Code Report

Table of content

Introduction

⚠ Organization of the project

The code was designed to be fairly modular and tidy, with readability in mind.
To achieve this, it was written as a static library.
The `main.cpp` file calls both the parallel and sequential version of the algorithms, whose implementations are stored in the respective files (`Par` for parallel and `Seq` for sequential) inside the `src` folder.
The main program runs both versions of the algorithm and compares the results obtained.
`main.cpp` can be compiled invoking the makefile with `make`, which produces an executable named `out`. On UNIX-like systems, prompting `make && ./out` should be enough to compile and run the program

The Smith-Waterman algorithm

✎ What's the Smith-Waterman algorithm?

The Smith-Waterman algorithms is used to determine the grade of similarity of two nucleic-acidic sequences.
More specifically, it finds the regions of the two strings that have the best match, within some set parameters.

Sequential implementation

Two strings (query Q and reference R) of set length `S_LEN` and representing the sequences to analyze are used to set up a matrix M. Each row of the matrix correpsonds to a character of the query, while columns are mapped to single elements of the reference.
The first row and column serve as zero-padding and do not correspond to any value of R or Q: their alignment starts from index 1.
This way, each cell of the matrix M_{ij} corresponds to a couple $(Q[i+1], R[j+1])$, and $M \in \text{Mat}_{S\_LEN+1 \times S\_LEN+1}(\mathbb{N})$.

The value of every cell of the matrix is computed from top-left to bottom-right.
For each $i, j \geq 1 \wedge i, j \geq S\_LEN + 1$ the algorithm:

- Checks wether $R[i] == Q[j]$. If yes, they MATCH.
- Checks if it would be more convenient to:
  1. Insert a diagonal movement (match so that the sequence will concatenate M_{ij} and $M_{i-1,j-1}$)
  2. Insert a space (and check wether $M_{i,j-1}$ would result in a better match)
  3. Delete the query character (by matching M_{ij} with M_{i-1}, j).
- To each combination is assigned a score, given by the sum of the value of the neighbor in said direction (`upLeftNeighbor`, `leftNeighbor` and `upNeighbor` respectively) and a penalty set as a paramether. In this version, a match increments the value by 1, a deletion or an insertion decrease it by 2, a mismatch decreases it by 1. The scores cannot grow below zero.
- The computed score (the maximum value among the three and 0) is stored in `score_matrix`, the direction chosen (the one corresponding to the highest score) is stored in `direction_matrix` at the same index (ij).
- Starting with the cell with the highest score in the matrix, the subsequence with the best match is retrieved following the directions (up, left, upleft) left in the `direction_matrix`.

A first parallel implementation

To implement a parallel version of this algorithm, multiple approaches can be combined.

> **Key idea**
>
> Every cell needs to be computed *after* its neighbors (as defined above) are computed → values can be computed diagonally

First layer of parallelism

> **Grid-level parallelism**
>
> Multiple couples of sequencies can be computed at the same time

Each couple of sequencies is indipendent from the others, therefore they can all be computed at the same time. Each couple is assigned to a block, with it's own score and direction matrices.
The efficiency of this approach is bound to the capabilities of the GPU: a device with more Streaming Multiprocessors can compute more blocks at the same time

Second layer of parallelism

> **Block-level parallelism**
>
> All diagonally-adjacent elements can be computed at the same time

Since the value of an element depends on the values of its up, left and upLeft neighbors, elements can be computed from the top left corner to the bottom right one. The first computed value is M_{11}, then $M_{2,1}$ and $M_{1,2}$ can be computed at the same time, then $M_{3,1}, M_{2,2}, M_{1,3}$ and so on.

This way, up to `S_LEN` elements of the matrix can be computed at the same time.

This level of parallelism is achieved by iterating over the matrix `2*S_LEN` times (as opposed to the $\mathrm{S\_LEN}^2$ iterations of the sequential algorithm), and by mapping each element of the diagonal to a thread. The first iteration will spawn one active thread, the second two and so on.

About the data

Queries and references are stored in two $N \cdot \mathrm{S\_LEN}$ matrices on the host. When copied to the device, these matrices are flattened into arrays. The first $\mathrm{S\_LEN}$ elements of said array correspond to the first sequence, and will be used by the block of index 0.

Score and direction matrices are also allocated in global memory by the host and set to zero with `memset()`.
They are allocated as 3D tensors of contiguous dynamic memory, and are passed as an array of size `N*(S_LEN+1)*(S_LEN+1)*sizeof(element)` on device. Each block will only access a `(S_LEN+1)*(S_LEN+1)` area of the array.

Pseudocode of the Kernel

Iterate 2*S_LEN times:

- Increase the number of active threads by one if iteration < S_LEN, decrease it by one otherwise
- If the considered thread is active:
    - Map it to a cell of the matrices accordingly to a function of it's thread ID and the current iteration
    - Map the cell to the corresponding chars in the sequencies
    - Compare the chars
    - Determine the score for the cell with the formula:

$$\mathit{score} = \max\left(\mathrm{upLeftNeighbor} + \mathit{comparison},\ \mathrm{leftNeighbor} + \mathit{ins}, \mathrm{upNeighbor} + \mathit{del}, 0\right)$$

    - Determine the direction accordingly
    - Save the values in the respective matrices. at the same index
- Send the matrices back to host

The host will then have to find the maximum value in `score_matrix`, and compute the sequence from there by following the directions inside `directions_matrix`.

Execution time

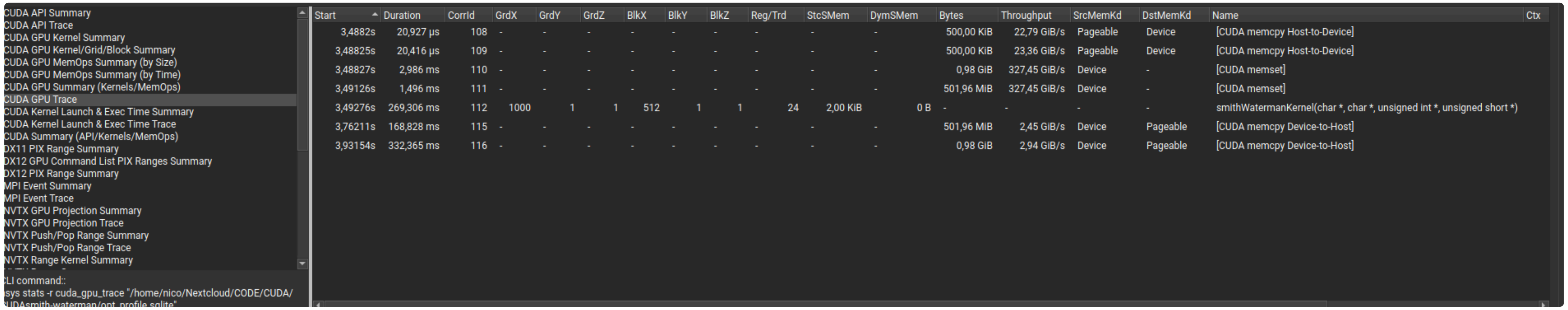
While the sequential algorithm takes around 3.27 seconds to compute on a ryzen 5 7600 processor, the parallel version runs in around 1.27 seconds on an NVIDIA RTX 3060 GPU.
This might suggest that using a GPU could reduce the computations required for this algorithm by 50-60%.

The NVIDIA nsight profiler reveals some more detailed information about the execution time of the algorithm. Specifically, it measures that the execution of the kernel lasts on average 273626097.6 ns, that is, 273.6260976 ms (average computed on 10 executions).

It's also worth noticing that the program spends 275,093699 ms busy with synchronization calls, and 533,723074 ms executing `cudaMemcpy` statements.

Profiling software

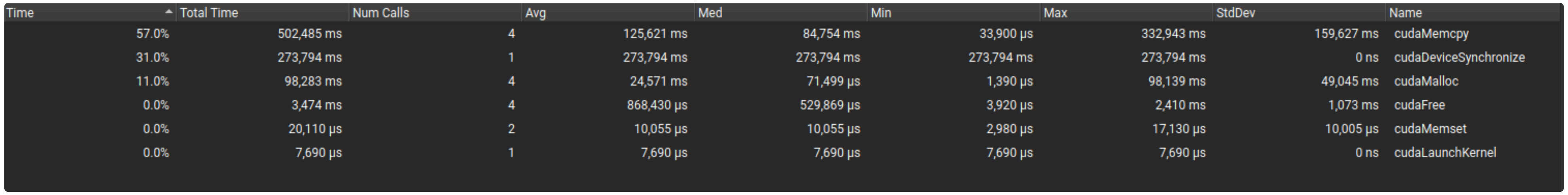
NVIDIA nsyght system tools for profiling highlighted the bottlenecks of the kernel:


CUDA API Summary
CUDA API Trace
CUDA GPU Kernel Summary
CUDA GPU Kernel/Grid/Block Summary
CUDA GPU MemOps Summary (by Size)
CUDA GPU MemOps Summary (by Time)
CUDA GPU Summary (Kernels/MemOps)
CUDA GPU Trace
CUDA Kernel Launch & Exec Time Summary
CUDA Kernel Launch & Exec Time Trace
CUDA Summary (API/Kernels/MemOps)
DX11 PIX Range Summary
DX12 GPU Command List PIX Ranges Summary
DX12 PIX Range Summary
MPI Event Summary
MPI Event Trace
NVTX GPU Projection Summary
NVTX GPU Projection Trace
NVTX Push/Pop Range Summary
NVTX Push/Pop Range Trace
NVTX Range Kernel Summary

```
CLI command::
sys stats -r cuda_gpu_trace "/home/nico/Nextcloud/CODE/CUDA/
UDAsmith-waterman/opt_profile.sqlite"
```

Start	Duration	CorrId	GrdX	GrdY	GrdZ	BlkX	BlkY	BlkZ	Reg/Trd	StcSMem	DymSMem	Bytes	Throughput	SrcMemKd	DstMemKd	Name	Ctx
3,4882s	20,927 µs	108	-	-	-	-	-	-	-	-	-	500,00 KiB	22,79 GiB/s	Pageable	Device	[CUDA memcpy Host-to-Device]	
3,48825s	20,416 µs	109	-	-	-	-	-	-	-	-	-	500,00 KiB	23,36 GiB/s	Pageable	Device	[CUDA memcpy Host-to-Device]	
3,48827s	2,986 ms	110	-	-	-	-	-	-	-	-	-	0,98 GiB	327,45 GiB/s	Device	-	[CUDA memset]	
3,49126s	1,496 ms	111	-	-	-	-	-	-	-	-	-	501,96 MiB	327,45 GiB/s	Device	-	[CUDA memset]	
3,49276s	269,306 ms	112	1000	1	1	512	1	1	24	2,00 KiB	0 B	-	-	-	-	smithWatermanKernel(char *, char *, unsigned int *, unsigned short *)	
3,76211s	168,828 ms	115	-	-	-	-	-	-	-	-	-	501,96 MiB	2,45 GiB/s	Device	Pageable	[CUDA memcpy Device-to-Host]	
3,93154s	332,365 ms	116	-	-	-	-	-	-	-	-	-	0,98 GiB	2,94 GiB/s	Device	Pageable	[CUDA memcpy Device-to-Host]	

The lowest thrughput in memory operations is obtained while transferring back the matrices from device to host. It's also clear that the time take to retreive data from the device is almost double the time it takes the device to compute all the sequencies.



Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
57.0%	502,485 ms	4	125,621 ms	84,754 ms	33,900 µs	332,943 ms	159,627 ms	cudaMemcpy
31.0%	273,794 ms	1	273,794 ms	273,794 ms	273,794 ms	273,794 ms	0 ns	cudaDeviceSynchronize
11.0%	98,283 ms	4	24,571 ms	71,499 µs	1,390 µs	98,139 ms	49,045 ms	cudaMalloc
0.0%	3,474 ms	4	868,430 µs	529,869 µs	3,920 µs	2,410 ms	1,073 ms	cudaFree
0.0%	20,110 µs	2	10,055 µs	10,055 µs	2,980 µs	17,130 µs	10,005 µs	cudaMemset
0.0%	7,690 µs	1	7,690 µs	7,690 µs	7,690 µs	7,690 µs	0 ns	cudaLaunchKernel

This table shows another huge potential bottleneck for this kernel: a third of the total execution time (from the first `cudaMemcpy` to the last `cudaFree`) is spent synchronizing with the device.
I'm not sure how this time is measured, since it's almost equal to the execution time of the kernel, but could be an indicator of thread divergence within the kernel, and it's worth exploring the matter further.

Optimizations

The primary bottleneck of this implementation of the algorithm is aused by memory operations.
Both score and direction matrices are frequently accessed, but are too large to fit inside the shared memory of a single block: they need to reside in global memory, introducing a sensible latency and reduction of thrughput.

Key optimization idea

The function used to map a thread to a cell of the matrices has a predictable nehavior:
A thread of `id = x` will compute an L shaped group of cells, moving down one column until the diagonal of the matrix, and then moving along a row
Furthermore, a thread needs only values from the two previous iterations over the matrix to compute its cell.
This means that:

1. It's possible to cache in shared memory all the scores needed to compute the value of a cell
2. If the maximum score in each `score_matrix` could be computed on device, there would be no need to even allocate a `score_matrix`

Reducing data transfer

Find maximum value on device

Since the different data transfers from host to device and back add up to almost 50% of execution time, a first way to cut it down would be decreasing the amount of data transferred.
In the first implementation, the maximum score is computed on the host. This task can be easily done on device as well, removing the need to transfer more than 1GB of data back to the host.

Each thread can keep track of the highest score it computes and it's position, and the highest score overall can be determined at the end just among those 512.
This way, there's no need to copy back `score_matrix` from device to host, since it's only purpose was to find the maximum score for each query-reference couple.

Use shared memory to keep track of the scores

Analyzing access patterns

To compute the value of its cell, a thread only needs the score of neighboring threads. In this implementation, threads "scan" the matrix diagonally: this means a computed score will be a neighbor to some thread for the next two iterations, and will be never used again after.

Thus, it's convenient to just keep track of the scores of the past two iteration: this amount of data can easily fit inside shared memory.

This final optimization greatly cuts down execution times to an average of 0.6 seconds (on an RTX 3060).
This optimization introduces a new synchronization barrier in the algorithm, but greatly reduces access to global memory: the overall execution time of the kernel is reduced by approximately a 4x factor.

```
Time (%)  Total Time (ns)  Instances   Avg (ns)    Med (ns)   Min (ns)  Max (ns)  StdDev (ns)                              Name
--------  ---------------  ---------  ----------  ----------  --------  --------  -----------  ---------------------------------------------------------------
   100.0         66410826          1  66410826.0  66410826.0  66410826  66410826          0.0  smithWatermanKernel(char *, char *, unsigned short *, int *, int *)
```

As shown in this profiling tab, it's evident that the optimization effectively addressed previous bottlenecks of the algorithm, presented by memory transfers and synchronization barriers. This performance gain comes at virtually no cost: the 10% percent increase in allocation time comes from using two more `cudamalloc()` statements in the optimized code, with little to no impact on real execution time: the percentage only increases because all other statistics decrease.

```
Time (%)  Total Time (ns)  Num Calls   Avg (ns)    Med (ns)    Min (ns)   Max (ns)   StdDev (ns)          Name
--------  ---------------  ---------  ----------  ----------  --------  ---------  -----------  ---------------------
    48.4        163064046          5  32612809.2     34999.0     14110  162941407   72855903.0  cudaMemcpy
    31.0        104346016          5  20869203.2      1500.0      1430  104268437   46621599.3  cudaMalloc
    20.2         68018435          1  68018435.0  68018435.0  68018435   68018435          0.0  cudaDeviceSynchronize
     0.4          1339145          5    267829.0      3590.0      2000    1056328     456297.2  cudaFree
     0.0            15880          1     15880.0     15880.0     15880      15880          0.0  cudaMemset
     0.0             9039          1      9039.0      9039.0      9039       9039          0.0  cudaLaunchKernel
```

Detailed explanation of the code

`main.cpp`

The main program allocates all the matrices needed to store the subsequences returned by the algorithms.

⚠ **Dynamic allocation**

The matrices are dynamically allocated by the custom function `allocateMatrix()` as blocks of contiguous memory on the heap. The classical approach shown in the sequential algorithm was not suitable for the parallel version, because it didn't allocate contiguous memory

It then randomly initializes the matrix containign the sequencies to compute, and then calls both versions of the algorithm.
When they are finished, it prints their execution times, and then compares the returned sequencies to check if they are equal.

`smithWatermanSeq`

This function runs the sequential algorithm contained in the file `smithWatermanSeq.cpp`. The algorithm is just an adaptation of the provided one.

`smithWatermanPar`

This function, defined in the file `smithWaterman.cu`, is just wrapper arund the CUDA kernel, defined in the same file. It's designed to provide an interface identical to the sequential one, making the difference transparent to the user.

This function allocates all the memory regions needed by the algorithm. More specifically:

- It allocates a memory region for query and reference matrices, then fills them with the values passed as an input
- It allocates a memory region for the direction matrices of *all* blocks and initializes it to 0
- It allocates two vectors to store the coordinates of the cell with the highest score for each query/reference couple → each block

It then launches the kernel with `N=1000` blocks (one for each couple) and `512` threads per block. This is the length of the diagonal of the matrices, and also the maximum number of possible active threads at once within the block.

> ✎ **What's computed by one block?**
>
> Each block is tasked with computing the smith-waterman algorithm for just one query/reference couple.

Finally, it copies back the score tensor and the coordinates arrays, then backtraces the sequence found using the method provided (function `backtrace()`).

`smithWatermanKernel`

This is the portion of code executed on device.
It's arguments are:

- `d_query`: a pointer to a region of memory holding `N=1000` different sequences of `512` chars
- `d_reference`: same as `d_query`
- `d_direction_tensor`: a pointer to a region of memory allocated to store `N` 513×513 matrices holding the directions computed by the algorithm
- `d_max_row`: holds track of 1000 y-coordinates of the max scores for each couple of sequencies
- `d_max_col`: stores 1000 x-coordinates of the max scores for each block

The kernel then allocates 5 arrays in shared memory for each block. These arrays are all of length `S_LEN`, but serve two different purposes:

- maxValues, maxValuesx and maxValuesy keep track of the max score computed by a single thread and it's coordinates. *Each thread will only modify the cell of the array with the same index as it's thread ID*. All the values are stored in arrays, so that a group of thread can retrieve the max across the whole matrix and its coordinates at the end of the algorithm

- scoreNeighbors and prevScoreNeighbors are used to cache the scores computed by a thread. During the `i`-th iteration (i'll properly define what an iteration is in the next few paragraphs) a thread of ID `tid=x` needs the scores of it's three neighbors to compute it's values. This neighbors always happen to be, given the function defined to map threads to index of the direction matrix:
    - Either the score computed by the thread of `tid=x-1`, `tid=x` or `tid=x+1` of the `i-1`-th iteration (left and up neighbors)
    - The score computed by the thread of `tid=x-1`, `tid=x` or `tid=x+1`of the `i-2`-th iteration (upLeft neighbor)

The next section is the core of the algorithm: it's a for cycle, repeated `S_LEN*2` times. This is the number of iterations needed to compute all values, if in each iteration all elements on a "diagonal" are computed at the same time.
For each iteration, a different number of thread is considered active: the loop starts with 1 active thread and increases the number by 1 for each iteration up until iteration number `512`. Then, it starts decreasing the counter back to 1.

The thread active during the first iteration has `tid=0` and computes the element in position `1,1` . On the next iteration, positions `2,1` and `1,2` will be computed, then `3,1`, `2,2`, `1,3` and so on, in a diagonal waveform pattern from the top left to the bottom right.
Each thread is mapped to an element of the flattened `direction_matrix` by a function, based on it's `tid` and the current iteration. This function assures the access pattern of the array and has made possible the optimization regarding `score_matrix`.

Once inside the loop, each active thread compares the corresponding values in the sequences and computes the score as it's done in the sequential implementation.
Then, it updates `direction_matrix` and checks wether this score is elegible to be cached in the array for maximum values.

After all threads have reached this point, they update the neighbors arrays. This step is put behind a synchronization barrier, because otherwise some threads could update their value inside the neighbor arrays before their neighbors ad actually used the value meant to be read during that iteration.

When the loop ends, all threads are synchronized.

The last steps consist of finding the maximum value nearest to the top left corner and saving it's final position in the given "return arrays". The maximum value is computed by making threads compute a sequence of binary checks.

Final considerations

There is sure room for some other optimizations, such as computing the backtracing of the final pattern on device, but i'm quite sure it wouln't benefit performance very much, as it would need quite some more highly sequential computations to be done on device, only to further reduce the size of the `cudaMemCopy()` operations.

A more aggressive compilation (-o3, not enabled in the final commit) cut sequential execution time to just a fraction of second more than parallel execution time: still, i consider my final algorithm to be optimized enough to beat even code optimized by a compiler.

This project has been somewhat challenging, since it required finding a quite different approach (with regards to the data structures implemented) from the one provided, in order to deliver decent performance. Nonetheless, it has been a fun and educational experience