Thank you, lets start right in

Recording is allowed for this talk

LEAST-SQUARES OPTIMIZATION UNDERPINS GRAPHICS

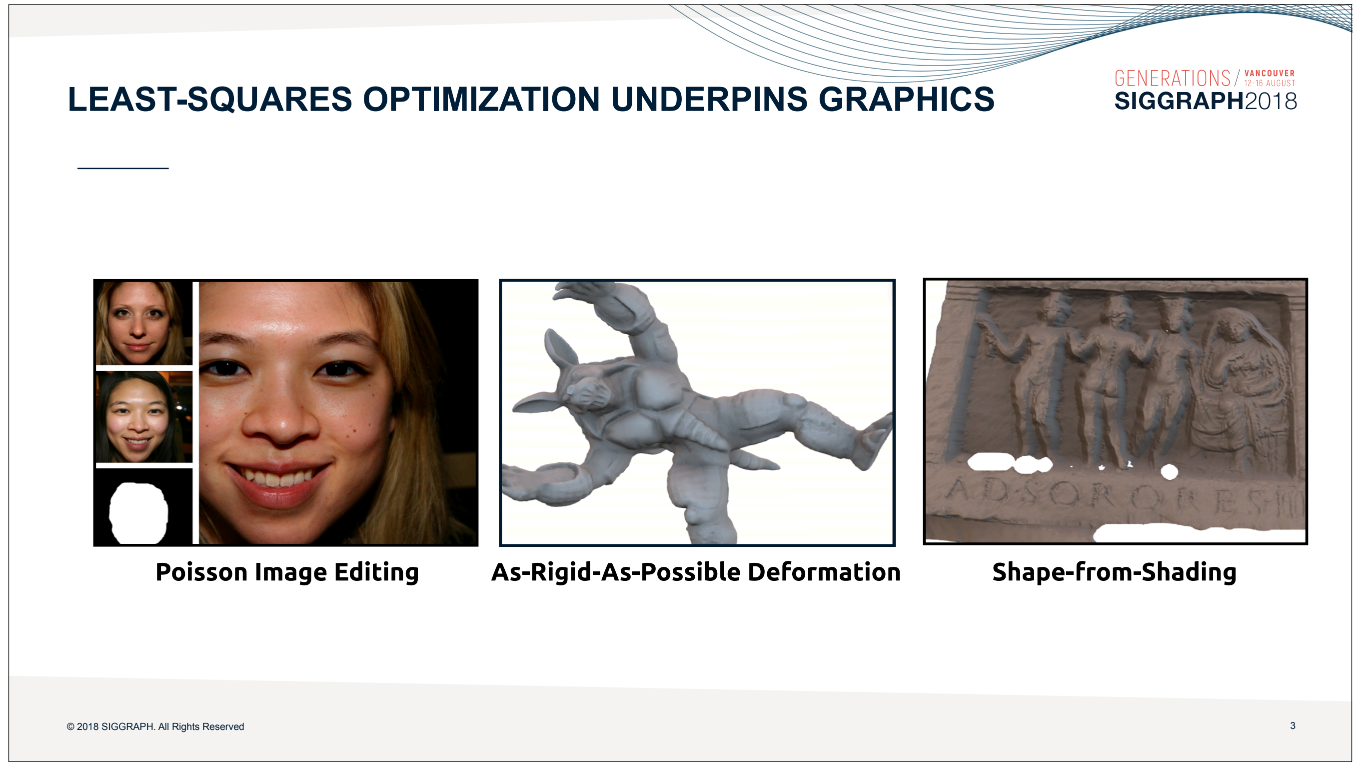
GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Optimization problems are found throughout graphics and vision.

Fundamental techniques like poisson image editing, as-rigid-as-possible warping, and shape from shading are all formulated this way.

At their core, they are just solving least-squares optimization problems over images or meshes.

Especially when executed in real-time, these techniques have a bunch of really interesting applications:

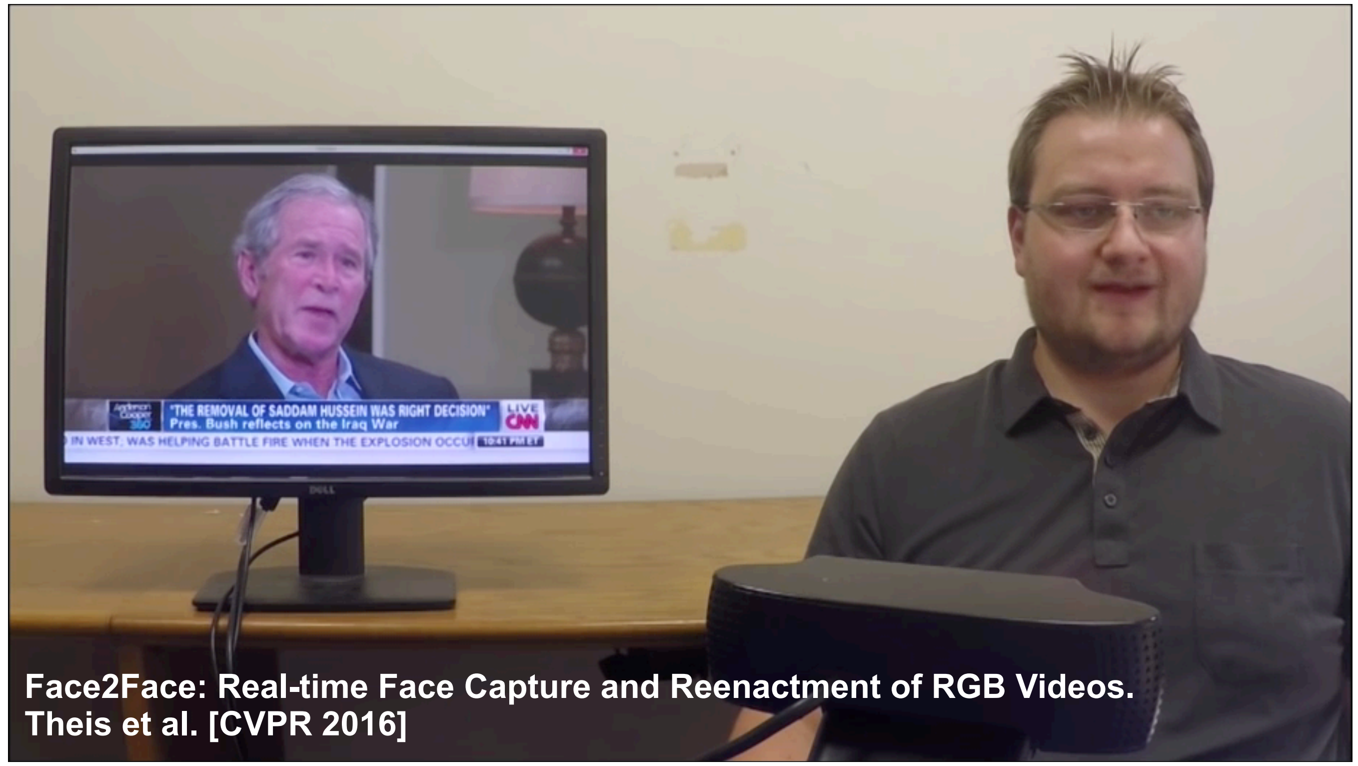


Optimization problems are found throughout graphics and vision.

Fundamental techniques like poisson image editing, as-rigid-as-possible warping, and shape from shading are all formulated this way.

At their core, they are just solving least-squares optimization problems over images or meshes.

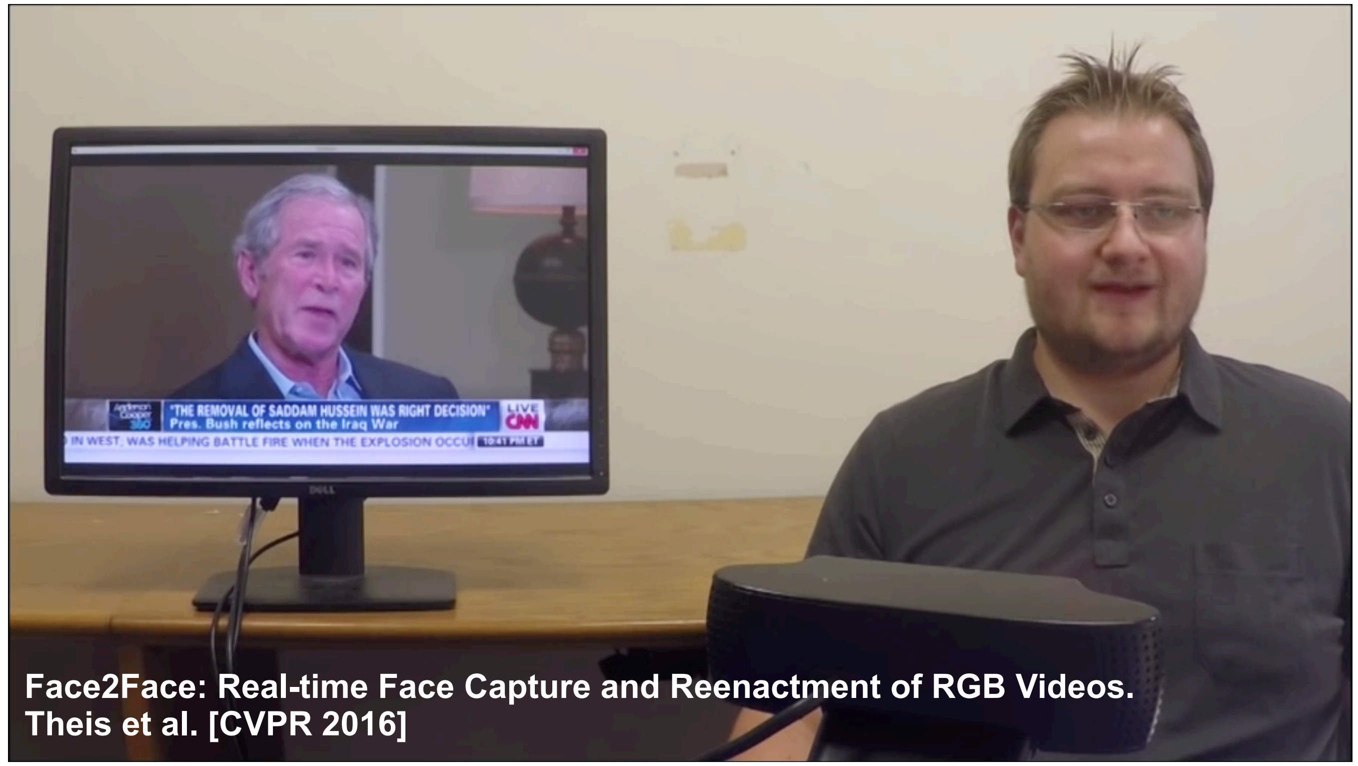
Especially when executed in real-time, these techniques have a bunch of really interesting applications:



Face2Face: Real-time Face Capture and Reenactment of RGB Videos.
Theis et al. [CVPR 2016]

<> You can use webcam to control the facial expressions of a person in a video stream in real-time.

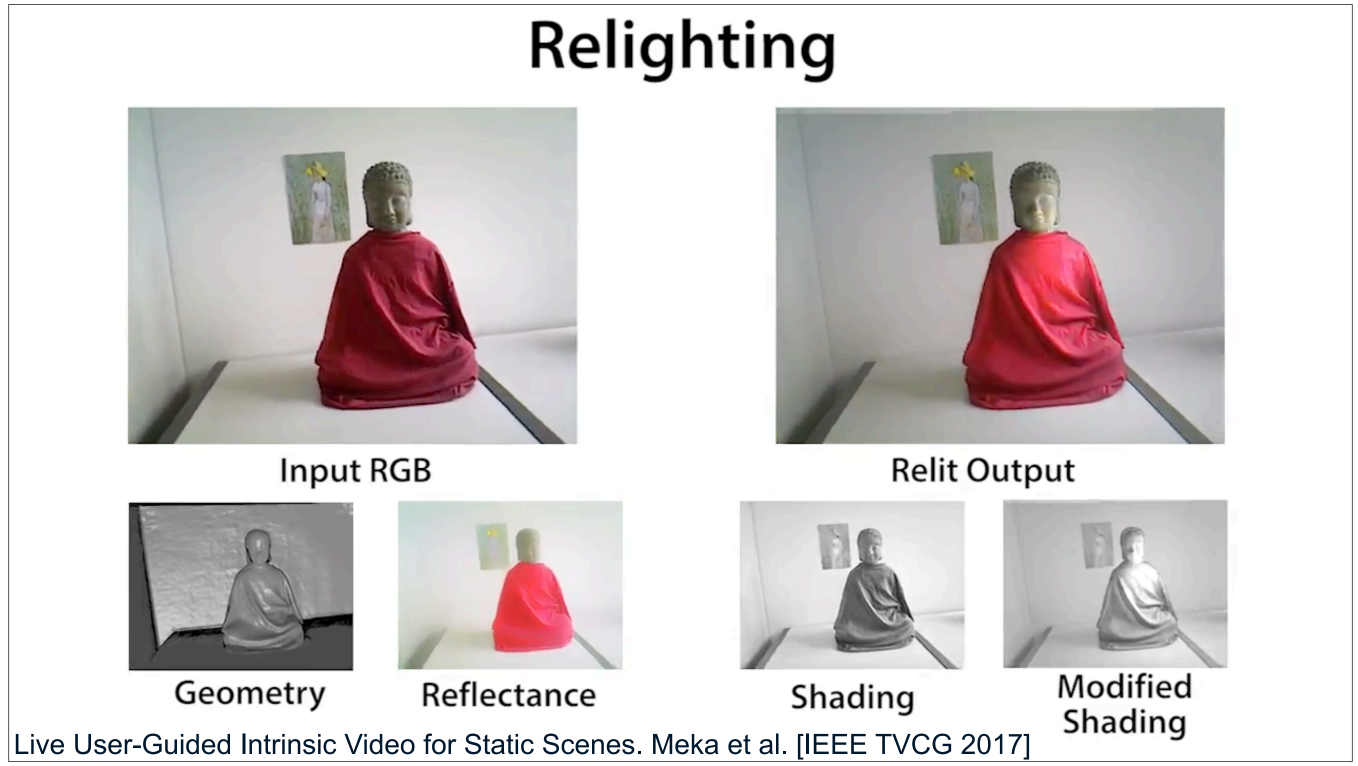
https://www.youtube.com/watch?v=ohmajJTcpNk



Face2Face: Real-time Face Capture and Reenactment of RGB Videos.
Theis et al. [CVPR 2016]

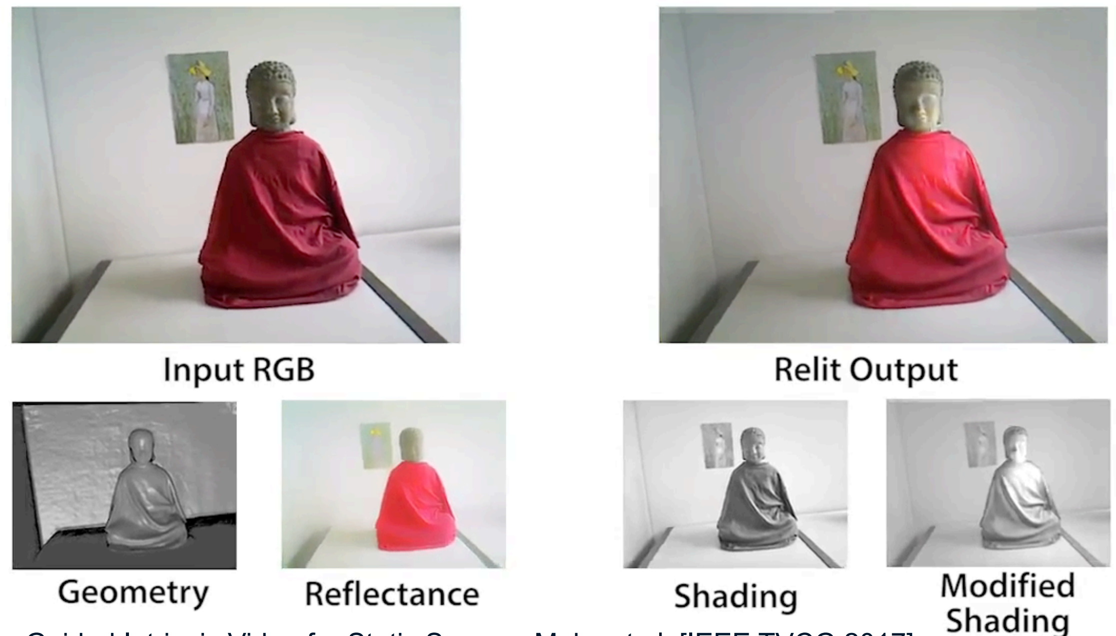
<> You can use webcam to control the facial expressions of a person in a video stream in real-time.

https://www.youtube.com/watch?v=ohmajJTcpNk



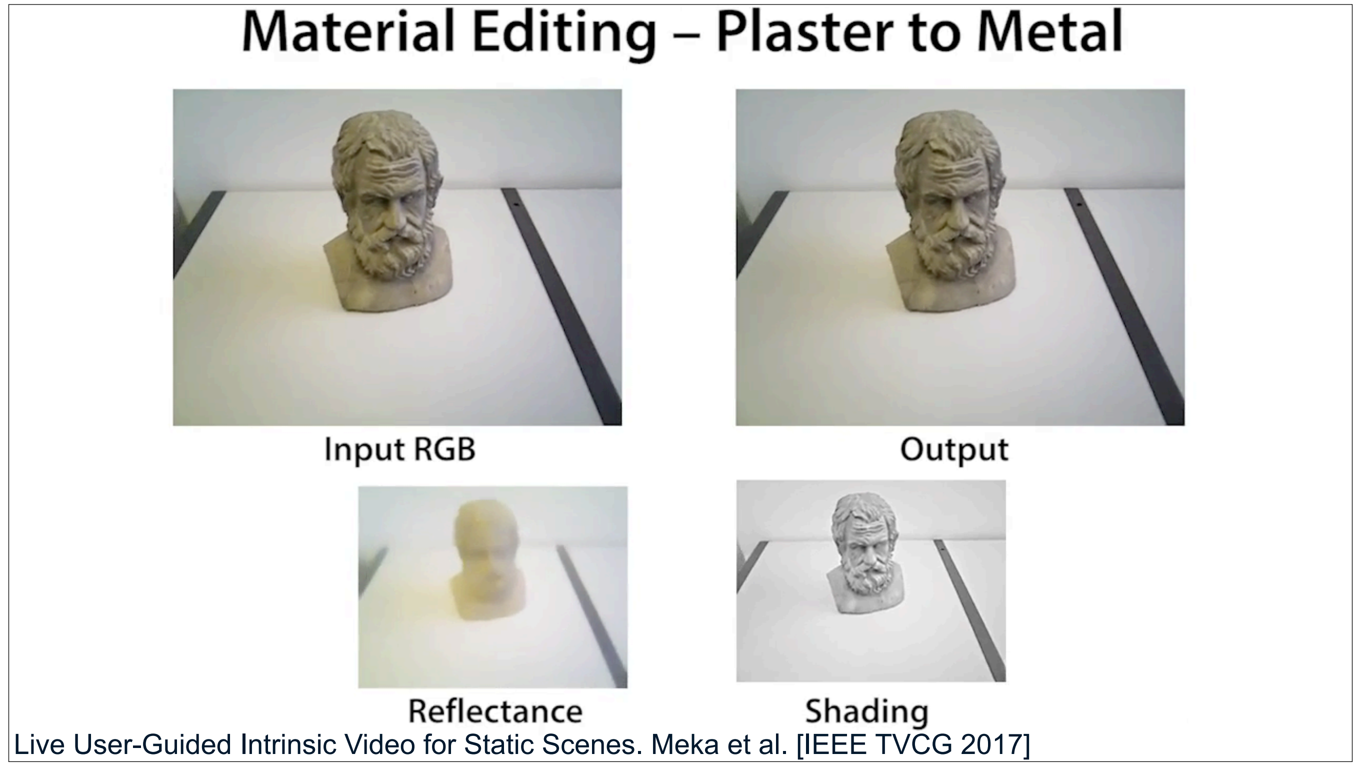
<> You can decompose scenes into geometry and reflectance, and interactively relight the scene.

Relighting

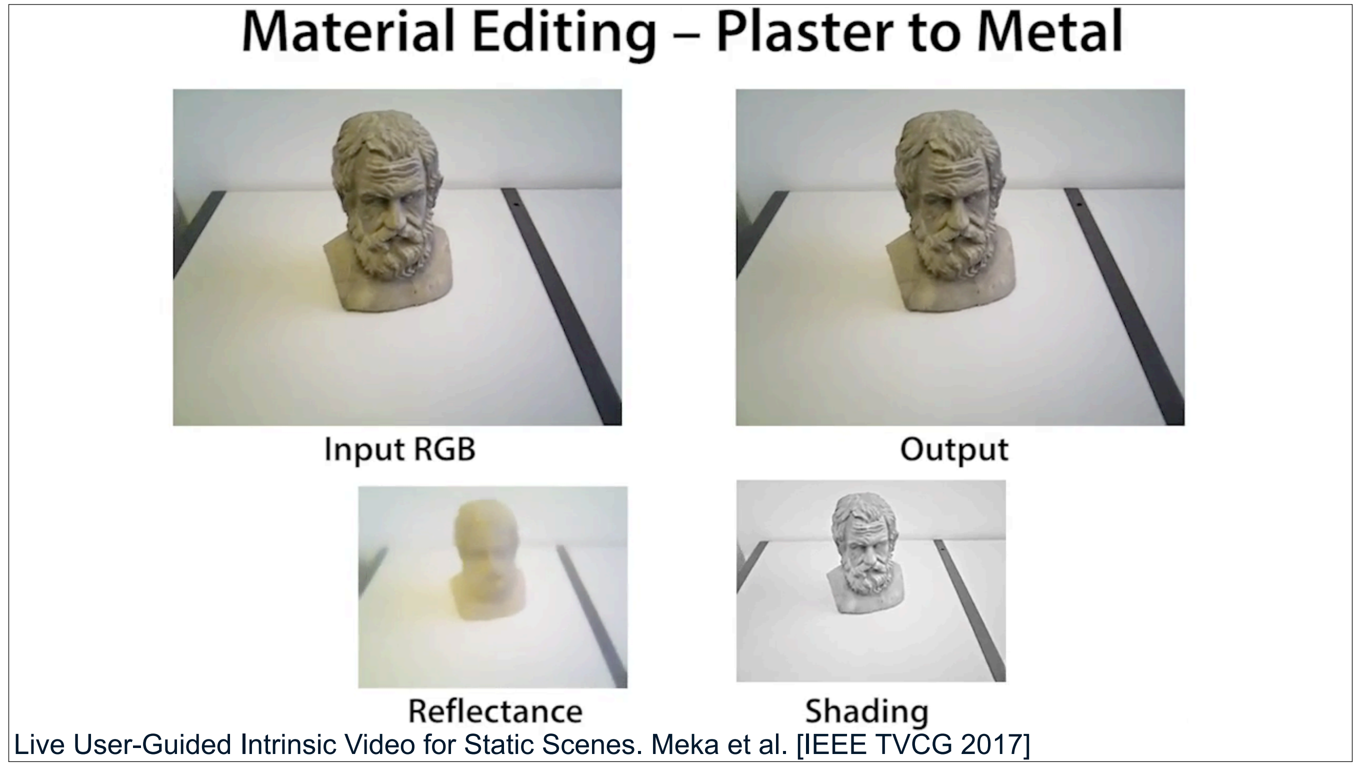


Live User-Guided Intrinsic Video for Static Scenes. Meka et al. [IEEE TVCG 2017]

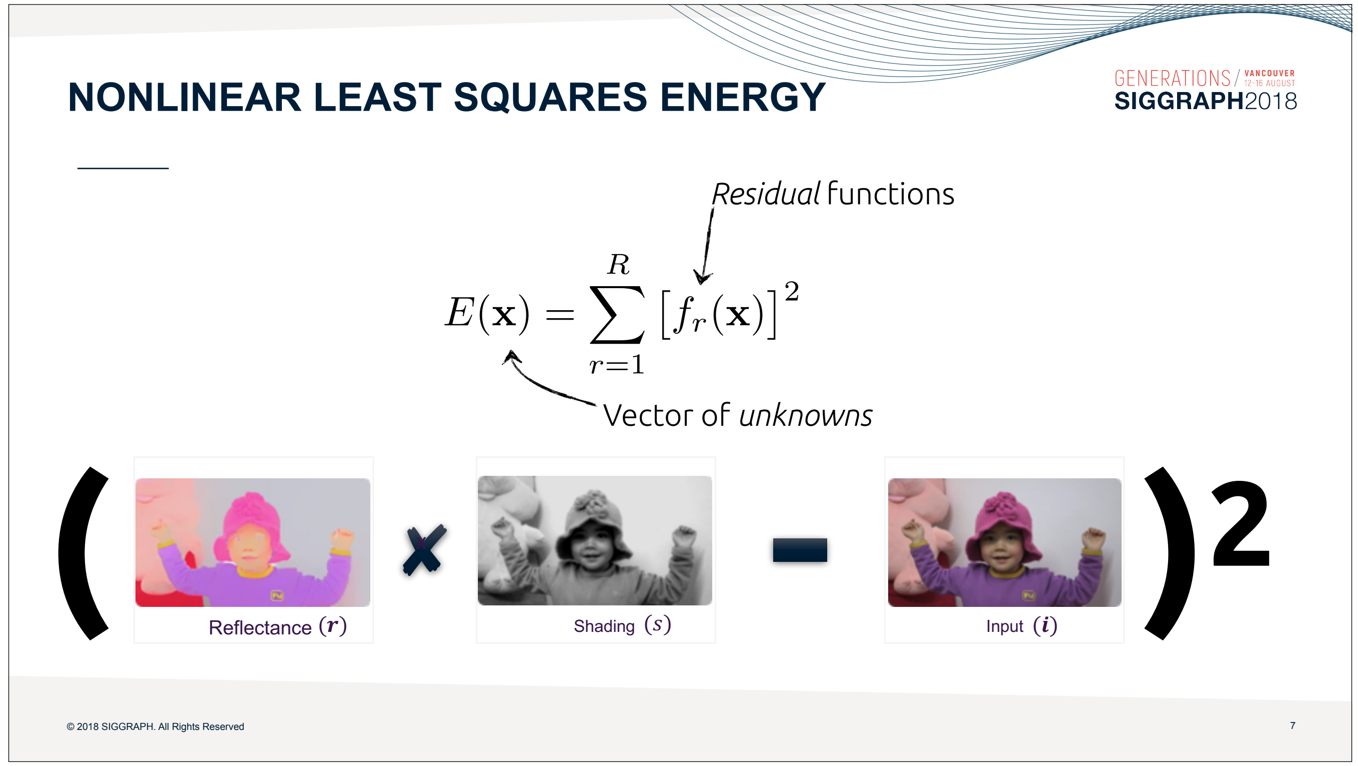
<> You can decompose scenes into geometry and reflectance, and interactively relight the scene.



Or you can change the materials of an object in a live scene.



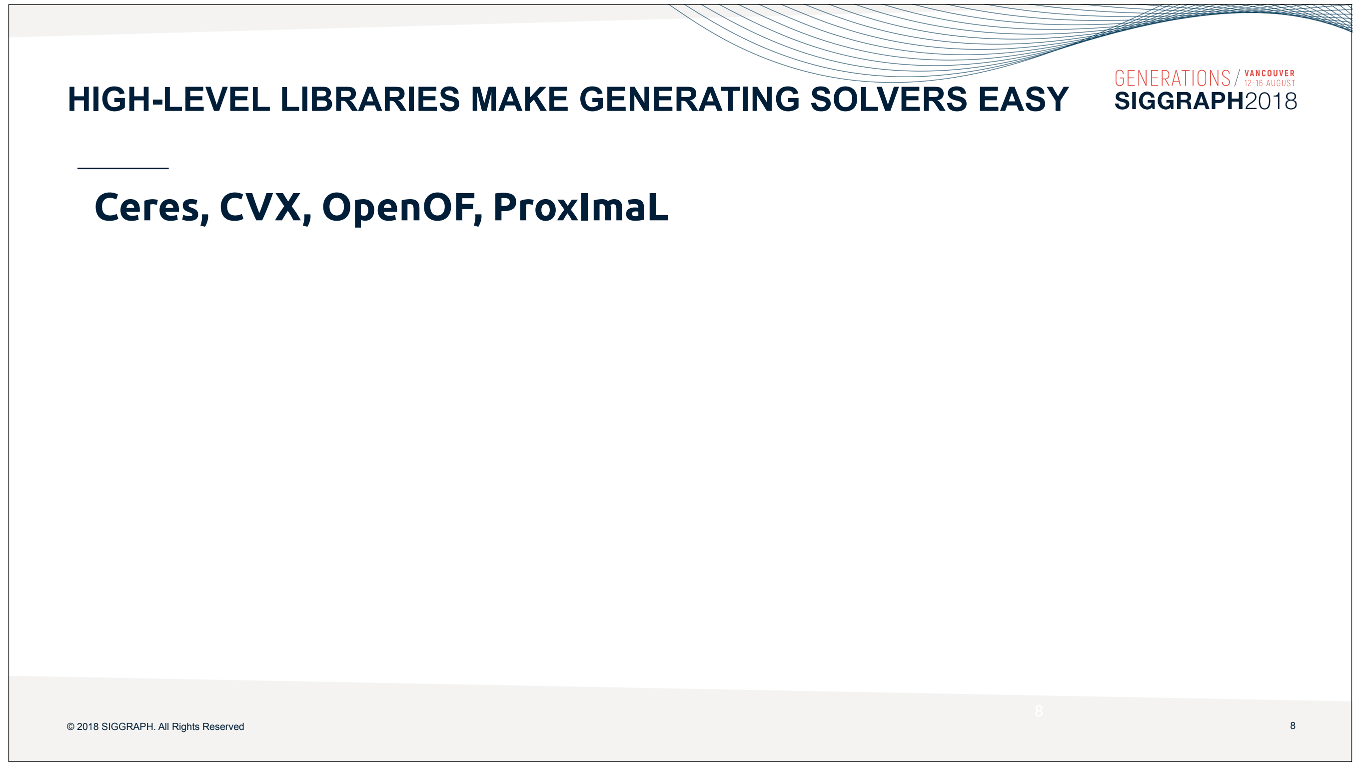
Or you can change the materials of an object in a live scene.



These problems are often described with non-linear least squared energies, which have this formulation:

There is a vector of unknowns, X, which might be pixels in an image or vertices in a graph, and the energy is described use a sum of squared terms, f_r which are arbitrary functions of the unknowns referred to as residuals.

For instance, when doing relighting our unknowns are the reflectance and shading images. The energy is then a residual per pixel that says the product of the reflectance and shading terms should equal the original image.



If you wanted to get a solver running without much effort, you could use off-the-self high-level libraries like Ceres, or CVX

<> These solvers would use techniques like automatic differentiation to construct explicit sparse matrices,
<> and would use a sparse matrix library which would run library routines to, for example, solve pcg on the problem to complete an iteration.
<> This is great from a developer or researchers point of view, since its easy to write new energy functions and try them out
<> Unfortunately they can be orders of magnitude slower than necessary

HIGH-LEVEL LIBRARIES MAKE GENERATING SOLVERS EASY

Ceres, CVX, OpenOF, ProxImaL

- CPU Autodiff to construct sparse matrix

If you wanted to get a solver running without much effort, you could use off-the-self high-level libraries like Ceres, or CVX

<> These solvers would use techniques like automatic differentiation to construct explicit sparse matrices,
<> and would use a sparse matrix library which would run library routines to, for example, solve pcg on the problem to complete an iteration.
<> This is great from a developer or researchers point of view, since its easy to write new energy functions and try them out
<> Unfortunately they can be orders of magnitude slower than necessary

HIGH-LEVEL LIBRARIES MAKE GENERATING SOLVERS EASY

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Ceres, CVX, OpenOF, ProxImaL

- CPU Autodiff to construct sparse matrix
- Hand off to sparse-matrix library (e.g. cuSparse) to run PCG loop

If you wanted to get a solver running without much effort, you could use off-the-self high-level libraries like Ceres, or CVX

<> These solvers would use techniques like automatic differentiation to construct explicit sparse matrices,
<> and would use a sparse matrix library which would run library routines to, for example, solve pcg on the problem to complete an iteration.
<> This is great from a developer or researchers point of view, since its easy to write new energy functions and try them out
<> Unfortunately they can be orders of magnitude slower than necessary

HIGH-LEVEL LIBRARIES MAKE GENERATING SOLVERS EASY

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Ceres, CVX, OpenOF, ProxImaL

- CPU Autodiff to construct sparse matrix
- Hand off to sparse-matrix library (e.g. cuSparse) to run PCG loop

(+) easy to write

If you wanted to get a solver running without much effort, you could use off-the-self high-level libraries like Ceres, or CVX

<> These solvers would use techniques like automatic differentiation to construct explicit sparse matrices,
<> and would use a sparse matrix library which would run library routines to, for example, solve pcg on the problem to complete an iteration.
<> This is great from a developer or researchers point of view, since its easy to write new energy functions and try them out
<> Unfortunately they can be orders of magnitude slower than necessary

HIGH-LEVEL LIBRARIES MAKE GENERATING SOLVERS EASY

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Ceres, CVX, OpenOF, ProxImaL

- CPU Autodiff to construct sparse matrix
- Hand off to sparse-matrix library (e.g. cuSparse) to run PCG loop

(+) easy to write

(-) significantly slower than optimal

If you wanted to get a solver running without much effort, you could use off-the-self high-level libraries like Ceres, or CVX

<> These solvers would use techniques like automatic differentiation to construct explicit sparse matrices,
<> and would use a sparse matrix library which would run library routines to, for example, solve pcg on the problem to complete an iteration.
<> This is great from a developer or researchers point of view, since its easy to write new energy functions and try them out
<> Unfortunately they can be orders of magnitude slower than necessary

HANDWRITE TO REACH REALTIME

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Per-Energy Custom GPU Solver[1-6]

[1] Real-time shading-based refinement for consumer depth cameras Wu et al. Transactions on Graphics 2014
[2] Real-time non-rigid reconstruction using an RGB-D camera Zollhofer et al. Transactions on Graphics 2014
[3] Shading-based refinement on volumetric signed distance functions Zollhofer et al. Transactions on Graphics 2015
[4] Real-time expression transfer for facial reenactment. Thies et al. Transactions on Graphics 2015
[5] Face2Face: Real-time face capture and reenactme of RGB videos. Thies et al. CVPR 2016
[6] VolumeDeform: Real-time volumetric non-rigid reconstruction. Innmann et al. ECCV 2016

In contrast, many real-time techniques, such as written by Wu, Zoll hopher, Teas, Innman and others rely instead on hand-written GPU solvers.

<> These solvers exploit the structure in the images or meshes.

<> They work matrix free, re-constructing needed values on the fly during PCG.

<> And because of this, handwritten derivatives are calculated inside the solvers inner loop.

<> This hand-written approach is incredibly fast, but its is also incredibly hard to get right.

HANDWRITE TO REACH REALTIME

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Per-Energy Custom GPU Solver[1-6]

- Exploit image/mesh structure

[1] Real-time shading-based refinement for consumer depth cameras Wu et al. Transactions on Graphics 2014
[2] Real-time non-rigid reconstruction using an RGB-D camera Zollhofer et al. Transactions on Graphics 2014
[3] Shading-based refinement on volumetric signed distance functions Zollhofer et al. Transactions on Graphics 2015
[4] Real-time expression transfer for facial reenactment. Thies et al. Transactions on Graphics 2015
[5] Face2Face: Real-time face capture and reenactme of RGB videos. Thies et al. CVPR 2016
[6] VolumeDeform: Real-time volumetric non-rigid reconstruction. Innmann et al. ECCV 2016

In contrast, many real-time techniques, such as written by Wu, Zoll hopher, Teas, Innman and others rely instead on hand-written GPU solvers.

<> These solvers exploit the structure in the images or meshes.

<> They work matrix free, re-constructing needed values on the fly during PCG.

<> And because of this, handwritten derivatives are calculated inside the solvers inner loop.

<> This hand-written approach is incredibly fast, but its is also incredibly hard to get right.

Per-Energy Custom GPU Solver[1-6]

- Exploit image/mesh structure
- Matrix-free -- Never construct or store system matrices

[1] Real-time shading-based refinement for consumer depth cameras Wu et al. Transactions on Graphics 2014
[2] Real-time non-rigid reconstruction using an RGB-D camera Zollhofer et al. Transactions on Graphics 2014
[3] Shading-based refinement on volumetric signed distance functions Zollhofer et al. Transactions on Graphics 2015
[4] Real-time expression transfer for facial reenactment. Thies et al. Transactions on Graphics 2015
[5] Face2Face: Real-time face capture and reenactme of RGB videos. Thies et al. CVPR 2016
[6] VolumeDeform: Real-time volumetric non-rigid reconstruction. Innmann et al. ECCV 2016

In contrast, many real-time techniques, such as written by Wu, Zoll hopher, Teas, Innman and others rely instead on hand-written GPU solvers.

<> These solvers exploit the structure in the images or meshes.

<> They work matrix free, re-constructing needed values on the fly during PCG.

<> And because of this, handwritten derivatives are calculated inside the solvers inner loop.

<> This hand-written approach is incredibly fast, but its is also incredibly hard to get right.

HANDWRITE TO REACH REALTIME

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Per-Energy Custom GPU Solver[1-6]

- Exploit image/mesh structure
- Matrix-free -- Never construct or store system matrices
- Handwritten derivatives are inside the solver kernel

[1] Real-time shading-based refinement for consumer depth cameras Wu et al. Transactions on Graphics 2014
[2] Real-time non-rigid reconstruction using an RGB-D camera Zollhofer et al. Transactions on Graphics 2014
[3] Shading-based refinement on volumetric signed distance functions Zollhofer et al. Transactions on Graphics 2015
[4] Real-time expression transfer for facial reenactment. Thies et al. Transactions on Graphics 2015
[5] Face2Face: Real-time face capture and reenactme of RGB videos. Thies et al. CVPR 2016
[6] VolumeDeform: Real-time volumetric non-rigid reconstruction. Innmann et al. ECCV 2016

In contrast, many real-time techniques, such as written by Wu, Zoll hopher, Teas, Innman and others rely instead on hand-written GPU solvers.

<> These solvers exploit the structure in the images or meshes.

<> They work matrix free, re-constructing needed values on the fly during PCG.

<> And because of this, handwritten derivatives are calculated inside the solvers inner loop.

<> This hand-written approach is incredibly fast, but its is also incredibly hard to get right.

Per-Energy Custom GPU Solver[1-6]

- Exploit image/mesh structure
- Matrix-free -- Never construct or store system matrices
- Handwritten derivatives are inside the solver kernel

(+) significantly faster, by orders of magnitude

(-) incredibly hard to write correctly

[1] Real-time shading-based refinement for consumer depth cameras Wu et al. Transactions on Graphics 2014
[2] Real-time non-rigid reconstruction using an RGB-D camera Zollhofer et al. Transactions on Graphics 2014
[3] Shading-based refinement on volumetric signed distance functions Zollhofer et al. Transactions on Graphics 2015
[4] Real-time expression transfer for facial reenactment. Thies et al. Transactions on Graphics 2015
[5] Face2Face: Real-time face capture and reenactme of RGB videos. Thies et al. CVPR 2016
[6] VolumeDeform: Real-time volumetric non-rigid reconstruction. Innmann et al. ECCV 2016

In contrast, many real-time techniques, such as written by Wu, Zoll hopher, Teas, Innman and others rely instead on hand-written GPU solvers.

<> These solvers exploit the structure in the images or meshes.

<> They work matrix free, re-constructing needed values on the fly during PCG.

<> And because of this, handwritten derivatives are calculated inside the solvers inner loop.

<> This hand-written approach is incredibly fast, but its is also incredibly hard to get right.

<> This is code for a real problem. The CUDA code to calculate the energy is pretty concise and completely represents the problem.

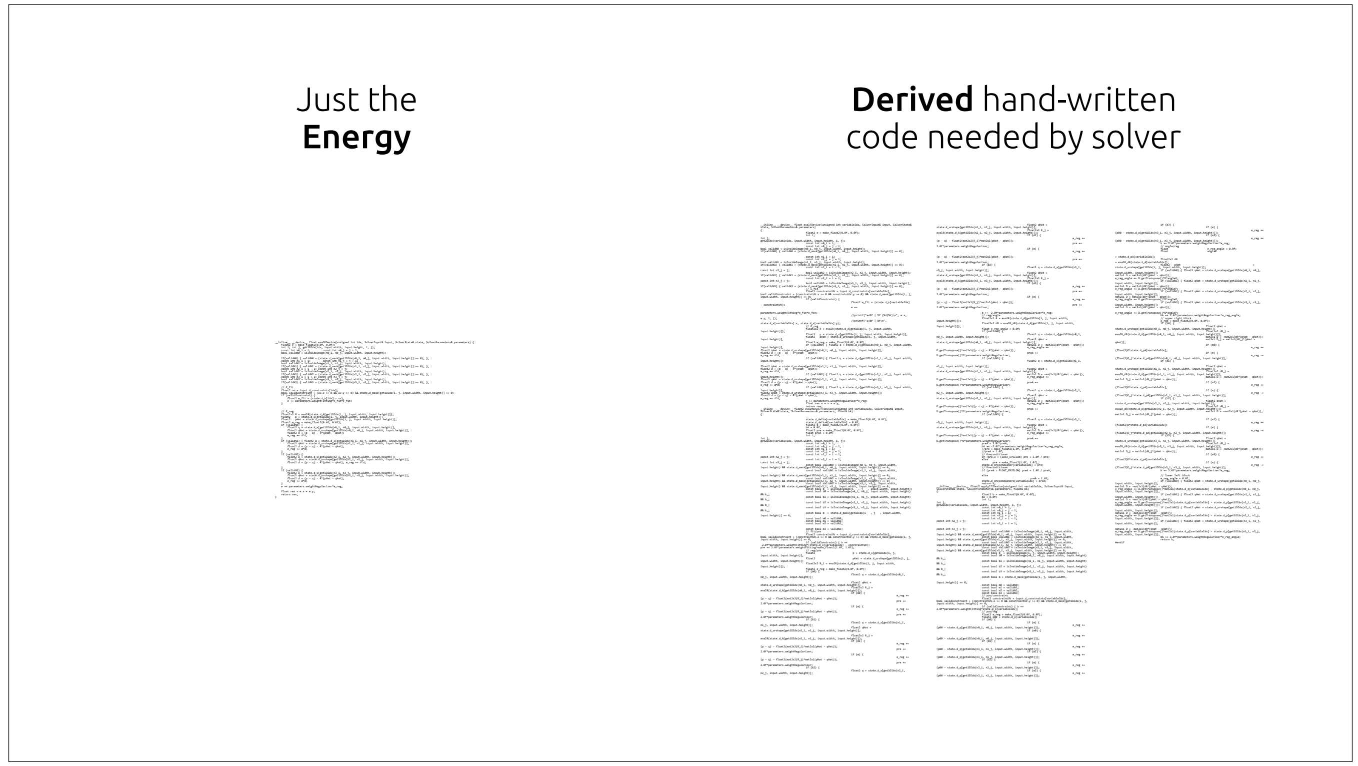
<> But to get high-performance, we need all of this code on the right to calculate these in-place matrix products. Writing it by hand is hard, it requires calculus and getting boundary conditions right. The solver code and the energy code is woven together in a complicated way. We never got it right on the first try and bugs would stay in the code for a really long time.

Just the
Energy

```
__inline__ __device__ float evalFDevice(unsigned int idx, SolverInput& input, SolverState& state, SolverParameters& parameters) {
    float2 e = make_float2(0.0f, 0.0f);
    int i; int j; get2DIdx(idx, input.width, input.height, i, j);
    const int n0_i = i;        const int n0_j = j - 1;
    bool validN0 = isInsideImage(n0_i, n0_j, input.width, input.height);

    if(validN0) { validN0 = (state.d_mask[get1DIdx(n0_i, n0_j, input.width, input.height)] == 0); };
    const int n1_i = i;        const int n1_j = j + 1;
    bool validN1 = isInsideImage(n1_i, n1_j, input.width, input.height);
    if(validN1) { validN1 = (state.d_mask[get1DIdx(n1_i, n1_j, input.width, input.height)] == 0); };
    const int n2_i = i - 1; const int n2_j = j;
    bool validN2 = isInsideImage(n2_i, n2_j, input.width, input.height);
    if(validN2) { validN2 = (state.d_mask[get1DIdx(n2_i, n2_j, input.width, input.height)] == 0); };
    const int n3_i = i + 1; const int n3_j = j;
    bool validN3 = isInsideImage(n3_i, n3_j, input.width, input.height);
    if(validN3) { validN3 = (state.d_mask[get1DIdx(n3_i, n3_j, input.width, input.height)] == 0); };

    // E_fit
    float2 uv = input.d_constraints[idx];
    bool validConstraint = (uv.x >= 0 && uv.y >= 0) && state.d_mask[get1DIdx(i, j, input.width, input.height)] == 0;
    if (validConstraint) {
        float2 e_fit = (state.d_x[idx] - uv);
        e += parameters.weightFitting*e_fit*e_fit;
    }


    // E_reg
    float2x2 R = evalR(state.d_A[get1DIdx(i, j, input.width, input.height)]);
    float2   p = state.d_x[get1DIdx(i, j, input.width, input.height)];
    float2   pHat = state.d_urshape[get1DIdx(i, j, input.width, input.height)];
    float2 e_reg = make_float2(0.0f, 0.0f);
    if (validN0) {
        float2 q = state.d_x[get1DIdx(n0_i, n0_j, input.width, input.height)];
        float2 qHat = state.d_urshape[get1DIdx(n0_i, n0_j, input.width, input.height)];
        float2 d = (p - q) - R*(pHat - qHat);
        e_reg += d*d;
    }
    if (validN1) { float2 q = state.d_x[get1DIdx(n1_i, n1_j, input.width, input.height)];
        float2 qHat = state.d_urshape[get1DIdx(n1_i, n1_j, input.width, input.height)];
        float2 d = (p - q) - R*(pHat - qHat);
        e_reg += d*d;
    }
    if (validN2) {
        float2 q = state.d_x[get1DIdx(n2_i, n2_j, input.width, input.height)];
        float2 qHat = state.d_urshape[get1DIdx(n2_i, n2_j, input.width, input.height)];
        float2 d = (p - q) - R*(pHat - qHat); e_reg += d*d;

    }
    if (validN3) {
        float2 q = state.d_x[get1DIdx(n3_i, n3_j, input.width, input.height)];
        float2 qHat = state.d_urshape[get1DIdx(n3_i, n3_j, input.width, input.height)];
        float2 d = (p - q) - R*(pHat - qHat);
        e_reg += d*d;
    }
    e += parameters.weightRegularizer*e_reg;

    float res = e.x + e.y;
    return res;
}
```

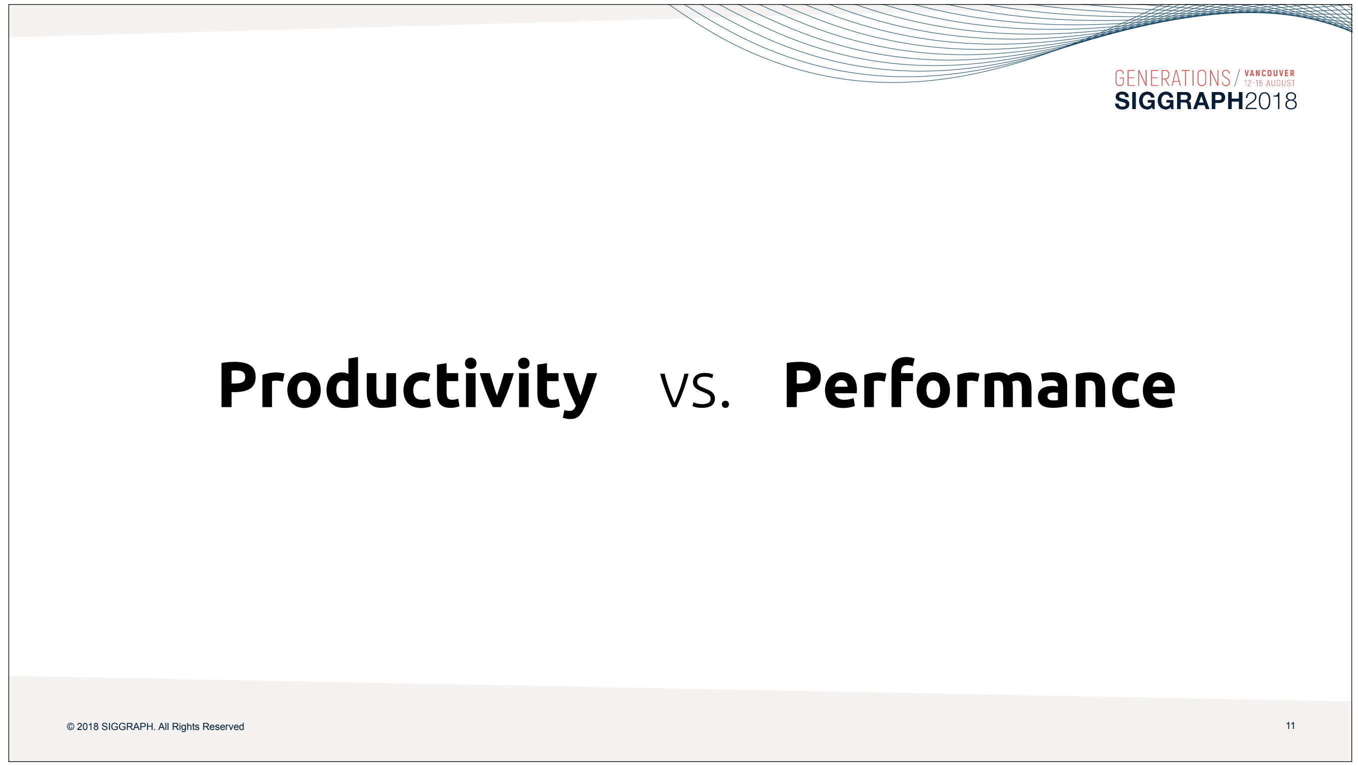
<> This is code for a real problem. The CUDA code to calculate the energy is pretty concise and completely represents the problem.

<> But to get high-performance, we need all of this code on the right to calculate these in-place matrix products. Writing it by hand is hard, it requires calculus and getting boundary conditions right. The solver code and the energy code is woven together in a complicated way. We never got it right on the first try and bugs would stay in the code for a really long time.



<> This is code for a real problem. The CUDA code to calculate the energy is pretty concise and completely represents the problem.

<> But to get high-performance, we need all of this code on the right to calculate these in-place matrix products. Writing it by hand is hard, it requires calculus and getting boundary conditions right. The solver code and the energy code is woven together in a complicated way. We never got it right on the first try and bugs would stay in the code for a really long time.



In these existing approaches you have to trade productivity for performance.

<> We set out to build a system that provides both.

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

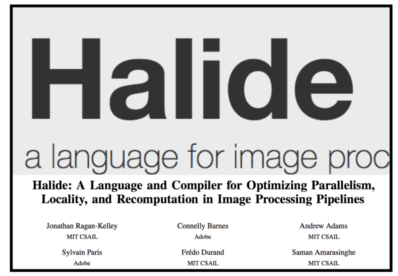
Productivity **and** Performance

In these existing approaches you have to trade productivity for performance.

<> We set out to build a system that provides both.

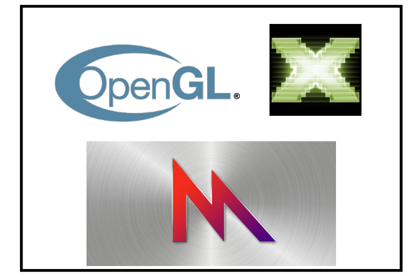
APPROACH: DSLs

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018



Imaging

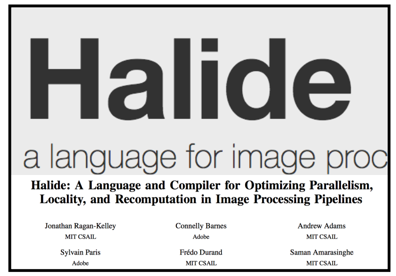
Simulation



Rendering

In other areas of graphics, such as imaging, physical simulation, or rendering, domain-specific languages have been used to express high-level programs, but still generate high-performance machine code.

APPROACH: DSLs


Halide
a language for image proc
Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

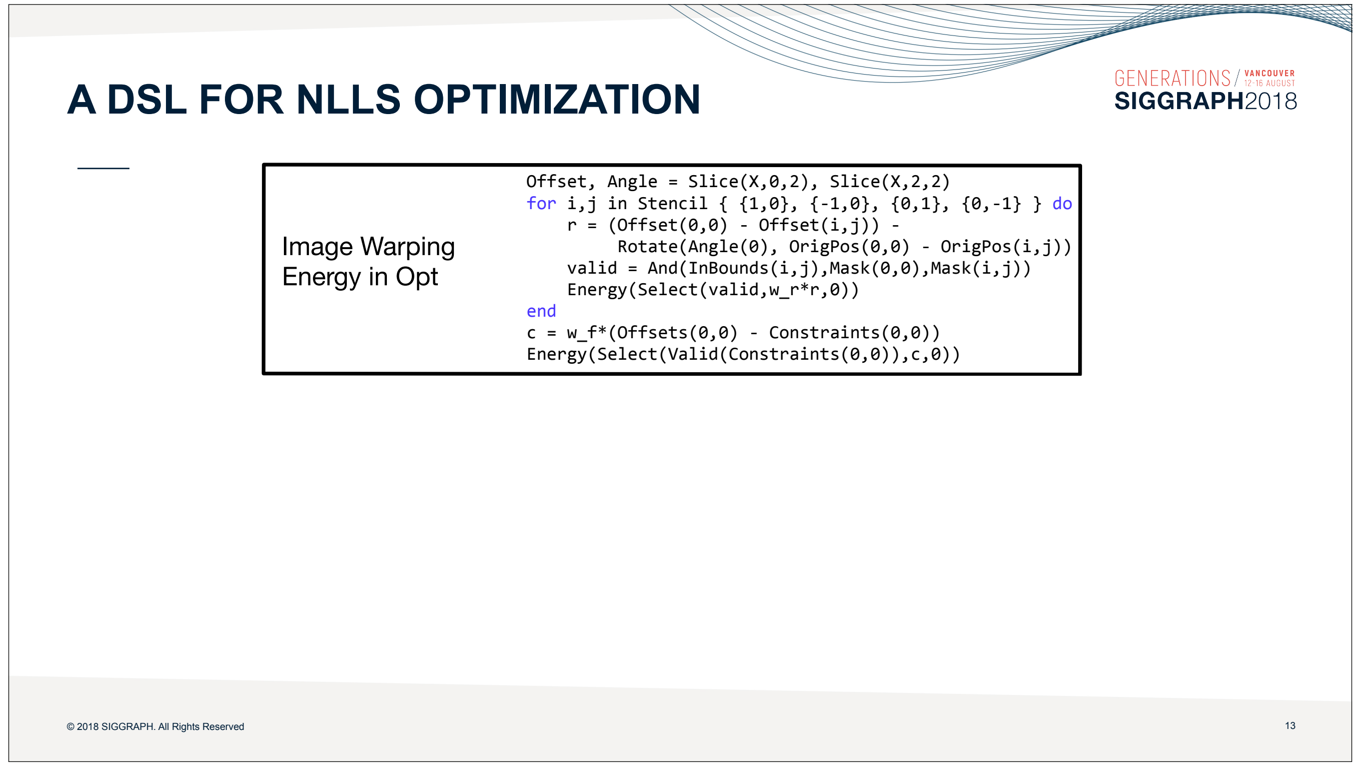
Jonathan Ragan-Kelley MIT CSAIL	Connelly Barnes Adobe	Andrew Adams MIT CSAIL
Sylvain Paris Adobe	Frédo Durand MIT CSAIL	Saman Amarasinghe MIT CSAIL

Imaging

Simulation

Rendering

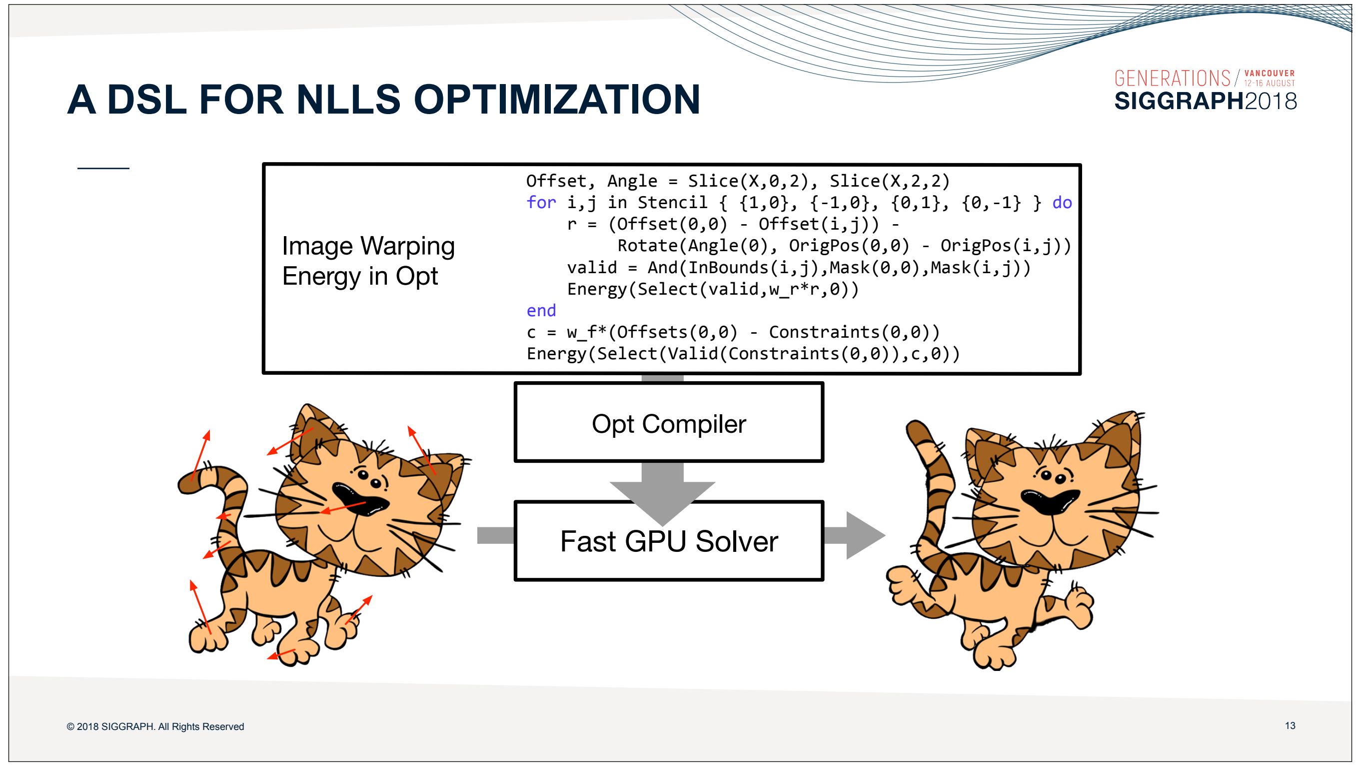
In other areas of graphics, such as imaging, physical simulation, or rendering, domain-specific languages have been used to express high-level programs, but still generate high-performance machine code.



Opt applies this domain-specific language approach to these optimizations problems. It takes the high-level form of the energy,

<> and automatically produces a real-time GPU solver without all the tedious work.

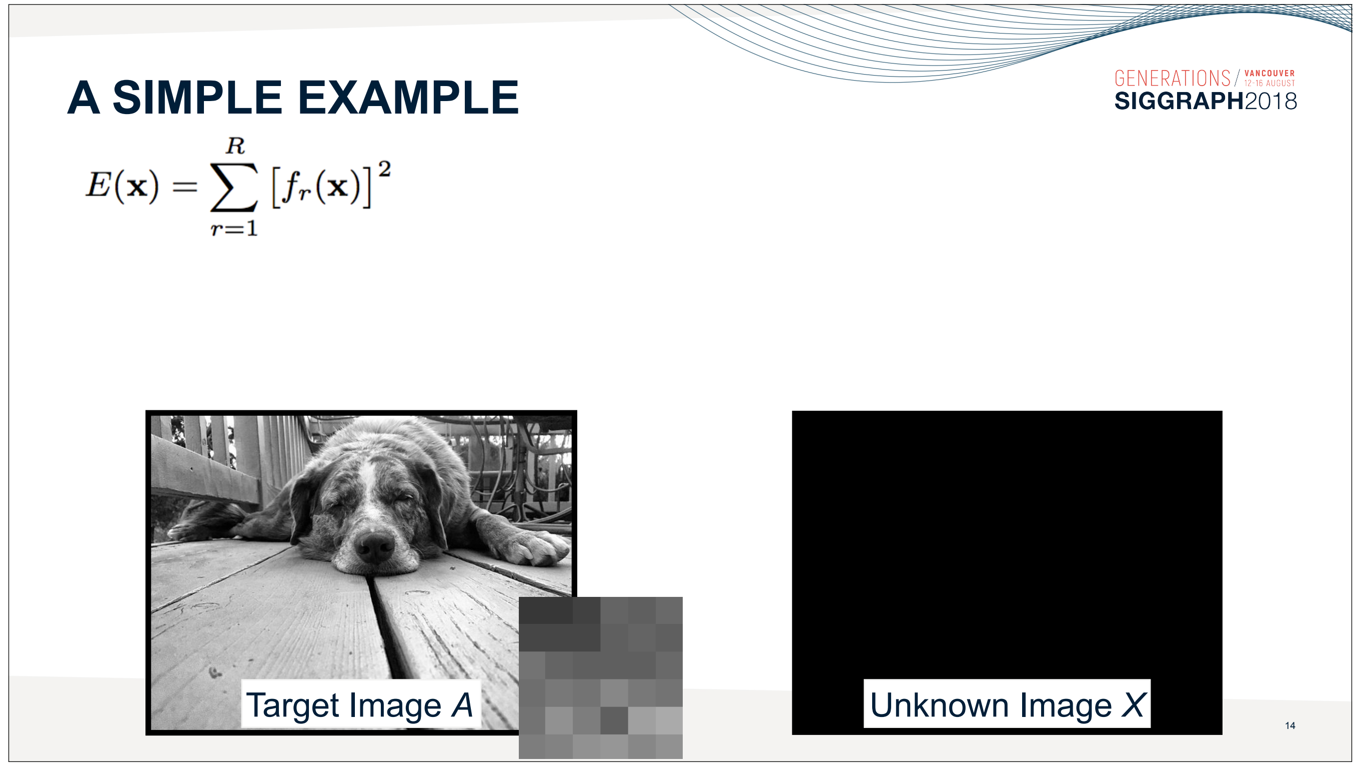
In this talk, we'll first show you what it looks like to write an energy function in Opt, and then walk you through how Opt automatically constructs a gauss-newton style solver from it.



Opt applies this domain-specific language approach to these optimizations problems. It takes the high-level form of the energy,

<> and automatically produces a real-time GPU solver without all the tedious work.

In this talk, we'll first show you what it looks like to write an energy function in Opt, and then walk you through how Opt automatically constructs a gauss-newton style solver from it.



Let’s look at a simple Laplacian smoothing problem.

We’ll have some notation for our energy term: A is the target input image, and X is the unknown image, which we are trying to find.

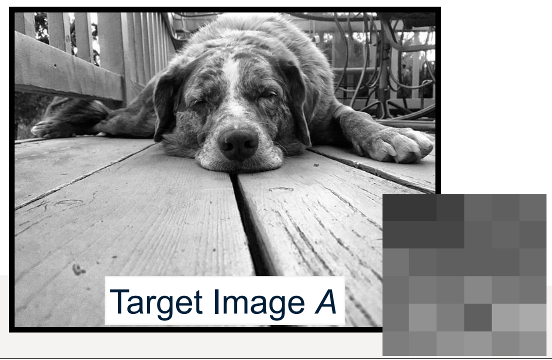
<> We can start out with a fitting term, minimizes the difference between X and the original image.

<> Now we can add regularization terms that penalizing the difference between each pixel and its neighbors, blurring the image.

Image from https://www.flickr.com/photos/cogdog/39525949350/ (public domain)

A SIMPLE EXAMPLE

$$E(\mathbf{x}) = \sum_{r=1}^{R} \left[f_r(\mathbf{x})\right]^2 = \boxed{\sum_{i=0}^{H} \sum_{j=0}^{W} (X_{i,j} - A_{i,j})^2 \quad \text{Fitting Term}}$$



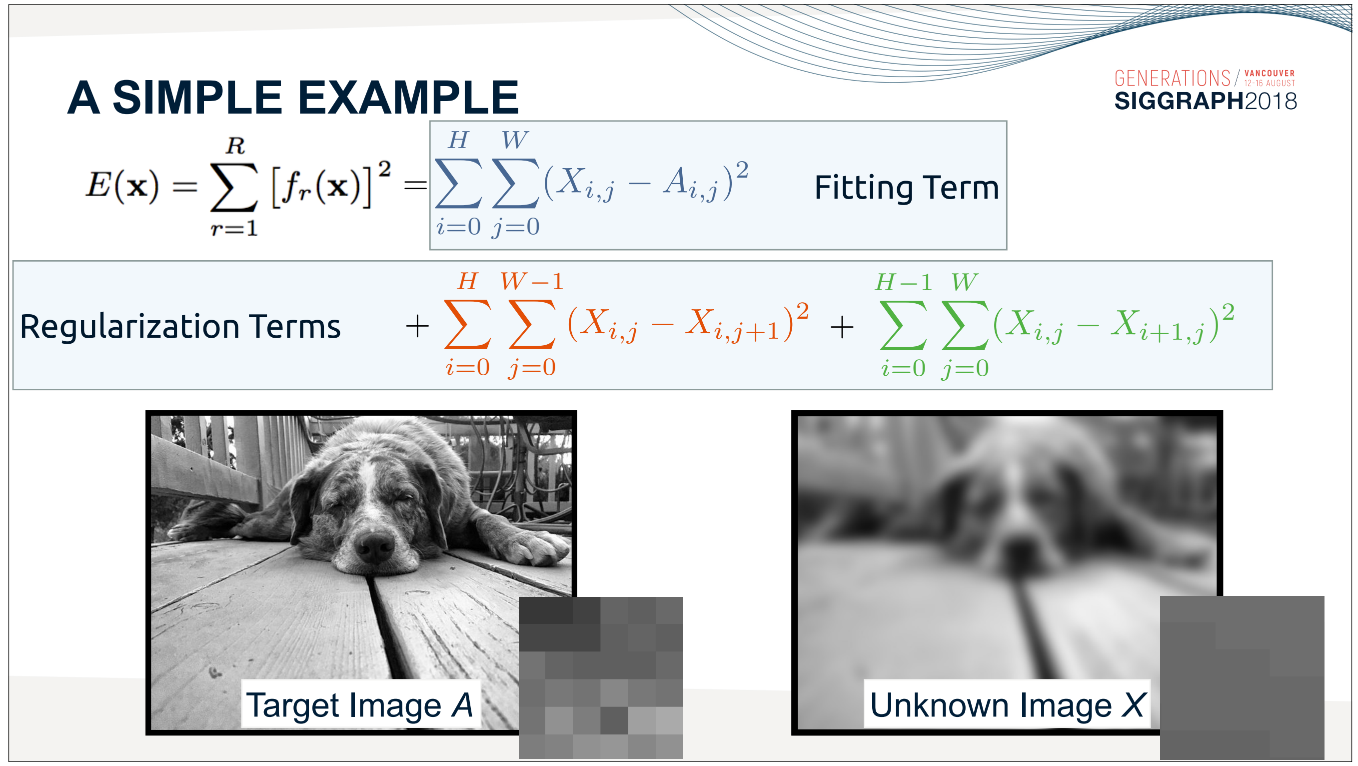
Let's look at a simple Laplacian smoothing problem.

We'll have some notation for our energy term: A is the target input image, and X is the unknown image, which we are trying to find.

<> We can start out with a fitting term, minimizes the difference between X and the original image.

<> Now we can add regularization terms that penalizing the difference between each pixel and its neighbors, blurring the image.

Image from https://www.flickr.com/photos/cogdog/39525949350/ (public domain)



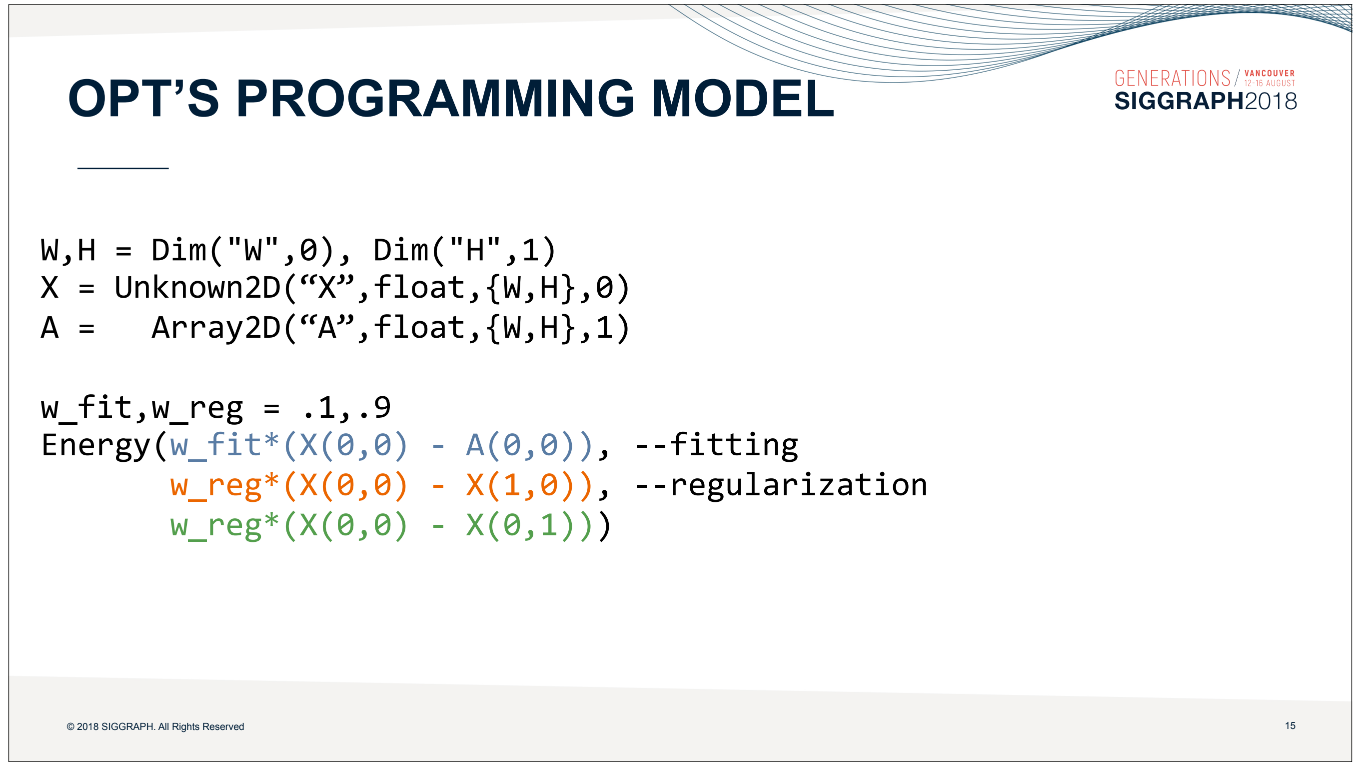
Let’s look at a simple Laplacian smoothing problem.

We’ll have some notation for our energy term: A is the target input image, and X is the unknown image, which we are trying to find.

<> We can start out with a fitting term, minimizes the difference between X and the original image.

<> Now we can add regularization terms that penalizing the difference between each pixel and its neighbors, blurring the image.

Image from https://www.flickr.com/photos/cogdog/39525949350/ (public domain)



Opt allows you to write these energies directly in your problem domain. That is, instead of representing it as a flat list of unknown and residuals, you express the problem in terms of images and meshes.

Here is an Opt version of the Laplacian energy from the last slide.

<> It defines the problem domain it is working on by creating a binding for the original image A, and also an image for the unknown X.

You can have multiple unknown images, and you can also mix images and meshes.

<> Once we have our problem defined in terms of images and meshes, we then define residual energy terms on elements of the domain: this blue term is the same fitting term as the last slide. Note that each term is implicitly squared.

<> These terms are implicitly defined over the entire image like in this illustration.

<> Energies can use a local neighborhood of data using pixel offsets.

OPT'S PROGRAMMING MODEL

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Problem Data,
including Unknowns

```
W,H = Dim("W",0), Dim("H",1)
X = Unknown2D("X",float,{W,H},0)
A =   Array2D("A",float,{W,H},1)

w_fit,w_reg = .1,.9
Energy(w_fit*(X(0,0) - A(0,0)), --fitting
       w_reg*(X(0,0) - X(1,0)), --regularization
       w_reg*(X(0,0) - X(0,1)))
```


Opt allows you to write these energies directly in your problem domain. That is, instead of representing it as a flat list of unknown and residuals, you express the problem in terms of images and meshes.

Here is an Opt version of the Laplacian energy from the last slide.

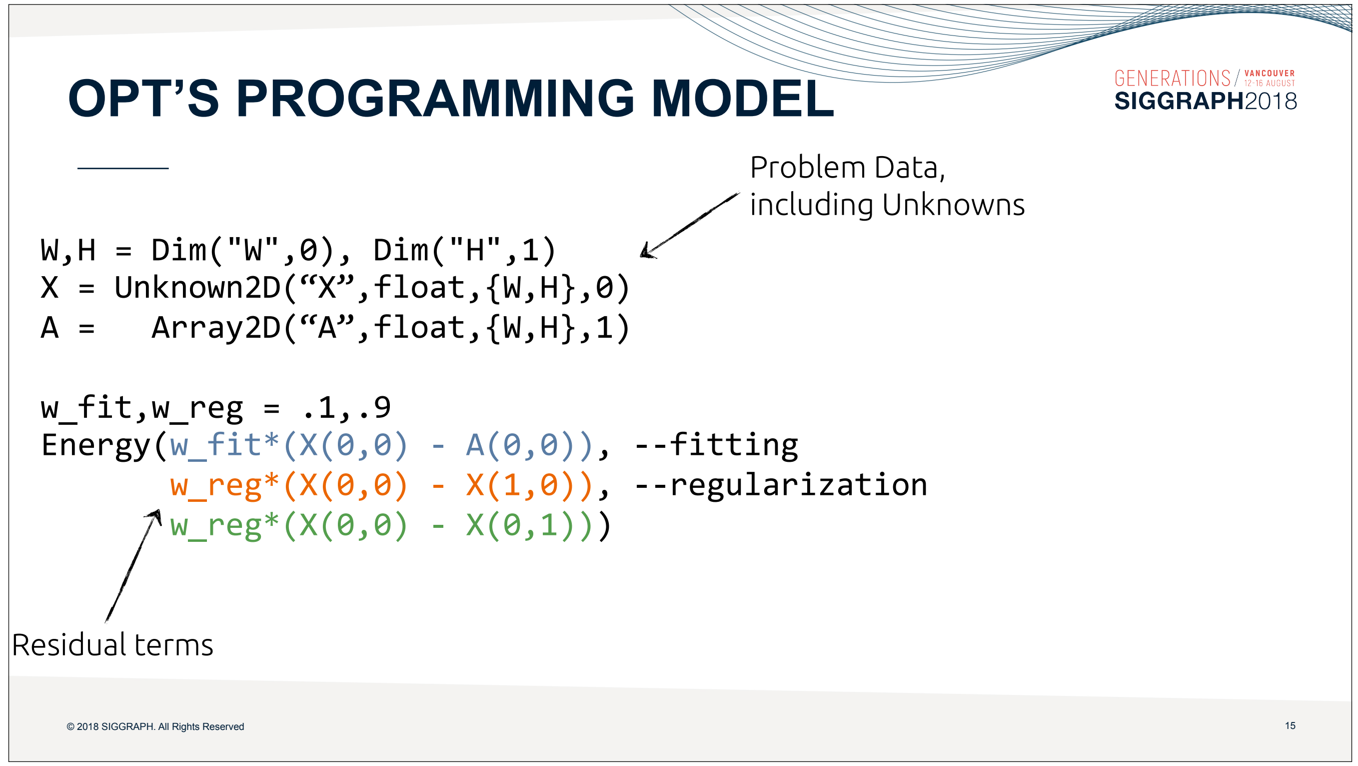
<> It defines the problem domain it is working on by creating a binding for the original image A, and also an image for the unknown X.

You can have multiple unknown images, and you can also mix images and meshes.

<> Once we have our problem defined in terms of images and meshes, we then define residual energy terms on elements of the domain: this blue term is the same fitting term as the last slide. Note that each term is implicitly squared.

<> These terms are implicitly defined over the entire image like in this illustration.

<> Energies can use a local neighborhood of data using pixel offsets.



Opt allows you to write these energies directly in your problem domain. That is, instead of representing it as a flat list of unknown and residuals, you express the problem in terms of images and meshes.

Here is an Opt version of the Laplacian energy from the last slide.

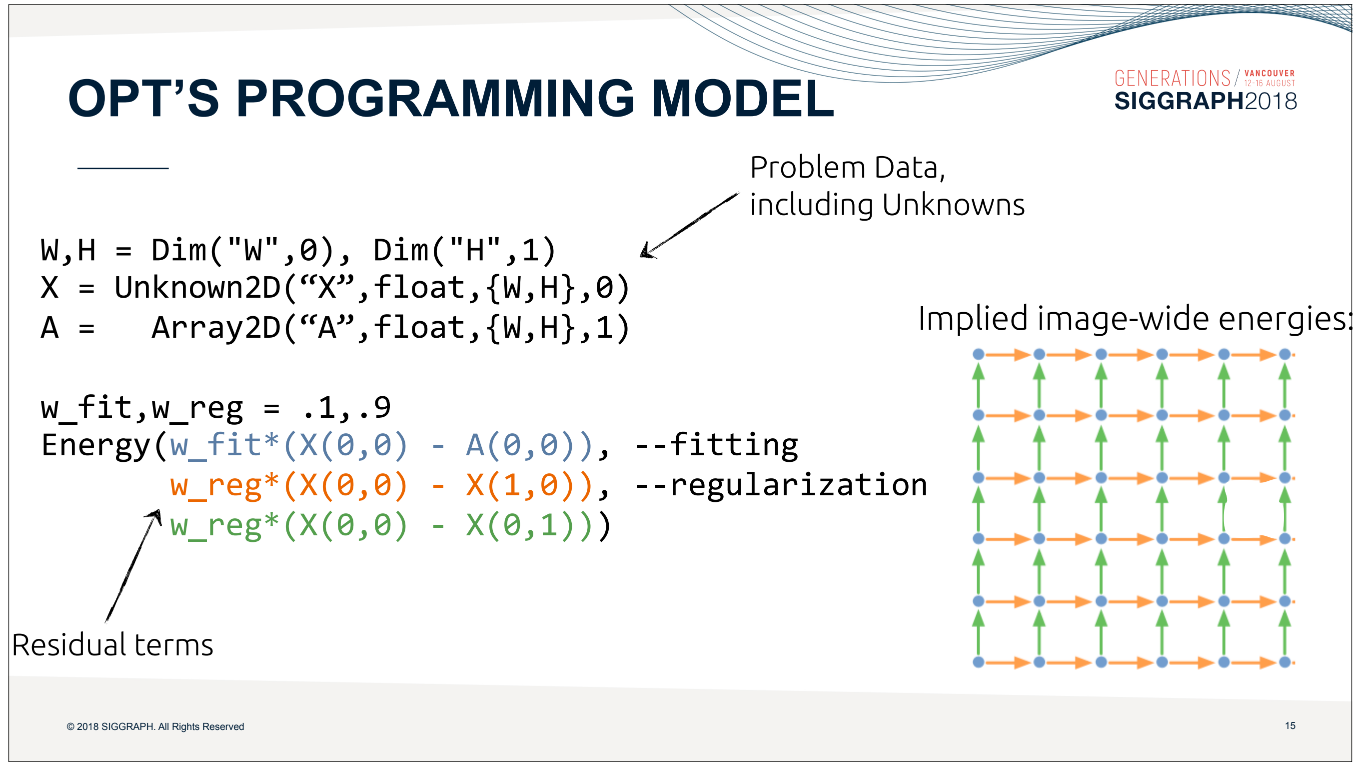
<> It defines the problem domain it is working on by creating a binding for the original image A, and also an image for the unknown X.

You can have multiple unknown images, and you can also mix images and meshes.

<> Once we have our problem defined in terms of images and meshes, we then define residual energy terms on elements of the domain: this blue term is the same fitting term as the last slide. Note that each term is implicitly squared.

<> These terms are implicitly defined over the entire image like in this illustration.

<> Energies can use a local neighborhood of data using pixel offsets.



Opt allows you to write these energies directly in your problem domain. That is, instead of representing it as a flat list of unknown and residuals, you express the problem in terms of images and meshes.

Here is an Opt version of the Laplacian energy from the last slide.

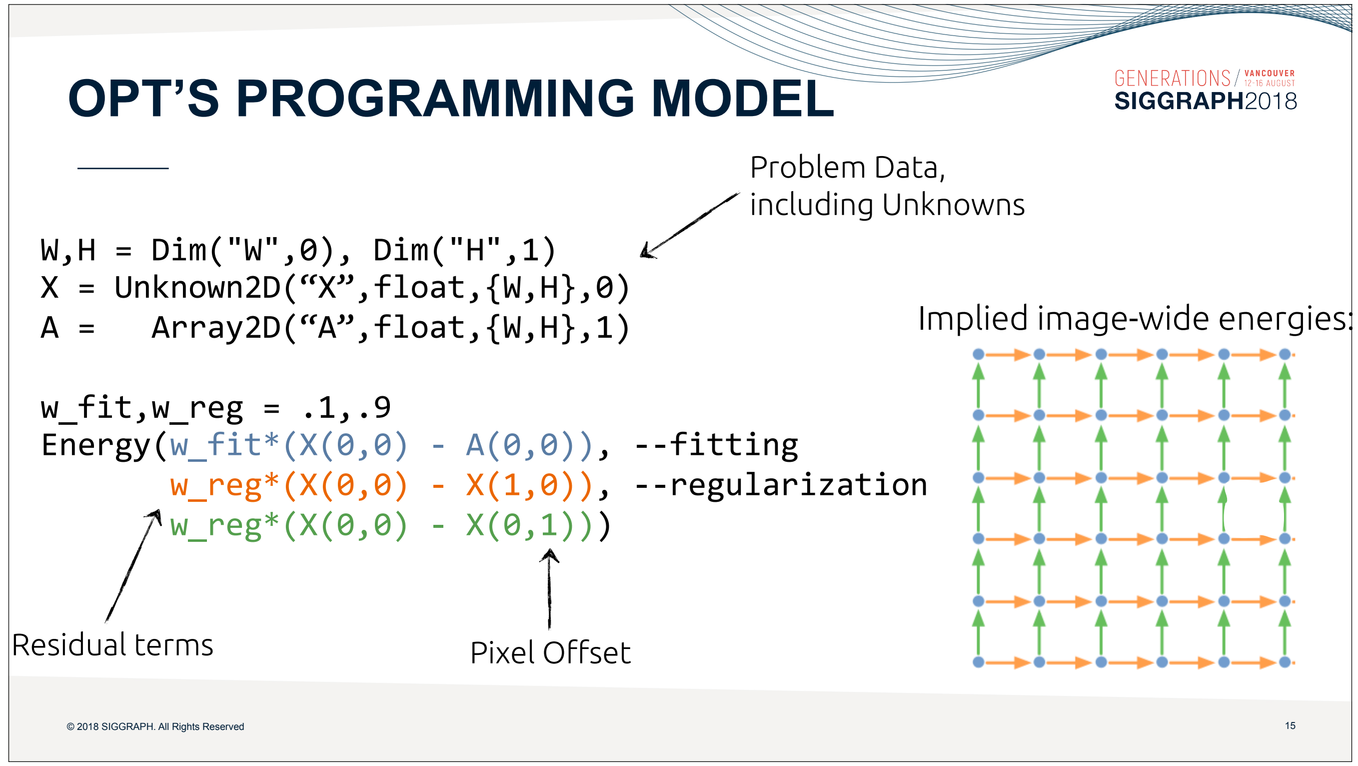
<> It defines the problem domain it is working on by creating a binding for the original image A, and also an image for the unknown X.

You can have multiple unknown images, and you can also mix images and meshes.

<> Once we have our problem defined in terms of images and meshes, we then define residual energy terms on elements of the domain: this blue term is the same fitting term as the last slide. Note that each term is implicitly squared.

<> These terms are implicitly defined over the entire image like in this illustration.

<> Energies can use a local neighborhood of data using pixel offsets.



Opt allows you to write these energies directly in your problem domain. That is, instead of representing it as a flat list of unknown and residuals, you express the problem in terms of images and meshes.

Here is an Opt version of the Laplacian energy from the last slide.

<> It defines the problem domain it is working on by creating a binding for the original image A, and also an image for the unknown X.

You can have multiple unknown images, and you can also mix images and meshes.

<> Once we have our problem defined in terms of images and meshes, we then define residual energy terms on elements of the domain: this blue term is the same fitting term as the last slide. Note that each term is implicitly squared.

<> These terms are implicitly defined over the entire image like in this illustration.

<> Energies can use a local neighborhood of data using pixel offsets.

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

MINIMIZATION USES DERIVATIVE TERMS

$$E(\mathbf{x}) = \sum_{r=1}^{R} \left[f_r(\mathbf{x})\right]^2 = \left\|\mathbf{F}(\mathbf{x})\right\|_2^2, \quad \mathbf{F}(\mathbf{x}) = \left[f_1(\mathbf{x}), \ldots, f_R(\mathbf{x})\right]^T$$

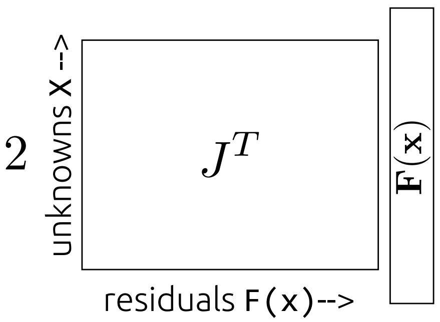
When minimizing one of these energy formulations, you don’t compute the direct energy, instead you typically compute terms based on the derivative of the energy.

<> In the simplest form, you would compute the gradient, which for least squares problems is this matrix product here,
<> you can then use gradient descent to step towards the solution, repeatedly calculating this product

MINIMIZATION USES DERIVATIVE TERMS

$$E(\mathbf{x}) = \sum_{r=1}^{R} \left[f_r(\mathbf{x})\right]^2 = \left\|\mathbf{F}(\mathbf{x})\right\|_2^2, \quad \mathbf{F}(\mathbf{x}) = [f_1(\mathbf{x}), \ldots, f_R(\mathbf{x})]^T$$

$$\nabla E(x) = 2J^T\mathbf{F}(\mathbf{x}) = 2$$



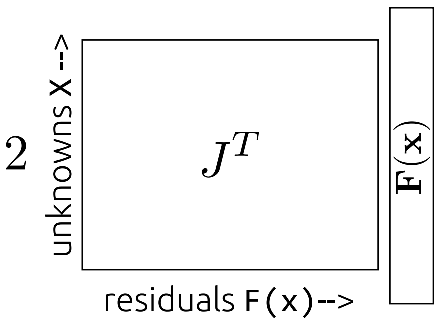
When minimizing one of these energy formulations, you don’t compute the direct energy, instead you typically compute terms based on the derivative of the energy.

<> In the simplest form, you would compute the gradient, which for least squares problems is this matrix product here,
<> you can then use gradient descent to step towards the solution, repeatedly calculating this product

MINIMIZATION USES DERIVATIVE TERMS

$$E(\mathbf{x}) = \sum_{r=1}^{R} \left[f_r(\mathbf{x})\right]^2 = \left\|\mathbf{F}(\mathbf{x})\right\|_2^2, \quad \mathbf{F}(\mathbf{x}) = [f_1(\mathbf{x}), \ldots, f_R(\mathbf{x})]^T$$

$$\nabla E(x) = 2J^T\mathbf{F}(\mathbf{x}) = 2$$



Gradient Descent: Repeat $\mathbf{x} \mathrel{+}= \alpha 2J^T\mathbf{F}(\mathbf{x})$

When minimizing one of these energy formulations, you don’t compute the direct energy, instead you typically compute terms based on the derivative of the energy.

<> In the simplest form, you would compute the gradient, which for least squares problems is this matrix product here,
<> you can then use gradient descent to step towards the solution, repeatedly calculating this product

GAUSS-NEWTON IS NEEDED FOR REAL TIME

In practice, however, we need higher-order solvers, like Gauss Newton.

These can advance to the solution in fewer steps, allowing us to get to real-time.

In these solvers, an optimization step requires solving a linear system with preconditioned conjugate gradient.

<>The inner-most loop requires calculation of this more complicated matrix product involving the Jacobean matrix J. Getting this step fast is key to good performance.

GAUSS-NEWTON IS NEEDED FOR REAL TIME

- Gauss-Newton solves a linear system to find $\Delta \mathbf{x}$ every step
- Use Preconditioned Conjugate Gradient (PCG)

In practice, however, we need higher-order solvers, like Gauss Newton.

These can advance to the solution in fewer steps, allowing us to get to real-time.

In these solvers, an optimization step requires solving a linear system with preconditioned conjugate gradient.

<>The inner-most loop requires calculation of this more complicated matrix product involving the Jacobean matrix J. Getting this step fast is key to good performance.

GAUSS-NEWTON IS NEEDED FOR REAL TIME

GENERATIONS / VANCOUVER 12-16 AUGUST SIGGRAPH2018

- Gauss-Newton solves a linear system to find $\Delta\mathbf{x}$ every step
- Use Preconditioned Conjugate Gradient (PCG)

Perf. critical inner loop does repeated calculations of:

$$2J^TJ\mathbf{p} = 2\,\boxed{J^T}\,\boxed{J}\,\boxed{\mathbf{p}}$$

<—unknowns X

In practice, however, we need higher-order solvers, like Gauss Newton.

These can advance to the solution in fewer steps, allowing us to get to real-time.

In these solvers, an optimization step requires solving a linear system with preconditioned conjugate gradient.

<>The inner-most loop requires calculation of this more complicated matrix product involving the Jacobean matrix J. Getting this step fast is key to good performance.

OPT IS FAST BECAUSE IT EXPLOITS THE STRUCTURE OF THE PROBLEM DOMAIN

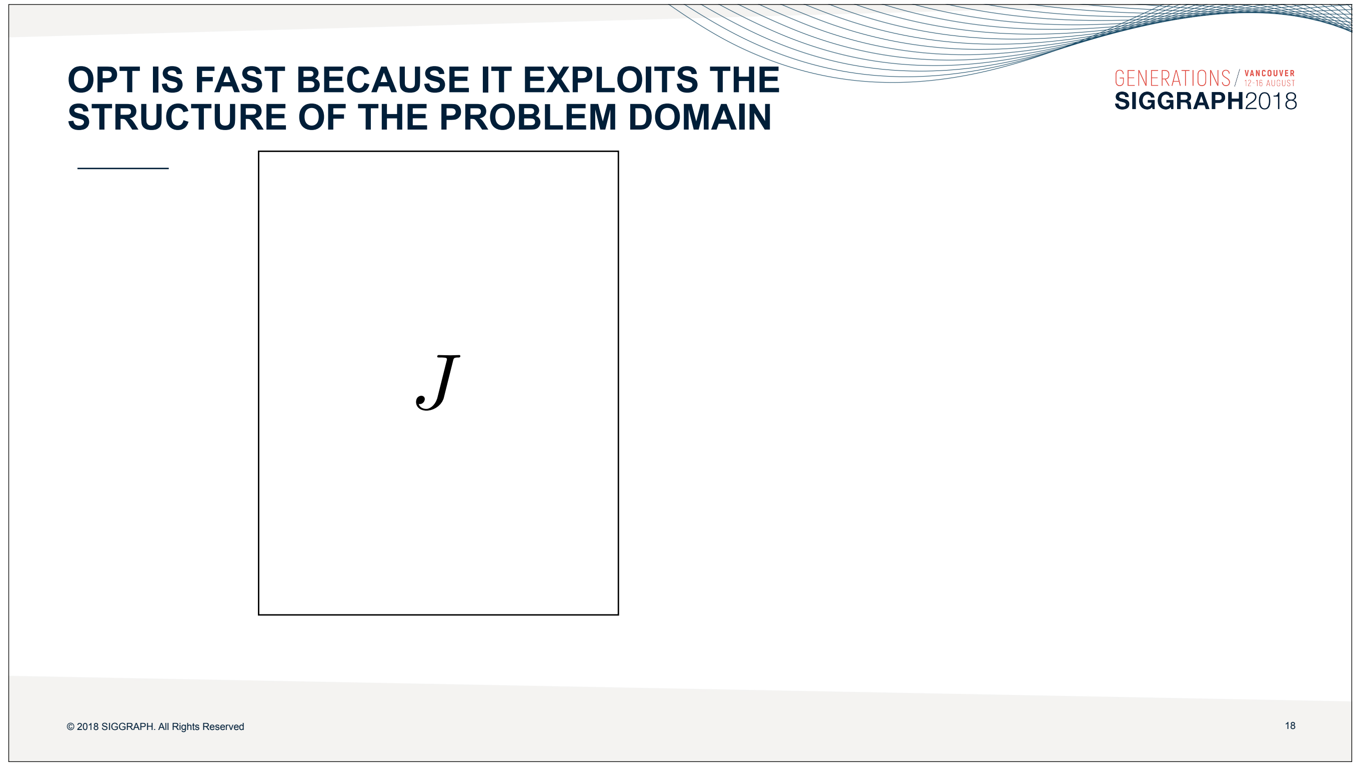
GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Typical solvers would explicitly construct the J matrix, using a sparse matrix format like compressed sparse row, throwing away structure.

The reason Opt can solve so much faster is because it preserves and exploits this structure of the images and meshes that we define the problem on. We can see this by looking at J, the jacobian matrix

<> The Jacobian matrix J contain the partial derivatives of each residual with each unknown.

<> For a mesh-based domain you might have unknowns that represent coordinates in space.

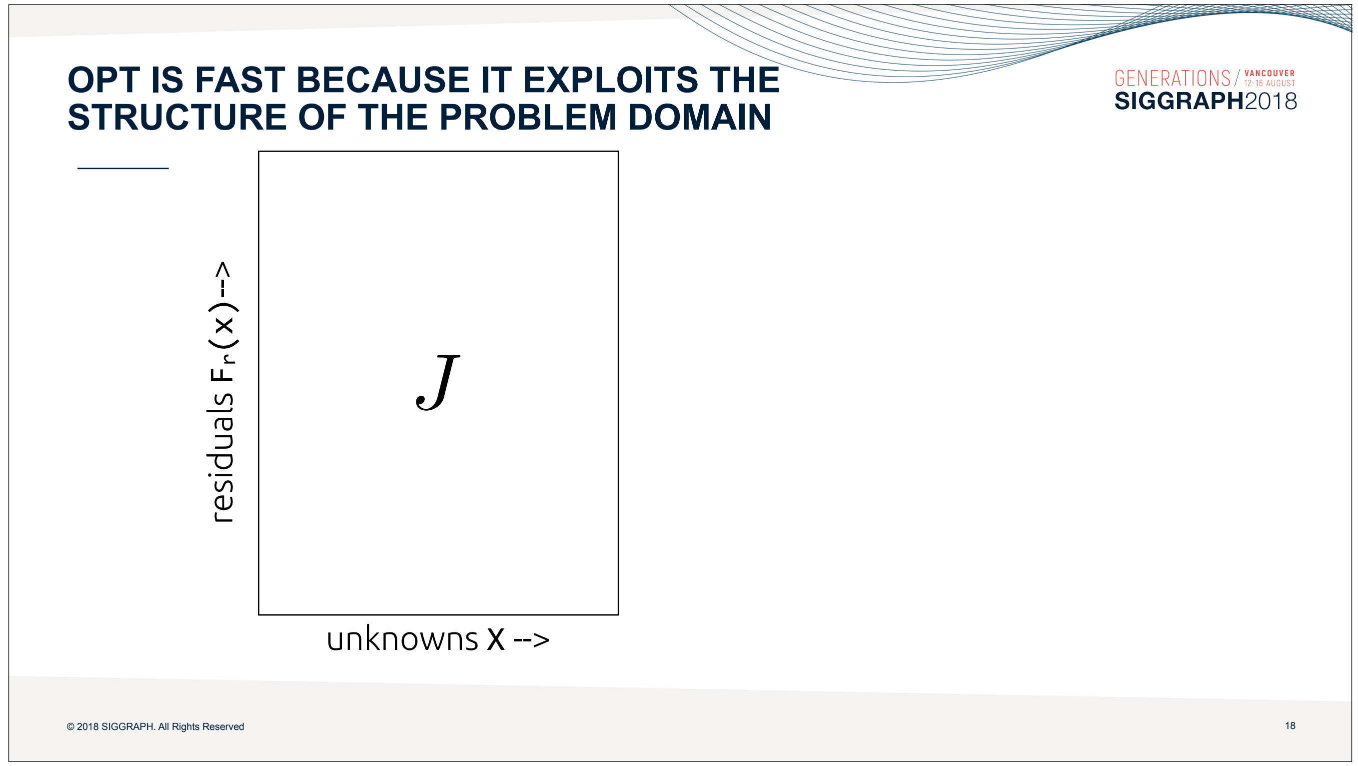


Typical solvers would explicitly construct the J matrix, using a sparse matrix format like compressed sparse row, throwing away structure.

The reason Opt can solve so much faster is because it preserves and exploits this structure of the images and meshes that we define the problem on. We can see this by looking at J, the jacobian matrix

<> The Jacobian matrix J contain the partial derivatives of each residual with each unknown.

<> For a mesh-based domain you might have unknowns that represent coordinates in space.

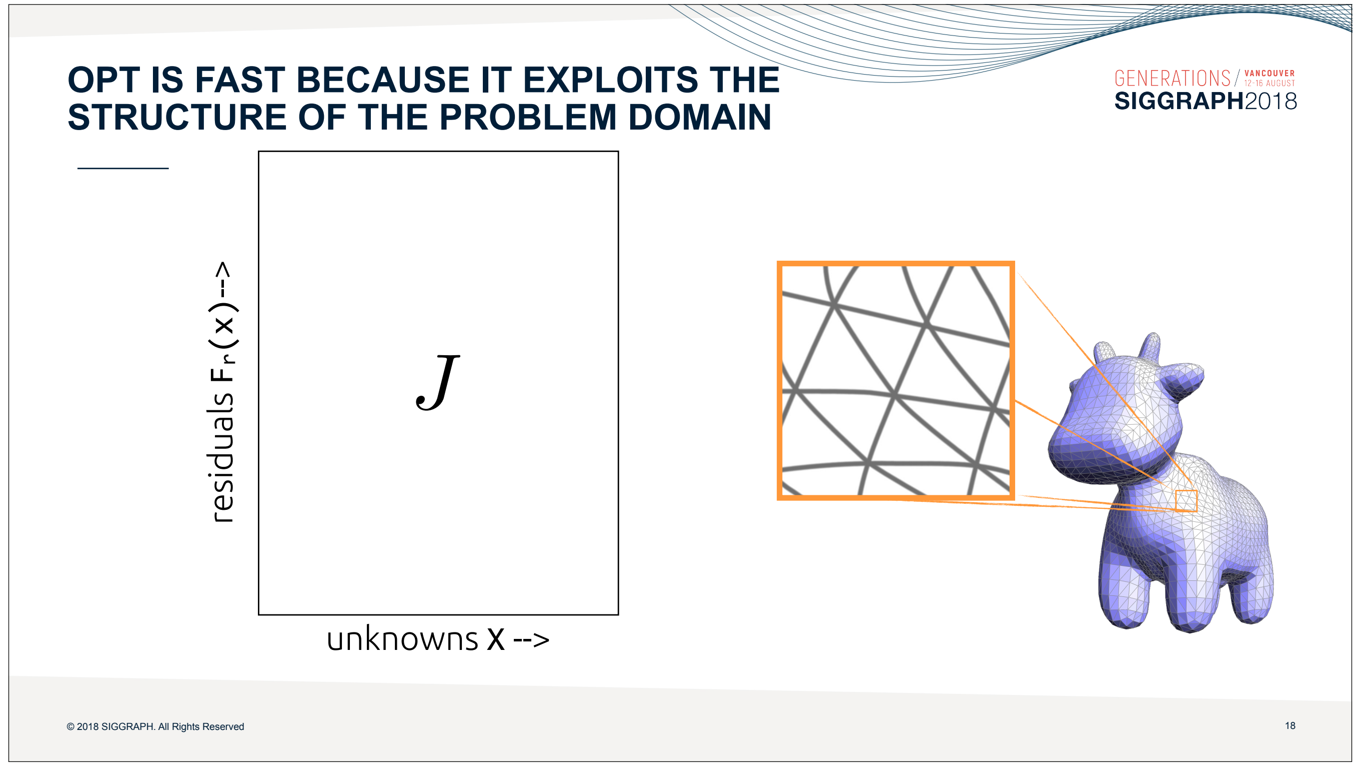


Typical solvers would explicitly construct the J matrix, using a sparse matrix format like compressed sparse row, throwing away structure.

The reason Opt can solve so much faster is because it preserves and exploits this structure of the images and meshes that we define the problem on. We can see this by looking at J, the jacobian matrix

<> The Jacobian matrix J contain the partial derivatives of each residual with each unknown.

<> For a mesh-based domain you might have unknowns that represent coordinates in space.

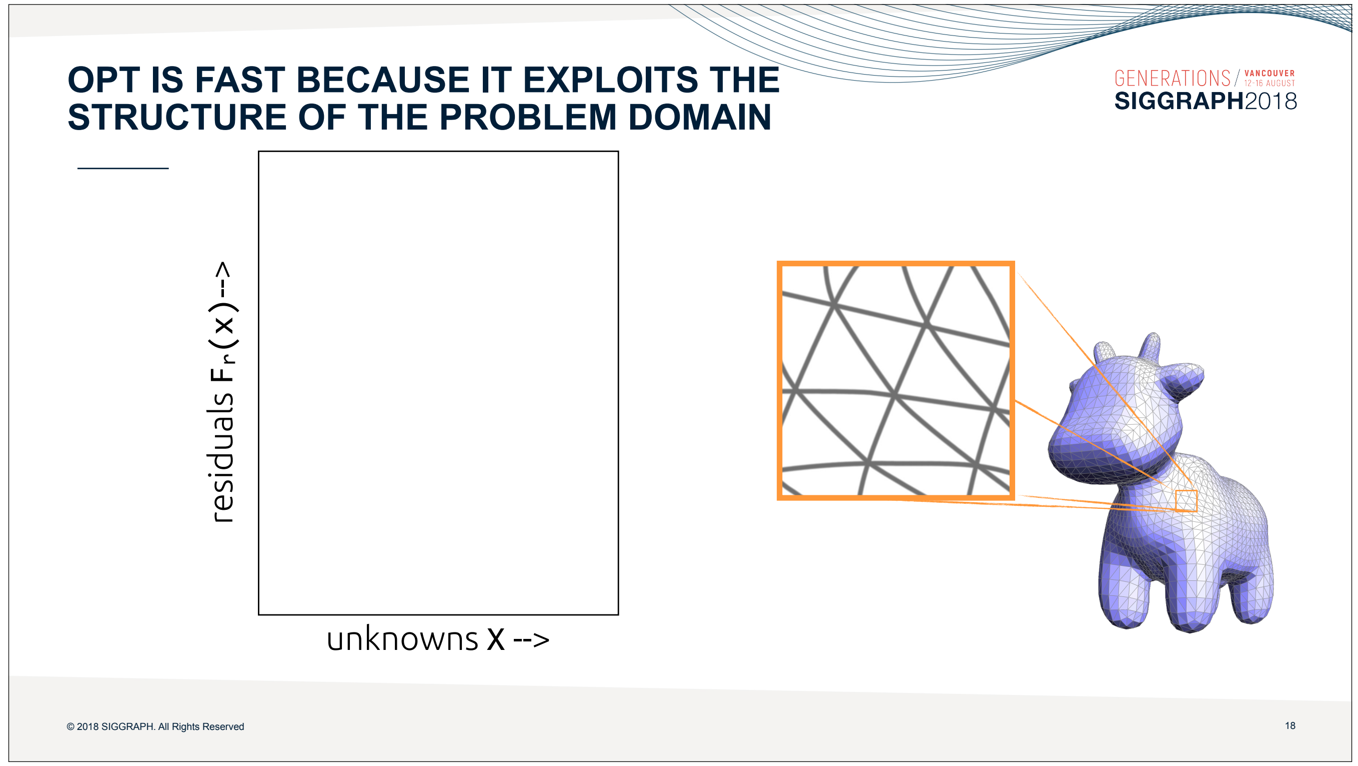


Typical solvers would explicitly construct the J matrix, using a sparse matrix format like compressed sparse row, throwing away structure.

The reason Opt can solve so much faster is because it preserves and exploits this structure of the images and meshes that we define the problem on. We can see this by looking at J, the jacobian matrix

<> The Jacobian matrix J contain the partial derivatives of each residual with each unknown.

<> For a mesh-based domain you might have unknowns that represent coordinates in space.

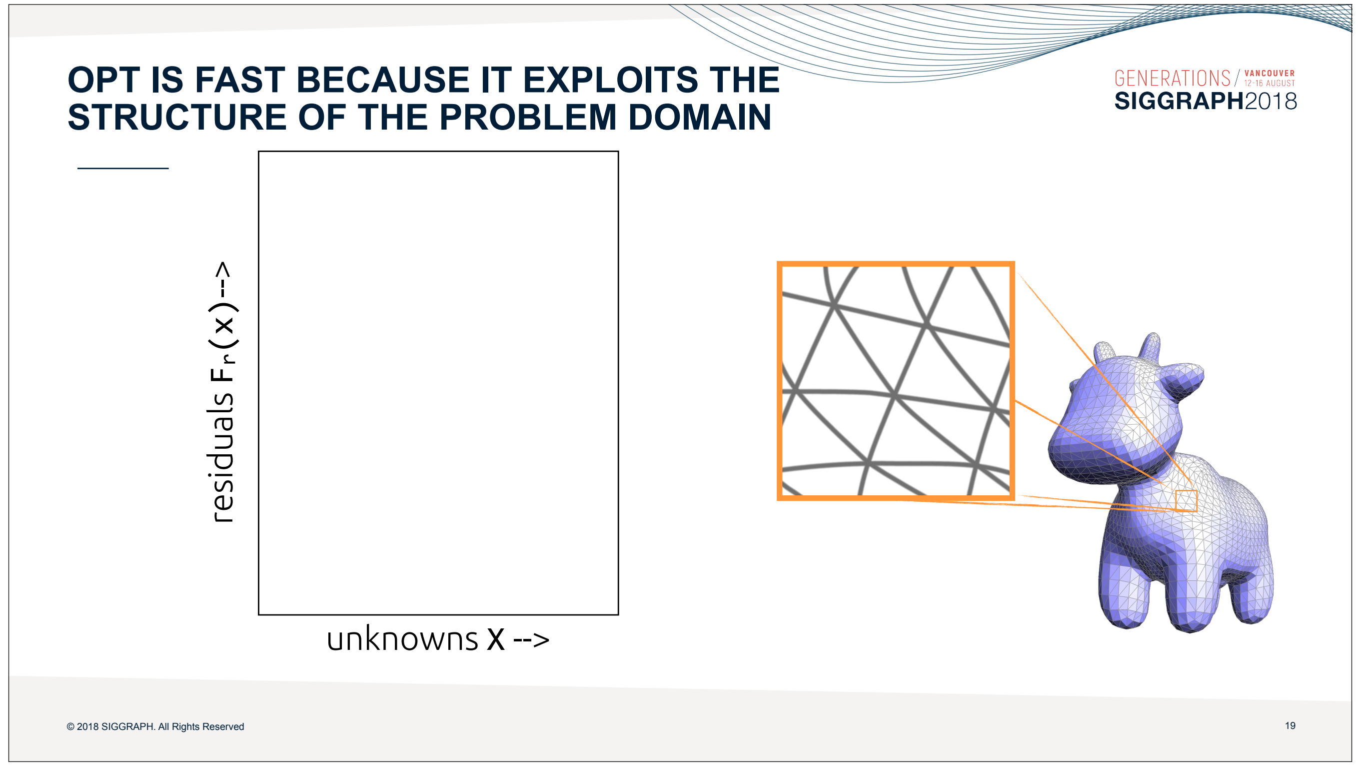


Typical solvers would explicitly construct the J matrix, using a sparse matrix format like compressed sparse row, throwing away structure.

The reason Opt can solve so much faster is because it preserves and exploits this structure of the images and meshes that we define the problem on. We can see this by looking at J, the jacobian matrix

<> The Jacobian matrix J contain the partial derivatives of each residual with each unknown.

<> For a mesh-based domain you might have unknowns that represent coordinates in space.

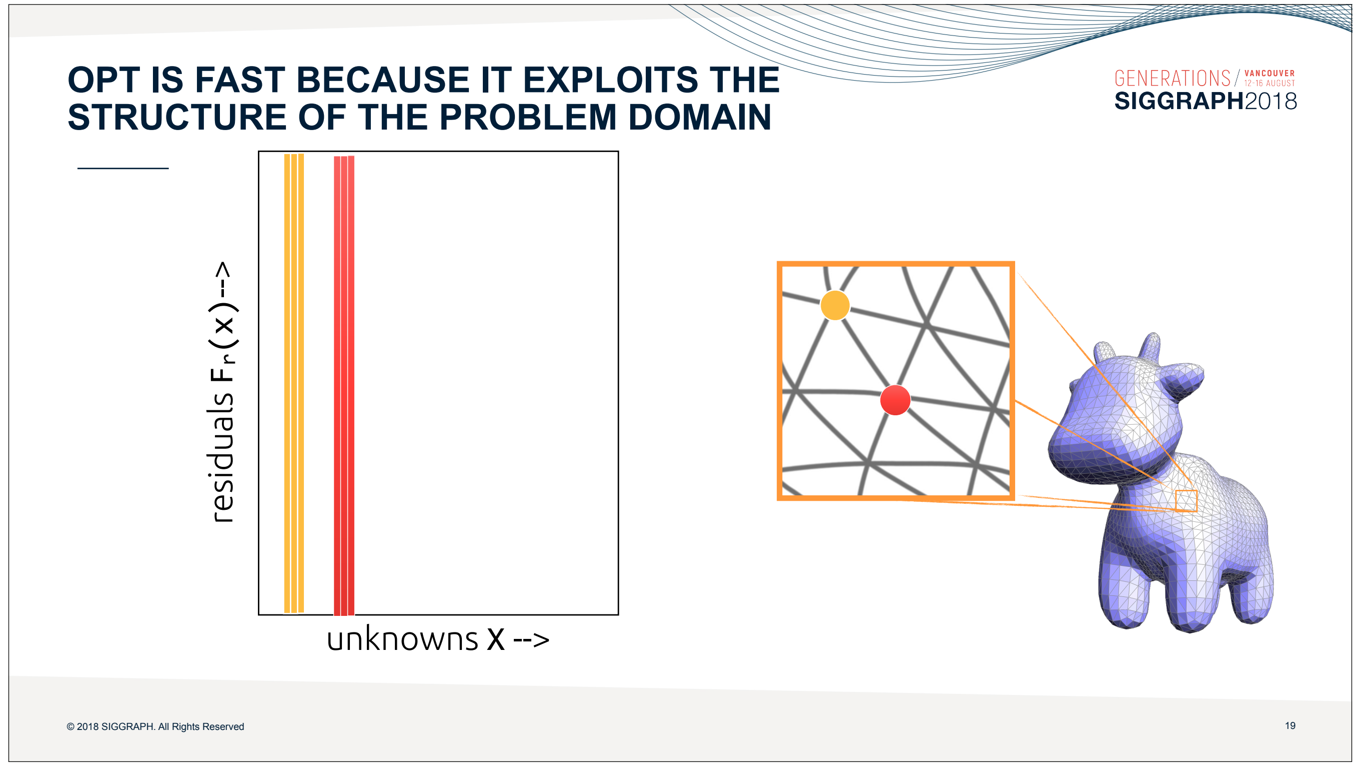


This corresponds to three columns of J per vertex

Then each edge might have several 3D residuals defined,
<> like a regularization term and a consistency term

Each of these terms are really 3 residuals and occupy 3 rows of the Jacobian

Because Opt understands the problem domain, it can exploit this structure.

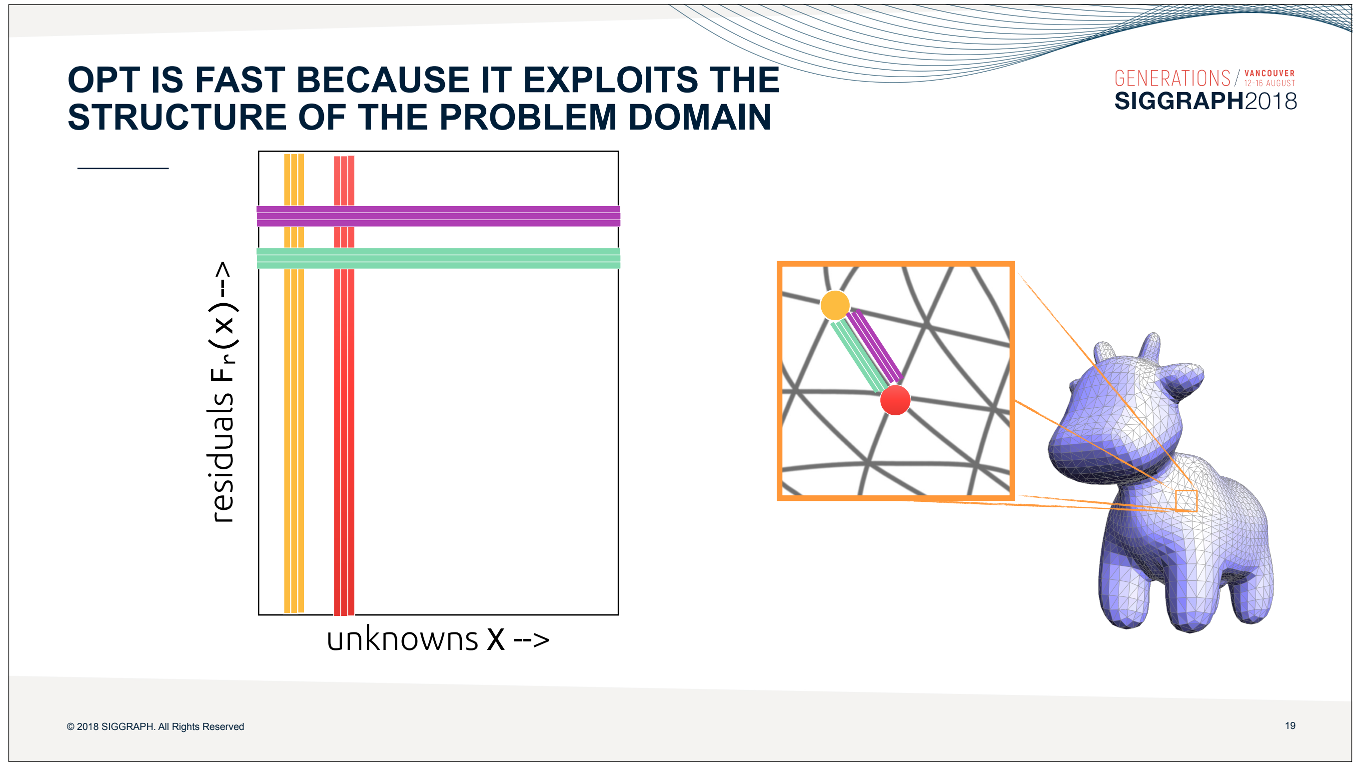


This corresponds to three columns of J per vertex

Then each edge might have several 3D residuals defined,
<> like a regularization term and a consistency term

Each of these terms are really 3 residuals and occupy 3 rows of the Jacobian

Because Opt understands the problem domain, it can exploit this structure.



This corresponds to three columns of J per vertex

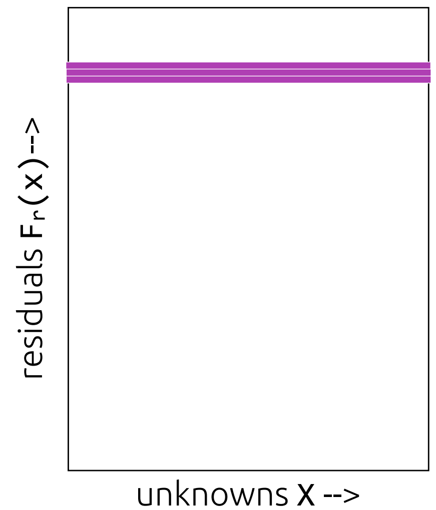
Then each edge might have several 3D residuals defined,
<> like a regularization term and a consistency term

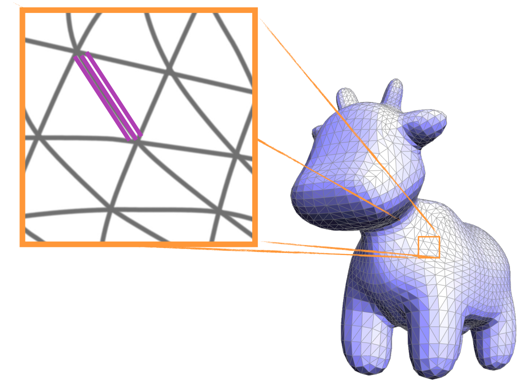
Each of these terms are really 3 residuals and occupy 3 rows of the Jacobian

Because Opt understands the problem domain, it can exploit this structure.

OPT IS FAST BECAUSE IT EXPLOITS THE STRUCTURE OF THE PROBLEM DOMAIN

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

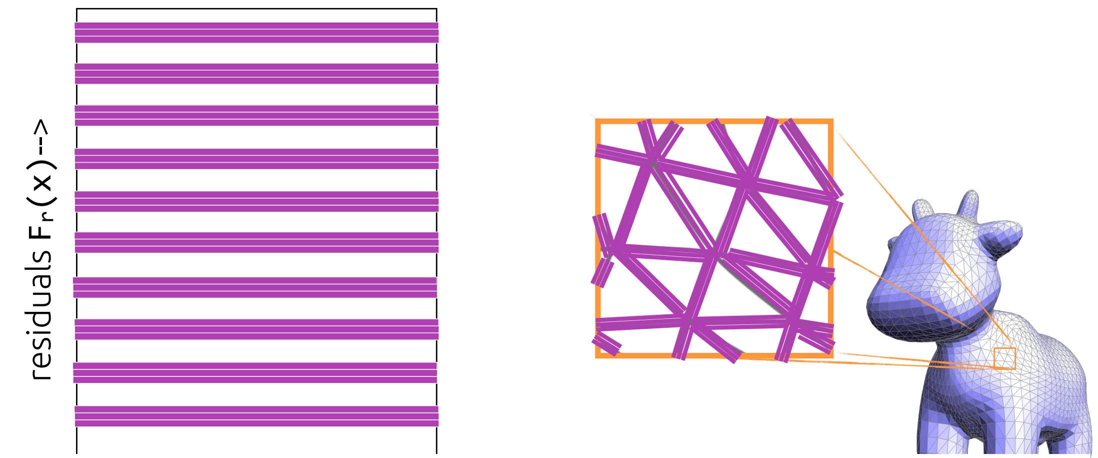




First, each residual of a particular type has the same structure, so Opt can calculate terms involving this residual in parallel on the GPU

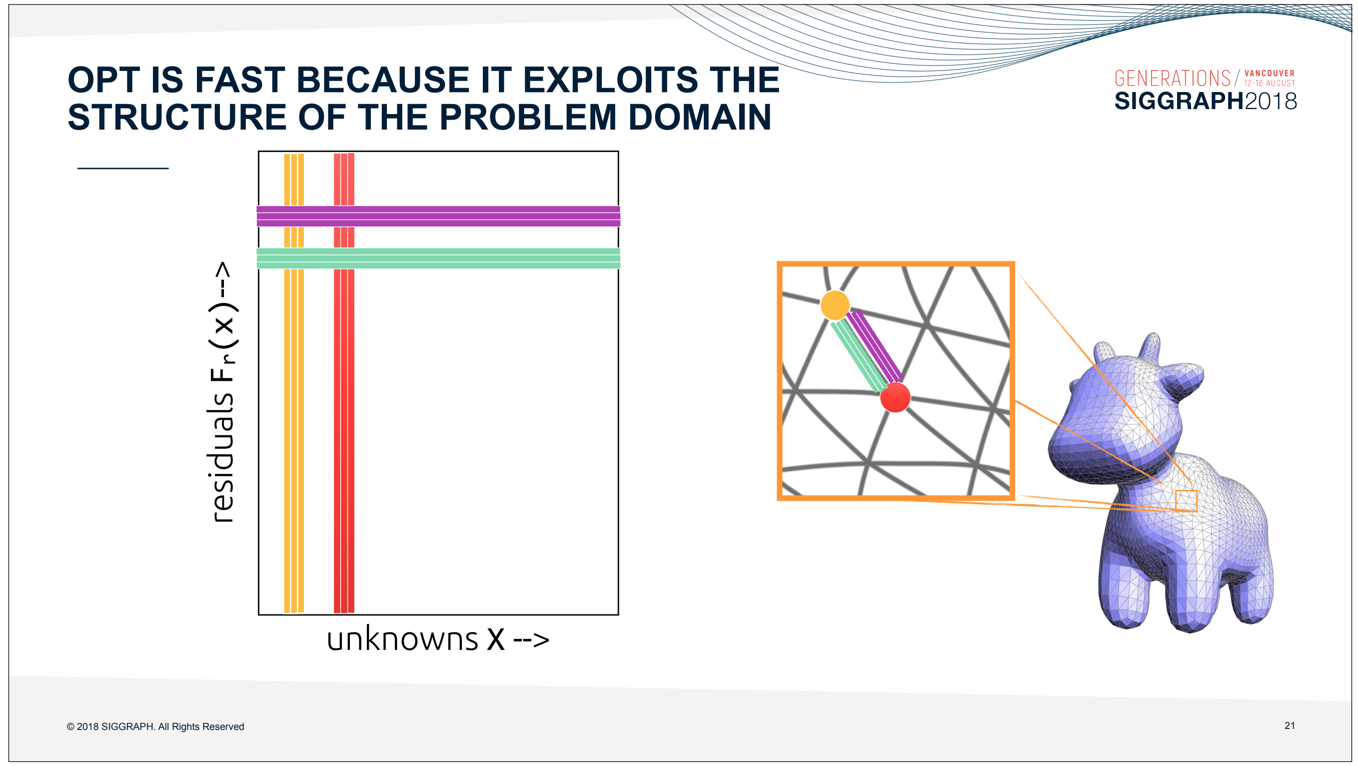
OPT IS FAST BECAUSE IT EXPLOITS THE STRUCTURE OF THE PROBLEM DOMAIN

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018



1. **Parallelism**: Each Residual Term Parallelizes by Construction

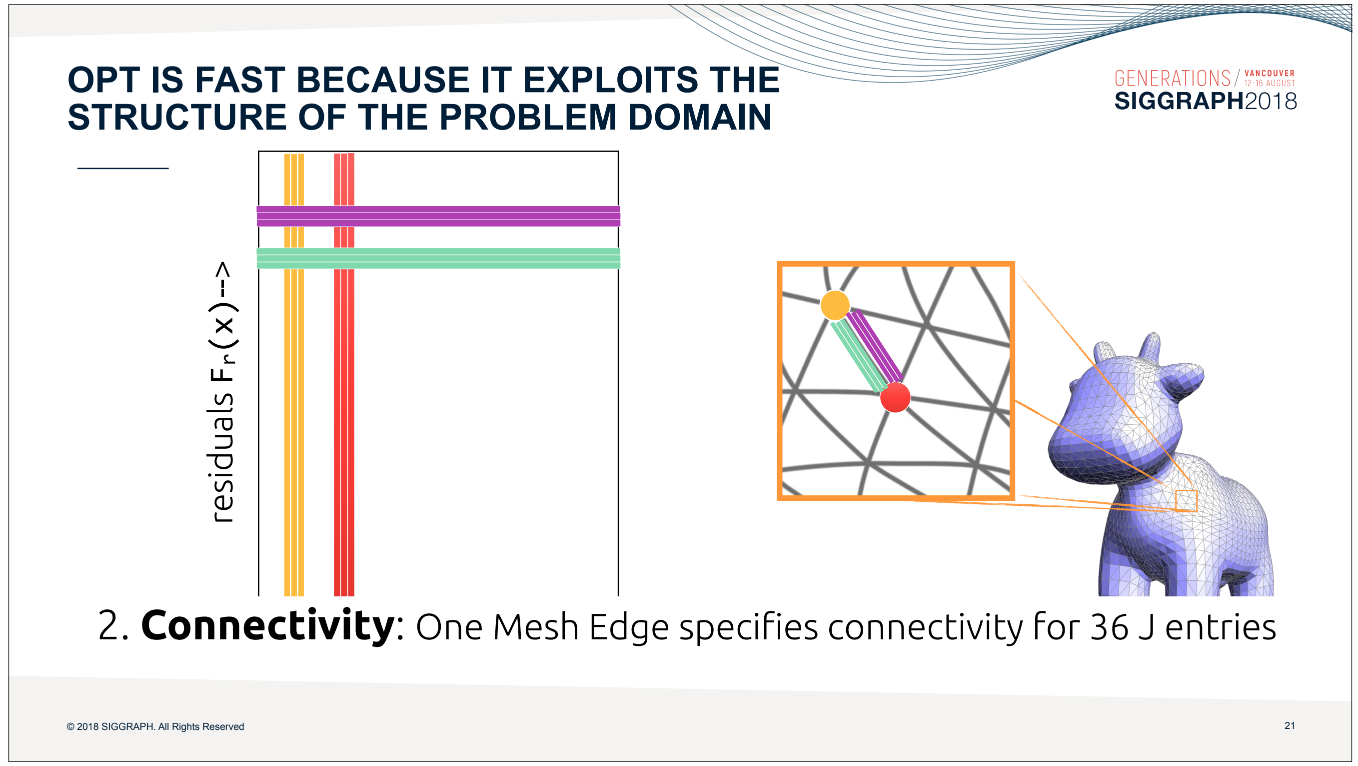
First, each residual of a particular type has the same structure, so Opt can calculate terms involving this residual in parallel on the GPU



Second, In typical sparse matrices, the connectivity for each row and each column are stored separately.

<> Here for instance, we have six unknowns and six residuals for a total of 36 non-zeros in J

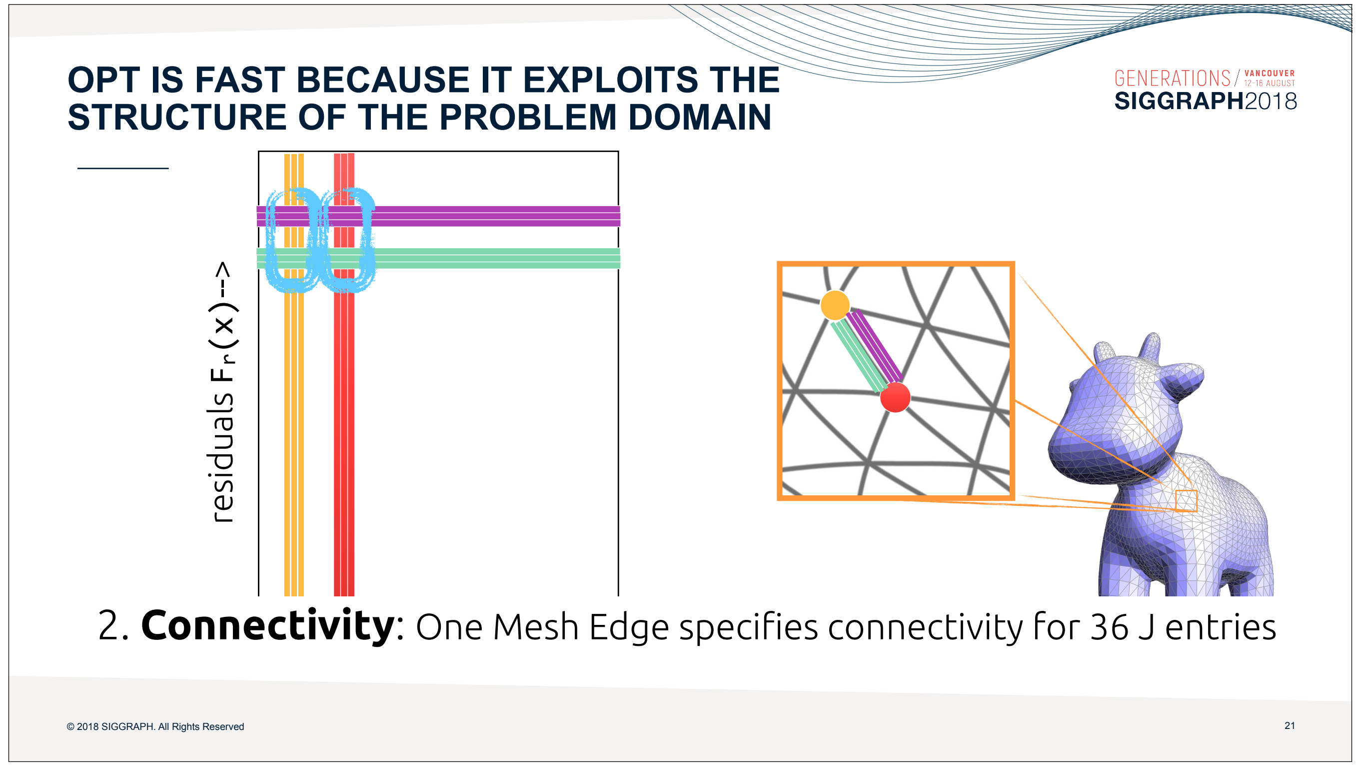
<> But all of these can be derived from a single edge. In the case of images, we don’t need to story any connectivity at all, the program defines it!



Second, In typical sparse matrices, the connectivity for each row and each column are stored separately.

<> Here for instance, we have six unknowns and six residuals for a total of 36 non-zeros in J

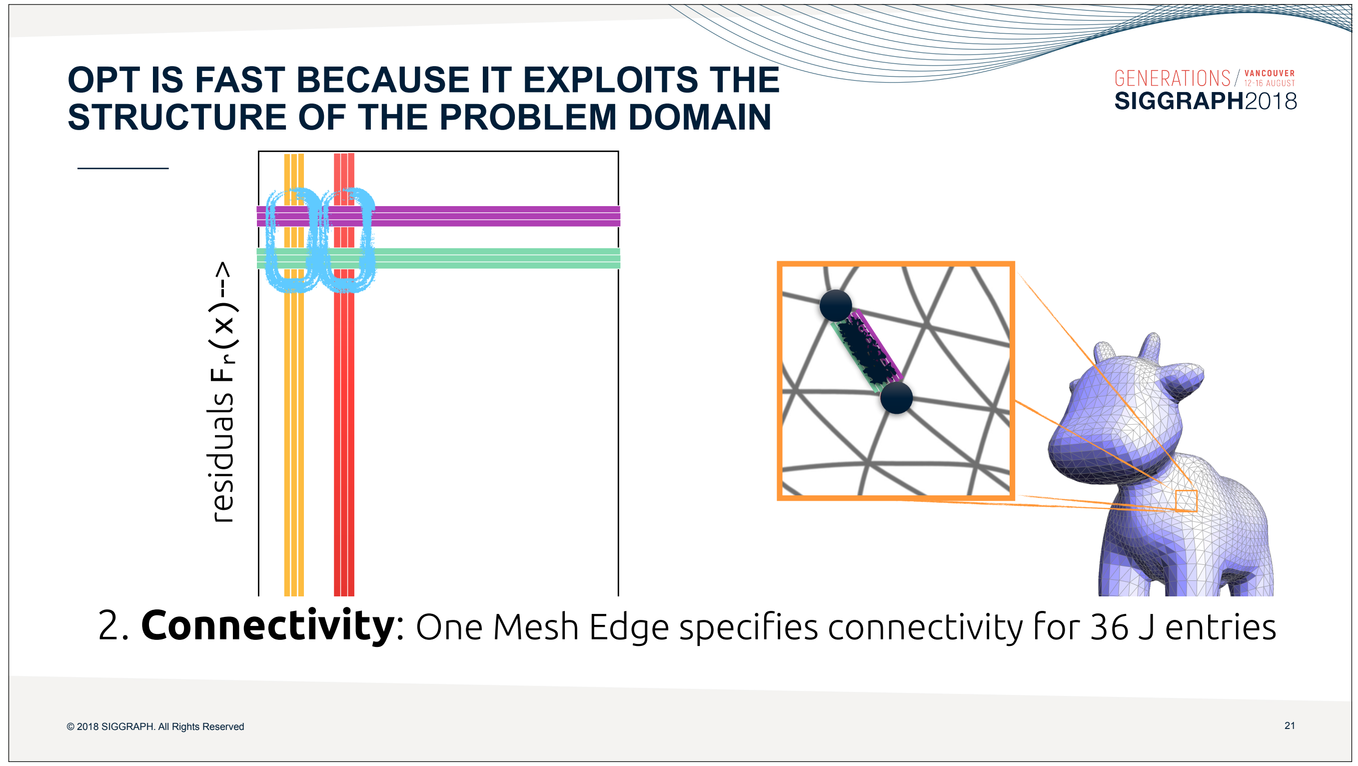
<> But all of these can be derived from a single edge. In the case of images, we don’t need to story any connectivity at all, the program defines it!



Second, In typical sparse matrices, the connectivity for each row and each column are stored separately.

<> Here for instance, we have six unknowns and six residuals for a total of 36 non-zeros in J

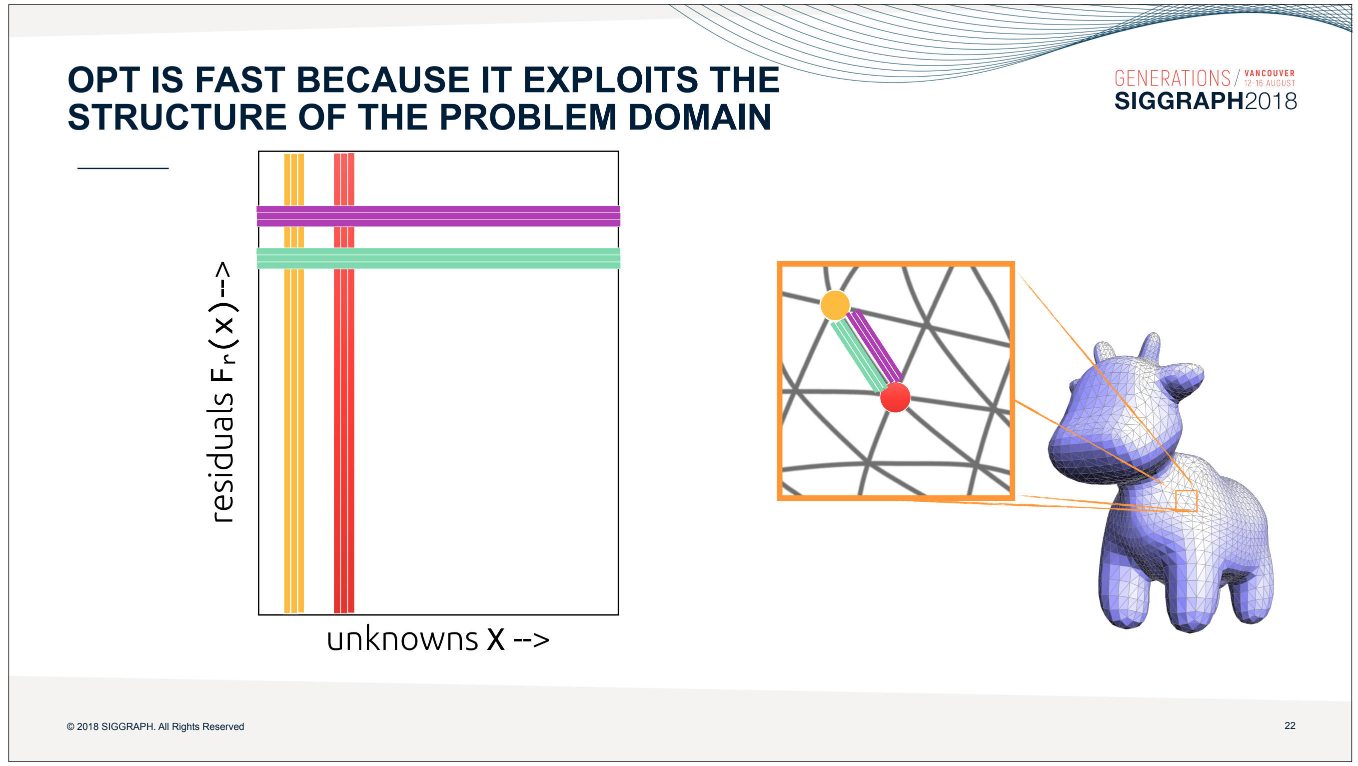
<> But all of these can be derived from a single edge. In the case of images, we don’t need to story any connectivity at all, the program defines it!



Second, In typical sparse matrices, the connectivity for each row and each column are stored separately.

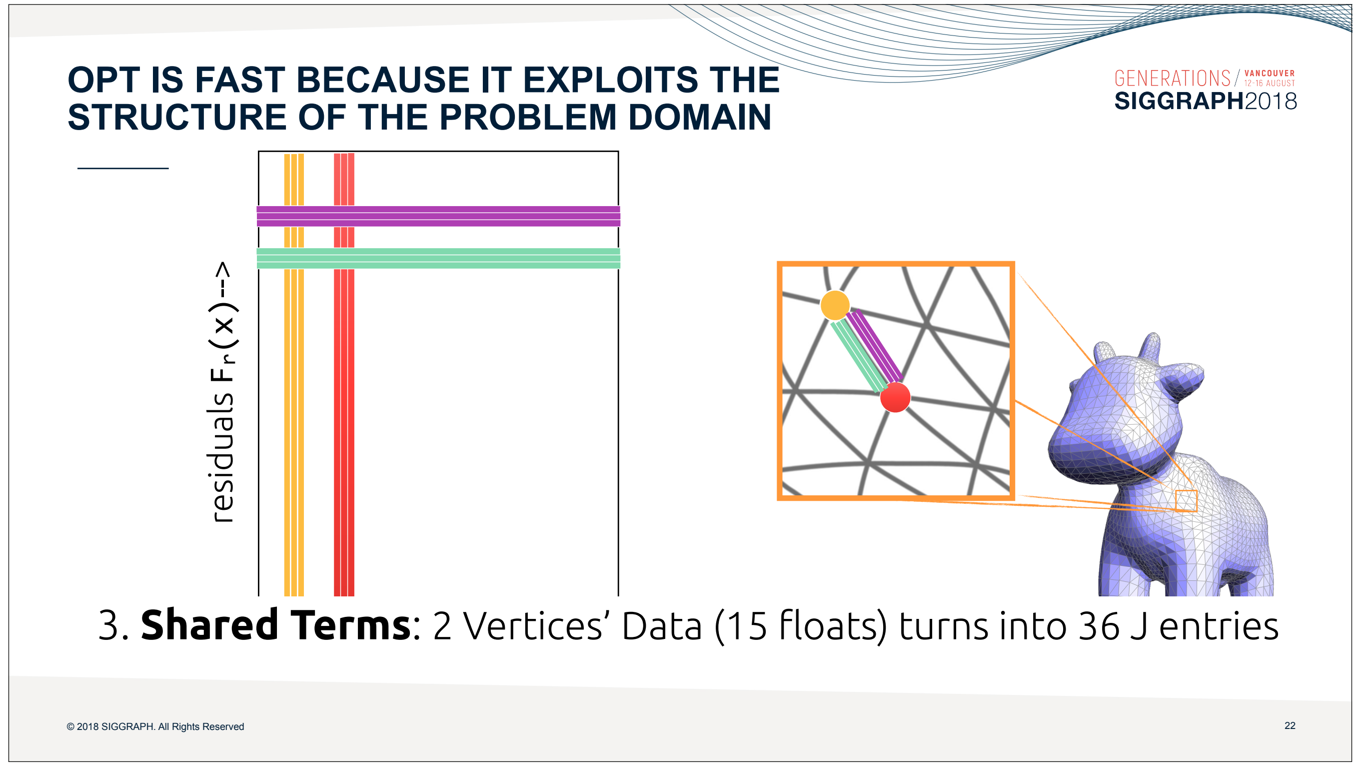
<> Here for instance, we have six unknowns and six residuals for a total of 36 non-zeros in J

<> But all of these can be derived from a single edge. In the case of images, we don’t need to story any connectivity at all, the program defines it!



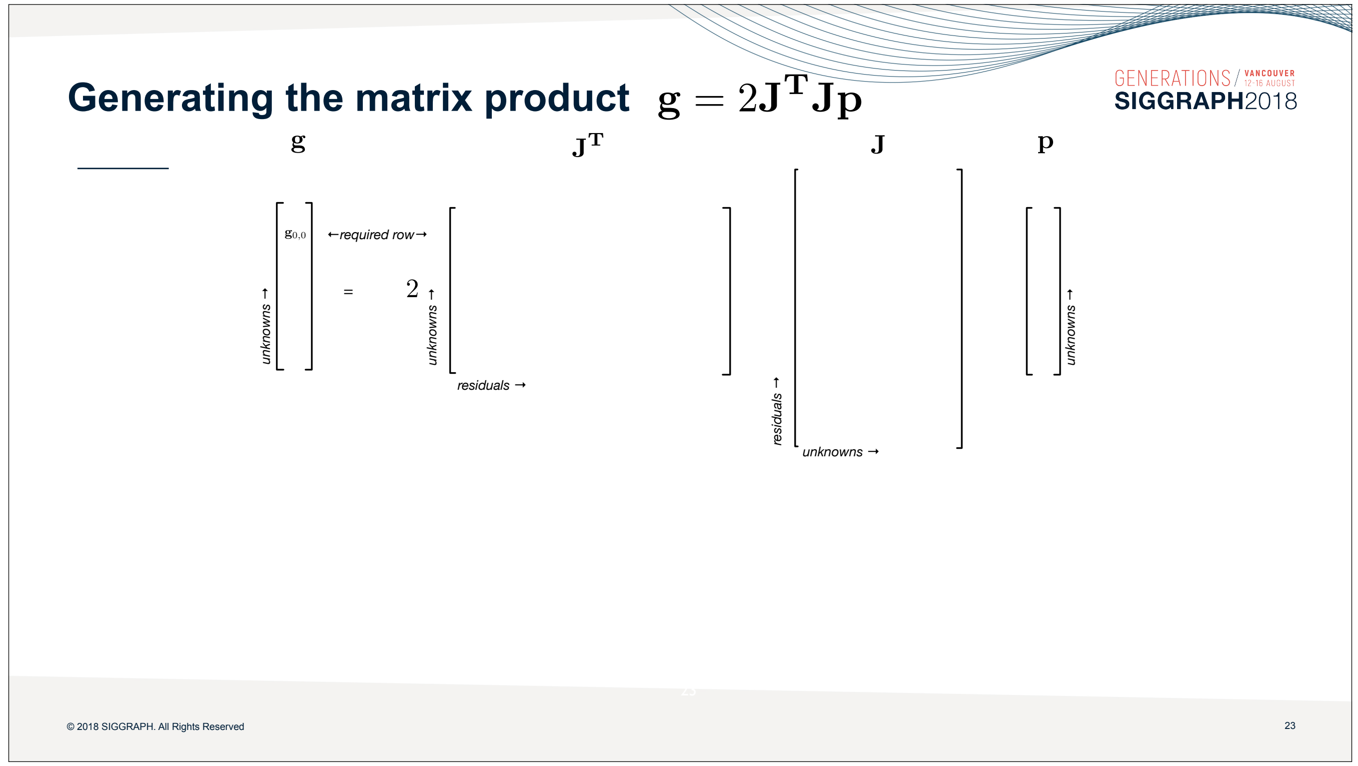
Finally, the 36 non-zeros in J can be calculated from only 15 problem terms defined on the vertices of the mesh, cutting the amount of memory loaded in half.

By exploiting all three of these properties, we can generate a very fast inner loop.



Finally, the 36 non-zeros in J can be calculated from only 15 problem terms defined on the vertices of the mesh, cutting the amount of memory loaded in half.

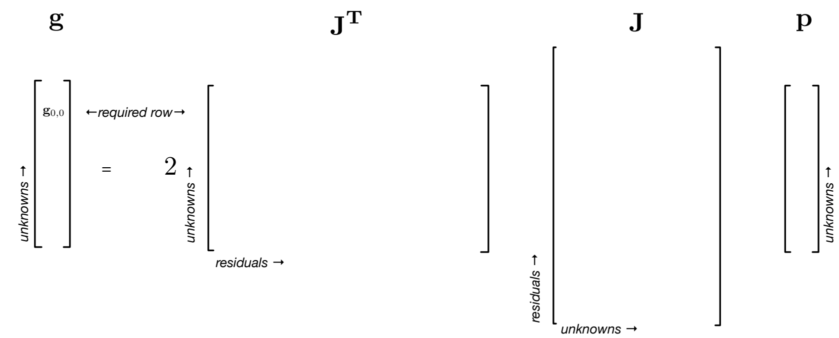
By exploiting all three of these properties, we can generate a very fast inner loop.



So the challenge in Opt is to generate this matrix product, needed for preconditioned conjugate gradient during Gauss-Newton optimization, from this high-level energy.
Handoff here
<>The structure of this matrix product is specific to a particular set of energy functions.
so we need to use compiler techniques to answer questions using the energy:
<> First, given the energy, we need to find the expression that calculates any particular entry in the Jacobian matrix J.
<> and since J is sparse, we need to use the energy to identify what entries are non-zero.

Both of these questions can be answered using simple program analysis of the energy.
<> For the first problem, we can use a differentiation method to turn the residuals into their partial derivatives.
<> For the second problem, we can use data-dependency analysis that inverts the mapping from residuals to unknowns

Generating the matrix product $\mathbf{g} = 2\mathbf{J}^\mathbf{T}\mathbf{J}\mathbf{p}$

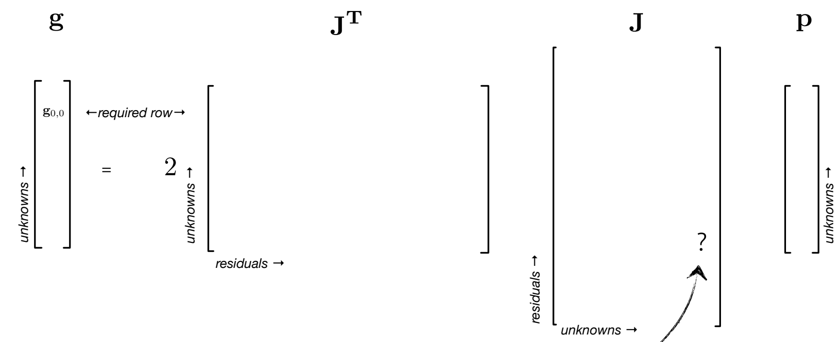


Need compiler techniques to answer two questions using the energy:

So the challenge in Opt is to generate this matrix product, needed for preconditioned conjugate gradient during Gauss-Newton optimization, from this high-level energy.
Handoff here
<>The structure of this matrix product is specific to a particular set of energy functions.
so we need to use compiler techniques to answer questions using the energy:
<> First, given the energy, we need to find the expression that calculates any particular entry in the Jacobian matrix J.
<> and since J is sparse, we need to use the energy to identify what entries are non-zero.

Both of these questions can be answered using simple program analysis of the energy.
<> For the first problem, we can use a differentiation method to turn the residuals into their partial derivatives.
<> For the second problem, we can use data-dependency analysis that inverts the mapping from residuals to unknowns

Generating the matrix product $\mathbf{g} = 2\mathbf{J}^{\mathbf{T}}\mathbf{J}\mathbf{p}$

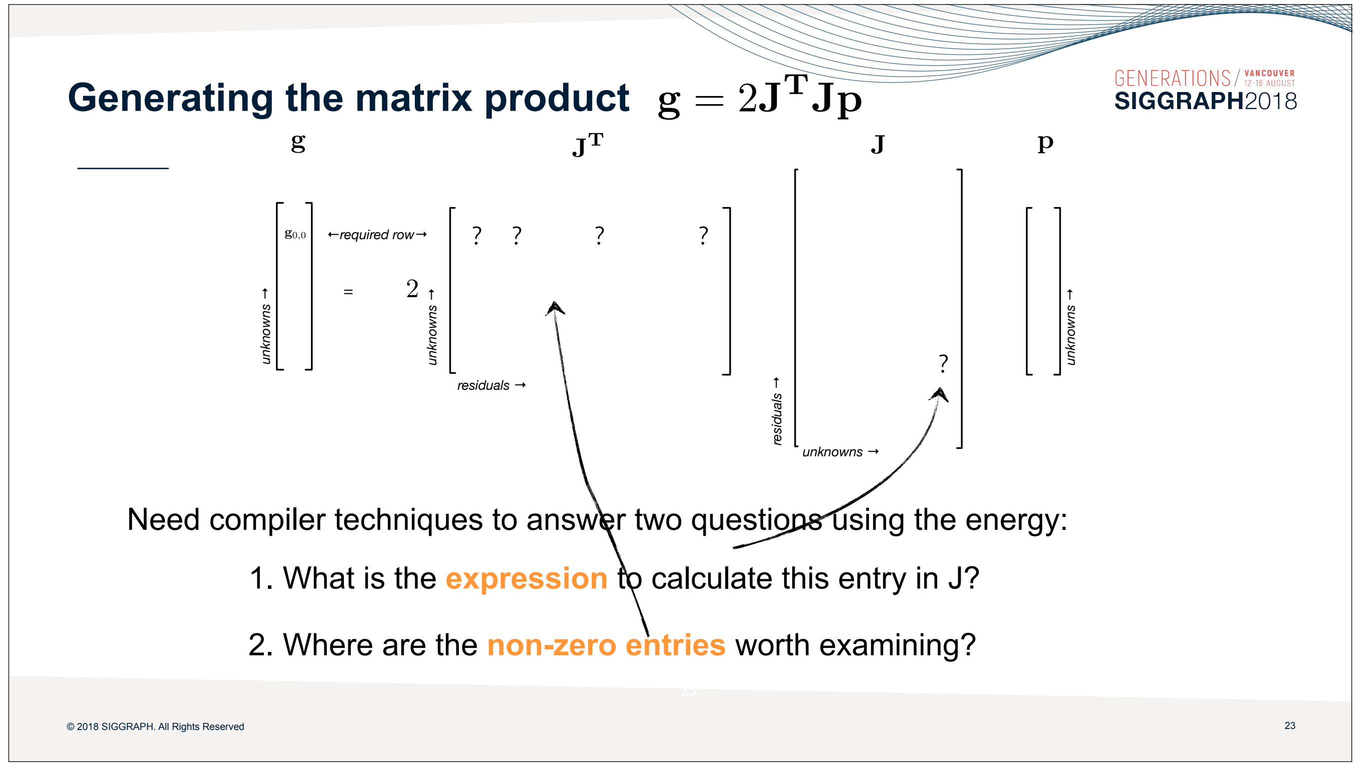


Need compiler techniques to answer two questions using the energy:

1. What is the **expression** to calculate this entry in J?

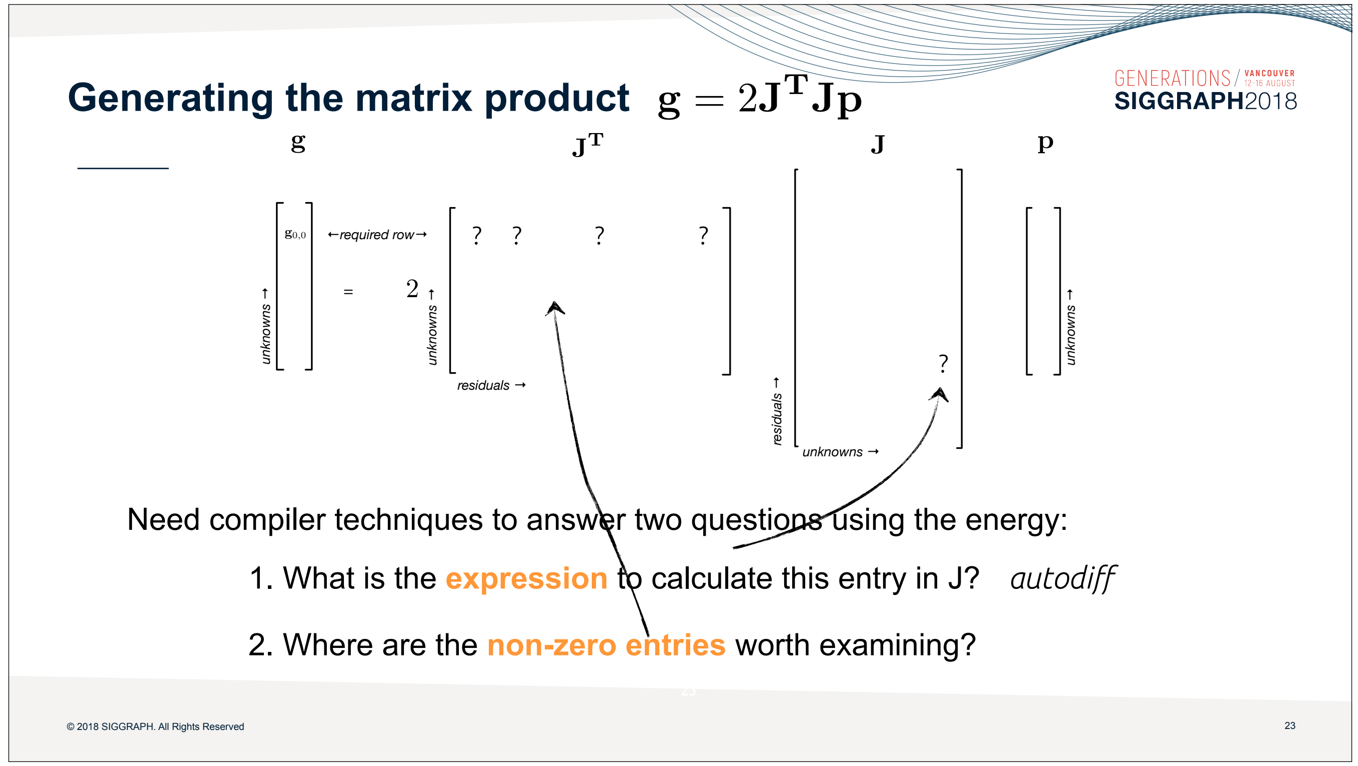
So the challenge in Opt is to generate this matrix product, needed for preconditioned conjugate gradient during Gauss-Newton optimization, from this high-level energy.
Handoff here
<>The structure of this matrix product is specific to a particular set of energy functions.
so we need to use compiler techniques to answer questions using the energy:
<> First, given the energy, we need to find the expression that calculates any particular entry in the Jacobian matrix J.
<> and since J is sparse, we need to use the energy to identify what entries are non-zero.

Both of these questions can be answered using simple program analysis of the energy.
<> For the first problem, we can use a differentiation method to turn the residuals into their partial derivatives.
<> For the second problem, we can use data-dependency analysis that inverts the mapping from residuals to unknowns



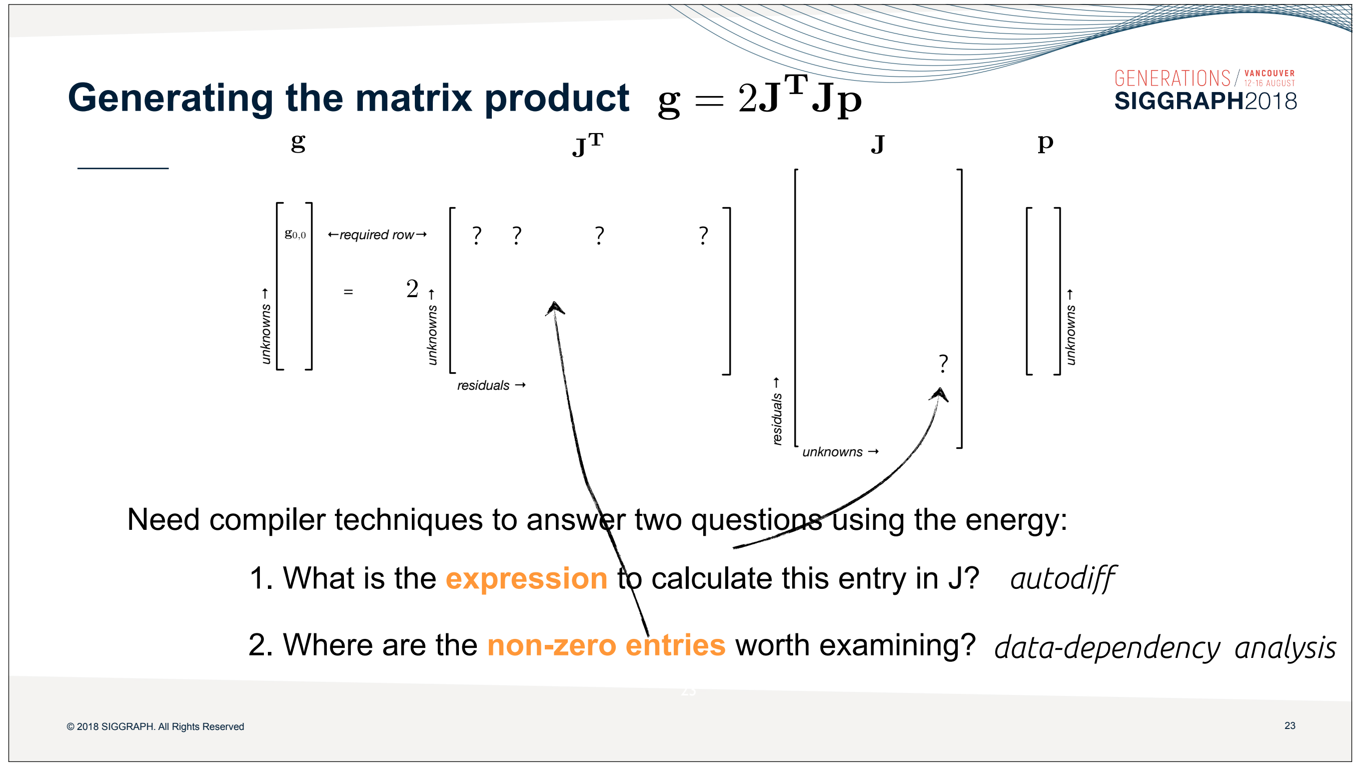
So the challenge in Opt is to generate this matrix product, needed for preconditioned conjugate gradient during Gauss-Newton optimization, from this high-level energy.
Handoff here
<>The structure of this matrix product is specific to a particular set of energy functions.
so we need to use compiler techniques to answer questions using the energy:
<> First, given the energy, we need to find the expression that calculates any particular entry in the Jacobian matrix J.
<> and since J is sparse, we need to use the energy to identify what entries are non-zero.

Both of these questions can be answered using simple program analysis of the energy.
<> For the first problem, we can use a differentiation method to turn the residuals into their partial derivatives.
<> For the second problem, we can use data-dependency analysis that inverts the mapping from residuals to unknowns



So the challenge in Opt is to generate this matrix product, needed for preconditioned conjugate gradient during Gauss-Newton optimization, from this high-level energy.
Handoff here
<>The structure of this matrix product is specific to a particular set of energy functions.
so we need to use compiler techniques to answer questions using the energy:
<> First, given the energy, we need to find the expression that calculates any particular entry in the Jacobian matrix J.
<> and since J is sparse, we need to use the energy to identify what entries are non-zero.

Both of these questions can be answered using simple program analysis of the energy.
<> For the first problem, we can use a differentiation method to turn the residuals into their partial derivatives.
<> For the second problem, we can use data-dependency analysis that inverts the mapping from residuals to unknowns



So the challenge in Opt is to generate this matrix product, needed for preconditioned conjugate gradient during Gauss-Newton optimization, from this high-level energy.
Handoff here
<>The structure of this matrix product is specific to a particular set of energy functions.
so we need to use compiler techniques to answer questions using the energy:
<> First, given the energy, we need to find the expression that calculates any particular entry in the Jacobian matrix J.
<> and since J is sparse, we need to use the energy to identify what entries are non-zero.

Both of these questions can be answered using simple program analysis of the energy.
<> For the first problem, we can use a differentiation method to turn the residuals into their partial derivatives.
<> For the second problem, we can use data-dependency analysis that inverts the mapping from residuals to unknowns

1. What is the **expression** to calculate this entry in J?


```
w_fit,w_reg = .1,.9
w_fit*(X(0,0) - A(0,0))
w_reg*(X(0,0) - X(1,0))
w_reg*(X(0,0) - X(0,1))
```

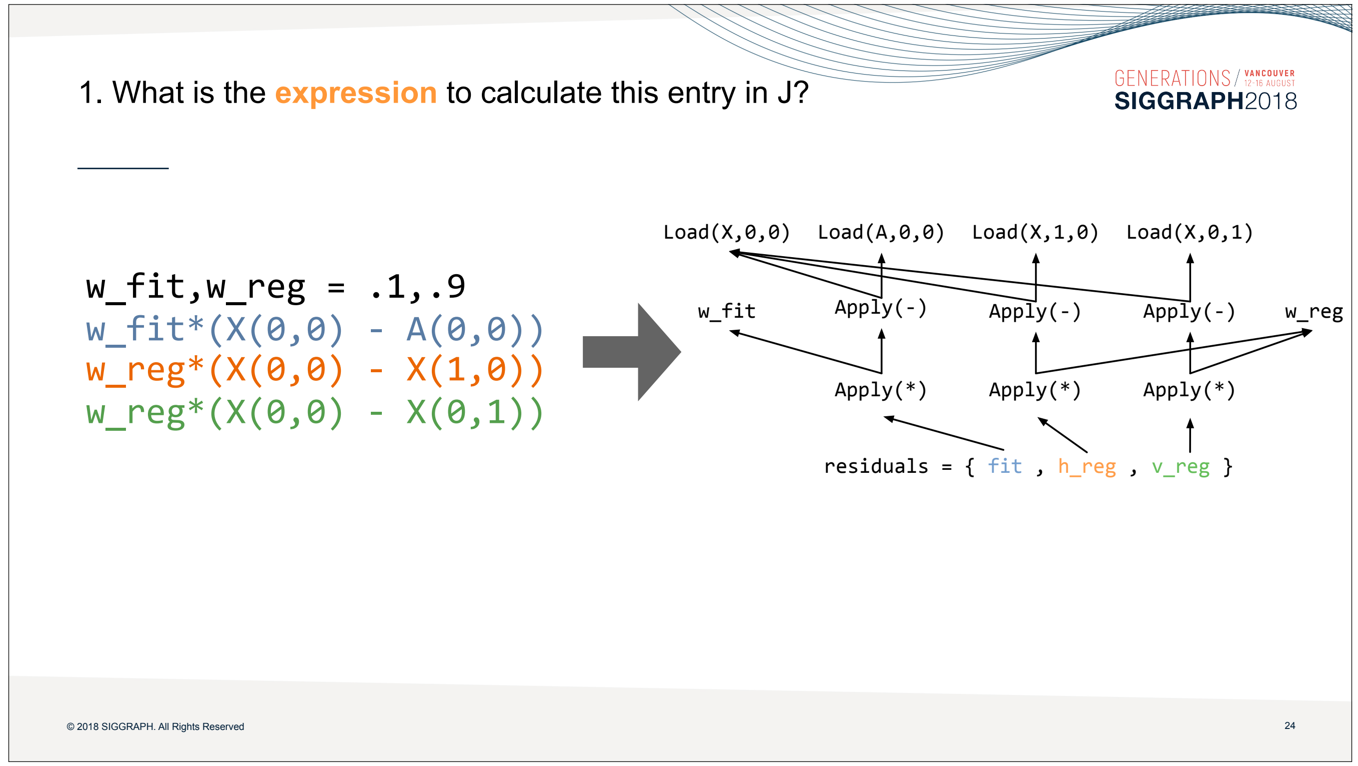

So the first thing we need to answer is: What is the expression to calculate this entry in J?

<> We transform code written in Opt into a DAG of operators, de-duplicating any reused terms.

At the top is data loaded from the unknown, and at the bottom are individual residual terms at a particular node.

<> Any pair of residual output and unknown input defines one non-zero partial derivative in the J matrix.

<> We can compute the expression for these derivatives using a compiler-based autodiff. In the paper we show how we apply additional optimizations to the resulting expression to minimize the number of terms generated, and handle boundary conditions efficiently.



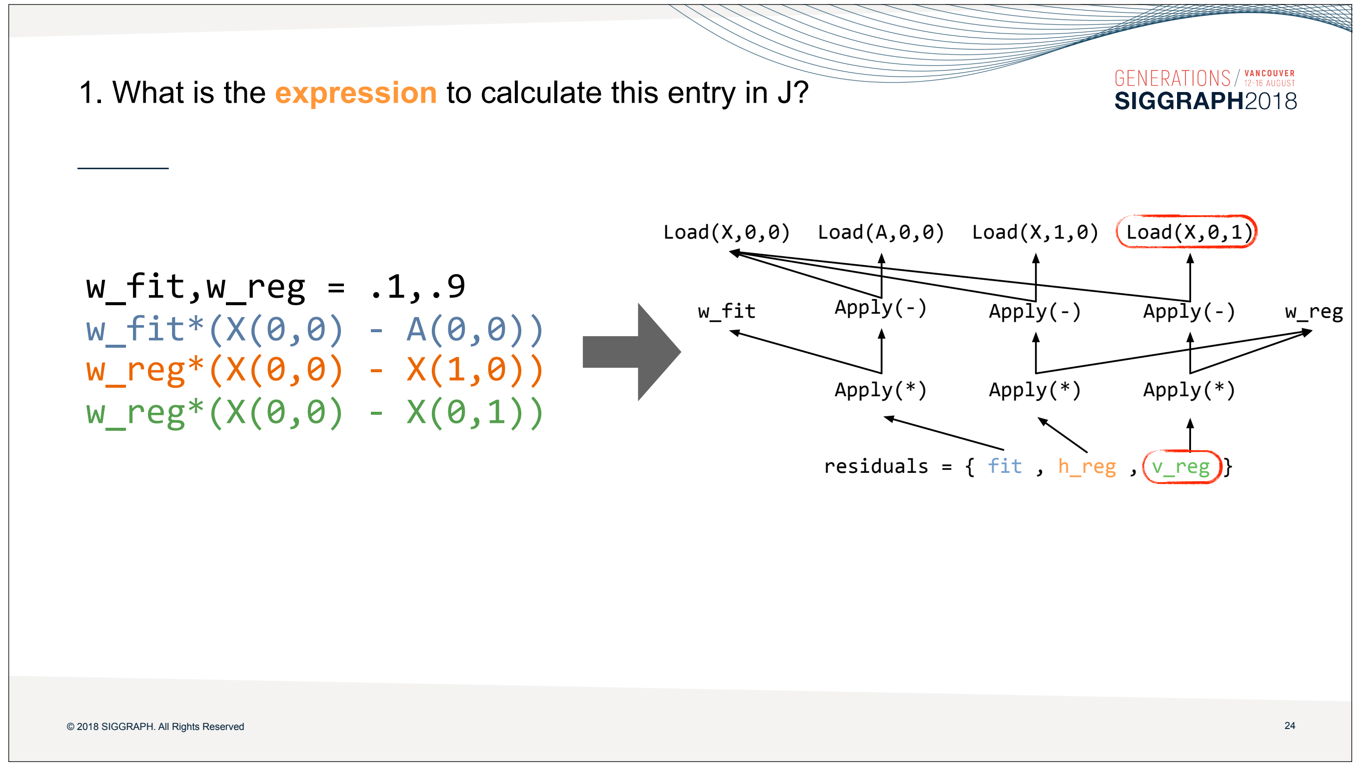
So the first thing we need to answer is: What is the expression to calculate this entry in J?

<> We transform code written in Opt into a DAG of operators, de-duplicating any reused terms.

At the top is data loaded from the unknown, and at the bottom are individual residual terms at a particular node.

<> Any pair of residual output and unknown input defines one non-zero partial derivative in the J matrix.

<> We can compute the expression for these derivatives using a compiler-based autodiff. In the paper we show how we apply additional optimizations to the resulting expression to minimize the number of terms generated, and handle boundary conditions efficiently.



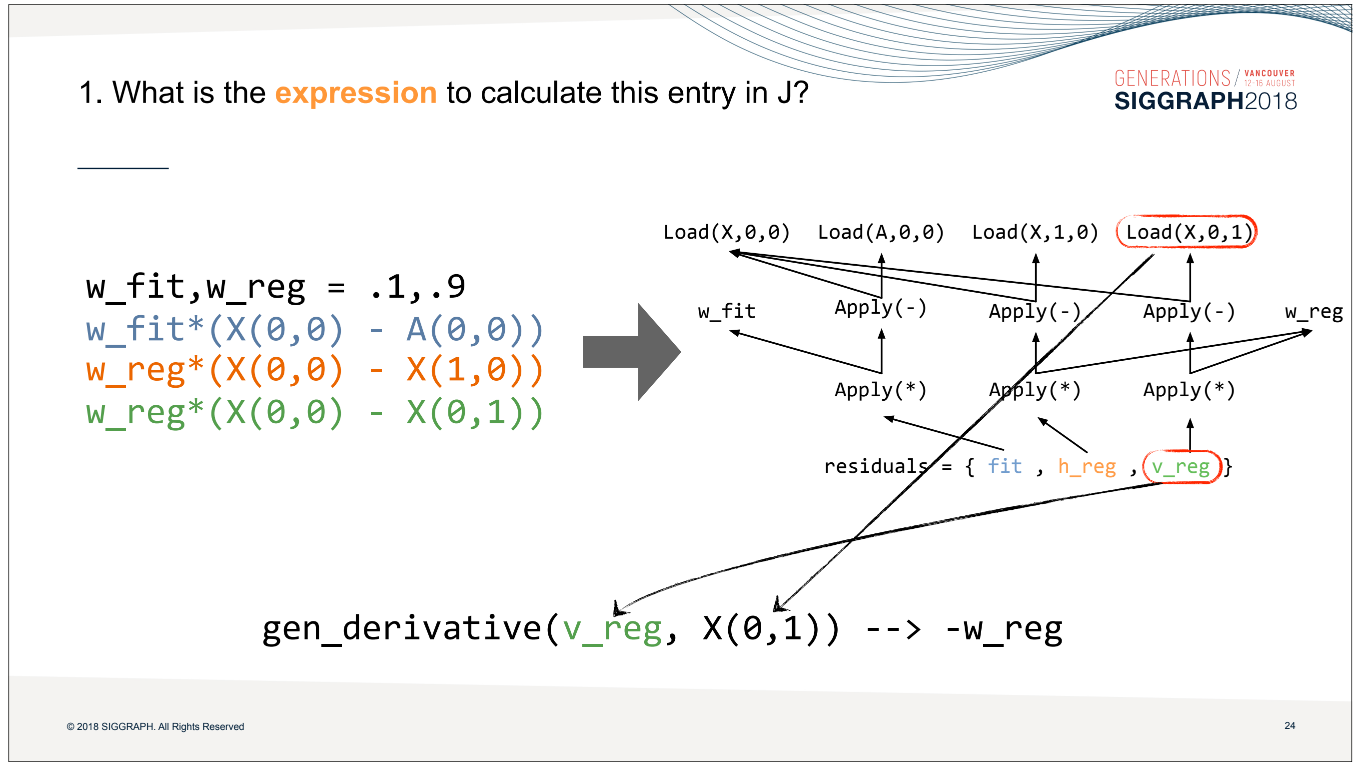
So the first thing we need to answer is: What is the expression to calculate this entry in J?

<> We transform code written in Opt into a DAG of operators, de-duplicating any reused terms.

At the top is data loaded from the unknown, and at the bottom are individual residual terms at a particular node.

<> Any pair of residual output and unknown input defines one non-zero partial derivative in the J matrix.

<> We can compute the expression for these derivatives using a compiler-based autodiff. In the paper we show how we apply additional optimizations to the resulting expression to minimize the number of terms generated, and handle boundary conditions efficiently.



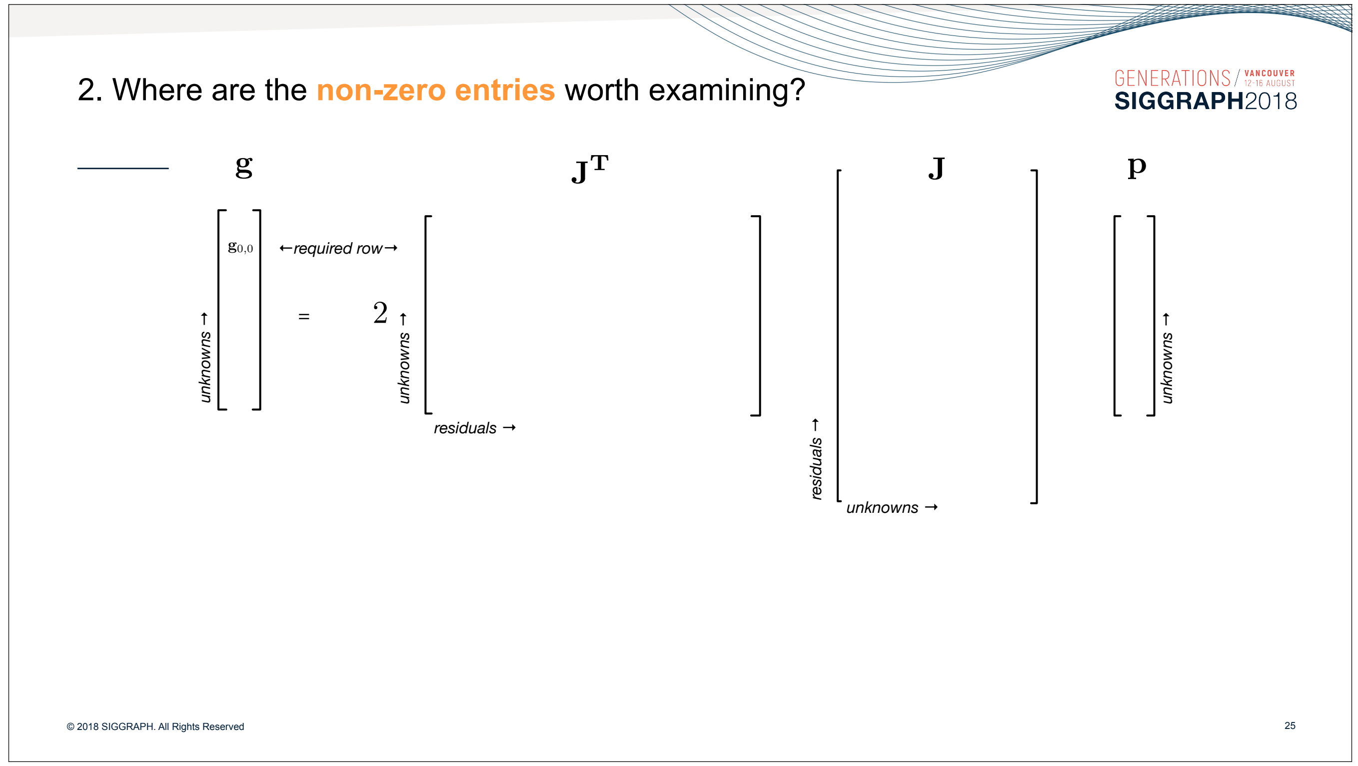
So the first thing we need to answer is: What is the expression to calculate this entry in J?

<> We transform code written in Opt into a DAG of operators, de-duplicating any reused terms.

At the top is data loaded from the unknown, and at the bottom are individual residual terms at a particular node.

<> Any pair of residual output and unknown input defines one non-zero partial derivative in the J matrix.

<> We can compute the expression for these derivatives using a compiler-based autodiff. In the paper we show how we apply additional optimizations to the resulting expression to minimize the number of terms generated, and handle boundary conditions efficiently.

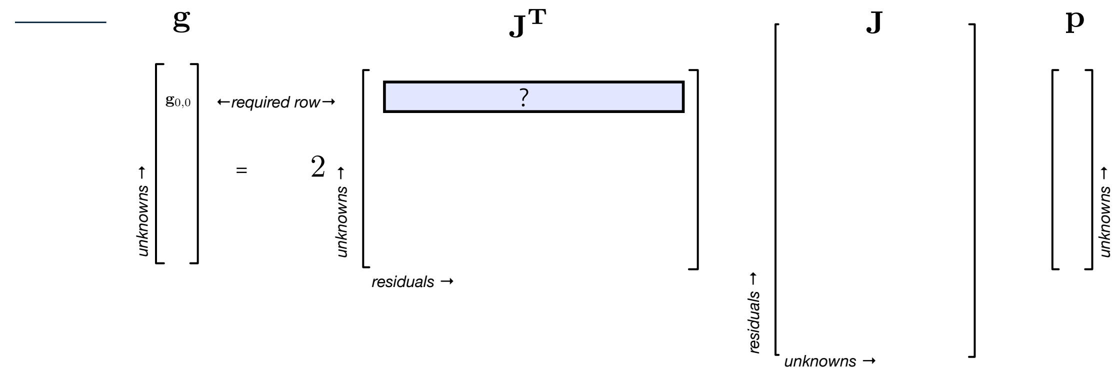


Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
“what are the non-zero columns related this unknown”
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.

GENERATIONS / VANCOUVER 12-16 AUGUST SIGGRAPH2018

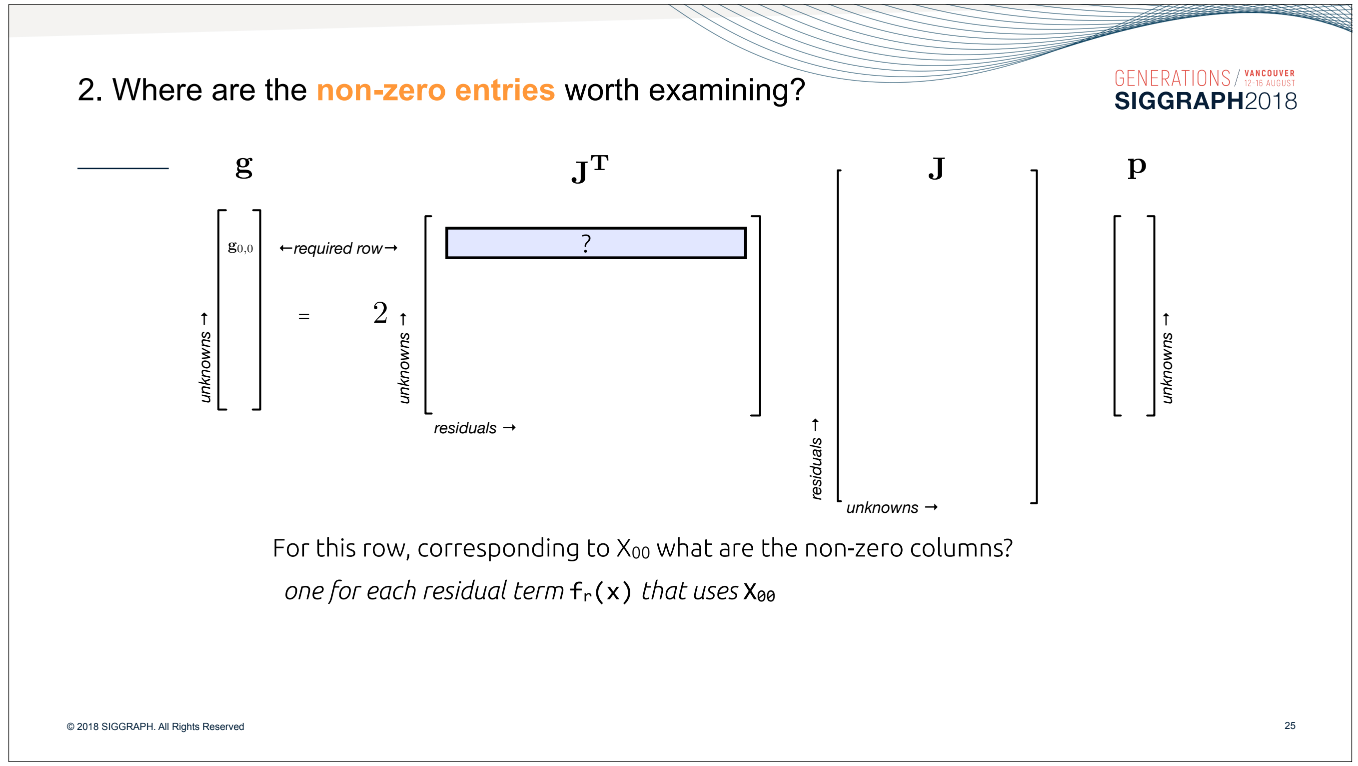
2. Where are the **non-zero entries** worth examining?



For this row, corresponding to X_{00} what are the non-zero columns?

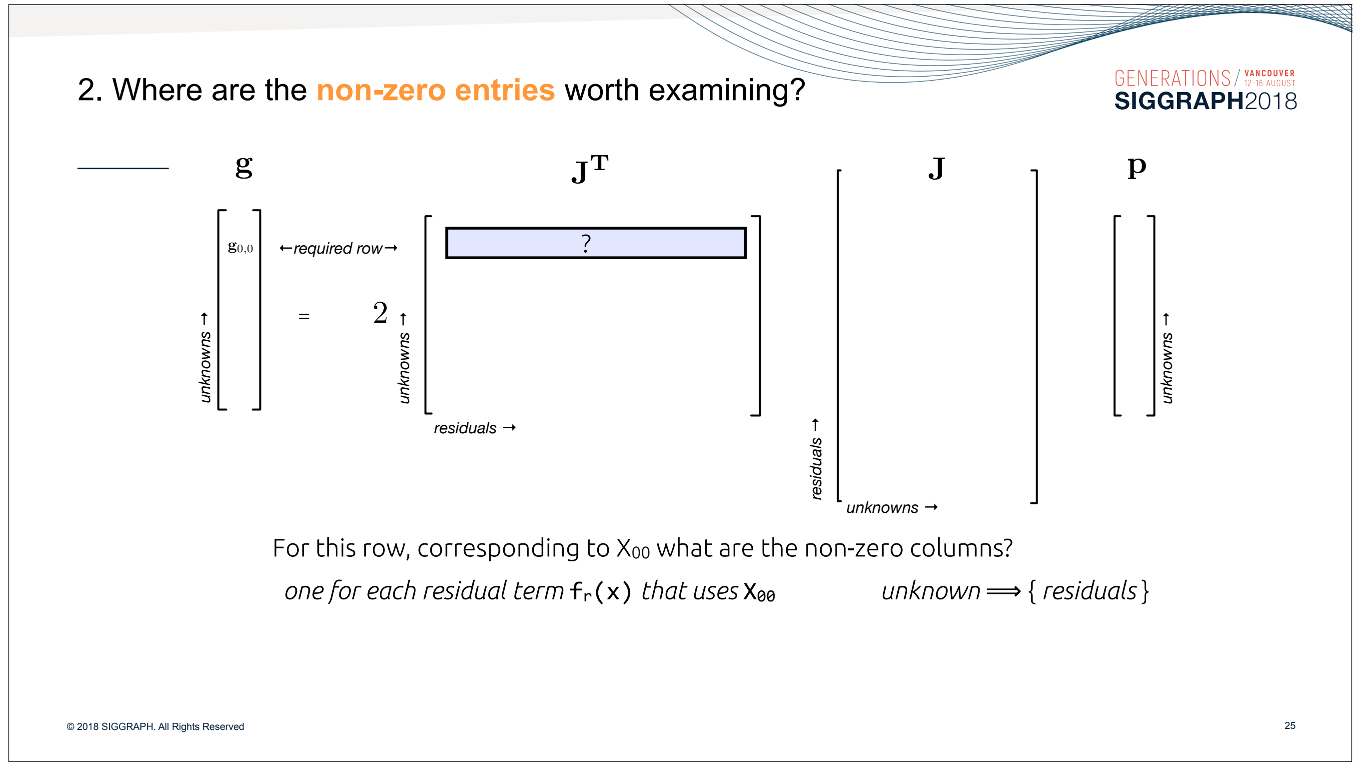
Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
“what are the non-zero columns related this unknown”
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.



Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
“what are the non-zero columns related this unknown”
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

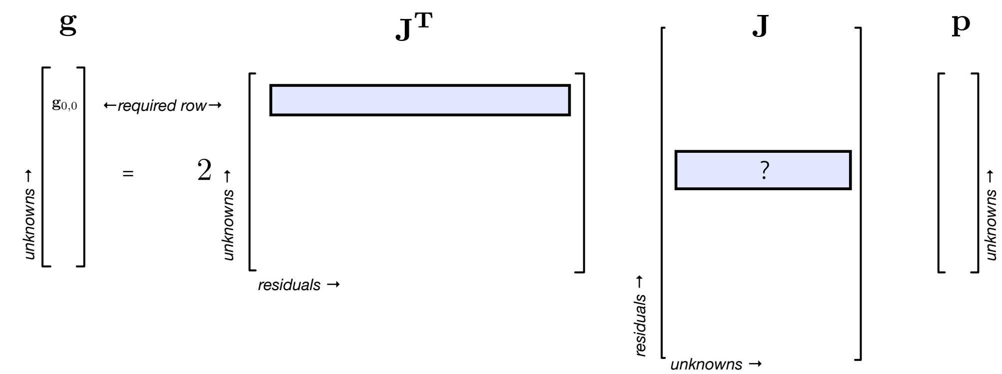
<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.



Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
“what are the non-zero columns related this unknown”
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.

2. Where are the **non-zero entries** worth examining?



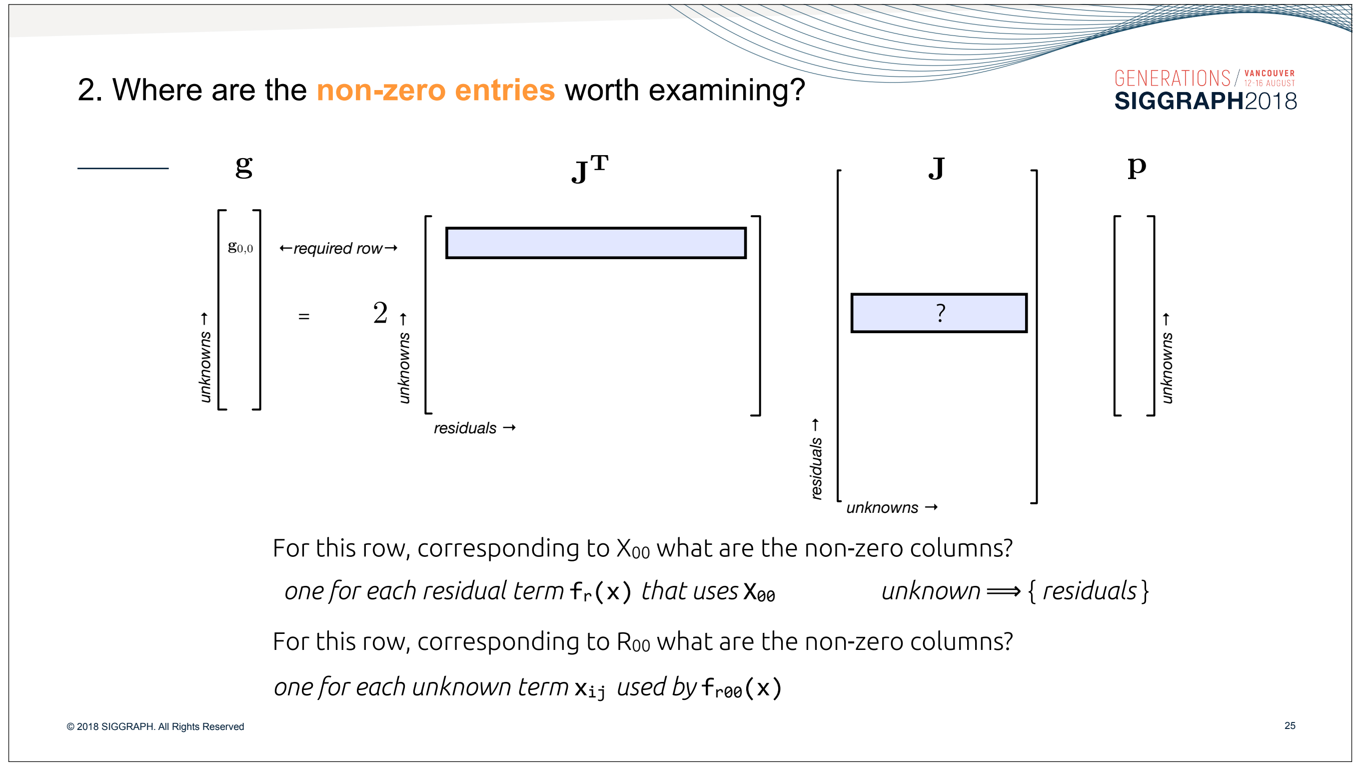
For this row, corresponding to X_{00} what are the non-zero columns?

one for each residual term $f_r(x)$ *that uses* X_{00} *unknown* $\Longrightarrow$ { *residuals* }

For this row, corresponding to R_{00} what are the non-zero columns?

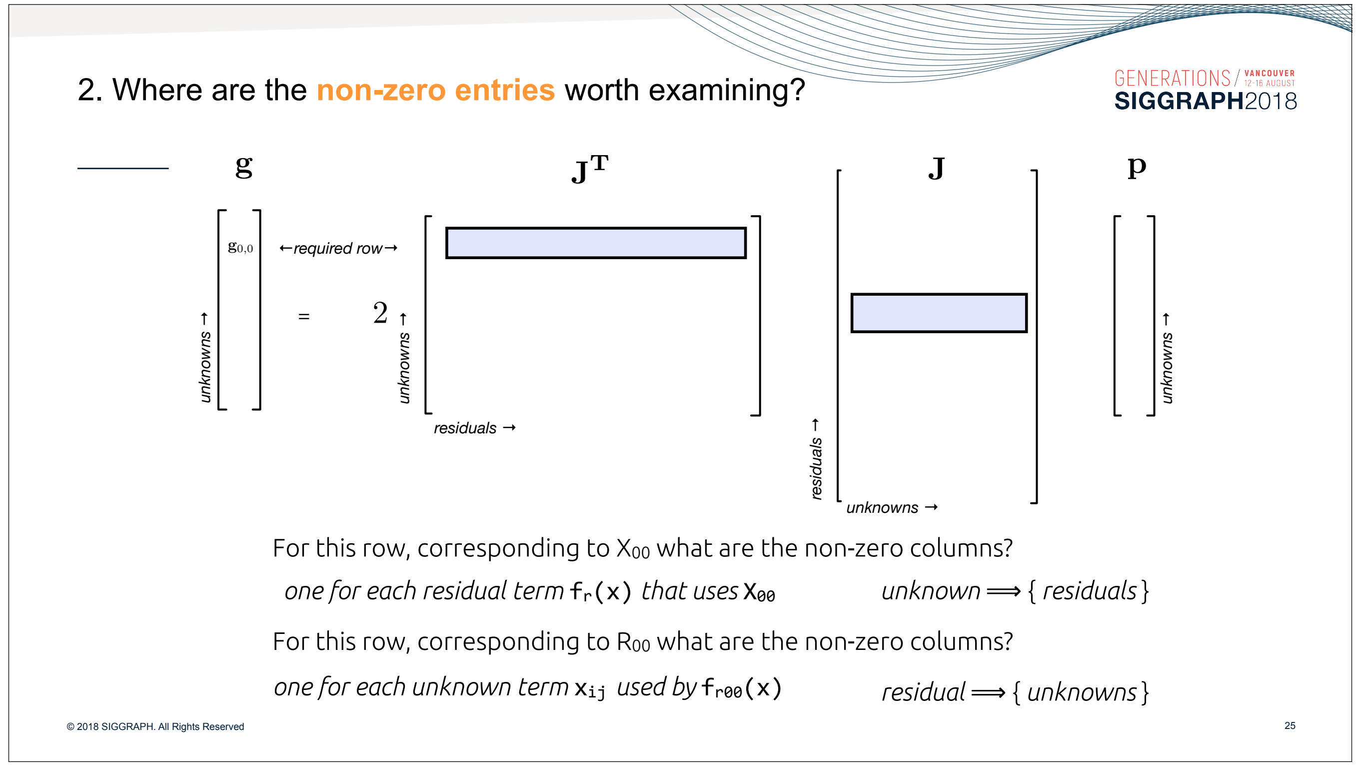
Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
"what are the non-zero columns related this unknown"
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.



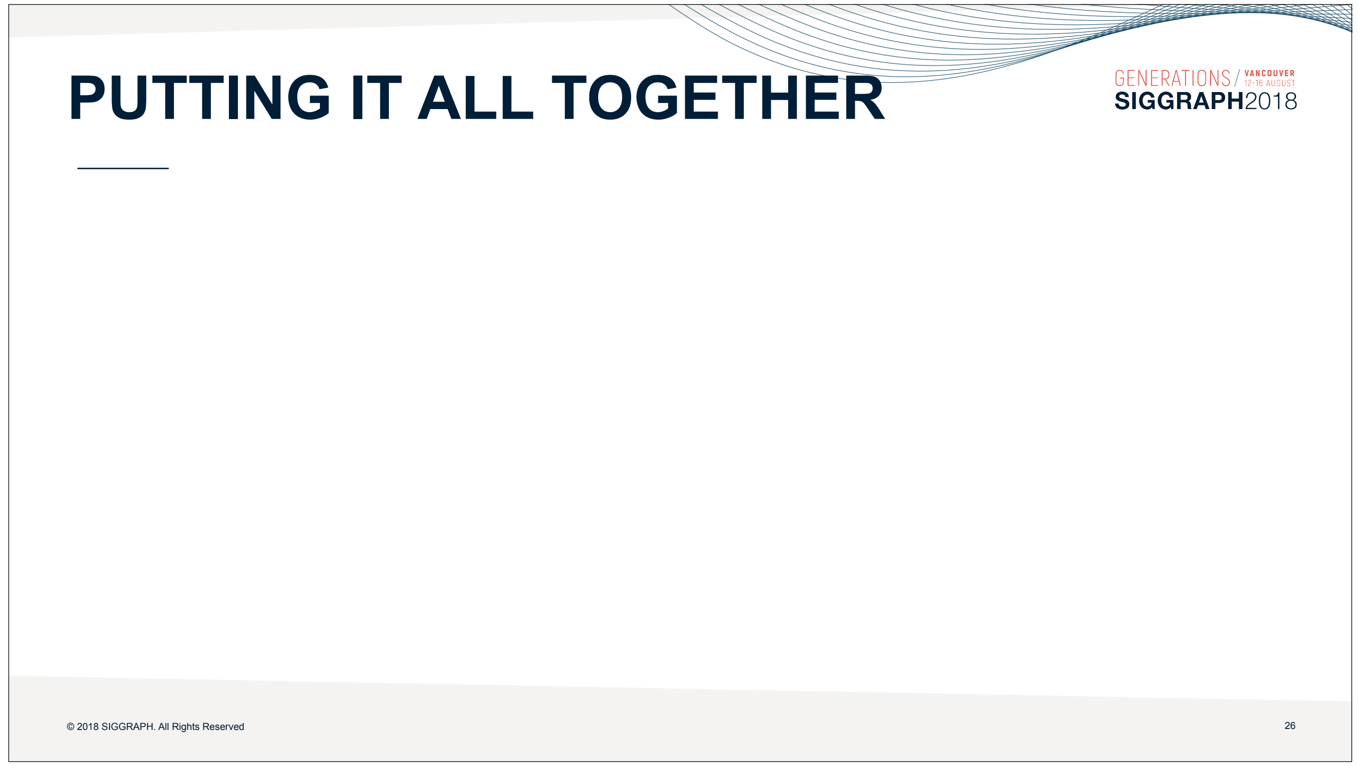
Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
“what are the non-zero columns related this unknown”
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.



Our second problem is to find the non-zeros needed to calculate the matrix products.
<> For a particular matrix product, We work left to right, identifying the non-zeros we need.
For example, in this row of J transpose, we ask
“what are the non-zero columns related this unknown”
<> These are the columns corresponding to the residuals that use that unknown.
<> to do this we need to compute a mapping from an unknown to the residuals that use it.

<> We can identify non-zeros in J as well.
<> This is a very similar problem, but because of the transpose we have one non-zero column for each unknown used by a residual.
<> This requires a map from residual to unknown.
In the image case, we can derive this information from the stencil access patterns, and in the mesh cases, we can recover it from the connectivity information itself. We provide more details in the paper.



Finally we compose the pieces.

<> We use our unknown<->residual mappings to find non-zero partial derivatives and come up with a matrix-free equation for a single entry of the output, treating the derivatives as placeholders

<> We then use compile-time auto diff to generate an expression for each non-zero partial derivative, slotting them into the equation from step 1.

<> Finally, we parallelize across outputs of the matrix product. For large problems this easily saturates even high-end GPUs.

PUTTING IT ALL TOGETHER

1. Where are the non-zeros in this expression?

Use dependency analysis to find them, and generate multiplication expression using them.

$$\mathbf{g}_{0,0} = 2\frac{\mathbf{dfit}_{0,0}}{\mathbf{dx}_{0,0}}\frac{\mathbf{dfit}_{0,0}}{\mathbf{dx}_{0,0}}\mathbf{p}_{0,0} + 2\underbrace{\frac{\mathrm{dh\_reg}_{0,0}}{\mathbf{dx}_{0,0}}}_{\text{from } \mathbf{J}^{\mathbf{T}}}\Big(\underbrace{\frac{\mathrm{dh\_reg}_{0,0}}{\mathbf{dx}_{0,0}}\mathbf{p}_{0,0} + \frac{\mathrm{dh\_reg}_{0,0}}{\mathbf{dx}_{1,0}}\mathbf{p}_{1,0}}_{\text{from } \mathbf{J}}\Big) + \ldots$$

Finally we compose the pieces.

<> We use our unknown<->residual mappings to find non-zero partial derivatives and come up with a matrix-free equation for a single entry of the output, treating the derivatives as placeholders

<> We then use compile-time auto diff to generate an expression for each non-zero partial derivative, slotting them into the equation from step 1.

<> Finally, we parallelize across outputs of the matrix product. For large problems this easily saturates even high-end GPUs.

PUTTING IT ALL TOGETHER

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

1. Where are the non-zeros in this expression?

Use dependency analysis to find them, and generate multiplication expression using them.

$$\mathbf{g}_{0,0} = 2\frac{\mathrm{dfit}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\frac{\mathrm{dfit}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\mathbf{p}_{0,0} + 2\underbrace{\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}}_{\text{from}\,\mathbf{J}^{\mathbf{T}}}(\underbrace{\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\mathbf{p}_{0,0} + \frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{1,0}}\mathbf{p}_{1,0}}_{\text{from}\,\mathbf{J}}) + \ldots$$

2. What are the values of the non-zeroes?

Use automatic differentiation.

$$= 2\mathrm{w\_fit}^2\mathbf{p}_{0,0} + 2\mathrm{w\_reg}(\mathrm{w\_reg}\mathbf{p}_{0,0} + -\mathrm{w\_reg}\mathbf{p}_{1,0}) + \ldots$$

Finally we compose the pieces.

<> We use our unknown<->residual mappings to find non-zero partial derivatives and come up with a matrix-free equation for a single entry of the output, treating the derivatives as placeholders

<> We then use compile-time auto diff to generate an expression for each non-zero partial derivative, slotting them into the equation from step 1.

<> Finally, we parallelize across outputs of the matrix product. For large problems this easily saturates even high-end GPUs.

PUTTING IT ALL TOGETHER

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

1. Where are the non-zeros in this expression?

Use dependency analysis to find them, and generate multiplication expression using them.

$$\mathbf{g}_{0,0} = 2\frac{\mathrm{dfit}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\frac{\mathrm{dfit}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\mathbf{p}_{0,0} + 2\underbrace{\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}}_{\text{from } \mathbf{J}^{\mathrm{T}}}(\underbrace{\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\mathbf{p}_{0,0} + \frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{1,0}}\mathbf{p}_{1,0}}_{\text{from } \mathbf{J}}) + \ldots$$

2. What are the values of the non-zeroes?

Use automatic differentiation.

$$= 2\mathrm{w\_fit}^2\mathbf{p}_{0,0} + 2\mathrm{w\_reg}(\mathrm{w\_reg}\mathbf{p}_{0,0} + -\mathrm{w\_reg}\mathbf{p}_{1,0}) + \ldots$$

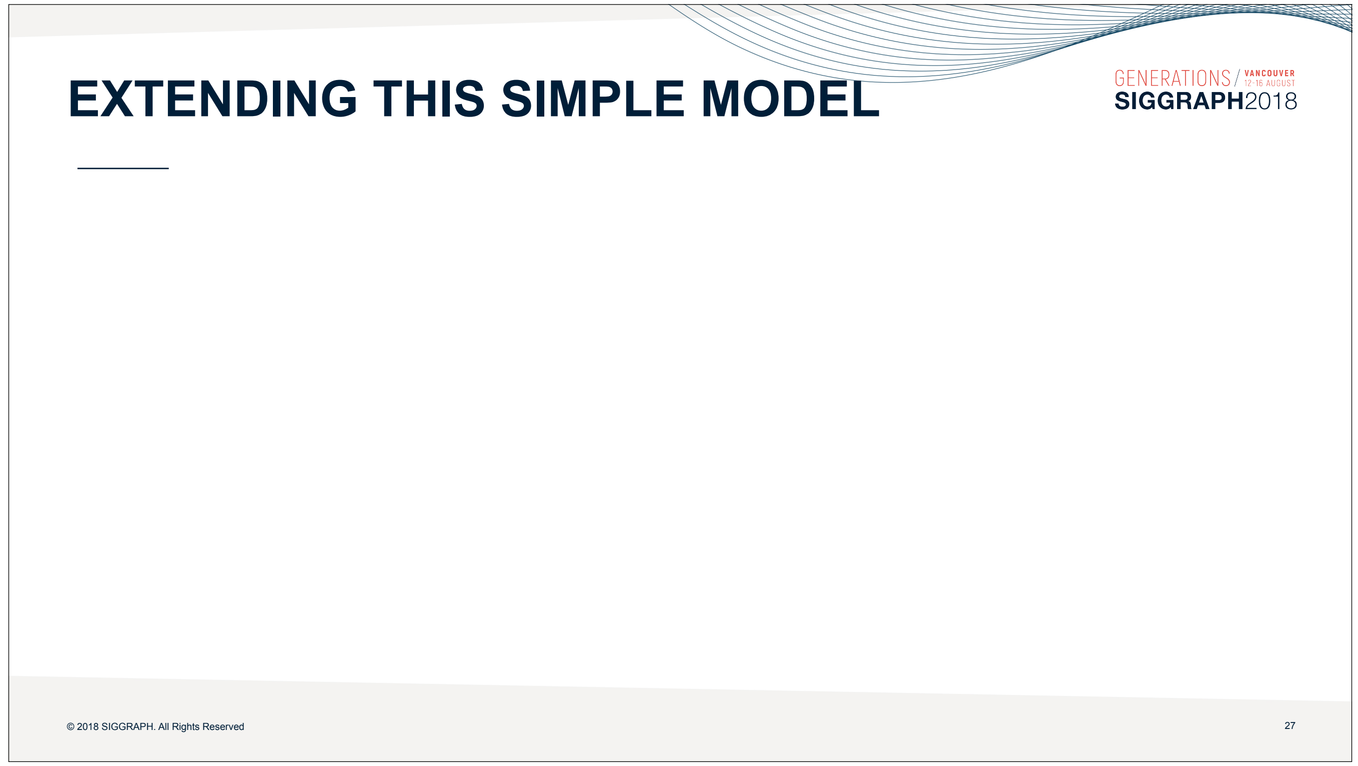
3. Parallelize across outputs.

Finally we compose the pieces.

<> We use our unknown<->residual mappings to find non-zero partial derivatives and come up with a matrix-free equation for a single entry of the output, treating the derivatives as placeholders

<> We then use compile-time auto diff to generate an expression for each non-zero partial derivative, slotting them into the equation from step 1.

<> Finally, we parallelize across outputs of the matrix product. For large problems this easily saturates even high-end GPUs.



In the paper, we show how we can expand this approach to handle more domains and solver techniques.

In particular, we

<> Demonstrate how you can also write energies over mixed domains of meshes and images.

<> Show how we can handle different solver variants of Gauss-Newton like Levenberg-Marquardt.

<> Explore the tradeoff between using completely matrix-free products on one hand, and selectively pre-computing some parts of the matrix products on the other, to improve performance.

EXTENDING THIS SIMPLE MODEL

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

1. Energies can be defined over mixed domains (meshes + images).

In the paper, we show how we can expand this approach to handle more domains and solver techniques.

In particular, we

<> Demonstrate how you can also write energies over mixed domains of meshes and images.

<> Show how we can handle different solver variants of Gauss-Newton like Levenberg-Marquardt.

<> Explore the tradeoff between using completely matrix-free products on one hand, and selectively pre-computing some parts of the matrix products on the other, to improve performance.

EXTENDING THIS SIMPLE MODEL

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

1. Energies can be defined over mixed domains (meshes + images).
2. Support of Gauss-Newton variants, like Levenberg-Marquardt.

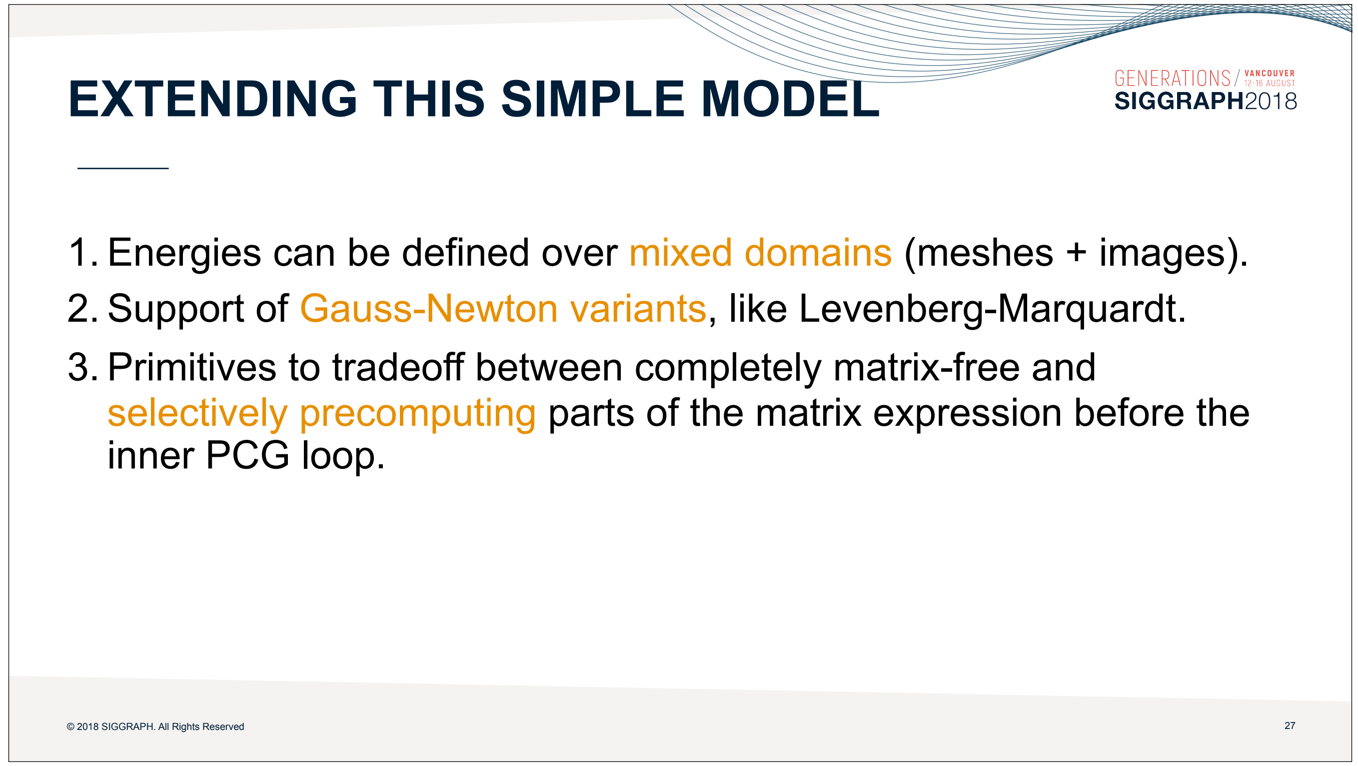
In the paper, we show how we can expand this approach to handle more domains and solver techniques.

In particular, we

<> Demonstrate how you can also write energies over mixed domains of meshes and images.

<> Show how we can handle different solver variants of Gauss-Newton like Levenberg-Marquardt.

<> Explore the tradeoff between using completely matrix-free products on one hand, and selectively pre-computing some parts of the matrix products on the other, to improve performance.



In the paper, we show how we can expand this approach to handle more domains and solver techniques.

In particular, we

<> Demonstrate how you can also write energies over mixed domains of meshes and images.

<> Show how we can handle different solver variants of Gauss-Newton like Levenberg-Marquardt.

<> Explore the tradeoff between using completely matrix-free products on one hand, and selectively pre-computing some parts of the matrix products on the other, to improve performance.

RESULTS

Now that we have a flavor for what Opt does, we can evaluate it along several axes

First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took

OPT IS EXPRESSIVE

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took

OPT IS **EXPRESSIVE**

GENERATIONS / VANCOUVER
SIGGRAPH2018

First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took

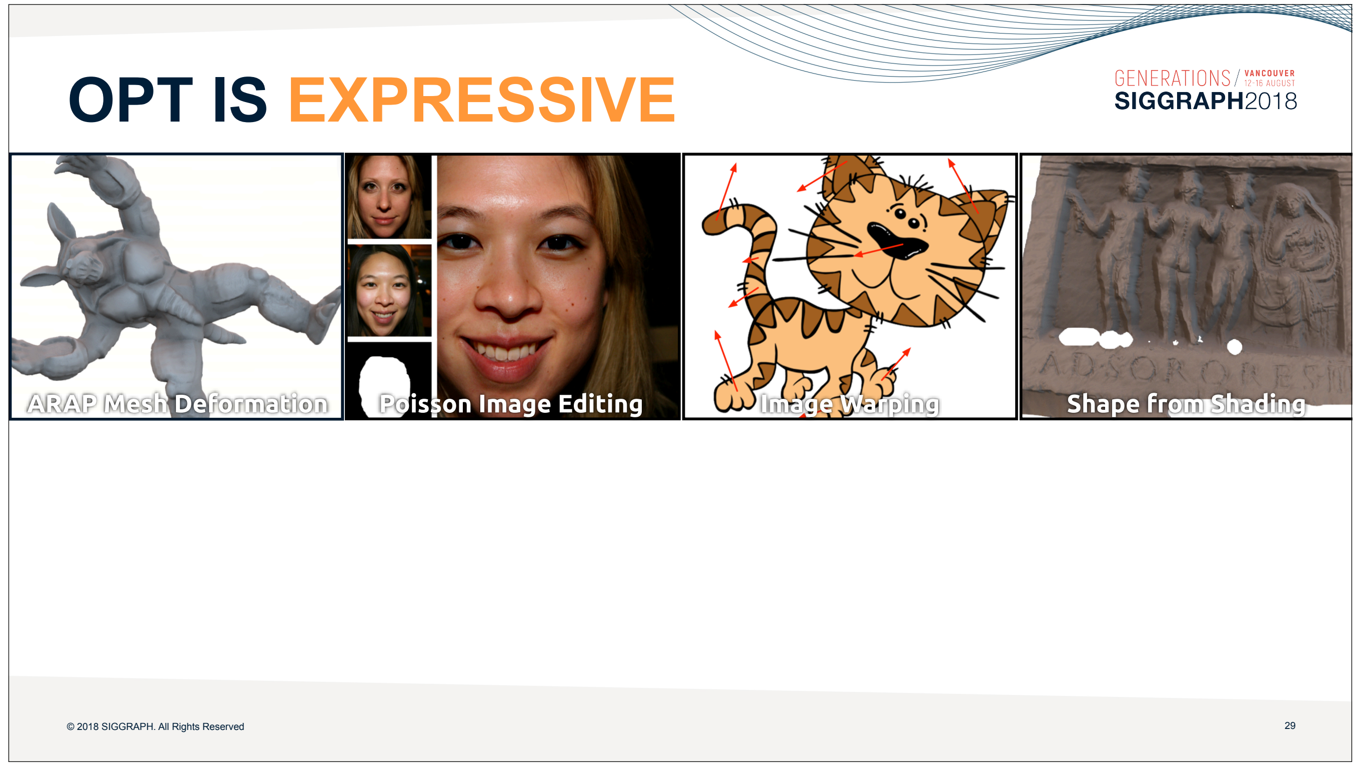
OPT IS EXPRESSIVE

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took

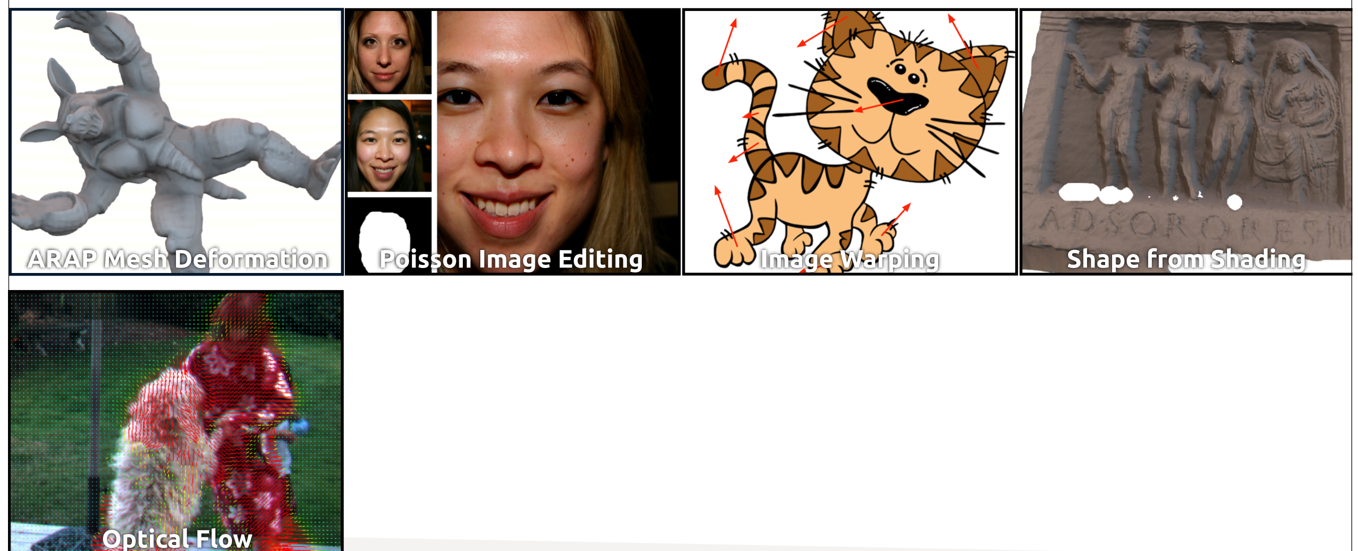


First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took

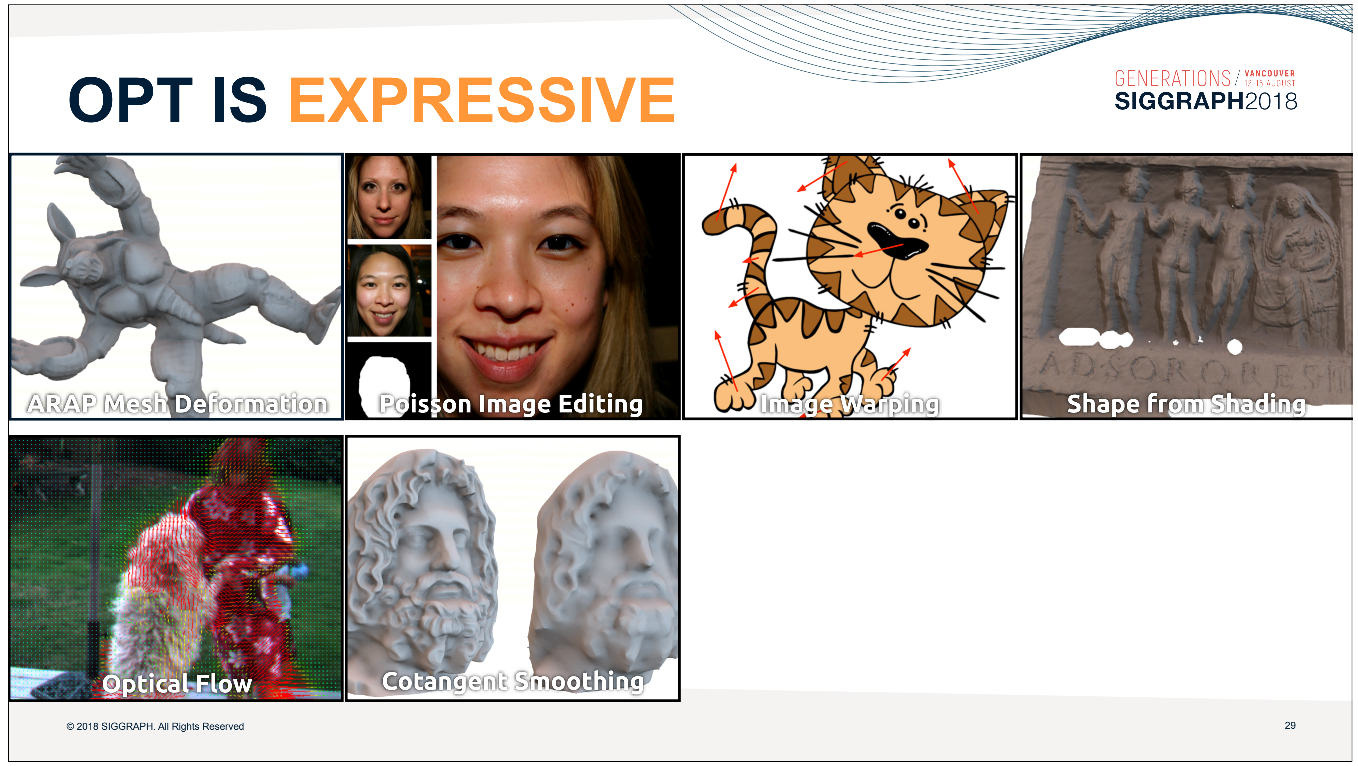
OPT IS EXPRESSIVE



First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

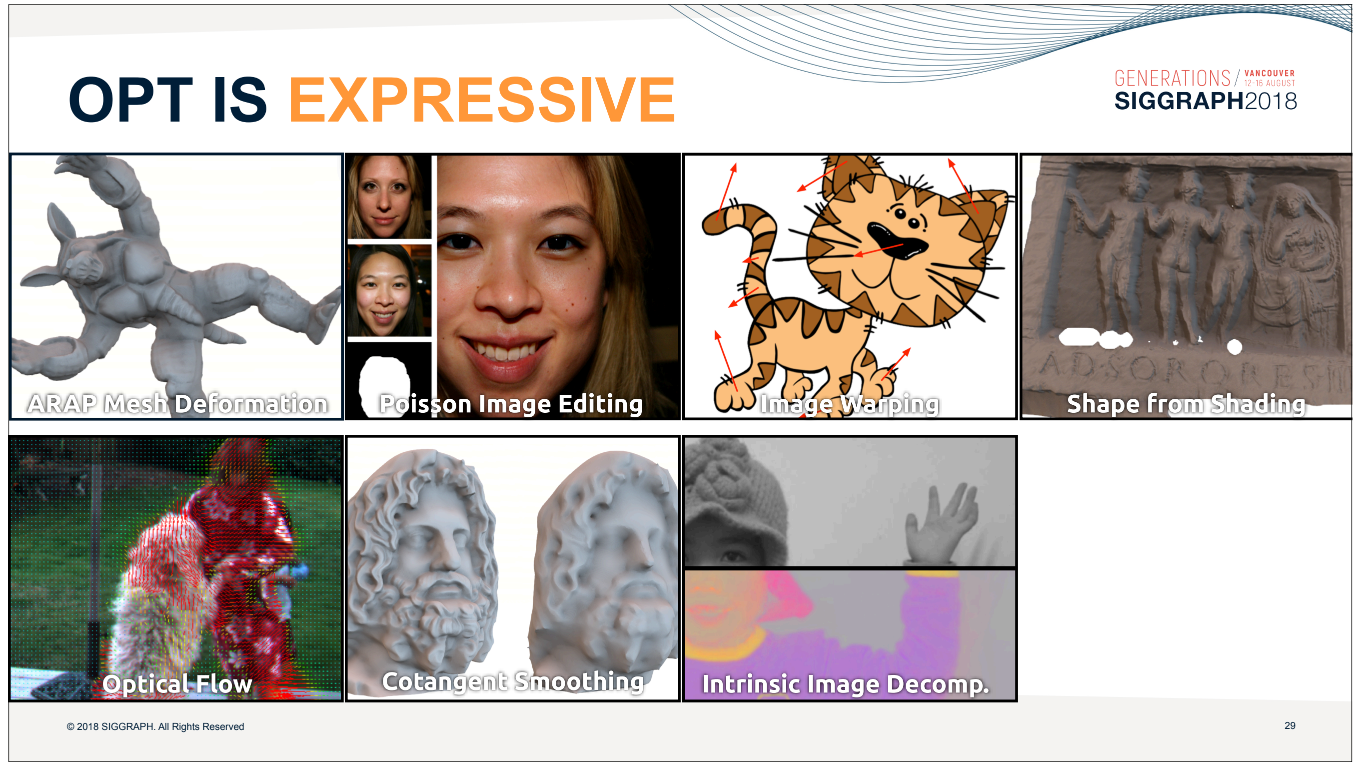
all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took



First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

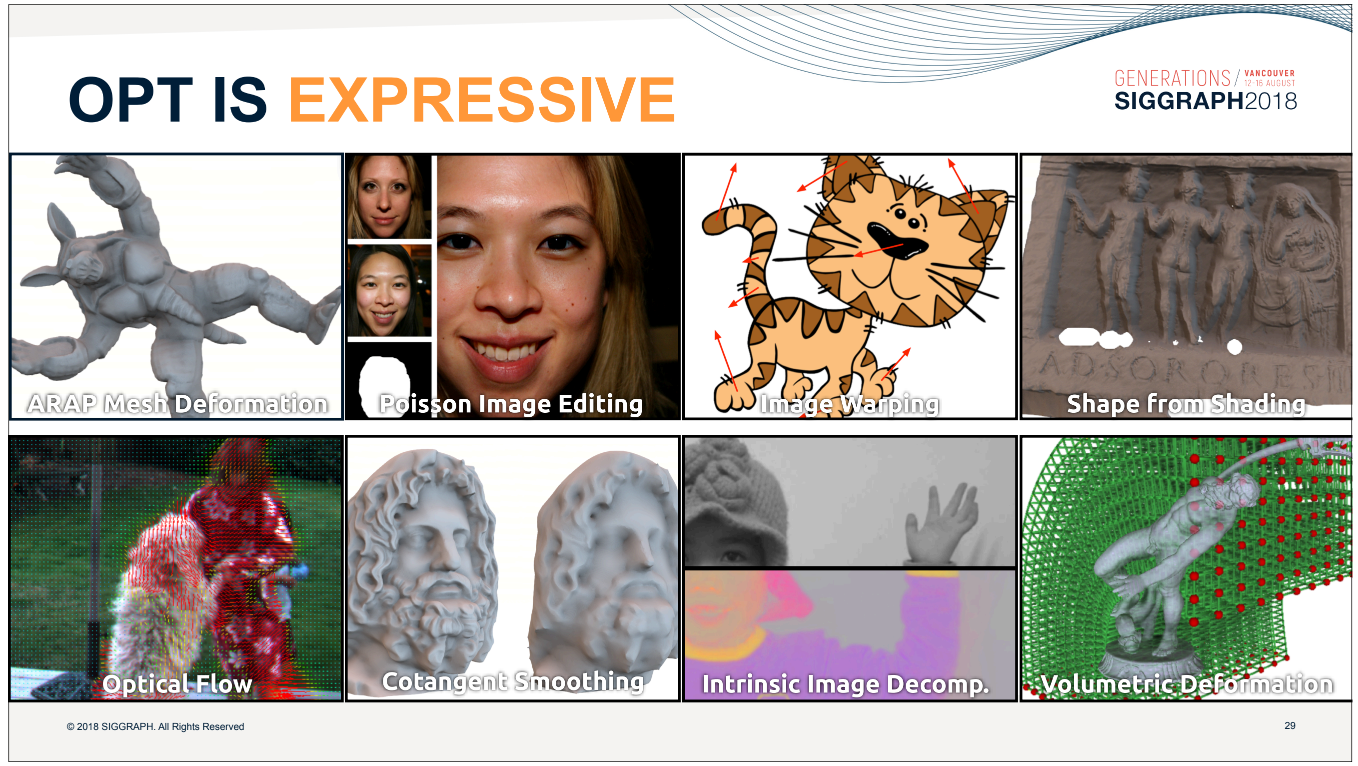
all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took



First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took



First, Opt is expressive enough to handle a wide range of problems. We implemented

<> As-rigid-as-possible Mesh Deformation
<> Poisson Image Editing
<> Image Warping
<> Shape from Shading
<> Optical Flow
<> Cotangent Mesh Smoothing
<> Intrinsic Image Decomposition
<> and Volumetric Mesh Deformation

all within Opt. This is only a subset of the applications we implemented for the paper. Each of these solvers only took

a handful of lines of code to implement in Opt, indeed all but one were less than 40 lines.

OPT IS CONCISE

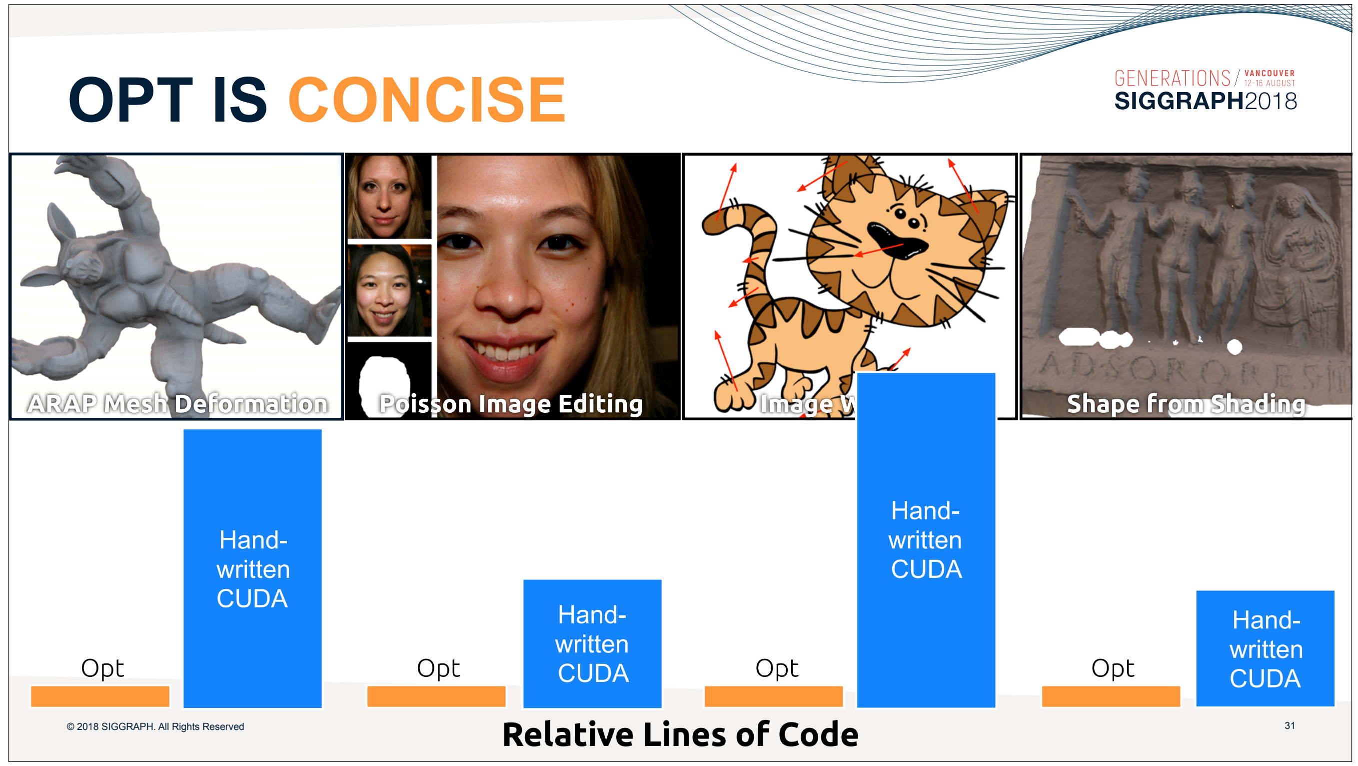
GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Relative Lines of Code

Four of these were previously implemented laboriously by hand in CUDA;

<> for these we can directly compare solver length and see Opt code is far more compact. Every solver is at least 4.5x more verbose in CUDA, and the worst is over 13x longer!

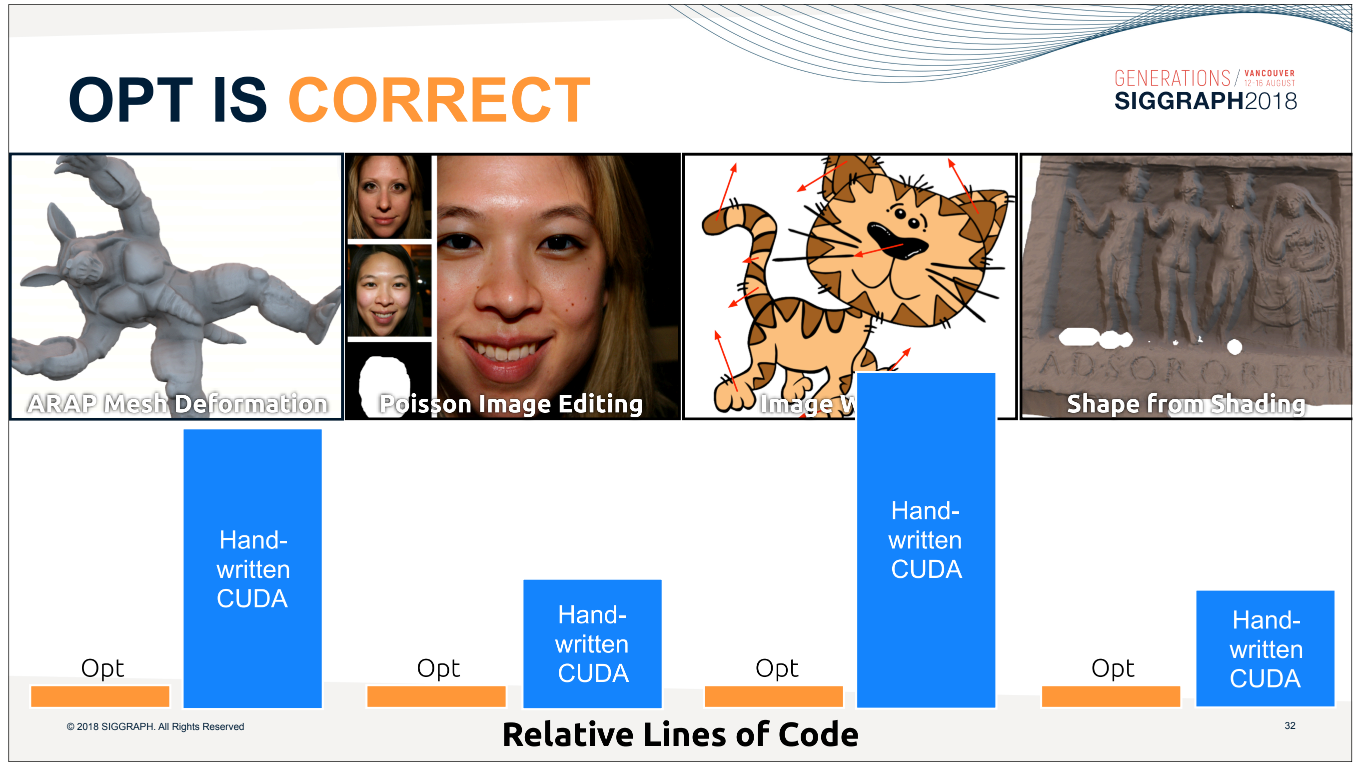
Note that an additional energy term requires adding a couple of lines of Opt code, but in the handwritten solvers requires surgery on at least three different pieces of code, massively increasing chances for mismatch errors or problems with by-hand differentiation.



Four of these were previously implemented laboriously by hand in CUDA;

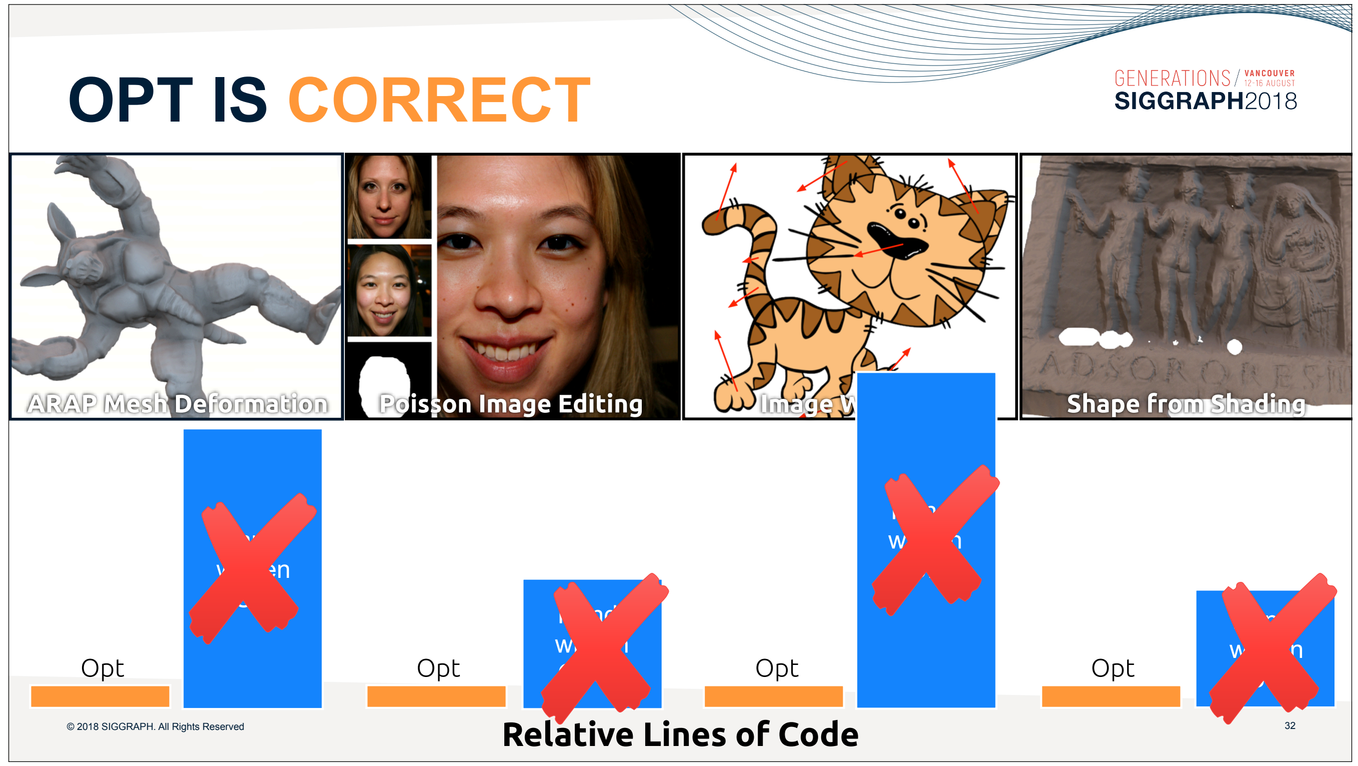
<> for these we can directly compare solver length and see Opt code is far more compact. Every solver is at least 4.5x more verbose in CUDA, and the worst is over 13x longer!

Note that an additional energy term requires adding a couple of lines of Opt code, but in the handwritten solvers requires surgery on at least three different pieces of code, massively increasing chances for mismatch errors or problems with by-hand differentiation.



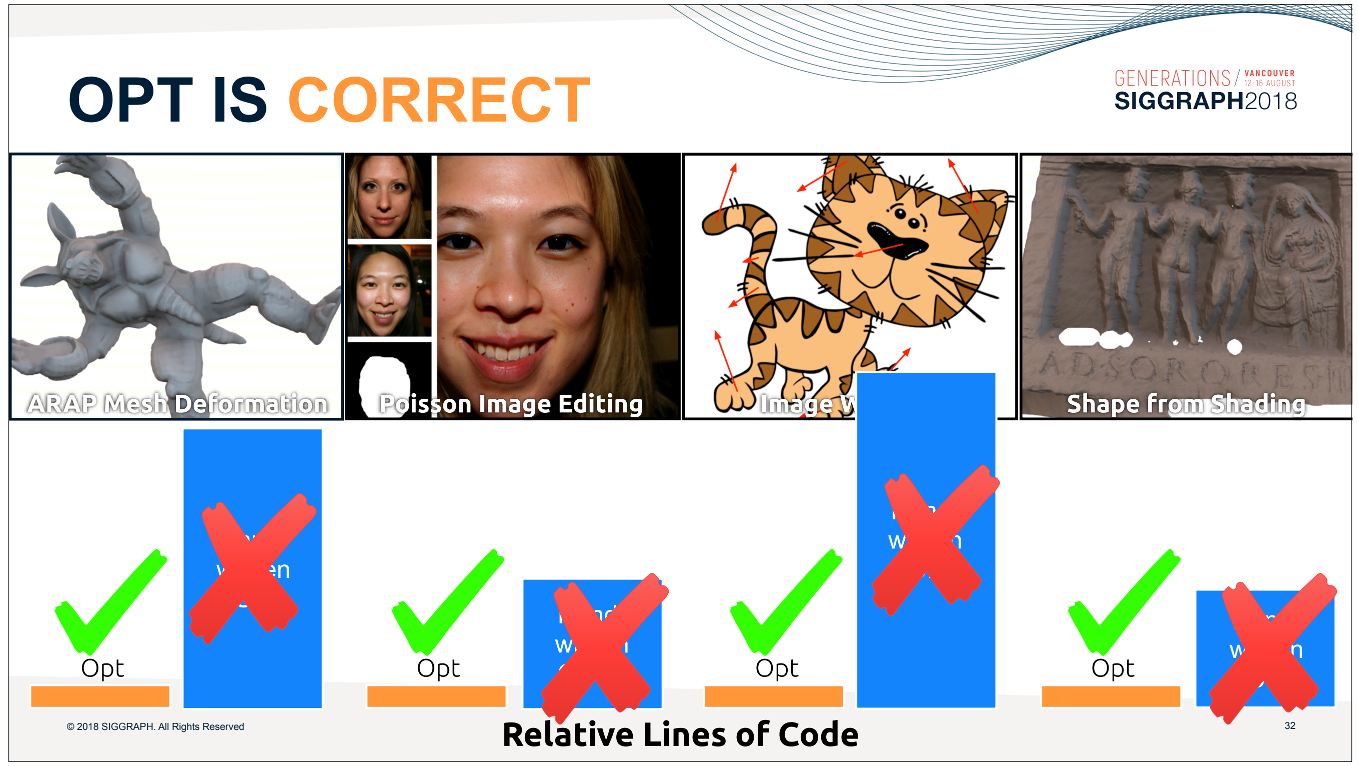
In fact, every single handwritten solver we compared against had at least some error in the derivative terms, either in the calculus or boundary conditions, which negatively impacted solver convergence until we fixed them.

<> Opt, by offloading the differentiation labor and code correspondence bookkeeping to the compiler, generates solvers that are correct by construction.



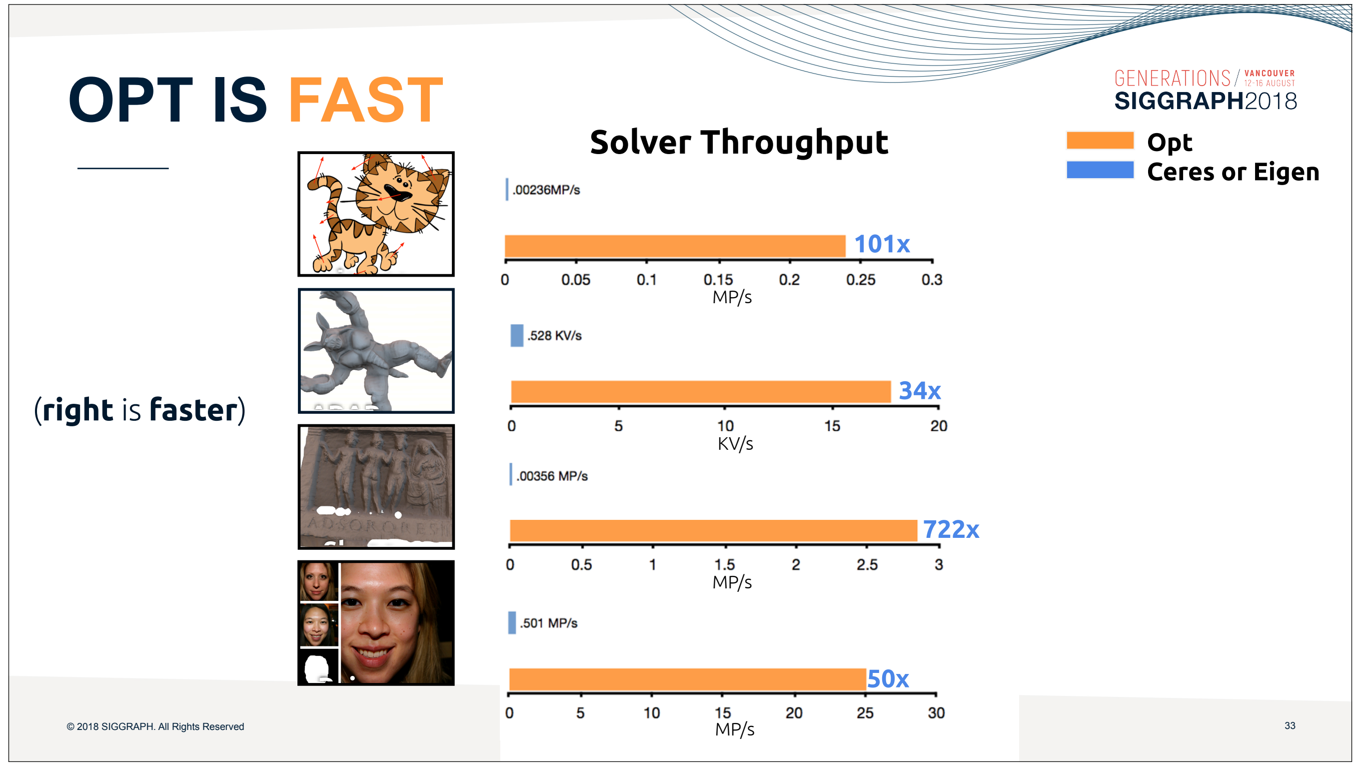
In fact, every single handwritten solver we compared against had at least some error in the derivative terms, either in the calculus or boundary conditions, which negatively impacted solver convergence until we fixed them.

<> Opt, by offloading the differentiation labor and code correspondence bookkeeping to the compiler, generates solvers that are correct by construction.



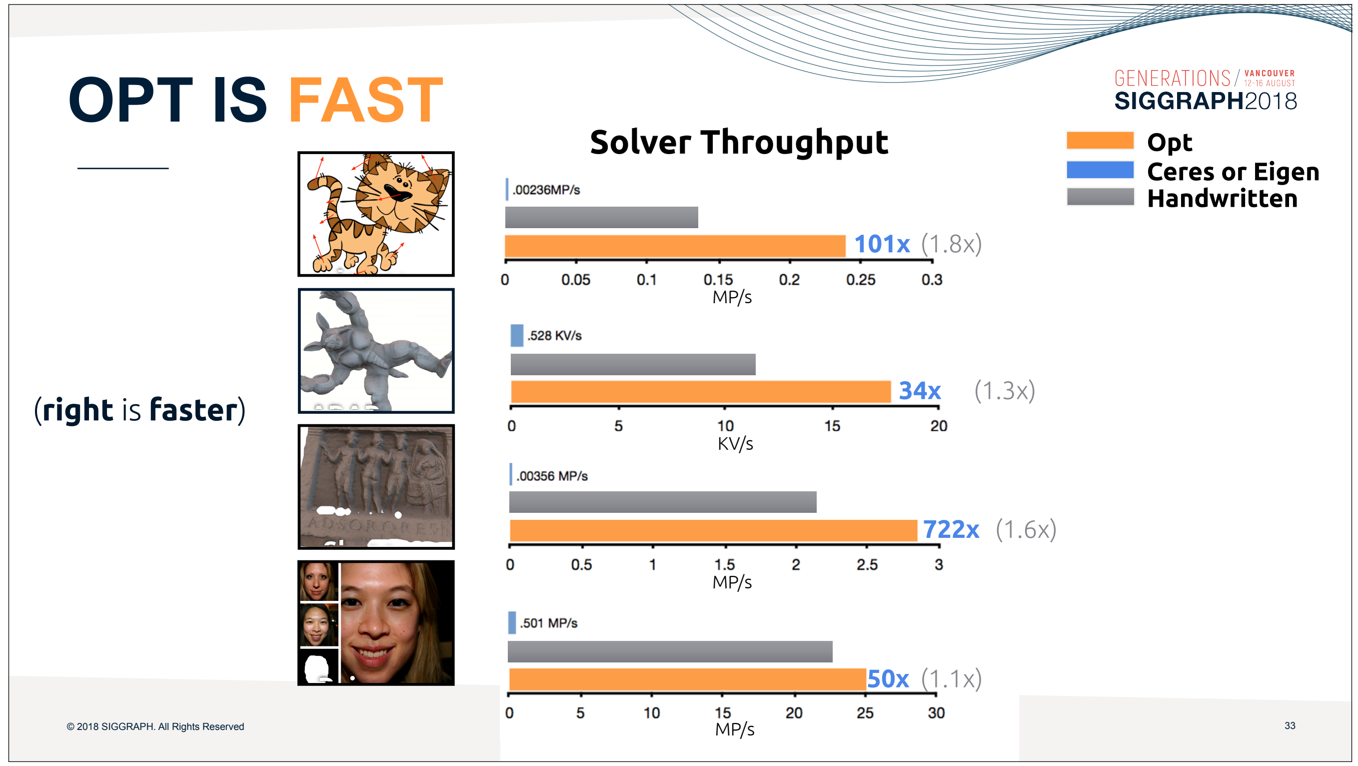
In fact, every single handwritten solver we compared against had at least some error in the derivative terms, either in the calculus or boundary conditions, which negatively impacted solver convergence until we fixed them.

<> Opt, by offloading the differentiation labor and code correspondence bookkeeping to the compiler, generates solvers that are correct by construction.



And performance wise, Opt blows away other high level solvers, by multiple orders of magnitude. Here we have the throughput of solvers implemented in Opt (shown in orange) and Ceres (or Eigen in the case of Poisson Image Editing), shown in blue. The lowest speedup we get with Opt is 33x on Mesh Deformation, the highest is over 700 times on our shape from shading implementation.

<> It even beats out all of the handwritten solvers that we compared against (shown in gray), ranging from 1.1x to 1.8x faster. Our compiler does some algebraic simplifications and boundary handling that the original authors of the handwritten implementations either did notice or did not bother with, you can read more about these in the paper.



And performance wise, Opt blows away other high level solvers, by multiple orders of magnitude. Here we have the throughput of solvers implemented in Opt (shown in orange) and Ceres (or Eigen in the case of Poisson Image Editing), shown in blue. The lowest speedup we get with Opt is 33x on Mesh Deformation, the highest is over 700 times on our shape from shading implementation.

<> It even beats out all of the handwritten solvers that we compared against (shown in gray), ranging from 1.1x to 1.8x faster. Our compiler does some algebraic simplifications and boundary handling that the original authors of the handwritten implementations either did notice or did not bother with, you can read more about these in the paper.

OPT SCALES

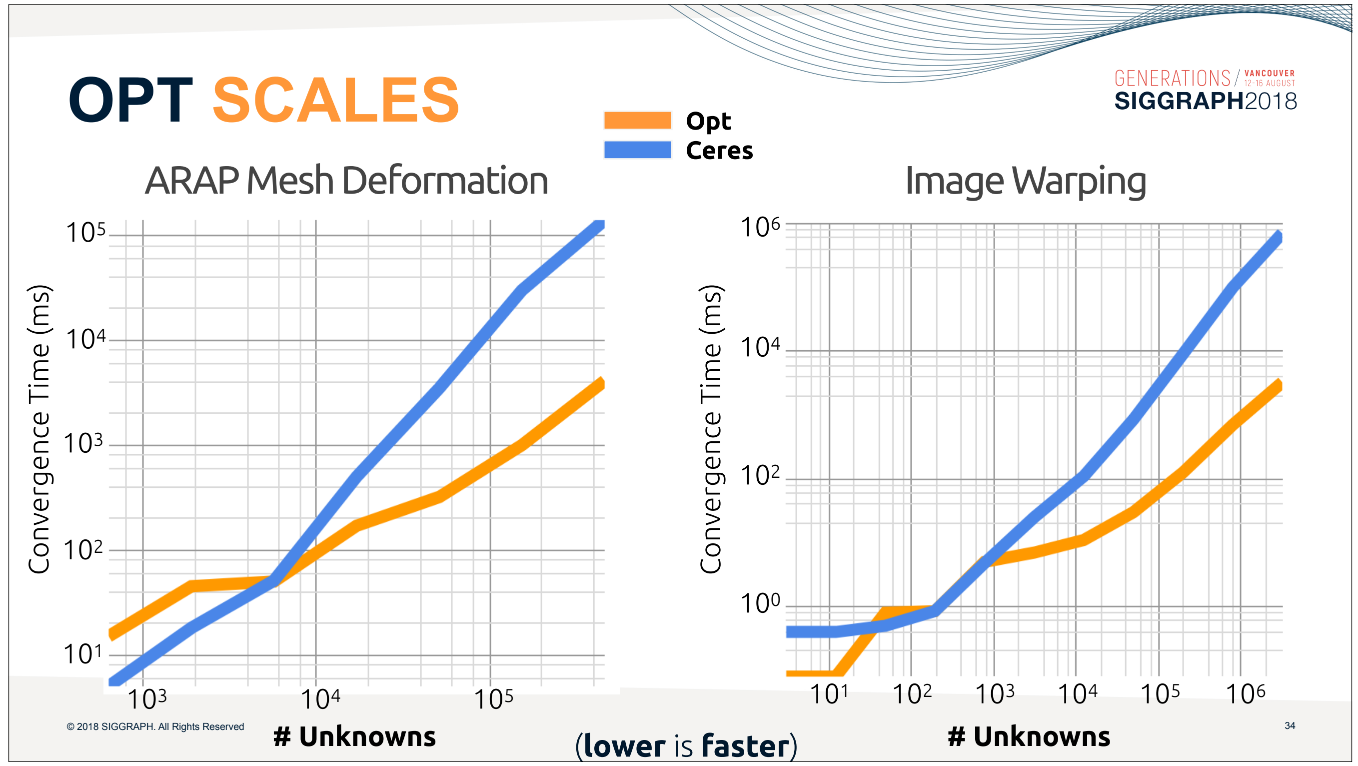
GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

And here we see the performance of Opt versus Ceres as we vary problem size. <>

Again, Opt is in orange and Ceres is in Blue. We are charting Convergence time vs # of unknowns, on a log-log chart, so large differences are quite compressed.

At low unknown count (in the several hundreds) high-level CPU solvers that do not have to transfer data back and forth from the GPU co-processor compare favorably to Opt,

<> but once we get into larger problems with thousands unknowns or more, where the massive parallelism of the GPU comes into play, Opt wins out in performance, and as size increases, so does the performance gap.

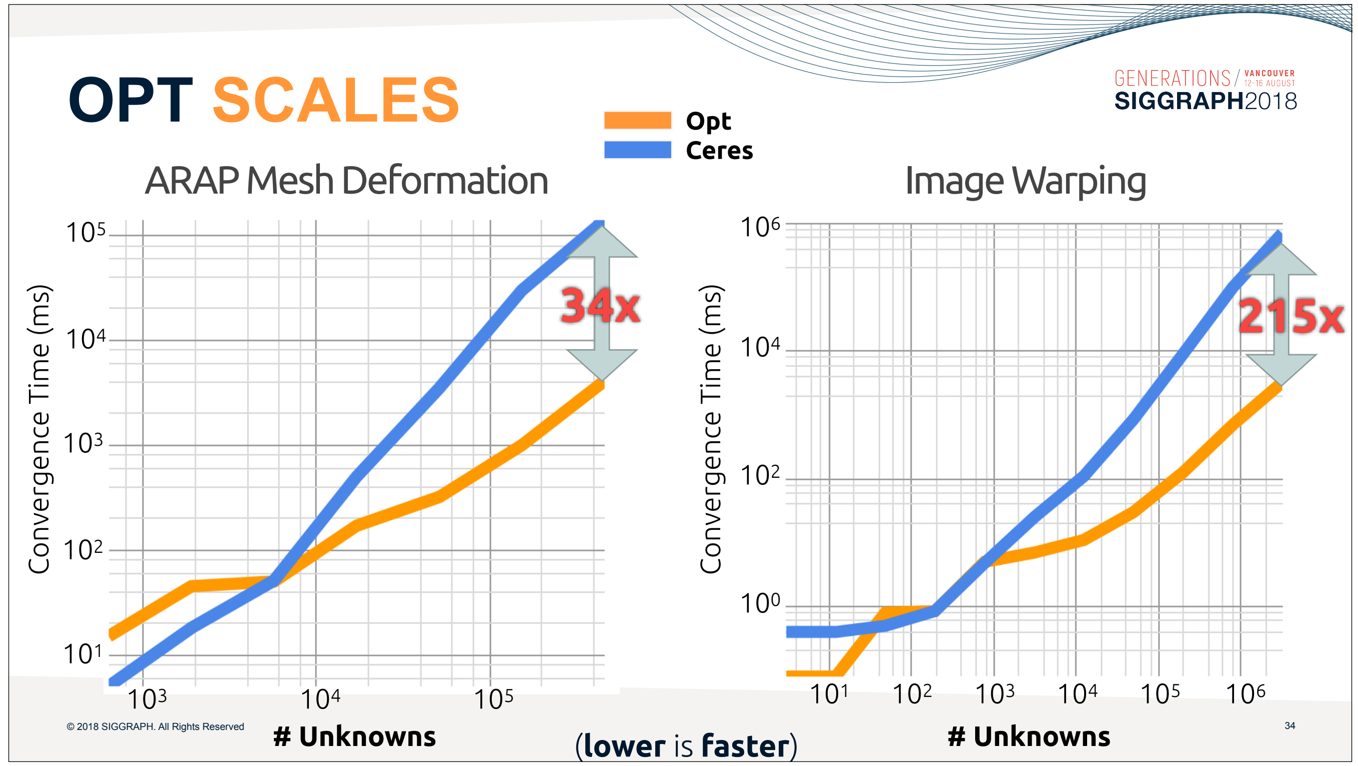


And here we see the performance of Opt versus Ceres as we vary problem size. <>

Again, Opt is in orange and Ceres is in Blue. We are charting Convergence time vs # of unknowns, on a log-log chart, so large differences are quite compressed.

At low unknown count (in the several hundreds) high-level CPU solvers that do not have to transfer data back and forth from the GPU co-processor compare favorably to Opt,

<> but once we get into larger problems with thousands unknowns or more, where the massive parallelism of the GPU comes into play, Opt wins out in performance, and as size increases, so does the performance gap.



And here we see the performance of Opt versus Ceres as we vary problem size. <>

Again, Opt is in orange and Ceres is in Blue. We are charting Convergence time vs # of unknowns, on a log-log chart, so large differences are quite compressed.

At low unknown count (in the several hundreds) high-level CPU solvers that do not have to transfer data back and forth from the GPU co-processor compare favorably to Opt,

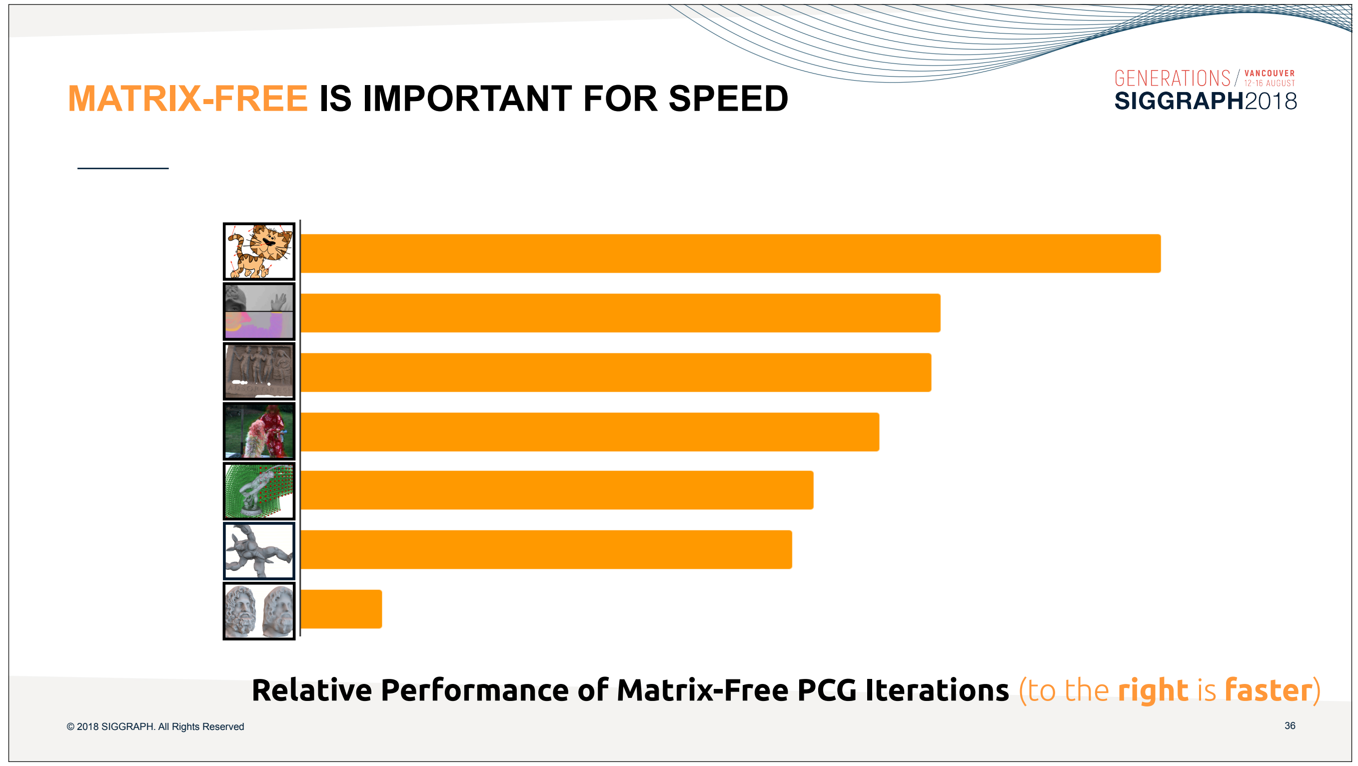
<> but once we get into larger problems with thousands unknowns or more, where the massive parallelism of the GPU comes into play, Opt wins out in performance, and as size increases, so does the performance gap.

MATRIX-FREE IS IMPORTANT FOR SPEED

MATRIX-FREE IS IMPORTANT FOR SPEED

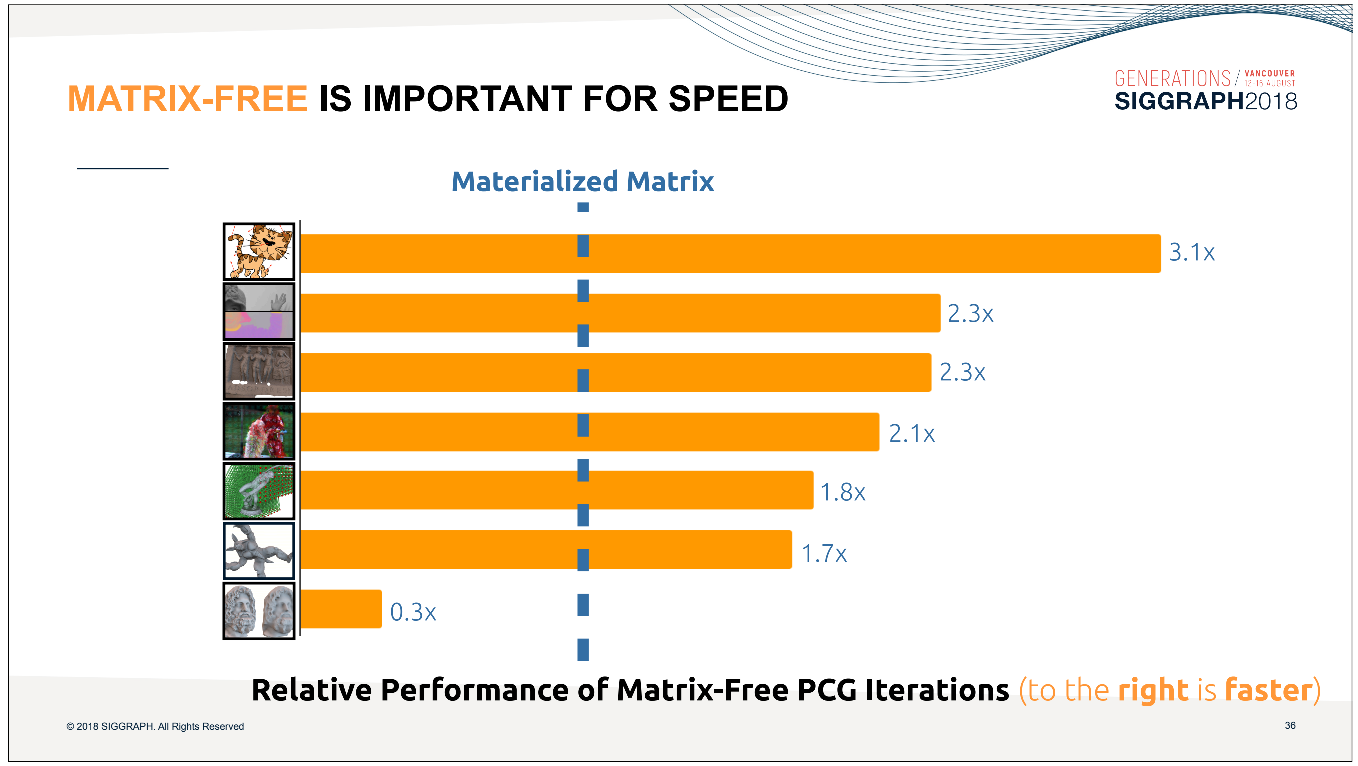
The ability for Opt to generate Matrix-Free code is important to get improved performance. Here we chart the

<> relative performance of our examples for doing PCG iterations using Opt-generated matrix free code, using PCG with a materialized JtJ matrix (in compressed-row storage form) as the baseline. For all but one of our problems we get a significant performance improvement by choosing the matrix free approach. And this chart doesn’t reflect the fact that materializing the matrix in the first place has a fixed overhead that our matrix-free code does not pay.



The ability for Opt to generate Matrix-Free code is important to get improved performance. Here we chart the

<> relative performance of our examples for doing PCG iterations using Opt-generated matrix free code, using PCG with a materialized JtJ matrix (in compressed-row storage form) as the baseline. For all but one of our problems we get a significant performance improvement by choosing the matrix free approach. And this chart doesn't reflect the fact that materializing the matrix in the first place has a fixed overhead that our matrix-free code does not pay.



The ability for Opt to generate Matrix-Free code is important to get improved performance. Here we chart the

<> relative performance of our examples for doing PCG iterations using Opt-generated matrix free code, using PCG with a materialized JtJ matrix (in compressed-row storage form) as the baseline. For all but one of our problems we get a significant performance improvement by choosing the matrix free approach. And this chart doesn't reflect the fact that materializing the matrix in the first place has a fixed overhead that our matrix-free code does not pay.

WHAT MAKES OPT FAST?

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

So ultimately, what makes Opt fast?

<> Well first, because of our restricted DSL and exploiting structure in the energy, we can generate code for the Jacobian that can take advantage of massive parallelism and wide-SIMD units, like on the GPU. This is responsible for a large factor of the speed difference between Opt and other high-level solvers

<> Second, we can go a step further and remove the potential overhead of materializing an explicit system matrix, which we just saw often leads to further performance improvements

WHAT MAKES OPT FAST?

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- GPU-ifies J construction

So ultimately, what makes Opt fast?

<> Well first, because of our restricted DSL and exploiting structure in the energy, we can generate code for the Jacobian that can take advantage of massive parallelism and wide-SIMD units, like on the GPU. This is responsible for a large factor of the speed difference between Opt and other high-level solvers

<> Second, we can go a step further and remove the potential overhead of materializing an explicit system matrix, which we just saw often leads to further performance improvements

WHAT MAKES OPT FAST?

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- GPU-ifies J construction
- Removes explicit matrix

So ultimately, what makes Opt fast?

<> Well first, because of our restricted DSL and exploiting structure in the energy, we can generate code for the Jacobian that can take advantage of massive parallelism and wide-SIMD units, like on the GPU. This is responsible for a large factor of the speed difference between Opt and other high-level solvers

<> Second, we can go a step further and remove the potential overhead of materializing an explicit system matrix, which we just saw often leads to further performance improvements

LIMITATIONS AND FUTURE WORK

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

Limitations | **Future Work**

Our design decisions provide us with some natural limitations

<> which suggests future work.

<> Our access patterns are restricted, which is the very thing ensuring we can do the data dependency analysis we need to to maximize parallelism. Future work would relax these restrictions

<> We allow for selectively precomputing parts of the Jacobian computation, which allows users to eke out extra performance, but it should be possible to generalize to a much wider range of schedules, like Halide.

<> We have fast matrix-free code generation for a small set of matrix calculus primitives, those necessary to generate Gauss-Newton-like solvers. Having a language that allows arbitrary tensor derivatives would open up a large set of new potential use-cases that are currently too laborious to explore.

Our design decisions provide us with some natural limitations

<> which suggests future work.

<> Our access patterns are restricted, which is the very thing ensuring we can do the data dependency analysis we need to to maximize parallelism. Future work would relax these restrictions

<> We allow for selectively precomputing parts of the Jacobian computation, which allows users to eke out extra performance, but it should be possible to generalize to a much wider range of schedules, like Halide.

<> We have fast matrix-free code generation for a small set of matrix calculus primitives, those necessary to generate Gauss-Newton-like solvers. Having a language that allows arbitrary tensor derivatives would open up a large set of new potential use-cases that are currently too laborious to explore.

LIMITATIONS AND FUTURE WORK

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Restricted access patterns

Limitations

- More access patterns

Future Work

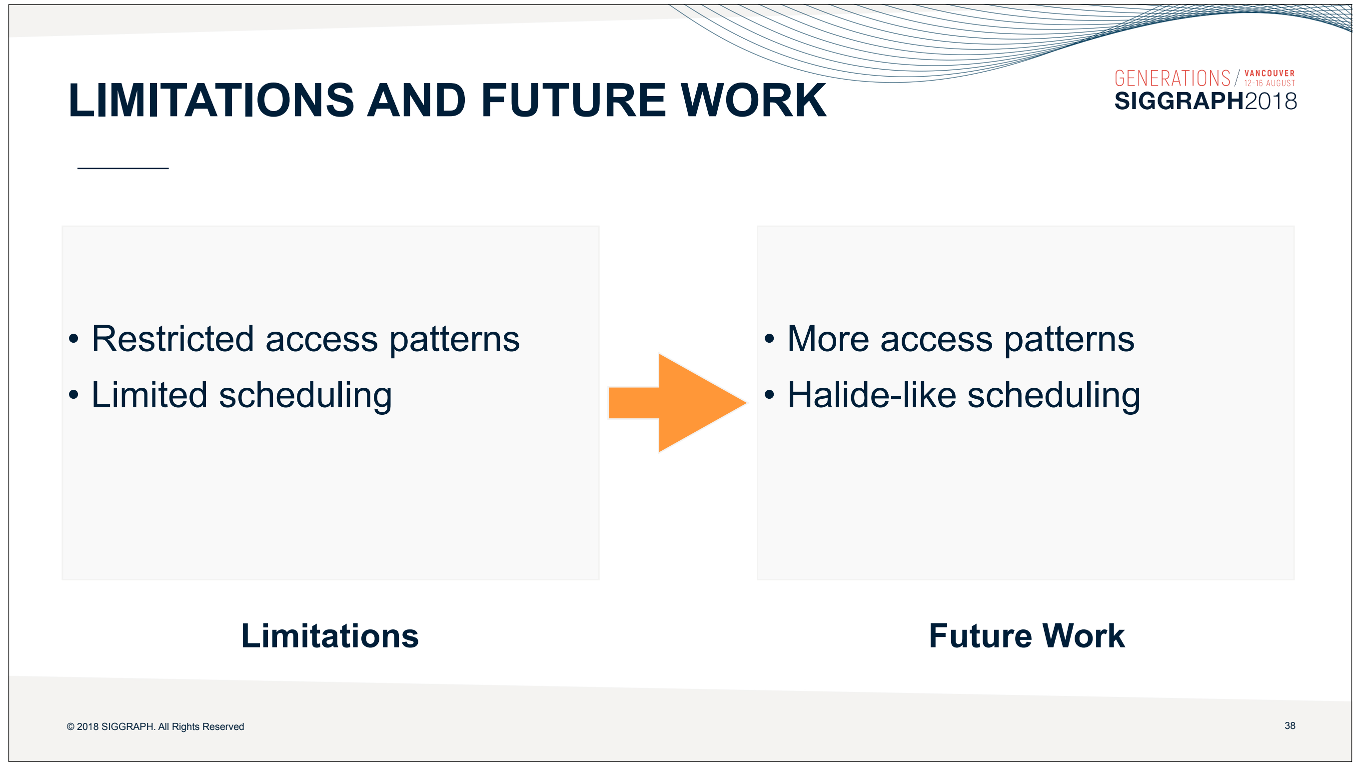
Our design decisions provide us with some natural limitations

<> which suggests future work.

<> Our access patterns are restricted, which is the very thing ensuring we can do the data dependency analysis we need to to maximize parallelism. Future work would relax these restrictions

<> We allow for selectively precomputing parts of the Jacobian computation, which allows users to eke out extra performance, but it should be possible to generalize to a much wider range of schedules, like Halide.

<> We have fast matrix-free code generation for a small set of matrix calculus primitives, those necessary to generate Gauss-Newton-like solvers. Having a language that allows arbitrary tensor derivatives would open up a large set of new potential use-cases that are currently too laborious to explore.



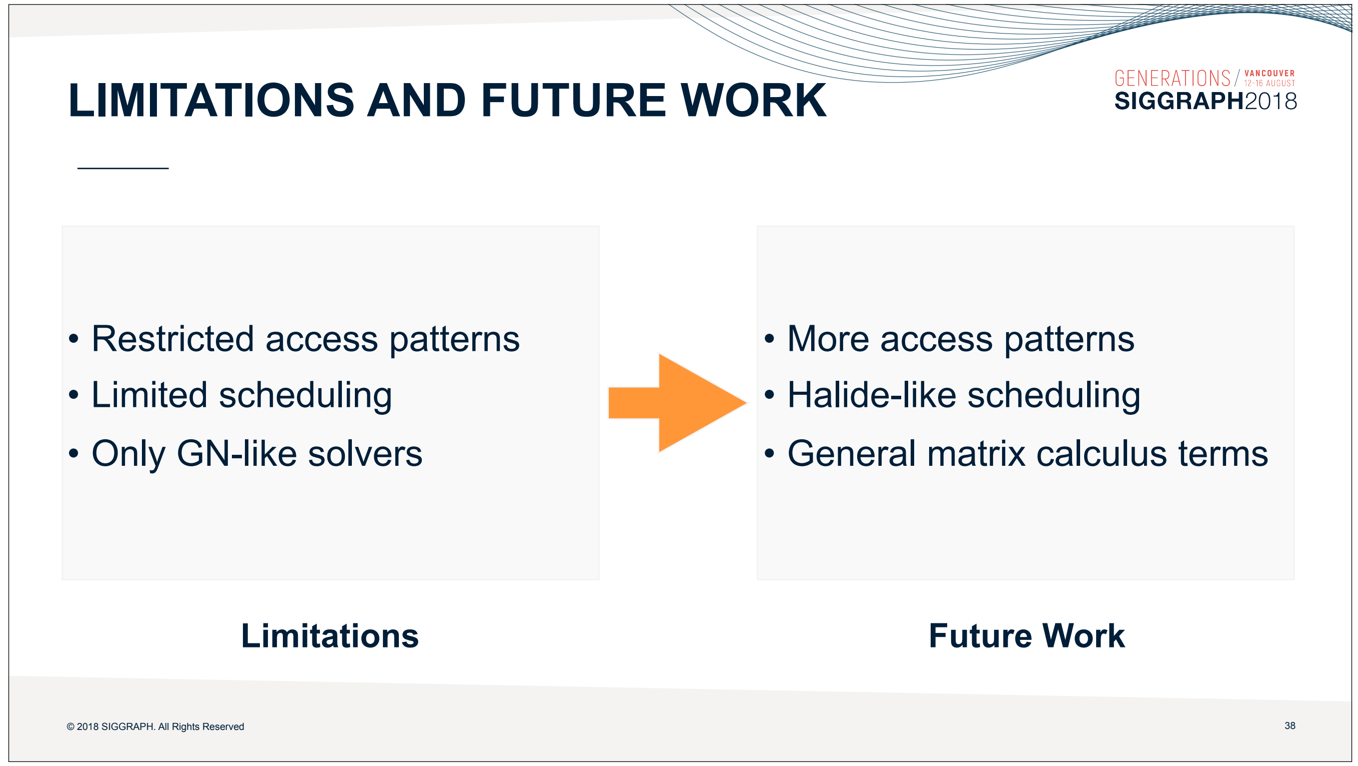
Our design decisions provide us with some natural limitations

<> which suggests future work.

<> Our access patterns are restricted, which is the very thing ensuring we can do the data dependency analysis we need to to maximize parallelism. Future work would relax these restrictions

<> We allow for selectively precomputing parts of the Jacobian computation, which allows users to eke out extra performance, but it should be possible to generalize to a much wider range of schedules, like Halide.

<> We have fast matrix-free code generation for a small set of matrix calculus primitives, those necessary to generate Gauss-Newton-like solvers. Having a language that allows arbitrary tensor derivatives would open up a large set of new potential use-cases that are currently too laborious to explore.



Our design decisions provide us with some natural limitations

<> which suggests future work.

<> Our access patterns are restricted, which is the very thing ensuring we can do the data dependency analysis we need to to maximize parallelism. Future work would relax these restrictions

<> We allow for selectively precomputing parts of the Jacobian computation, which allows users to eke out extra performance, but it should be possible to generalize to a much wider range of schedules, like Halide.

<> We have fast matrix-free code generation for a small set of matrix calculus primitives, those necessary to generate Gauss-Newton-like solvers. Having a language that allows arbitrary tensor derivatives would open up a large set of new potential use-cases that are currently too laborious to explore.

WRAP-UP

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

To wrap things up,
<> we solved the problem we set out to: Very fast non-linear least squares optimizers on images/meshes/graphs are easy to write
<> We have several actually working implementations of real problems including recent SIGGRAPH papers
<> And hundreds of people have used it in one form or another

<> Opt has its own website, and an active community on github, where its released under the open-source MIT license.

Thank you for your attention.

WRAP-UP

• Very fast NLLS optimizers on images/meshes/graphs are easy to write

To wrap things up,
<> we solved the problem we set out to: Very fast non-linear least squares optimizers on images/meshes/graphs are easy to write
<> We have several actually working implementations of real problems including recent SIGGRAPH papers
<> And hundreds of people have used it in one form or another

<> Opt has its own website, and an active community on github, where its released under the open-source MIT license.

Thank you for your attention.

WRAP-UP

- Very fast NLLS optimizers on images/meshes/graphs are easy to write
- 10+ working examples

To wrap things up,
<> we solved the problem we set out to: Very fast non-linear least squares optimizers on images/meshes/graphs are easy to write
<> We have several actually working implementations of real problems including recent SIGGRAPH papers
<> And hundreds of people have used it in one form or another

<> Opt has its own website, and an active community on github, where its released under the open-source MIT license.

Thank you for your attention.

WRAP-UP

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Very fast NLLS optimizers on images/meshes/graphs are easy to write
- 10+ working examples
- Many people use it

To wrap things up,
<> we solved the problem we set out to: Very fast non-linear least squares optimizers on images/meshes/graphs are easy to write
<> We have several actually working implementations of real problems including recent SIGGRAPH papers
<> And hundreds of people have used it in one form or another

<> Opt has its own website, and an active community on github, where its released under the open-source MIT license.

Thank you for your attention.

WRAP-UP

GENERATIONS / VANCOUVER 12-16 AUGUST
SIGGRAPH2018

- Very fast NLLS optimizers on images/meshes/graphs are easy to write
- 10+ working examples
- Many people use it

optlang.org/

Open source under the MIT license

To wrap things up,
<> we solved the problem we set out to: Very fast non-linear least squares optimizers on images/meshes/graphs are easy to write
<> We have several actually working implementations of real problems including recent SIGGRAPH papers
<> And hundreds of people have used it in one form or another

<> Opt has its own website, and an active community on github, where its released under the open-source MIT license.

Thank you for your attention.

THANK YOU!

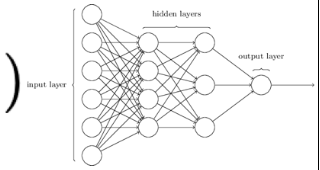
optlang.org/

Open source under the MIT license

EXTRA SLIDES

Why Gauss-Newton?

Gradient Descent? $$\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n)$$



Newton's Method $$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Single variable…

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H}f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n)$$

$$[\mathbf{H}f(\mathbf{x}_n)] \boldsymbol{\Delta} \mathbf{x} = -\nabla f(\mathbf{x}_n)$$

Why are we doing Gauss-Newton? (and what is it anyway?)

Why don't we use gradient descent for optimization like all the cool kids? It's simple to understand, and obviously works.

Well, it actually has bad convergence on interesting functions. There is a reason even beyond sheer data size why neural nets take forever to train.

But simplicity is good! The problem with gradient descent is we aren't using much information, we are linearly approximating our function at each step to find our next move. If we quadratically approximate instead, we get Newton's method for optimization, which is taught as a root finding method in high schools all over. That seems promisingly simple, while having better convergence properties. Of course what I put up there has a problem, its one dimensional.

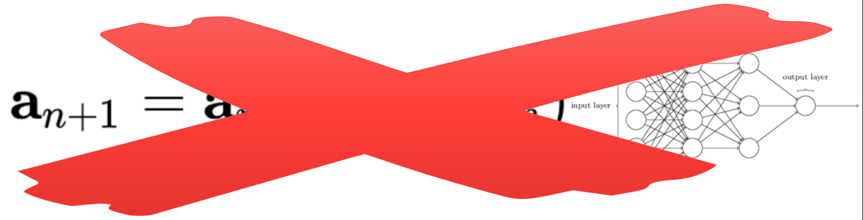
We can move to higher dimensions through analogy. The derivative becomes the gradient vector, and the second derivative is the Hessian matrix.

Problem, we can't divide by a matrix, we must multiply by its inverse instead. But inverting a matrix can be expensive!

So we can move it to the other side of the equation and solve this linear system instead. However, the Hessian itself can be quite expensive to compute!

Why Gauss-Newton?

Gradient Descent? $\mathbf{a}_{n+1} = \mathbf{a}$



Newton's Method $$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Single variable…

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H}f(\mathbf{x}_n)]^{-1}\nabla f(\mathbf{x}_n)$$

$$[\mathbf{H}f(\mathbf{x}_n)]\boldsymbol{\Delta}\mathbf{x} = -\nabla f(\mathbf{x}_n)$$

Why are we doing Gauss-Newton? (and what is it anyway?)

Why don't we use gradient descent for optimization like all the cool kids? It's simple to understand, and obviously works.

Well, it actually has bad convergence on interesting functions. There is a reason even beyond sheer data size why neural nets take forever to train.

But simplicity is good! The problem with gradient descent is we aren't using much information, we are linearly approximating our function at each step to find our next move. If we quadratically approximate instead, we get Newton's method for optimization, which is taught as a root finding method in high schools all over. That seems promisingly simple, while having better convergence properties. Of course what I put up there has a problem, its one dimensional.

We can move to higher dimensions through analogy. The derivative becomes the gradient vector, and the second derivative is the Hessian matrix.

Problem, we can't divide by a matrix, we must multiply by its inverse instead. But inverting a matrix can be expensive!

So we can move it to the other side of the equation and solve this linear system instead. However, the Hessian itself can be quite expensive to compute!

Gauss Newton as Approximate Newton

$$[\mathbf{H}f(\mathbf{x}_n)]\mathbf{\Delta x} = -\nabla f(\mathbf{x}_n)$$

$$f(\mathbf{x}) = \sum_{i=1}^{m} r_i^2(\mathbf{x})$$

$$H_{jk} = 2\sum_{i=1}^{m}\left(\frac{\partial r_i}{\partial x_j}\frac{\partial r_i}{\partial x_k} + r_i\frac{\partial^2 r_i}{\partial x_j \partial x_k}\right)$$

$$H_{jk} \approx 2\sum_{i=1}^{m}\frac{\partial r_i}{\partial x_j}\frac{\partial r_i}{\partial x_k}$$

$$\mathbf{J}^T\mathbf{J}\mathbf{\Delta x} = -\mathbf{J}^T\mathbf{f}$$

$$H_{jk} \approx 2\sum_{i=1}^{m} J_{ij}J_{ik}$$

We have Newton's method.

In the special case of the function f being a sum of squared residual terms, we can exploit the extra structure. If we write out an element of the Hessian matrix we get this sum of terms dependent on a single residual (and its derivatives).

The trick is if we drop the higher order terms on the right, we are simply using the Jacobian transpose Jacobian, which is often a much easier to compute value.

One Step of Gauss-Newton

$$\mathbf{J}^T\mathbf{J}\boldsymbol{\Delta}\mathbf{x} = -\mathbf{J}^T\mathbf{f}$$

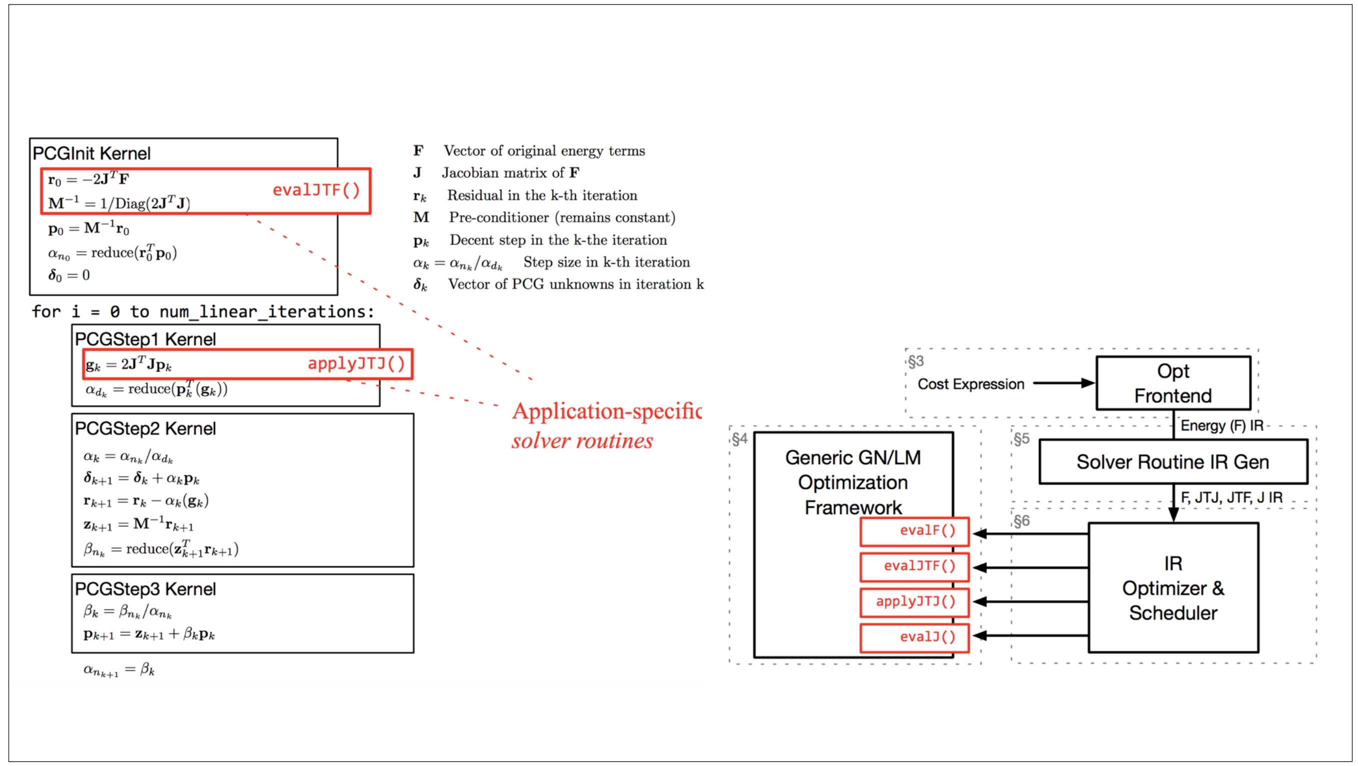
Δx = Step to apply to unknowns
r = vector of residuals
J = Jacobian of **r** with respect to unknowns

Generic Gauss Newton

while (nonlinear convergence criteria false):

  Solve $\mathbf{J}^T\mathbf{J}\ \boldsymbol{\Delta}\mathbf{x} = -\mathbf{J}^T\mathbf{r}$ for $\boldsymbol{\Delta}\mathbf{x}$

  $\mathbf{x} = \mathbf{x} + \boldsymbol{\Delta}\mathbf{x}$

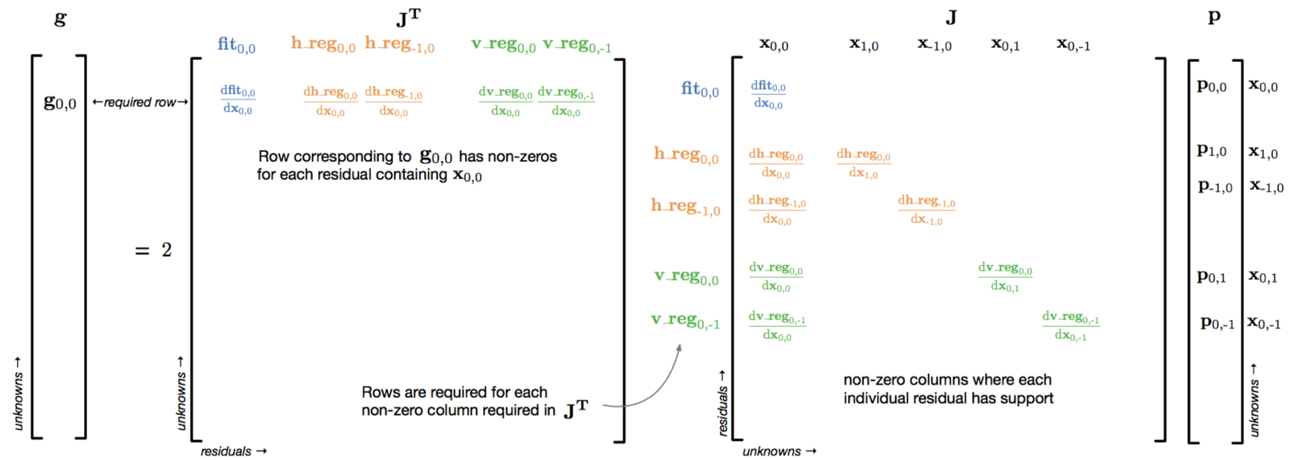

PCGInit Kernel
$\mathbf{r}_0 = -2\mathbf{J}^T\mathbf{F}$
$\mathbf{M}^{-1} = 1/\mathrm{Diag}(2\mathbf{J}^T\mathbf{J})$
evalJTF()
$\mathbf{p}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
$\alpha_{n_0} = \mathrm{reduce}(\mathbf{r}_0^T\mathbf{p}_0)$
$\boldsymbol{\delta}_0 = 0$
for i = 0 to num_linear_iterations:
PCGStep1 Kernel
$\mathbf{g}_k = 2\mathbf{J}^T\mathbf{J}\mathbf{p}_k$
applyJTJ()
$\alpha_{d_k} = \mathrm{reduce}(\mathbf{p}_k^T(\mathbf{g}_k))$
PCGStep2 Kernel
$\alpha_k = \alpha_{n_k}/\alpha_{d_k}$
$\boldsymbol{\delta}_{k+1} = \boldsymbol{\delta}_k + \alpha_k\mathbf{p}_k$
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k(\mathbf{g}_k)$
$\mathbf{z}_{k+1} = \mathbf{M}^{-1}\mathbf{r}_{k+1}$
$\beta_{n_k} = \mathrm{reduce}(\mathbf{z}_{k+1}^T\mathbf{r}_{k+1})$
PCGStep3 Kernel
$\beta_k = \beta_{n_k}/\alpha_{n_k}$
$\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k\mathbf{p}_k$
$\alpha_{n_{k+1}} = \beta_k$
$\mathbf{F}$ Vector of original energy terms
$\mathbf{J}$ Jacobian matrix of $\mathbf{F}$
$\mathbf{r}_k$ Residual in the k-th iteration
$\mathbf{M}$ Pre-conditioner (remains constant)
$\mathbf{p}_k$ Decent step in the k-the iteration
$\alpha_k = \alpha_{n_k}/\alpha_{d_k}$ Step size in k-th iteration
$\boldsymbol{\delta}_k$ Vector of PCG unknowns in iteration k
Application-specific
solver routines
§3
Cost Expression
Opt
Frontend
Energy (F) IR
§5
Solver Routine IR Gen
F, JTJ, JTF, J IR
§4
Generic GN/LM
Optimization
Framework
evalF()
evalJTF()
applyJTJ()
evalJ()
§6
IR
Optimizer &
Scheduler

Laplacian Smoothing bandwidth SoL

GeForce 980 ~224GB/s of bandwidth, 1MP image

- Full matrix: ~20.1s touch every element
- CRS matrix: ~0.17ms touch every element
- Matrix-free: ~0.017ms touch every pixel

(d) Representation of non-zero entries in the expression $\mathbf{g} = 2\mathbf{J}^{\mathbf{T}}\mathbf{J}\mathbf{p}$ that are required to calculate $\mathbf{g}_{0,0}$



Miscellaneous

- Levenberg-Marquardt
- Cusparse backend
- Float/Double precision
- IRLS – Can solve for L_1 energies
- OpenGL-like C-API