Slide 1	Good morning. In this talk I will present work, done in collaboration with Hugues Hoppe of Microsoft Research, on streaming the multigrid solver. Our project is motivated by recent work in gradient-domain image processing, and focuses on the challenge of extending these techniques to huge images. Let's consider two examples.
Slide 2	When compositing a set of images into a panorama… [Click] Even with well-chosen stitching boundaries, exposure differences can result in seams. Correcting this is easily done in the gradient domain. To remove the sharp transitions due to large gradients, we… [Click] Set the values of all boundary-crossing gradients to zero, and solve for the image with the new gradients, obtaining a seamless composition.
Slide 3	Another challenging problem is visualizing high-dynamic-range images on low-dynamic-range displays. Because of the limited range, no single exposure reveals all the detail. Once again, this problem is easily addressed in the gradient domain. Amplifying small gradients and dampening large ones… [Click] We can exaggerate subtle color variations to obtain a single visualization exposing all of the detail.
Slide 4	More generally, gradient-domain image processing proceeds as follows: Starting with an image, or a set of images, [Click] We extract the gradients, [Click] Combine and modify them, [Click] And then solve for the image satisfying the new constraints. [Click]
Slide 5	Previous works have shown that by processing gradients in different ways, a variety of visual effects can be realized.
Slide 6	Though they differ in how they process the gradients, they all must solve for the image that fits the new constraints. And this is what we focus on: Finding the best-fit image, specifically when the image and the constraints are too large to fit into memory.
Slide 7	We begin by describing the linear system defined by the gradient constraints and reviewing traditional methods for solving the system when it is small enough to fit into memory.

Slide 8	To perform gradient-domain processing we must be able to fit an image to a desired gradient field. That is, we must invert the gradient operator. The problem is that this system of equations is over-constrained. At each pixel, we have two constraints and only one color value. To address this problem, we can apply the divergence operator to both sides. [Click] This transforms the constraints from a vector field… [Click] To a scalar field, And changes the linear system… [Click] So that now we are left with the problem of inverting the Laplacian operator.
Slide 9	Since the Laplacian measures the average difference between a pixel and its neighbors, the system can be solved using a simple Gauss-Seidel solver: Iterating over the pixels… [Click] We assume that the values of all but the current pixel are correct, [Click] And we update the value of the current pixel so that the averaged difference constraint is satisfied. [Click] We iterate, successively updating the values of the pixels, and converge to the correct solution.
Slide 10	Though this approach converges, it does so slowly. The problem is that after a small number of iterations, we only have a good solution for the high-frequency part of the image. [Click] And as a result, we are still left with the problem of solving for the lower frequencies.

Slide 11	This problem is addressed by the V-cycle of the classic multigrid solver. Starting with a set of desired constraints and an initial guess [Click] We perform a few Gauss-Seidel updates to get the high-frequency part of the solution. [Click] We subtract off the constraints satisfied by this solution, to get the low-frequency, residual constraints. [Click] Since these are low-frequency, we can solve them by: Down-sampling to a lower resolution, Solving the problem more efficiently in the lower-dimensional space, And up-sampling back. [Click] We add in the previously computed high-frequency solution [Click] And refine by performing a few more Gauss-Seidel updates. [Click] To solve the lower-dimensional problem, we apply the multigrid solver recursively, repeating until the dimension is low enough that a brute-force solver can be used.
Slide 12	Due to the size of the images we will be processing, it is essential that our solver converge as efficiently as possible. This implies that we must carefully choose how we define the up-sampling, down-sampling, and Laplacian operators.
Slide 13	To define these operators, we must first choose how to interpret a discrete set of pixels as elements of a continuous function. We could think of the elements as box functions… [Click] in which case a set of pixels defines a piecewise constant function. We could think of them as tent functions, or first order B-splines… [Click] And then the pixels define the piecewise linear interpolant. As second order B-splines… [Click] The pixels define a smoother function approximating the sample values. Which becomes still smoother with third order B-splines. [Click] The choice we make defines both the re-sampling operators and the Laplacian.

Slide 14	It defines the re-sampling operators because it dictates how we express coarser elements as linear combinations of finer ones. For example, if we are using 0^{th} order elements… [Click] Then we must express a coarser element as the sum of two finer elements. Using 1^{st} order elements… [Click] We must express a coarser element as the linear combination of three finer elements. [Click] And similarly for the higher order elements.
Slide 15	The choice of element also defines the notion of "neighborhood" used for computing the Laplacian. For 0^{th}-order elements… [Click] The supports of two different pixels never overlap, pixels don't have "neighbors", and the Laplacian can't be defined. For 1^{st} order elements… [Click] The supports will only overlap if the pixels are adjacent to each other, so the Laplacian is defined by averaging over the 1-ring. [Click] And for higher order elements, the increased supports results in a Laplacian computed over a larger neighborhood.
Slide 16	In implementing the solver, we choose the order of the element that converges fastest to an accurate solution. This can be empirically evaluated by computing the gradient field of an image, Using the different solvers to fit an image to the gradient field, And measuring the difference between the original image and the solution. The images on the right show these differences for the different order solvers. Looking at these, it is clear that first order elements provide the worst accuracy and second order elements slightly outperform third order elements. This trend becomes more evident when we look at the plots of the errors. [Click] The best performance is obtained using 2^{nd} order elements, Which reach an error of 1/256 in just 5 updates. An error that would take almost twice as many updates to achieve with the less compact 3^{rd} order elements, and would be still harder to achieve with 1^{st} order elements.

Slide 17	The faster convergence can also be validated analytically by considering the simplified multigrid operator transitioning between two levels. [Click] We can measure the quality of the operator by considering the efficiency with which it dampens the residual error, a property captured by its dominant eigenvalues. [Click] The plots on the right show the eigenvalues for the two-grid operators defined by 1^{st}-, 2^{nd}-, and 3^{rd}-order elements. As expected, the largest eigenvalue for each of the elements has magnitude less than one, ensuring that the residual will always be reduced and that the solver converges. We also see the empirical performance of the solvers reflected in the spectra. [Click] First order elements have the largest dominant eigenvalue and converge least efficiently. [Click] 2^{nd} and 3^{rd} order elements have similar dominant eigenvalues, indicating that in the worst case their performances will converge with similar efficiency. [Click] However, the 2^{nd} order elements have smaller subsequent eigenvalues, and converge more quickly in practice. So we choose 2^{nd} order elements for implementing the solver.
Slide 18	Having decided on the solver, the challenge is to design an implementation that can handle large images. We begin by describing a naïve, out-of-core, implementation that is heavily I/O bound, and then show how this can be improved to obtain a solver requiring just two streaming passes through the data.
Slide 19	Because of the locality of the data access, a streaming solver can be implemented by advancing a fixed size window over the data. As an example, consider the simple case when computing the value at a pixel only requires access to information in a one-ring neighborhood. Starting with an empty window… [Click] We read in the first two rows from disk. And we compute the values in the top row… [Click] Since its one-ring neighbors are now in memory. Reading in the next row… [Click] We compute the values in the second row because now its one-ring neighbors are in memory. Since data in the top row won't be required for subsequent computations we flush it to disk. [Click] And we continue in this fashion… [Click] Advancing the window and computing the values of the pixels in the middle row. In doing so, we process all the pixels while keeping only three rows in memory at any time.

Slide 20	Using this approach, it is easy to implement a V-cycle with a small memory footprint. Streaming through the constraints… [Click] We perform a Gauss-Seidel update of the values as we write the modified solution to disk. Of course, if we want to perform multiple Gauss-Seidel updates… [Click] We can repeatedly read in the solution and constraints, update, and write the new values to disk. Next… [Click] We read in the solution, subtract its Laplacian from the desired constraints, and write out the residual. [Click] And then we read in the residual, down-sample it, and write out the lower-dimensional constraints. Continuing on we can perform the rest of the down-sampling phase… [Click] Repeating until we get to the coarsest resolution where the system fits into memory and can be solved in-core. Similarly… [Click] We can perform the up-sampling phase of the V-cycle. Though this is a low-memory implementation of the solver… [Click] It requires too many streaming passes through the data. We improve on this simple approach in two ways.
Slide 21	The first modification is motivated by earlier works on improving the cache-locality of Gauss-Seidel solvers. The observation made in these works is that we don't have to complete the updates of all the pixels before we can start on the next sequence of Gauss-Seidel updates.

Slide 22	As an example, let's look at performing two Gauss-Seidel updates in a single streaming pass. [Click] We start by loading the first three rows of the image into memory. Since we are using second-order elements, a Gauss-Seidel update requires access to the two-ring neighborhood so we can't update the two rows at the front of the window. But we can update the back row. [Click] We can't perform a second update on the back row, because its neighbors haven't been updated. So we advance the window. [Click] Again, we can't update either of the front two rows, or the back row. But we can update the pixels in the third row from the front. [Click] We advance the window and update again. [Click] And in the naïve approach, we would just keep going: advancing and updating, one row at a time. But we can do better. Having updated the third row, the back row's two-ring neighbors have now all been updated… [Click] So we can update it a second time. And now we proceed as follows: [Click] Advancing the window, updating the third row from the front and the row two behind that, [Click] Advancing one row and updating two. So that as we flush the back row out to disk… [Click] All the pixels in this row have the desired property – in one streaming pass they have been updated multiple times.
Slide 23	Using this approach, we can replace the naïve streaming implementation, which requires a separate streaming pass for every Gauss-Seidel update… [Click] With an implementation that performs all the Gauss-Seidel updates in a single streaming pass.
Slide 24	We can further reduce the number of streaming passes by observing that we don't have to wait for the Gauss-Seidel updates at one level to complete before we begin processing the data at the next level.

Slide 25	As an example let's look at implementing the down-sampling phase of the V-cycle. [Click] We start as before, reading in the first three rows and then repeatedly advancing one row and updating two. And, after a few advances of the window, the values in the top three rows will have been finalized, since all of the pixels have been updated twice. Since the Laplacian requires a two-ring neighborhood, and since the two-ring neighborhood of the first row has been finalized, we can compute the first row of residual constraints. [Click] Advancing the window again… [Click] The two-ring neighborhood of the second row has now been finalized, and we compute the second row of residual constraints. [Click] Repeating this process once more… [Click] we compute the residual constraints for the third row. But, since we're using 2nd order elements we can down-sample… [Click] to get the first row of the lower-resolution constraints. [Click] After two more advances, we can down-sample to get the second row of the lower-resolution constraints. [Click] And after another two advances, we have the first three rows of the lower resolution constraints in memory. Which means that we can start processing at the coarser resolution… [Click] Providing an implementation in which we process the different levels simultaneously.
Slide 26	This allows us to replace the previous, sequential approach, in which we buffered the results of one phase of the solver on disk before proceeding on to the next… [Click] With an interleaved implementation in which the output at one level is piped directly into the next.

Slide 27	Putting everything together, we obtain a system capable of performing the full image processing pipeline in two streaming passes. [Click] In the first pass, we: Read in the images from disk, Compute the gradients, Composite and edit the gradients to obtain the constraint field, Compute the divergence to get the Laplacian constraints, And perform the down-sampling phase of the multigrid solver. [Click] In the second pass, we: Perform the up-sampling phase of the solver, And write the solution out to disk. And this turns out be optimal. [Click] Since Laplacian constraints at one pixel can effect pixel values arbitrarily far away, a pixel's value can't be finalized until all of the constraints have been read in from disk. Since the image is too large to fit in memory, this implies that data has to be buffered during the processing, so at least two streaming passes are required to solve for the image, one for reading the constraints and one for reading the buffered data. Interestingly, if we want to further improve the accuracy of the solution by performing additional V-cycles… [Click] We can do this at the cost of only one extra streaming pass per V-cycle.
Slide 28	Now that we have an efficient streaming multigrid solver in place, let's take a look at some results.
Slide 29	We'll start with image stitching. To date, the only technique that can stitch together large images by explicitly solving the Poisson equation is the one presented here last year by Aseem Agarwala. [Click] This approach was motivated by the observation that when stitching together images, the residual is low-frequency away from the seams. As a result, a high-frequency solution is only needed near the stitching boundaries, and the system can be solved over a quadtree.

Slide 30	We compared our results to those obtained using Agarwala's adaptive quadtree approach, on a number of images, ranging in resolution from 12 to 87 megapixels. [Click] The results can be seen in the table below. [Click] Both methods accurately solve the linear system. [Click] They also both have a memory usage that is sub-linear in the number of pixels. For the quadtree approach this is due to the fact that the system is much smaller than the total number of pixels. For our approach this is due to the fact that we never have more than a few rows in memory at any one time. [Click] Even though our representation is everywhere high-resolution, and the system we solve is much larger than the one solved by the quadtree approach, we obtain an efficient implementation whose running time remains comparable, particularly for larger panoramas made up of many images.
Slide 31	As another application, we consider the problem of tone-mapping in which we modulate the gradient field of an image by amplifying smaller gradients and dampening larger ones… [Click] heightening perceptible detail in low-contrast regions. Since tone-mapping modulates the gradients everywhere, the system can't be localized and the quadtree approach can't be applied. To perform the tone-mapping we have to solve over a grid that is everywhere high-resolution, something that could not previously be done for large images.
Slide 32	Now let's take a look at something a little bit bigger. This 3.3 gigapixel image was obtained by compositing 643 images of the Seattle skyline, taken at different exposures. Zeroing out the seam-crossing gradients and using our method to solve for the image we obtain the following: [Click] A seamless reconstruction, obtained in less than 90 minutes and requiring less than half a gigabyte of RAM, with error markedly below the color resolution of an 8-bit channel. Using the gradients of this new image, we can also perform tone-mapping. [Click] Since the tone-mapping is performed solely on the luminance channel, both the running time and the memory usage have gone down.

Slide 33	To get a better sense of the performance of our method, let's revisit both the memory usage and the running time for the stitching application. [Click] Because we store pixels at two-bytes per channel, the down-sampling phase requires reading in 40 gigabytes worth of gradient constraints, and writing out 53 gigabytes of data. Similarly, the up-sampling phase requires reading in the 53 gigabytes of data and writing out the 20 gigabyte image. This means that had we tried solving the entire image in-core…. [Click] We would have needed at least 53 gigabytes of RAM. More interestingly, since the laptop on which this was run could perform between 30 and 35 megabytes of I/O a second, and since a total of 166 gigabytes of I/O needs to be performed, our solver could not complete in less than 80 minutes… [Click] So the 88 minutes of running time we end up taking is in fact pretty much optimal.
Slide 34	In summary, we have presented a streaming multigrid solver, capable of performing gradient-domain image processing on gigapixel images. Carefully choosing the elements defining the system and scheduling the operations to minimize the I/O, we are able to implement the entire solver in two streaming passes. An implementation that is optimal, given the global nature of the Laplacian constraints. Though not explicitly discussed in this talk, our work also addresses a number of practical issues. [Click] We show that the finite-difference representation traditionally used for gradient-domain processing, can be integrated with our finite-elements system. We show that the mean color value, which is unconstrained by the system, can be set without requiring a separate pass through the data. And we show that the system can be solved over images of arbitrary dimensions without requiring padding up to the next power-of-two.
Slide 35	There are also a number of issues that we do not address – questions that we would like to consider in future work. We would like to extend our system to support arbitrary stitching boundaries and adaptive weighting – allowing for the implementation of methods such as image cut-and-paste, image matting, and local tone adjustment. We would like to extend our method to higher dimensions, providing a way for solving some of the linear systems arising in physical simulations. And finally, we would like to consider ways of improving the performance of our solver; including minimizing the I/O by performing on-the-fly compression and decompression of data, and extending our method to parallelized and distributed settings.
Slide 36	Finally, I would like to conclude by acknowledging the generosity of the community, in sharing their data, their thoughts, and their experiences. Thank you to them…
Slide 37	And thank you to you for your attention.