Efficient Parsimony Algorithm Using Radix Sort for Enhanced SIRFS Model and Dense Deformable SLAM

Shromm Gaind

August 3, 2024

Why This Project is Worth Doing

Accurately reconstructing 3D scenes from single images is a complex and ill-posed problem in computer vision. The SIRFS model, integrating priors, addresses this challenge but needs computational efficiency for real-time applications. Implementing an efficient parsimony algorithm using Radix Sort, as seen in Morphogenesis and Fluid-v3, can significantly enhance the model's real-time capabilities. Parsimony, achieved by reducing entropy, ensures resulting maps are accurate, smooth, interpretable, and efficient. This approach meets the real-time processing demands of Dense Deformable SLAM systems, handles specularities, and maintains computational feasibility.

The primary goal of this project is to develop an OpenCL implementation of a parsimony algorithm using Radix Sort. Integrating this into the full SIRFS model is beyond the scope of this project, but the foundation laid here will facilitate future integration.

Steps to Implement the Proposal

Objective and Problem Formulation

- Develop an efficient parsimony algorithm using GPU Radix Sort.
- Integrate this within the SIRFS model to enhance performance in generating accurate maps of reflectance, depth, curvature, shading, and illumination.

Algorithm Design and Mechanism

- **Reflectance Map**: Apply a parsimony prior to minimize global entropy of log-reflectance, promoting piecewise constant reflectance values.

- **Depth Map**: Implement a smoothness prior to ensure smooth depth values across the image, maintaining realism.
- **Curvature Map**: Enforce parsimony to produce coherent curvature maps, avoiding noise.
- **Shading Map**: Maintain a balance between fine details and smooth transitions using a smoothness prior.
- **Illumination Model**: Regularize spherical harmonics coefficients to ensure plausible and consistent lighting conditions.

Computation Process for Entropy Reduction and Parsimony

- **Bin Width** W: Determine based on the range of pixel values in x. It defines the range of values that each bin will cover in the histogram.
- **Number of Bins** M: Calculate based on the chosen bin width and the range of pixel values.
- **Histogram Calculation**:
  - For each pixel value x_i, find the corresponding bin by determining the lower and upper fenceposts.
  - Assign weights to the bins based on the pixel value's position relative to the fenceposts.
  - Increment the histogram counts using these weights.
- **Partial Derivatives**: Calculate the partial derivatives of the histogram counts with respect to each pixel value, forming the Jacobian matrix J.
- **Jacobian Matrix** J: An $M \times N$ matrix where M is the number of bins and N is the length of a vector x. It represents the partial derivative of the histogram count with respect to the input values.
- **Gaussian Filter** g: Define to smooth the histogram, promoting parsimony by reducing entropy.
- **Convolution**: Compute the convolution of the histogram with the Gaussian filter.
- **Entropy Gradient** H: Calculate to understand how changes in pixel values affect the histogram.
- **Gradient Multiplication**: Multiply the transpose of the Jacobian J with H to adjust pixel values and decrease entropy, promoting parsimony.

By computing the gradient of entropy with respect to the pixel values, we can adjust the pixel values to decrease entropy, promoting parsimony. This gradient descent approach ensures that the resulting maps are simpler and more interpretable.

Reducing Complexity

By focusing on a sorted subset of pixels, the proposed technique simplifies the computation, reducing the complexity from $O(N^2)$ to $O(N)$. This is achieved by using the sorted sample to guide the adjustment of pixel values, promoting parsimony and ensuring efficient computation.

Implementation of Radix Sort

We plan to implement the Radix Sort algorithm in the following steps:

1. Sort a sample of pixels ($< 1\%$) to find the distribution of values.
2. Have the sorted sample pixels pull each other together.
3. Use the sample set to pull the remaining image pixels towards their values, minimizing pairwise comparisons and ensuring computational efficiency.

This technique is borrowed from the methods used in Morphogenesis and Fluids-v3. The reason for sorting particles in SPH is to keep the data read-write operations well-ordered, as random read-write is very expensive on GPUs.

The specifics of this implementation are detailed in the appendix, but a brief rundown is provided here:

In both Morphogenesis and Fluids-v3, the implementation is done using four kernels and makes use of GPU local memory and recursive sorting.

Implementing Radix Sort provides the following advantages:

- **Scalability:**
  - The algorithm uses a small, fixed-size sample for mutual interaction while the majority (99%) of pixels are unilaterally attracted to the values of the sample pixels. This means that the complexity of updating pixel values is proportional to the number of pixels, $O(N)$. This linear scaling is achieved due to using a small sample to adjust all the other pixel values, reducing the pairwise interactions which typically result in $O(N^2)$ complexity to a manageable subset through sampling.
- **Portability:**
  - Implementing the algorithm in OpenCL ensures that it can be executed on a variety of hardware platforms, including mobile SoCs and NUC GPUs used onboard drones and robots. OpenCL provides a platform-independent API for parallel programming, allowing the same codebase to be executed on different devices without modification. This makes our algorithm versatile and adaptable to various real-time computer vision applications, ensuring it can be deployed across different types of hardware commonly used in robotics and embedded systems.