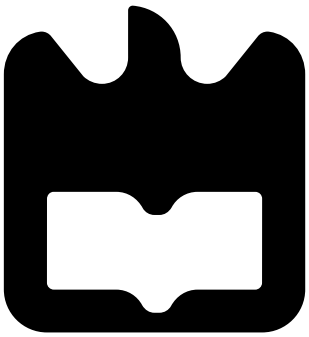

AAD - Assignment 2

107474-Joseane Pereira
109050-Gabriel Costa
Universidade de Aveiro, DETI

December 23, 2024

Contents

1 Introduction

2 Method

2.1 AVX and AVX2

For the AVX implementation, part of the code was already provided to us, so our task was to only implement the search function. To achieve this, we initialized all lanes with the standard DETI coin format. To introduce variety, we modified a given integer of the coin (stored as **v1** in each lane) by adding the lane number.

To maintain continuous variety across iterations, we increment the v1 value in each lane by the total number of lanes (4). To ensure these values remain within the ASCII range, we utilized the "next_value_to_try_ascii" function. If v1 overflows, we increment the next integer (v2) and reset v1 to its initial value.

The AVX2 implementation required only minor adjustments to the AVX code, as it processes 8 lanes instead of 4.

2.2 OpenMP

For OpenMP we used the AVX2 search function and parallelized it using OpenMP. We used the pragma **#pragma omp parallel for** to parallelize the search function. Similiar to the AVX2 implementation, we initialized all lanes with the standard DETI coin format. To introduce variety, we modified a given integer of the coin (stored as **v1** in each lane) by adding the thread number to the coin's last integer, and at the end all the numbers of the attempts, as well as the coins found, are summed giving the complete result. This way we ensure that each thread will explore different possible coins.

2.3 CUDA

For CUDA, we initialized the **hash_size** with 0 since the hashes are not being stored. Instead, each thread calculates a hash, and if it meets the condition of ending with 8 hexadecimal zeros, the corresponding **coin** is saved in a shared list accessible by all threads. This shared list is defined in "deti_coins_cuda_search.h" and has its first index set to 1, indicating the next available position. When a thread finds a valid coin, it uses **atomicAdd** to increment this index by 13 (which represents the size of the coin), updating the next available position in the list.

To generate a coin, each thread initializes its own unique candidate coin using the "v1" and "v2" values passed to them in the parameters. The thread modifies a specific part of the coin by adding its thread number (n) to an integer in the coin, while ensuring the values remain within the ASCII range. Each thread generates and evaluates 95 potential coins before completing its execution.

To ensure that the threads explore different coin possibilities in successive runs, 95 (or 0x5E in hexadecimal) is added to the "v1" value for each iteration. If it overflows, we add 1 to the v2 value. The "next_value_to_try_ascii" function ensures that the values remain within the valid ASCII range.

2.4 Web Assembly

In WebAssembly, we started with the standard **deti_coins_cpu_search**. We initialized the coin using the data from **deti_coin_example.txt**. Then, instead of directly using the **md5_cpu()** function, we created a modified version, replacing **#define DATA(idx) data[idx]** with **coin[idx]** to work with the coin that we initialized before.

To validate the coin, we checked if the resulting hash had its last 32 bits equal to 0 (i.e., **hash[3]==0**). Since the coin cannot be saved, we printed it upon success. To generate a new coin, we used the **next_value_to_try** function to modify one of the integers in the coin, incrementing the next one if it overflowed.

2.5 special

Here, we copied the code of "deti_coins_cpu_search" and we changed the initialization to "DETI coin lorem ipsum: ". Then, we added 0s (instead of spaces) until we reached the 51 characters since the last character has to be a "\n". Lastly, we added "1" as if we were adding to a regular integer number and we eventualy found the following coin:

DETI coin lorem ipsum: 00000000000000000000925941103

3 Results

After implementing the search functions, we conducted experiments to evaluate the performance of each method. We measured the time taken to find a valid DETI coin, the number of attempts made, and the number of coins found. The experiments were conducted on two different systems: one with

a AMD Ryzen 5 4600H and NVIDIA GeForce RTX 2060 and another with a intel core i7-11370H and a NVIDIA GeForce RTX 3060.

Method	Time (h)	N of attempts
AVX	1	$9.07 * 10^{10}$
AVX2	1	$1.51 * 10^{11}$
OpenMP(avx2)	1	$1.17 * 10^{12}$
CUDA	1	$4.06 * 10^{13}$

Table 1: Results of the experiments on a AMD Ryzen 5 4600H and NVIDIA GeForce RTX 2060

Method	Time (h)	N of attempts
AVX	1	$9.57 * 10^{10}$
AVX2	1	$1.50 * 10^{11}$
OpenMP(avx2)	1	$7.93 * 10^{11}$
CUDA	1	$3.89 * 10^{13}$

Table 2: Results of the experiments on a intel core i7-11370H and a NVIDIA GeForce RTX 3060

For WebAssembly only one test was made and the result was the following:

2 DETI coins found in 1000000000 attempts (expected 0.23 coins) in 60.086s

4 Conclusions

In this assignment, we explored various parallel computing methods to implement a search function for generating valid DETI coins. Each method leveraged different technologies, ranging from vectorization with AVX/AVX2 to multithreading with OpenMP and GPU programming with CUDA. The performance of these methods varied significantly, reflecting the strengths and limitations of the underlying architectures.

From our results, CUDA demonstrated the highest efficiency, significantly outperforming the other approaches in terms of the number of attempts per unit time. This is attributed to the massive parallelism provided by GPU threads, which is ideal for the highly parallel nature of this search problem. OpenMP, when combined with AVX2, also delivered notable performance improvements over standalone AVX2 by utilizing multithreading on the CPU

it delivered about double the performance in some cases. Lastly, AVX and AVX2 provided a solid baseline, with AVX2 benefiting from the increased lane count for processing data.

The key takeaway is that leveraging hardware-specific optimizations can yield dramatic improvements in computational efficiency. CUDA, with its specialized GPU capabilities, is the best choice for this problem, especially when scalability and performance are the primary objectives. OpenMP provides a strong alternative for systems where GPU resources are unavailable.

Overall, this assignment reinforced the importance of understanding and effectively utilizing the strengths of different parallel computing frameworks. Each method showcased unique advantages that can be leveraged based on specific hardware and performance requirements.