Rodinia Particle Filter analysis and optimization

[DD2360] Applied GPU Programming

Expected grade: A

1st Martin Forslund
Project Group 9
uz6@kth.se

2nd Valeria Grotto
Project Group 9
vgrotto@kth.se

3rd Ruimin Shi
Project Group 9
ruimins@kth.se

4th Bowen Tian
Project Group 9
bowent@kth.se

CONTRIBUTIONS

Martin Forslund: Analyzing using different amount of threads per block in the executions in both the float and naive version of Particle Filter. Wrote primarily parts of the methodology, experimental setup and discussion and conclusion.
Valeria Grotto: Implementation and analysis of random number generation with cuRAND API of the CUDA Toolkit. Implementation and analysis of CUDA streams in the Naive version of Particle Filter. I focused on the introduction, part of the methodology, result and future work sections.
Ruimin Shi: Modification and analysis of CUDA stream in naive version and implementation in float version. Exploration of the binary search in PF. I wrote part of the methodology, experimental setup and results with partners.
Bowen Tian: Explore and analyze the implementation of Unified Memory and prefetch operations in naive and float versions. Complete the implementation of Shared Memory in the float version. I focus on the methodology and result sections.

I. INTRODUCTION

In this report we summarize the findings of our analysis of the Particle Filter (PF) application in the Rodinia suite by the University of Virginia. The Rodinia suite offers a wide range of benchmarks for heterogeneous computing [3]. Among those we choose the Particle Filter application, which consists in a statistical estimator of the state of a target given noisy observations. In particular, the implementation by Matthew A. Goodrum et al. [1], focuses on the tracking of leukocytes (white blood cells).

The Rodinia suite provides two different implementations of the PF, a *Naïve version* (present in the file `ex_particle_CUDA_naive_seq.cu`) with only one CUDA kernel and a *CUDA version* (implemented in the file `ex_particle_CUDA_float_seq.cu`), more efficient and with multiple kernels to carry out different operations.

II. METHODOLOGY

At the beginning of the project, the research goals and research methods needed to be confirmed. The team was looking forward to finding ways to optimize the ParticleFilter in the Rodinia suite and exploring its scaling performance under different iterations.

A. Analysis of original implementation

To improve the PF application, the first step is to analyze the source code. PF application should process six steps: initializing the particles, prediction, measurement update according to importance weights, normalization, resampling and, state estimation. The initialization and state estimation are running at the host because they have a high requirement for sequential execution. In *Naïve version*, only the resampling step is executed in the GPU device, and every pixel uses one thread of sequential traversal to find the weights index. In *float version*, prediction, measurement update, and normalization steps are also executed in GPU devices. Due to the access to global memory meanwhile, in these kernels, they need synchronization to avoid conflict.

Using profiling tools such as `nvprof` and `Nvidia Nsight` mentioned in section III, the distribution of time spent executing different functions can be analyzed. Improving functions that consume more time can significantly improve the overall application runtime. Moreover, analyzing the trace, some warnings were a low Memcpy and Kernel Overlap (0%), a low Kernel Concurrency (0%) and a low Memcpy Overlap (0%); all of these leave room for improvement by exploiting CUDA Streams.

B. CUDA Streams and pinned memory

A way to improve overlapping is to use CUDA streams, a feature in CUDA programming that helps with achieving concurrency of different operations on the GPU. They allow for parallel execution of kernel and memory operation, which leads to better GPU utilization. This is possible through asynchronous execution, where the CPU queues up operations in a stream without waiting for each operation to complete. This also allows to reduce the idle time on the GPU and improves overall throughput.

Pinned memory is another critical aspect of optimizing performance in CUDA. It refers to the host memory that has been locked to prevent it from being paged out. When data is being transferred from pageable memory, it often gives additional overhead, as it might need to get copied to pinned memory internally before it gets transferred to the GPU. By using pinned memory, these overheads are eliminated which lead to faster data transfers. The use of pinned memory needs to be carefully implemented, as it cannot be swapped out of disk, it can limit the amount of RAM that is available for other processes, which leads to potential bottlenecks.

C. Shared memory

Shared memory is the memory that is available on the GPU and can be shared between threads that are located in the same block, which offers more unique performance advantages. Since shared memory is on the GPU, it is it faster than global memory in terms of access speed. It becomes the most useful in implementations where the data is getting reused, such as where multiple threads need to read or write the same data.

When shared memory is limited in size, to use it efficiently means to maximize the performance. If shared memory gets overused, it can lead to issues such as bank conflicts, where multiple threads try to access the same memory location, causing a performance decrease.

Optimization about shared memory was primarily focused on the *find_index_kernel* within the Float version of ParticleFilter. This kernel is designed to locate an index within a cumulative distribution function (CDF) array based on a randomly generated value (u). The key improvement involves leveraging shared memory to cache a segment of the CDF array, to enhance overall performance by minimizing global memory access latency.

At the onset of the kernel, shared memory is dynamically allocated, and *_syncthreads()* is employed to synchronize threads.

It is essential to notice that the CDF contains a substantial amount of data, and the shared memory size may not suffice for a one-time allocation of the entire dataset. Consequently, iterative batch processing was implemented, utilizing *thread_per_block* as the iterative batch size. Beginning with the initial set of data, the team traverses the entirety of the CDF data through multiple iterations, searching for matching items and updating the index. This iterative approach is necessary due to the limited size of shared memory.

D. Unified memory and Prefetch

For the particle filter versions of Float and Naive, the unified memory in CUDA was used for object tracking. Unified memory in CUDA allows transparent memory access between the CPU and GPU, simplifying memory management. The main modifications in the code involve replacing explicit CUDA memory allocation (cudaMalloc) and copies (cudaMemcpy) with unified memory allocation (cudaMallocManaged).

In theory, this should provide a more seamless integration of CPU and GPU memory, reducing the need for manual memory transfers. We expected that it could result in performance gains due to reduced data transfer overhead between CPU and GPU; evaluation of execution times will be outlined in subsequent sections.

For the provided code, the unified memory allocation was not employed for all the data within the ParticleFilter. Specifically, data exclusively utilized by the GPU retained its original explicit memory allocation. Conversely, for data involved in both GPU and CPU computations, the unified memory was chosen for allocation.

Besides, in the unified memory model, the explicit prefetching with cudaMemPrefetchAsync was used to move data between CPU and GPU memory before it was used. This should enable the CUDA runtime to predict data needs and initiate transfers asynchronously.

E. Threads per Block

When programming in CUDA, threads per block are a parameter that impacts the performance of the CUDA application. The purpose of it tell how threads in CUDA are grouped within a block. The blocks can have a lot of threads, but it is required that it is a multiple of 32, because it is the size of a warp, which is NVIDIA's basic execution unit and usually does not exceed 1024 due to hardware limitations.

The performance aspects looking into when modifying threads per block are the following:

- **Resource Utilization:** Utilizing more threads can lead to better GPU resource usage, including more efficient access to registers and shared memory.
- **Achieved Occupancy:** Modifying the number of threads per block can lead to higher occupancy, resulting in more efficient scheduling and minimized idle threads.

Even if you increase the amount of threads per block, it can become too many threads which limits the available shared memory per thread, which affects the performance for the worse.

F. Binary search algorithm

Resampling is an important part of ParticleFilter, which consumes the most time. In this step, N random numbers uniformly distributed between 0 and 1 are generated and are compared with the cumulative distribution function (CDF) from the normalized weights, calculating the last two steps. Particles with higher cumulative probabilities are more likely to be selected multiple times, while particles with lower probabilities are less likely to be chosen.

$$\begin{aligned} &Resampled\_Particle[i] = Particle[j] \\ &j = min\{k|CDF[k] \leq Random\_Number[i]\} \\ &\text{for i,k} \in \{0, \dots, np\} \end{aligned} \tag{1}$$

So the main process is finding the index that CDF is more than or equal to random numbers. Supposing the PF filter requires N particles, in the original code, each thread executes sequential comparison and linear search, which means the time complexity is $O(N)$ per thread. Since CDF is an ordered array, using streams can lead to severe workload imbalance.

To resolve load imbalances and reduce lookup times, a binary search algorithm can be applied. CDF is an ascending array. Binary search is efficient for sorted arrays and the time complexity is reduced to $O(\log_2 N)$.

Algorithm 1 Binary Search in Resampling Step

```
function BinarySearch(CDF, random_number)
    low ← 0
    high ← length(CDF) − 1
    while low ≤ high do
        mid ← ⌊(low + high)/2⌋
        if CDF[mid] = random_number then
            return mid
        else if CDF[mid] < random_number then
            low ← mid + 1
        else
            high ← mid − 1
        end if
    end while
    return −1
end function
```

G. *Random Number Generation in CUDA*

In the original PF in the Rodinia suite [1], the authors point out that a problem of parallelizing the code is the generation of random numbers on the GPU. In fact, CUDA at the time did not provide a built-in random number generator library; they solved the issue by first sampling a uniform number according to a Linear Congruential Generator (LCG) and then transforming it to a Gaussian sample with the Box-Muller Transform [6]. Nowadays, the CUDA Toolkit provides the NVIDIA CUDA Random Number Generation library (cuRAND) [5]; we analyzed the impact of using the CUDA built in library, with the function `curand_normal_double()` to generate random numbers on the GPU in the PF application.

III. Experimental setup

Before we started on our project, we got to learn about some tools from the homework assignments that helped us to analyze the performance of our CUDA program. Two of these programs were *nvprof* and *Nvidia Nsight*.

A. *Nvprof*

Nvprof is a command-line profiler that is part of NVIDIA's CUDA toolkit, which is designed for profiling CUDA applications to see the performance on NVIDIA GPU's. This tool plays an important role to understand a CUDA application's behavior on a GPU. It gives a great analysis of the performance of the GPU kernels in the application by recording the execution time and assessing the utilization of GPU resources.

B. *NVIDIA Visual Profiler*

NVIDIA Visual Profiler is a software tool provided by NVIDIA for profiling and optimizing CUDA applications running on NVIDIA GPUs. The profiler helps to analyze the performance of their CUDA code and identify areas for improvement. NVIDIA Visual Profiler is used to check the overlapping of streams.

C. *NVIDIA Tesla T4 GPU*

The NVIDIA Tesla T4 GPU was our choice of GPU for this project. It is part of NVIDIA's Tesla series, which is designed primarily for data centers and deep learning applications. It is available for free for anyone through Google Colab, which we used for the project. Since we were collaborating with each others, it is more convenient to run it from other computers. In addition, since all of us would be using the same GPU, it would be easier for this project as we want to see how we can optimize this program based on one GPU as the behavior can be different from different GPUs.

The NVIDIA Tesla T4 key advantages is that it is using GDDR6 memory with 256-bit memory bus, which makes it a good choice in handling large datasets. The memory speed is running at 1250 MHz. It has a clock-rate of 1590 MHz and utilizes 2560 CUDA Cores [2]. These specifications are better than anything that we own ourselves and also makes it more interesting for this project to use a dedicated GPU for computations like in this benchmark.

D. *Setup steps*

The setup for this project consists of the use of the GPU provided by Google Colab, a Tesla T4 based on the Turing architecture with compute capability 7.5. Therefore, all code written was compiled using nvcc with the flag `-arch=`*sm*_75.

The Particle Filter application has four input arguments: the coordinate of the initial particle (-x, -y), the number of frames (-z) the number of particles (-np). We used a default initial position (128, 128) and number of frames (10), and we focused on a varying number of particles.

```
./particlefilter_naive -x 128 -y 128 -z 10
-np 1000000
./particlefilter_float -x 128 -y 128 -z 10
-np 1000000
```

To realize streams in this application, another argument is added as -nstream. To run the stream version of PF application use:

```
./particlefilter_naive_stream -x 128 -y 128
-z 10 -np 1000000 -nstream 8
```

IV. Results

This section critically analyzes the performance results obtained from various optimizations applied to the Particle Filter (PF) project. The primary focus was to enhance the performance by tweaking various parameters such as threads per block, implementing CUDA streams, and utilizing Unified and Shared Memory.

A. *Changes in Threads per Block*

One of the first observations we examined when looking into the code was changing the Threads per block. When experimenting, we made 10 iterations of the code which goes from 1000 to 100000, as it would be more interesting to see if the performance is just for a set of particles or if the quantity does not matter. We kept using the default values for the input parameters $x = 128$, $y = 128$, $z = 10$.

From experimenting, we got different results when we tried to change the values in both the original naive and float version. In the naive version, the default threads per block was 128. While I was first experimenting with increasing the amount of threads per block to greater than 128, it would perform worse. But it became more interesting when I was decreasing to lower amount of threads per block instead. As observed in Figure 1, we get a faster execution time when we reduced the amount of threads per block.

For the float version of ParticleFilter, it uses 512 threads per block which the first idea was to change it to 1024 threads per block. The program could not run when threads per block was changed to 1024. But when reducing the amount of threads to 256 and 128, the executes faster.

The interesting part here is that all of these benchmarks seem to be almost identical in the graph, which we can observe in Figure 2, and also it got interesting that 256 threads per block seems to be the best choice between these three options. In another run between those two, with the Nparticles set to 100000, we can see in figure 3 that it is not as huge difference, but we can see which part that is improving the most. The find_index_kernel seems to have the most impact of reduced time. But in comparison to the liklihood_kernel and normalize_weights_kernel, we can see that 512 threads/block has a small advantage, but since most of the program is made in the find_index_kernel, it makes it more optimal on 256 threads/block.

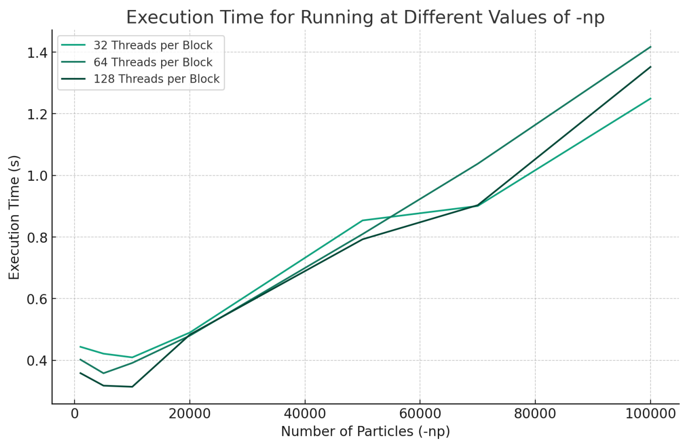

Fig. 1. PF with changed Threads per Block values (naive)

B. Streams and binary search algorithm

Figure 5 and figure 4 show the overlapping between communication and find_index_kernel, using linear and binary search respectively. They are tested with the same parameters $x = 128$, $y = 128$, $z = 10$, and $np = 100000$. 8 streams are used, which means the segment size per stream is 12500. From figure 4, the workload imbalance can be seen. This is because the size of the arrays to be processed by each stream is divided

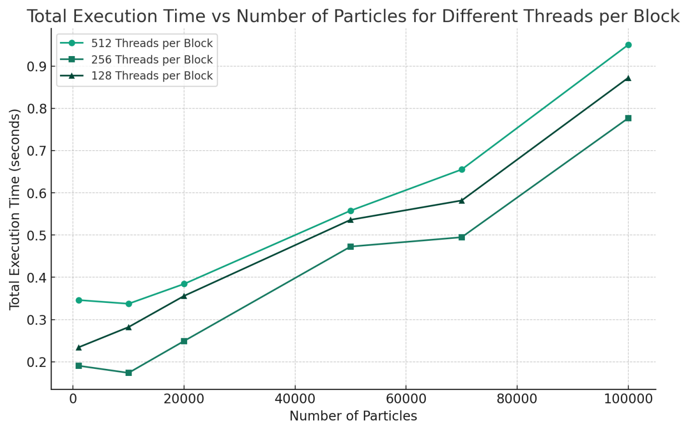

Fig. 2. PF with changed Threads per Block values (float)

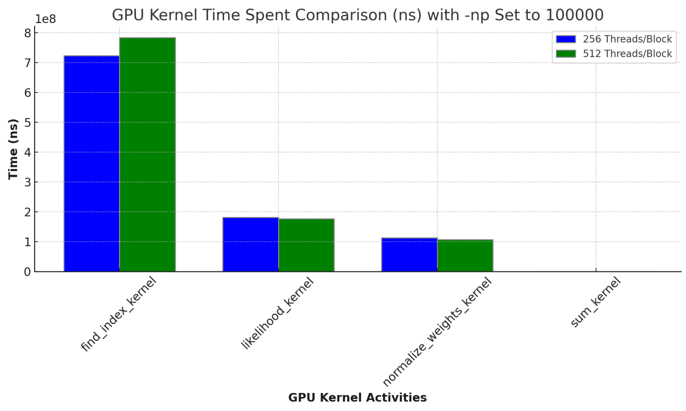

Fig. 3. PF (float) utilization for Nparticles set to 100000. Data from nvprof.

equally, using Eq2. For example, the first stream compares u from 0 to segment_size with the CDF from 0 to Nparticle respectively and jumps out of the loop when it looks for a CDF larger than the random number. The CDF and random number arrays(represented by u) are sorted in ascending order, so the first stream will terminate the loops earlier than the others. After resampling it is necessary to synchronize all threads and copy the computation results back to the host in order to set up the estimation process, so the slowest running threads limit the speed of the whole GPU kernel. This bottleneck leads to no significant increase in the efficiency of the PF after the original stream is implemented. This imbalance issues can be dealt with binary search seeing in figure 5.

$$segment\_size = \left\lceil \frac{Nparticles}{Nstreams} \right\rceil \tag{2}$$

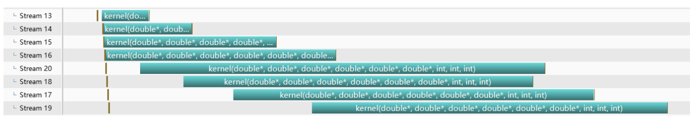
Stream 13
Stream 14
Stream 15
Stream 16
Stream 20
Stream 18
Stream 17
Stream 19
kernel(do...
kernel(double*, doub...
kernel(double*, double*, double*, double*, ...
kernel(double*, double*, double*, double*, double*, double...
kernel(double*, double*, double*, double*, double*, double*, int, int, int)
kernel(double*, double*, double*, double*, double*, double*, int, int, int)
kernel(double*, double*, double*, double*, double*, double*, int, int, int)
kernel(double*, double*, double*, double*, double*, double*, int, int, int)

Fig. 4. Naive PF trace with CUDA streams using linear search

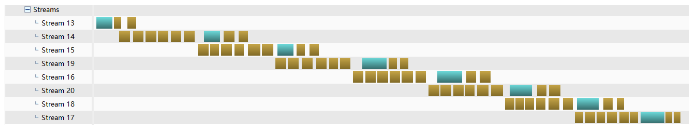

Fig. 5. Naive PF trace with CUDA streams using binary search

Figure 6 shows the variation of execution time with the number of streams, which are tested in $np = 10000$. The performance reaches to peak when using 2 streams in the binary search version and 16 streams in the linear search version. This means that for binary search when the number of streams is equal to 2, the communication time between the host and the device, as well as the computation time of the kernel, can be overlapped.

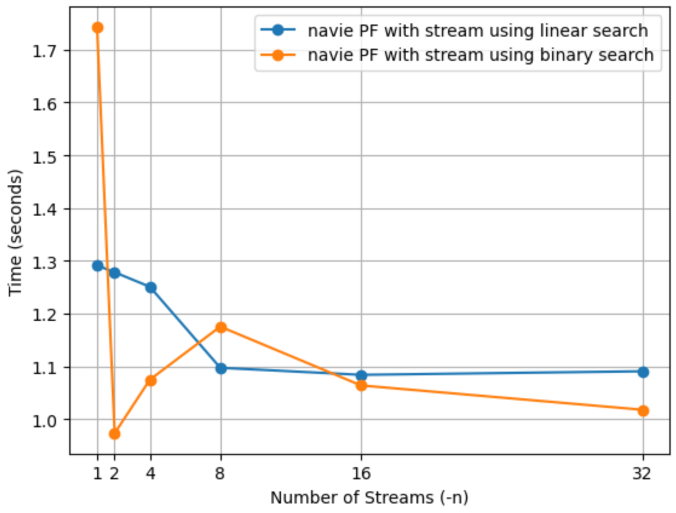

Fig. 6. Filter execution time changes with stream number

Figure 7 shows the execution time changing with the number of particles in different types of optimization. As the increasing of particle numbers, the float version of PF with streams and binary search speeds up significantly. When the problem input size is less than 20000, the float version does not perform well, and the navie version with the stream has the best efficiency. That is because the cost of one-time communication between host and device in float version is more than navie version. And the binary search is more suit for large scale of application.

C. Unified Memory and Shared Memory

In the related tests of unified memory and shared memory, the default values of the input parameters were also set to x = 128, y = 128, z = 10, and 10 iterations with the number of particles from 1000 to 100000 were also used.

In the naive and float versions of particle filter, the comparison of the execution time optimized using unified memory and prefetch with the original execution time can be obtained from figure 8 and figure 9 respectively. It can be seen from the results of optimization using Unified memory that there is

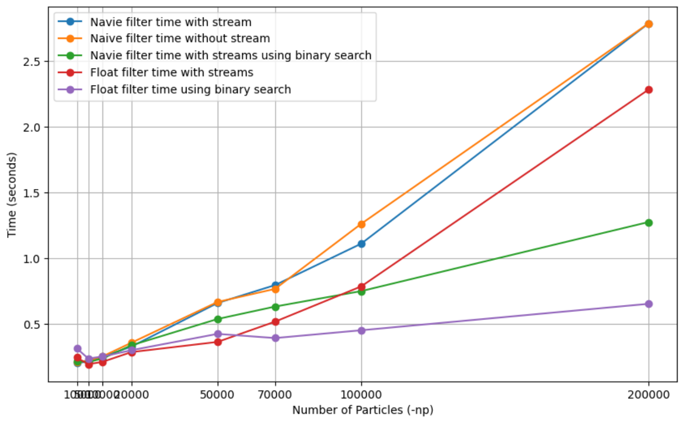

Fig. 7. Comparison of different optimization

a certain performance improvement for both the naive version and the float version, but the effect is not significant.

Analyzing the reasons for poor optimization results, since unified memory eliminates the need for explicit data movement, unified memory will have a good optimization effect in application scenarios where memory bandwidth is the limiting factor. However, from the visualization results of nvprof, it is known that the execution time of the program is mainly determined by calculation rather than memory transfer. GPU programs are highly limited by calculation, so the advantages of unified memory may not be effectively utilized.

The difference between the prefetch optimization effects in the float version and the naive version requires further discussion. While unified memory is designed to automatically migrate data between the CPU and GPU as needed, the prefetching strategies employed by the CUDA runtime system may not always perfectly predict data usage patterns. Therefore, the difference caused by manually adjusting the prefetched data is further explored in the figure10: the adjustment of the prefetched data has a greater impact on the execution time, and prefetching all unified memory data at the same time seems to reduce performance.

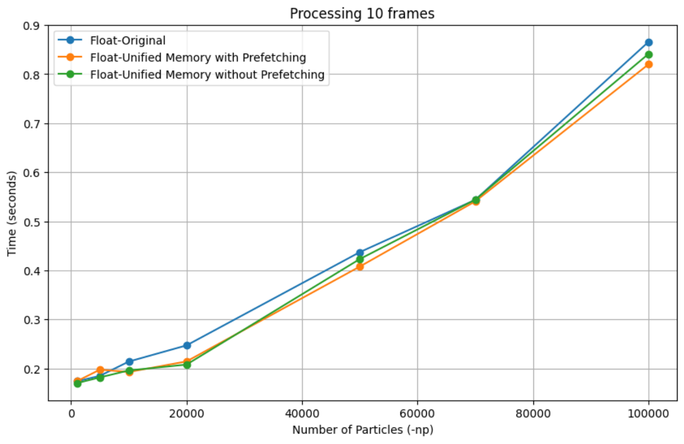

Fig. 8. PF with Unified Memory(float)

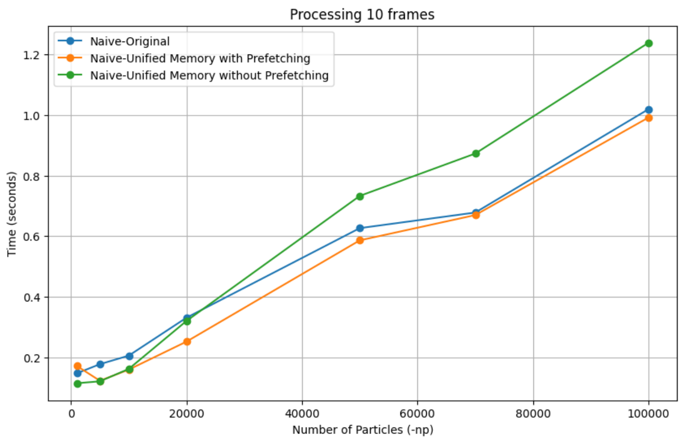

Fig. 9. PF with Unified Memory(naive)

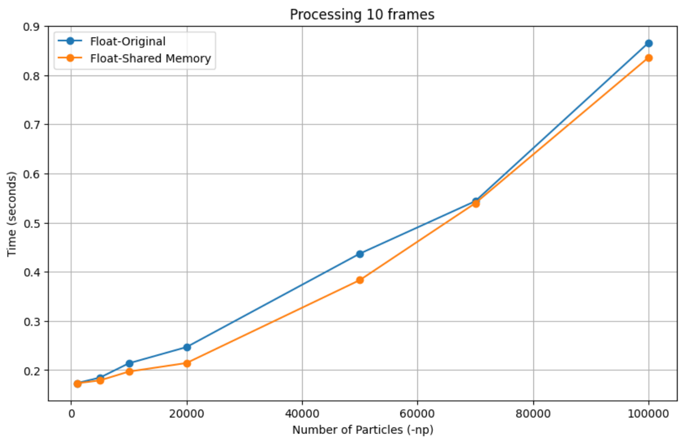

Fig. 11. PF with Shared Memory

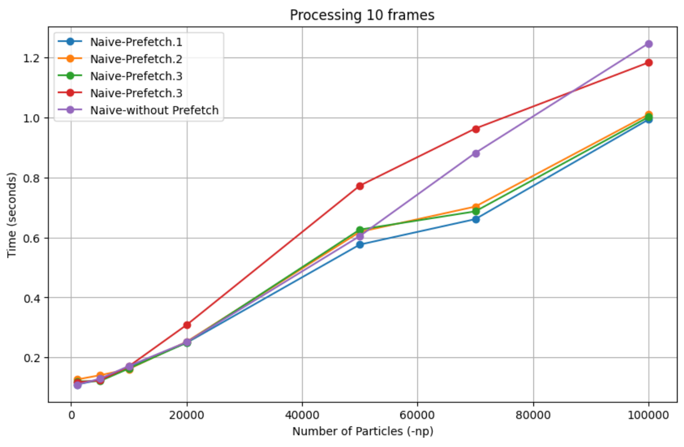

Fig. 10. Manual adjustion on Prefetch

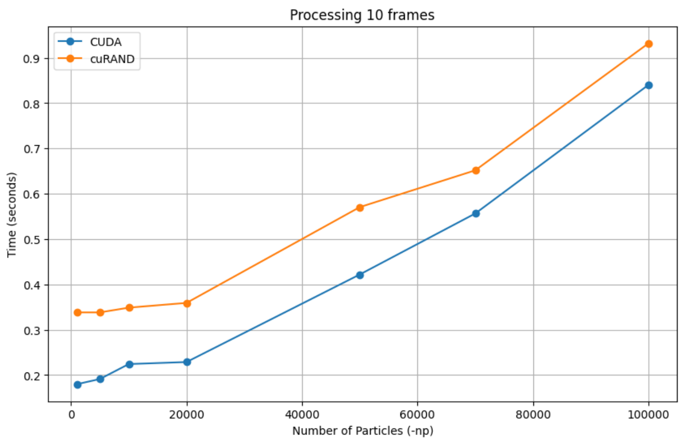

Fig. 12. Processing times: cuRAND vs Rodinia CUDA

While the results of optimization using shared memory are shown in the figure 11 Similar to unified memory, shared memory also introduces overhead, potentially offsetting its benefits in certain scenarios. In cases where the kernel is primarily compute-bound rather than memory-bound, the advantages of shared memory may not be as apparent, especially when data movement between the CPU and GPU is not a significant issue.

Furthermore, the *find_index_kernel* algorithm may not be well-suited for shared memory optimization. This is because the kernel operation involves a sequential search for data in the Cumulative Distribution Function (CDF), where the data in the CDF is arranged in ascending order. Essentially, the kernel faces limitations in handling sequential operations, the advantages of utilizing shared memory are constrained.

D. Random Number Generation in CUDA

A minor modification to the original PF it has been to use the *cuRAND API* provided by Nvidia instead of the original. However, in figure 12 is possible to see that the implementation of the PF with the cuRAND API worsened the speed of the program. This is probably due to the fact that the initial implementation already exploited the GPU parallelism. More in detail, the cuRAND API utilizes the Box-Muller transform as the original PF did, however the one in the Rodinia suite seems more lightweight, hence faster. The Box-Muller transform, even though is computationally expensive with the computation of the cosine and sine, does not incur in branching [4], for this reason it is suitable for the use on the GPU. We still recommend to use the built-in CUDA library in order for the code to be more robust and always updated.

V. Discussion and Conclusion

In this study, we examined and analyzed the performance results of various optimizations applied to the CUDA version of the Rodinia benchmark Particle Filter (PF). The aim was to enhance the computational efficiency through different methods such as modifications in threads per block, adoption of CUDA streams and utilization of Unified and Shared memory. This section discusses the results that was given, outlines the limitations and discusses future work for this benchmark.

A. Performance Results and Optimizations

The optimizations that were made revealed many different insights into CUDA programming and its impact on the PF application. Adjusting the threads per block value showed that the performance can be very different depending on which value you choose for it.
We have shown that the implementation of CUDA streams by itself is not enough to improve the overall speed of the program. It was necessary to modify the bottleneck due to the *linear search* with a *binary search* to notice a significant improvement. This underscores the importance of workload balance and efficiency in parallel processing.

B. Future work

Some of the things to consider to be experiment with are the following:

- **Advanced Algorithmic Techniques:** Investigating alternative or more advanced algorithms that might be better suited to GPU optimization. For instance, improving the tree reductions so that they do not serially add block sums, and finding a way to remove the need for global synchronization related to the reduction would provide additional speedup. Another possible improvement could be to explore the use of half precision.
- **Adaptive Optimization Strategies:** Developing methods for dynamically adjusting GPU resources like threads per block and the number of streams according to real-time performance metrics. This would help the optimizations also to be working for all GPUs and not examine just a single type of GPU.
- **Hardware-Specific Tuning:** Tailoring optimizations to specific GPU architectures, potentially exploring the capabilities of newer and more advanced GPUs.

References

[1] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron, "Parallelization of Particle Filter Algorithms," in *Computer Architecture*, A. L. Varbanescu, A. Molnos, and R. Van Nieuwpoort, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6161, pp. 139–149, doi: 10.1007/978-3-642-24322-6_12.

[2] NVIDIA Tesla T4 specs. Link: https://www.techpowerup.com/gpu-specs/tesla-t4.c3316

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44-54, doi: 10.1109/IISWC.2009.5306797.

[4] Howes, Lee and Thomas, David. (2007). Efficient Random Number Generation and Application Using CUDA.

[5] Random Number Generation on NVIDIA GPUs, NVIDIA Developer, Link: https://developer.nvidia.com/curand.

[6] Howes, Lee and Thomas, David. Chapter 37. Efficient Random Number Generation and Application Using CUDA, Link: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-37-efficient-random-number-generation-and-application.