CME 342 Final Project

Christopher Maes

Abstract

We implement a method, in CUDA, of performing image colorization. The method relies on a parallel implementation of preconditioned conjugate gradient. We use this method to solve two linear systems on two GPUs each of 3 million unknowns in just under 4 minutes. This produces a coloring of an 1050×3360 image (the size of a modern dual screen desktop background) of El Capitan and the lower Yosemite valley.

Suppose we are given a $m \times n$ grayscale image, and a collection of colored annotations as shown in Figure 1.

Figure 1: A grayscale image along with color annotations.

We are interested in computing a *colorization* of the image, that is colored $m \times n$ image C. This colorization should have the property that it respects the user's color annotations, and that pixels with similar grayscale values (intensities) should have similar colors.

To make the above colorization properties more precise, we consider each pixel in the image as linked with its four nearest neighbors, then we wish to find a colorization x that solves the following problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \sum_{i \sim j} c_{ij}(x_i - x_j)^2 \\ \text{subject to} & x_i = d_i \quad i \in \mathcal{M} \end{array} \tag{1}$$

Here we use $i \sim j$ to denote that pixels i and j are neighbors, or that pixels i and j share an edge. The weights $c_{ij} = c_{ji} > 0$ are a measure of the closeness of the grayscale values g_i and g_j of pixel i and j. If $g_i = g_j$ we want c_{ij} to be large, if g_i and g_j are very different we want c_{ij} to be small. To achieve these properties we use the relation

$$c_{ij} = \exp(-\beta \, |g_i - g_j|).$$

Here β is a free parameter, for all colorizations we take $\beta = 200$. Note that we also scale the greyscale values of the image so the $\max_{ij} |g_i - g_j| = 1$.

Observe that Problem 1 has a quadratic objective function. If we define the Laplacian matrix $L \in \mathbb{R}^{mn \times mn}$ as $L = A^T C A$ where A is the edge-node adjacency matrix, and C is a diagonal matrix containing the edge weights c_{ij}, then we have that $x^T A^T C A x = \sum_{i \sim j} c_{ij}(x_i - x_j)^2$. Thus we can rewrite Problem 1 in matrix form as

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^T L x \\ \text{subject to} & x_m = d \end{array} \tag{2}$$

Here L is the Laplacian matrix for the image, x_m is the set of unknowns corresponding to marked pixels, and d is the vector of annotated values obtained from the user. Observe that if P is a permutation that orders x so that the marked and unmarked unknowns are grouped, such that

$$Px = \begin{bmatrix} x_m \\ x_u \end{bmatrix}$$

Then we have that

$$\frac{1}{2}x^T L x = \frac{1}{2}x^T P^T (PLP^T) P x = \frac{1}{2}\begin{bmatrix} x_m^T & x_u^T \end{bmatrix} \begin{bmatrix} L_m & R^T \\ R & L_u \end{bmatrix} \begin{bmatrix} x_m \\ x_u \end{bmatrix} \quad \text{with} \quad PLP^T \equiv \begin{bmatrix} L_m & R^T \\ R & L_u \end{bmatrix}.$$

Here we have that $L_u = L_u^T \succ 0$. Since the values of the variables x_m are fixed ($x_m = d$), we need only optimize over the variables x_u. Thus we can solve the unconstrained problem

$$\underset{x_u}{\text{minimize}}\ F(x_u) = \frac{1}{2}x_u^T L_u x_u + x_m R^T x_u$$

Since $L_u \succ 0$, the unique minimizer $x_u^\star$ is characterized by $\nabla F(x_u^\star) = 0$. We have that $\nabla F(x_u) = L_u x_u + R x_m$. Thus we seek to solve the linear symmetric positive-definite system

$$L_u x_u = -Rd. \tag{3}$$

In fact we have to solve two such systems, with two different right-hand sides d_1 and d_2 (but the same cofficients L_u) since the color annotations come in two different components. Figure 2 shows the different color annotation components.

Figure 2: The grayscale image (the Y luma component), the annotated I chrominance component, and the annotated Q chrominance component.

Note that because we consider each pixel as linked with its four nearest neighbors L_u is pentadiagonal. We could solve (3) with a sparse direct method, for instance compuating the sparse Cholesky factorization $L_u = R_u^T R_u$. However as m and n get bigger the amount of fill in of the Cholesky factor R_u makes the memory requirements too expensive.

Thus, we choose to solve (3) via an iterative method: preconditioned conjugate gradient (PCG). PCG solves the system $Ax = b$ when A is symmetric positive-definite, by solving the equivalent system $M^{-1}Ax = M^{-1}b$. The number of iterations of PCG depends on the distribution of the eigenvalues of $M^{-1}A$. For PCG to be computationally efficient a good preconditioner M must be found. Several papers have considered advanced preconditioners for this problem; these preconditioners often have a multigrid flavor and try to take into account the annotations and grayscale properties of the image. Implementing one of these preconditioners is beyond the scope of this work. Instead, for this work, we use a simple preconditioner of $M = \mathbf{diag}(L_u)$.

A sample implementation of PCG is given in Algorithm 1. Note that the main work involved in PCG is the computation of the matrix vector product $w = Ap$. We present four implementations of the above algorithm: a serial implementation, a parallel matrix-vector product but serial PCG implementation, an unoptimized single GPU parallel PCG implementation, and an optimized multi-GPU parallel PCG implementation.

The $\mathcal{L}$ operator

Although, the matrices L and L_u are pentadiagonal, constructing, storing, and reloading the diagonals of these matrices would be inefficient in terms of computation, storage, and memory access. However, given

Algorithm 1 Precondition Conjugate Gradient

1: $x_0 = 0$
2: $r_0 = b$
3: $z_0 = M^{-1} r_0$
4: $p_0 = z_0$
5: $\rho_0 = r_0^T z_0$
6: **for** $k = 0, 1, \ldots,$ until $\sqrt{r_k^T r_k} < \epsilon(1 + \| b \|)$ **do**
7: $w = A p_k$
8: $\gamma = p_k^T w$
9: $\alpha = \rho_k / \gamma$
10: $x_{k+1} = x_k + \alpha p_k$
11: $r_{k+1} = r_k - \alpha w$
12: Solve $M z_{k+1} = r_{k+1}$
13: $\rho_{k+1} = r_{j+1}^T z_{k+1}$
14: $\beta = \rho_{k+1} / \rho_k$
15: $p_{k+1} = z_{k+1} + \beta_k p_k$
16: **end for**

the original grayscale image computing the matrix-vector product $y = Lx$ is quite simple and efficient. Therefore, we will perform all our computations without forming the matrices L and L_u. To emphasize this we will use the notation $y = \mathcal{L}(x)$ to denote the fast matrix-vector product operator.

We will show how to use the $\mathcal{L}$ operator to compute the right-hand size $b = -Rd$, and to compute matrix-vector products $y_u = L_u x_u$. Observe that if we set $x_u = 0$ and $x_m = d$ then we have that

$$x_d = P^T \begin{bmatrix} d \\ 0 \end{bmatrix} \quad \text{and so} \quad \begin{bmatrix} y_m \\ y_u \end{bmatrix} = P^T \begin{bmatrix} 0 \\ R^T d \end{bmatrix} = \mathcal{L}(x_d).$$

So we can easily extract b from the unmarked components of $\mathcal{L}(x_d)$. Also observe that if we set $x_m = 0$ then we have that

$$x = P^T \begin{bmatrix} 0 \\ x_u \end{bmatrix} \quad \text{and so} \quad \begin{bmatrix} y_m \\ y_u \end{bmatrix} = P^T \begin{bmatrix} R^T x_u \\ L_u x_u \end{bmatrix} = \mathcal{L}(x).$$

So we can easily extract $y_u = L_u x_u$ from the unmarked components of $\mathcal{L}(x)$. In fact, for our purposes, since x_m is fixed as $x_m = d$, we can ignore the components y_m. We can easily construct a operator $\tilde{\mathcal{L}}$ that always sets $y_m = 0$; this will be important in our parallel implementation of PCG.

Serial Implementation

We first implement a serial version of PCG and our colorization method in C based off the authors MATLAB implementation. If our original image is $m \times n$, then $n_{\text{tot}} = mn$, n_m is the number of annotations, and n_u, the number of annotations is equal to $n_{\text{tot}} - n_m$. For ease of implementation in this serial version of PCG we work with vectors $x, p, w, r, z \in \mathbb{R}^{n_u}$. To use the matrix-vector product operator $\mathcal{L}$ we must expand these vectors to $\mathbb{R}^{n_{\text{tot}}}$ by placing zeros in the annotated components. When we compute a vector $y = \mathcal{L}(x)$, we must also remove the annotated components in $y \in \mathbb{R}^{n_{\text{tot}}}$ to get a vector in $\mathbb{R}^{n_m}$. Other than these operatations the heart of the serial implementation of the colorization method is a straight-forward implementation of the PCG algorithm already given in Algorithm 1.

We use optimized BLAS operations to implement the level 1 linear algebra operations in PCG. Lines 5, 8, and 13 of the algorithm utilize the `sdot` subroutine (note although the operands are single precision that the summation is accumulated in double precision). Lines 10 and 11 use the `saxpy` subroutine. Line 15, which isn't quite in the form of a `saxpy` (since your multiplying p by β and then storing the result back into p) requires the `sscal` routine, followed by a `saxpy`. Line 6 uses `snrm2`.

Since M is a diagonal matrix we implement the solve $Mz = r$ in Line 13 by simply performing the componentwise scaling $z = r./d$ (to borrow MATLAB notation).

Note that we require two solves $L_u x_u^{(i)} = d_i$, $i = 1, 2$. This serial version contains two routines, one which performs the solves in order (first solving for $x_u^{(1)}$ then solving for $x_u^{(2)}$) and in a single shot, and another version that interleaves the iterations of conjugate gradient for each of the right-hand sides. This last routines allows us to perform an animation of the colorization.

Parallel Matrix-vector product implementation

Our first step toward a parallel implementation for the GPU is parallelizing the matrix-vector operator $\mathcal{L}$. We implement this operator as a single CUDA kernel. We assign a single thread on the GPU to each pixel in the grayscale image. Each thread reads in the grayscale value of its pixel, as well as its four neighbors, and uses this to compute the edge weights c_{ij}. Each thread must also read in the values of the vector $x \in \mathbb{R}^{n_{\text{tot}}}$ corresponding to these five pixels. Suppose we are the ith thread then

$$d_i = \sum_{j \sim i} c_{ij} = \sum_{j \sim i} \exp(-\beta \left| g_i - g_j \right|)$$

and the ith component of the vector $y = \mathcal{L}(x)$ is given by

$$y_i = d_i x_i - \sum_{j \sim i} c_{ij} x_j.$$

Thus we see that each thread requires 10 fetches from global memory and performs a single store to global memory. We can speed things up by having each thread in a block pull in its value of g_i and x_i and store them in shared memory, so these values can be reused by other threads in the block. We will also need to pull in the values of g_i and x_i surrounding the boundary of the block. Thus a thread on the boundary must (in the worst case) perform 6 fetches from global memory. Considering the discrepancy between global memory access and floating point computations, this makes the matrix-vector product kernel memory bound.

For this implementation the *only* computation we perform on the GPU is this matrix-vector product. Thus, on each iteration of PCG we take $p \in \mathbb{R}^{n_u}$ in Line 11, add zeros to form a vector $p_{\text{full}} \in \mathbb{R}^{n_{\text{tot}}}$, `CudaMemcpy` p_{full} to the GPU, compute $w_{\text{full}} = \mathcal{L}(p_{\text{full}})$ on the GPU, `CudaMemcpy` w_{full} back to the host, and remove the zeros to get $w \in \mathbb{R}^{n_u}$. All other operations in PCG are performed using the BLAS on the host. Nevertheless, we see a factor of 10 reduction in the time required to perform our colorization method (using the example in Figure 1).

Parallel PCG implementation: single GPU

The next step is moving the level-1 BLAS operations onto the GPU. This means performing parallel reductions for the dot products $r^T r, \gamma = p^T w, \rho = r^T z$ in Lines 6, 8, and 13. As well as vector operations for the `saxpy`'s on Lines 10 and 11, the almost `saxpy` on Line 15 and the scaling $z = r./d$ on Line 12.

In the serial implementation of PCG we worked with vectors in $\mathbb{R}^{n_u}$, moving to and from vectors in $\mathbb{R}^{n_{\text{tot}}}$ only when computing $\mathcal{L}(x)$. However, the indexing involved with these conversions would be difficult to implement in parallel and would result in uncoalesced reads and writes. Thus in our parallel implementation of PCG on the GPU we work entirely with vectors in $\mathbb{R}^{n_{\text{tot}}}$. An inspection of Algorithm 1 will confirm that provided we start with a vector b whose components corresponding to annotated variables are zero, all subsequent vectors produced by the algorithm will also have zeros in the annotated components. Provided that $w = \mathcal{L}(p)$ is zeroed appropriately or the operator $w = \tilde{\mathcal{L}}(p)$ is used. If the annotated components in the vectors p, r, w and z are zero then the dot products γ and ρ will be the same as if we were working in the

lower dimension $\mathbb{R}^{n_u}$. Thus the iterates x_k of PCG in the expanded space $\mathbb{R}^{n_{\text{tot}}}$ will be identical to those computed in the reduced space $\mathbb{R}^{n_u}$.

Note that instead of performing the basic BLAS vector operations of `saxpy`'s and `sscal`'s, we can construct kernels specifically tuned to PCG. Since the most expensive operation on the GPU is moving data onto chip we constructed these kernels to make the most of the data once we had pulled it into a register.

For example we implement a single kernel to perform Lines 11 and 12 and the first step in the reduction for Lines 13 and 6. Each thread in this kernel is assigned an index j of the vectors w, r and z and the vector $d = \mathbf{diag}(L_u)$. Each thread loads w_j, r_j and d_j and then does the operatations

$$\begin{aligned} r_j &\leftarrow r_j - \alpha w_j \\ z_j &\leftarrow r_j / d_j \\ f_j &\leftarrow r_j z_j \\ h_j &\leftarrow r_j r_j \end{aligned}$$

The vectors f and h are local to a block of threads and stored in shared memory on chip. Once all threads in a block are done with the above operatations they cooperate to compute the partial sums $u_i = \sum_{j \text{ in block } i} f_i$ and $v_i = \sum_{j \text{ in block } i} h_j$. The vector u contains the partial sums after the first step in the reduction for $\rho_{k+1} = r_{k+1}^T z_{k+1}$ and the vector v contains the partial sums after the first step in the reduction for $r_{k+1}^T r_{k+1}$.

With this implementation of PCG we can now stay almost entirely on the GPU. There is no need to transfer vectors back and forth from the host to the device. However, upon the completion of each iteration of PCG we transfer a small set of scalars back to host, to decide if we should continue with the next iteration and to report progress information to the user. Finally since all data is now stored on the GPU, we can perform the final color conversion from YIQ to RGB on the device. If the GPU is connected directly to a display we can even blit the final texture to the screen without having to transfer back to the host.

By moving the Level-1 BLAS operations to the GPU, and avoiding transfers from the host to the device, we see a factor of four reduction in the time required to perform our colorization method.

Parallel PCG implementation: multi-GPU

Finally, in one sense our colorization method is embarrassingly parallel. We need to do two entirely separate and independent solves $L_u x_u^{(1)} = d_1$ and $L_u x_u^{(2)} = d_2$. This is the prefect use of a mult-GPU system. By using two GPUs to perform both solves in parallel we see a factor of two reduction in the time required to perform our colorization method.

Numerical Results

We use four sample images of increasing size to test the implementations: the child image already shown in Figure 1, the MATLAB clown with annotations taken from the original color image, the author's annotated version of Rosenfeld's photograph *Hoag's Outboard*, originally taken in 1925 and for which no color information exists, shown in Figure 5, and a part of a 360° degree panorama taken by the author in April of 2008 on top of Sentinel dome in Yosemite looking back into the Yosemite Valley and El Capitan with color annotation from the original, shown in Figure 6. Timing information is shown in Table 1. Timings were performed on a compute node with two Quadro FX 5600 GPUs.

Further Work

A year ago the author contributed to a plugin for the open-source photoshop clone THE GIMP that implemented a similar method of colorization. This plugin was abandoned by THE GIMP community because

Table 1: Timings for the four programs on the four sample images. The † indicates timing information for only 3000 iterations.

Image	Size	n_{tot}	n_u	Serial	Parallel $\mathcal{L}(x)$	Parallel PCG	Multi-GPU
child	264×320	84480	67076	123s	13	3.8	1.62
clown	200×320	64000	49381	263	28	9.2	3.6
hoag	639×800	511200	484414	$927^{\dagger}$	560	83	35
sentinel	1050×3360	3528000	2164668	NA	NA	589	230

of the dependencies on the sparse linear algebra package UMFPACK required to solve the large linear systems the colorization method produces. This implementation of parallel PCG might make colorization computationally feasible; provided a good preconditioner can be found.

Test Images

Figure 3: Child Image

Figure 4: Clown image

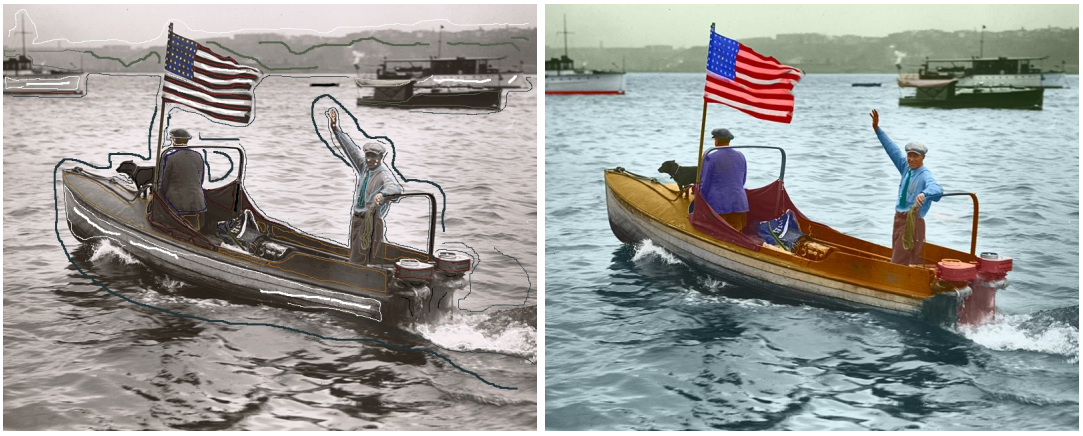

Figure 5: Hoag's outboard

Figure 6: Panorama from Sentinel Dome