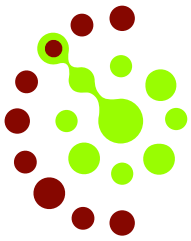

IPBeja

Instituto Politécnico de Beja

School of Technology and Management

Master's Degree in Computer Engineering and Internet of Things

High-Performance Computing Report: Kernel Density Estimation Optimization using OpenMP

Pedro Henrique Nunes Souza

Beja, February 15, 2025

INSTITUTO POLITÉCNICO DE BEJA

School of Technology and Management

Master's Degree in Computer Engineering and Internet of Things

High-Performance Computing Report: Kernel Density Estimation Optimization using OpenMP

Pedro Henrique Nunes Souza

Advisor :

José Jasnau Caeiro, IPBeja

High-Performance Computing

2025

Contents

Contents **i**

1 Introduction **1**

1.0.1 Work Undertaken and Document Structure 1

2 Problem Description **3**

2.0.1 Mathematical Formulation 3

2.0.2 Computational Challenges 4

2.0.3 Implementation of Parallel Solutions 4

2.0.4 Performance Evaluation 5

3 Experimental Part **6**

3.0.1 How the Optimization Was Performed with OpenMP 6

4 Conclusions **10**

4.0.1 Summary of the Results 10

4.0.2 Future Work 10

4.0.3 Final Considerations 10

Bibliography **11**

Chapter 1

Introduction

Kernel Density Estimation (KDE) is used to smoothly estimate data distributions and is applied in several fields, such as machine learning, image processing, and computational statistics. KDE has a high computational cost, with a complexity of $\mathcal{O}(n^2)$, making it unfeasible for applications with large volumes of data or for real-time processing. To address this problem, this work proposes optimizing KDE using parallel computing techniques, employing OpenMP to exploit parallelism on multicore CPUs and offloading with OpenMP Target for execution on GPUs [Węg18; LW17].

The goal is to compare the performance of sequential, CPU-parallel, and GPU-parallel implementations, evaluating the execution time and the speedup obtained through parallel computing.

1.0.1 Work Undertaken and Document Structure

In this project, three versions of the KDE algorithm were implemented and compared:

- **Sequential Version**: Does not include parallelism and serves as a reference for performance comparison.
- **Multicore Parallel Version**: Uses OpenMP to distribute calculations among multiple CPU threads, reducing execution time.
- **GPU Parallel Version**: Employs OpenMP Offloading to transfer the calculations to the GPU, taking advantage of a large number of processing cores.

An automated compilation system was developed using CMake, which allowed the organization and execution of the codes.

This report is structured as follows:

- **Section 2 - Problem Description**: Explains KDE and its mathematical formula.

- **Section 3 - Experimental Part**: Presents the code implementations, the parallelization methods used, and the structure of the build system (CMake).
- **Section 4 - Conclusion**: Presents the results obtained and suggests future improvements.

Chapter 2

Problem Description

Unlike histograms, which create discontinuities in the representation of data distribution, KDE smooths the function, allowing for better analysis and decision-making. This method is used in several fields, such as [Węg18]:

- **Computer Vision and Image Processing**: KDE is used for image segmentation and pattern recognition.
- **Financial Analysis**: KDE is used for risk modeling and forecasting price fluctuations of financial assets.
- **Machine Learning and Artificial Intelligence**: KDE is used in building probabilistic models and identifying data anomalies.

The high computational cost of KDE makes its implementation challenging when dealing with large volumes of data. Its calculation requires comparing each data point with all others, resulting in a complexity of $\mathcal{O}(n^2)$ [LW17]. Consequently, optimizing the KDE calculation through parallel computing techniques is necessary to better utilize the hardware.

2.0.1 Mathematical Formulation

Mathematical definition of KDE:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{2.1}$$

where:

- $\hat{f}(x)$ is the probability density estimate at point x;
- n is the number of points in the dataset;
- h is the bandwidth parameter, which controls the smoothing level of the distribution;

- $K(\cdot)$ is the kernel function, usually chosen to be Gaussian:

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}} \tag{2.2}$$

[Wik25]

The main computational challenge of the Kernel function arises because it must be evaluated for all pairs of points in the dataset, wich means that the execution grows quadratically with the sdataset size. [LW17]

2.0.2 Computational Challenges

The KDE implementation faces several computational difficulties and setbacks:

- **Computational Complexity**: Since each point must be compared to all others, the computational cost grows exponentially as the number of samples increases.
- **Inefficient Hardware Usage**: Sequential implementations utilize only one CPU core, resulting in high execution times.
- **Memory Bottlenecks**: With large volumes of data, excessive data movement in RAM can cause slowdowns and low efficiency.

[LW17]

To overcome these challenges, three versions of KDE were implemented and compared.

2.0.3 Implementation of Parallel Solutions

1. **Sequential Implementation (Baseline):**
   - Executed on a single CPU core, with no parallelization.
   - Serves as a reference for comparing the efficiency of the optimized versions.
2. **Parallel CPU Implementation with OpenMP:**
   - Uses OpenMP to distribute operations among multiple CPU threads.
   - Allows for parallelization of the double loop, reducing execution time.
3. **Parallel GPU Implementation with OpenMP Offloading:**
   - Uses OpenMP Target Offloading to transfer calculations to the GPU.
   - The GPU can process a large number of operations simultaneously, providing performance gains.

The parallel implementation was based on three main optimization techniques:

- Use of the `#pragma omp parallel for` directive to distribute operations across CPU cores.
- Use of the `#pragma omp target` directive to offload operations to the GPU.
- Use of the `reduction(+:sum)` clause to prevent race conditions.

2.0.4 Performance Evaluation

To measure the efficiency of each approach, the execution time was used as the main metric. The evaluation was carried out with different dataset sizes, and times were collected using the `omp_get_wtime()` function. The Speedup was calculated using the following formula:

$$S = \frac{T_{sequencial}}{T_{parallel}} \tag{2.3}$$

where:

- $T_{sequencial}$ is the execution time of the non-parallelized version.
- $T_{parallel}$ is the execution time of the optimized (CPU or GPU) version.

After testing, the following aspects were analyzed:

- **Scalability**: How the parallel implementation behaves as the dataset size increases.
- **Efficiency**: Which version offers the best performance in terms of computation time.
- **Impact of Parallelization**: How much real gain was achieved by using multiple cores and a GPU.

Conclusion

Optimizing Kernel Density Estimation using OpenMP allowed a significant reduction in execution time, making it feasible to apply KDE to large datasets efficiently. Parallel implementations take advantage of multicore CPUs and the computational power of GPUs, resulting in substantial speedup and scalability gains.

Chapter 3

Experimental Part

At this stage of the work, the main objective was to optimize and speed up the Kernel Density Estimation (KDE), taking advantage of the capabilities of both processors and GPUs. The chosen method was OpenMP, wich made it possible to incorporate parallelization directives into the C++ code.

3.0.1 How the Optimization Was Performed with OpenMP

The directive `#pragma omp parallel for` was used to run the code on the CPU so that each of the loops could be divided between different threads. To better handle nested loops (when one loop is inside another), the `collapse(2)` option was also used, which helped "merge" the two loops into a single iteration structure. In practice, this reduced synchronization overhead and improved thread utilization.

In addition to the parallel version on the CPU, the `#pragma omp target` directive was applied for offloading to the GPU, which transfers the code and relevant data to the graphics card. To create greater parallelism, the `teams distribute parallel for` directive was employed, where each team executes a portion of the code, and within each team there are multiple threads working together. To map the variables and ensure minimal communication between the CPU and GPU, `map(to: ...)` and `map(from: ...)` were used.

To calculate the speedup, the `reduction(+:sum)` clause was utilized, which instructs OpenMP to let each thread perform its own local accumulation and only combine everything into a global sum at the end. This way, there is no overlap in writing to a single variable.

Resulting Codes

Below is the skeleton of the code for the three implemented versions: 1. The **sequential** version, which runs on a single core. 2. The **multicore** version, which distributes tasks

among multiple CPU cores. 3. The **GPU** version, which aims to take advantage of the massive parallelism of graphics cards.

3.2.1 The Reference: Sequential Version

```
void kde_sequencial(const std::vector<double>&
dados, std::vector<double>& densidade, int n, int h) {
    for (int i = 0; i < n; i++) {
        double soma = 0.0;
        for (int j = 0; j < n; j++) {
            soma += kernel((dados[i] - dados[j]) / h);
        }
        densidade[i] = soma / (n * h);
    }
}
```

This code iterates over all the data twice, summing the contributions of the kernel and dividing by the factor $(n \times h)$. This version was used as the baseline for measuring the performance of the others.

3.2.2 Multicore CPU Version

```
void kde_multicore(const std::vector<double>&
dados, std::vector<double>& densidade, int n, int h) {
    #pragma omp parallel for collapse(2) reduction(+:soma)
    for (int i = 0; i < n; i++) {
        double soma = 0.0;
        for (int j = 0; j < n; j++) {
            soma += kernel((dados[i] - dados[j]) / h);
        }
        densidade[i] = soma / (n * h);
    }
}
```

Here, `#pragma omp parallel for` was added before the loop, and combining it with `collapse(2)` helps the compiler better distribute the workload across threads. The `reduction(+:soma)` clause ensures that the `soma` variable is handled correctly.

3.2.3 Offloading to GPU

```
void kde_gpu(std::vector<double>&
dados, std::vector<double>& densidade, int n, int h) {
```

```
    double* dados_ptr = dados.data();
    double* densidade_ptr = densidade.data();

    #pragma omp target data map(to: dados_ptr[0:n])
    map(from: densidade_ptr[0:n])
    {
        #pragma omp target teams distribute parallel for collapse(2)
        for (int i = 0; i < n; i++) {
            double soma = 0.0;
            for (int j = 0; j < n; j++) {
                soma += kernel((dados_ptr[i] - dados_ptr[j]) / h);
            }
            densidade_ptr[i] = soma / (n * h);
        }
    }
}
```

Here, the first step is to map the `dados` array to the GPU and set it so that the final result, `densidade`, is copied back to the CPU at the end of the block. Inside the `#pragma omp target teams distribute parallel for`, `collapse(2)` is used again to parallelize the two dimensions of the loop.

Tests were conducted with various dataset sizes, focusing on how long each version takes and its speedup.

Metrics Analyzed

The metric used was execution time (in seconds). Then, speedup was calculated by:

$$S = \frac{T_{sequencial}}{T_{paralelo}}.$$

Performance variation was also observed as data size increased (a scalability analysis).

How to Run the Code

1. **Compilation**:

```
cd ~/hpc/build
cmake ../repo
make
```

2. **Execution**:

```
../repo/run_tests.sh
```

3. **Reading the results**:

```
cat results_plot.csv
```

Obtained Results

3.4.1 Time and Speedup Table

Below is a comparative table of the obtained results for data sizes ranging from 64 to 512:

Size	Sequential (s)	Multicore (s)	GPU (s)	Speedup (Multicore)	Speedup (GPU)
64	0.0448066	0.0153747	0.324886	2.9143	0.1379
128	0.0448795	0.0150333	0.297027	2.9853	0.1510
256	0.04557	0.0146112	0.297871	3.1188	0.1529
512	0.0445145	0.0141523	0.292162	3.1453	0.1523

Table 3.1: Execution times and speedup for different implementations.

General Observations

- The multicore version nearly tripled the speed (speedup $\sim 3\times$).
- The GPU version did not achieve the expected result, possibly due to the extra cost of transferring data between the CPU and the GPU.

Final Considerations on the Tests

In summary, parallelization with OpenMP proved very effective for running KDE faster on the CPU. The GPU did not yield the same return, especially because, for smaller datasets, the extra cost of transferring data between the CPU and the GPU is significant. However, the GPU can be better leveraged for much larger data sizes or with other memory optimizations. Each choice has its pros and cons; still, the final code fulfills its purpose of illustrating how to use OpenMP both on the CPU and the GPU.

Chapter 4

Conclusions

4.0.1 Summary of the Results

The tests perfomed in this work indicate that the parallelization of Kernel Density Estimation (KDE) with OpenMP produced good performance results, reducing the algorithm's execution time compared to the sequential version.

The multicore (CPU) implementation with OpenMP obtained an average speedup of 3x, showing that distributing tasks among multiple CPU cores clearly increases computational efficiency. On the other hand, the version using GPU with OpenMP Offloading did not produce the expected results, due to the data transfer overhead between the CPU and the GPU.

4.0.2 Future Work

Although concrete gains have been observed, there are several opportunities to improve this implementation: for example, improving data transfer between the CPU and GPU by exploring strategies such as memory coalescing and adjustments in global memory access to minimize latency. Another option is to invest in other forms of GPU parallelism, such as CUDA or HIP, which can offer finer control over resource allocation.

4.0.3 Final Considerations

The main conclusion of this study is that parallel computing strategies are essential to make KDE viable in scenarios that require high processing rates. The multicore version with OpenMP showed a significant gain, substantially reducing execution time.

On the other hand, the use of GPU faces barriers related to data movement, indicating the need for more specific optimizations.

Bibliography

[LW17] Nicolas Langrené and Xavier Warin. "Fast and stable multivariate kernel density estimation by fast sum updating". In: *arXiv preprint arXiv:1712.00993* (2017). URL: https://arxiv.org/abs/1712.00993 (cit. on pp. 1, 3, 4).

[Węg18] Stanisław Węglarczyk. "Kernel density estimation and its application". In: *ITM Web of Conferences* 23 (2018), p. 00037. DOI: 10.1051/itmconf/20182300037. URL: https://www.researchgate.net/publication/328785939_Kernel_density_estimation_and_its_application (cit. on pp. 1, 3).

[Wik25] Wikipédia. *Estimativa de densidade kernel*. Acessed: 10 feb. 2025. 2025. URL: https://pt.wikipedia.org/wiki/Estimativa_de_densidade_kernel (cit. on p. 4).

Documento elaborado com base no *template for final reports and dissertations (Instituto Politécnico de Beja)*, disponível em https://www.overleaf.com/project/5d936b9ea273390001434a37, Version 1.0, 2023/01/25, Autor: Pedro Henrique Nunes Souza, 26699@stu.ipbeja.pt