In [1]:

```
!nvidia-smi
!nvcc --version
```

```
Sat Jan 10 16:19:54 2026
+-----------------------------------------------------------------------------------------
---+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4
|
|-----------------------------------------+------------------------+----------------------
---+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. EC
C |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M.
|
|                                         |                        |               MIG
M. |
|=========================================+========================+======================
=|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |
0 |
| N/A   63C    P8             11W /   70W |       0MiB /  15360MiB |      0%      Defaul
t |
|                                         |                        |                   N
/A |
+-----------------------------------------+------------------------+----------------------
---+



+-----------------------------------------------------------------------------------------
---+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                              GPU Memor
y |
|        ID   ID                                                               Usage
|
|=========================================================================================
=|
|  No running processes found
|
+-----------------------------------------------------------------------------------------
---+
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Jun__6_02:18:23_PDT_2024
Cuda compilation tools, release 12.5, V12.5.82
Build cuda_12.5.r12.5/compiler.34385749_0
```

filter

cpu + gpu

In [2]:

```
# @title
%%file /tmp/t2v1.cu

#include <stdio.h>
#include <cuda_runtime.h>
#include <stdlib.h>
#include <chrono>
#include <math.h>

#define _DEBUGY 1
```

```
#ifdef _DEBUGY
#define CHECK_ERR(err, a) { if (err != cudaSuccess) { \
    printf("%s(%d): %s\n", __FILE__, __LINE__, a); \
    } \
}
#else
#define CHECK_ERR(err, a) {}
#endif

// SoA: separate arrays for R, G, B
// Input: r_in[], g_in[], b_in[]
// Output: r_out[], g_out[], b_out[]

__global__ void blurKernelSoA(
    const float* __restrict__ r_in,//restrict  hint to the compiler that this pointer is
the only way to access the data it points to during the scope of the function.
    const float* __restrict__ g_in,
    const float* __restrict__ b_in,
    float* __restrict__ r_out,
    float* __restrict__ g_out,
    float* __restrict__ b_out,
    int width, int height)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    auto clamp = [](int v, int lo, int hi) -> int {
        return (v < lo) ? lo : (v > hi ? hi : v);
    };

    float sum_r = 0.0f, sum_g = 0.0f, sum_b = 0.0f;
    int count = 0;

    for (int dy = -1; dy <= 1; dy++) {
        for (int dx = -1; dx <= 1; dx++) {
            int nx = clamp(x + dx, 0, width - 1);
            int ny = clamp(y + dy, 0, height - 1);
            int idx = ny * width + nx;

            sum_r += r_in[idx];
            sum_g += g_in[idx];
            sum_b += b_in[idx];
            count++;
        }
    }

    int out_idx = y * width + x;
    r_out[out_idx] = sum_r / count;
    g_out[out_idx] = sum_g / count;
    b_out[out_idx] = sum_b / count;
}

// CPU version (SoA)
void cpuBlurSoA(
    const float* r_in, const float* g_in, const float* b_in,
    float* r_out, float* g_out, float* b_out,
    int width, int height)
{
    auto clamp = [](int v, int lo, int hi) -> int {
        return (v < lo) ? lo : (v > hi ? hi : v);
    };

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float sum_r = 0.0f, sum_g = 0.0f, sum_b = 0.0f;
            int count = 0;

            for (int dy = -1; dy <= 1; dy++) {
                for (int dx = -1; dx <= 1; dx++) {
                    int nx = clamp(x + dx, 0, width - 1);
                    int ny = clamp(y + dy, 0, height - 1);
                    int idx = ny * width + nx;

                    sum_r += r_in[idx];
                    sum_g += g_in[idx];
                    sum_b += b_in[idx];
                    count++;
                }
            }

            int out_idx = y * width + x;
            r_out[out_idx] = sum_r / count;
            g_out[out_idx] = sum_g / count;
            b_out[out_idx] = sum_b / count;
        }
    }
}

//=======================================================================================
======

int main() {
    //const int WIDTH = 512;
    //const int HEIGHT = 512;
    //const int WIDTH = 10240;
    //const int HEIGHT = 10240;
    const int WIDTH = 1024;
    const int HEIGHT = 1024;
    const int NUM_PIXELS = WIDTH * HEIGHT;
    size_t size = NUM_PIXELS * sizeof(float);

    printf("SoA Blur: Image size %dx%d (%d pixels)\n", WIDTH, HEIGHT, NUM_PIXELS);

    // Timers
    cudaEvent_t t_start, t_stop;
    cudaEventCreate(&t_start);
    cudaEventCreate(&t_stop);

    // Host memory (pinned)
    float *h_r_in, *h_g_in, *h_b_in;
    float *h_r_out_gpu, *h_g_out_gpu, *h_b_out_gpu;
    cudaMallocHost(&h_r_in, size);
    cudaMallocHost(&h_g_in, size);
    cudaMallocHost(&h_b_in, size);
    cudaMallocHost(&h_r_out_gpu, size);
    cudaMallocHost(&h_g_out_gpu, size);
    cudaMallocHost(&h_b_out_gpu, size);

    float *h_r_out_cpu = (float*)malloc(size);
    float *h_g_out_cpu = (float*)malloc(size);
    float *h_b_out_cpu = (float*)malloc(size);

    if (!h_r_in || !h_g_in || !h_b_in ||
        !h_r_out_gpu || !h_g_out_gpu || !h_b_out_gpu ||
        !h_r_out_cpu || !h_g_out_cpu || !h_b_out_cpu) {
        printf("Host alloc failed!\n");
        return -1;
    }

    // Device memory
    float *d_r_in, *d_g_in, *d_b_in;
    float *d_r_out, *d_g_out, *d_b_out;
    cudaMalloc(&d_r_in, size);
    cudaMalloc(&d_g_in, size);
    cudaMalloc(&d_b_in, size);
    cudaMalloc(&d_r_out, size);
    cudaMalloc(&d_g_out, size);
    cudaMalloc(&d_b_out, size);
    CHECK_ERR(cudaGetLastError(), "cudaMalloc device");

    // Initialize input
    srand(42);
    for (int i = 0; i < NUM_PIXELS; i++) {
        h_r_in[i] = (float)rand() / RAND_MAX * 255.0f;
        h_g_in[i] = (float)rand() / RAND_MAX * 255.0f;
        h_b_in[i] = (float)rand() / RAND_MAX * 255.0f;
    }

    // --- CPU blur ---
    auto tcpu_start = std::chrono::high_resolution_clock::now();
    cpuBlurSoA(h_r_in, h_g_in, h_b_in,
               h_r_out_cpu, h_g_out_cpu, h_b_out_cpu,
               WIDTH, HEIGHT);
    auto tcpu_stop = std::chrono::high_resolution_clock::now();
    float cpu_ms = std::chrono::duration_cast<std::chrono::microseconds>(tcpu_stop - tcp
u_start).count() / 1000.0f;
    printf("CPU time: %.3f ms\n", cpu_ms);

    // --- GPU blur ---
    cudaMemcpy(d_r_in, h_r_in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_g_in, h_g_in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b_in, h_b_in, size, cudaMemcpyHostToDevice);
    CHECK_ERR(cudaGetLastError(), "H2D input");

    dim3 blockSize(16, 16);
    dim3 gridSize((WIDTH + blockSize.x - 1) / blockSize.x,
                  (HEIGHT + blockSize.y - 1) / blockSize.y);

    cudaEventRecord(t_start);
    blurKernelSoA<<<gridSize, blockSize>>>(
        d_r_in, d_g_in, d_b_in,
        d_r_out, d_g_out, d_b_out,
        WIDTH, HEIGHT);
    CHECK_ERR(cudaGetLastError(), "blurKernelSoA launch");

    cudaMemcpy(h_r_out_gpu, d_r_out, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_g_out_gpu, d_g_out, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_b_out_gpu, d_b_out, size, cudaMemcpyDeviceToHost);
    CHECK_ERR(cudaGetLastError(), "D2H output");

    cudaEventRecord(t_stop);
    cudaEventSynchronize(t_stop);
    float gpu_ms = 0;
    cudaEventElapsedTime(&gpu_ms, t_start, t_stop);
    printf("GPU time: %.3f ms\n", gpu_ms);

    // --- Validation ---
    const float tolerance = 1e-4f;
    int errors = 0;
    float max_err = 0.0f;
    const int sample_count = 1000;

    for (int i = 0; i < sample_count; i++) {
        int idx = rand() % NUM_PIXELS;
        float dr = fabsf(h_r_out_gpu[idx] - h_r_out_cpu[idx]);
        float dg = fabsf(h_g_out_gpu[idx] - h_g_out_cpu[idx]);
        float db = fabsf(h_b_out_gpu[idx] - h_b_out_cpu[idx]);
        float err = fmaxf(fmaxf(dr, dg), db);

        if (err > tolerance) {
            errors++;
            if (errors <= 3) {
                printf("Mismatch at %d: GPU(%.3f,%.3f,%.3f) CPU(%.3f,%.3f,%.3f) diff=%.6
f\n",
                       idx,
                       h_r_out_gpu[idx], h_g_out_gpu[idx], h_b_out_gpu[idx],
                       h_r_out_cpu[idx], h_g_out_cpu[idx], h_b_out_cpu[idx],
                       err);
            }
        }
        if (err > max_err) max_err = err;
    }
    printf("\nSoA Validation: %d errors in %d samples, max error = %.6f\n",
           errors, sample_count, max_err);

    // Cleanup
    cudaFreeHost(h_r_in); cudaFreeHost(h_g_in); cudaFreeHost(h_b_in);
    cudaFreeHost(h_r_out_gpu); cudaFreeHost(h_g_out_gpu); cudaFreeHost(h_b_out_gpu);
    free(h_r_out_cpu); free(h_g_out_cpu); free(h_b_out_cpu);
    cudaFree(d_r_in); cudaFree(d_g_in); cudaFree(d_b_in);
    cudaFree(d_r_out); cudaFree(d_g_out); cudaFree(d_b_out);
    cudaEventDestroy(t_start);
    cudaEventDestroy(t_stop);

    return 0;
}
```

```
Writing /tmp/t2v1.cu
```

In [3]:

```
# @title
!nvcc -arch=sm_75 /tmp/t2v1.cu -o /tmp/t2v1
```

In [4]:

```
# @title
# run the executable with nvprof
!nvprof /tmp/t2v1
```

```
SoA Blur: Image size 1024x1024 (1048576 pixels)
==570== NVPROF is profiling process 570, command: /tmp/t2v1
CPU time: 111.295 ms
GPU time: 1.392 ms

SoA Validation: 0 errors in 1000 samples, max error = 0.000000
==570== Profiling application: /tmp/t2v1
==570== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   46.81%  1.0299ms         3  343.29us  341.40us  346.43us  [CUDA memcpy
HtoD]
                   43.66%  960.52us         3  320.17us  319.61us  320.51us  [CUDA memcp
y DtoH]
                    9.53%  209.76us         1  209.76us  209.76us  209.76us  blurKernelS
oA(float const *, float const *, float const *, float*, float*, float*, int, int)
      API calls:   84.94%  119.12ms         2  59.562ms     842ns  119.12ms  cudaEventCr
eate
                    8.43%  11.827ms         6  1.9712ms  1.7140ms  2.9816ms  cudaHostAll
oc
                    2.35%  3.2933ms         6  548.88us  332.57us  1.3846ms  cudaMemcpy
                    2.33%  3.2682ms         6  544.70us  491.19us  695.34us  cudaFreeHos
t
                    0.82%  1.1478ms         6  191.30us  101.18us  216.31us  cudaFree
                    0.57%  799.71us         1  799.71us  799.71us  799.71us  cuDeviceGet
PCIBusId
                    0.29%  406.28us         6  67.713us  60.594us  89.897us  cudaMalloc
                    0.13%  179.43us         1  179.43us  179.43us  179.43us  cudaLaunchK
ernel
                    0.12%  161.88us       114  1.4190us     109ns  71.247us  cuDeviceGet
Attribute
                    0.01%  12.683us         1  12.683us  12.683us  12.683us  cuDeviceGet
Name
                    0.01%  10.522us         2  5.2610us  3.6270us  6.8950us  cudaEventRe
cord
                    0.00%  6.0220us         1  6.0220us  6.0220us  6.0220us  cudaEventSy
nchronize
                    0.00%  3.6140us         1  3.6140us  3.6140us  3.6140us  cudaEventEl
apsedTime
                    0.00%  2.6210us         2  1.3100us     478ns  2.1430us  cudaEventDe
stroy
                    0.00%  2.0920us         4     523ns     154ns  1.0500us  cudaGetLast
Error
                    0.00%  1.4150us         3     471ns     146ns     846ns  cuDeviceGet
Count
```

```
                    0.00%      923ns          1      923ns      923ns      923ns   cuDeviceTot
alMem
                    0.00%      806ns          2      403ns      175ns      631ns   cuDeviceGet
                    0.00%      458ns          1      458ns      458ns      458ns   cuModuleGet
LoadingMode
                    0.00%      274ns          1      274ns      274ns      274ns   cuDeviceGet
Uuid
```

In [ ]:

Urmat teplo

cpu

In [5]:

```
# @title
%%file /tmp/t1v1.cu

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <algorithm>
#include <chrono>

// Параметры задачи
#define NX 256            // Количество узлов по x
#define NY 256            // Количество узлов по y
#define DX (1.0/(NX-1)) // Шаг по пространству
#define DY (1.0/(NY-1))
#define C 1.0            // Коэффициент теплопроводности

// Автоматический расчет устойчивого шага по времени
#define DT (0.25 * fmin(DX*DX, DY*DY) / C) // Условие Куранта

double** allocate_2d_array(int rows, int cols) {
    double** arr = (double**)malloc(rows * sizeof(double*));
    for (int i = 0; i < rows; i++) {
        arr[i] = (double*)calloc(cols, sizeof(double)); // Используем calloc для инициа
лизации нулями
    }
    return arr;
}

void free_2d_array(double** arr, int rows) {
    for (int i = 0; i < rows; i++) {
        free(arr[i]);
    }
    free(arr);
}

// Инициализация начальных и граничных условий
void initialize(double** u) {
    // Начальное условие: u(x,y,0) = 1.0
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 1.0;
        }
    }

    // Граничные условия: u = 2.0 на всех границах
    for (int i = 0; i < NX; i++) {
        u[i][0] = 2.0;
        u[i][NY-1] = 2.0;
    }
    for (int j = 0; j < NY; j++) {
        u[0][j] = 2.0;
        u[NX-1][j] = 2.0;
    }
}

// Один шаг по времени методом крест
void time_step(double** u, double** u_new) {
    double dx2 = DX * DX;
    double dy2 = DY * DY;

    // Копируем граничные условия
    for (int i = 0; i < NX; i++) {
        u_new[i][0] = u[i][0];
        u_new[i][NY-1] = u[i][NY-1];
    }
    for (int j = 0; j < NY; j++) {
        u_new[0][j] = u[0][j];
        u_new[NX-1][j] = u[NX-1][j];
    }

    // Вычисление новых значений во внутренних точках
    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            double laplacian = (u[i-1][j] - 2*u[i][j] + u[i+1][j]) / dx2 +
                              (u[i][j-1] - 2*u[i][j] + u[i][j+1]) / dy2;

            u_new[i][j] = u[i][j] + DT * C * laplacian;

            // Проверка на численную устойчивость
            if (!std::isfinite(u_new[i][j])) {
                printf("Numerical instability at (%d, %d)! Value = %f\n", i, j, u_new[i]
[j]);
                return;
            }
        }
    }
}

// Вычисление максимального изменения за шаг (для мониторинга)
double max_change(double** u, double** u_new) {
    double max_diff = 0.0;
    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            double diff = fabs(u_new[i][j] - u[i][j]);
            if (diff > max_diff) {
                max_diff = diff;
            }
        }
    }
    return max_diff;
}

void save_to_file(double** u, const char* filename, double t) {
    FILE* file = fopen(filename, "w");
    if (!file) {
        printf("Error opening file %s\n", filename);
        return;
    }

    fprintf(file, "# Time: %.6f\n", t);
    fprintf(file, "# X Y U\n");

    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            fprintf(file, "%.6f %.6f %.6f\n", i*DX, j*DY, u[i][j]);
        }
    }

    fclose(file);
    printf("Saved to %s\n", filename);
}
```

```
int main(int argc, char* argv[]) {
    double target_time = 0.1;
    if (target_time <= 0) {
        printf("Error: target_time must be positive\n");
        return 1;
    }

    printf("Solving heat equation until t = %.6f\n", target_time);
    printf("Grid: %dx%d, dx=%.6f, dy=%.6f\n", NX, NY, DX, DY);
    printf("Automatic time step: dt=%.8f (for stability)\n", DT);
    printf("Courant number: %.6f\n", C * DT * (1.0/(DX*DX) + 1.0/(DY*DY)));

    double** u_current = allocate_2d_array(NX, NY);
    double** u_next = allocate_2d_array(NX, NY);
    initialize(u_current);
    initialize(u_next);

    double t = 0.0;
    int step = 0;
    printf("Step 0: t = %.6f\n", t);
    auto start_computation = std::chrono::high_resolution_clock::now();

    // Основной цикл по времени
    while (t < target_time) {
        // Если осталось меньше чем DT, уменьшаем шаг
        double actual_dt = DT;
        if (t + DT > target_time) {
            actual_dt = target_time - t;
        }

        time_step(u_current, u_next);
        std::swap(u_current, u_next);

        t += actual_dt;
        step++;

        // Вывод прогресса каждые 1000 шагов
        if (step % 1000 == 0) {
            double max_diff = max_change(u_next, u_current);
            printf("Step %d: t = %.6f, max change = %.8f\n", step, t, max_diff);
        }

        // Аварийный выход при неустойчивости
        if (step > 1000000) { // Защита от бесконечного цикла
            printf("Too many steps, possible instability!\n");
            break;
        }
    }

    auto end_computation = std::chrono::high_resolution_clock::now();

    printf("Final: step %d, t = %.6f\n", step, t);

    save_to_file(u_current, "solution_final.txt", t);

    // СИММЕТРИЧНЫЙ вывод среза для проверки (ИСПРАВЛЕННЫЙ)
    printf("\nSymmetric slice at y = %d (middle):\n", NY/2);
    printf("Left part (from center to left boundary):\n");

    // Используем безопасные границы
    int center = NX / 2;
    int max_offset = center; // максимальный offset до левой границы

    for (int offset = 0; offset <= max_offset; offset += NX/16) {
        int left_x = center - offset;
        int right_x = center + offset;

        // Проверяем границы для безопасности
        if (left_x >= 0 && left_x < NX && right_x >= 0 && right_x < NX) {
            printf("  u[%3d][%3d] = %8.6f  |  u[%3d][%3d] = %8.6f\n",
                   left_x, NY/2, u_current[left_x][NY/2],
                    right_x, NY/2, u_current[right_x][NY/2]);
        }
    }

    // Проверка симметрии более детально (только для безопасных offset)
    printf("\nDetailed symmetry check around center:\n");
    for (int offset = 1; offset <= 5; offset++) {
        int left_x = center - offset;
        int right_x = center + offset;

        if (left_x >= 0 && right_x < NX) {
            double left_val = u_current[left_x][NY/2];
            double right_val = u_current[right_x][NY/2];
            double symmetry_error = fabs(left_val - right_val);
            printf("  Offset %d: u[%d]=%.6f, u[%d]=%.6f, error=%.8f\n",
                   offset, left_x, left_val, right_x, right_val, symmetry_error);
        }
    }

    printf("\nBoundary check:\n");
    printf("Top-left: u[0][%d] = %.6f (should be 2.0)\n", NY-1, u_current[0][NY-1]);
    printf("Center: u[%d][%d] = %.6f\n", NX/2, NY/2, u_current[NX/2][NY/2]);
    printf("Bottom-right: u[%d][0] = %.6f (should be 2.0)\n", NX-1, u_current[NX-1][0]);

    auto computation_time = std::chrono::duration_cast<std::chrono::microseconds>(end_co
mputation - start_computation);
    printf("\n=== Performance Results ===\n");
    printf("Computation time: %.3f ms\n", computation_time.count() / 1000.0);
    printf("Time per step: %.3f microseconds\n", computation_time.count() / (double)step
);

    free_2d_array(u_current, NX);
    free_2d_array(u_next, NX);

    return 0;
}
```

```
Writing /tmp/t1v1.cu
```

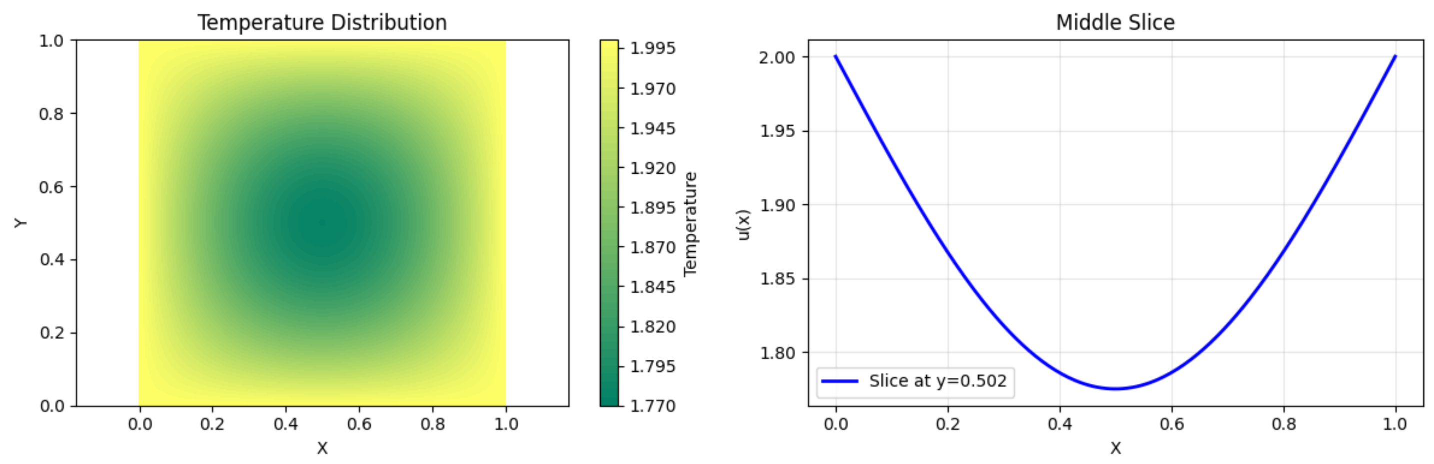
In [6]:

```
# @title
!nvcc -arch=sm_75 /tmp/t1v1.cu -o /tmp/t1v1
# run the executable with nvprof
!nvprof /tmp/t1v1
```

```
Solving heat equation until t = 0.100000
Grid: 256x256, dx=0.003922, dy=0.003922
Automatic time step: dt=0.00000384 (for stability)
Courant number: 0.500000
Step 0: t = 0.000000
Step 1000: t = 0.003845, max change = 0.00036008
Step 2000: t = 0.007689, max change = 0.00018007
Step 3000: t = 0.011534, max change = 0.00012005
Step 4000: t = 0.015379, max change = 0.00009007
Step 5000: t = 0.019223, max change = 0.00007232
Step 6000: t = 0.023068, max change = 0.00006113
Step 7000: t = 0.026913, max change = 0.00005419
Step 8000: t = 0.030757, max change = 0.00005036
Step 9000: t = 0.034602, max change = 0.00004903
Step 10000: t = 0.038447, max change = 0.00004854
Step 11000: t = 0.042291, max change = 0.00004716
Step 12000: t = 0.046136, max change = 0.00004521
Step 13000: t = 0.049981, max change = 0.00004294
Step 14000: t = 0.053825, max change = 0.00004051
Step 15000: t = 0.057670, max change = 0.00003803
Step 16000: t = 0.061515, max change = 0.00003559
Step 17000: t = 0.065359, max change = 0.00003321
Step 18000: t = 0.069204, max change = 0.00003094
Step 19000: t = 0.073049, max change = 0.00002879
Step 20000: t = 0.076894, max change = 0.00002676
Step 21000: t = 0.080738, max change = 0.00002485
Step 22000: t = 0.084583, max change = 0.00002307
Step 23000: t = 0.088428, max change = 0.00002141
Step 24000: t = 0.092272, max change = 0.00001986
Step 25000: t = 0.096117, max change = 0.00001842
Step 26000: t = 0.099962, max change = 0.00001708
Final: step 26010, t = 0.100000
Saved to solution_final.txt

Symmetric slice at y = 128 (middle):
Left part (from center to left boundary):
  u[128][128] = 1.774887  |  u[128][128] = 1.774887
  u[112][128] = 1.778971  |  u[144][128] = 1.779514
  u[ 96][128] = 1.791608  |  u[160][128] = 1.792673
  u[ 80][128] = 1.812313  |  u[176][128] = 1.813859
  u[ 64][128] = 1.840289  |  u[192][128] = 1.842256
  u[ 48][128] = 1.874457  |  u[208][128] = 1.876769
  u[ 32][128] = 1.913495  |  u[224][128] = 1.916063
  u[ 16][128] = 1.955891  |  u[240][128] = 1.958615

Detailed symmetry check around center:
  Offset 1: u[127]=1.774887, u[129]=1.774921, error=0.00003413
  Offset 2: u[126]=1.774921, u[130]=1.774989, error=0.00006826
  Offset 3: u[125]=1.774989, u[131]=1.775092, error=0.00010238
  Offset 4: u[124]=1.775092, u[132]=1.775228, error=0.00013648
  Offset 5: u[123]=1.775228, u[133]=1.775399, error=0.00017057

Boundary check:
Top-left: u[0][255] = 2.000000 (should be 2.0)
Center: u[128][128] = 1.774887
Bottom-right: u[255][0] = 2.000000 (should be 2.0)

=== Performance Results ===
Computation time: 30785.412 ms
Time per step: 1183.599 microseconds
======== Warning: No profile data collected.
```

In [7]:

```python
# @title
import numpy as np
import matplotlib.pyplot as plt

def quick_plot():
    try:
        data = np.loadtxt('solution_final.txt', comments='#')
        x = data[:, 0]
        y = data[:, 1]
        u = data[:, 2]

        # Преобразование в сетку
        x_unique = np.unique(x)
        y_unique = np.unique(y)
        nx = len(x_unique)
        ny = len(y_unique)

        u_grid = u.reshape(nx, ny)
        X, Y = np.meshgrid(x_unique, y_unique, indexing='ij')

        plt.figure(figsize=(12, 4))

        plt.subplot(1, 2, 1)
        #plt.contourf(X, Y, u_grid, levels=50, cmap='cividis')
        plt.contourf(X, Y, u_grid, levels=50, cmap='summer')
        #plt.contourf(X, Y, u_grid, levels=50, cmap='Reds')
        plt.colorbar(label='Temperature')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Temperature Distribution')
        plt.axis('equal')

        plt.subplot(1, 2, 2)
        middle_y = ny // 2
        plt.plot(x_unique, u_grid[:, middle_y], 'b-', linewidth=2, label=f'Slice at y={y_
unique[middle_y]:.3f}')
        plt.xlabel('X')
        plt.ylabel('u(x)')
        plt.title('Middle Slice')
        plt.grid(True, alpha=0.3)
        plt.legend()

        plt.tight_layout()
        plt.show()

        print(f"Min temperature: {u_grid.min():.4f}")
        print(f"Max temperature: {u_grid.max():.4f}")
        print(f"Center temperature: {u_grid[nx//2, ny//2]:.4f}")

    except Exception as e:
        print(f"Error: {e}")

quick_plot()
```



```
Min temperature: 1.7749
Max temperature: 2.0000
Center temperature: 1.7749
```

In [ ]:

gpu

In [8]:

```
# @title
%%file /tmp/t2v1.cu

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <algorithm>
#include <chrono>

// Параметры задачи
#define NX 256
#define NY 256
#define DX (1.0f/(NX-1))
#define DY (1.0f/(NY-1))
#define C 1.0f
#define DT (0.25f * fminf(DX*DX, DY*DY) / C)

// Размер блока------------
#define BLOCK_SIZE 16

#define CHECK_CUDA_ERROR(err) \
if (err != cudaSuccess) { \
        printf("CUDA error: %s at %s:%d\n", cudaGetErrorString(err), __FILE__, __LINE__); \
        exit(1); \
    }


// CUDA kernel для одного шага по времени
__global__ void time_step_kernel(double* u, double* u_new, int width, int height,
                                 double dx2, double dy2, double dt_c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= width || j >= height) return;

    int idx = j * width + i;

    // Граничные условия
    if (i == 0 || i == width-1 || j == 0 || j == height-1) {
        u_new[idx] = u[idx];
        return;
    }

    // Вычисление лапласиана для внутренних точек
    double laplacian = (u[idx - 1] - 2*u[idx] + u[idx + 1]) / dx2 +
                      (u[idx - width] - 2*u[idx] + u[idx + width]) / dy2;

    u_new[idx] = u[idx] + dt_c * C * laplacian;
}

// CUDA kernel для инициализации
__global__ void initialize_kernel(double* u, int width, int height) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= width || j >= height) return;

    int idx = j * width + i;
    u[idx] = 1.0;

    if (i == 0 || i == width-1 || j == 0 || j == height-1) {
        u[idx] = 2.0;
    }
}

double compute_max_change_cpu(double* u, double* u_new, int width, int height) {
    double max_diff = 0.0;
    for (int j = 1; j < height-1; j++) {
        for (int i = 1; i < width-1; i++) {
            int idx = j * width + i;
            double diff = fabs(u_new[idx] - u[idx]);
            if (diff > max_diff) {
                max_diff = diff;
            }
        }
    }
    return max_diff;
}

double* allocate_gpu_memory(int size) {
    double* ptr;
    CHECK_CUDA_ERROR(cudaMalloc(&ptr, size * sizeof(double)));
    return ptr;
}

void free_gpu_memory(double* ptr) {
    CHECK_CUDA_ERROR(cudaFree(ptr));
}

void copy_between_cpu_gpu(double* dst, double* src, int size, bool to_gpu) {
    cudaMemcpyKind kind = to_gpu ? cudaMemcpyHostToDevice : cudaMemcpyDeviceToHost;
    CHECK_CUDA_ERROR(cudaMemcpy(dst, src, size * sizeof(double), kind));
}
```

```
double** allocate_2d_array(int rows, int cols) {
    double** arr = (double**)malloc(rows * sizeof(double*));
    for (int i = 0; i < rows; i++) {
        arr[i] = (double*)calloc(cols, sizeof(double));
    }
    return arr;
}

void free_2d_array(double** arr, int rows) {
    for (int i = 0; i < rows; i++) {
        free(arr[i]);
    }
    free(arr);
}

void save_to_file(double* u, const char* filename, double t, int width, int height) {
    FILE* file = fopen(filename, "w");
    if (!file) {
        printf("Error opening file %s\n", filename);
        return;
    }

    fprintf(file, "# Time: %.6f\n", t);
    fprintf(file, "# X Y U\n");

    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            int idx = j * width + i;
            fprintf(file, "%.6f %.6f %.6f\n", i*DX, j*DY, u[idx]);
        }
    }

    fclose(file);
    printf("Saved to %s\n", filename);
}

int main() {
    double target_time = 0.1;

    printf("Solving heat equation with CUDA (SYMMETRIC OUTPUT) until t = %.6f\n", target
_time);
    printf("Grid: %dx%d, dx=%.6f, dy=%.6f\n", NX, NY, DX, DY);
    printf("Time step: dt=%.8f\n", DT);
    printf("Block size: %dx%d\n", BLOCK_SIZE, BLOCK_SIZE);

    double* h_u = (double*)malloc(NX * NY * sizeof(double));
    double* h_u_new = (double*)malloc(NX * NY * sizeof(double));
    double* d_u = allocate_gpu_memory(NX * NY);
    double* d_u_new = allocate_gpu_memory(NX * NY);

    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridSize((NX + BLOCK_SIZE - 1) / BLOCK_SIZE,
                  (NY + BLOCK_SIZE - 1) / BLOCK_SIZE);

    printf("Grid size: %dx%d blocks\n", gridSize.x, gridSize.y);

    for (int i = 0; i < NX * NY; i++) {
        h_u[i] = 1.0;
    }
    // Граничные условия
    for (int i = 0; i < NX; i++) {
        h_u[i] = 2.0;
        h_u[(NY-1)*NX + i] = 2.0;
    }
    for (int j = 0; j < NY; j++) {
        h_u[j*NX] = 2.0;
        h_u[j*NX + (NX-1)] = 2.0;
    }

    copy_between_cpu_gpu(d_u, h_u, NX * NY, true);
    copy_between_cpu_gpu(d_u_new, h_u, NX * NY, true);
    double t = 0.0;
    int step = 0;

    printf("Step 0: t = %.6f\n", t);

    auto start_computation = std::chrono::high_resolution_clock::now();
    while (t < target_time) {
        double actual_dt = DT;
        if (t + DT > target_time) {
            actual_dt = target_time - t;
        }

        time_step_kernel<<<gridSize, blockSize>>>(d_u, d_u_new, NX, NY, DX*DX, DY*DY, ac
tual_dt);
        CHECK_CUDA_ERROR(cudaDeviceSynchronize());

        std::swap(d_u, d_u_new);
        t += actual_dt;
        step++;

        // Проверка максимального изменения каждые 1000 шагов
        if (step % 1000 == 0) {
            // Копируем оба массива на CPU для вычисления max_change
            copy_between_cpu_gpu(h_u, d_u, NX * NY, false);
            copy_between_cpu_gpu(h_u_new, d_u_new, NX * NY, false);

            double max_diff = compute_max_change_cpu(h_u, h_u_new, NX, NY);
            printf("Step %d: t = %.6f, max change = %.8f\n", step, t, max_diff);
        }

        if (step > 1000000) {
            printf("Too many steps, possible instability!\n");
            break;
        }
    }

    auto end_computation = std::chrono::high_resolution_clock::now();

    printf("Final: step %d, t = %.6f\n", step, t);

    // Копируем результат обратно на CPU
    copy_between_cpu_gpu(h_u, d_u, NX * NY, false);
    save_to_file(h_u, "solution_final_cuda_symmetric.txt", t, NX, NY);

    printf("\nSymmetric slice at y = %d (middle):\n", NY/2);
    printf("Left part (from center to left boundary):\n");
    for (int offset = 0; offset <= NX/2; offset += NX/16) {
        int left_idx = (NY/2) * NX + (NX/2 - offset);
        int right_idx = (NY/2) * NX + (NX/2 + offset);
        printf("  u[%3d][%3d] = %8.6f  |  u[%3d][%3d] = %8.6f\n",
               NX/2 - offset, NY/2, h_u[left_idx],
               NX/2 + offset, NY/2, h_u[right_idx]);
    }

    // Проверка симметрии более детально
    printf("\nDetailed symmetry check around center:\n");
    int center = NX / 2;
    for (int offset = 1; offset <= 5; offset++) {
        int left_idx = (NY/2) * NX + (center - offset);
        int right_idx = (NY/2) * NX + (center + offset);
        double left_val = h_u[left_idx];
        double right_val = h_u[right_idx];
        double symmetry_error = fabs(left_val - right_val);
        printf("  Offset %d: u[%d]=%.6f, u[%d]=%.6f, error=%.8f\n",
               offset, center-offset, left_val, center+offset, right_val, symmetry_error
);
    }

    printf("\nBoundary check:\n");
    printf("Top-left corner:     u[0][%d] = %.6f (should be 2.0)\n", NY-1, h_u[(NY-1)*NX
+ 0]);
    printf("Center:              u[%d][%d] = %.6f\n", NX/2, NY/2, h_u[(NY/2)*NX + NX/2]
);
    printf("Bottom-right corner: u[%d][0] = %.6f (should be 2.0)\n", NX-1, h_u[0*NX + (N
X-1)]);

    auto computation_time = std::chrono::duration_cast<std::chrono::microseconds>(end_co
mputation - start_computation);
    printf("\n=== CUDA Performance Results ===\n");
    printf("Computation time: %.3f ms\n", computation_time.count() / 1000.0);
    printf("Time per step: %.3f microseconds\n", computation_time.count() / (double)step
);

    free(h_u);
    free(h_u_new);
    free_gpu_memory(d_u);
    free_gpu_memory(d_u_new);

    return 0;
}
```

```
Overwriting /tmp/t2v1.cu
```

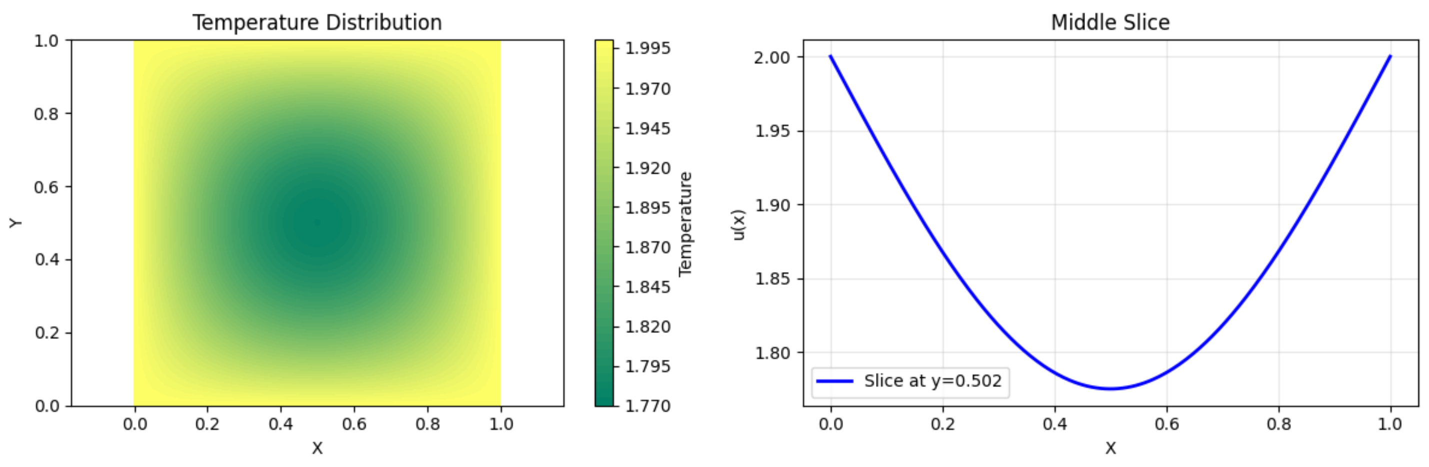
In [9]:

```
# @title
!nvcc -arch=sm_75 /tmp/t2v1.cu -o /tmp/t2v1
# run the executable with nvprof
!nvprof /tmp/t2v1
```

```
Solving heat equation with CUDA (SYMMETRIC OUTPUT) until t = 0.100000
Grid: 256x256, dx=0.003922, dy=0.003922
Time step: dt=0.00000384
Block size: 16x16
==1026== NVPROF is profiling process 1026, command: /tmp/t2v1
Grid size: 16x16 blocks
Step 0: t = 0.000000
Step 1000: t = 0.003845, max change = 0.00036008
Step 2000: t = 0.007689, max change = 0.00018007
Step 3000: t = 0.011534, max change = 0.00012005
Step 4000: t = 0.015379, max change = 0.00009007
Step 5000: t = 0.019223, max change = 0.00007232
Step 6000: t = 0.023068, max change = 0.00006113
Step 7000: t = 0.026913, max change = 0.00005419
Step 8000: t = 0.030757, max change = 0.00005036
Step 9000: t = 0.034602, max change = 0.00004903
Step 10000: t = 0.038447, max change = 0.00004854
Step 11000: t = 0.042291, max change = 0.00004716
Step 12000: t = 0.046136, max change = 0.00004521
Step 13000: t = 0.049981, max change = 0.00004294
Step 14000: t = 0.053825, max change = 0.00004051
Step 15000: t = 0.057670, max change = 0.00003803
Step 16000: t = 0.061515, max change = 0.00003559
Step 17000: t = 0.065359, max change = 0.00003321
Step 18000: t = 0.069204, max change = 0.00003094
Step 19000: t = 0.073049, max change = 0.00002879
Step 20000: t = 0.076894, max change = 0.00002676
Step 21000: t = 0.080738, max change = 0.00002485
Step 22000: t = 0.084583, max change = 0.00002307
Step 23000: t = 0.088428, max change = 0.00002141
Step 24000: t = 0.092272, max change = 0.00001986
Step 25000: t = 0.096117, max change = 0.00001842
Step 26000: t = 0.099962, max change = 0.00001708
Final: step 26010, t = 0.100000
Saved to solution_final_cuda_symmetric.txt

Symmetric slice at y = 128 (middle):
Left part (from center to left boundary):
  u[128][128] = 1.774887  |  u[128][128] = 1.774887
  u[112][128] = 1.778971  |  u[144][128] = 1.779514
  u[ 96][128] = 1.791608  |  u[160][128] = 1.792673
  u[ 80][128] = 1.812313  |  u[176][128] = 1.813859
  u[ 64][128] = 1.840289  |  u[192][128] = 1.842256
  u[ 48][128] = 1.874457  |  u[208][128] = 1.876769
  u[ 32][128] = 1.913495  |  u[224][128] = 1.916063
  u[ 16][128] = 1.955891  |  u[240][128] = 1.958615
  u[  0][128] = 2.000000  |  u[256][128] = 2.000000

Detailed symmetry check around center:
  Offset 1: u[127]=1.774887, u[129]=1.774921, error=0.00003413
  Offset 2: u[126]=1.774921, u[130]=1.774989, error=0.00006826
  Offset 3: u[125]=1.774989, u[131]=1.775092, error=0.00010238
  Offset 4: u[124]=1.775092, u[132]=1.775228, error=0.00013648
  Offset 5: u[123]=1.775228, u[133]=1.775399, error=0.00017057

Boundary check:
Top-left corner:     u[0][255] = 2.000000 (should be 2.0)
Center:              u[128][128] = 1.774887
Bottom-right corner: u[255][0] = 2.000000 (should be 2.0)

=== CUDA Performance Results ===
Computation time: 856.832 ms
Time per step: 32.942 microseconds
==1026== Profiling application: /tmp/t2v1
==1026== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.55%  575.31ms     26010  22.118us  15.679us  84.030us  time_step_ke
rnel(double*, double*, int, int, double, double, double)
                    0.44%  2.5376ms        53  47.878us  42.239us  53.790us  [CUDA memcp
y DtoH]
                    0.02%  90.590us         2  45.295us  44.991us  45.599us  [CUDA memcp
y HtoD]
      API calls:   72.90%  670.87ms     26010  25.792us  3.0620us  1.8114ms  cudaDeviceSy
nchronize
                   16.25%  149.51ms     26010  5.7480us  3.2570us  585.96us  cudaLaunchK
ernel
                    9.68%  89.041ms         2  44.520ms  3.2050us  89.037ms  cudaMalloc
                    1.13%  10.422ms        55  189.50us  135.05us  492.94us  cudaMemcpy
                    0.02%  193.86us         2  96.930us  29.736us  164.12us  cudaFree
                    0.02%  156.67us       114  1.3740us     104ns  66.910us  cuDeviceGet
Attribute
                    0.00%  14.630us         1  14.630us  14.630us  14.630us  cuDeviceGet
Name
                    0.00%  5.5570us         1  5.5570us  5.5570us  5.5570us  cuDeviceGet
PCIBusId
                    0.00%  2.3800us         3     793ns     132ns  1.9690us  cuDeviceGet
Count
                    0.00%  1.1110us         2     555ns     203ns     908ns  cuDeviceGet
                    0.00%     509ns         1     509ns     509ns     509ns  cuDeviceTot
alMem
                    0.00%     423ns         1     423ns     423ns     423ns  cuModuleGet
LoadingMode
                    0.00%     239ns         1     239ns     239ns     239ns  cuDeviceGet
Uuid
```

In [10]:

```python
# @title
import numpy as np
import matplotlib.pyplot as plt

def quick_plot1():
    try:
        data = np.loadtxt('solution_final_cuda_symmetric.txt', comments='#')
        x = data[:, 0]
        y = data[:, 1]
        u = data[:, 2]

        # Преобразование в сетку
        x_unique = np.unique(x)
        y_unique = np.unique(y)
        nx = len(x_unique)
        ny = len(y_unique)

        u_grid = u.reshape(nx, ny)
        X, Y = np.meshgrid(x_unique, y_unique, indexing='ij')
        plt.figure(figsize=(12, 4))

        plt.subplot(1, 2, 1)
        #plt.contourf(X, Y, u_grid, levels=50, cmap='cividis')
        plt.contourf(X, Y, u_grid, levels=50, cmap='summer')
        #plt.contourf(X, Y, u_grid, levels=50, cmap='Reds')
        plt.colorbar(label='Temperature')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Temperature Distribution')
        plt.axis('equal')

        plt.subplot(1, 2, 2)
        middle_y = ny // 2
        plt.plot(x_unique, u_grid[:, middle_y], 'b-', linewidth=2, label=f'Slice at y={y_
unique[middle_y]:.3f}')
        plt.xlabel('X')
        plt.ylabel('u(x)')
        plt.title('Middle Slice')
        plt.grid(True, alpha=0.3)
        plt.legend()

        plt.tight_layout()
        plt.show()

        print(f"Min temperature: {u_grid.min():.4f}")
        print(f"Max temperature: {u_grid.max():.4f}")
        print(f"Center temperature: {u_grid[nx//2, ny//2]:.4f}")

    except Exception as e:
        print(f"Error: {e}")

quick_plot1()
```



```
Min temperature: 1.7749
Max temperature: 2.0000
Center temperature: 1.7749
```

In [ ]:

```

```

urmat laplac

cpu

In [11]:

```
# @title
%%file /tmp/t1v1.cu

#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <algorithm>
#include <chrono>

// Параметры задачи
//#define NX 256           // Количество узлов по x
//#define NY 256           // Количество узлов по y
#define NX 128            // Количество узлов по x
#define NY 128            // Количество узлов по y
#define DX (1.0/(NX-1)) // Шаг по пространству
#define DY (1.0/(NY-1))
#define MAX_ITER 50000    // Максимальное количество итераций
#define TOLERANCE 1e-5    // Точность решения

double** allocate_2d_array(int rows, int cols) {
    double** arr = (double**)malloc(rows * sizeof(double*));
    for (int i = 0; i < rows; i++) {
        arr[i] = (double*)calloc(cols, sizeof(double));
    }
    return arr;
}

void free_2d_array(double** arr, int rows) {
    for (int i = 0; i < rows; i++) {
        free(arr[i]);
    }
    free(arr);
}

// Инициализация начальных и граничных условий
void initialize(double** u) {
    // Внутренняя область инициализируется нулями
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 0.0;
        }
    }

    // Граничные условия из новой задачи
    for (int i = 0; i < NX; i++) {
        u[i][0] = exp(1.0 - i*DX);     // u(x,0) = e^(1-x)
        u[i][NY-1] = 1.0;              // u(x,1) = 1.0
    }

    for (int j = 0; j < NY; j++) {
        u[0][j] = exp(1.0 - j*DY);     // u(0,y) = e^(1-y)
        u[NX-1][j] = 1.0;              // u(1,y) = 1.0
    }
}

// Простой метод Якоби для уравнения Лапласа
void jacobi_step(double** u, double** u_new) {
    double dx2 = DX * DX;
    double dy2 = DY * DY;
    double factor = 0.5 / (1.0/dx2 + 1.0/dy2);

    // Копируем граничные условия
    for (int i = 0; i < NX; i++) {
        u_new[i][0] = u[i][0];
        u_new[i][NY-1] = u[i][NY-1];
    }
    for (int j = 0; j < NY; j++) {
        u_new[0][j] = u[0][j];
        u_new[NX-1][j] = u[NX-1][j];
    }

    // Вычисление новых значений во внутренних точках
    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            u_new[i][j] = factor * (
                (u[i-1][j] + u[i+1][j]) / dx2 +
                (u[i][j-1] + u[i][j+1]) / dy2
            );
        }
    }
}

// Вычисление максимального изменения за шаг
double max_change(double** u, double** u_new) {
    double max_diff = 0.0;
    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            double diff = fabs(u_new[i][j] - u[i][j]);
            if (diff > max_diff) {
                max_diff = diff;
            }
        }
    }
    return max_diff;
}

// Вычисление невязки (норма C)//---------------------------------------------------------
-------------------
double compute_residual(double** u) {
    double max_residual = 0.0;
    double dx2 = DX * DX;
    double dy2 = DY * DY;

    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            double residual = fabs(
                (u[i-1][j] - 2*u[i][j] + u[i+1][j]) / dx2 +
                (u[i][j-1] - 2*u[i][j] + u[i][j+1]) / dy2
            );
            if (residual > max_residual) {
                max_residual = residual;
            }
        }
    }
    return max_residual;
}

// Вычисление нормы L1
double compute_l1_norm(double** u) {
    double sum = 0.0;
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            sum += fabs(u[i][j]);
        }
    }
    return sum * DX * DY;
}

void save_to_file(double** u, const char* filename) {
    FILE* file = fopen(filename, "w");
    if (!file) {
        printf("Error opening file %s\n", filename);
        return;
    }

    fprintf(file, "# X Y U\n");
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            fprintf(file, "%.6f %.6f %.6f\n", i*DX, j*DY, u[i][j]);
        }
    }

    fclose(file);
    printf("Saved to %s\n", filename);
}

int main(int argc, char* argv[]) {
    printf("Solving Laplace equation: d²u/dx² + d²u/dy² = 0\n");
    printf("Grid: %dx%d, dx=%.6f, dy=%.6f\n", NX, NY, DX, DY);
    printf("Tolerance: %.2e, Max iterations: %d\n", TOLERANCE, MAX_ITER);

    double** u_current = allocate_2d_array(NX, NY);
    double** u_next = allocate_2d_array(NX, NY);
    initialize(u_current);
    initialize(u_next);

    auto start_computation = std::chrono::high_resolution_clock::now();

    // Основной итерационный цикл
    int iter = 0;
    double max_diff = 0.0;
    double residual = 0.0;

    printf("\nStarting iterations...\n");
    printf("Iter  Residual(C-norm)    Max-Change      L1-Norm\n");
    printf("--------------------------------------------------\n");

    for (iter = 1; iter <= MAX_ITER; iter++) {
        jacobi_step(u_current, u_next);

        max_diff = max_change(u_current, u_next);
        residual = compute_residual(u_next);
        double l1_norm = compute_l1_norm(u_next);

        std::swap(u_current, u_next);

        // Вывод прогресса каждые 1000 итераций
        if (iter % 1000 == 0) {
            printf("%5d  %12.6e  %12.6e  %12.6e\n",
                   iter, residual, max_diff, l1_norm);
        }

        // Критерий остановки
        if (residual < TOLERANCE && max_diff < TOLERANCE) {
            printf("%5d  %12.6e  %12.6e  %12.6e\n",
                   iter, residual, max_diff, l1_norm);
            printf("Converged!\n");
            break;
        }
    }

    auto end_computation = std::chrono::high_resolution_clock::now();

    if (iter > MAX_ITER) {
        printf("Reached maximum iterations without convergence\n");
        printf("Current residual: %.6e (target: %.1e)\n", residual, TOLERANCE);
    }

    printf("\n=== Final Results ===\n");
    printf("Iterations: %d\n", iter);
    printf("Final residual (C-norm): %.6e\n", residual);
    printf("Final max change: %.6e\n", max_diff);
    printf("Final L1 norm: %.6e\n", compute_l1_norm(u_current));

    save_to_file(u_current, "solution_laplace.txt");

    // Анализ решения
    printf("\n=== Solution Analysis ===\n");

    // Проверка граничных условий
    printf("Boundary conditions check:\n");
    printf("u(0,0) = %.6f (should be e = %.6f)\n", u_current[0][0], exp(1.0));
    printf("u(1,0) = %.6f (should be 1.0)\n", u_current[NX-1][0]);
    printf("u(0,1) = %.6f (should be 1.0)\n", u_current[0][NY-1]);
    printf("u(1,1) = %.6f (should be 1.0)\n", u_current[NX-1][NY-1]);

    // Проверка внутренних точек
    printf("\nInternal points:\n");
    printf("u(0.5, 0.5) = %.6f\n", u_current[NX/2][NY/2]);
    printf("u(0.25, 0.25) = %.6f\n", u_current[NX/4][NY/4]);
    printf("u(0.75, 0.75) = %.6f\n", u_current[3*NX/4][3*NY/4]);

    auto computation_time = std::chrono::duration_cast<std::chrono::microseconds>(end_co
mputation - start_computation);
    printf("\n=== Performance Results ===\n");
    printf("Total computation time: %.3f ms\n", computation_time.count() / 1000.0);
    printf("Time per iteration: %.3f microseconds\n", computation_time.count() / (double
)iter);
    printf("Iterations per second: %.0f\n", iter / (computation_time.count() / 1000000.0
));

    free_2d_array(u_current, NX);
    free_2d_array(u_next, NX);

    return 0;
}
```

```
Overwriting /tmp/t1v1.cu
```

In [12]:

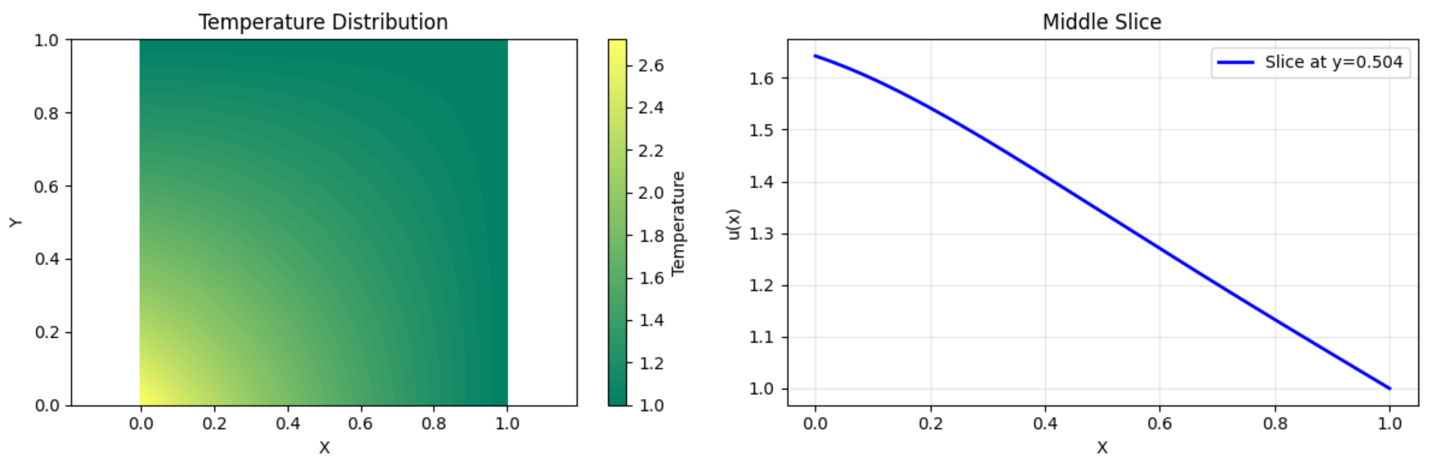
```
# @title
!nvcc -arch=sm_75 /tmp/t1v1.cu -o /tmp/t1v1
# run the executable with nvprof
!nvprof /tmp/t1v1
```

```
Solving Laplace equation: d²u/dx² + d²u/dy² = 0
Grid: 128x128, dx=0.007874, dy=0.007874
Tolerance: 1.00e-05, Max iterations: 50000

Starting iterations...
Iter  Residual(C-norm)    Max-Change      L1-Norm
---------------------------------------------------
 1000  4.825725e+01  7.485691e-04  6.747362e-01
 2000  2.282065e+01  3.537575e-04  8.815974e-01
 3000  1.647062e+01  2.553533e-04  1.015109e+00
 4000  1.249145e+01  1.936543e-04  1.109617e+00
 5000  9.287306e+00  1.439822e-04  1.178395e+00
 6000  6.855223e+00  1.062780e-04  1.228867e+00
 7000  5.049931e+00  7.829375e-05  1.265995e+00
 8000  3.719094e+00  5.765806e-05  1.293328e+00
 9000  2.738657e+00  4.245810e-05  1.313454e+00
10000  2.016613e+00  3.126406e-05  1.328275e+00
11000  1.484947e+00  2.302152e-05  1.339188e+00
12000  1.093468e+00  1.695284e-05  1.347225e+00
13000  8.052044e-01  1.248398e-05  1.353143e+00
14000  5.929389e-01  9.193138e-06  1.357501e+00
15000  4.366326e-01  6.769775e-06  1.360711e+00
16000  3.215317e-01  4.985224e-06  1.363074e+00
17000  2.367732e-01  3.671091e-06  1.364814e+00
18000  1.743580e-01  2.703370e-06  1.366096e+00
19000  1.283961e-01  1.990746e-06  1.367039e+00
20000  9.455003e-02  1.465974e-06  1.367734e+00
21000  6.962606e-02  1.079535e-06  1.368246e+00
22000  5.127221e-02  7.949636e-07  1.368623e+00
23000  3.775656e-02  5.854066e-07  1.368901e+00
24000  2.780371e-02  4.310901e-07  1.369105e+00
25000  2.047449e-02  3.174523e-07  1.369255e+00
26000  1.507730e-02  2.337701e-07  1.369366e+00
27000  1.110284e-02  1.721470e-07  1.369448e+00
28000  8.176065e-03  1.267680e-07  1.369508e+00
29000  6.020808e-03  9.335126e-08  1.369552e+00
30000  4.433688e-03  6.874334e-08  1.369585e+00
31000  3.264943e-03  5.062220e-08  1.369609e+00
32000  2.404285e-03  3.727790e-08  1.369626e+00
33000  1.770502e-03  2.745123e-08  1.369639e+00
34000  1.303788e-03  2.021493e-08  1.369649e+00
35000  9.601018e-04  1.488616e-08  1.369656e+00
36000  7.070135e-04  1.096208e-08  1.369661e+00
37000  5.206407e-04  8.072417e-09  1.369665e+00
38000  3.833969e-04  5.944482e-09  1.369668e+00
```

```
39000  2.823313e-04  4.377483e-09  1.369670e+00
40000  2.079071e-04  3.223553e-09  1.369672e+00
41000  1.531016e-04  2.373806e-09  1.369673e+00
42000  1.127432e-04  1.748057e-09  1.369674e+00
43000  8.302344e-05  1.287260e-09  1.369674e+00
44000  6.113799e-05  9.479306e-10  1.369675e+00
45000  4.502170e-05  6.980512e-10  1.369675e+00
46000  3.315375e-05  5.140413e-10  1.369675e+00
47000  2.441420e-05  3.785368e-10  1.369675e+00
48000  1.797849e-05  2.787524e-10  1.369675e+00
49000  1.323928e-05  2.052720e-10  1.369676e+00
49918  9.997030e-06  1.550018e-10  1.369676e+00
Converged!

=== Final Results ===
Iterations: 49918
Final residual (C-norm): 9.997030e-06
Final max change: 1.550018e-10
Final L1 norm: 1.369676e+00
Saved to solution_laplace.txt

=== Solution Analysis ===
Boundary conditions check:
u(0,0) = 2.718282 (should be e = 2.718282)
u(1,0) = 1.000000 (should be 1.0)
u(0,1) = 1.000000 (should be 1.0)
u(1,1) = 1.000000 (should be 1.0)

Internal points:
u(0.5, 0.5) = 1.337383
u(0.25, 0.25) = 1.815447
u(0.75, 0.75) = 1.080496

=== Performance Results ===
Total computation time: 20807.800 ms
Time per iteration: 416.840 microseconds
Iterations per second: 2399
======== Warning: No profile data collected.
```

In [13]:

```python
# @title
import numpy as np
import matplotlib.pyplot as plt

def quick_plot():
    try:
        data = np.loadtxt('solution_laplace.txt', comments='#')
        x = data[:, 0]
        y = data[:, 1]
        u = data[:, 2]

        # Преобразование в сетку
        x_unique = np.unique(x)
        y_unique = np.unique(y)
        nx = len(x_unique)
        ny = len(y_unique)

        u_grid = u.reshape(nx, ny)
        X, Y = np.meshgrid(x_unique, y_unique, indexing='ij')

        plt.figure(figsize=(12, 4))

        plt.subplot(1, 2, 1)
        #plt.contourf(X, Y, u_grid, levels=50, cmap='cividis')
        plt.contour(X, Y, u_grid, levels=50, cmap='summer')
        #plt.contourf(X, Y, u_grid, levels=50, cmap='Reds')
        plt.colorbar(label='Temperature')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Temperature Distribution')
        plt.axis('equal')
        plt.subplot(1, 2, 2)
        middle_y = ny // 2
        plt.plot(x_unique, u_grid[:, middle_y], 'b-', linewidth=2, label=f'Slice at y={y_
unique[middle_y]:.3f}')
        plt.xlabel('X')
        plt.ylabel('u(x)')
        plt.title('Middle Slice')
        plt.grid(True, alpha=0.3)
        plt.legend()

        plt.tight_layout()
        plt.show()

        print(f"Min temperature: {u_grid.min():.4f}")
        print(f"Max temperature: {u_grid.max():.4f}")
        print(f"Center temperature: {u_grid[nx//2, ny//2]:.4f}")

    except Exception as e:
        print(f"Error: {e}")

quick_plot()
```



```
Min temperature: 1.0000
Max temperature: 2.7183
Center temperature: 1.3374
```

In [ ]:

gpu

In [14]:

```
# @title
%%file /tmp/t2v1.cu

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <thrust/device_ptr.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <algorithm>
#include <chrono>

// Параметры задачи (как в CPU версии)
#define NX 128             // Количество узлов по x
#define NY 128             // Количество узлов по y
#define DX (1.0/(NX-1)) // Шаг по пространству
#define DY (1.0/(NY-1))
#define MAX_ITER 50000    // Максимальное количество итераций
#define TOLERANCE 1e-5    // Точность решения
```

```
// Размер блока
#define BLOCK_SIZE 16

#define CHECK_CUDA_ERROR(err) \
    if (err != cudaSuccess) { \
        printf("CUDA error: %s at %s:%d\n", cudaGetErrorString(err), __FILE__, __LINE__)
; \
        exit(1); \
    }

// CUDA kernel для одного шага Якоби
__global__ void jacobi_step_kernel(const double* u, double* u_new, int width, int height
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= 1 && i < width-1 && j >= 1 && j < height-1) {
        int idx = j * width + i;

        double dx2 = DX * DX;
        double dy2 = DY * DY;
        double factor = 0.5 / (1.0/dx2 + 1.0/dy2);

        double left = u[idx - 1];
        double right = u[idx + 1];
        double bottom = u[idx - width];
        double top = u[idx + width];

        u_new[idx] = factor * ((left + right) / dx2 + (bottom + top) / dy2);
    }
}

// CUDA kernel для вычисления невязки (норма C)
__global__ void compute_residual_kernel(const double* u, double* residual, int width, int
height) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= 1 && i < width-1 && j >= 1 && j < height-1) {
        int idx = j * width + i;

        double dx2 = DX * DX;
        double dy2 = DY * DY;

        double laplacian = (u[idx - 1] - 2*u[idx] + u[idx + 1]) / dx2 +
                          (u[idx - width] - 2*u[idx] + u[idx + width]) / dy2;

        residual[idx] = fabs(laplacian);
    } else {
        // Для граничных точек невязка = 0
        int idx = j * width + i;
        residual[idx] = 0.0;
    }
}

// CUDA kernel для копирования граничных условий
__global__ void copy_boundaries_kernel(const double* u, double* u_new, int width, int he
ight) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= width || j >= height) return;

    int idx = j * width + i;

    // Копируем граничные условия
    if (i == 0 || i == width-1 || j == 0 || j == height-1) {
        u_new[idx] = u[idx];
    }
}
```

```
// CUDA kernel для инициализации граничных условий
__global__ void initialize_boundaries_kernel(double* u, int width, int height) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= width || j >= height) return;

    int idx = j * width + i;

    // Нижняя граница: u(x,0) = e^(1-x)
    if (j == 0) {
        u[idx] = exp(1.0 - i * DX);
    }
    // Верхняя граница: u(x,1) = 1.0
    else if (j == height-1) {
        u[idx] = 1.0;
    }
    // Левая граница: u(0,y) = e^(1-y)
    else if (i == 0) {
        u[idx] = exp(1.0 - j * DY);
    }
    // Правая граница: u(1,y) = 1.0
    else if (i == width-1) {
        u[idx] = 1.0;
    }
    // Внутренние точки
    else {
        u[idx] = 0.0;
    }
}

// Kernel для вычисления разностей между массивами
__global__ void compute_diff_kernel(const double* a, const double* b, double* diff, int
size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size) {
        diff[idx] = fabs(a[idx] - b[idx]);
    }
}

// Функция для вычисления нормы C (максимум) с использованием Thrust
double compute_norm_c(const double* d_data, int size) {
    thrust::device_ptr<const double> thrust_ptr(d_data);
    return thrust::reduce(thrust_ptr, thrust_ptr + size, 0.0, thrust::maximum<double>())
;
}

// Функция для вычисления нормы L1 с использованием Thrust
double compute_norm_l1(const double* d_data, int size) {
    thrust::device_ptr<const double> thrust_ptr(d_data);
    double sum = thrust::reduce(thrust_ptr, thrust_ptr + size, 0.0, thrust::plus<double>
());
    return sum * DX * DY;
}

// Функция для вычисления максимального изменения между итерациями
double compute_max_change(const double* d_u, const double* d_u_new, int size) {
    // Выделяем память для разностей
    double* d_diff;
    CHECK_CUDA_ERROR(cudaMalloc(&d_diff, size * sizeof(double)));

    // Вычисляем разности
    int blockSize1D = 256;
    int gridSize1D = (size + blockSize1D - 1) / blockSize1D;
    compute_diff_kernel<<<gridSize1D, blockSize1D>>>(d_u, d_u_new, d_diff, size);
    CHECK_CUDA_ERROR(cudaDeviceSynchronize());

    // Находим максимум с помощью Thrust
    double max_diff = compute_norm_c(d_diff, size);
    // Освобождаем память
    cudaFree(d_diff);

    return max_diff;
}

// Функция для вычисления невязки (норма C)
double compute_residual(const double* d_u, int width, int height) {
    // Выделяем память для невязки
    double* d_residual;
    CHECK_CUDA_ERROR(cudaMalloc(&d_residual, width * height * sizeof(double)));

    // Настройка размеров блоков и гридов
    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridSize((width + BLOCK_SIZE - 1) / BLOCK_SIZE,
                  (height + BLOCK_SIZE - 1) / BLOCK_SIZE);

    // Вычисляем невязку
    compute_residual_kernel<<<gridSize, blockSize>>>(d_u, d_residual, width, height);
    CHECK_CUDA_ERROR(cudaDeviceSynchronize());

    // Находим максимальную невязку
    double max_residual = compute_norm_c(d_residual, width * height);

    // Освобождаем память
    cudaFree(d_residual);

    return max_residual;
}

int main() {
    printf("Solving Laplace equation with CUDA\n");
    printf("Grid: %dx%d, dx=%.6f, dy=%.6f\n", NX, NY, DX, DY);
    printf("Tolerance: %.2e, Max iterations: %d\n", TOLERANCE, MAX_ITER);
    printf("Block size: %dx%d\n", BLOCK_SIZE, BLOCK_SIZE);

    // Выделение памяти на GPU
    double *d_u, *d_u_new;
    CHECK_CUDA_ERROR(cudaMalloc(&d_u, NX * NY * sizeof(double)));
    CHECK_CUDA_ERROR(cudaMalloc(&d_u_new, NX * NY * sizeof(double)));

    // Выделение памяти на CPU для вывода
    double* h_u = (double*)malloc(NX * NY * sizeof(double));

    // Настройка размеров блоков и гридов
    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridSize((NX + BLOCK_SIZE - 1) / BLOCK_SIZE,
                  (NY + BLOCK_SIZE - 1) / BLOCK_SIZE);

    printf("Grid size: %dx%d blocks\n", gridSize.x, gridSize.y);

    // Инициализация
    initialize_boundaries_kernel<<<gridSize, blockSize>>>(d_u, NX, NY);
    initialize_boundaries_kernel<<<gridSize, blockSize>>>(d_u_new, NX, NY);
    CHECK_CUDA_ERROR(cudaDeviceSynchronize());

    auto start_computation = std::chrono::high_resolution_clock::now();

    // Основной итерационный цикл
    int iter = 0;
    double residual = 0.0;
    double max_change_val = 0.0;

    printf("\nStarting iterations...\n");
    printf("Iter  Residual(C-norm)    Max-Change      L1-Norm\n");
    printf("--------------------------------------------------\n");

    for (iter = 1; iter <= MAX_ITER; iter++) {
        // Шаг Якоби
        jacobi_step_kernel<<<gridSize, blockSize>>>(d_u, d_u_new, NX, NY);
        CHECK_CUDA_ERROR(cudaDeviceSynchronize());
        // Копируем граничные условия
        copy_boundaries_kernel<<<gridSize, blockSize>>>(d_u, d_u_new, NX, NY);
        CHECK_CUDA_ERROR(cudaDeviceSynchronize());

        // Вычисляем невязку и изменения
        residual = compute_residual(d_u_new, NX, NY);
        max_change_val = compute_max_change(d_u, d_u_new, NX * NY);
        double l1_norm = compute_norm_l1(d_u_new, NX * NY);

        // Меняем массивы местами
        std::swap(d_u, d_u_new);

        // Вывод прогресса каждые 1000 итераций
        if (iter % 1000 == 0) {
            printf("%5d  %12.6e  %12.6e  %12.6e\n",
                   iter, residual, max_change_val, l1_norm);
        }

        // Критерий остановки (как в CPU версии)
        if (residual < TOLERANCE && max_change_val < TOLERANCE) {
            printf("%5d  %12.6e  %12.6e  %12.6e\n",
                   iter, residual, max_change_val, l1_norm);
            printf("Converged!\n");
            break;
        }
    }

    auto end_computation = std::chrono::high_resolution_clock::now();

    if (iter > MAX_ITER) {
        printf("Reached maximum iterations without convergence\n");
        printf("Current residual: %.6e (target: %.1e)\n", residual, TOLERANCE);
    }

    // Копируем результат обратно на CPU
    CHECK_CUDA_ERROR(cudaMemcpy(h_u, d_u, NX * NY * sizeof(double), cudaMemcpyDeviceToHo
st));

    // Вывод результатов
    printf("\n=== Final Results ===\n");
    printf("Iterations: %d\n", iter);
    printf("Final residual (C-norm): %.6e\n", residual);
    printf("Final max change: %.6e\n", max_change_val);
    printf("Final L1 norm: %.6e\n", compute_norm_l1(d_u, NX * NY));

    // Сохранение в файл
    FILE* file = fopen("solution_laplace_cuda.txt", "w");
    if (file) {
        fprintf(file, "# X Y U\n");
        for (int j = 0; j < NY; j++) {
            for (int i = 0; i < NX; i++) {
                int idx = j * NX + i;
                fprintf(file, "%.6f %.6f %.6f\n", i*DX, j*DY, h_u[idx]);
            }
        }
        fclose(file);
        printf("Saved to solution_laplace_cuda.txt\n");
    }

    // Анализ решения
    printf("\n=== Solution Analysis ===\n");
    printf("Boundary conditions check:\n");
    printf("u(0,0) = %.6f (should be e = %.6f)\n", h_u[0], exp(1.0));
    printf("u(1,0) = %.6f (should be 1.0)\n", h_u[NY-1], 1.0);
    printf("u(0,1) = %.6f (should be 1.0)\n", h_u[(NY-1)*NX], 1.0);
    printf("u(1,1) = %.6f (should be 1.0)\n", h_u[NY*NX-1], 1.0);

    printf("\nInternal points:\n");
    printf("u(0.5, 0.5) = %.6f\n", h_u[(NY/2)*NX + NX/2]);
    printf("u(0.25, 0.25) = %.6f\n", h_u[(NY/4)*NX + NX/4]);
    printf("u(0.75, 0.75) = %.6f\n", h_u[(3*NY/4)*NX + 3*NX/4]);
    auto computation_time = std::chrono::duration_cast<std::chrono::microseconds>(
        end_computation - start_computation);

    printf("\n=== CUDA Performance Results ===\n");
    printf("Total computation time: %.3f ms\n", computation_time.count() / 1000.0);
    printf("Time per iteration: %.3f microseconds\n", computation_time.count() / (double
)iter);
    printf("Iterations per second: %.0f\n", iter / (computation_time.count() / 1000000.0
));

    // Освобождение памяти
    free(h_u);
    cudaFree(d_u);
    cudaFree(d_u_new);

    return 0;
}
```

```
Overwriting /tmp/t2v1.cu
```

In [15]:

```
# @title
!nvcc -arch=sm_75 /tmp/t2v1.cu -o /tmp/t2v1
# run the executable with nvprof
!nvprof /tmp/t2v1
```

```
/tmp/t2v1.cu(289): warning #225-D: the format string ends before this argument
      printf("u(1,0) = %.6f (should be 1.0)\n", h_u[128 -1], 1.0);
                                                              ^


Remark: The warnings can be suppressed with "-diag-suppress <warning-number>"

/tmp/t2v1.cu(290): warning #225-D: the format string ends before this argument
      printf("u(0,1) = %.6f (should be 1.0)\n", h_u[(128 -1)*128], 1.0);
                                                                    ^


/tmp/t2v1.cu(291): warning #225-D: the format string ends before this argument
      printf("u(1,1) = %.6f (should be 1.0)\n", h_u[128*128 -1], 1.0);
                                                                  ^


/tmp/t2v1.cu: In function ‘int main()’:
/tmp/t2v1.cu:289:8: warning: too many arguments for format []8;;https://gcc.gnu.org/onlin
edocs/gcc/Warning-Options.html#index-Wformat-extra-args-Wformat-extra-args]8;;]
  289 |     printf("u(1,0) = %.6f (should be 1.0)\n", h_u[NY-1], 1.0);
      |        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/tmp/t2v1.cu:290:8: warning: too many arguments for format []8;;https://gcc.gnu.org/onlin
edocs/gcc/Warning-Options.html#index-Wformat-extra-args-Wformat-extra-args]8;;]
  290 |     printf("u(0,1) = %.6f (should be 1.0)\n", h_u[(NY-1)*NX], 1.0);
      |        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/tmp/t2v1.cu:291:8: warning: too many arguments for format []8;;https://gcc.gnu.org/onlin
edocs/gcc/Warning-Options.html#index-Wformat-extra-args-Wformat-extra-args]8;;]
  291 |     printf("u(1,1) = %.6f (should be 1.0)\n", h_u[NY*NX-1], 1.0);
      |        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Solving Laplace equation with CUDA
Grid: 128x128, dx=0.007874, dy=0.007874
Tolerance: 1.00e-05, Max iterations: 50000
Block size: 16x16
==1387== NVPROF is profiling process 1387, command: /tmp/t2v1
Grid size: 8x8 blocks

Starting iterations...
Iter  Residual(C-norm)    Max-Change      L1-Norm
--------------------------------------------------
 1000  4.825725e+01  7.485691e-04  6.747362e-01
 2000  2.282065e+01  3.537575e-04  8.815974e-01
 3000  1.647062e+01  2.553533e-04  1.015109e+00
 4000  1.249145e+01  1.936543e-04  1.109617e+00
 5000  9.287306e+00  1.439822e-04  1.178395e+00
 6000  6.855223e+00  1.062780e-04  1.228867e+00
 7000  5.049931e+00  7.829375e-05  1.265995e+00
```

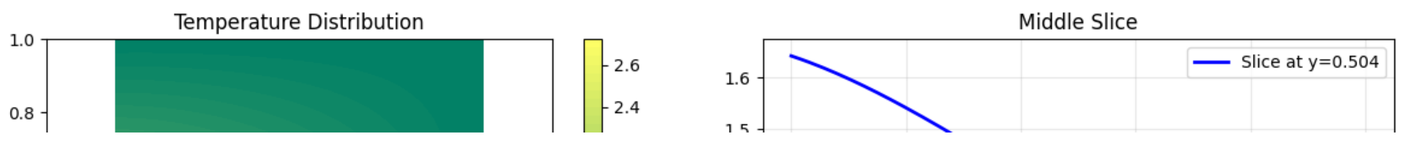
```
  8000  3.719094e+00  5.765806e-05  1.293328e+00
  9000  2.738657e+00  4.245810e-05  1.313454e+00
 10000  2.016613e+00  3.126406e-05  1.328275e+00
 11000  1.484947e+00  2.302152e-05  1.339188e+00
 12000  1.093468e+00  1.695284e-05  1.347225e+00
 13000  8.052044e-01  1.248398e-05  1.353143e+00
 14000  5.929389e-01  9.193138e-06  1.357501e+00
 15000  4.366326e-01  6.769775e-06  1.360711e+00
 16000  3.215317e-01  4.985224e-06  1.363074e+00
 17000  2.367732e-01  3.671091e-06  1.364814e+00
 18000  1.743580e-01  2.703370e-06  1.366096e+00
 19000  1.283961e-01  1.990746e-06  1.367039e+00
 20000  9.455003e-02  1.465974e-06  1.367734e+00
 21000  6.962606e-02  1.079535e-06  1.368246e+00
 22000  5.127221e-02  7.949636e-07  1.368623e+00
 23000  3.775656e-02  5.854066e-07  1.368901e+00
 24000  2.780371e-02  4.310901e-07  1.369105e+00
 25000  2.047449e-02  3.174523e-07  1.369255e+00
 26000  1.507730e-02  2.337701e-07  1.369366e+00
 27000  1.110284e-02  1.721470e-07  1.369448e+00
 28000  8.176065e-03  1.267680e-07  1.369508e+00
 29000  6.020808e-03  9.335126e-08  1.369552e+00
 30000  4.433688e-03  6.874334e-08  1.369585e+00
 31000  3.264943e-03  5.062220e-08  1.369609e+00
 32000  2.404285e-03  3.727790e-08  1.369626e+00
 33000  1.770502e-03  2.745123e-08  1.369639e+00
 34000  1.303788e-03  2.021493e-08  1.369649e+00
 35000  9.601018e-04  1.488616e-08  1.369656e+00
 36000  7.070135e-04  1.096208e-08  1.369661e+00
 37000  5.206407e-04  8.072417e-09  1.369665e+00
 38000  3.833969e-04  5.944482e-09  1.369668e+00
 39000  2.823313e-04  4.377483e-09  1.369670e+00
 40000  2.079071e-04  3.223553e-09  1.369672e+00
 41000  1.531016e-04  2.373806e-09  1.369673e+00
 42000  1.127432e-04  1.748057e-09  1.369674e+00
 43000  8.302344e-05  1.287260e-09  1.369674e+00
 44000  6.113799e-05  9.479306e-10  1.369675e+00
 45000  4.502170e-05  6.980512e-10  1.369675e+00
 46000  3.315375e-05  5.140413e-10  1.369675e+00
 47000  2.441420e-05  3.785368e-10  1.369675e+00
 48000  1.797849e-05  2.787524e-10  1.369675e+00
 49000  1.323928e-05  2.052720e-10  1.369676e+00
 49918  9.997030e-06  1.550018e-10  1.369676e+00
Converged!

=== Final Results ===
Iterations: 49918
Final residual (C-norm): 9.997030e-06
Final max change: 1.550018e-10
Final L1 norm: 1.369676e+00
Saved to solution_laplace_cuda.txt

=== Solution Analysis ===
Boundary conditions check:
u(0,0) = 2.718282 (should be e = 2.718282)
u(1,0) = 1.000000 (should be 1.0)
u(0,1) = 1.000000 (should be 1.0)
u(1,1) = 1.000000 (should be 1.0)

Internal points:
u(0.5, 0.5) = 1.337383
u(0.25, 0.25) = 1.815447
u(0.75, 0.75) = 1.080496

=== CUDA Performance Results ===
Total computation time: 18305.850 ms
Time per iteration: 366.718 microseconds
Iterations per second: 2727
==1387== Profiling application: /tmp/t2v1
==1387== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   19.66%  758.31ms     99836  7.5950us  7.4870us  7.7750us  void cub::CU
```

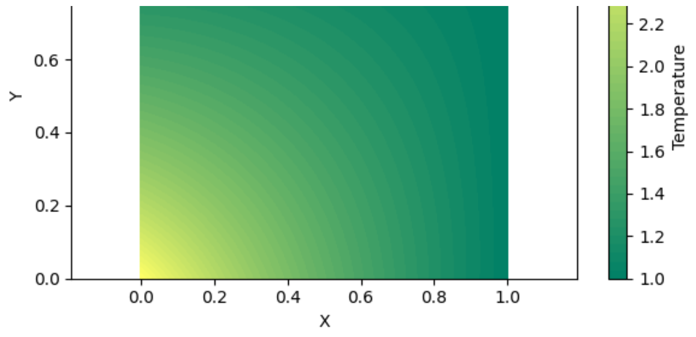
```
B_200400_750_NS::DeviceReduceKernel<cub::CUB_200400_750_NS::DeviceReducePolicy<double, un
signed int, thrust::THRUST_200400_750_NS::maximum<double>>::Policy600, thrust::THRUST_200
400_750_NS::device_ptr<double const >, unsigned int, thrust::THRUST_200400_750_NS::maximu
m<double>, double, cuda::std::__4::__identity>(unsigned int, cub::CUB_200400_750_NS::Devi
ceReducePolicy<double, unsigned int, thrust::THRUST_200400_750_NS::maximum<double>>::Poli
cy600*, double, cub::CUB_200400_750_NS::GridEvenShare<cub::CUB_200400_750_NS::DeviceReduc
ePolicy<double, unsigned int, thrust::THRUST_200400_750_NS::maximum<double>>::Policy600*>
, thrust::THRUST_200400_750_NS::maximum<double>, double const )
                   18.09%  697.57ms     49918  13.974us  13.887us  25.759us  compute_res
idual_kernel(double const *, double*, int, int)
                   16.11%  621.17ms     49918  12.443us  12.383us  24.160us  jacobi_step
_kernel(double const *, double*, int, int)
                   13.97%  538.57ms     99836  5.3940us  5.0870us  5.8880us  void cub::C
UB_200400_750_NS::DeviceReduceSingleTileKernel<cub::CUB_200400_750_NS::DeviceReducePolicy
<double, unsigned int, thrust::THRUST_200400_750_NS::maximum<double>>::Policy600, double*
, double*, int, thrust::THRUST_200400_750_NS::maximum<double>, double, double, cuda::std:
:__4::__identity>(unsigned int, double, thrust::THRUST_200400_750_NS::maximum<double>, cu
b::CUB_200400_750_NS::DeviceReducePolicy<double, unsigned int, thrust::THRUST_200400_750_
NS::maximum<double>>::Policy600, double*, int)
                    9.40%  362.51ms     49919  7.2620us  7.1350us  7.4870us  void cub::C
UB_200400_750_NS::DeviceReduceKernel<cub::CUB_200400_750_NS::DeviceReducePolicy<double, u
nsigned int, thrust::THRUST_200400_750_NS::plus<double>>::Policy600, thrust::THRUST_20040
0_750_NS::device_ptr<double const >, unsigned int, thrust::THRUST_200400_750_NS::plus<dou
ble>, double, cuda::std::__4::__identity>(unsigned int, cub::CUB_200400_750_NS::DeviceRed
ucePolicy<double, unsigned int, thrust::THRUST_200400_750_NS::plus<double>>::Policy600*,
double, cub::CUB_200400_750_NS::GridEvenShare<cub::CUB_200400_750_NS::DeviceReducePolicy<
double, unsigned int, thrust::THRUST_200400_750_NS::plus<double>>::Policy600*>, thrust::T
HRUST_200400_750_NS::plus<double>, double const )
                    6.95%  267.84ms     49919  5.3650us  4.8320us  5.7270us  void cub::C
UB_200400_750_NS::DeviceReduceSingleTileKernel<cub::CUB_200400_750_NS::DeviceReducePolicy
<double, unsigned int, thrust::THRUST_200400_750_NS::plus<double>>::Policy600, double*, d
ouble*, int, thrust::THRUST_200400_750_NS::plus<double>, double, double, cuda::std::__4::
__identity>(unsigned int, double, thrust::THRUST_200400_750_NS::plus<double>, cub::CUB_20
0400_750_NS::DeviceReducePolicy<double, unsigned int, thrust::THRUST_200400_750_NS::plus<
double>>::Policy600, double*, int)
                    6.48%  249.98ms    149756  1.6690us  1.6310us  11.584us  [CUDA memcp
y DtoH]
                    5.57%  214.65ms     49918  4.3000us  4.1920us  4.4480us  compute_dif
f_kernel(double const *, double const *, double*, int)
                    3.78%  145.69ms     49918  2.9180us  2.8480us  2.9760us  copy_bounda
ries_kernel(double const *, double*, int, int)
                    0.00%  26.336us         2  13.168us  12.960us  13.376us  initialize_
boundaries_kernel(double*, int, int)
      API calls:   27.16%  4.18863s    499184  8.3900us  3.3270us  4.5818ms  cudaLaunchKe
rnel
                   18.58%  2.86506s    199673  14.348us  2.8330us  4.3109ms  cudaDeviceS
ynchronize
                   15.86%  2.44611s    149755  16.334us  10.926us  5.0397ms  cudaMemcpyA
sync
                   10.48%  1.61643s    299510  5.3960us  1.4470us  6.1797ms  cudaStreamS
ynchronize
                    7.68%  1.18419s    249593  4.7440us  1.5750us  4.0849ms  cudaFree
                    7.67%  1.18349s    249593  4.7410us  1.8420us  89.263ms  cudaMalloc
                    6.16%  950.48ms   4193143     226ns      80ns  4.0196ms  cudaGetLast
Error
                    2.20%  338.55ms    599021     565ns     198ns  4.0192ms  cudaGetDevi
ce
                    2.07%  319.12ms    299510  1.0650us     374ns  4.0188ms  cudaOccupan
cyMaxActiveBlocksPerMultiprocessorWithFlags
                    1.34%  206.58ms    299510     689ns     216ns  4.0149ms  cudaDeviceG
etAttribute
                    0.81%  124.67ms    599020     208ns      82ns  3.5584ms  cudaPeekAtL
astError
                    0.00%  154.97us       114  1.3590us     118ns  67.864us  cuDeviceGet
Attribute
                    0.00%  131.17us         1  131.17us  131.17us  131.17us  cudaMemcpy
                    0.00%  13.715us         1  13.715us  13.715us  13.715us  cuDeviceGet
Name
                    0.00%  8.8240us         1  8.8240us  8.8240us  8.8240us  cudaFuncGet
Attributes
                    0.00%  6.0470us         1  6.0470us  6.0470us  6.0470us  cuDeviceGet
PCIBusId
```

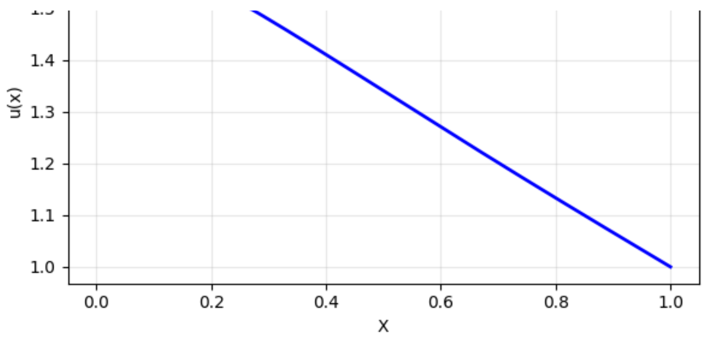
```
                    0.00%  1.7330us         3     577ns     159ns  1.3070us  cuDeviceGet
Count
                    0.00%     746ns         2     373ns     216ns     530ns  cuDeviceGet
                    0.00%     641ns         1     641ns     641ns     641ns  cuDeviceTot
alMem
                    0.00%     462ns         1     462ns     462ns     462ns  cuDeviceGet
Uuid
                    0.00%     420ns         1     420ns     420ns     420ns  cuModuleGet
LoadingMode
                    0.00%     299ns         1     299ns     299ns     299ns  cudaGetDevi
ceCount
```

In [16]:

```python
# @title
import numpy as np
import matplotlib.pyplot as plt

def quick_plot1():
    try:
        data = np.loadtxt('solution_laplace_cuda.txt', comments='#')
        x = data[:, 0]
        y = data[:, 1]
        u = data[:, 2]

        # Преобразование в сетку
        x_unique = np.unique(x)
        y_unique = np.unique(y)
        nx = len(x_unique)
        ny = len(y_unique)

        u_grid = u.reshape(nx, ny)
        X, Y = np.meshgrid(x_unique, y_unique, indexing='ij')

        plt.figure(figsize=(12, 4))

        plt.subplot(1, 2, 1)
        #plt.contourf(X, Y, u_grid, levels=50, cmap='cividis')
        plt.contourf(X, Y, u_grid, levels=50, cmap='summer')
        #plt.contourf(X, Y, u_grid, levels=50, cmap='Reds')
        plt.colorbar(label='Temperature')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('Temperature Distribution')
        plt.axis('equal')

        plt.subplot(1, 2, 2)
        middle_y = ny // 2
        plt.plot(x_unique, u_grid[:, middle_y], 'b-', linewidth=2, label=f'Slice at y={y_
unique[middle_y]:.3f}')
        plt.xlabel('X')
        plt.ylabel('u(x)')
        plt.title('Middle Slice')
        plt.grid(True, alpha=0.3)
        plt.legend()

        plt.tight_layout()
        plt.show()

        print(f"Min temperature: {u_grid.min():.4f}")
        print(f"Max temperature: {u_grid.max():.4f}")
        print(f"Center temperature: {u_grid[nx//2, ny//2]:.4f}")

    except Exception as e:
        print(f"Error: {e}")

quick_plot1()
```







```
Min temperature: 1.0000
Max temperature: 2.7183
Center temperature: 1.3374
```

In [ ]:

```
```

other

In [ ]:

```
```