ! dstOut – после каждой серии вызовов:

```
void anim_gpu (DataBlock *d, int ticks)
{ HANDLE_ERROR (cudaEventRecord (d->start,0));
  dim3 blocks(DIM/16, DIM/16);
  dim3 threads(16,16);
  CPUAnimBitmap * bitmap = d-> bitmap;
// так как текстура глобальная и "привязанная", то используем флаг, который указывает,
// какой буфер на данной итерации является входным, а какой – выходным.
  volatile bool dstOut = true;
  for (int i=0; i<90; i++)
    {  float *in, *out;
       if (dstOut)
          { in = d->dev_inSrc;
            out = d->dev_outSrc;
          }
        else
            { out = d->dev_inSrc;
              in = d->dev_outSrc;
            }
       copy_const_kernel<<<blocks, threads>>>( in );
       blend_kernel<<<blocks, threads>>>( out, dstOut );
       dstOut = ! dstOut;
    }
  float_to_color<<<blocks, threads>>>( d->output_bitmap, d->dev_inSrc );
  HANDLE_ERROR (cudaMemcpy (bitmap->get_ptr(), d->output_bitmap,
                            bitmap->image_size(), cudaMemcpyDeviceToHost));
  HANDLE_ERROR (cudaEventRecord (d->stop,0));
  HANDLE_ERROR (cudaEventSynchronize (d->stop,0));
  float elapsedTime;
  HANDLE_ERROR (cudaEventElapsedTime (&elapsedTime, d->start, d->stop));
  d->totalTime += elapsedTime;
  ++d->frames;
  printf("Среднее время на один кадр: %3.1f ms\n", d->totalTime / d->frames);
}
```

И последнее изменение касается очистки в конце работы приложения. Нужно не только освобождать глобальные буферы, но и "отвязать" текстуры:

```
// очистить память, выделенную на GPU
void anim_exit(DataBlock *d)
```

```
{  cudaUnbindTexture (texIn);
   cudaUnbindTexture (texOut);
   cudaUnbindTexture (texConstSrc);

   cudaFree (d->dev_inSrc);
   cudaFree (d->dev_outSrc);
   cudaFree (d->dev_constSrc);

   HANDLE_ERROR (cudaEventDestroy (d->start));
   HANDLE_ERROR (cudaEventDestroy (d->stop));
}
```

Использование двумерной текстурной памяти

Есть много случаев, когда полезно иметь двумерную область памяти. Посмотрим, как можно модифицировать программу расчета теплообмена с применением двумерных текстур.

Во-первых, необходимо изменить объявление ссылок на текстуры. По умолчанию ссылки на текстуры одномерны, поэтому добавляем аргумент 2, задающий размерность.

```
texture<float, 2> texConstSrc;
texture<float, 2> texIn;
texture<float, 2> texOut;
```

Упрощение, связанное с переходом на двумерные текстуры, проявляется в функции blend_kernel(). Нужно заменить обращения к text1Dfetch() на text2D(), но зато больше не нужна переменная offset, которая применялась для вычисления линейных смещений top, left, right и bottom, т.к. двумерную текстуру можно адресовать непосредственно с помощью координат x и y.

Далее. После перехода на text2D() уже не нужно беспокоиться о выходе за границы массива. Если x или y меньше нуля, то text2D() вернет значение соответствующей координаты, равной нулю, если какая-то координата больше ширины, то text2D() вернет значение соответствующей координаты, равной ширине. Но нужно помнить, что если в данном приложении такое поведение идеально, то в других оно может оказаться неприемлемо.

```
__global__ void  blend_kernel(float *dst, bool dstOut)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = thredIdx.x + blockIdx.x * blockDim.x;
  int y = thredIdx.y + blockIdx.y * blockDim.y;

  int offset = x + y * blockDim.x * gridDim.x;

  float t, l, c, r, b;
  if (dstOut)
    { t = text2D(texIn, x, y - 1);
      l = text2D(texIn, x – 1, y);
      c = text2D(texIn, x, y);
      r = text2D(texIn, x + 1, y);
      b = text2D(texIn, x, y + 1);
    }
      else
        { t = text2D(texOut, x, y - 1);
          l = text2D(texOut, x – 1, y);
          c = text2D(texOut, x, y);
          r = text2D(texOut, x + 1, y);
```

```
        b = text2D(texOut, x, y + 1);
      }

  dst[offset] = c + SPEED*( t + b + r + l – 4*c);
}
```

Т.к. все предыдущие обращения к text1Dfetch() нужно заменить на обращения к text2D(), то соответствующие изменения вносятся и в ядро copy_const_kernel(). Как и в blend_kernel(), больше нет необходимости использовать смещение для адресации текстуры, это можно сделать с помощью самих координат x и y:

```
__global__ void copy_const_kernel(float *iPtr, const float *cPtr)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = thredIdx.x + blockIdx.x * blockDim.x;
  int y = thredIdx.y + blockIdx.y * blockDim.y;

  int offset = x + y * blockDim.x * gridDim.x;

  float c = tex2D(texConstSrc, x, y);

  if (c != 0) iPtr[offset] = c;
}
```

Далее, в функции main() следует изменить "привязку", сообщив исполняющей среде, что буфер следует рассматривать как двумерную, а не одномерную текстуру:

```
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_inSrc, imageSize));
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_outSrc, imageSize));
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_constSrc, imageSize));

  cudaChannelFormatDesc desc =   cudaChannelFormatDesc<float>();

  HANDLE_ERROR (cudaBindTexture2D (NULL, texConstSrc, data.dev_constSrc, desc, DIM,
                                   DIM, sizeof(float)*DIM));
  HANDLE_ERROR (cudaBindTexture2D (NULL, texIn, data.dev_inSrc, desc, DIM,
                                   DIM, sizeof(float)*DIM));
  HANDLE_ERROR (cudaBindTexture2D (NULL, texOut, data.dev_outSrc, desc, DIM,
                                   DIM, sizeof(float)*DIM));
```

Как и в версиях без текстур и с одномерной текстурой, вначале выделяем память для входных массивов. От одномерной версии этот код отличается тем, что в случае привязки двумерных текстур исполняющая среда CUDA требует передачи *cudaChannelFormatDesc* при привязке текстур. В листинге имеется объявление дескриптора формата канала. В данном случае можно принять параметры по умолчанию и указать лишь то, что требуется дескриптор с плавающей точкой. Затем с помощью функции *cudaBindTexture2D()* "привязываем" все три входных буфера как двумерные текстуры, передавая размеры текстуры (DIM x DIM) и дескриптор формата канала (desc). Больше ничего не изменяется.

```
int main(void)
{ DataBlock data;
  CPUAnimBitmap bitmap(DIM, DIM, &data);
  data.bitmap = &bitmap;
```

```
  data.totalTime = 0;
  data.frames = 0;
  HANDLE_ERROR (cudaEventCreate (&data.start));
  HANDLE_ERROR (cudaEventCreate (&data.stop));

  int imageSize = bitmap.image_size();

  HANDLE_ERROR (cudaMalloc ((void**)&data.output_bitmap, imageSize));
        // предполагаем, что размер float равен 4 байтам (т.е. rgba)
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_inSrc, imageSize));
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_outSrc, imageSize));
  HANDLE_ERROR (cudaMalloc ((void**)&data.dev_constSrc, imageSize));

cudaChannelFormatDesc desc =   cudaChannelFormatDesc<float>();

  HANDLE_ERROR (cudaBindTexture2D (NULL, texConstSrc, data.dev_constSrc, desc, DIM,
                                                DIM, sizeof(float)*DIM));
  HANDLE_ERROR (cudaBindTexture2D (NULL, texIn, data.dev_inSrc, desc, DIM,
                                                DIM, sizeof(float)*DIM));
  HANDLE_ERROR (cudaBindTexture2D (NULL, texOut, data.dev_outSrc, desc, DIM,
                                                DIM, sizeof(float)*DIM));

        // инициализировать константные данные
 float *temp = (float*)malloc(imageSize);
  for (int i = 0; i<DIM*DIM; i++)
     { temp[i] = 0;
       int x = i % DIM;
       int y = i / DIM;
       if ((x>300) && (x<600) && (y>310) && (y<601))
                temp[i] = MAX_TEMP;
     }

  temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP) / 2;
  temp[DIM*700+100] = MIN_TEMP;
  temp[DIM*300+300] = MIN_TEMP;
  temp[DIM*200+700] = MIN_TEMP;
  for (int y = 800; y < 900; y++)
    {  for (int x = 400; x < 500; x++)
          temp[x+y*DIM] = MIN_TEMP;
    }
  HANDLE_ERROR (cudaMemcpy (data.dev_constSrc, temp, imageSize,
                          cudaMemcpyHostToDevice));
        // инициализировать входные данные
  for (int y = 800; y < DIM; y++)
    {  for (int x = 0; x < 200; x++)
          temp[x+y*DIM] = MAX_TEMP;
    }
  HANDLE_ERROR (cudaMemcpy (data.dev_inSrc, temp, imageSize,
                          cudaMemcpyHostToDevice));
  free(temp);
  bitmap.anim_and_exit ( (void (*)(void*, int)) anim_gpu,
                          (void (*)(void*)) anim_exit);
}
```

Хотя для "привязки" одномерных и двумерных текстур нужны разные функции, для "отвязывания" используется одна и та же функция cudaUnbindTexture(). Поэтому функция очистки остается без изменения:

```
// очистить память, выделенную на GPU
void anim_exit(DataBlock *d)
{  cudaUnbindTexture (texIn);
   cudaUnbindTexture (texOut);
   cudaUnbindTexture (texConstSrc);

   cudaFree (d->dev_inSrc);
   cudaFree (d->dev_outSrc);
   cudaFree (d->dev_constSrc);

   HANDLE_ERROR (cudaEventDestroy (d->start));
   HANDLE_ERROR (cudaEventDestroy (d->stop));
}
```

У версии программы моделирования теплообмена с двумерными текстурами практически такие же характеристики производительности, как и у версии с одномерными текстурами. Поэтому с точки зрения производительности решение о выборе одномерной или двумерной текстуры, скорее всего, несущественно. В данном конкретном приложении применение двумерных текстур позволило немного упростить код, поскольку сама предметная область двумерна. Но в общем случае так бывает не всегда, поэтому выбор размерности текстур следует осуществлять с учетом особенностей конкретной задачи.

Лабораторное задание.

Отладить три варианта программы моделирования – без текстурной памяти, с одномерной текстурной памятью и с двумерной текстурной памятью.

Сравнить результаты работы всех трех вариантов (картинки на экране ☺) при одинаковых входных данных

Сравнить времена выполнения программ при одинаковых входных данных.