Report CUDA

**Authors:**

*Riccardo Isola - S4943369*

*Gabriele Dellepere - S4944557*

*Kevin Cattaneo - S4944382*

# Index

[**Index 1**](#_heading=h.30j0zll)

[**Head of analysis 1**](#_heading=h.1fob9te)

[**Hotspot identification 1**](#_heading=h.3znysh7)

[First optimization (manual) 3](#_heading=h.2et92p0)

[**Vectorization issues 3**](#_heading=h.tyjcwt)

[**Best sequential time 4**](#_heading=h.3dy6vkm)

[**OpenMP Parallelization 5**](#_heading=)

[**Speedup and efficiency 7**](#_heading=)

[**Conclusions 9**](#_heading=)

# Head of analysis

**Algorithm:** the code provided has a complexity of rows \* columns \* iterations.

**Tools:**

* on Colab
  + *gcc* compiler
  + *nvcc* compiler for parallel versions
* Intel Advisor GUI to perform the roofline analysis
* icx to obtain the vectorization report

**Machines:** we run code on

* the Colab machine:
  + 4 processors:
    - 2 core with hyperthreading up to 2
  + one NVIDIA Tesla T4 GPU with 16 GB VRAM
* the lab machine:
  + 20 processors
    - 12 core
      * 8 core with hyperthreading up to 2
      * 4 performance core

# Hotspot identification

In first place we compiled the program:

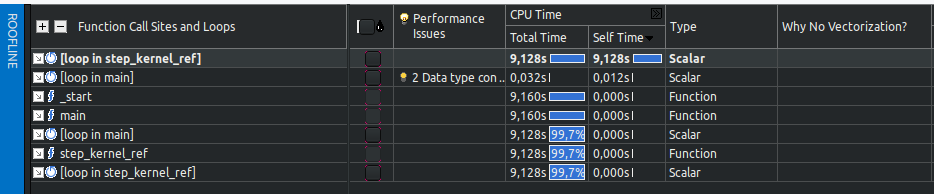
- **compiling line**: gcc heat.c

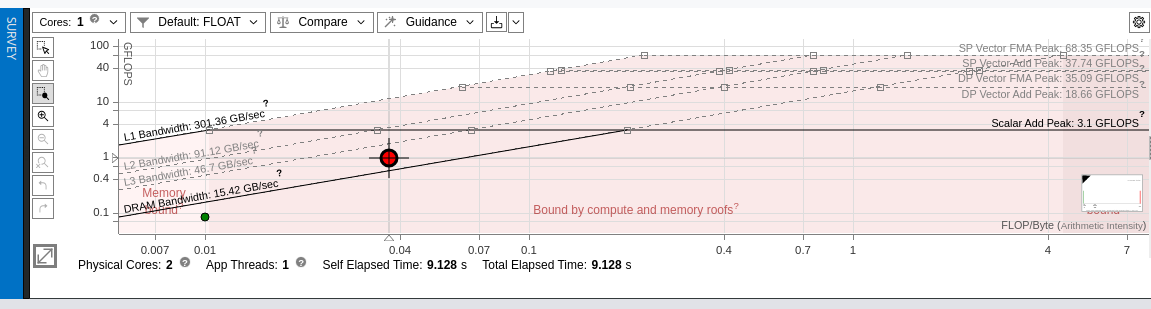
- **data size**: 1000x1000 matrix size, 1000 steps

- **time taken**: 9.20 seconds

Then we put the executable into intel advisor to perform a detailed analysis, where we identified the following hotspots. VA BENE NOSTRI PC?

| **Name** | **Time taken (seconds)** |
| --- | --- |
| Overall program | 9.20 |
| loop in step\_kernel\_ref | 9.13 |





As a hotspot we have the loop that involves the computation inside the step\_kernel\_ref, that is the one that is not optimized.

# Vectorization issues

To obtain the report about vectorization infos, we specified the level 3:

- **compiling line**: icx -fopenmp -O3 -xHost -qopt-report=3 heat.c

- **data size**: 6500x6500 matrix size, 1000 steps

- **time taken**: 2.46 seconds

We observed that:

* no major vectorization issues reported
* all the inner loops, apart the one where we print the result, are successfully vectorized
* no vectorization of outer loops (those should be the one that are parallelized) and of the line 102 because of step dependence and function call

# Best sequential time

Now we perform different runs to find the best sequential time, to be used as reference for the further parallel application, by passing several arguments and flags to the intel compiler.

- **compiling line**: icx -fopenmp -O3 -xHost -ffast-math heat.c

- **data size**: 1000x1000 matrix size, 1000 steps

- **time taken**: 0.797145 seconds

Overall we are applying the best optimizations arguments:

* **O3**: to allow the compiler to optimize the code, so that it can reorder the operations in the most efficient way
* **xHost**: to achieve the best assembly instructions specifically on the current machine architecture
* **fast-math**: to reduce the time needed for mathematical operations, altough reducing the precision of our calculations.

Then we performed several runs to reach at least 30 seconds of execution, and to do so we increased the data size:

- **compiling line**: gcc -O3 -ffast-math heat.c

- **data size**: 13000x13000, 200 steps

- **time taken**: 29.625438 seconds

# CUDA Parallelization

As first thing we explored some specifications of the machine on which we are running on via *devicequery:*

Device 0: "Tesla T4"

CUDA Driver Version / Runtime Version 12.2 / 12.2

CUDA Capability Major/Minor version number: 7.5

Total amount of global memory: 15102 MBytes (15835660288 bytes)

(040) Multiprocessors, (064) CUDA Cores/MP: 2560 CUDA Cores

GPU Max Clock rate: 1590 MHz (1.59 GHz)

Memory Clock rate: 5001 Mhz

Memory Bus Width: 256-bit

L2 Cache Size: 4194304 bytes

Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)

Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers

Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Total shared memory per multiprocessor: 65536 bytes

Total number of registers available per block: 65536

Warp size: 32

Maximum number of threads per multiprocessor: 1024

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

[...]

So the maximum number of threads that we can use per block is 1024 scheduled via a warp of 32.blocks

## First version

The main idea was to remove the for loop inside the function *step\_kernel\_mod* that exploits the blocks scanning them via indexes:

int x = threadIdx.x + blockIdx.x \* blockDim.x + 1;

int y = threadIdx.y + blockIdx.y \* blockDim.y + 1;

Also we exit when we reach indexes that doesn’t map to any cell since CUDA allocate gridSize \* blockSize threads which does not necessarily are equal to matrix size

if(x >= ni-1 || y >= nj-1) return;

And in the derivative evaluation we put directly as index the I2D transformation done splitted in the *step\_kernel\_ref.*

Then in the main:

float \*temp1\_dev, \*temp2\_dev;

cudaMalloc((void\*\*) &temp1\_dev, size);

cudaMalloc((void\*\*) &temp2\_dev, size);

printf("%.5f %.5f %.5f \\n" , temp1[0], temp1[4500], temp1[6700]);

cudaMemcpy(temp1\_dev, temp1, size, cudaMemcpyHostToDevice);

cudaMemcpy(temp2\_dev, temp2, size, cudaMemcpyHostToDevice);

dim3 block\_size(BLOCK\_WIDTH, BLOCK\_HEIGHT);

dim3 grid\_size = dim3((nj - 3) / BLOCK\_WIDTH + 1, (ni - 3) / BLOCK\_HEIGHT + 1);

printf("\\n%d %d \\n", (nj - 3) / BLOCK\_WIDTH + 1, (ni - 3) / BLOCK\_HEIGHT + 1);

// Execute the modified version using same data

for (istep=0; istep < nstep; istep++) {

step\_kernel\_mod<<<grid\_size, block\_size>>>(ni, nj, tfac, temp1\_dev, temp2\_dev);

// swap the temperature pointers

temp\_tmp = temp1\_dev;

temp1\_dev = temp2\_dev;

temp2\_dev = temp\_tmp;

}

cudaMemcpy(temp1, temp1\_dev, size, cudaMemcpyDeviceToHost);

cudaDeviceSynchronize();

[...]

cudaFree(temp1\_dev);

cudaFree(temp2\_dev);

So in summary we cudaMalloc some space for the variable used by the device, copied the temp1 and temp2 values into them; then we declared the threads number (block size) and the blocks number (grid size) to then call the *step\_kernel\_mod.*

The grid size computation presents some subtractions because the algorithm ignores the computation of values on the borders of the matrix, so we need to take the -3 position to allow the approximation on the nth index.

After exiting from the for loop, we transfer the data from the device into temp1, then we synchronize the kernel completion. Finally we free the allocated variables.

The original run:

- **compiling line**: gcc -O3 -ffast-math heat.c

- **data size**: 13000x13000 matrix size, 200 steps

- **time taken**: 29.625438 seconds

The run with the parallelization:

- **compiling line**: nvcc heat.c

- **data size**: 13000x13000 matrix size, 200 steps

- **time taken**: 2.157767 seconds

The GPU allowed us to save almost 15 times the time spent with only the CPU run with optimization.

## Second version - Shared memory

In this version we try to exploit the usage of shared memory:

\_\_shared\_\_ float sharedMem[(BLOCK\_WIDTH + 2) \* (BLOCK\_HEIGHT + 2)];

That we initialize:

sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y + 1)] = temp\_in[I2D(ni, x, y)];

And by doing so, the calls done previously on the temp1\_dev are now shifted to sharedMem:

if(threadIdx.x == 0) sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x, threadIdx.y + 1)] = temp\_in[I2D(ni, x - 1, y)];

else if(threadIdx.x == blockDim.x - 1 || x == ni - 2) sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 2, threadIdx.y + 1)] = temp\_in[I2D(ni, x + 1, y)];

if(threadIdx.y == 0) sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y)] = temp\_in[I2D(ni, x, y - 1)];

else if(threadIdx.y == blockDim.y - 1 || y == nj - 2) sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y + 2)] = temp\_in[I2D(ni, x, y + 1)];

\_\_syncthreads();

// evaluate derivatives

d2tdx2 = sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x, threadIdx.y + 1)]

- 2 \* sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y + 1)]

+ sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 2, threadIdx.y + 1)];

d2tdy2 = sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y)]

- 2 \* sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y + 1)]

+ sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y + 2)];

// update temperatures

temp\_out[I2D(ni, x, y)] = sharedMem[I2D(BLOCK\_WIDTH + 2, threadIdx.x + 1, threadIdx.y + 1)] + fact \* (d2tdx2 + d2tdy2);

To see better the difference compared to the first version we also increased the data size.

**First version**

- **compiling line**: nvcc heat.c

- **data size**: 10000x10000 matrix size, 5000 steps

- **time taken**: 21.750136 seconds

**Second version**

- **compiling line**: nvcc heat.c

- **data size**: 10000x10000 matrix size, 5000 steps

- **time taken**: 32.240333 seconds

We didn’t achieve the performance that we hoped for and still the first version that doesn’t exploit the shared memory performs faster.

# Third version - cudaMallocPitch

# As a third version we proposed a variation that substitutes the cudaMalloc with cudaMallocPitch and the cudaMemcpy with cudaMemcpy2D. That allows us to tell the GPU to exploit the coalescence, so the alignment in the device memory of the elements present in the 2D buffers with an inner padding.

So the *step\_kernel\_mod* becomes:

d2tdx2 = temp\_in[I2D(pitch1, x - 1, y)] - 2 \* temp\_in[I2D(pitch1, x, y)] + temp\_in[I2D(pitch1, x + 1, y)];

d2tdy2 = temp\_in[I2D(pitch1, x, y - 1)] - 2 \* temp\_in[I2D(pitch1, x, y)] + temp\_in[I2D(pitch1, x, y + 1)];

// update temperatures

temp\_out[I2D(pitch2, x, y)] = temp\_in[I2D(pitch1, x, y)] + fact \* (d2tdx2 + d2tdy2);

And in the main:

size\_t pitch1;

cudaMallocPitch((void\*\*) &temp1\_dev, &pitch1,nj,ni);

size\_t pitch2;

cudaMallocPitch((void\*\*) &temp2\_dev, &pitch2,nj,ni);

cudaMemcpy2D(temp1\_dev, pitch1,temp1, nj,nj,ni, cudaMemcpyHostToDevice);

cudaMemcpy2D(temp2\_dev, pitch2,temp2, nj,nj,ni, cudaMemcpyHostToDevice);

Then we compared the three versions:

**First version**

- **compiling line**: nvcc heat.c

- **data size**: 10000x10000 matrix size, 5000 steps

- **time taken**: 21.750136 seconds

**Second version**

- **compiling line**: nvcc heat.c

- **data size**: 10000x10000 matrix size, 5000 steps

- **time taken**: 32.240333 seconds

**Third version**

- **compiling line**: nvcc heat.c

- **data size**: 10000x10000 matrix size, 5000 steps

- **time taken**: 30.195786 seconds

We obtained a better result than the second version, but still the first one is the winner.

# Speedup

For this computation we used different combinations of block size and by doing so also of grid size, that is computed starting from the block width and height, observing the time spent.

Respectively <<<grid\_size, block\_size>>>

i=0 <<(2998,2998),(1,1)>>

took 62.646371 s

i=1 <<(1499,1499),(2,2)>>

took 24.665744 s

i=2 <<(750,750),(4,4)>>

took 7.775655 s

i=3 <<(375,375),(8,8)>>

took 3.337811 s

i=4 <<(188,188),(16,16)>>

took 3.193612 s

i=5 <<(94,94),(32,32)>>

took 2.973721 s

CREATE TABLE AND CHARTS

# Conclusions

We observed that, in the best sequential case, our code performs the run in about 30 seconds, on an amount of data that has to be triplicated from the original unoptimized sequential run (that performed in about 60 seconds for N=30000). So the compilation optimization flags increased by a lot the performance of our code given also by the quadratic complexity, so the result obtained with the new amount of data indeed shows a huge improvement.

By applying the *pragma* instructions to achieve better performance exploiting our processors via the OpenMP library, we observed an improvement that follows the increase of the number of threads. By just using 2 processors, we halved the time run, and by using 4 we halved again.

The optimal number of threads to be used, without losing too much in terms of efficiency is around 10: so 12 in our choices. We can get better performance by increasing that number to the maximum number of processors, that is 20, but the gain is of just a few seconds, not so relevant, also because the efficiency dramatically decreases.

Finally, by applying a number of processors greater than the one owned in the machine, we don’t observe any major overhead that significantly delays the time run of our code.