A CUDA implementation of Blocked All Pairs Shortest Path

Zuccato Francesco 143095
Lena Emanuele 142411

September 10, 2022

Abstract

For the exam of Parallel Architectures is required to implement an algorithm in parallel using CUDA (the option opted) or OpenCL.

In specific, the algorithm chosen in this is a variant of Floyd Warshall APSP designed by Gayathri, Sartaj, Srabani. The basic idea is, given a graph of size n, to split its adjancency matrix in many squared sub-matrixes (*blocks*) of a constant length (*blocking factor* or B).

The algorithm has three main steps, and in all of them there is a sort of Floyd Warshall. The first one operates inside the *self-dependent* block (on the diagonal), the second one inside the blocks on the same row or column of the *self-dependent*, the third on all the others blocks. For more details check the original paper.

Since many parts can be performed at the same time, this algorithm is perfect to be implemented in parallel.

Contents

Chapter 1

The algorithm

The object of this project is to propose a parallel implementation of the Floyd Warshall APSP variant proposed by Gayathri, Sartaj, Srabani [1]. In this chapter, will describe briefly the general structure of the algorithm and the parallelization opportunities.

1.1 Classical Floyd Warshall

Floyd Warshall All Pair Shortest Path [2] is one of the classical graph algorithm for the graphs. It calculates the shortest path between all pairs of nodes.

Floyd Warshall's takes in input the adjacency matrix M of a graph $G = (V, E, W)$ with $|V| = n$ nodes and the weight of the arc (i, j), $W[i, j]$, written in the cell $M[i, j]$. The algorithm checks for each pair of nodes (i, j) the shortest path by taking the $min(M[i, j], M[i, k] + M[k, j])$ iterating on $k = 0$ *to* n. The idea is to find the better path $i \to j$ by passing through the node k. At the end of the algorithm, the matrix M will contain the cheapest cost foreach pair of nodes. The code contains three nested for loops on all the matrix and so the complexity is trivially $O(n^3)$.

1.2 Blocked Floyd Warshall

Blocked Floyd Warshall relies on the idea to partitionate the $n * n$ cost matrix in a set of sub-matrices called *blocks*. Those blocks have constant size $B * B$; B is called also *blocking factor* and the choice of its value may influence the obtained performance (that topic will be discussed further in section 1.3).

1.2.1 Blocks, Rounds and Phases structure

Algorithm is organized in n/B rounds; on each round $T \in \{0..\frac{n}{B} - 1\}$ you call the block (T, T) the round's *self-dependent block*. The block is made by all the cells

$$M[i,j] \text{ s.t. } K*B \leq i < (K+1)*B \ , \ K*B \leq j < (K+1)*B$$

We can say that each round has the purpose of applying Floyd Warshall's inspired path comparison between each path $M[i,j]$ and the alternative paths that use T block nodes $\{(T*B)..(T+1)*B-1\}$.

Comparison is organized in three phases.

1. In *phase 1* you apply the classical Floyd Warshall algorithm to self-dependent block's cells.
2. In *phase 2* you apply the path comparison on all the cells located in the self-depended block's same row or in the same column. In practice, for each cell

   $$M[i,j] \text{ s.t. } T*B \leq i < (T+1)*B \text{ XOR } T*B \leq j < (T+1)*B$$

   you compare the current best path with alternative paths which use $\{(T*B)..(T+1)*B-1\}$ nodes.

   Your phase 2 code will look like a Floyd-Warshall algorithm (executed on each block which shares a row or a column with (T,T)) where:

   - external cycle goes from $T*B$ to $(T+1)*B-1$;
   - the two internal cycles iterate all block's cells.
3. *Phase 3* is similar to phase 2, but you operate on all remaining blocks (so the ones who don't share either a row or a column with (T,T)).

In figure 1.1 we show which blocks are involved in each phase. For serial pseudo-code and further explanations you may check the original Gayathri, Sartaj and Srabani's original paper [1].

1.3 Advantages of Floyd Warshall Blocked

At first, the Blocked Floyd Warshall may seem just a fancy and more complex variant of the classical algorithm; in fact, the comparisons made for each cell $M[i,j]$ of the cost matrix are exactly the original ones, just organized in a different way. The advantage of the solution is based on modern calculators' memory architecture.

Blocked Floyd Warshall's core idea relies on the fact that modern calculators' main memory is organized in several cache tiers characterized by different access times. The idea is that breaking the algorithm into blocks will allow you to use more efficiently L1 and L2 cache, minimizing cache misses and so also the average memory's access time.

This concept is well deepen in Gayathri, Sartaj and Srabani's original paper [1].

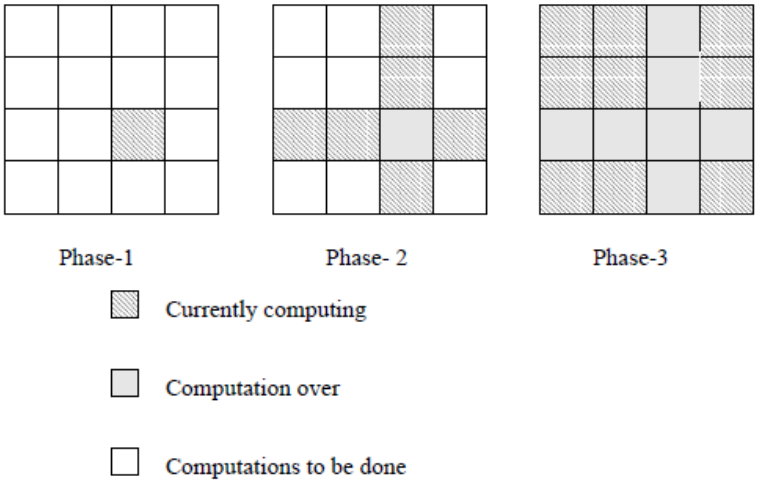

Figure 1.1: Blocks computed on each phase. You can clearly notice how phase 1 involves just the self-dependent block, phase 2 computes the rows and the columns and phase 3 computes the remaining blocks. You may also notice that the self-dependent blocks of rounds are the ones which lay in the matrix's diagonal. This figure is taken from Gayathri, Sartaj and Srabani's original paper [1].

Chapter 2

Parallelization Opportunities

To analyze parallelization opportunities in Blocked Floyd Warshall we have to start on possible parallelizations we could have in the original algorithm. From there, we can start exploring further parallelizations and optimal uses of typical graphic card features.

2.1 Parallel Floyd Warshall basic idea

Floyd Warshall serial algorithm is structured in *three nested for cycles* on all nodes of the graph. The two internal cycles $i \in \{0..n-1\}$ and $j \in \{0..n-1\}$ are needed to consider all $i \to j$ paths of the graph, while the external cycle $k \in \{0..n-1\}$ can be seen as an interaction on all "possible better alternative". The exploration of a better alternative $i \to k \to j$ for a path $i \to j$ can guarantee a sub-optimal (and so an optimal at the end of the algorithm) if and only if:

- I already explored all previous $i \to k' \to j$ paths for $k' \in \{0..k-1\}$;
- I have done the job even for all others (i', j') pairs and not just for the current one.

From this intuition, we can imagine the structure of a *parallel Floyd Warshall* were:

- to explore all possible alternative routes we keep the external cycle $k \in \{0..n-1\}$;
- the two internal cycles are computed in parallel, but guaranteeing that before we check $i \to k \to j$, we have already explored all previous $i' \to k' \to j'$ paths with $i', j' \in \{0..n-1\}$ and $k' \in \{0..k-1\}$.

Starting from this idea, we can design the first level of parallelization in *Blocked Floyd Warshall*, where the execution of the round code on each block parallelizes the two internal cycles. In algorithm 1 we show the structure of such a code. This basic idea is implemented in code explained in our first basic version of code (explained in subsection 3.1.2). This version replicates pretty much what is explained by the paper's pseudo-code.

Algorithm 1 Blocked Floyd Warshall where you parallelize In-Block operations in a round

procedure BLOCKEDFLOYDWARSHALL(M, n, B)
  $nRounds \leftarrow n/B$
  for T **in** $\{0..nRounds-1\}$:
    // Phase 1
    execute ROUND(M, n, B, T, T, T)

    // Phase 2 (blocks in row w. T)
    execute ROUND(M, n, B, T, T, J) **foreach** J **in** $\{0..nRounds-1\}/\{T\}$

    // Phase 2 (blocks in column w. T)
    execute ROUND(M, n, B, T, I, T) **foreach** I **in** $\{0..nRounds-1\}/\{T\}$

    // Phase 3 (all remaining blocks)
    execute ROUND(M, n, B, T, I, J) **foreach** I **in** $\{0..nRounds-1\}/\{T\}$
      and J **in** $\{0..nRounds-1\}/\{T\}$

end procedure

procedure ROUND(M, n, B, T, blockRow, blockCol)
  for k **in** $\{T*B\ ..\ (T+1)*B-1\}$:
    foreach i **in** $\{blockRow*B\ ..\ (blockRow+1)*B-1\}$
      and j **in** $\{blockCol*B\ ..\ (blockCol+1)*B-1\}$ **do in parallel**:

      if $M[i,k]+M[k,j] < M[i,j]$ **then** $M[i,j] \leftarrow M[i,k]+M[k,j]$

    Synchronize all threads of the same block

end procedure

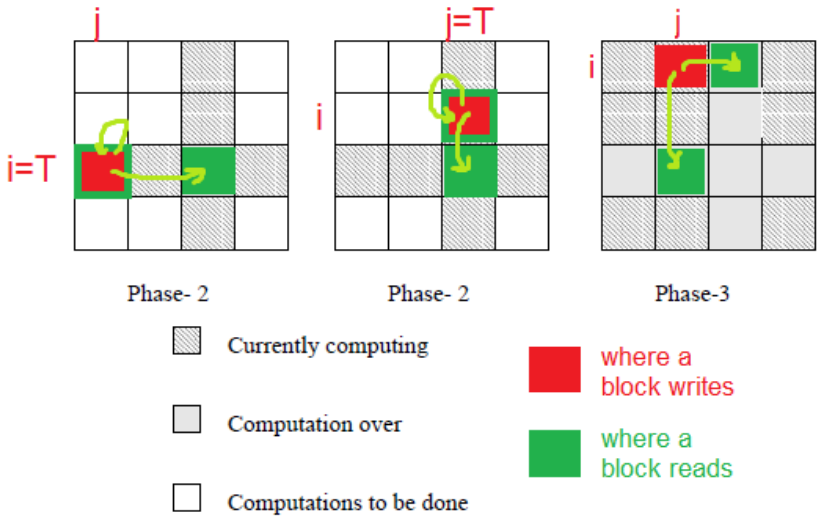

Figure 2.1: In-Round dependencies of blocks of phase 2 and 3.

2.2 Dependencies between phases and in-round parallelizations

The basic in-block parallelization explained in section 2.1 is just one of two important possible parallelizations. Analyzing the algorithm from a wider perspective, we noticed that each round's phases blocks are potentially executable together (**until you keep the phases distinct**). That is possible because in phases 2 and 3 each block (I, J):

- reads (I, T) and (T, J);
- writes just on (I, J).[1]

This concept is visually explained in figure 2.1 and represents the first main optimization we developed, in the version 3.1.2.

2.2.1 Dependencies relaxation with a graph

If you look deeper into the algorithm, you may notice how phases' distinction assumption can be relaxed in a more generic constraint:

Run round K on a block (I, J) only when:

- you already had run round $K - 1$ on (I, J),

[1] Small note for phase 2: for the blocks who share the row with (T, T), that means they read (T, T) and (T, J) and they write in (T, J); for the blocks who share the column instead they read in (I, T) and (T, T) and they write in (I, T).

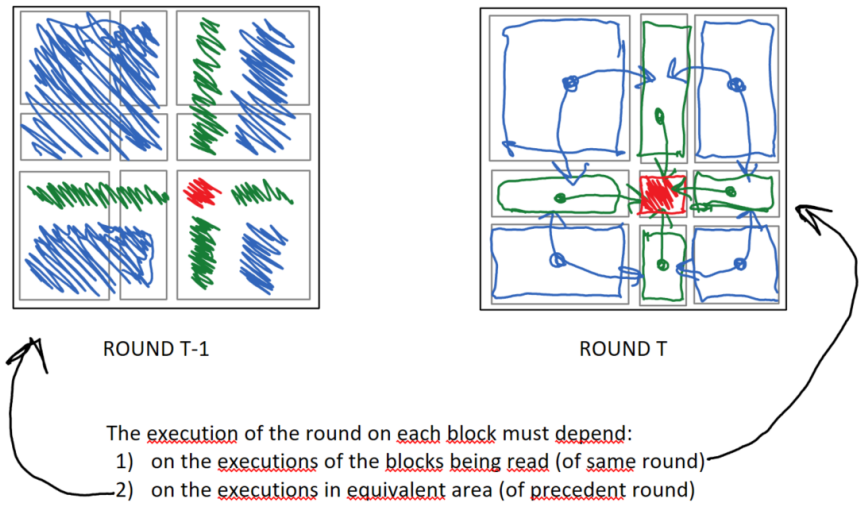

Figure 2.2: General idea of how dependencies can be modelled through a graph.

- you also had run round K on all blocks (I, J) reads.

From this relaxed assumption, we can imagine modelling the algorithm as a *dependency graph of executions on single blocks*, where each phase depends on the block it reads and the equivalent area in the precedent round (see figure 2.2).

This idea is what made us develop versions 4.* (see 3.4), where we experimented CUDA Graphs. For now, our best achievement from this point of view is version 3.2 (see subsection 3.4.2), where we model dependencies as shown in the figure 2.3.

2.3 Blocking factor size choice and CUDA Shared Memory

2.3.1 Blocking factor size choice

Choosing the right block size is a key aspect of algorithm efficiency. As explained in section 1.3, we want to choose a block size optimal for our calculator's architecture. In our case, we work with the NVIDIA graphic cards, so we expect to have:

- a relatively big ($\geq 8GB$) global memory, shared between all the threads of all grids, (which can reasonably contain the whole adjacency matrix of graphs with $n \leq 10.000$ nodes);

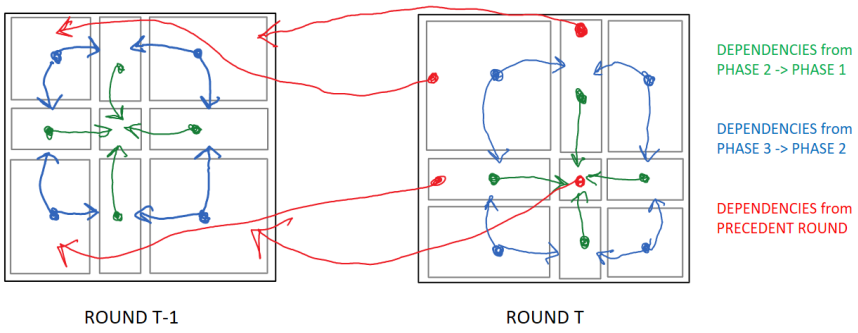

Figure 2.3: Rough modelling of dependencies between phases, as implemented in version 3.2 (subsection 3.4.2).

- an L2 cache (shared between all threads) of some thousands KB, managed by the device (which is used as data cache, but also as constant cache and instruction cache);
- a $128KB$ L1 cache *for each streaming multi-processor*.

Because of this architecture, you want to compute each matrix block in the same streaming multiprocessor, so L1 cache will presumably optimize access for data elaborated by block. Because in CUDA you organize your thread grid in blocks (and because you know the threads of a block will be executed together on the same streaming multiprocessor), it is reasonable to make correspond a matrix block on a CUDA block. Moreover, this helps even with the writing of the code, because it permits you to handle thread synchronization in an easy way.

For all the versions we developed, we make also correspond CUDA block size with matrix block size (making each thread work just on one cell), so if the maximum CUDA threads per block are $32 * 32 = 1024$, we use a maximum blocking factor of 32. We recognize this is a limit and that would be appropriate to try making each thread analyse more than one cell through a further FOR cycle.

2.3.2 CUDA Shared Memory

Another advantage of the choice to make correspond a block of the algorithm to a block of the matrix is the possibility to use CUDA shared memory. Shared memory will be an effective tool for all those situations where you are sure a certain cell will be accessed several times by different threads of the same block. Given a grid of $B * B$ thread that is executing a round on a $B * B$ piece of the matrix, we have the situation described below.

- Each thread accesses his correspondent cell $M[i, j]$ B times (through the

interaction $k \in \{T * B..(T+1) * B - 1\}$) → you wanna guarantee faster access to it, so you want to store this value in a thread's local register or in block shared memory (you choose the first if that cell will be accessed only by his correspondent thread, the second if it will be at least read also by others threads in the block).

- In phase 1, the thread cell $M[i, j]$ is compared with all $M[i, k] + M[k, j]$, so threads of row i access each $M[i, k]$ exactly B times (and same is for column j); you may notice each cell is written by its correspondent thread, but read by all others → the logical choice is to execute phase 1 entirely using shared memory.

- As explained in figure 2.1, phase 2 always write on its correspondent block and read on correspondent block and (T, T). As in phase 1, you can see each cell $M[i, k]$ or $M[k, j]$ will be accessed multiple times, and, as in phase 1, you may notice cells $M[i, j]$ are not just written by their thread but also read by others → you choose to store both the block you are working on and the self-dependent one in shared memory.

- From figure 2.1 you can notice phase 3 is different than the previous: this time threads access each $M[i, j] \in$ *current block* just through their correspondent thread (i, j) → you choose to store only the read blocks in shared memory and instead you use local registers for the current block.

In version 2.0 (subsection 3.2.1) we implemented a basic use of shared memory (as described by this subsection). Shared memory maximum size hasn't been an issue, because even when we allocated space for two blocks of size $2 * B * B * sizeof(int) = B * B * 4$ *bytes*, in the worse case we had blocks of size $32 * 32$, so a memory requirement of just $2 * 32 * 32 * 4$ *bytes* $= 8192$ *bytes*.

2.4 Some notes on bank conflict prevention

As studied during the course, concurrent access to array matrices stored in shared memory may lead to bank conflict, and so on serialization of accesses. To prevent that, we take the countermeasures described below.

2.4.1 Different solutions on different phases

In this subsection, we analyze how bank conflict affects different phases in different ways. To simplify, let's assume we are working with blocks of size $B * B$, where B is also CUDA shared memory bank size. We will discuss what happens when you work with blocks of different sizes in subsection 2.4.2.

Note that for simplicity, to discuss this, we will use an in-block local addressing format (so M will be seen as a $B * B$ matrix, where indexes i, j are in $\{0..B - 1\}$).

First of all, we recognized that in almost all phases there are at least $\Theta(B*B)$ concurrent accesses where bank conflicts are just inevitable (the threads that,

for each k access to their block). Those accesses lead to $Theta(B)$ conflicts, so those accesses are scheduled in at least $Theta(B)$ clock cycles.

Bank conflict happens when threads perform a "column access" on a certain set of cells $M[i, k]$. That happens because, on a certain round k, all threads of block access cells are located in the same column k (so, because bank size is B, they lay in the same bank).

It doesn't instead happen when access is in a row $M[k, j]$, because in this case threads access to cell which lays on different banks.

By analyzing how different shared memory areas are used by different phases' codes, you can distinguish:

- areas accessed just "by row", which are:
  - block (T, J) in phase 3,
  - block (T, T) in phase 2 on blocks which shares a column with (T, T),
  - block (T, J) in phase 2 on blocks which shares a row with (T, T);
- areas accessed just "by column", which are:
  - block (I, T) in phase 3,
  - block (I, T) in phase 2 on blocks which shares a column with (T, T),
  - block (T, T) in phase 2 on blocks which shares a row with (T, T);
  - areas accessed in both ways (just block (T, T) in phase 1).

  Bank conflict on areas accessed just "by column" can be easily solved *transposing* the matrix; while areas accessed just "by row" instead have no bank conflict, so they can and should be left as they are.

  Areas accessed both ways instead should be solved with the classical technique of padding, so you add an empty cell after each logical row in array matrix and access to cell $M[i, j]$ is not anymore done with formula $ArrMatrix[i * B + j]$, but with $PaddedArrMatrix[i * (B + 1) + j]$. That helps you prevent bank conflict both for row and column access.

2.4.2 Bank conflict detection according to block size

A final important note on bank conflict is that it doesn't always happen. In a previous discussion about bank conflict prevention techniques, we assumed the blocking factor B is equal to bank size, but that isn't always the case.

Typically, NVIDIA GPU architectures have 32 banks of the size of an *int* (let's call bank size $B^* = 32$). Our program's array matrices in shared memory have instead size $B * B$, where $B \leq 32$. So, when do we have bank conflicts?

Bank conflict happens if you access at the same times two cells $M[i, j]$ and $M[i, j']$ s.t. their array matrix indexes $i * B + j = i' * B + j' \ (mod\ B^*)$.

Because our concern is on column access (you can easily see how row accesses don't bring to conflicts), we have bank conflicts when $i * B + k = i' * B + k \ (mod\ B^*) \iff i * B = i' * B \ (mod\ B^*)$.

Let's take $i = 0$ and $i' = lcm(B, B^*)/B$ (this last value is surely an admissible index when $lcm(B, B^*) \leq (B-1) * B$). If we take them we can clearly see that

$$0 * B = \frac{lcm(B, B^*)}{B} * B \ (mod\ B^*) \iff 0 = lcm(B, B^*) \ (mod\ B^*)$$

is true, because $lcm(B, B^*)$ is surely a multiple of B^* and so it $= 0 \ (mod\ B^*)$.

This way, **we demonstrated that** $lcm(B, B^*) \leq (B-1) * B$ **implies we will surely have bank conflict**. For this project, we didn't reach to demonstrate the contrary (so that each bank conflict on column access implies $lcm(B, B^*) \leq (B-1) * B$), but with this small math reasoning we created a way to detect situations where we know bank conflict happens, and so we have a tool to decide where to apply or not apply countermeasures.[2]

All this is implemented in v

[2]Note that there are some situations where there is no bank conflict, but the applying of the countermeasure make it happen. Take as an example case where $B = 31$ and $B^* = 32$: you don't have bank conflict, but you will have it if you add padding in your array matrices.

Chapter 3

Our parallel implementations proposals

3.1 Versions 1.* - Global Memory

As for every project, in the beginning, the aim is to understand the algorithm and apply it correctly. Once it is done, the focus moves to optimization.

3.1.1 Version 1.0

The first implementation, so bad that will not even be included in the analysis, uses one function for all three phases. Inside the first for loop ($t = 0$ .. n/B), which can't be parallelized due to dependencies, after the code for phase 1, there are 4 for loops for phase 2, and then again 4 for loops for phase 3 (launching one block at a time). Although CUDA is asynchronous, this is inefficient since all blocks of phase 2 could be launched together, and the same is for phase 3.

3.1.2 Version 1.1

This version achieves two main optimizations: (1) the internal loops have been removed and all blocks of stage 2,3 are launched together and (2) the kernel function has been specialized, one for each stage (before was just one). The code resulting is much cleaner. Obviously, now the kernel for stages 2 and 3 is launched with many blocks. This feature sophisticates how to calculate properly the indexes to use. To avoid an

over-complication all exiting blocks were executed, with some of them remaining always inactive.

3.1.3 Version 1.2

Version 1.1 was launching some blocks for no reason. The number of blocks per row is $r = n/B$, and the total number of blocks is r^2. In phase 2 the blocks needed are the ones on the same row or column as the block on the diagonal, so $(2 * r - 1)$ blocks. This implies that the blocks actually used in percentage were less than $2/r$, a value which tends to zero increasing the dimensions. In phase 3 the situation was not so bad, the blocks used were $(r-1)^2$ (in this case the percentage tends to be 1). In any case, from now only the needed kernels are launched.

3.2 Versions 2.* - Shared Memory

After version 1.2 there weren't any more trivial optimizations, except one: the use of the shared memory instead of the global memory, which is known to be pretty much faster.

3.2.1 Version 2.0

The code here is very similar to version 1.3, the only difference is the use of the shared memory. Which in practical means to add the code for the dynamic allocation of the space for the phases:

1. B^2 ints for $block[t, t]$
2. $2 * B^2$ ints for $block[t, t]$ and $block[i, j]$
3. $2 * B^2$ ints for $block[i, t]$ and $block[t, j]$ (note that the shared memory is not needed for $block[i, j]$ since is sufficient to use a variable).

Inside the functions needs to be added the calculation of the indexes and the copy of the values from the global to the shared memory. Once all the threads have copied their value is possible to proceed as before. After that remains only, at the end, to copy back the new values from the shared to the global.

3.2.2 Version 2.1

The version 2.1 is pretty similar to the 2.0. The main known problem of the shared memory in CUDA is the so called bank-conflict. The aim of this version is to avoid it. How it is done it is already explained in chapter .... .

3.3 Versions 3.* - CUDA Streams

3.3.1 Version 3.0

The version 3.0 has been developed to try the performances of different streams than default. For an easer indexes manipulation, from the version 2.0 the phase 2 was divided in two functions, one for the blocks on the same row of the self-dependent block and the other for the ones on the same column. These two functions are launched one after the other. Thanks to CUDA asynchronous execution, anybody would expect that this shouldn't impact the performances. Actually this is the case, unless the two functions are both launched on the same stream, as in all precedent versions. In fact, using the same stream causes an implicit synchronization. So, to avoid it, we ran the functions into different streams.

However, as you will see on chapter **??**, there isn't any improvement.

3.4 Versions 4.* - CUDA Graphs

In the versions 4.* we implemented the dependencies between the various steps in a graph. Of course the creation of all the nodes and the edges adds some time, but, if the dependencies are elastic maybe some good results may come out.

Sadly, this is not our case. Probably the dependencies could be implemented in a more efficient way, but seems like that graphs brings a lot of supplementary work, both to the programmer during implementations and to the devices while executing.

In fact, although using the functions of the version 3.2.2, the performances we obtained were really awful, worst than all the versions except 3.1.1 and, only for version 4.0, 3.1.2.

3.4.1 Version 4.0

In the first implementation we already relaxed the dependencies present in the other versions. dipendenze piu elastiche ma sempre allinterno dei round

3.4.2 Version 4.1

cuda graph dipendenze piu elastiche (in modo molto migliorabile) dipendenze tra fasi di round diversi

Chapter 4

Performance Analysis

The performances of the various versions implemented has been measured both using *nvprof*, official cuda profiler, and *chrono*, a known c-library.

There are many advantages using *nvprof*: is possible to use many parameters to be applied on the kernels. Moreover, shows the time needed by each function, distinguishing between API calls and GPU activities, ecc. There is just one thing that is somehow not immediate to obtain: the total amount of time needed by a piece of code. The problem is that some of the rows described in the output may be executed totally or partially at the same time. So, a trivial sum of all the times should be avoided. In any case nvprof is very practical and useful for a lot of use cases, and is really recommended during the developments to understand the processes.

Chrono instead, is just a library that allows to store in a variable a *time-point* during the execution of your code. So, by using two variables, one at the start and one at the end, the total time can be computed easily with a difference.

The graphs following are originated using chrono as timer.

As a premise, we remember version 1.0 is so slow is not even be included in the graphs, otherwise all other lines would be shrinked together.

We used as test cases all the following couples of number of nodes: $n \in \{80, 160, 240, 320, 480, 640, 720, 1200\}$ and blocking factor size: $B \in \{8, 16, 24, 32\}$ s.t. B is a divisor of n. These dimensions have been used by Gayathri, Sartaj, and Srabani in their paper.

4.1 Best version by Total Time

The figure 4.1 helps to make an idea on which are the worst versions: as expected the 1.1, but also the graph-based 4.0 and 4.1 which have two

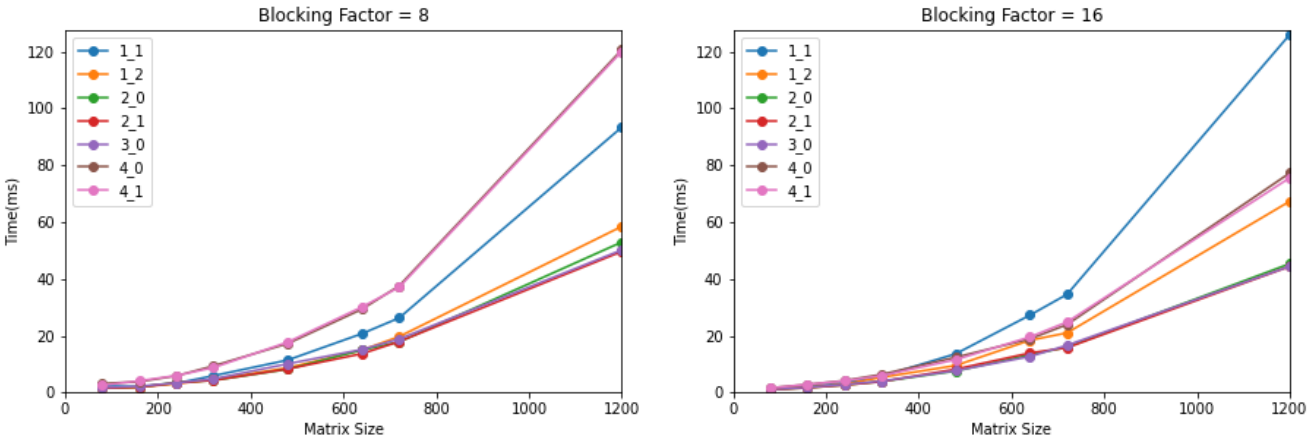

Figure 4.1: Execution time of all the versions

lines almost identical. The version 1.2 also appears to be a bit slower. So, in the next graphs we will show only the possible optimal versions: 2.0, 2.1 and 3.0.

The figure 4.3 is very similar to the 4.1, but: has been filtered out the slow versions, has a detail on the x axis ($n = 1200$ has been removed) and has 4 boxes (added $B = 16$ and $B = 32$).

From the chart emerges that there is no a best version, but their performance depends on the blocking factor size. With $B = 8$ the 3.0 is the slower and the 2.1 is the best one. With $B = 16$ and $B = 24$ the lines almost overlaps, with the 2.1 slightly worse with $B = 16$. A big difference is visible on the fourth panel with $B = 32$, where the 2.0 performs really bad. This probably is caused by the non prevention on bank-conflicts, which increase in number having bigger blocks.

For the analysis of the blocking factor sizes we chose the version 2.2 to be the comparator.

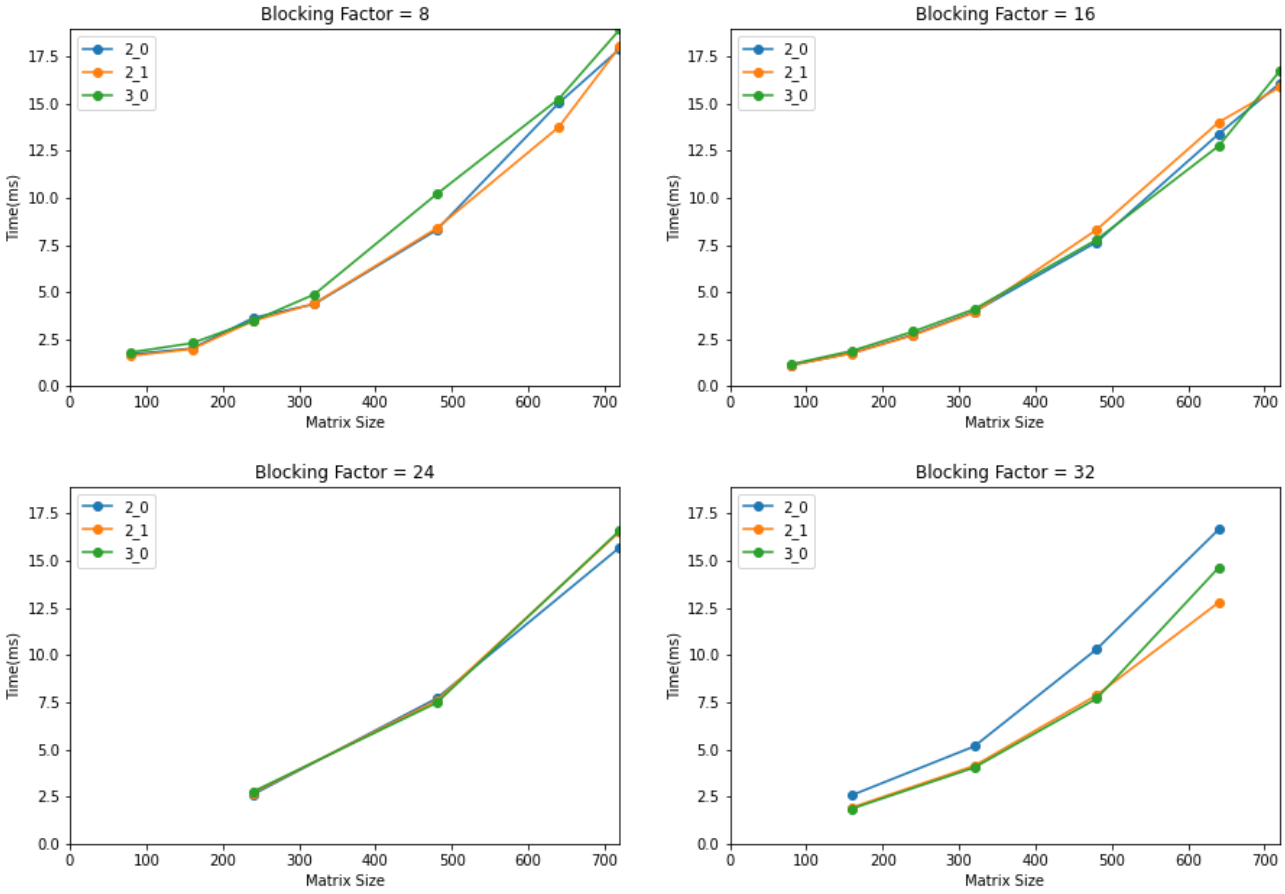

Figure 4.2: Execution time of the best versions

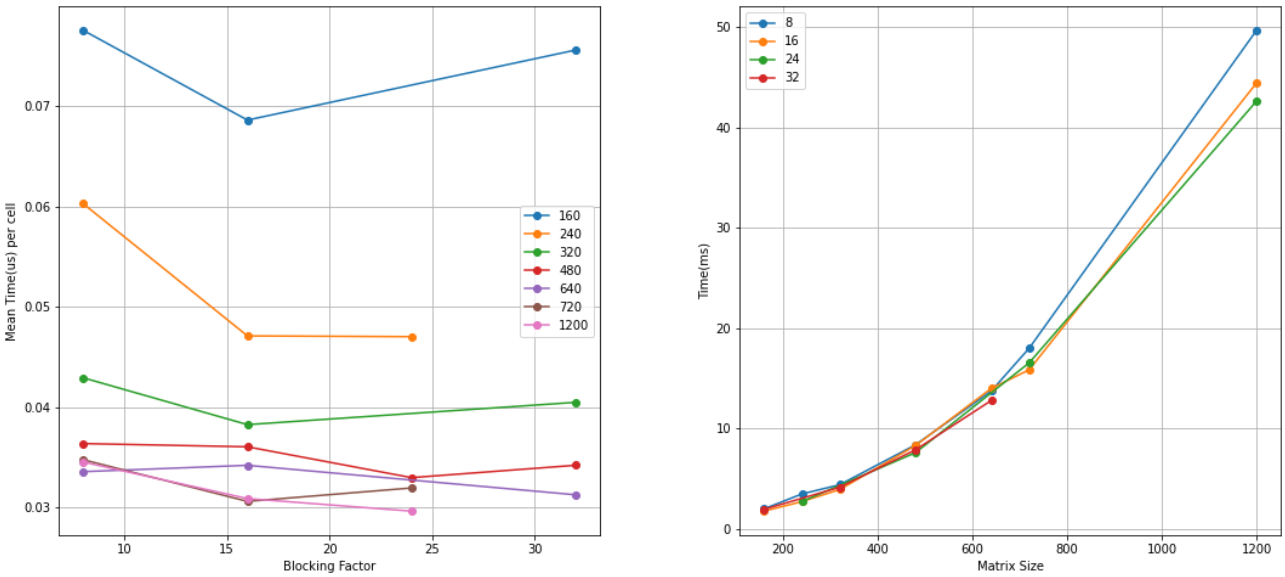

Figure 4.3: Execution time of the best versions

Chapter 5

Conclusions

5.1 Missing features and possible future developments

Due to some timing reasons, the implementation lacks some features, here is a brief recap.

Values for the Blocking Factor

The original algorithm works with any blocking factor. Obviously using as B a divisor of n simplifies the code. Otherwise, the last blocks will have a size smaller than the others and some additional care must be performed to avoid errors like index out of bounds or unexpected behaviours.

The following implementation works correctly only using as B a value which is a divisor of the matrix size (or n for brevity). So, theoretically, a graph with a prime number of nodes is excluded, but, in practice, a trivial workaround can be applied by padding the adjacency matrix with some dummy nodes without any edge.

Scalability

The main missing feature is surely the scalability of the algorithm. For the sake of simplicity, there are some upper bounds both on the matrix size and on the blocking factor, given by the CUDA device. The maximum n can be ??LENA??, meanwhile for B is the square root of the maximum number of threads per CUDA block, which means, for *c.c.* ≥ 2.0 it is $\sqrt[2]{1024} = 32$.

Next-Hop Matrix

Bibliography

[1] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 8:2–2, 2003. Article available on `https://www.cise.ufl.edu/~sahni/papers/shortj.pdf`.

[2] Floyd–warshall algorithm, Aug 2022. Website: `https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`.