Лабораторная работа №7

Выполнил Мацур Дмитрий ст.грп ПМ-21М

Реализация без текстурной памяти.

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

constexpr size_t DIM = 1024;
constexpr float PI = 3.1415926535897932f;
constexpr float MAX_TEMP = 1.0f;
constexpr float MIN_TEMP = 0.0001f;
constexpr float SPEED =  0.25f;

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

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

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

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 );
  cudaMemcpy (bitmap->get_ptr(), d->output_bitmap,
              bitmap->image_size(), cudaMemcpyDeviceToHost);
  cudaEventRecord (d->stop,0);
  cudaEventSynchronize(d->stop);

  float elapsedTime;
  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);
  cudaEventDestroy (d->start);
  cudaEventDestroy (d->stop);
}

int main(void)
{
  DataBlock data;
  CPUAnimBitmap bitmap(DIM, DIM, &data);
  data.bitmap = & bitmap;
  data.totalTime = 0;
  data.frames = 0;

  cudaEventCreate (&data.start);
  cudaEventCreate (&data.stop);

  HANDLE_ERROR (cudaMalloc ((void**)&data.output_bitmap, bitmap.image_size()));
  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()));
  auto *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);
}
```

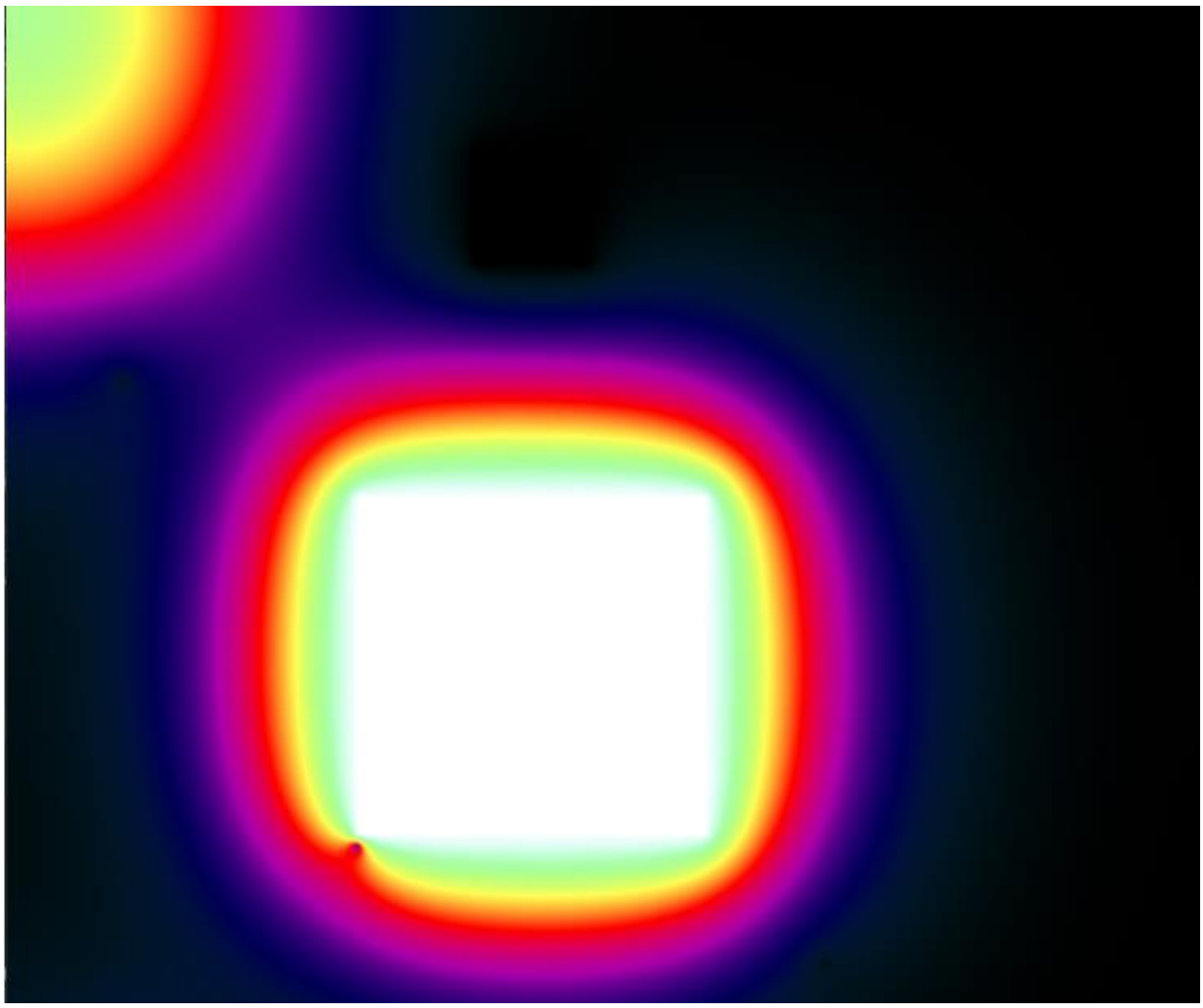

Рис. 1: Пример реализации без текстурной памяти

С одномерной текстурной памятью

```
#include <cuda.h>
#include <cuda_runtime.h>
#include "common/book.h"
#include "common/cpu_anim.h"

constexpr size_t DIM = 1024;
constexpr float PI = 3.1415926535897932f;
constexpr float MAX_TEMP = 1.0f;
constexpr float MIN_TEMP = 0.0001f;
constexpr float SPEED =  0.25f;

texture<float> texConstSrc;
texture<float> texIn;
```

```
texture<float> texOut;

// глобальные данные, необходимые функции обновления

__global__ void copy_const_kernel(float *iPtr)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x;
  float c = tex1Dfetch<float>(texConstSrc, offset);
  if (c != 0) iPtr[offset] = c;
}

__global__ void blend_kernel(float *dst, bool dstOut)
{
  // отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.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 = tex1Dfetch<float>(texIn, top);
    l = tex1Dfetch<float>(texIn, left);
    c = tex1Dfetch<float>(texIn, offset);
    r = tex1Dfetch<float>(texIn, right);
    b = tex1Dfetch<float>(texIn, bottom);
  }
  else
  {
    t = tex1Dfetch<float>(texOut, top);
    l = tex1Dfetch<float>(texOut, left);
    c = tex1Dfetch<float>(texOut, offset);
    r = tex1Dfetch<float>(texOut, right);
    b = tex1Dfetch<float>(texOut, bottom);
  }
  dst[offset] = c + SPEED*( t + b + r + l -4*c);
}


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;
// 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));
  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));
}

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()));

  HANDLE_ERROR (cudaBindTexture (nullptr, texConstSrc, data.dev_constSrc, bitmap.image_
size()));
  HANDLE_ERROR (cudaBindTexture (nullptr, texIn, data.dev_inSrc, bitmap.image_size()));
  HANDLE_ERROR (cudaBindTexture (nullptr, texOut, data.dev_outSrc, bitmap.image_size())
);

  auto *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);
}
```

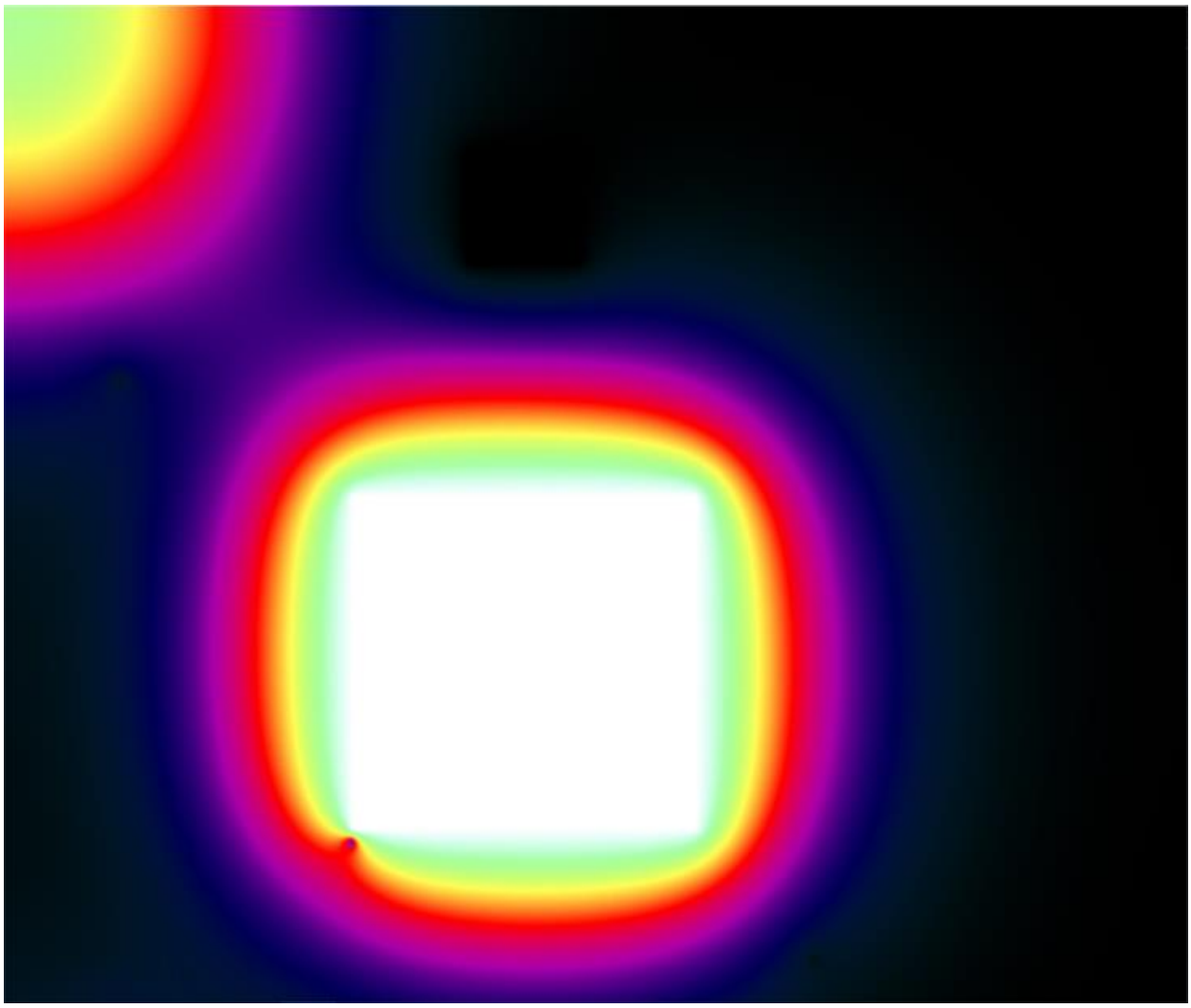

Рис. 2: Пример реализации с одномерной текстурной памятью

С двумерной текстурной памятью

```
#include "cuda.h"
#include "common/book.h"
#include "common/cpu_anim.h"
constexpr size_t DIM = 1024;
constexpr float PI = 3.1415926535897932f;
constexpr float MAX_TEMP = 1.0f;
constexpr float MIN_TEMP = 0.0001f;
constexpr float SPEED =  0.25f;

texture<float, 2> texConstSrc;
```

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

// глобальные данные, необходимые функции обновления

__global__ void copy_const_kernel(float *iPtr, const float *cPtr)
{// отобразить пару thredIdx/blockIdx на позицию пикселя
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x;
  float c = tex2D<float>(texConstSrc, x, y);

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

__global__ void blend_kernel(float *dst, bool dstOut)
{// отобразить пару threadIdx/blockIdx на позицию пикселя
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x;
  float t, l, c, r, b;
  if (dstOut)
  {
    t = tex2D<float>(texIn, x, y - 1);
    l = tex2D<float>(texIn, x - 1, y);
    c = tex2D<float>(texIn, x, y);
    r = tex2D<float>(texIn, x + 1, y);
    b = tex2D<float>(texIn, x, y + 1);
  }
  else
  {
    t = tex2D<float>(texOut, x, y - 1);
    l = tex2D<float>(texOut, x - 1, y);
    c = tex2D<float>(texOut, x, y);
    r = tex2D<float>(texOut, x + 1, y);
    b = tex2D<float>(texOut, x, y + 1);
  }
  dst[offset] = c + SPEED*( t + b + r + l - 4*c);
}


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));
  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));
}

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()));

  cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
  HANDLE_ERROR (cudaBindTexture2D (nullptr, texConstSrc, data.dev_constSrc, desc, DIM,
                                   DIM, sizeof(float)*DIM));
  HANDLE_ERROR (cudaBindTexture2D (nullptr, texIn, data.dev_inSrc, desc, DIM,
                                   DIM, sizeof(float)*DIM));
  HANDLE_ERROR (cudaBindTexture2D (nullptr, texOut, data.dev_outSrc, desc, DIM,
                                   DIM, sizeof(float)*DIM));

  auto *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);
}
```

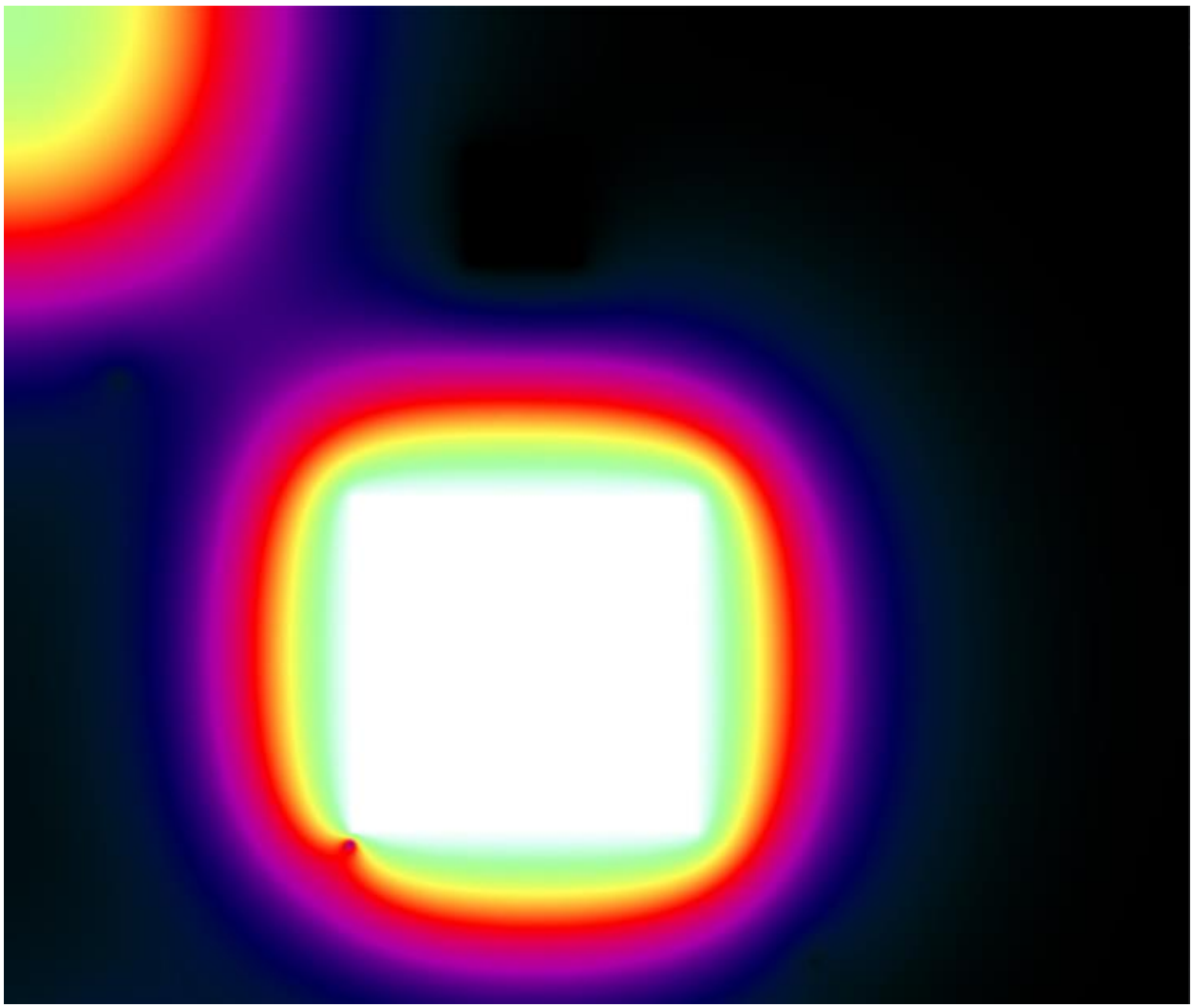

Рис. 3: Пример реализации с трехмерной текстурной памятью

Сравнение вермени выполнения

Среднее время 200000 итераций. Размерность: 2048
No texture: 20675.7 ms
1-D: 20431.5 ms
2-D: 16701.6 ms

Таким образом, наше предположение о том, что для двумерной задачи эффективнее использовать двумерную сетку оправдалось.