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.

Chosen algorithm is the variant of Floyd Warshall APSP presented by Gayathri, Sartaj, Srabani [1]. The basic idea is, given a graph of size n, to split its adjacency matrix in many squared sub-matrices (*blocks*) of a constant size (*blocking factor* or B). The algorithm has three main steps (called *phases*), and in all of them, there is a sort of Floyd Warshall execution on some blocks. These steps are partially independent and so there are many opportunities to make the executions concurrent. For this reason, this algorithm is perfect to be implemented with the patterns of parallel programming.

In the following report, we start by explaining the main concepts of the algorithm. We proceed to explain the parallelization opportunities the algorithm offers and then we illustrate our implementations, our choices and our ideas. The last chapters regard the measurement of performances of our models and the further possible improvements.

Contents

Chapter 1

The algorithm

The purpose of this project is to propose a parallel implementation of the Floyd Warshall APSP variant proposed by Gayathri, Sartaj, Srabani [1]. This chapter will describe briefly the main aspects of the general algorithm, which are the basis of parallelization opportunities proposed in the next chapter.

1.1 Classical Floyd Warshall

Floyd Warshall All Pair Shortest Path [2] is a classical graph algorithm for the calculation of 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 each node k, so at the end of the algorithm, the matrix M will contain the cheapest cost for each pair of nodes.

The code contains three nested for loops on all the nodes and so the complexity is trivially $O(n^3)$.

1.2 Blocked Floyd Warshall

Blocked Floyd Warshall relies on the idea to partition 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 $\frac{n}{B}$ 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. Practically speaking, 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 computed 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 deepened in Gayathri, Sartaj and Srabani's original paper [1].

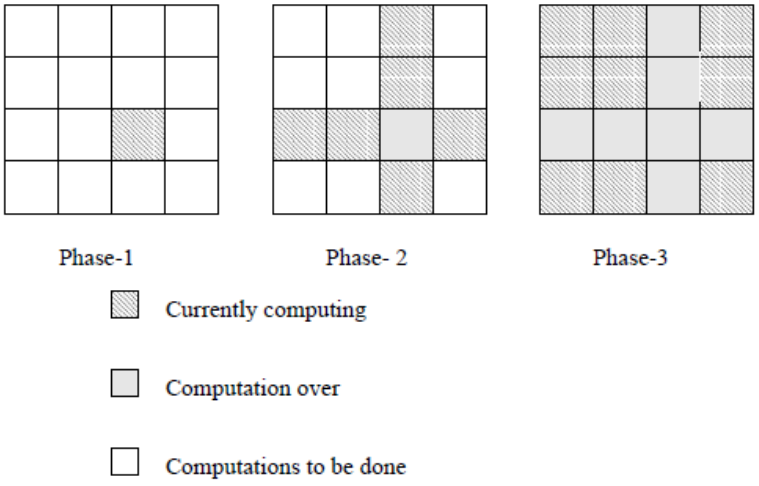

Figure 1.1: Blocks computed on each phase. You can clearly notice how phase 1 involves just the self-dependent block, phase 2 computes the rows and the columns and phase 3 computes the remaining blocks. You may also notice that the self-dependent blocks of rounds are the ones which lay in the matrix's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

Chapter 2

Parallelization Opportunities

In this chapter, we analyze the parallelization opportunities offered by the algorithm. We begin considering a naive parallel Floyd Warshall idea, that is used to implement simple in-block parallelizations on our code. After that, we continue analyzing how dependencies between rounds and phases work. We finish discussing the use of shared memory and bank conflict prevention countermeasures.

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 alternatives" which uses the known paths to each other node. 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 if only:

- 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's code on each block parallelizes the two internal cycles. In algorithm 1 we show the structure of such a code. This idea is implemented in our first basic version of code (explained in subsection 3.1.2) and will be kept for all the next versions.

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

```
procedure BlockedFloydWarshall(M, n, B)
    for T in {0..n/B − 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..n/B − 1}/{T}

        // Phase 2 (blocks in column w. T)
        execute Round(M, n, B, T, I, T) foreach I in {0..n/B − 1}/{T}

        // Phase 3 (all remaining blocks)
        execute Round(M, n, B, T, I, J) foreach I in {0..n/B − 1}/{T}
            and J in {0..n/B − 1}/{T}

end procedure

procedure Round(M, n, B, T, I, J)
    for k in {T ∗ B .. (T + 1) ∗ B − 1}:
        foreach i in {I ∗ B .. (I + 1) ∗ B − 1}
            and j in {J ∗ B .. (J + 1) ∗ B − 1} do in parallel:

            if M[i, k] + M[k, j] < M[i, j] then M[i, j] ← M[i, k] + M[k, j]

        Synchronize all threads of the same block

end procedure
```

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

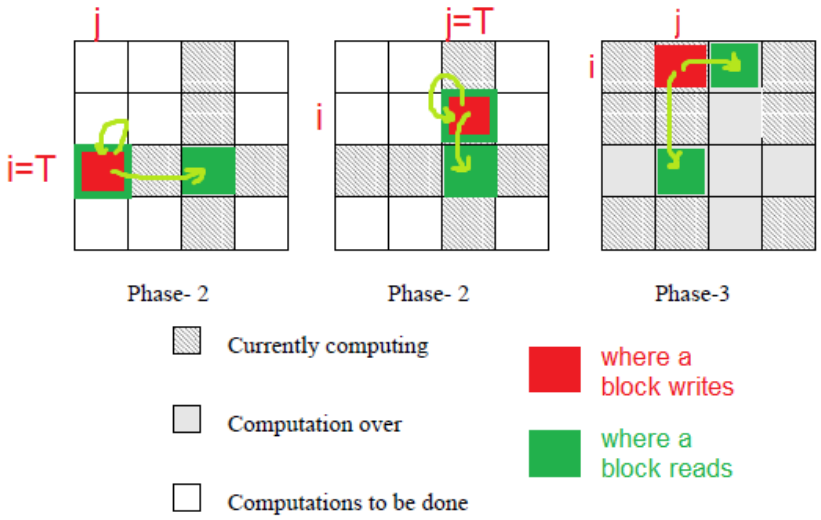

Figure 2.1: In-Round dependencies of blocks of phase 2 and 3.

phases 2 and 3 each block (I, J) doesn't interfere with other blocks of same phase. In detail, each block:

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

This concept is visually explained in figure 2.1 and it is used to implement the first main optimization of version 1.2 (explained in subsection 3.1.3).

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

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).

[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).

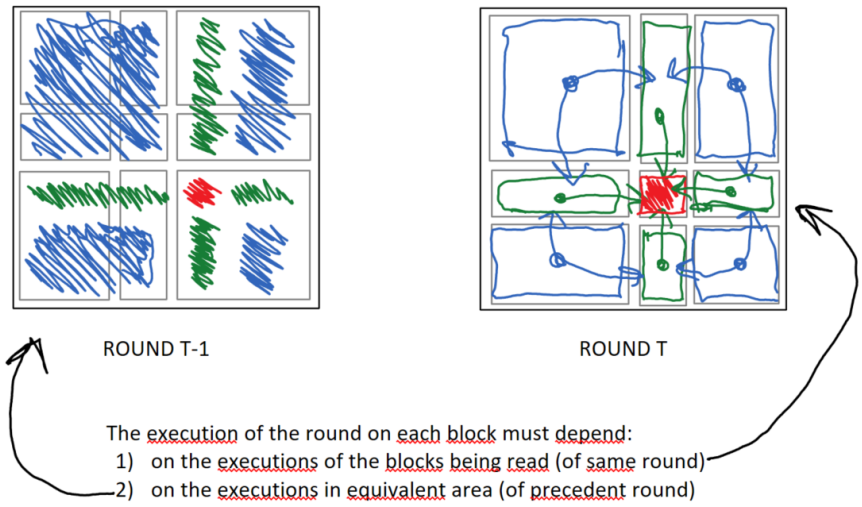

Figure 2.2: General idea of how dependencies can be modelled through a graph.

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. We acknowledge it is possible to realize more sophisticated dependency models we didn't complete for lack of time.

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 NVidia's GPUs, 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 of $n \leq 10.000$ nodes);
- 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*.

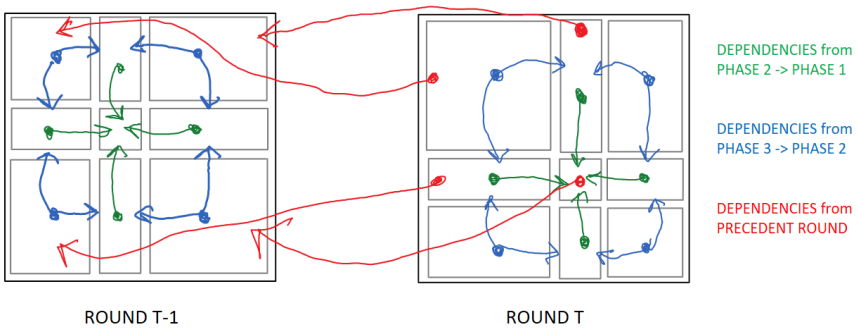

Figure 2.3: Rough modelling of dependencies between phases, as implemented in version 3.2 (subsection 3.4.2).

Because of this architecture, you want to compute cells of the same matrix block in the same streaming multiprocessor, so L1 cache will presumably optimize access for data computed 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 threads block*. Moreover, this also helps the handling of in-block thread synchronization.

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 one or more further FOR cycles.

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 $k \in \{T * B..(T + 1) * B - 1\}$) $\rightarrow$ 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 option when the cell will be accessed only by his correspondent thread and the second when you know cell will be at least read 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 $\rightarrow$ 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 (T,T) and also on the correspondent block. 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 $\rightarrow$ 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 $M[i,j] \in \textit{current block}$ just through their correspondent thread $(i,j) \rightarrow$ 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 worst-case scenario we have blocks of size $32 * 32$, so a shared memory requirement of just $2 * 32 * 32 * 4$ *bytes* $= 8192$ *bytes* (+ registers).

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, each of size $\Theta(B)$; that implies accesses are scheduled in at least $\Theta(B)$ clock cycles.

An avoidable bank conflict happens when threads perform "column accesses" on a certain set of cells $M[i,k]$ w. $i \in \{0..B-1\}$ (because bank size is B and so

all $\Theta(B)$ accessed cells 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 cells which lays all 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 (if you transpose them too, it will happen you create bank conflict).

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 matrices and so, access to cell $M[i, j]$ is not anymore accomplished with instruction $ArrMatrix[i * B + j]$, but with $PaddedArrMatrix[i * (B + 1) + j]$. That helps you prevent bank conflict both for row and column access.

To not waste memory, it's reasonable to apply padding only when necessary (so in our case only in phase 1 self-dependent block copy).

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 integer (let's call the 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 value is an admissible index **only if** $lcm(B, B^*) \leq (B-1) * B$).[2]

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

If we apply them to the formula we can clearly see the condition is true, because $lcm(B, B^*)$ is surely a multiple of B^* and so it $= 0$ $(mod\ B^*)$. This demonstrates 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 thanks to this small math pill, we can build a way to detect situations where we surely know bank conflict happens, and so we have a tool to decide where it's opportune to apply or not apply countermeasures.[3]

All this logic we explain in this section is implemented in version 2.1 code (explained in subsection 3.2.2).

[2] Where *lcm* is the *smallest common multiple* between two numbers.

[3] 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. That's why it's important to apply countermeasures only when necessary.

Chapter 3

Our parallel implementations proposals

3.1 Versions 1.* - Global Memory

Versions 1.* have the purpose to realize naive parallel implementations of the algorithm. We started introducing a simple parallel execution of round on each block, then we progressively parallelized blocks of the same phase and we made some reasonable code cleaning to reach a rational structure (where you don't launch more threads than it's needed and you rationalize indexing).

3.1.1 Version 1.0

The first implementation, so bad that will not even be included in the analysis, defines a basic structure we will keep for most of the next versions.

Code is made by a host function which loops on all self-dependent blocks ($t \in \{0 \,..\, \frac{n}{B}\}$) to perform a sequence of rounds (which can't really be parallelized due to dependencies). Each round is technically made by a serial sequence of kernel calls on all blocks, starting from phase 1 and then launching blocks of phases 2 and 3 through the call of the same kernel function on each block of matrix.

This version is absurdly inefficient since you launch (serially, on default stream) an $\frac{n}{B}$ sequence of kernel functions.

3.1.2 Version 1.1

This version achieves two main optimizations:

1. the internal loops have been removed, so all blocks of phases 2 and 3 are launched together (as just two kernel executions on a grid of blocks, instead of one separate execution for each block).

2. the kernel function has been specialized, one for each phase.

The code now is much cleaner and we also get rid of the illogical serialization of blocks of phases 2 and 3. The two main flaws of this version are that it sophisticates the calculous of indexes and that each kernel execution uses an excessive amount of sleeping threads (especially in phases 1 and 2).

3.1.3 Version 1.2

Version 1.1 was always launching a full r^2 grid of blocks on each phase (where $r = \frac{n}{B}$ is the number of rounds, and r^2 is the total number of blocks of the matrix). This version instead rationalizes the number of blocks launched on each phase:

- In phase 1, you launch a kernel of just one block (the self-dependent one).
- In phase 2 the blocks needed are the ones on the same row or column as the self-dependent one, so you launch $(2 * r - 1)$ blocks (this implies that in version 1.1 the blocks actually used in percentage were less than $2/r$, a value which tends to zero increasing the dimensions).
- In phase 3 you launch a kernel which cover the remaining $(r - 1)^2$ blocks (in version 1.1 the situation was not so bad because the percentage of active blocks was tending to be 1).

3.2 Versions 2.* - Shared Memory

In previous versions, we used to copy the full problem instance on global memory and make all kernel executions works directly on it. Versions 2.* aim to take the structure of version 1.2 and enhance it with shared memory, which is known to be pretty much faster than the global one.

3.2.1 Version 2.0

The code structure is very similar to version 1.2, the only difference is that blocks, before starting their computing, copy what is convenient on a dynamically allocated shared memory portion (and at the end, of course, they copy back on global memory eventual modifications). In detail:

1. phase 1 uses B^2 ints for storing block (T, T)
2. phase 2 uses $2 * B^2$ ints for storing blocks (T, T) and (I, J);
3. phase 3 uses $2 * B^2$ ints for storing blocks (I, T) and (T, J) (note that the shared memory is not needed for block (I, J) since each thread, from all the cells of the block, just uses its correspondent cell; that make more convenient to just use a local register).

The introduction of shared memory implies new responsibilities for threads. Now, other than performing the in-round piece the of algorithm, threads should calculate indexes, copy the values from the global to the shared memory at the beginning and copy back the results at the end.

3.2.2 Version 2.1

Version 2.1 is pretty similar to 2.0, but this time we introduced bank-conflict prevention techniques explained in section 2.4.

3.3 Versions 3.* - CUDA Streams

3.3.1 Version 3.0

Version 3.0 has been originally developed with the purpose of "breaking" phases into pieces, to permit then the modelling of dependencies through CUDA Graphs. Actually, this has been also an occasion to experiment with the use of streams different from the default one.

This version is based on 2.1 code. We started taking phase 2 and breaking it into 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. Then, we modified both phase 2 and 3 device functions to allow the launch on just a portion of the blocks instead of all at once.

Thanks to this, we can break phases 2 and 3 into 4 pieces each (the pieces shown in figure 2.2). Then, single pieces of phases are launched concurrently using different streams.

3.4 Versions 4.* - CUDA Graphs

In versions 4.* we experimented with modelling dependencies between the various steps through CUDA Graphs. That change deeply both the way the programmer writes the code and the actual flow of execution.

3.4.1 Version 4.0

For the first implementation of CUDA Graph version, we take what we made in version 3.0 and we try connecting device function calls with a simple CUDA Graph. The first graph tries following the classical linear structure of algorithm, but with small improvements:

- we start with a whole *mem. copy* node, to copy the matrix from host to device;
- for each round, we create one kernel node for phase 1, four for phase 2 (up, $down$, $left$, $right$) and other four for phase 3 ($up - left$, $up - right$, $down - right$, $down - left$);

- those nodes are connected in a way which replicates the original algorithm structure:
  - phase 1 depends on all previous phase 3 nodes (or on mem. copy operation if it's the first round),
  - phase 2 nodes depends on phase 1,
  - each phase 3 node depends on the two correspondent of phase 2 (as shown in figure 2.2).
- when everything is finished, you add a final mem. copy operation to copy back results on the host.

3.4.2 Version 4.1

Version 4.1 is similar to 4.0, but it tries modelling in a more fluid way the dependencies between different rounds. As shown in figure 2.3:

- we make phase 1 node depend on just final $down - right$ phase 3 piece of precedent round;
- we make phases 2 up and $left$ node depend also on precedent phases 3 $up - right$ and $down - left$;
- we make phase 3 $up - left$ depend on previous $up - left$.

This way we expect to obtain a more fluid workflow, and potentially a higher level of parallelism.

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, distinguishes 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 *timepoint* 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.

Following charts data are obtained 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 are used also by Gayathri, Sartaj, and Srabani in their paper.

All tests have been executed on Kaggle's notebooks, which offers those machine specifics:

- CPU is a one core Intel(R) Xeon(R) CPU @s 2.20GHz
- RAM total space is 3059712 kB
- OS is a Linux de67827c50a5 5.10.133+
- GPU is a NVidia K80 GPU

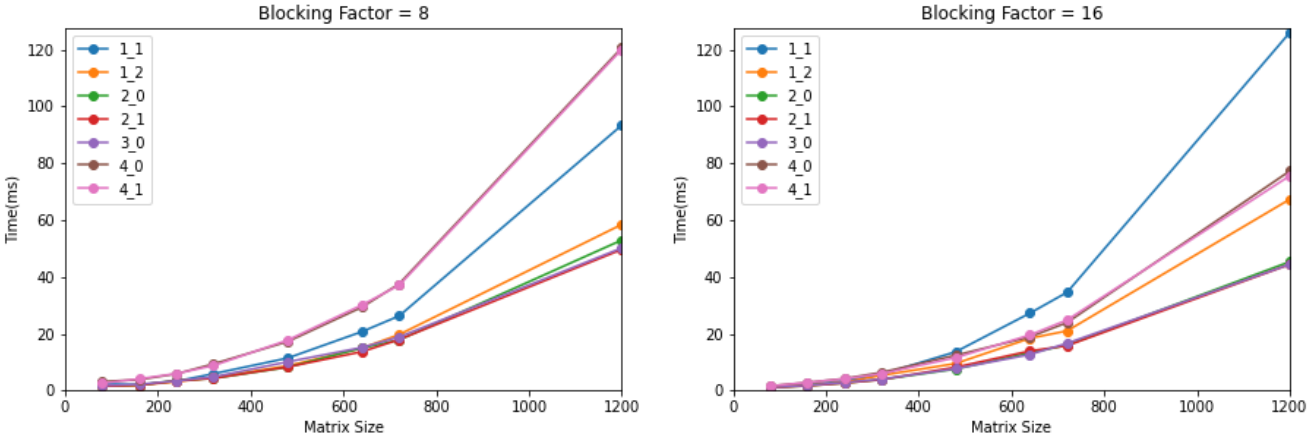

Figure 4.1: Execution time of all the versions

4.1 Best version by Total Time

4.1.1 All versions comparison

The figure 4.1 helps to make an idea on which are the worst versions. As expected version 1.1 is the worse; also the graph-based 4.0 and 4.1 have very bad performances (which are two almost identical lines). Version 1.2 also appears to be a bit slower than others. In the next charts, we will focus on more performing versions 2.0, 2.1 and 3.0.

We may explain version 1.1 inefficiency knowing it launches a great number of useless threads, which are always inactive and keep the GPU busy. Version 1.2 instead launches only the needed threads, saving a lot of time and memory. Nevertheless, it is still based on the global memory which is incredibly much slower than the shared.

Each kernel makes 3-4 operations of r/w for each iteration of the for loop (which is repeated B times). So, the total number of operations performed on the global memory per kernel is $>= 3 * B$.

Figure 4.3 shows how different blocking factors perform with version 2.1 (which uses the shared memory, with bank conflict prevection). For all the sizes of the matrices, the best blocking factor is never $B = 8$, because each r/w op. is very fast and what makes slow the algorithm is not related to the access to the memory. Instead, if you would create the same plots using version 1.2, the performances of the blocking factor would be the opposite: a smaller B implies a less number of accesses to global memory and so to a lower time. The aforesaid chart has been omitted for brevity.

Versions 4.0 and 4.1 can be considered together since they have the same timings for all the graph sizes. The two main reasons for their inefficiency are those:

- the dependencies implemented in 4.0, also if theoretically are quite coarse, are so good that the fine-grained of the 4.1 doesn't make any real improvements;
- although the structure of the dependencies' graph of 4.1 is smart, there is no actual improvement because the inefficiency of the two versions is caused by the overhead originating from the creation of the nodes and the edges.

A deeper analysis of the inefficiency of the CUDA graphs in our code has a distinct section in the conclusions 5.4.

4.1.2 Comparison of 2.0, 2.1 and 3.0

Figure 4.2 is very similar to 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 isn't a universally best version, but their performance depends on the blocking factor size: with a small blocking factor ($B = 8$) the stream based 3.0 is the slower and the 2.1 is the best one; with $B = 16$ and $B = 24$ the lines almost overlap, with the 2.1 slightly worse with $B = 16$.

A big difference is visible on the fourth panel with $B = 32$, where 2.0 performs really bad compared to the other two. This probably is caused by the non-prevention of bank conflicts, which increase in number having bigger blocks.

In general, 2.0 seems to be slightly worse than 2.1 and 3.0.

4.2 Best Blocking Factor by Total Time

To analyse the performances of the different blocking factor sizes we chose version 2.2 as comparator.

In the left chart of figure 4.3 we have a line per each blocking factor. As expected there isn't an absolute best blocking factor but it depends on the matrix size and on many other factors (for example, as we said before, the use of the shared or the global memory).

To choose the best B value there is no a direct correlation (the bigger the n, the bigger the B): with $n = 640$ the best $B = 32$ meanwhile with $n = 1200$ $B = 16$ performs better than $B = 24$ ($B = 32$ is not currently possible, it's not a divisor of 1200). Surely $B = 8$ is never a good choice, in all cases exists a bigger B (16, 24 or 32) which has a lower execution time (in the chart per each colour the dot of $B = 8$ is never the minimum and also all the trend lines, the dashed ones, have a negative angular inclination).

What's most important here to say is that, differently from the sequential implementation of the paper, the execution time is not that much effected from the size of the blocking factors. Some B for sure are better or worse than

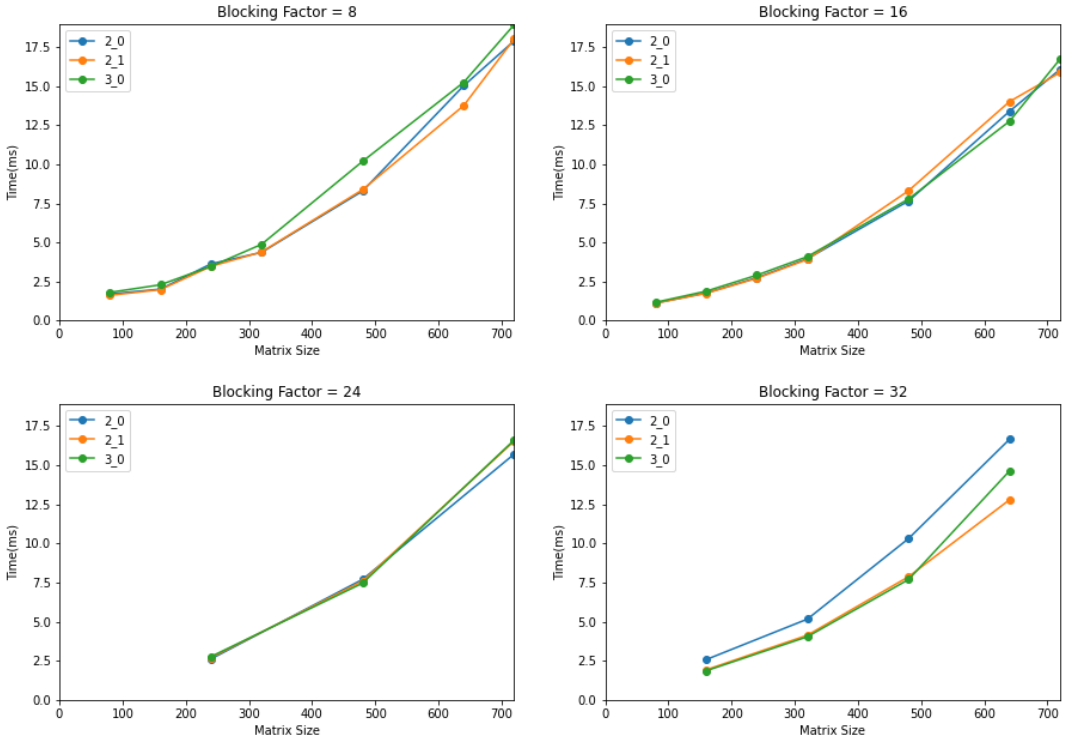

Figure 4.2: Execution time of the best versions

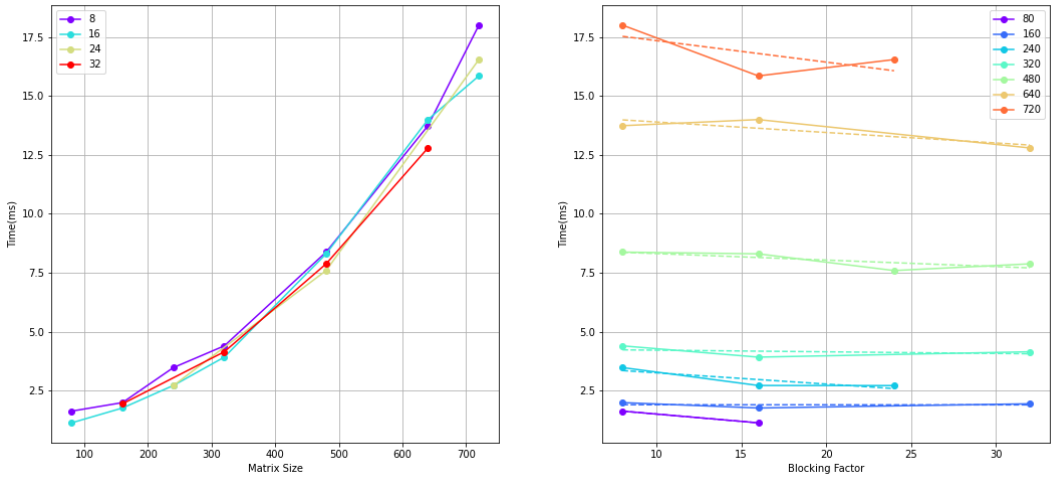

Figure 4.3: Blocking Factors performance comparison

others, but the use or the non use of the optimal B effects only relatively the performance.

Chapter 5

Next Steps

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

5.1 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.

Remove this limit is quite easy, needs just some care on handling the indexes of the last blocks that will be smaller.

5.2 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 depends on the size of the global memory, meanwhile the upper bound for the B is given by the square root of the maximum number of threads per CUDA block, which means, for *c.c.* ≥ 2.0 it is $\sqrt[2]{1024} = 32$.

Scalability surely would be a great improvement on the algorithm, but it is also the hardest part. Realize a solution for a general blocking factor could be done in two different ways:

- making each thread working on many cells using a for loop. The impact

on the performance will depend on the number of different cells assigned to each thread.

- breaking the initial constraint of the relation (1,1) between CUDA blocks and matrix blocks. Also in this case some worsening is expected since the function $\_\_syncthreads()$ wouldn't synchronize all threads of the matrix block, which means a more sophisticated synchronization becomes necessary.

5.3 Next-Hop Matrix

The current algorithm provides only the matrix of the APSP, without giving the actual paths to follow to obtain such cost. The implementation of the next-hop matrix should be quite easy: for each couple on nodes a list of all intermediary nodes is kept. Every time a better path through the node k from i to j is found, the path (i,j) becomes the concatenation of the paths i to k and k to j.

5.4 Faults on CUDA Graphs based versions

It's important to note that versions 4.* are just a very naive experiment of what can be done with graphs. Both 4.0 and 4.1 share some important faults which may be serious impediments to performance; down here we try to explain the main flaws.

- The memory copy operations are performed as unitary nodes while, potentially, given a different data structure, could be broken into smaller separate executions. That would permit us to start executing the computations *before* all memory copy is performed (and so, that would permit us to start copying results before finishing all computations).

  To solve this issue we should investigate alternative data structures (or more complex copy procedures) which would allow breaking the mem. copy operations into smaller pieces similar to phases.

- Each kernel node uses an excessive amount of threads. Because of a limitation of CUDA Graphs, all kernel nodes seem forced to share the same grid size. That made kernel nodes often launch more threads than it's needed, and so have again similar problems of version 1.1.

  To solve that you may try experimenting to use very small device functions which operate on just one block (similarly to what we did in version 1.0, but launched concurrently thanks to graphs dependencies).

- All the graph structure management brings a big overhead; in fact, the graph is first built and then executed. To solve that you should work on launching a graph before is complete (to add dynamically the nodes while the first stages of computation are made).

- Graph management code becomes really complex to handle for the programmer. This last issue could be solved through continuous refactoring and breaking of the code logic into smaller, more rational, pieces (e.g., applying classical refactoring techniques, such as extracting functions, grouping parameters into structs, etc.)

Chapter 6

Conclusions

In this project we implemented an NVidia GPU-based version of the Blocked Floyd Warshall APSP algorithm. We experimented with many possible implementations, starting with some basic ideas and obtaining more complex versions which perform better and/or cover different parallel programming issues.

Through this project, we experienced some of the main advantages and challenges of GPU-based parallel programming. But most of all, we understood that there is still a lot of work to be done to reach state-of-art results.

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`.