High Performance Computing - Project

Leschi Cyprien, Beziaud Jordan, Jaman Steven

11/02/2022

1 Introduction

The goal of this project is to design, implement and profile the performance of a program that uses a parallel paradigm. The approach here will be to parallelize a serial program *multiattack.c* used for cracking hash coming from a shadow file by comparing known hashes from a wordlist on a given shadow file.
The sources are available here : https://github.com/Anatharr/HPCProject

2 Architecture

We narrowed down the entire problem to a comparison between 2 files : the wordlist (hashed version) and the shadow file. In that type of exercise, the wordlist is usually way larger than the shadow file as it must contains the larger pool of possible guesses in order to increase the chances that a given hash from the shadow file can be guessed.
Therefore, we decided that the wordlist needed to be divided into smaller chunks of words, to feed it more quickly to the given hash. This way, if the first word in the wordlist was the plain text of the first hash, we wouldn't need to pass the entire database to find out.
Then, we focused our attention on parallelizing the hash guessing in the sense that each processor would get his own wordlist's block and the hash that he needs to guess. This way, we only need to load the shadow file once !
For a short wrap-up, our main objective with this implementation is to load the 2 file just once, and use the parallelisation paradigm to guess each hash at the same time, speeding-up the process really quickly. For that, we decided to use **CUDA** on top of the **C++ language**.

3 First measurements and theorical speedup

The first metrics that we decided to use was the rough time of execution of the program, using built-in C function. It would then gives us insights about the potentials factors that needed to be tweaked to get a better speed-up.
The first thing that came in mind was the way of programming things with

CUDA. There was basically 2 functions used to allocate the memory and then copy it to the GPU. One managed by CUDA that used unified memory (linking GPU and CPU address spaces) and the other that let us handle the memory management by hand. The use of the un-managed one allow us to reduce the total execution time by a factor of 17 which was a decent way to start toward optimization (appendix-1). As such, not using unified memory (see appendix-3) allows us to get a **stable total time of execution despite increasing the workload, which demonstrate a great scalabality of our implementation**.

For the theorical speed-up, as in our implementation one hash is linked to one thread, we tried to scale up the number of hashes read from the shadow file to draw up the Amdahl law. From a sample of 3000 hashes, we calculated the serial and parallel execution time and deduced an average of :

- average total execution time : 7.72 seconds
- average serial execution time : 6.756 seconds
- average parallel execution time : 0.96 seconds

Note that increasing the number of threads increase the difficulty of the tasks the more there is thread, the more there is hashses to guess. Therefore, some metrics such as efficiency cannot be considered or are biased if we compute it using the usual way.
Therefore, we get the following serial fraction : $f = \frac{6.756}{7.72} = 0.87$ which gives us the theorical max speedup : $\frac{1}{f} = 1.142$.
Then, we can have the theorical max Amdahl's law plot (appendix-2).

4 Optimizing the speedup

In order to optimize our program, we tried three different versions :

- The first one uses CUDA's Unified Memory paradigm, with calls to cudaMallocManaged(). This mode dramatically lowers Developper effort by adding an abstraction to memory management between CPU and GPU automatically handled by CUDA, but it comes at a cost. We called this implementation "Managed" mode.
- In the second one we replaced all calls to cudaMallocManaged() with calls to cudaMalloc() and we handle all memory transfers manually, which is far more efficient, especially on Window. We called this implementation "Unmanaged" mode.
- The third version stores all hashes which were found in order to stop trying to crack them afterwards. Unfortunately we ran out of time and we couldn't finish this "Optimized" implementation.

We observed a huge improvement between Managed and Unmanaged versions, however as our implementation uses one thread per hash and we already cut the wordlist in several blocks, the next optimization could be to run several wordlist blocks in parallel.

5 Experimentation results and conclusion

If we take a look at our benchmarking results, we can see that the amount of time spent in parallel functions is pretty low in comparison with the amount of operation realized, which means that our parallel kernels are not useless.

However we computed our practical parallel proportion by doing the average over several runs (with different number of hashes to crack), and it is around 12.5% (on windows) which is really low. It means that a lot of time is spent in serial operations, and we could maybe parallelize some of those operations in order to improve the speedup of our implementation.

The speed up graph shows that for both linux and windows the "Managed" implementation is better than the "Unmanaged" one. Furthermore, The linux version out perform the windows version. All curves tend to increase and we can observe an acceleration in that increase for the windows implementation "Unmanaged" mode.

6 Appendix

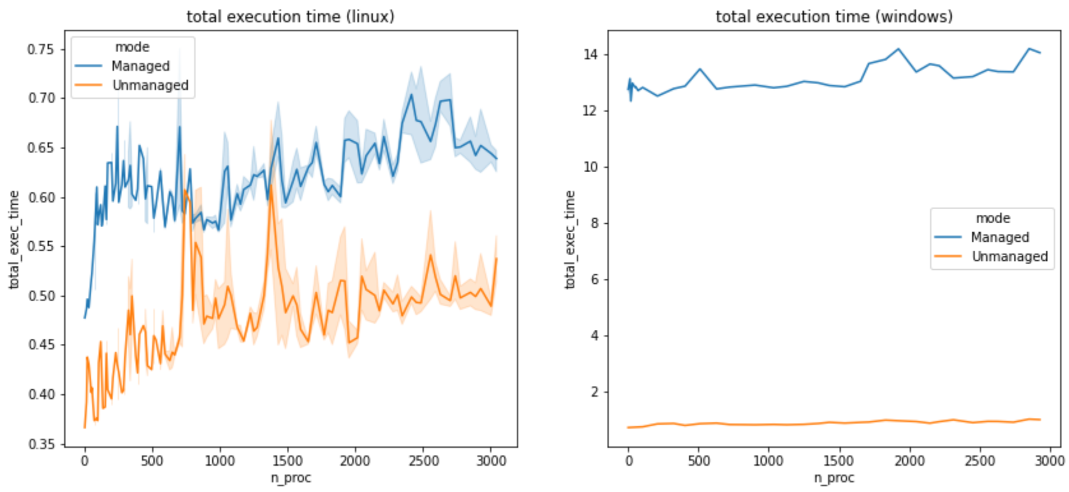

Figure 1: Total execution time given the mode of memory management

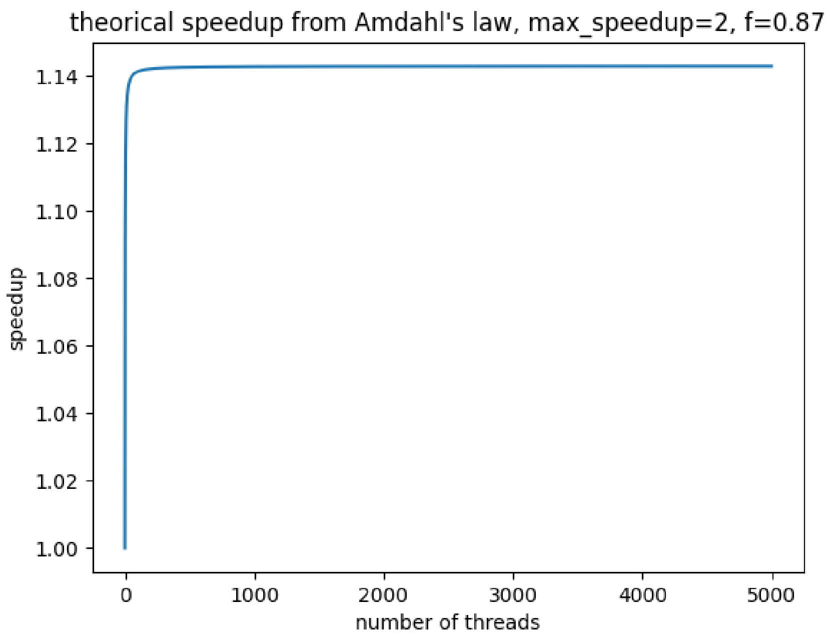

Figure 2: Theorical Amdham law

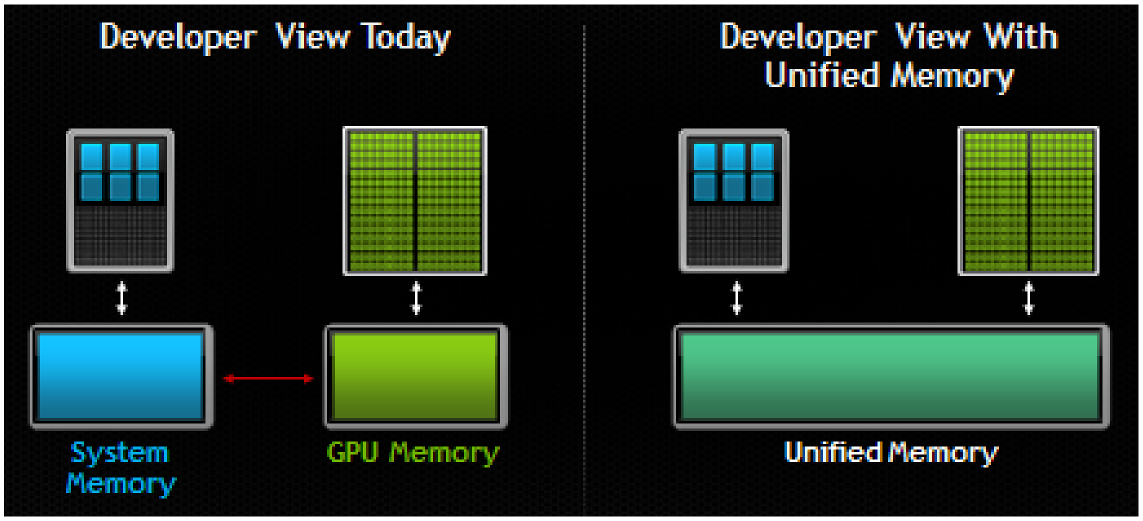

Figure 3: Explanation of the unified memory