Report CUDA

Authors:
Riccardo Isola - S4943369
Gabriele Dellepere - S4944557
Kevin Cattaneo - S4944382

Index

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

Hotspot identification

In first place we compiled the program:
- **compiling line:** `icx -g heat.c`
- **data size:** `1000x1000 matrix size, 2000 steps`
- **time taken:** `9.72 seconds`

Then we put the executable into intel advisor to perform a detailed analysis, where we identified the following hotspots.

Name	**Time taken (seconds)**
Overall program	9.72
loop in step_kernel_ref	9.70

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 -O3 -xHost -qopt-report=3 heat.c`
- **data size:** `1000x1000 matrix size, 2000 steps`
- **time taken:** `0.457764 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 -O3 -xHost -ffast-math heat.c`
- **data size:** `1000x1000 matrix size, 2000 steps`
- **time taken:** `0.453736 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:** `icx -O3 -ffast-math heat.c`
- **data size:** `13000x13000 matrix size, 400 steps`
- **time taken:** `29.592816 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 in warps of 32 threads. Also we have available 40 Streaming Multiprocessors (each with 2 warp schedulers), for a total available maximum of 2560 concurrent threads scheduled at each clock cycle.

First version

The main idea was to unroll the for loop inside the function *step_kernel_mod* that exploits the blocks scanning the heat matrix via indexes:

```
int x = threadIdx.x + blockIdx.x * blockDim.x + 1;
int y = threadIdx.y + blockIdx.y * blockDim.y + 1;
```

(the +1 was put there as our algorithm does not update the edges of the matrix)
Also we exit whenever 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'ed some space for the buffers 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*. We kept the block_size as user-defined and computed the needed grid-size so that there were enough threads to span the whole matrix.
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 size - 2 and then the ceiling of its division by the block size.
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, in the remote Colab machine:
- **compiling line:** `gcc -O3 -ffast-math heat.c`
- **data size:** 13000x13000 matrix size, 400 steps
- **time taken:** 55.795435 seconds

The run with the parallelization:
- **compiling line:** `nvcc heat.c`
- **data size:** 13000x13000 matrix size, 400 steps
- **time taken:** 3.514749 seconds

In the lab machine with OpenMP we obtained:
- **compiling line:** icx -O3 -ffast-math heat.c
- **data size:** 13000x13000 matrix size, 400 steps
- **time taken:** 29.592816 seconds

The GPU allowed us to save almost 15 times the time spent with only the CPU run with optimization and 10 times the parallelization with OpenMP.

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)];
```

The idea was the following: as each thread is responsible for one single cell, it means each thread needs to access 5 times into the global memory to retrieve the needed information for the update. 4 out of these 5 reads, however, are redundant with the ones performed by the threads responsible for the immediately adjacent cells, meaning that by allocating a 2D buffer in shared memory we could spare some time during data retrieval.
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. This is likely due to the overhead of re-creating the 2D buffer in shared memory after each kernel launch (shared memory is reset after each launch, and there is no way to force all the threads in all SMs to synchronize to keep everything into one long single kernel launch) plus the fact that the shared memory "steals" space from the L1 cache, that the first version of the program probably exploits.
There was the possibility to furtherly optimize the creation of the shared memory buffer, by forcing the threads responsible for the creation of the buffer "halo" inside the same warp, but we abandoned this idea because:

- mapping a 1D id to a 2D halo without using if statements is hard (we thought about using max/min operations but those are still implemented with an if condition)
- the performance improvement would have been too small to compensate for those 10 seconds difference between the first program and the second one

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

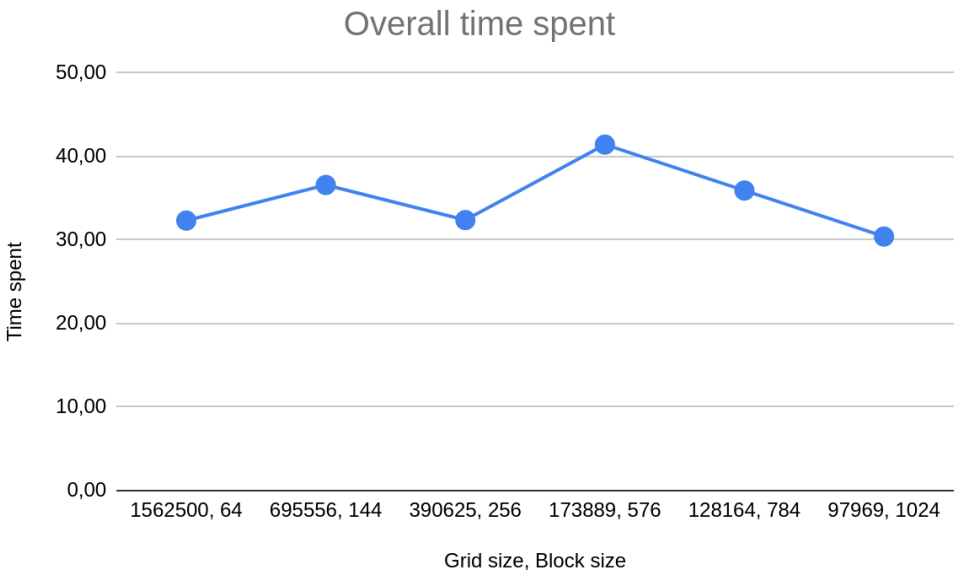
Third version
- **compiling line:** nvcc heat.c
- **data size:** 10000x10000 matrix size, 5000 steps
- **time taken:** 30.195786 seconds

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

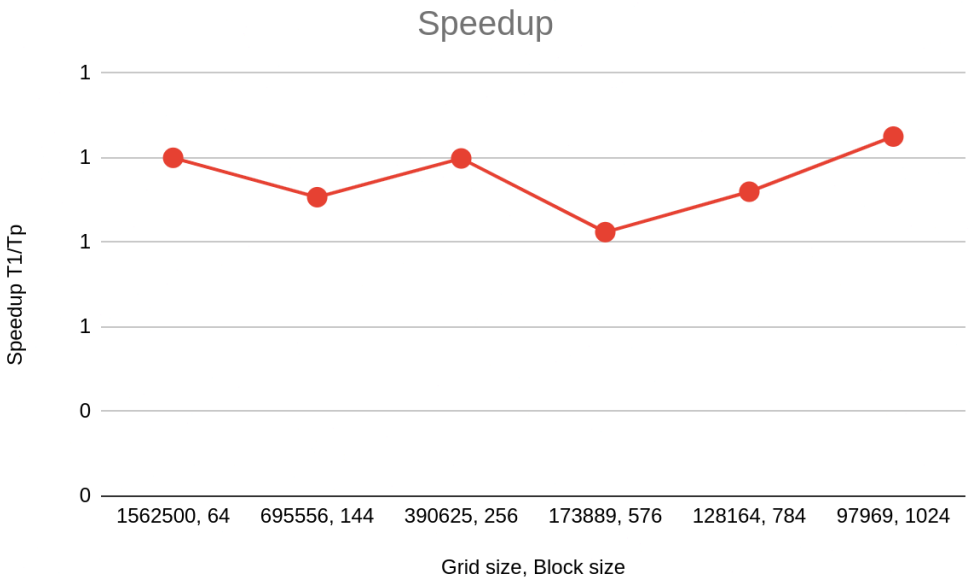
Speedup

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. We start from a block_size that is above the warp size of 32 threads.
Respectively in the following <<<grid_size, block_size>>>

Data size	Steps = 5000	Matrix = 10000
Grid size, Block size	Time spent	Speedup T1/Tp
1562500, 64	32,33	1
695556, 144	36,59	1
390625, 256	32,39	1
173889, 576	41,43	1
128164, 784	35,93	1
97969, 1024	30,42	1



The amount of threads used in the computation is the same because of the computation of gridSize and blockSize that are inverse proportional dependent. So the amount of time spent is almost the same between several combinations.



The same applies for the speedup, so we don't achieve great performances.

Conclusions

What we observed is that the accelerator allowed us to save almost 15 times the time spent with only the CPU run with optimization and 10 times with respect to the parallelization with OpenMP.
Unfortunately also by exploiting the shared memory and the coalescence we didn't achieve better results in terms of time spent.
Also the speedup values are not significant because of the same time spent with different combinations of <<<gridSize, blockSize>>>.