

Лекция 2

- Обработка ошибок.
- Профилирование кода.

Программный интерфейс CUDA

CUDA C - расширение языка *C* , и набор функций и структур *CUDA API* предоставляют простой инструмент для программирования на GPU.

Некоторые конструкции программного интерфейса CUDA.

Функция-ядро (*kernel*)

Код, выполняемый на устройстве (ядро), определяется в виде функции типа *void* со спецификатором `__global__`:

```
__global__ void gFunc(<params>){...}
```

Конфигурация нитей

При вызове ядра программист определяет количество нитей в блоке и количество блоков в `grid`. При этом допустима линейная, двумерная или трехмерная индексация нитей:

```
gFunc<<<dim3(bl_xdim, bl_ydim, bl_zdim),  
           dim3(th_xdim, th_ydim, th_zdim)>>>(<params>);
```

Программный интерфейс CUDA (самый простой пример)

```
#include <cuda.h>
#include <stdio.h>

__global__ void gTest(float* a){
    a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
}

int main(){
    float *da, *ha;
    int num_of_blocks=10, threads_per_block=32;
    int N=num_of_blocks*threads_per_block;

    ha=(float*)calloc(N, sizeof(float));
    cudaMalloc((void**)&da, N*sizeof(float));

    gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
    //cudaThreadSynchronize();
    CudaDeviceSynchronize();
    cudaMemcpy(ha,da,N*sizeof(float), cudaMemcpyDeviceToHost);

    for(int i=0;i<N;i++)
        printf("%g\n", ha[i]);

    free(ha);
    cudaFree(da);
    return 0;
}
```

CUDA C - расширение языка C , и набор функций и структур CUDA API предоставляют простой инструмент для программирования на GPU.

Комментарии: использование глобальной памяти

```
cudaError_t  cudaMalloc (void ** devPtr, size_t size)
```

```
cudaError_t  cudaFree (void * devPtr)
```

```
cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)
```

```
enum cudaError
{
    cudaSuccess = 0,
    cudaErrorMissingConfiguration,
    cudaErrorMemoryAllocation,
    cudaErrorInitializationError,
    cudaErrorLaunchFailure,
    .....
};

typedef enum cudaError cudaError_t;
```

```
enum cudaMemcpyKind
{
    cudaMemcpyHostToHost = 0,
    cudaMemcpyHostToDevice,
    cudaMemcpyDeviceToHost,
    cudaMemcpyDeviceToDevice
};
```

Глобальная память **выделяется только на хосте**, к глобальной памяти возможен **доступ только на устройстве**.

Документация CUDA: <http://docs.nvidia.com/cuda/index.html>

Комментарии: встроенные типы и переменные

- uint3 **threadIdx** – индекс нити в блоке
- dim3 **blockDim** – размер блока
- uint3 **blockIdx** – индекс блока в гриде
- dim3 **gridDim** - размер грида
- int **warpSize** - количество нитей в варпе (warp)

Комментарии: синхронизация всех нитей

Запуск ядра на устройстве (вызов функции с модификатором `__global__`) происходит в асинхронном режиме.

Для синхронизации нитей служат следующие вызовы:

```
//cudaThreadSynchronize(); //устаревшая функция (depricated)  
cudaDeviceSynchronize();
```

Обработка ошибок (код с обработчиком ошибок)

```
#include <cuda.h>
#include <stdio.h>

#define CUDA_CHECK_RETURN(value) {\
    cudaError_t _m_cudaStat = value;\
    if (_m_cudaStat != cudaSuccess) {\
        fprintf(stderr, "Error %s at line %d in file %s\n",\
            cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);\
        exit(1);\
    } } //макрос для обработки ошибок

__global__ void gTest(float* a){
    a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
}

int main(){
    float *da, *ha;
    int num_of_blocks=10, threads_per_block=2048;
    int N=num_of_blocks*threads_per_block;

    ha=(float*)calloc(N, sizeof(float));
    CUDA_CHECK_RETURN(cudaMalloc((void**)&da, N*sizeof(float)));
```

Обработка ошибок (код с обработчиком ошибок, продолжение)

```
gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);  
// cudaThreadSynchronize();  
CUDA_CHECK_RETURN(cudaDeviceSynchronize());  
CUDA_CHECK_RETURN(cudaGetLastError());  
CUDA_CHECK_RETURN(cudaMemcpy(ha,da,N*sizeof(float), cudaMemcpyDeviceToHost));  
  
for(int i=0;i<N;i++)  
    printf("%g\n", ha[i]);  
  
free(ha);  
cudaFree(da);  
return 0;  
}
```

```
bash-4.2$ nvcc 1.cu -o 1
```

```
bash-4.2$ ./1
```

```
Error invalid configuration argument at line 28 in file 1.cu
```


Обработка ошибок (комментарии)

char * cudaGetErrorString (cudaError_t code); - возвращает строку в таблице кодов ошибок.

cudaError_t cudaGetLastError (); - возвращает код ошибки при асинхронном вызове.

Контроль производительности (код с контрольными точками)

```
#include <cuda.h>
#include <stdio.h>

#define CUDA_CHECK_RETURN(value) {
    cudaError_t _m_cudaStat = value;
    if (_m_cudaStat != cudaSuccess) {
        fprintf(stderr, "Error %s at line %d in file %s\n",
            cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);
        exit(1);
    }
}

__global__ void gTest(float* a){
    a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
}

int main(){
    float *da, *ha;
    int num_of_blocks=10, threads_per_block=32;
    int N=num_of_blocks*threads_per_block;

    ha=(float*)calloc(N, sizeof(float));
    CUDA_CHECK_RETURN(cudaMalloc((void**)&da, N*sizeof(float)));
```

Контроль производительности (код с контрольными точками, продолжение)

```
cudaEvent_t start, stop; // встроенный тип данных – структура, для фиксации контрольных точек
float elapsedTime;
```

```
cudaEventCreate(&start); // инициализация
cudaEventCreate(&stop); // событий
```

```
cudaEventRecord(start, 0); // привязка события
```

```
gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
// cudaThreadSynchronize();
//CUDA_CHECK_RETURN(cudaDeviceSynchronize());
//CUDA_CHECK_RETURN(cudaGetLastError());
```

```
cudaEventRecord(stop, 0); // привязка события
cudaEventSynchronize(stop); // синхронизация по событию
cudaEventElapsedTime(&elapsedTime, start, stop); // вычисление затраченного времени
```

```
fprintf(stderr, "gTest took %g\n", elapsedTime);
```

```
cudaEventDestroy(start); // освобождение
cudaEventDestroy(stop); // памяти
```

Контроль производительности (код с контрольными точками, окончание)

```
CUDA_CHECK_RETURN(cudaMemcpy(ha,da,N*sizeof(float), cudaMemcpyDeviceToHost));

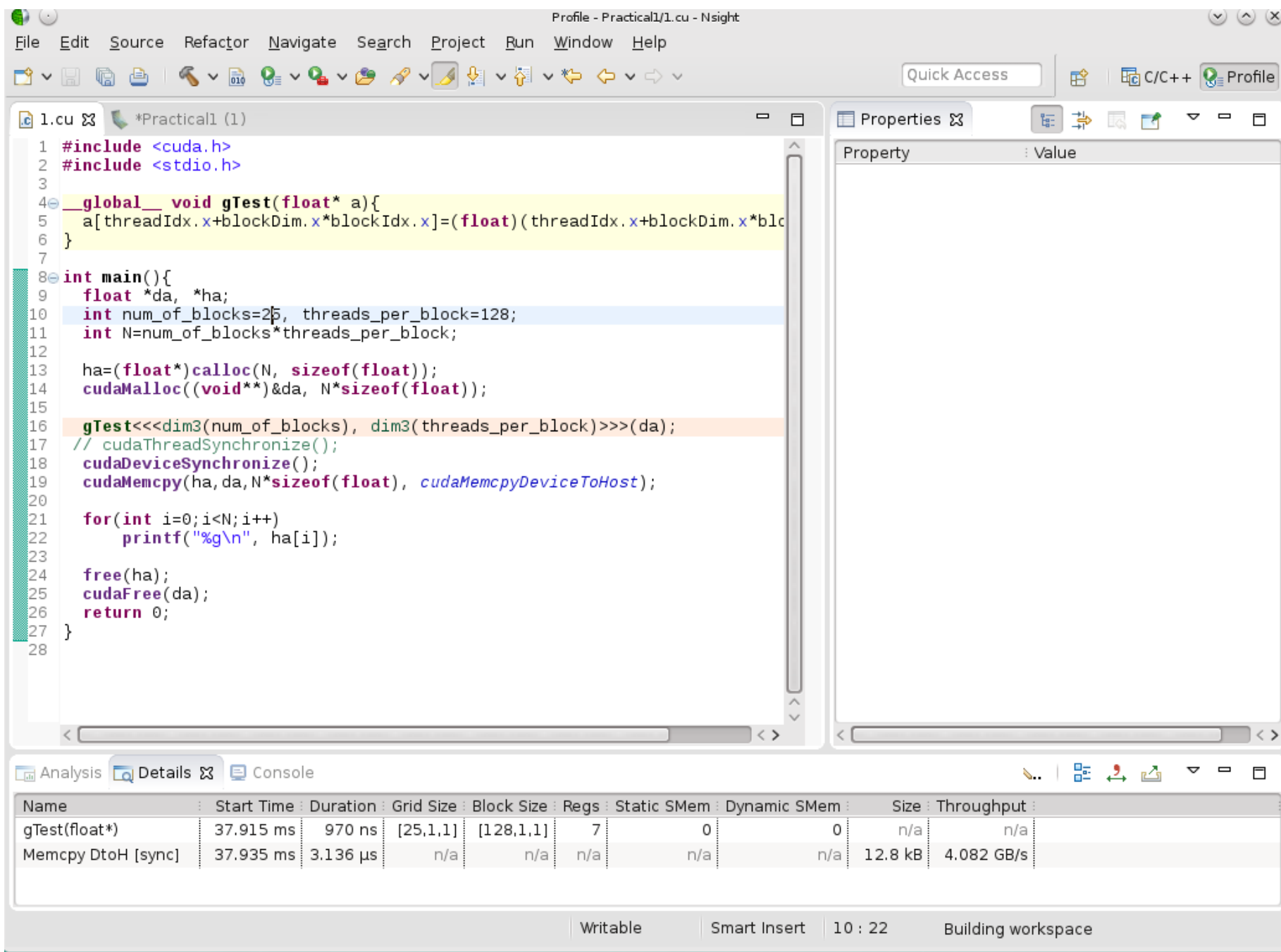
for(int i=0;i<N;i++)
    printf("%g\n", ha[i]);

free(ha);
cudaFree(da);
return 0;
}
```

```
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> ./3>tmp
gTest took 0.030656
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> time(./3>tmp)
gTest took 0.030848

real    0m0.086s
user    0m0.006s
sys     0m0.058s
```

Контроль производительности (профилировщик в Nsight)



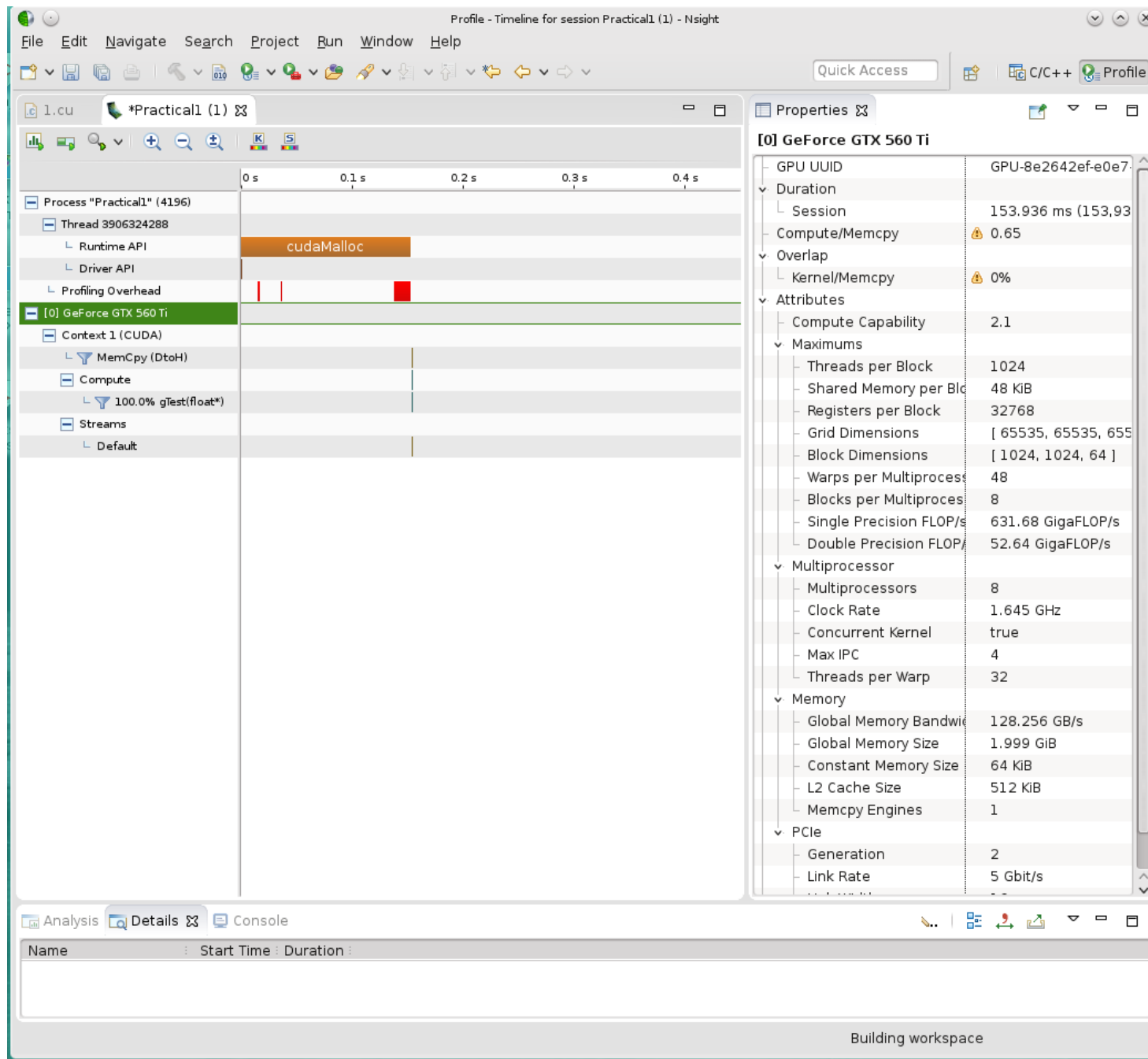
The screenshot displays the NVIDIA Nsight system profiler interface. The main window shows the source code of a CUDA program (1.cu) with the following content:

```
1 #include <cuda.h>
2 #include <stdio.h>
3
4 __global__ void gTest(float* a){
5     a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
6 }
7
8 int main(){
9     float *da, *ha;
10    int num_of_blocks=25, threads_per_block=128;
11    int N=num_of_blocks*threads_per_block;
12
13    ha=(float*)calloc(N, sizeof(float));
14    cudaMalloc((void**)&da, N*sizeof(float));
15
16    gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
17    // cudaThreadSynchronize();
18    cudaDeviceSynchronize();
19    cudaMemcpy(ha, da, N*sizeof(float), cudaMemcpyDeviceToHost);
20
21    for(int i=0; i<N; i++)
22        printf("%g\n", ha[i]);
23
24    free(ha);
25    cudaFree(da);
26    return 0;
27 }
28
```

The right-hand pane shows the Properties window, which is currently empty. The bottom pane displays the Analysis Details, showing a table of performance metrics for the executed kernel and memory copy operations.

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
gTest(float*)	37.915 ms	970 ns	[25,1,1]	[128,1,1]	7	0	0	n/a	n/a
Memcpy DtoH [sync]	37.935 ms	3.136 µs	n/a	n/a	n/a	n/a	n/a	12.8 kB	4.082 GB/s

The bottom status bar indicates the current state: Writable, Smart Insert, 10 : 22, and Building workspace.



Компиляторы

Сравнение компиляторов nvcc и pgi:

```
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> pgcpp 3.cu -o 3p
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> nvcc 3.cu -o 3n
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> time(./3p>tmp)
gTest took 0.002405
```

```
real 0m0.025s
user 0m0.025s
sys 0m0.001s
```

```
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> time(./3n>tmp)
gTest took 0.004992
```

```
real 0m0.113s
user 0m0.011s
sys 0m0.058s
```

Пример строки компиляции nvcc (компиляция и компоновка dll):

```
librelaxation.so: relaxation.cpp kerns_st.cu
nvcc -arch=compute_20 -Xcompiler -fPIC -shared relaxation.cpp kerns_st.cu -I../include
-I/home/ewgenij/common/inc -L/home/ewgenij/common/lib -lcutil_x86_64 -o ../lib/librelaxation.so
```

Упражнения

- Сравнить время выполнения последовательного и параллельного кода инициализации и сложения двух векторов большой размерности на выбранном CPU и GPU, установленными на серверах home, dew и кластере clu.nusc.ru.
- То же самое для задачи транспонирования матрицы.