2D Poisson Solvers for Gradient Domain HDR Tonemapping

Runfeng Li
Brown University
{runfeng_li}@brown.edu

1. Introduction

Cameras capture photon energy using their sensors, encoding it as radiance at each pixel. However, while camera sensors detect a broad spectrum of radiance, the images are stored within a limited range of 0 to 255. This limitation leads to compromises in color representation and detail. To address this issue, the Gradient Domain High Dynamic Range (HDR) Compression method [1], inspired by the human visual system, proposes a solution. This method involves reducing pixel gradients at various resolutions and reconstructing images using the divergence of these attenuated gradients. This process, which requires solving a Poisson equation, enables more accurate tonemapping of real-world radiance into digital format compared to global Gamma compression, thereby preserving detail across a wide dynamic range.

This project explores various optimization methods to iteratively solve the 2D Poisson equation using CUDA and C++. These methods include single-grid techniques such as Jacobi, Gauss-Seidel with red-black reordering, and successive over-relaxation, as well as multigrid approaches like V-cycle, W-cycle, and Full multigrid. By combining a parallelized Poisson solver with gradient attenuation using LibTorch, we significantly enhance the efficiency of the gradient domain HDR compression algorithm. The code is available at https://github.com/ranrandy/hdrc.

2. Iterative Poisson Solvers

After rescaling the per-pixel graidents from varying resolutions, we obtain the attenuated gradients, G_x, G_y, in the x and y directions. We then use the divergence $\text{div}G = \frac{\partial G_x}{\partial x} + \frac{\partial G_y}{\partial y}$ to recover the tonemapped image I by solving the Poisson equation:

$$\nabla^2 I = \text{div}G \tag{1}$$

Numerically, we approximate the left hand side as

$$\begin{aligned}\nabla^2\text{I(x, y)} \approx \text{I(x, y-1)} + \text{I(x-1, y)} + \text{I(x+1, y)} + \text{I(x, y+1)}\\ - 4\text{I(x, y)},\end{aligned} \tag{2}$$

and we treat the right hand side, $\text{div}G$, as a constant across the image plane. Since the solver is applied to recover an image, we implement Neumann boundary conditions $\nabla I \cdot \mathbf{n} = 0$. Thus, whenever we require pixel values outside of the image, we assign the value from the closest pixel within the image.

2.1. Single-Grid Solvers

Single-grid solvers are the simplest form of iterative solvers, including methods like Jacobi, Gauss-Seidel and successive over relaxation (SOR). By flatenning the pixels into a 1D array, solving the Poisson equation becomes equivalent to solving a linear system $\mathbf{Ax} = \mathbf{b}$. To iteratively solve this linear system, we can use different preconditioners P to split the equation into $\mathbf{Px} = (\mathbf{P} - \mathbf{A})\mathbf{x} + \mathbf{b}$, then update the value of $\mathbf{x}$ using:

$$\mathbf{P}\mathbf{x}^{(k+1)} = (\mathbf{P} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b} \tag{3}$$

Different preconditioners P lead to different iterative strategies. Let L, D, U be the lower triangular, diagonal and upper triangular part of $\mathbf{A}$, respectively. We then derive the following methods:

2.1.1 Jacobi

In the Jacobi method, we set $\mathbf{P}$ as $\mathbf{D}$.

$$\mathbf{D}\mathbf{x}^{(k+1)} = -(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} + \mathbf{b} \tag{4}$$

Then numerically, the update formula is:

$$\begin{aligned}\text{I(x,y)}^{(k+1)} =& \frac{1}{4}\big[\text{I(x, y-1)}^{(k)} + \text{I(x-1, y)}^{(k)} + \text{I(x+1, y)}^{(k)}\\ &+ \text{I(x, y+1)}^{(k)} - \text{div}G(x, y)\big]\end{aligned} \tag{5}$$

2.1.2 Gauss-Seidel

In the Gauss-Seidel method, we set $\mathbf{P}$ as $\mathbf{D} + \mathbf{L}$.

$$(\mathbf{D} + \mathbf{L})\mathbf{x}^{(k+1)} = -\mathbf{U}\mathbf{x}^{(k)} + \mathbf{b} \tag{6}$$

Then numerically, the update formula is:

$$\begin{aligned}\text{I(x,y)}^{(k+1)} =& \frac{1}{4}\big[\text{I(x, y-1)}^{(k+1)} + \text{I(x-1, y)}^{(k+1)} + \text{I(x+1, y)}^{(k)} \\ &+ \text{I(x, y+1)}^{(k)} - \text{div}G(x,y)\big] \end{aligned} \tag{7}$$

2.1.3 Succesive Over Relaxation (SOR)

In the SOR method, we set $\mathbf{P}$ as $\mathbf{D} + w\mathbf{L}$.

$$(\mathbf{D} + w\mathbf{L})\mathbf{x}^{(k+1)} = -\mathbf{U}\mathbf{x}^{(k)} + \mathbf{b} \tag{8}$$

Then numerically, the update formula is:

$$\begin{aligned}\text{I(x,y)}^{(k+1)} =& (1-\omega)\text{I(x,y)}^{(k)} + \frac{\omega}{4}\big[\text{I(x, y-1)}^{(k+1)} \\ &+ \text{I(x-1, y)}^{(k+1)} + \text{I(x+1, y)}^{(k)} \\ &+ \text{I(x, y+1)}^{(k)} - \text{div}G(x,y)\big], \end{aligned} \tag{9}$$

where $\omega \in [1, 2)$. If $\omega = 1$, SOR reverts to the Gauss-Seidel method.

Instead of delving into the mathematical reasons behind the convergence and speed of these solvers, we focus on optimization techniques to accelerate these iterative solvers.

2.2. Single-Grid Paralleliztion

Given the nature of using results from the previous iteration to update the next, the Jacobi method can be easily parallelized as each pixel can be updated independently. However, this is not the case for the Gauss-Seidel solvers. To enhance the parallelizability of Gauss-Seidel solvers, we apply red-black reordering.

2.2.1 Red-Black Reordering

The core idea of red-black reordering is to divide the 2D image plane into a checkerboard pattern of red and black pixels, where red pixels only have black neighbors and vice versa. This allows for the separate and simultaneous updating of all red and black values. Specifically, we define red pixels if $i + j \mod 2$ is 0, and black pixels if $i + j \mod 2$ is 1 (Figure 1).

For CUDA implementation, we design two custom CUDA kernels, one for red and one for black iterations. In the red kernel, we update the value if $i + j \mod 2$ is 0; otherwise, we return. The opposite applies for the black kernel.

2.2.2 Optimized Red-Black Reordering

The initial red-black implementation is not optimal; half of the threads remain idle, and it does not exploit faster memory access for neighboring threads. To improve memory access speed and thread utilization, we construct separate red and black "images" before any iteration. We use the black image to update the red image and vice versa, as illustrated in Figure 1.

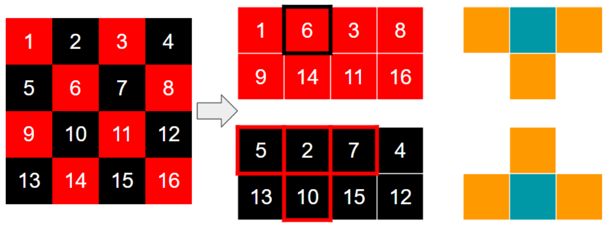
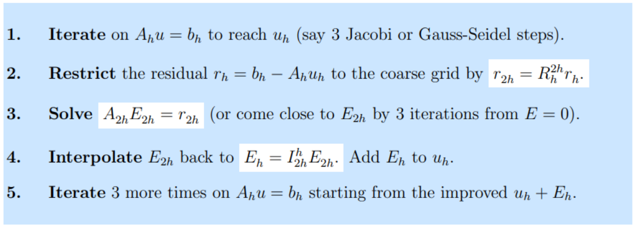


Figure 1. Red black optimization strategy. *Left:* An example of the red-black checkerboard. *Middle:* Reorder of the pixels to build the red and the black "image". *Top Right:* To update pixels at odd columns in the red image or at even columns in the black image, we use orange and green pixel values from black (red) image to update the value of green pixel in the red (black) image. *Bottom Right:* Vice versa. *Remark:* Column indices start from 0.



1. **Iterate** on $A_h u = b_h$ to reach u_h (say 3 Jacobi or Gauss-Seidel steps).
2. **Restrict** the residual $r_h = b_h - A_h u_h$ to the coarse grid by $\boxed{r_{2h} = R_h^{2h} r_h.}$
3. **Solve** $\boxed{A_{2h} E_{2h} = r_{2h}}$ (or come close to E_{2h} by 3 iterations from $E = 0$).
4. **Interpolate** E_{2h} back to $\boxed{E_h = I_{2h}^h E_{2h}.}$ Add E_h to u_h.
5. **Iterate** 3 more times on $A_h u = b_h$ starting from the improved $u_h + E_h$.

Figure 2. Steps for a two-grid V-cycle.

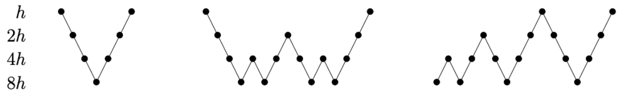


Figure 3. V-cycles, W-cycles, and F-cycles (FMG).

2.3. Multi-Grid Solvers

While single-grid solvers are effective, their performance can be further improved. A notable issue with single-grid methods in recovering the tonemapped image from pixel-wise divergence is their tendency to minimize high-frequency errors first, leading to blockiness in the output rather than smoother gradients. To overcome this, we introduce multi-grid methods, which solve the Poisson equation at varying resolution levels. This approach allows for a more balanced recovery of frequencies at different resolutions. Figure 2 illustrates the algorithm for a two-grid V-cycle, which involves a single instance of downsampling and upsampling. Common multi-grid cycles include the V-cycle, W-cycle, and F-cycle (or full multigrid). The paths for downsampling and upsampling in these three types of cycles are depicted in Figure 3 [2].

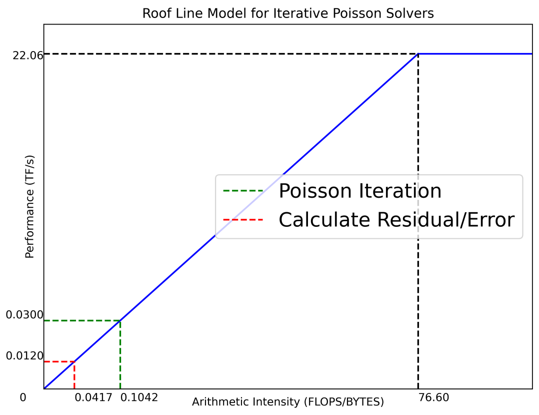


Figure 4. Roof line model for the Poisson iteration and residual calculation.

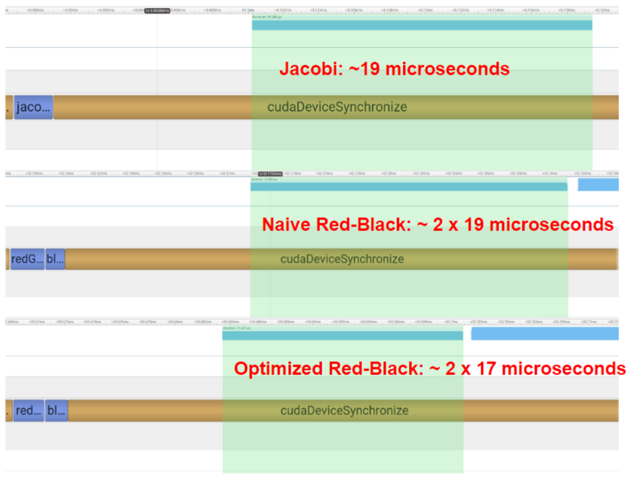


Figure 5. Execution time of Jacobi, Naive Red-Black Gauss-Seidel, and optimized Red-Black Gauss-Seidel CUDA kernels.

3. Results

3.1. Roof Line Model

We conducted all experiments using an NVIDIA 4060Ti GPU (16GB), which has a peak memory bandwidth of 288 GB/s and a peak flop rate of 22.06 TFLOP/s. Reference: Specs for NVIDIA 4060Ti (16GB). The flops/bytes ratio for a Poisson iteration is $\frac{5}{8*5+8}$=0.1042. Consequently, the peak flop rate for a Poisson iteration is 0.0300 TF/s = 30.0 GF/s. The residual computation is $\frac{1}{2*8+8}$=0.0417, leading to a peak flop rate for this step of 0.0120 TF/s = 12.0 GF/s, as illustrated in Figure 4.

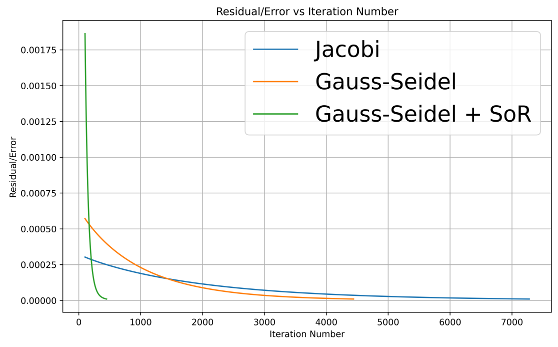


Figure 6. Residual and error with respect to the iteration count.

3.2. Influence of Optimized Red-Black Reordering

One Jacobi iteration kernel takes approximately 19 microseconds, and one red-black Gauss-Seidel iteration takes about $2 \times 19 = 38$ microseconds, as both the red and black kernels require 19 microseconds each. Our aim is to reduce the execution time of a single Gauss-Seidel iteration. By implementing the strategy outlined in section 2.2.2, we achieved a reduction of about 10.5% in iteration time. Additionally, having separate red and black images at each iteration allows us to use only one of them for residual/error calculation, enhancing overall convergence time, especially when checking the residual frequency is high.

3.3. Convergence Rate

Mathematically, the Jacobi method exhibits the slowest convergence rate. The Gauss-Seidel method is marginally faster, while the SOR method is the quickest among all single-grid methods. Experimental results show these findings, as depicted in Figure 6.

3.4. Performances

We use the following abbreviations for brevity: GS for Gauss-Seidel, RB for Red-black reordering, SOR for Successive Over Relaxation, and RB++ for Optimized Red-black Reordering. The convergence times using single-grid methods for recovering a sample function and a sample tonemapped image are presented in Table 1 and Table 2, respectively. In Table 1, the performance for recovering the sample function aligns with expectations, where Jacobi $\geq$ GS $\geq$ SOR in terms of convergence time. However, Table 2 shows that when applying these solvers to real-world image divergences, SOR does not significantly outperform the others. This could be due to a suboptimal choice of ω or a high threshold, making GS sufficiently fast.

A comparison between single-grid and multi-grid methods is presented in Table 3 and Figure 7. The iteration count for multi-grid methods includes all iterations at any resolu-

Method	Time(ms)	Iterations	L1 Error
Jacobi	248.982	7500	0.0218
GS & RB	246.874	4500	0.0120
GS & RB & SOR	30.7845	500	0.0008
GS & RB++	226.298	4500	0.0120
GS & RB++ & SOR	28.2442	500	0.0008

Table 1. Convergence statistics using various single-grid methods to recover the function $f(x, y) = \sin \frac{\pi(x+y)}{100}$. L1 error stands for the error between the recovered function and the real function but not the residual between iterations. Parameters: max iteration = 10000, check frequency = 500, residual tolerance = 1e-5.

Method	Time(ms)	Iterations
Python	12832.0298	2683
Jacobi	239.699	2683
GS & RB	31.8722	307
GS & RB & SOR	36.4054	340
GS & RB++	28.0181	307
GS & RB++ & SOR	30.9033	340

Table 2. Convergence statistics using various single-grid methods to recover the tonemapped image of the Belgium scene. Parameters: max iteration = 10000, check frequency = 1, residual tolerance = 1e-4.

Method	Time(ms)	Iterations
Jacobi	239.699	2683
GS & RB++	28.0181	307
V-cycle	4.6275	40
W-cycle	9.8029	115
F-cycle	8.9652	95

Table 3. Convergence statistics using various single-grid methods to recover the tonemapped image of the Belgium scene. Parameters: cycle count = 1, coarsest level check frequency = 1, coarsest level residual tolerance = 1e-4, coarsest level side length = 100-300, pre/post iterations = 3,

tion level. According to Table 3, the V-cycle is the fastest, while W- and F-cycles are less optimal. This discrepancy might be due to the sequential nature of multigrid methods, which are more suited for CPU than CUDA. Nevertheless, multi-grid methods are substantially faster than single-grid methods, with quicker convergence rates. Furthermore, as shown in Figure 7, the SOR solver tends to produce blockiness (dark areas), whereas V- or F-cycle solvers yield much smoother gradients, preferred for HDR tonemapping.

4. Conclusion

In summary, this project delves into parallel optimization techniques to expedite the process of solving a Poisson equa-

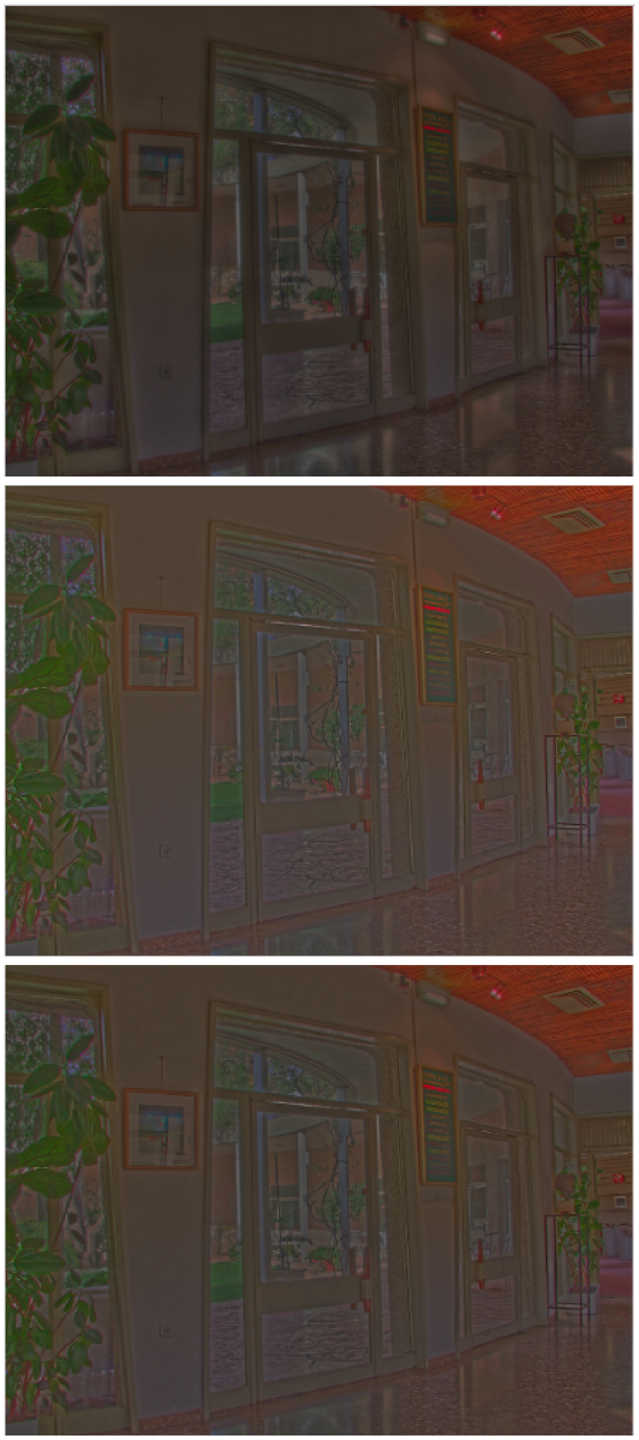


Figure 7. HDR tonemapping results using (top to bottom) the SOR, V-cycle, and F-cycle algorithms separately.

tion, which is subsequently applied to the gradient domain HDR tonemapping algorithm. This application significantly enhances the efficiency of the algorithm. For images ranging from 1K to 2K resolution, we can now solve the Pois-

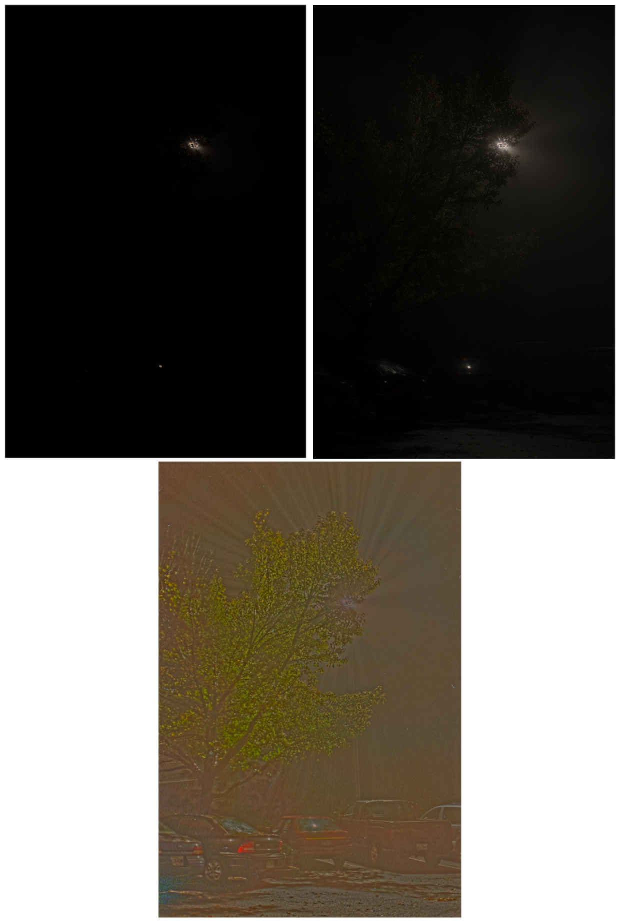


Figure 8. Comparison of linear, global Gamma, and HDR compression tonemapping results on the BigForMap scene.

son equation within 10 milliseconds. However, the entire pipeline still requires approximately 100-200 milliseconds to execute. The primary bottleneck lies in attenuating the gradients before solving the Poisson equation. Currently, this step is implemented using C++ and LibTorch on a CPU. Developing customized CUDA kernels for these parts seems a promising avenue for further performance improvement, but this remains a goal for future work. We are optimistic that real-time gradient domain HDR tonemapping using exclusively CUDA can be achieved in the future.

References

[1] Raanan Fattal, Dani Lischinski, and Michael Werman. Gradient domain high dynamic range compression. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, pages 671–678. 2023. 1

[2] Gilbert Strang. 18.086 mathematical methods for engineers ii, spring 2006. 2006. 2