# Optimization techniques for CUDA parallel processing- report

Tomasz Gajger

## Problem statement

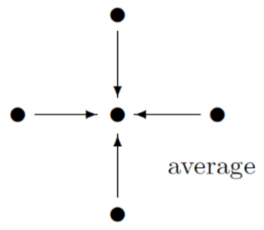
Application created for the purpose of this project is a simulation of heat distribution in a solid. Finite difference method was used to discretize[1] steady state differential equation describing heat diffusion in the object, environment temperature, which is a boundary condition was assumed to be constant.

Additionally, the object is assumed to be a two-dimensional square mesh, with a side of size *N*, so there are N*N distinct points on the mesh for which the value of the temperature has to be updated during each simulation step.

The final equation, applied to each cell, is as follows:

$$T_{n,m}^{k+1} = \frac{T_{n-1,m}^{k} + T_{n+1,m}^{k} + T_{n,m-1}^{k} + T_{n,m+1}^{k}}{4}$$

where:
*T* – cell temperature
*k* – iteration step
*n,m* – cell coordinates



If the cell is on the border of the mesh, then temperature of its nonexistent neighbor, which would be located outside of the mesh, is assumed to be some arbitrary constant (as stated at the very beginning).

[1] https://people.ucalgary.ca/~hugo/WEBPAGES/heat%20transfer/heattransfer_lecture_list.html#tag12

## Initial implementation

Below a code for initial implementation is listed, worth noticing is the *getElem* function responsible for translating 2D address into linear address space which was used[2] for memory allocation both on the host and the device.

Two memory buffers were allocated on the device, *mesh_in* and *mesh_out*, used respectively for reading and writing temperature values. At the end of each simulation step these pointers are swapped so that the output from previous iteration becomes input to the next one.

```
template <typename T>
__device__ inline T* getElem(T* BaseAddress, size_t pitch, unsigned Row, unsigned Column) {
   return reinterpret_cast<T*>(reinterpret_cast<char*>(BaseAddress) + Row * pitch) + Column;
}

__global__ void meshUpdateKernel(float* mesh_in, float* mesh_out, size_t pitch, unsigned
size) {
   const auto x = blockIdx.x * blockDim.x + threadIdx.x;
   const auto y = blockIdx.y * blockDim.y + threadIdx.y;

   if ( x > 0 && x < size - 1 && y > 0 && y < size - 1) {
      const float t_left = *getElem(mesh_in, pitch, y, x - 1);
      const float t_right = *getElem(mesh_in, pitch, y, x + 1);
      const float t_top = *getElem(mesh_in, pitch, y - 1, x);
      const float t_bottom = *getElem(mesh_in, pitch, y + 1, x);

      const auto newTemperature = (t_left + t_right + t_top + t_bottom) / 4;

      *getElem(mesh_out, pitch, y, x) = newTemperature;
   }
}
```

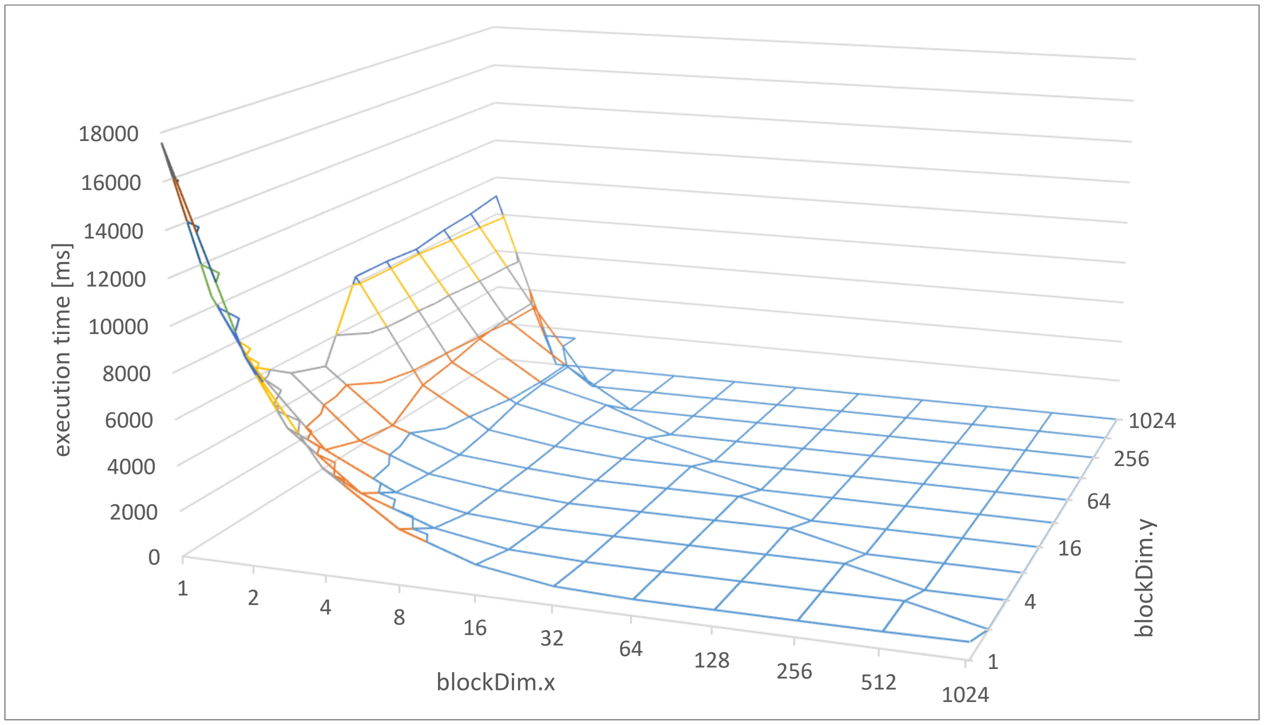
### Selecting optimal block size

The next step was to launch the simulation using different block size configuration to determine the optimal one. In the chart below it is clearly visible that the execution time is better for bigger values of blockDim.x. It was decided that the application profiling will be performed for blockDim.x = 128 and blockDim.y = 1 as such configuration yields the best results.

Execution time value of 0 indicates that such configuration is invalid, the constraint imposed by CUDA runtime is that:

$$blockDim.x \times blockDim.y \leq 1024$$

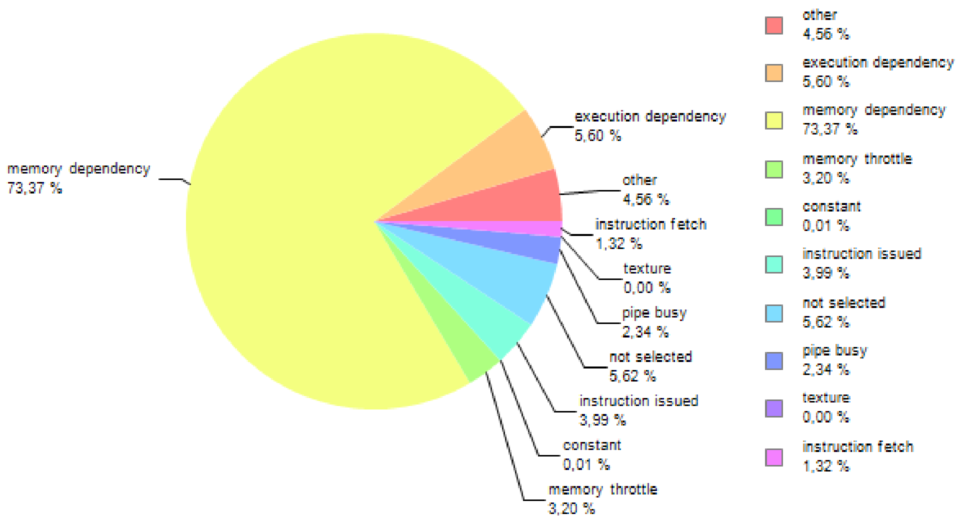
[2] http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g32bd7a39135594788a542ae72217775c



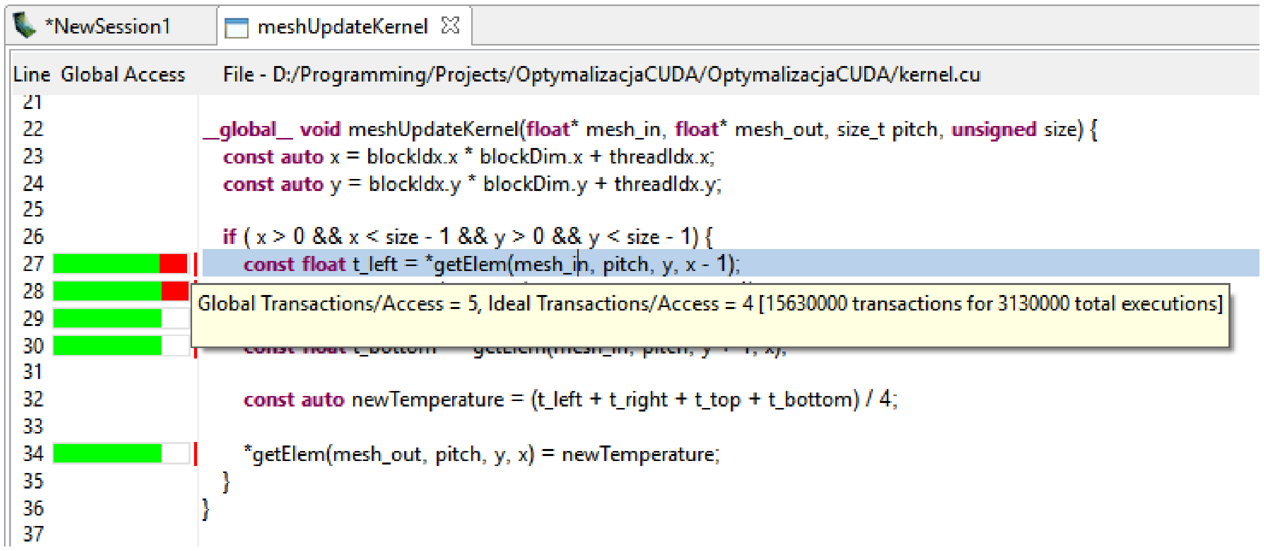
Measurements were performed for k=100 N=10000

## Profiling the application

Profiling was performed using Nvidia Visual Profiler for k = 100 N = 10000 blockDim.x = 128 blockDim.y = 1. Chart below shows that the accesses to the memory contribute for the vast majority of kernel execution time. It is quite expected, a short glance on the code presented earlier reveals its simplicity, each thread performs: a few arithmetic operations, 4 reads from global memory and one write. As a consequence, there is not much place for optimization, usage of shared memory does not yield any satisfying results (it was verified), also instruction reordering in the kernel code didn't result in a decrease of execution time.



There is a suboptimal global memory access pattern that can possibly be improved, but it has yet to be verified and the main focus will now be on creation of hybrid version of the code, i.e. one that utilizes both the GPU (possibly multiple) and CPU available on the host as this will have much bigger impact on the achieved speed-up.



File - D:/Programming/Projects/OptymalizacjaCUDA/OptymalizacjaCUDA/kernel.cu

```
__global__ void meshUpdateKernel(float* mesh_in, float* mesh_out, size_t pitch, unsigned size) {
  const auto x = blockIdx.x * blockDim.x + threadIdx.x;
  const auto y = blockIdx.y * blockDim.y + threadIdx.y;

  if ( x > 0 && x < size - 1 && y > 0 && y < size - 1) {
    const float t_left = *getElem(mesh_in, pitch, y, x - 1);
    const float t_bottom = *getElem(mesh_in, pitch, y + 1, x);

    const auto newTemperature = (t_left + t_right + t_top + t_bottom) / 4;

    *getElem(mesh_out, pitch, y, x) = newTemperature;
  }
}
```

Global Transactions/Access = 5, Ideal Transactions/Access = 4 [15630000 transactions for 3130000 total executions]

## Hybrid implementation

In hybrid version the mesh is divided in a way that upper $N \times ratio$ rows are assigned to the CPU and lower $N \times (1 - ratio)$ to the GPU where $ratio \in (0; 1)$. At the end of every simulation step a lowermost row owned by CPU and uppermost owned by GPU need to be exchanged. The rest of the mesh is transferred only at the beginning and at the end of simulation to significantly reduce the overhead caused by data transfers. Two versions of hybrid code running on CPU and GPU were tested, one with nested parallelism and one without. Application launch parameters are as follows:

```
./cuda-heat <STEPS> <MESH_SIZE> <NUM_THREADS> <BLOCK_DIM_X> <BLOCK_DIM_Y> <RATIO>
```

### Observations

- On a CPU with 4 physical cores and 8 logical threads the optimal number of threads to launch equals 4.
- On a CPU with 8 logical cores launching 7 threads results in shorter execution time than when launching 8. 8th thread is simply busy with other tasks scheduled by the system and thus cannot focus fully on computation and cripples the performance of the entire app.
- Hybrid implementatino with 100% share of computation assigned to GPU behaves almost the same as pure-CUDA implementation in terms of execution time. This means that the synchronization done on the host that incurs two copies of one row from the mesh between the host and the device is not a costly operation. Cuda  version is slightly faster (few milliseconds) because it just swaps input and output buffers directly on the device, without a need to copy any data. The synchronization on the host is done only for the kernel launch itself. Profiler output confirms this.
- For Intel i7-4790k and Nvidia GeForce 970GTX the optimal ratio seems to be 12% CPU, 88% GPU

- The version using nested parallelism performs worse when compared to the one without it. The difference is not really big though, few percent.
- Hybrid version is noticably faster for larger data sizes (7000+), the number of simulation steps does not play a significant role here execpt the fact that for a very small number of steps the overhead of initial and final data copy dominates over the computation.

## How the numer of threads affects the overall execution time:

| threads | execution time [ms] |
|---:|---:|
| 1 | 28207 |
| 2 | 21318 |
| 3 | 21665 |
| 4 | **21078** |
| 5 | 21304 |
| 6 | 21434 |
| 7 | 21708 |
| 8 | **23428** |

Measurements were performed for: Iterations: 1000; Size: 20000; Block dim x,y: 128, 1; Ratio: 0.12

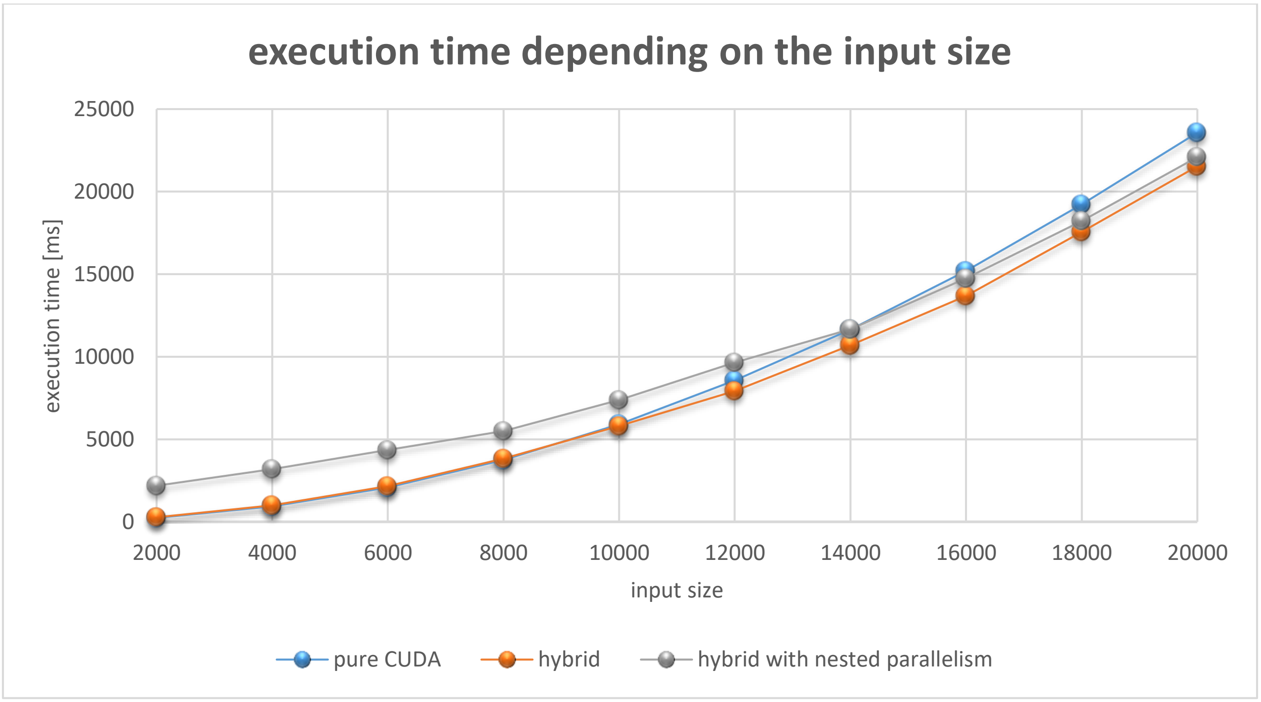
As we can see in the table above, optimal number of threads for this configuration is 4, this result is not so suprising considering that the CPU in the testbed is Intel Core i7-4790k which has 4 physical and 8 logical cores. Using all of the available threads is not a good idea as we should leave at least one of them for tasks scheduled by the operating system.

| threads | pure CPU [ms] | hybrid [ms] |
|---:|---:|---:|
| 1 | 22747 | 3021 |
| 2 | 16494 | 2299 |
| 3 | 17383 | 2330 |
| 4 | 15907 | 2296 |
| 5 | 16448 | 2329 |
| 6 | 15966 | 2345 |
| 7 | 16121 | 2453 |
| 8 | 16148 | 2452 |

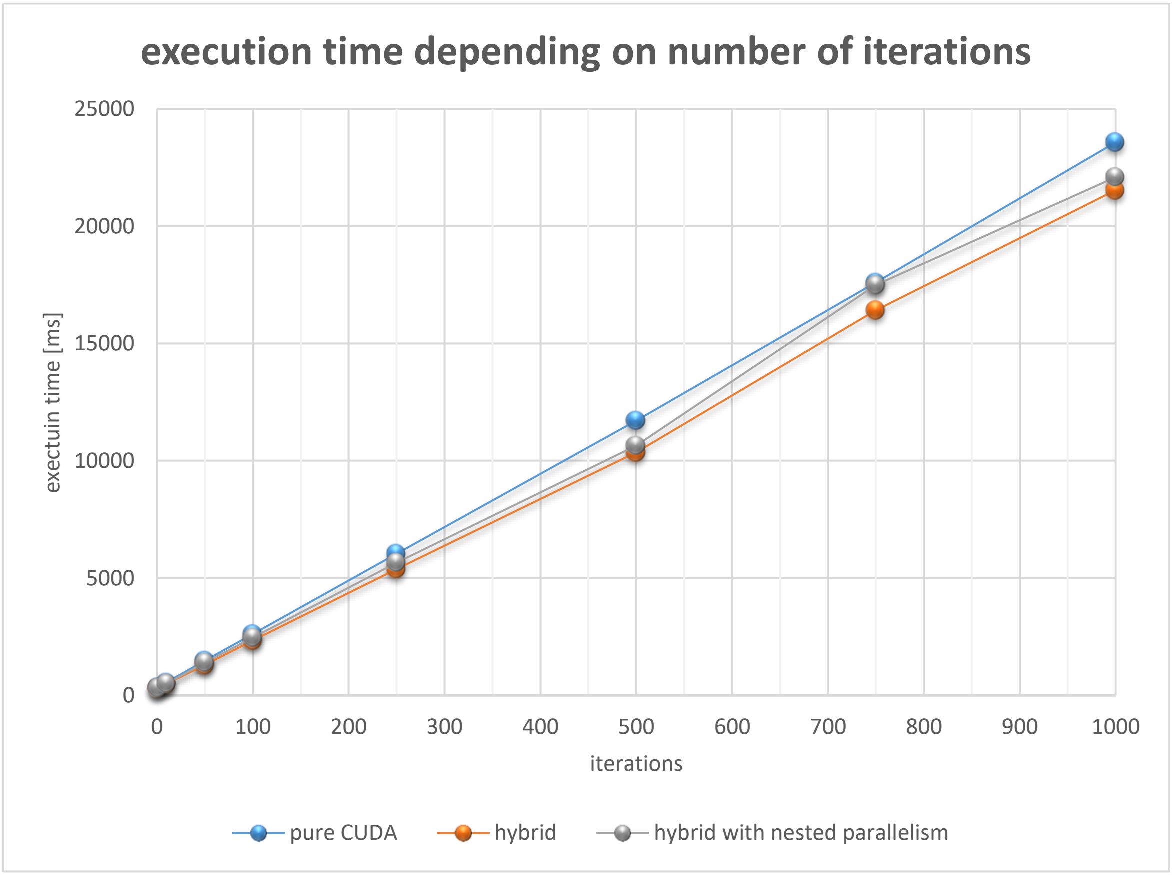
Measurements were performed for: Iterations: 100; Size: 20000; Block dim x,y: 128, 1; Ratio: 0.12

Looking at the table above we can easily tell that the pure CPU version does not scale really well and likely there is a field for improvement here.

## Execution time for various implementations



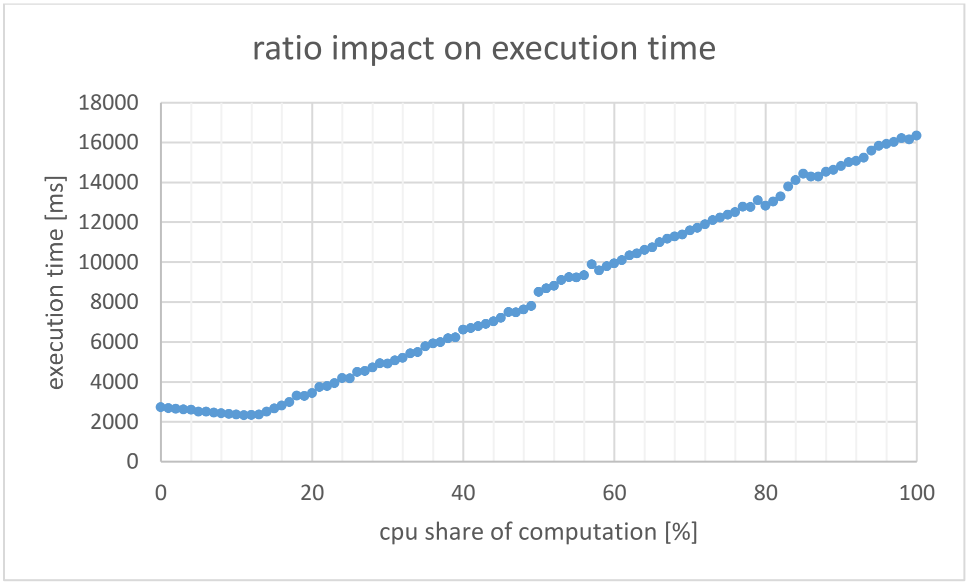
Measurements were performed for: Iterations: 1000; Threads: 7; Block dim x,y: 128, 1; Ratio: 0.12



Measurements were performed for: Size: 20000; Threads: 7; Block dim x,y: 128, 1; Ratio: 0.12

## Selecting optimal ratio

The optimal ratio for this testbed (Intel i7-4790k and Nvidia GeForce 970GTX) the optimal ratio seems to be 12% CPU, 88% GPU.



Measurements were performed for: Iterations: 100; Size: 20000; Threads: 7; Block dim x,y: 128, 1

## Code snippets

```
cuda.copyInitial(linearMesh_in);

#pragma omp parallel
for (int step = 0; step < STEPS; ++step) {
   #pragma omp master
   cuda.launchCompute(linearMesh_in);

   #pragma omp for
   for (int i = 0; i < DIVISION_POINT - 1; ++i) {
      #pragma ivdep
      for (int j = 0; j < MESH_SIZE; ++j) {
         //cpu computation
      }
   }

   #pragma omp master
   cuda.finalizeCompute(linearMesh_out);

   #pragma omp barrier

   #pragma omp single
   std::swap(linearMesh_in, linearMesh_out);

   #pragma omp barrier
}
cuda.copyFinal(linearMesh_in);
```

Hybrid version without nested parallelism

```
cuda.copyInitial(linearMesh_in);

for (int step = 0; step < STEPS; ++step) {
   #pragma omp parallel sections num_threads(7)
   {
      #pragma omp section
      {
         cuda.launchCompute(linearMesh_in);
         cuda.finalizeCompute(linearMesh_out);
      }

      #pragma omp section
      {
         #pragma omp parallel for num_threads(6)
         for (int i = 0; i < DIVISION_POINT - 1; ++i) {
            #pragma ivdep
            for (int j = 0; j < MESH_SIZE; ++j) {
               //cpu computation
            }
         }
      }
   }

   std::swap(linearMesh_in, linearMesh_out);
}

cuda.copyFinal(linearMesh_in);
```

Hybrid version with nested parallelism

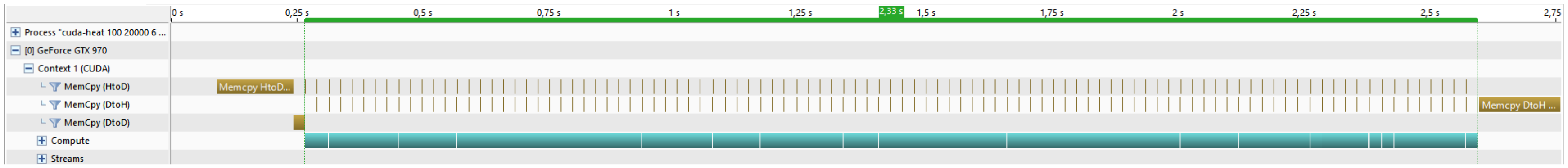
```
for (int i = 0; i < DIVISION_POINT - 1; ++i) {
   #pragma ivdep
   for (int j = 0; j < MESH_SIZE; ++j) {
      const int y = i + 1;
      const int x = j + 1;
      const float t_l = *getElem(linearMesh_in, pitch, y, x - 1);
      const float t_r = *getElem(linearMesh_in, pitch, y, x + 1);
      const float t_t = *getElem(linearMesh_in, pitch, y - 1, x);
      const float t_b = *getElem(linearMesh_in, pitch, y + 1, x);

      const float newTemperature = (t_l + t_r + t_b + t_t) / 4;
      *getElem(linearMesh_out, pitch, y, x) = newTemperature;
   }
}
```

CPU computation code

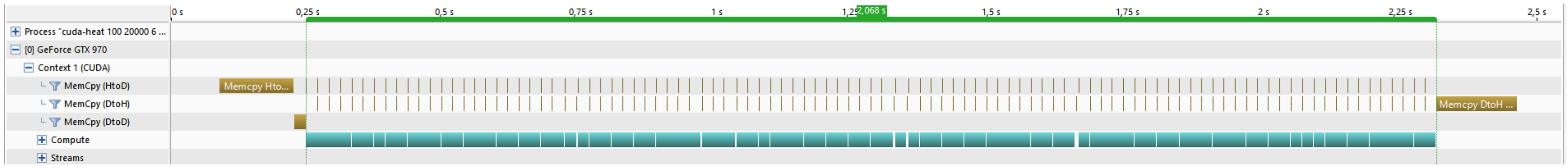
## Profiler output

Configuration used for generation of dumps visible in the profiler output in the next figure, was:

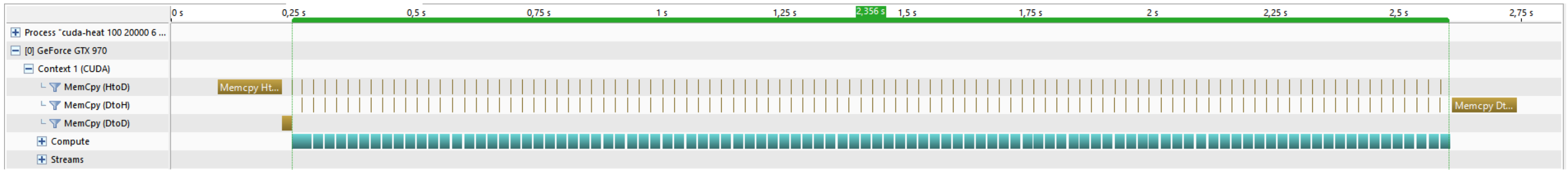
- Iterations: 100
- Size: 20000
- Threads: 6
- Block dim x,y: 128, 1
- Varying ratio, starting from the top: 0.001; 0.12; 0.15; 0.3; 0.12 (zoomed); 0.15 (zoomed)



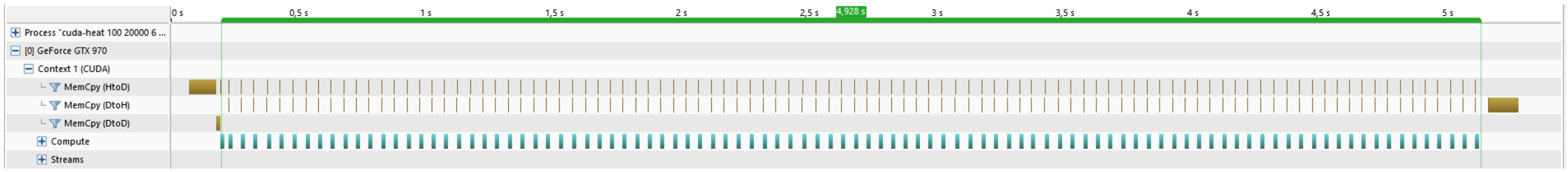
0 s
0,25 s
0,5 s
0,75 s
1 s
1,25 s
2,33 s
1,5 s
1,75 s
2 s
2,25 s
2,5 s
2,75
Process "cuda-heat 100 20000 6 ...
[0] GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
Streams
Memcpy HtoD...
Memcpy DtoH ...



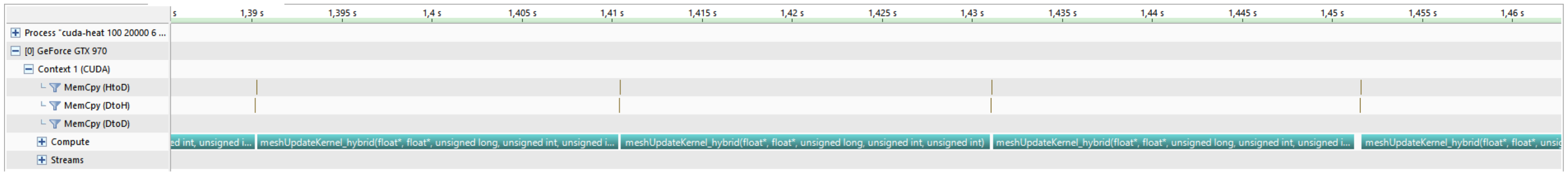
0 s
0,25 s
0,5 s
0,75 s
1 s
1,25
2,068 s
1,5 s
1,75 s
2 s
2,25 s
2,5 s
Process "cuda-heat 100 20000 6 ...
[0] GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
Streams
Memcpy Hto...
Memcpy DtoH ...



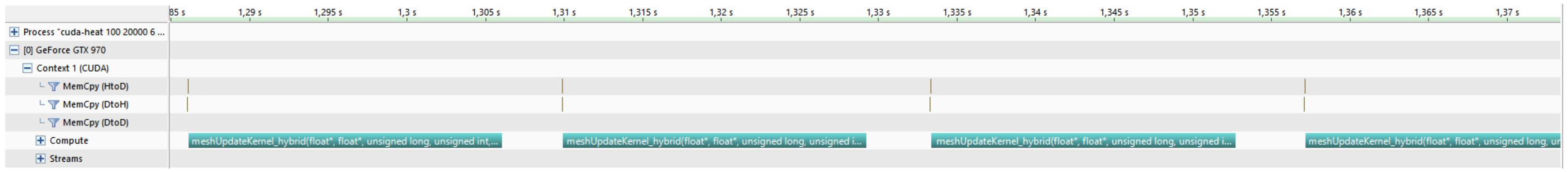
0 s
0,25 s
0,5 s
0,75 s
1 s
1,25 s
2,356 s
1,5 s
1,75 s
2 s
2,25 s
2,5 s
2,75 s
Process "cuda-heat 100 20000 6 ...
[0] GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
Streams
Memcpy Ht...
Memcpy Dt...



0 s
0,5 s
1 s
1,5 s
2 s
2,5 s
4,928 s
3 s
3,5 s
4 s
4,5 s
5 s
Process "cuda-heat 100 20000 6 ...
[0] GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
Streams



s
1,39 s
1,395 s
1,4 s
1,405 s
1,41 s
1,415 s
1,42 s
1,425 s
1,43 s
1,435 s
1,44 s
1,445 s
1,45 s
1,455 s
1,46 s
Process "cuda-heat 100 20000 6 ...
[0] GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
Streams
ed int, unsigned i...
meshUpdateKernel_hybrid(float*, float*, unsigned long, unsigned int, unsigned i...
meshUpdateKernel_hybrid(float*, float*, unsigned long, unsigned int, unsigned int)
meshUpdateKernel_hybrid(float*, float*, unsigned long, unsigned int, unsigned i...
meshUpdateKernel_hybrid(float*, float*, unsi



85 s
1,29 s
1,295 s
1,3 s
1,305 s
1,31 s
1,315 s
1,32 s
1,325 s
1,33 s
1,335 s
1,34 s
1,345 s
1,35 s
1,355 s
1,36 s
1,365 s
1,37 s
Process "cuda-heat 100 20000 6 ...
[0] GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
Streams
meshUpdateKernel_hybrid(float*, float*, unsigned long, unsigned int,...
meshUpdateKernel_hybrid(float*, float*, unsigned long, unsigned i...
meshUpdateKernel_hybrid(float*, float*, unsigned long, unsigned i...
meshUpdateKernel_hybrid(float*, float*, unsigned long, ur