Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging

ZACHARY DEVITO and MICHAEL MARA
Stanford University
MICHAEL ZOLLHÖFER
Max-Planck-Institute for Informatics
GILBERT BERNSTEIN and JONATHAN RAGAN-KELLEY
Stanford University
CHRISTIAN THEOBALT
Max-Planck-Institute for Informatics
PAT HANRAHAN and MATTHEW FISHER
Stanford University
and
MATTHIAS NIESSNER
Stanford University

Many graphics and vision problems are naturally expressed as optimizations with either linear or non-linear least squares objective functions over visual data, such as images and meshes. The mathematical descriptions of these functions are extremely concise, but their implementation in real code is tedious, especially when optimized for real-time performance in interactive applications.

We propose a new language, Opt[1], in which a user simply writes energy functions over image- or graph-structured unknowns, and a compiler automatically generates state-of-the-art GPU optimization kernels. The end result is a system in which real-world energy functions in graphics and vision applications are expressible in tens of lines of code. They compile directly into highly-optimized GPU solver implementations with performance competitive with the best published hand-tuned, application-specific GPU solvers, and 1–2 orders of magnitude beyond a general-purpose auto-generated solver.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Digitizing and Scanning*

Additional Key Words and Phrases: Non-linear least squares, Domain-specific Languages, Levenberg-Marquardt, Gauss-Newton

1. INTRODUCTION

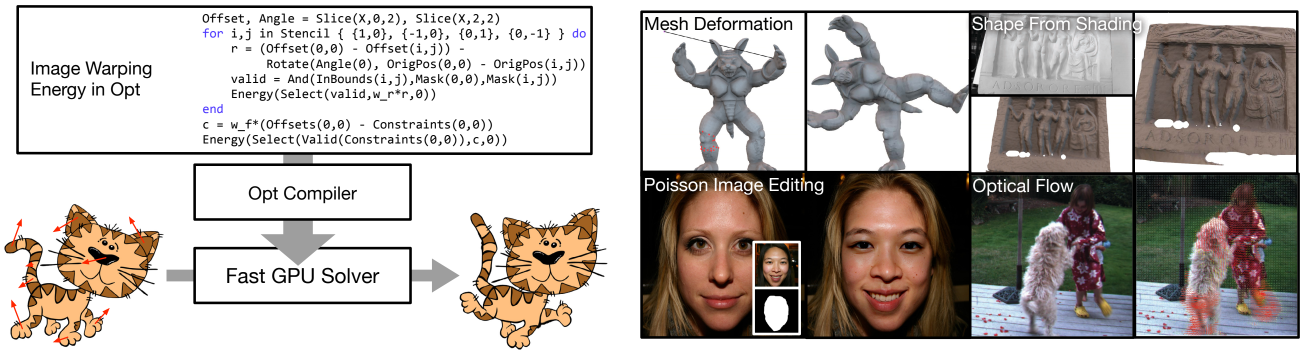
Many problems in graphics and vision can be concisely formulated as least squares optimizations on an image, mesh, or graph-structured domains. For example, Poisson image editing, shape-from-shading, and as-rigid-as-possible warping have all been phrased as least squares optimizations, allowing them to be described tersely as energy functions over pixels or meshes [Pérez et al. 2003; Wu et al. 2014; Sorkine and Alexa 2007]. In many of these applications, high performance is critical for interactive feedback, requiring efficient parallel or GPU-based solvers. However, making efficient parallel solvers in general is an open problem. Solving the optimization using primitives from a generic linear algebra framework is inefficient because the explicitly-represented sparse matrices in these libraries are often several times larger than the problem data needed to construct them.

Recent work has achieved real-time performance for non-linear least squares graphics problems by working directly on the problem data using a variant of Gauss-Newton optimization with a preconditioned conjugate gradient inner loop run on the GPU [Wu et al. 2014; Zollhöfer et al. 2014]. A similar approach also supports the Levenberg-Marquardt algorithm. Key to the performance of these methods are two ideas: operating in-place on problem data to avoid ever forming the full matrices needed by the solve, and implicitly representing the connectivity of the sparse matrices using the structure of the problem domain to increase locality. However, this comes at enormous implementation cost: the terse and simple energy function must be manually transformed into a complex product of partial derivatives and preconditioner terms ($\mathbf{J}^T\mathbf{F}$ and $\mathbf{J}^T\mathbf{J}\mathbf{p}$ at each point in the image or graph). The in-place formulation tightly intertwines this application-specific logic derived from the energy with the complex details of the solver. This approach delivers excellent performance, but requires hundreds of lines of highly-tuned CUDA code that no longer resembles the energy function. The result is error-prone and difficult to change, since the derived forms of the energy terms are complex and subtle. It often requires months of engineering effort, with simultaneous expertise in the application domain, optimization methods, and high-performance GPU programming.

We want to make this type of high performance optimization accessible to a much wider community of graphics and vision programmers. We have created a new language, Opt, which lets a programmer easily write sum-of-squares energy functions over pixels or graphs, such as the one show in Fig. 1 for as-rigid-as-possible image warping. A compiler takes these energies and automatically generates highly optimized in-place solvers using the parallel Gauss-Newton or Levenberg-Marquardt method with a preconditioned conjugate gradient inner loop.

Our system is able to do this due to four key ideas. First, we provide a generalization of the parallel Gauss-Newton method pop-

[1] Opt is publicly available under `http://optlang.org`.



```
Offset, Angle = Slice(X,0,2), Slice(X,2,2)
for i,j in Stencil { {1,0}, {-1,0}, {0,1}, {0,-1} } do
    r = (Offset(0,0) - Offset(i,j)) -
         Rotate(Angle(0), OrigPos(0,0) - OrigPos(i,j))
    valid = And(InBounds(i,j),Mask(0,0),Mask(i,j))
    Energy(Select(valid,w_r*r,0))
end
c = w_f*(Offsets(0,0) - Constraints(0,0))
Energy(Select(Valid(Constraints(0,0)),c,0))
```

Fig. 1. From a high-level description of an energy, Opt produces high-performance optimizers for many graphics problems.

ular in recent work, abstracted to work with arbitrary least squares energies over images and graphs, and extend it to the more general Levenberg-Marquardt method. Second, our language provides key abstractions for representing energies at a high-level. Unknowns and other data are arranged on 2-dimensional pixel grids, meshes, or general graphs. Energies are defined over these domains and access data through *stencil* patterns (fixed-size and shift-invariant local windows). Third, our compiler exploits the regularity of stencils and graphs to automatically generate efficient in-place solver routines. Derivative terms required by these routines are created using hybrid symbolic-automatic differentiation based on a simplified version of the D$\star$ algorithm [Guenter 2007]. Finally, we use a specialized code generator to emit efficient GPU code for the derivative terms and use metaprogramming to combine the skeleton of our solver methods with the generated terms without incurring any runtime overhead.

Our method provides both far better performance and simpler problem specification than traditional general-purpose solver libraries: performance is better than state-of-the-art, application-specific, in-place GPU solvers, and it requires the programmer to provide only the energy to be minimized, not the fully-formed derivative matrices of the system to be solved. In particular, we present the following contributions:

—We propose a high-level programming model for defining energies over image and graph domains.

—We introduce a generic framework for solving non-linear least square problems on GPUs based on the efficient in-place methods used in state-of-the-art application-specific solvers.

—We provide algorithms based on symbolic differentiation that exploit the regularity of energies defined on images and graphs to produce efficient in-place solver routines for our framework. Our optimizations produce code competitive with hand-written routines.

—We implement a variety of graphics problems, including mesh/image deformations, smoothing, and shape-from-shading refinement using Opt. Our implementations outperform state-of-the-art application-specific solvers and are up to two orders of magnitude faster than the CPU-based Ceres solver [Agarwal et al. 2010].

2. BACKGROUND

Non-linear Least Squares Optimization. A variety of optimization methods are used in the graphics community to solve a wide range of problems. Our approach focuses specifically on *unconstrained non-linear least squares* optimizations, where a solver minimizes an energy function that is expressed as a sum of squared *residual* terms: $E(\mathbf{x}) = \sum_{r=1}^{R} \left[f_r(\mathbf{x})\right]^2$. The residuals $f_r(\mathbf{x})$ are generic functions, making the problems potentially non-linear and non-convex [Boyd and Vandenberghe 2004].

As a backend for optimization, we use the Gauss-Newton (**GN**) and Levenberg-Marquardt (**LM**) methods. GN and LM are specifically tailored towards these kind of problems. Their second-order optimization approach has been shown well-suited for the solution of a large variety of problems, and has also been successfully applied in the context of real-time optimization [Zollhöfer et al. 2014; Wu et al. 2014]. If the non-linear energy is convex, then Gauss-Newton will converge to the global minimum; otherwise it will converge to some local minimum (the same applies to LM). Furthermore, GN and LM internally solve a linear system. While these systems can generally be solved with direct methods, our solvers need to scale to large sizes and run on massively parallel GPUs; hence, we implement GN/LM with a preconditioned conjugate gradient (PCG) [Nocedal and Wright 2006] in the inner loop.

In the current implementation, we focus on GN and LM rather than other variants such as L-BFGS [Nocedal and Wright 2006], since they reflect the approaches used in state-of-the-art hand-written GPU implementations, allowing us to compare our performance to existing solvers directly. However, we believe our programming model and program transformations can also be extended to various other solver backends.

Application-specific GPU Solvers. Application-specific Gauss-Newton solvers written for GPUs have been frequently used in the last two years. Wu et al. [2014] use a blocked version of GN to refine depth from RGB-D data using shape-from-shading. Zollhoefer et al. [2014] minimize an as-rigid-as-possible energy [Sorkine and Alexa 2007] on a mesh as part of a framework for real-time non-rigid reconstruction. Zollhöfer et al. [2015] use a similar solver to enforce shading constraints on a volumetric signed-distance field in order to refine over-smoothed geometry with RGB data. Thies et al. [2015; 2016] transfer local facial expressions between people in a video by optimizing photo-consistency between the video and synthesized output. Dai et al. [2016] solve a global bundling adjustment problem to achieve real-time rates for globally-consistent 3D reconstruction.

These solvers achieve high-performance by working *in-place* on the problem domain. That is, during the PCG step, they never form the entire Jacobian $\mathbf{J}$ of the energy. Instead, they compute it on demand, for instance by reading neighboring pixels to compute the derivative of a regularization energy. Performance improves in two ways: first, they do not explicitly store and load sparse matrix connectivity; rather, this is implied by pixel relationships or meshes. Second, reconstructing terms is often faster than storing them, since the size of the problem data is *smaller* than the full matrix implied by the energy.

However, these application-specific solvers are tedious to write because they mix code that calculates complicated matrix products with partial derivatives based on the energy.

High-level Solvers. Higher-level solvers such as CVX [Grant and Boyd 2014; 2008], Ceres [Agarwal et al. 2010], and OpenOF [Wefelscheid and Hellwich 2013] work directly from an energy specified in a domain-specific language. CVX uses disciplined programming to ensure that modeled energy functions are convex, then constructs a specialized solver for the given type of convex problem. Ceres uses template meta-programming and operator overloading to solve non-linear least squares problems on the CPU using backwards auto-differentiation. Unlike Opt, these two solvers do not generate efficient GPU implementations and often explicitly form sparse matrices. OpenOF does run on GPUs but it also explicitly forms sparse matrices [Wefelscheid and Hellwich 2013]. In contrast, Opt's approach of working in-place can be significantly faster than explicit matrices (Sec. 7.2). CPU libraries such as Alglib [Bochkanov 1999] and g2o [Kummerle et al. 2011] abstract the solver, requiring users to provide numeric routines for energy evaluation and, optionally, gradient calculation. While they can in principle work in-place, they cannot optimize the compilation of energy terms and solver code, unlike application-specific solvers, and require hand-written gradients to run fast. Similar to high-level solvers, Opt only requires a description of the energy, but it uses code transformations to generate an application-specific in-place GPU solver automatically.

Differentiation Methods. In-place solvers need to efficiently compute derivatives of the energy, since they are required in each iteration of the solver loop. *Numeric differentiation*, which uses finite differences to estimate derivatives, is numerically unreliable and inefficient [Guenter 2007]. Instead, packages like Mathematica [Wolfram Research 2000] allow users to compute *symbolic* derivatives using rewrite rules. Because they frequently represent math as trees, they do not handle common sub-expressions well, making them impractical for large expressions [Guenter 2007]. *Automatic-differentiation* is transformation on *programs* rather than symbols [Griewank and Walther 2008; Grund 1982]. They replace numbers in a program with "dual"-numbers that track a specific partial derivative using the chain rule. However, because the transform does not work on symbols, simplifications that result from the chain rule are not always applied. We use a hybrid *symbolic-automatic* approach similar to D⋆, which represents math symbolically but stores it as a directed acyclic graph (DAG) of operators to ensure that it can scale to large problems [Guenter 2007]. A symbolic representation of derivatives is important for Opt since solver routines use many derivative terms which share common expressions that would not be addressed by auto-differentiation methods.

3. PROGRAMMING MODEL

An overview of Opt's architecture is given in Fig. 2. In this section, we describe our programming model to construct expressions for the energy of problems. Sec. 4 describes our generic solver architecture on GPUs, which contains the building blocks for Gauss-Newton and Levenberg-Marquardt. To work in-place, it requires application-specific *solver routines* (`evalF()`,`evalJTF()`, `applyJTJ()`). Sec. 5 describes how we generate these routines from the energy.

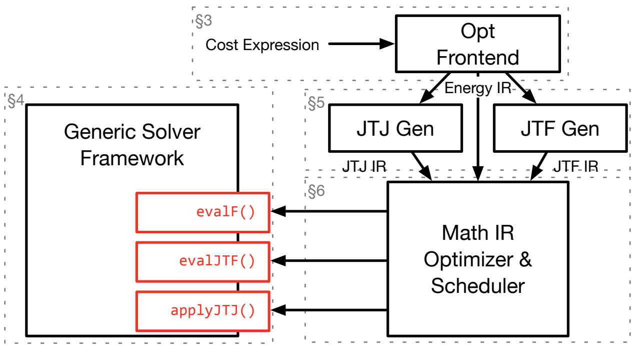


Fig. 2. An overview of the architecture of Opt, labeled with the sections where each part is described.

```
W,H = Dim("W",0), Dim("H",1)
X = Array2D("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)))
```

Fig. 3. The Laplacian smoothing energy in Opt.

We introduce our programming model using the example of Laplacian smoothing of an image. A fitting term encourages a pixel X to be close to its original value A:

$$E_{fit}(i,j) = [X(i,j) - A(i,j)]^2$$

A regularization term encourages neighboring pixels to be similar:

$$E_{reg}(i,j) = \sum_{(l,m)\in\mathcal{N}(i,j)} [X(i,j) - X(l,m)]^2$$

$$\text{where } \mathcal{N}(i,j) = \{(i+1,j),(i,j+1)\}$$

The energy is a weighted sum of both terms:

$$E_{\Delta} = \sum_{(i,j)\in\mathcal{I}} w_{fit}E_{fit}(i,j) + w_{reg}E_{reg}(i,j)$$

While this example is linear, Opt supports arbitrary non-linear energy expressions.

Language. Similar to shading languages such as OpenGL, Opt programs are composed of a "shader" file that describes the energy, and a set of C APIs for running the problem. Fig. 3 expresses the Laplacian energy in Opt. Opt is embedded in the Lua programming language and operator overloading is used to create a symbolic representation of the energy. The first line specifies problem dimensions. Lines 2–3 use the function `Array2D` to declare two pixel arrays that represent the unknown X and the starting image A. By convention X is always the array of unknown variables being optimized. `Array2D`'s last argument is an index that binds the array to data provided by the API.

`Energy` adds residual expressions to the problem's energy. A key part of Opt's abstraction is that residuals are described at elements of images or graphs and implicitly mapped over the entire domain. The term `w_fig*(X(0,0) - A(0,0))` defines an energy at each pixel that is the difference between the images. We support arrays and

```
void SolveLaplacian(int width, int height,
                    float * unknown, float * target) {
    OptState *  state = Opt_NewState();
    // load the Opt DSL file containing the cost description
    OptProblem * problem = Opt_ProblemDefine(m_optimizerState,
                                             "laplacian.opt");
    // describe the dimensions of the instance of the problem
    uint32_t dims[] = { width, height };
    uint32_t strides[] = { width * sizeof(float),
                           width * sizeof(float) };
    uint32_t elemsizes[] = { sizeof(float), sizeof(float) };
    OptPlan * m_plan = Opt_ProblemPlan(state, problem, dims,
                                       elemsizes, strides);
    // run the solver
    void * array_data[] = { unknown_pixel_data, target_pixel_data };
    Opt_ProblemSolve(state, plan, array_data,
                     NULL, NULL, NULL, NULL, NULL, NULL);
}
```

Fig. 4. Opt API calls that use the Laplacian smoothing program.

```
N = Dim("N",0)
X = Array1D("X", opt.float3,N,0)
A = Array1D("A", opt.float3,N,1)
G = Graph("Edges", 0,
          "vertex0", N, 0,
          "vertex1", N, 1)

w_fit,w_reg = .1,.9
Energy(w_fit*(X(0,0) - A(0,0)),
       w_reg*(X(G.vertex0) - X(G.vertex1)))
```

Fig. 5. The Laplacian cost defined on the edges of a mesh instead of an image.

energies that include both vector and scalar terms. The `Energy` function implicitly squares the terms and sums them over the domain to enforce the linear least-squares model. Terms can also include a statically-defined *stencil* of neighboring pixels. The regularization term `w_reg*(X(0,0) - X(1,0))` defines an energy that is the difference between a pixel and the pixel to its right. Our solver framework exploits this regularity to produce efficient code.

API. Applications interact with Opt programs using a C API. Fig. 4 shows an example using this API. To amortize the cost of preparing a problem used multiple times, we separate the compilation, memory allocation, and execution of a problem into different API calls.

Mesh-based problems. Opt also includes primitives for defining energies on graphs to support meshes or other irregular structures. Fig. 5 shows an example that smooths a mesh rather than an image. The `Graph` function defines a set of hyper-edges that connect entries in the unknown together. In this example, each edge connects two entries `vertex0` and `vertex1`, but in general our edges allow an arbitrary number of entries to represent elements like triangles. Energies can be defined on these elements, as seen in the regularization term (line 10), which defines an energy on the edge between two vertices.

Boundaries. Defining energies on arrays of pixels requires handling boundaries. By default out-of-bounds values are clamped to zero, but we also provide the ability to query whether a pixel is valid (`InBounds`) and select a different expression if it is not (`Select`):

```
term = w_reg*(X(0,0) - X(1,0))
Energy(Select(InBounds(1,0),term,0))
```

Boundary expressions are optimized later in the compilation process to ensure they do not cause excessive overhead.

Pre-computing shared expressions. Energy functions for neighboring pixels can share expensive-to-compute expressions. For instance, our shape-from-shading example (Sec. 7) uses an expensive lighting calculation that is shared by neighboring pixels. We allow the user to turn these calculations into *computed arrays*, which behave like arrays when used in energy functions, but are defined as an expression of other arrays:

```
computed_lighting = ComputedArray(W,H,lighting_calculation(0,0))
```

Computed arrays can include computations using the unknown X, and are recalculated as necessary during the optimization. Similar to scheduling annotations in Halide [Ragan-Kelley et al. 2012], they allow the user to balance recompute with locality at a high-level.

4. NON-LINEAR LEAST SQUARES OPTIMIZATION FRAMEWORK

Our optimization framework is a generalization of the design of application-specific GPU solvers [Zollhöfer et al. 2014; Wu et al. 2014; Zollhöfer et al. 2015; Thies et al. 2015; Thies et al. 2016; Innmann et al. 2016]. While our solver API is general and abstracts away a specific optimization algorithm, our framework currently provides implementations for Gauss-Newton and Levenberg-Marquardt. In the context of non-linear least square problems, we consider the optimization objective $E : \mathbb{R}^N \to \mathbb{R}$, which is a sum of squares in the following canonical form:

$$E(\mathbf{x}) = \sum_{r=1}^{R} \left[f_r(\mathbf{x})\right]^2$$

The R scalar residuals f_r can be general linear or non-linear functions of the N unknowns $\mathbf{x}$. The objective takes the traditional form used in the Gauss-Newton method:

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

The R-dimensional vector field $F : \mathbb{R}^N \to \mathbb{R}^R$ stacks all scalar *residuals* f_r. The minimizer $\mathbf{x}^*$ of E is given as the solution of the following optimization problem:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\, E(\mathbf{x}) = \underset{\mathbf{x}}{\operatorname{argmin}}\, \left|\left|\mathbf{F}(\mathbf{x})\right|\right|_2^2$$

It is solved based on a fixed-point iteration that incrementally computes a sequence of better solutions $\{\mathbf{x}_k\}_{k=1}^{K}$ given an initial estimate $\mathbf{x}_0$. Here, K is the number of iterations; i.e., $\mathbf{x}^* \approx x_K$. In every iteration step, a *linear* least squares problem is solved to find the best linear parameter update. The vector field $\mathbf{F}$ is first linearized using a first-order Taylor expansion around the last solution $\mathbf{x}_k$:

$$\mathbf{F}(\mathbf{x}_k + \delta_k) \approx \mathbf{F}(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)\delta_k$$

Here, $\mathbf{J}$ is the Jacobian matrix and contains the first-order partial derivatives of $\mathbf{F}$. By applying this approximation, the original non-linear least squares problem is reduced to a quadratic problem:

$$\delta_k^* = \underset{\delta_k}{\operatorname{argmin}}\, \left|\left|\mathbf{F}(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)\delta_k\right|\right|_2^2$$

After the optimal update δ_k^* has been computed, a new solution $\mathbf{x}_{k+1} = \mathbf{x}_k + \delta_k$ can be easily obtained. Since this problem is highly over-constrained and quadratic, the least squares minimizer is the solution of a linear system of equations. This system is obtained by setting the partial derivatives to zero, which results in the well known *normal equations*:

$$2 \cdot \mathbf{J}(\mathbf{x}_k)^T \mathbf{J}(\mathbf{x}_k)\delta_k^* = -2 \cdot \mathbf{J}(\mathbf{x}_k)^T \mathbf{F}(\mathbf{x}_k)$$

This process is iterated for K steps to obtain an approximation to the optimal solution $\mathbf{x}^* \approx \mathbf{x}_K$.

The GN approach can be interpreted as a variant of Newton's method that only requires first-order derivatives and is computationally less heavy. To this end, it uses a first-order Taylor approximation $2(\mathbf{J}^T\mathbf{J})$ instead of the real second-order Hessian $\mathbf{H}$.

LM additionally introduces a steering parameter λ to switch between LM and Steepest Descent (**SD**). To this end, the normal equations are augmented with an additional diagonal term. This is similar to Tikhonov regularization and leads to:

$$2(\mathbf{J}(\mathbf{x}_k)^T\mathbf{J}(\mathbf{x}_k) + \lambda \operatorname{diag}(\mathbf{J}(\mathbf{x}_k)^T\mathbf{J}(\mathbf{x}_k)))\delta_k^* = -2\mathbf{J}(\mathbf{x}_k)^T\mathbf{F}(\mathbf{x}_k)$$

4.1 Parallelizing the Optimization with PCG

The core of the GN/LM methods is the iterative solution of *linear least squares problems* for the computation of the optimal linear updates δ_k^*. This boils down to the solution of a system of linear equations in each step, i.e., the *normal equations*. While it is possible to use direct solution strategies for linear systems, they are inherently sequential, while our goal is a fast parallel solution on a many-core GPU architecture with conceptually several thousand independent threads of execution. Consequently, we use a parallel preconditioned conjugate gradient (PCG) solver [Weber et al. 2013; Zollhöfer et al. 2014], which is fully parallelizable on modern graphics cards.

The PCG algorithm and our strategy to distribute the computations across GPU kernels is visualized in Fig. 6. We run a *PCGInit* kernel (one time initialization) and three *PCGStep* kernels (inner PCG loop). Before the PCG solve commences, we initialize the unknowns δ_0 to zero. For preconditioning, we employ the Jacobi preconditioner, which scales the residuals with the inverse diagonal of $\mathbf{J}^T\mathbf{J}$. Jacobi preconditioning is especially efficient if the system matrix is diagonally dominant, which is true for many problems. When the matrix is not diagonally dominant, we fall back to a standard conjugate gradient descent. We also use single-precision floating point numbers throughout, which matches the approach of the recent application-specific solvers. We believe this strategy is a good compromise between computational effort and efficiency.

Stencil-based Array Access. Our techniques for parallelizing work are different for array and graph residuals. For arrays, we group the computation required for each element in the *unknown* domain onto one GPU thread. For a matrix product such as $-2\mathbf{J}^T\mathbf{F}$, each row of the output is generated by the thread associated with the unknown. If the unknown is a vector (e.g., RGB pixel), all channels are handled by one thread since these values will frequently share sub-expressions.

The computations in a GPU thread work *in-place*. For instance, if they conceptually require a particular partial derivative from matrix $\mathbf{J}$, they will compute it from the original problem state which includes the unknowns and any supplementary arrays. Matrices such as $\mathbf{J}$, which are conceptually larger than the problem state, are never written to memory, which minimizes memory accesses. Section 5 describes how we automatically generate these computations from our stencil- and graph-based energy specification.

Graph-based Array Access. For graph-based domains, such as 3D meshes, the connectivity is explicitly encoded in a user-provided data structure. Users specify the mapping from graph edges to vertices. Residuals are defined on graph (hyper-) edges and access unknowns on vertices. To make it easy for the user to change the graph over time, we do not require a reverse mapping from unknowns to residuals for graphs. Kernels that use the residuals (`PCGInit` and `PCGStep1`) assign one edge in the graph to one GPU

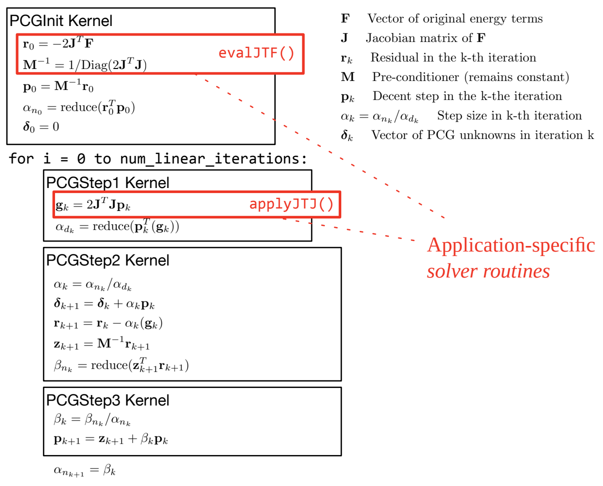

PCGInit Kernel
$\mathbf{r}_0 = -2\mathbf{J}^T\mathbf{F}$
$\mathbf{M}^{-1} = 1/\mathrm{Diag}(2\mathbf{J}^T\mathbf{J})$
evalJTF()
$\mathbf{p}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
$\alpha_{n_0} = \mathrm{reduce}(\mathbf{r}_0^T\mathbf{p}_0)$
$\boldsymbol{\delta}_0 = 0$
for i = 0 to num_linear_iterations:
PCGStep1 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

Fig. 6. Generic GPU architecture for Gauss-Newton and Levenberg-Marquardt solvers whose linearized iteration steps are solved in parallel using the preconditioned conjugate gradient (PCG) method.

thread. Since the output vectors have the same dimension as the unknowns, we have to *scatter* the terms in the residual evaluations into these values. All threads involving partial sums for a given variable then scatter into the corresponding parts of variables using a floating-point atomic addition.

4.2 Modularizing the Solver

A key contribution of our approach is the modularization of the application-specific components of GPU Gauss-Newton or Levenberg-Marquardt solvers into compartmentalized *solver routines*. The first routine, `evalF()`, simply generates the application specific energy for each residual. It only runs outside of the main loop to report progress.

evalJTF. The second routine appears in the `PCGInit` kernel and is shown in red in Fig. 6. Here, the initial descent direction $\mathbf{p}_0$ is computed using the application-specific `evalJTF()` routine, which is generated by our compiler. It computes an in-place version of $-2\mathbf{J}^T\mathbf{F}$. `evalJTF()` is also responsible for computing the preconditioner $\mathbf{M}$, which is simply the dot product of a row of $\mathbf{J}^T$ with itself. For arrays, a thread computes the rows of an output associated with one element of the unknown. For graphs, each thread only computes the parts of the dot product between $\mathbf{J}^T$ and $\mathbf{F}$ which belong to the handled residual.

applyJTJ. The third routine, `applyJTJ()`, is part of the inner PCG iteration. It computes the multiplication of $2\mathbf{J}^T\mathbf{J}$ with the current descent direction $\mathbf{p}_k$, and incorporates the steering factor λ when using Levenberg-Marquardt. Handling arrays and graphs is similar to `evalJTF()`. It tends to use more values since it needs to compute entries from both $\mathbf{J}$ and $\mathbf{J}^T$. For many problems this routine is the most expensive step, so it has to be optimized well.

5. GENERATING SOLVER ROUTINES

A key idea of Opt is that we can exploit the regularity of stencil- and graph-based energies to automatically generate application-specific solver routines. We represent the mathematical form of the energy as a DAG of operators, or our *intermediate representation* (IR). We

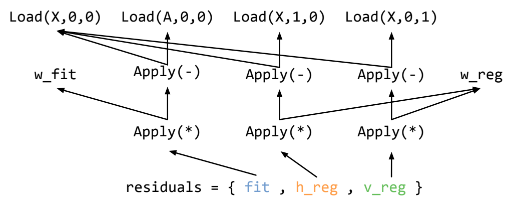


Fig. 7. The Laplacian example represented in our IR.

```
-- generates the derivative of expression with respect to variable
function derivative(expression, variable)
   if a cached version of this partial derivative exists then
     return the cached version
   elseif expression == variable then
     return 1
   end
   result = 0
   for i = 0, the number of arguments used by expression do
     result += derivative(argument[i],variable)*partial[i]
   end
   cache and return result
end
```

Fig. 8. Pseudocode of the OnePass algorithm for generating derivatives. `partial[i]` is the partial derivative of the particular operator (e.g., `*`) with respect to the argument `i`, which is defined for each operator.

transform the IR to create new IR expressions needed for `evalJTF()` and `applyJTJ()`. This process requires partial derivatives of energy. We then optimize this IR and generate code that calculates it.

5.1 Intermediate Representation

Since the Opt language is embedded in Lua, we generate the IR by running the Lua program which uses overloaded operators to build the graph. Fig. 7 shows the IR that results from the Laplacian example. Roots of the IR are residuals that we want to compute. Leaves are constants (e.g., `w_fit`), input data (e.g the known image `A(0,0)`), and the the unknown image (e.g., `X(0,0)`). We de-duplicate the graph as it is built, ensuring common-subexpressions are eliminated. We scalarize vectors from our frontend in the IR to improve the simplification of expressions that become zeros during differentiation.

5.2 Differentiating IR

Since we do not store the Jacobian $\mathbf{J}$ matrix in memory, we need to generate residuals on the fly. The approach we use for differentiation is similar to Guenter's D$\star$ [Guenter 2007]. It symbolically generates new IR that represents a partial derivative of an existing IR node. Unlike traditional symbolic differentiation (e.g., Mathematica), differentiation is done on a graph where terms can share common sub-expressions. In our implementation we use OnePass, a simplification of D$\star$ that can achieve good results by doing the symbolic equivalent of forward auto-differentiation [Guenter et al. 2011]. Pseudocode for the algorithm is given in Fig. 8. It works by memoizing a result for each partial derivative and generates a new derivative of an expression by propagating derivatives from its arguments via the chain rule.

5.3 Generating IR for Matrix Products

The IR for `evalF()` is simply the input energy IR. We generate IR for `evalJTF()` and `applyJTJ()` as transformations of this input IR. These terms are conceptually derived from matrix-matrix or

```
function create_jtj(residual_templates,X,P)
  P_hat = 0
  residuals = residuals_including_x00(residual_templates)
  foreach residual do
    dr_dx00 = differentiate(residual,X(0,0))
    foreach unknown u used by residual do
      dr_du = differentiate(residual,u)
      P_hat += dr_dx00*dr_dx*P(u.offset_i,u.offset_j)
    end
  end
  return 2*P_hat
end
function residuals_including_x00(residual_templates)
  residuals = {}
  foreach residual_template do
    foreach unknown x appearing in residual_template do
      -- shift the template such that x is centered (i.e. it is x00)
      R = shift_exp(residual_template,-x.offset_i,-x.offset_j)
      table.insert(residuals,R)
    end
  end
  return residuals
end
function shift_exp(exp, shift_i, shift_j)
    replace each access of any image at (x,y) in exp
    with an access at (x + shift_i,j + shift_j)
end
```

Fig. 10. Pseudocode that generates JTJ from residual templates.

matrix-vector multiplications of the Jacobian. Since we compute these values in-place, we must generate the IR that will calculate the output given our specific problem. Each term has two versions: one for handling stencil-based and one for graph-based residuals. Here, we describe the approach for the Gauss-Newton solver. LM is a straightforward extension that incorporates the λ parameter for terms on the diagonal in `applyJTJ()`.

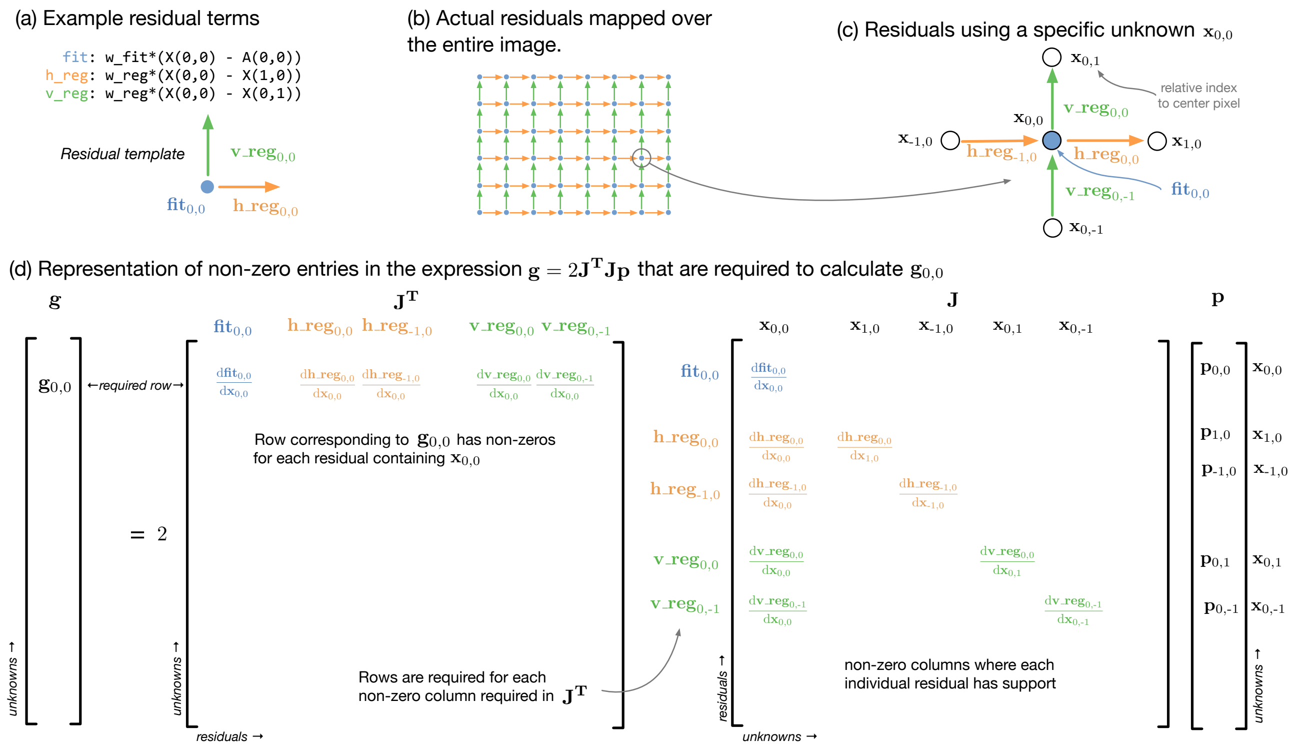
5.3.1 *Stencil Residuals.*

Our solver calls `applyJTJ()` to calculate a single entry of $\mathbf{g}$, where $\mathbf{g} = 2\mathbf{J^TJp}$ per thread. We need to determine which values from $\mathbf{J}$ are required and create IR that calculates them. The non-zero entries in $\mathbf{J}$ are determined by the *stencil* of a particular problem. Fig. 9 illustrates the process of discovering the non-zeros. In the Laplacian case, the partials used in these expressions are actually constants because it is a linear system. However, Opt supports the generic non-linear case, where the partials will be functions of the unknown.

The pseudocode to generate $\mathbf{J}^T\mathbf{J}$ for stencils is shown in Fig. 10. It first finds the residuals that use unknown $\mathbf{x}_{0,0}$ because they correspond to the non-zeros of $\mathbf{J}^T$. Some of these residuals are not actually *defined* at pixel $(0, 0)$, but use $\mathbf{x}_{0,0}$ from neighboring pixels. To find them, we exploit the fact that stencils are *invertible*. For each residual *template* in the energy, we examine each place it uses an unknown $\mathbf{x}_{i,j}$. We then *shift* that residual on the pixel grid, taking each place it loads a stencil value and changing its offset by $(-i, -j)$, which generates a residual in the grid that uses $\mathbf{x}_{0,0}$. We find all the residuals using $\mathbf{x}_{0,0}$ by repeating the process for each use of an unknown in the template. While we only allow constant stencil offsets, in principle this approach will work for any neighborhood function which is invertible.

For each discovered residual, we need other unknowns it uses which are found by examining the IR symbolically. We then generate the expressions for the part of the matrix-vector products that calculate $\mathbf{g}_{0,0}$. In this code, we symbolically compute the partial derivatives that are the entries of $\mathbf{J}$.

Another routine `create_jtf()` is used to generate the expression $r = -2\mathbf{J}^T\mathbf{F}$ for the `evalJTF()` routine. Each row of $\mathbf{J}^T$ can be obtained using the same approach previously described. The partials


(a) Example residual terms

```
fit: w_fit*(X(0,0) - A(0,0))
h_reg: w_reg*(X(0,0) - X(1,0))
v_reg: w_reg*(X(0,0) - X(0,1))
```

Residual template
$\mathrm{v\_reg}_{0,0}$
$\mathrm{fit}_{0,0}$
$\mathrm{h\_reg}_{0,0}$
(b) Actual residuals mapped over the entire image.
(c) Residuals using a specific unknown $\mathbf{x}_{0,0}$
$\mathbf{x}_{0,1}$
relative index to center pixel
$\mathbf{x}_{0,0}$
$\mathrm{v\_reg}_{0,0}$
$\mathbf{x}_{-1,0}$
$\mathrm{h\_reg}_{-1,0}$
$\mathrm{h\_reg}_{0,0}$
$\mathbf{x}_{1,0}$
$\mathrm{v\_reg}_{0,-1}$
$\mathrm{fit}_{0,0}$
$\mathbf{x}_{0,-1}$
(d) Representation of non-zero entries in the expression $\mathbf{g} = 2\mathbf{J^TJp}$ that are required to calculate $\mathbf{g}_{0,0}$
$\mathbf{g}$
$\mathbf{J^T}$
$\mathbf{J}$
$\mathbf{p}$
$\mathbf{g}_{0,0}$
←required row→
= 2
unknowns →
residuals →
$\mathrm{fit}_{0,0}$ $\mathrm{h\_reg}_{0,0}$ $\mathrm{h\_reg}_{-1,0}$ $\mathrm{v\_reg}_{0,0}$ $\mathrm{v\_reg}_{0,-1}$
$\frac{\mathrm{dfit}_{0,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dh\_reg}_{-1,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dv\_reg}_{0,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dv\_reg}_{0,-1}}{\mathrm{dx}_{0,0}}$
Row corresponding to $\mathbf{g}_{0,0}$ has non-zeros for each residual containing $\mathbf{x}_{0,0}$
Rows are required for each non-zero column required in $\mathbf{J^T}$
$\mathbf{x}_{0,0}$ $\mathbf{x}_{1,0}$ $\mathbf{x}_{-1,0}$ $\mathbf{x}_{0,1}$ $\mathbf{x}_{0,-1}$
$\mathrm{fit}_{0,0}$ $\frac{\mathrm{dfit}_{0,0}}{\mathrm{dx}_{0,0}}$
$\mathrm{h\_reg}_{0,0}$ $\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dh\_reg}_{0,0}}{\mathrm{dx}_{1,0}}$
$\mathrm{h\_reg}_{-1,0}$ $\frac{\mathrm{dh\_reg}_{-1,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dh\_reg}_{-1,0}}{\mathrm{dx}_{-1,0}}$
$\mathrm{v\_reg}_{0,0}$ $\frac{\mathrm{dv\_reg}_{0,0}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dv\_reg}_{0,0}}{\mathrm{dx}_{0,1}}$
$\mathrm{v\_reg}_{0,-1}$ $\frac{\mathrm{dv\_reg}_{0,-1}}{\mathrm{dx}_{0,0}}$ $\frac{\mathrm{dv\_reg}_{0,-1}}{\mathrm{dx}_{0,-1}}$
residuals →
unknowns →
non-zero columns where each individual residual has support
$\mathbf{p}_{0,0}$ $\mathbf{x}_{0,0}$
$\mathbf{p}_{1,0}$ $\mathbf{x}_{1,0}$
$\mathbf{p}_{-1,0}$ $\mathbf{x}_{-1,0}$
$\mathbf{p}_{0,1}$ $\mathbf{x}_{0,1}$
$\mathbf{p}_{0,-1}$ $\mathbf{x}_{0,-1}$
unknowns →

(e) Matrix free expression for $\mathbf{g}_{0,0}$

$$\mathbf{g}_{0,0} = 2\frac{\mathrm{d}\mathbf{fit}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\frac{\mathrm{d}\mathbf{fit}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\mathbf{p}_{0,0} + 2\underbrace{\frac{\mathrm{d}\mathbf{h\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}}_{\text{from }\mathbf{J^T}}(\underbrace{\frac{\mathrm{d}\mathbf{h\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{0,0}}\mathbf{p}_{0,0} + \frac{\mathrm{d}\mathbf{h\_reg}_{0,0}}{\mathrm{d}\mathbf{x}_{1,0}}\mathbf{p}_{1,0}}_{\text{from }\mathbf{J}}) + ... = 2\mathrm{w\_fit}^2\mathbf{p}_{0,0} + 2\mathrm{w\_reg}(\mathrm{w\_reg}\mathbf{p}_{0,0} + -\mathrm{w\_reg}\mathbf{p}_{1,0}) + ...$$

Fig. 9. Generating an expression for `applyJTJ()` at a high-level. (a) The input to this analysis is a list of individual residuals defined in the IR (`fit`, `h_reg`, `v_reg`) that form a *template*. (b) The residual template is repeated over the image to generate the actual energy function. (c) We look at a specific unknown $\mathbf{x}_{0,0}$ and show the residuals that refer to it. Unknowns and residuals are named relative to this pixel (e.g., $\mathtt{h\_reg}_{-1,0}$ is the horizontal residual from the pixel to the left). (d) A visualization of $\mathbf{g} = 2\mathbf{J^TJp}$, showing the components needed to generate $\mathbf{g}_{0,0}$. The row of $\mathbf{J}^T$ corresponding to unknown $\mathbf{x}_{0,0}$ is needed. It has one non-zero for each residual in (c). This row will be multiplied against $\mathbf{Jp}$. The only rows of $\mathbf{Jp}$ needed correspond to the residuals appearing in $\mathbf{J^T}$ since other rows will be multiplied by 0. A row of $\mathbf{Jp}$ is calculated by multiplying non-zero entries in a row of $\mathbf{J}$, which occur each time a residual uses an unknown, against the corresponding row of $\mathbf{p}$. (e) Finally, Opt forms a matrix-free in-place version of the expression for $\mathbf{g}_{0,0}$ implied by the matrix multiplications, calculating each partial using one-pass differentiation.

in this row are then multiplied directly with their corresponding residual term in $\mathbf{F}$.

5.3.2 *Graph Residuals.* For graphs, residuals are defined on hyper-edges rather than on the domain of the unknown and our solver routines are mapped over residuals directly so we do not need an inverse mapping from unknown to residual. Instead one thread computes the part of an output that relates to the residual. Pseudocode to generate `applyJTJ()` for graphs is given in Fig. 11. At each residual, it generates one row of $\mathbf{Jp}$, and then performs the part of the multiplication for the rows of $\mathbf{g}$ that include partials for that residual. The output of this routine is a list of IR nodes that are atomically added into entries of $\mathbf{g}$.

6. OPTIMIZING GENERATED SOLVER ROUTINES

We need to translate IR for `evalF()`, `evalJTF()`, and `applyJTJ()` into efficient GPU functions. We simplify IR based on polynomial

```
function create_jtj_graph(graph_residuals)
   foreach graph_residual do
      Jp = 0
      -- handle Jp multiply against this residual
      foreach unknown u appearing in graph_residual do
        dr_du = differentiate(graph_residual,u)
        Jp +=  dr_du*P(u.index)
     end
      -- handle partial sums for Jt*Jp
      foreach unknown u appearing in graph_residual do
        dr_du = differentiate(graph_residual,u)
        insert atomic scatter:
            P_hat(u.index) += 2*dr_du*Jp
      end
    end
    return set of atomic scatters
end
```

Fig. 11. Pseudocode for generating JTJ for graph residual terms.

simplification rules, optimize the handling of boundary condition statements, and schedule the IR which generates GPU code.

Polynomial Simplification. Taking the derivative of IR tends to introduce more complicated IR. In particular, the application of the multi-variable chain rules introduces statements of the form $d_1 * p_1 + d_2 * p_2 + ...$ for each argument of an operator. Often some partials are zero, and terms in the sum can be grouped together. We take the approach of other libraries like `SymPy` [SymPy Development Team 2014] and represent primitive math operations as polynomials. In particular, additions and multiplications are represented as n-arity operators rather than binary, and we include a `pow` operator that raises an expression to a constant a^c. Where possible, primitives are represented in terms of these operators. For instance a/b is represented as ab^{-1} and $a - b$ as $a + -1 * b$.

Polynomial representation makes it easier to find opportunities for optimization such as constant propagation when the optimization first requires re-associating, commuting, or factoring expressions. For instance, when we add polynomials together, we factor out common terms when we can fold their constants together (e.g., the addition $(2 * a + 4 * b) + 3 * a$ simplifies to $5 * a + 4 * b$).

Importantly, the polynomial representation also gives our scheduler freedom to reorder long sums and products to achieve other goals, such as grouping terms with the same boundary statement into a single if-statement or minimizing register pressure.

During construction we optimize non-polynomial terms using constant propagation and applying algebraic identities. Before lowering into code, we also apply a factoring pass that applies a greedy multi-variate version of Horner's scheme [Ceberio and Kreinovich 2004] to pull common factors out of large sums.

Bounds Optimization. Boundary conditions introduce another source of inefficiency. Opt uses `InBounds` and `Select` to create boundary conditions and masks. Translating these expressions to code can introduce inefficiency in two ways. First, it is possible for the same bound to be checked multiple times. This frequently occurs in `applyJTJ()` when two partials are multiplied together since both partials often contain the same bound. Redundant checks also occur when reading from arrays since Opt must always check array bounds to avoid crashes. This check is often redundant with a `Select` already in the energy. Secondly, without optimization, `Select` statements need to execute both the true and false expressions. For many cases, this means that large parts of the IR, including expensive reads from global memory, do not actually need to be calculated but are performed anyway.

The common approach of generating two versions of code, one for the boundary region and a bounds-free one for the interior, is less effective on GPUs because they group threads into wide vector lines of 32 elements, which increases the size of the boundary by the vector width. For smaller sized problems, large portions of the image fall in the boundary region.

Instead, we address these two sources of inefficiency directly. We address the redundant bounds checks by augmenting our polynomial simplification routines to handle bounds as well. We represent bounds internally as polynomials containing boolean values b that are either 0 or 1. A `Select(b,e_0,e_1)` is then represented as `b*e_0 + ~b*e_1`. We simplify booleans raised to a power b^e to b. This representation allows polynomial simplification rules to remove redundant bounds through factoring. We favor booleans over other values during factoring to ensure this simplification occurs.

We address excessive computation and memory use due to bounds by determining when values in the IR need to be calculated. We associate a boolean *condition* with each IR node that conservatively bounds when it is used. These conditions are generated at `Select` statements and propagated to their arguments. To improve the effectiveness of this approach, we split large sums into individual reductions that update a summation variable. Each reduction can then be assigned a different condition. When we actually schedule code, we will only execute the code if its condition is true.

Scheduling and Code Generation. We translate optimized IR into actual GPU code by *scheduling* the order in which the code executes the IR. Our scheduler uses a greedy approach that is aware of our boundary optimizations. It starts with the instructions that generate the output values and schedules backwards, maintaining a list of nodes that are *ready* to be scheduled according to their dependencies. It iteratively chooses an instruction from the ready list that has the lowest *cost*, schedules it, and updates the list. Our cost function first prioritizes scheduling an instruction with the same *condition* as the previous instruction, grouping expressions that have the same bounds together into a single if-statement. It then prioritizes choices that greedily minimize the set of live variables at that point in the program, which can provide a small benefit for large expressions. We also prioritize the instruction that has been ready the longest, which also helps reduce the required registers [Tiemann 1989].

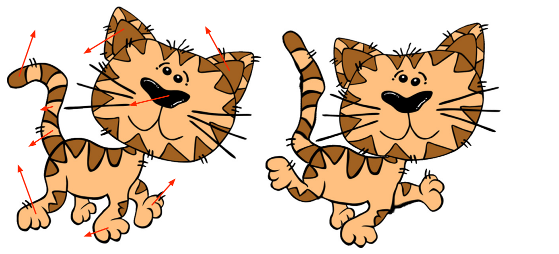
We translate the scheduled instructions into GPU code using Terra [DeVito et al. 2013]. Terra is a multi-stage programming language with meta-programming features that allow it to generate high-performance code dynamically. We use its GPU backend to produce CUDA code for the solver routines. To improve the performance, we automatically generate code to bind and load input data from GPU **textures**. In addition to having better caching behavior, textures also can perform the bounds check for loads automatically. Finally, the solver routines are inlined into the generic solver framework presented earlier. Because this code is compiled together, there is no overhead when invoking solver routines.

7. EVALUATION

To evaluate Opt, we implemented several optimization problems from the graphics literature in the language which are summarized in Figure 12. We evaluate overall performance by comparing Opt to four state-of-the-art application-specific in-place solvers optimized for GPUs and to the high-level Ceres solver [Agarwal et al. 2010]. We evaluate the effectiveness of our in-place solver framework by comparing to the performance achievable by general libraries such as cuSPARSE [NVIDIA 2012]. We also show the efficiency of our automatically generated solver routines by comparing them to hand-optimized equivalents. Finally, we implement three other problems which demonstrate the generality and expressiveness of Opt. The Opt code used for the energies of each example is provided as supplementary material.

7.1 Comparison to Existing Approaches

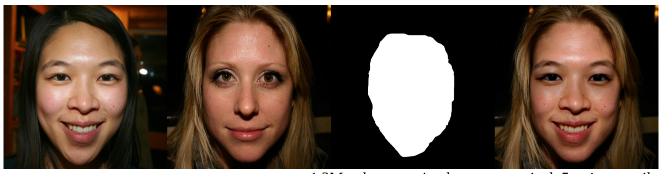
We compare Opt to four application-specific GPU solvers written and hand-optimized in CUDA: As-rigid-as possible (ARAP) Image Warping, ARAP Mesh Deformation, Shape from Shading, and Poisson Image Editing. We chose these applications, since they are commonly used in graphics research and optimized GPU code previously existed or could be easily adapted for the problem. We compare their performance to the performance of our automatically generated solver. In these comparisons, we select the Gauss-Newton backend of Opt to match the algorithmic design in the hand-written reference implementations.



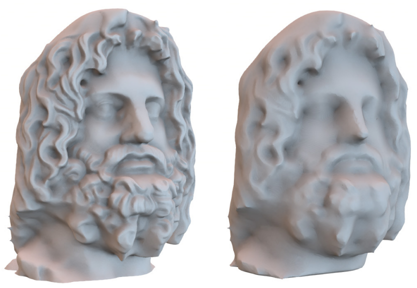
ARAP Image Warping 539k unknowns, 3 unknowns per pixel, 5-point stencil 0.928 MP/s, 1.1x faster than CUDA, 101x Ceres

ARAP Mesh Deformation 120k unknowns, 6 per vertex,vertex and edge energies 18.3 kverts/s, 1.5x faster than CUDA, 20.0x Ceres

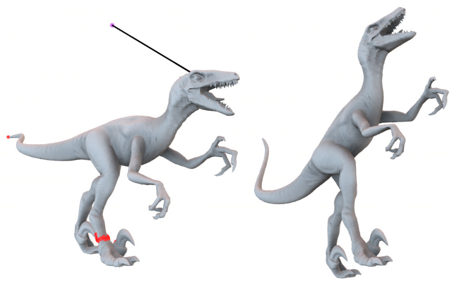
Shape From Shading 192k unknowns, 1 per pixel, 9-point stencil 1.84 MP/s, 1.4x faster than CUDA, 442x Ceres



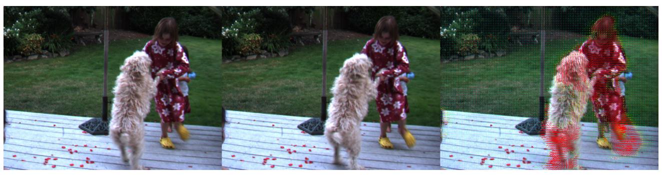
Poisson Image Editing 1.3M unknowns, 4 unknowns per pixel, 5-point stencil 22.4 MP/s 1.2x faster than CUDA, 40.3x Eigen



Cotangent Mesh Smoothing 132k unknowns, 3 per vertex, triangle and vertex energies, 236 kverts/s



Embedded Deformation 154k unknowns, 12 per vertex, vertex and edge energies 147kverts/s



Optical Flow 614k unknowns, 2 per pixel, 5-point stencil 4.70 MP/s

Fig. 12. Example problems implemented in Opt.

Using the same examples, we also compare against Ceres, a popular state-of-the-art CPU optimizer for non-linear least squares [Agarwal et al. 2010], because it has a similar programming model and solver approach. However, they are not entirely identical: Ceres uses double-precision and only supports Levenberg-Marquardt. We configured Opt and Ceres so that they finish with results of comparable image quality and made a best-faith effort to tweak the parameters of Ceres so it runs as fast as possible with acceptable results. To get the fastest results for the internal linear system, we configure Ceres to use its parallel PCG solver for Image Warping and Shape From Shading, and Cholesky factorization for Mesh Deformation.

Fig. 13 summarizes our performance. Opt performs better than the CUDA code in all cases, and 1–2 orders of magnitude faster than the Ceres code. Results are reported as throughput of an entire solve step using a GeForce TITAN Black GPU and, for CPU results, an Intel i7-4820K, a CPU released within the same year.

7.1.1 *ARAP Image Warping.* As-rigid-as-possible image warping is used to interactively edit 2D shapes in a way that minimizes a warping energy. It penalizes deviations from a rigid rotation, while warping to a set of user-specified constraints [Dvoroznak 2014]. It co-optimizes the new pixel coordinates along with the per-pixel rotation.

The CUDA implementation was adapted from the hand-written solver created by Zollhöfer et al. [2014] for real-time non-rigid reconstruction. It requires around 480 lines of code to implement. Of that, 200 were devoted to the Gauss-Newton solver, and 280 to expressions for the solver routines. In this comparison, we jointly solve for rotations and translations, following the hand-written reference implementation. Note that alternating between rotation and translation in a global-local flip-flop solve is also feasible in Opt; however, overall convergence is typically worse than the joint solve [Zollhöfer et al. 2014].

In comparison, the solver generated by Opt runs about 10% faster (likely due to better bounds handling), and only requires about 20 lines of code to describe. Furthermore, the effort required to get correct results is significantly less. Debugging the hand-written in-place solver routine in the CUDA solver originally took weeks due to the complicated cross terms that create dependencies between offsets of one pixel and the angles at a neighbor.

Ceres code is more comparable in size to Opt, at around 100 lines, but it runs 100 times slower. One of the reasons is that Opt represents the connectivity of the problem implicitly through stencils relations, while Ceres requires the user to specify energies using a graph formulation.

7.1.2 *Mesh Deformation.* As-rigid-as possible mesh deformation [Sorkine and Alexa 2007] is a variant of the previous example that shows Opt's ability to run on mesh-based problems using its graph abstraction. It defines a warping energy on the edges of the mesh rather than neighboring pixels and uses 3D coordinate frames.

The CUDA solver was also adapted from Zollhöfer et al. [2014]. It is similar in size to the previous example, with around 200 lines devoted to expressing the energy, JTF, and JTJ.

The Opt solver is expressed in only around 25 lines, but runs 40% faster due to our reduction-based approach for calculating residuals. In the original solver, the authors only tried the simpler approach of using one pass to compute $\mathbf{t} = (\mathbf{J}\mathbf{p})$ and a second for $\mathbf{J}^T\mathbf{t}$. Opt's high-level model allowed us to experiment with different approaches more easily during development.

Finally, Opt performs around 20 times faster than a Ceres example implemented in around 100 lines of code. The performance difference is less dramatic because in this case Opt needs to load the connectivity of the problem from the graph data structure.

7.1.3 *Shape From Shading.* In Shape From Shading we use an optimizer to refine depth data captured by RGB-D scanners [Wu et al. 2014]. It uses a high-resolution color image and an estimate of the lighting based on spherical harmonics to refine the lower resolution depth information.

Shape from Shading is our most complex problem. It is adapted from Wu et al.'s work [2014]. The original implementation was a patch solver variation of a Gauss Newton solver that used shared memory at the expense of per-iteration convergence. For a more direct comparison, we ported the original code into a non-patch solver, which actually improved the convergence time. The CUDA code includes 445 lines to express the energy, JTJ, and JTF calculations. It took several months for a group of researchers to implement and optimize.

In comparison, the Opt solver code is around 100 lines and runs 50% faster. Some of this improvement is due to using texture objects to represent the images, which is an optimization that the original authors did not have time to do.

Shape from Shading also benefits from using pre-computed arrays. We instruct Opt to pre-compute a lighting term and a boundary term that are expensive to calculate and used by the energy of multiple pixels. Without this annotation, Opt runs nearly 10 times slower. We expect that other complicated problems will have similar behavior and pre-computed arrays will give the user an easy way to experiment with how the computation is scheduled.

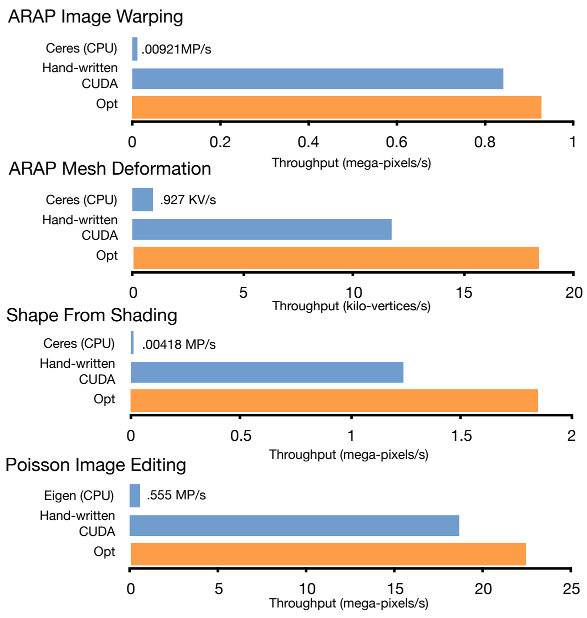


Fig. 13. The solvers generated by Opt perform better than application-specific GPU solvers, despite requiring significantly less effort to implement. In some cases, they perform two orders of magnitude better than Ceres implementations which take comparable effort to implement.

Compared to Ceres (340 lines), Opt runs over 400 times faster. Opt can take advantage of the implicit connectivity of an image-based problem, and can pre-compute expensive lighting terms, which Ceres does not do.

7.1.4 *Poisson Image Editing.* Poisson Image Editing is used to splice a source image into a target image without introducing seams [Pérez et al. 2003]. Its energy function preserves the gradients of the source image while matching the boundary to gradients in the target image. The energy in this problem is actually linear. In addition to non-linear energies, the Gauss-Newton method handles *linear* problems in a unified way that does not require algorithmic changes. In this case, because all residuals are linear functions of the unknowns, $\mathbf{J}$ is a constant matrix independent of $\mathbf{x}$. All second order derivatives are zero, which implies that the Gauss-Newton approximation is exact and the optimum can be reached after a single non-linear iteration.

To compare against a CUDA version, we adapted the Image Warping CUDA example to use this Poisson Image Editing term, which uses about 67 lines for the energy, JTF, and JTJ. Opt performs about 20% faster and uses only about 15 lines of code. Since this problem is linear, we also compare against Eigen [Guennebaud et al. 2010], a high-performance linear-algebra library for CPUs using `Cholesky with pre-ordering`, since it was fastest. The entire Opt solve was over 40 times faster than Eigen's matrix solve (not including its matrix setup time), due to Opt's ability to implicitly represent the connectivity of the matrix.

7.2 Evaluation of the Solver Approach

A key insight of previous hand-written GPU methods adapted by our framework is that it is more efficient to compute $\mathbf{J}$ in-place rather than store $\mathbf{J}$ or $\mathbf{J}^T\mathbf{J}$ as a sparse matrix. This approach can be faster

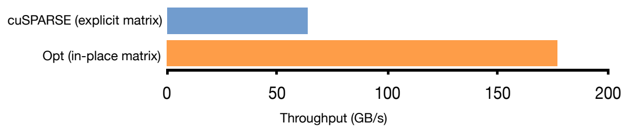


Fig. 14. Performance of the matrix product $(\mathbf{J}^T\mathbf{J})\mathbf{p}$ for Image Warping. Opt's in-place method performs several times faster than the multiplication of an explicit matrix used by cuSPARSE. This does not include the time that cuSPARSE requires to compute the matrix values and store them, so cuSPARSE will be much slower in practice.

for two reasons. First, the locations of the non-zero entries in the matrix are implicitly represented by the problem domain (either an image or a graph), and are not loaded explicitly. Secondly, entries in the matrix can often be recomputed using less total memory bandwidth than loading the full $\mathbf{J}$ matrix. In the extreme case, such as Poisson Image Editing, the problem is linear and the weights can be folded directly into the code.

We compare this architecture to the traditional library-based approach in Fig. 14 by porting the $(\mathbf{J}^T\mathbf{J})\mathbf{p}$ multiplication of the Image Warping example to cuSPARSE, a high-performance GPU sparse matrix library [NVIDIA 2012]. Opt is nearly 3 times faster than cuSPARSE. Furthermore, cuSPARSE will perform worse in practice because this measurement does not include the actual calculation of the matrix nor the code to store it. Opt performs these calculations implicitly inside the multiplication. Note that an alternate cuSPARSE approach, which multiplies in two passes $\mathbf{J}^T(\mathbf{J}\mathbf{p})$, is slower at 29.4GB/s.

Opt is faster because it (1) represents the connectivity implicitly and (2) calculates matrix entries in-place. We can separate these effects by simulating implicit connectivity without in-place matrices using Opt's pre-computed arrays. Marking the residual values as pre-computed arrays causes their values and derivatives (that is, $\mathbf{J}$) to be stored in memory. In this configuration for Image Warping, Opt runs at 92GB/s (slower than regular Opt, but faster than cuSPARSE). For Image Warping, loading the matrix from memory is slower because JTJ has 12M non-zeros (J has 10M), while the entire problem can be represented using 2.3M floats. We expect users to use pre-computed arrays to experiment with the balance between re-compute and memory usage.

7.3 Evaluation of generated solver routines

Our approach relies on the symbolic translations of energy functions into efficient solver routines using the optimizations described in Sec. 6. Compared to hand-written code, this code is much easier to write and maintain, but inefficient translations could make it too slow. To show the effectiveness of our symbolic translations and optimization, we compare our generated solver routines to hand-written versions that were taken from the pre-existing CUDA code and slotted into our solver.

Fig. 15 shows the results of our optimizations compared to the hand-written versions of JTF and JTJ ported from the CUDA examples and modified to use texture loads. We show the effect of our optimizations by starting with a version that naively translates the IR to code doing no simplification of the expressions (none). This roughly simulates how an auto-differentiation approach based on dual numbers would perform. We then turn on polynomial simplification (poly), bounds optimization (bounds), texture loads (texture), and register minimization (register).

The optimizations increase performance up to 8x in the case of Shape From Shading, and are necessary for Opt to perform at or above the speed of hand-written code. Performing polynomial simplifications improves the results of all examples. The improvement is more pronounced for the image-based examples, probably because graph-based examples are bottle-necked by fetching sparse data from memory rather than the expressions themselves.

Our optimizations to bounds address redundant bounds checks and unnecessary reads that can occur when translating `Select` expressions to code. They include representing bounds as booleans, factoring the bounds out of polynomial terms, and scheduling expressions to run conditionally. They provide a significant improvement for both Shape From Shading and Image Warping. Mesh Deformation does not improve because it does not use `Select`.

Shape From Shading shows a significant benefit from texture use, but the other example show no effect. In hand-written code this transform is not always attempted, since it may not improve all code and puts additional restrictions on data layout. Opt does the transform automatically, benefiting where possible. Finally our register minimization heuristic provides a small benefit to Shape From Shading's JTJ function.

8. EXPRESSIVENESS OF OPT

Our results section focuses on examples from the literature where state-of-the-art hand-written code previously existed and can be compared. Opt's programming model is also able to handle a wider variety of *general* non-linear least squares problems, which is at the core of a variety of computer graphics and vision problems tasks.

8.1 Examples

We implement three new GPU solvers with Opt that highlight the expressiveness of Opt's programming model.

Embedded Deformation. is a popular alternative method to as-rigid-as-possible deformation [Sumner et al. 2007]. Rather than solving for a per-vertex rotation, it solves for a full affine transformation. Compared to writing a solver by hand, writing Embedded Deformation in Opt was an easy process because it only required increasing the number of unknowns and changing the energy terms of our as-rigid-as-possible energy, which amounted to tens of lines of code and under an hour of work. The solver it produces can deform a 12k vertex mesh at an interactive rate of 12 frames per second and only requires an energy function of around 40 lines.

Cotangent-weighted Laplacian Smoothing. is a method for smoothing meshes that tries to preserves the area of triangles adjacent to each edge [Desbrun et al. 1999]. It adapts well to meshes with non-uniform tessellations. This example highlights Opt's ability to define residuals on larger components of a mesh by defining a hyper-edge in our graph representation that contains all the vertices in a wedge at each edge. We show the power of Opt by implementing a small variant that allows the cotangent weights to be recomputed during deformation instead of using the values from the original mesh. This variant is normally hard to write since it introduces complicated derivative terms. Opt generates them automatically, making it easier to experiment with small variants on existing energy functions. Opt generates a solver from around 45 lines of code that can smooth a 44k vertex mesh at 5 fps.

Dense Optical Flow. computes the apparent motion of objects between frames in a video at the pixel level. We implement a hierarchical version of Horn and Schunck's algorithm [1981] using an iterative relaxation scheme. Because optical flow is searching for correspondences in images, its unknowns are used to sample values from the input frames. We support this pattern using a *sampled*

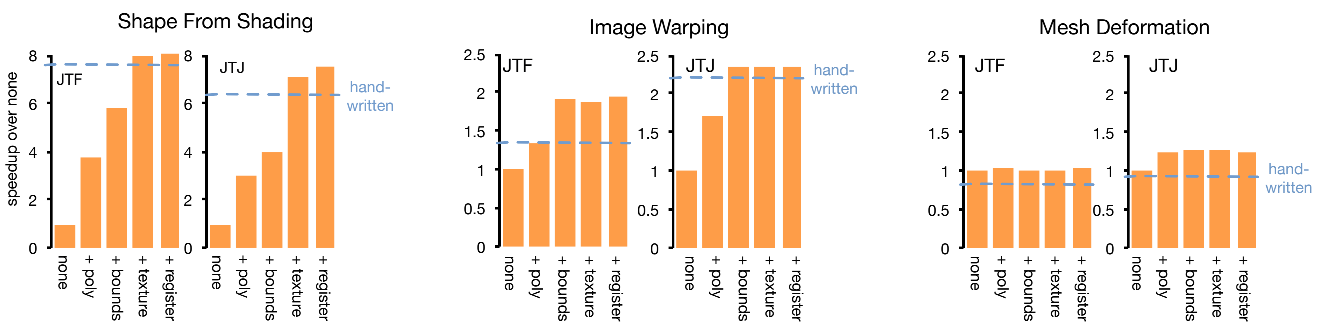


Fig. 15. The effect of our optimizations on generated code for solver routines. The optimizations are necessary to match the performance of handwritten equivalents, and in many cases the functions run faster.

image operator, which can be accessed with arbitrary (u, v) coordinates. When these coordinates are dependent on the unknown image, the user provides the directional derivatives of the sampled image as other input images, which will be used to lookup the partials for the operator in the symbolic differentiation. The solver Opt generates from around 20 lines of code solves optical flow at 4.70MP/s.

8.2 Functionality

Opt's features also allow many additional kinds of programs to be expressed.

Domains. Opt is able to exploit the implicit structure and connectivity of general n-dimensional arrays. While we have shown several examples on images (i.e., 2D-arrays), optimizations are often performed on volumetric (e.g., [Innmann et al. 2016]) or time-space (e.g., [Wand et al. 2007]) domains, all of which are subsets of n-D arrays and fall within the scope of Opt.

In addition to regular domains, Opt efficiently handles explicit structure, provided in the form of general graphs. These domains include manifold meshes and general non-manifolds. For instance, non-rigid mesh deformation objects (e.g., [Sumner et al. 2007; Sorkine and Alexa 2007]) fall into this category, as well as widely-used global bundle adjustment methods [Triggs et al. 1999; Snavely et al. 2006; Agarwal et al. 2009].

Opt also allows the definition of energies on *mixed* domains. For example, an objective may contain dense regularization terms affecting every pixel of an image and a sparse set of correspondences from a fitting term. Here, the regularization energy is implicitly encoded in a 2D image domain, and the data term may be provided by a sparse graph structure.

On all of these domains, Opt provides automatic derivation of objective terms, and generates GPU specifically optimized for a given energy function at compile time.

Multi-pass Optimization. In many scenarios, solving a single optimization is not enough, but instead requires multiple passes of different non-linear solves. Often, hierarchal, coarse-to-fine solves are used to achieve better convergence, or sometimes problem-specific flip-flip iteration can be applied (e.g., ARAP flip-flop by Sorkine and Alexa [2007]). Another common case are dynamic changes in the structure the optimization problem. For instance, fitting a mesh to point-cloud data in a non-rigid fashion is typically achieved by searching for correspondences between optimization passes (e.g., non-rigid iterative closest point) [Li et al. 2009; Zollhöfer et al. 2014]. Changes to the correspondences also change the structure of the sparse fitting terms.

In all of these examples, custom code is required at specific stages during optimization. To support this code in Opt, we take an approach similar to multi-pass rendering in OpenGL. Between iterations of the Opt solver or between entire solves, users can perform arbitrary modifications to the underlying problem state in C/C++. Optimization weights can be changed (e.g., for parameter relaxation), underlying data structures may be dynamically updated (e.g., correspondence search or feature match pruning in bundle adjustment problems), or hierarchal and flip-flop strategies can be applied using multiple-passes. This approach allows Opt to support a wide range of solver approaches, while providing an efficient optimization backend for their inner kernels.

Robustness. A common approach for non-linear least squares optimization problems in computer vision is the use of robust kernels. Here, auxiliary variables are introduced in order to determine the relevance of a data term components as part of the optimization formulation. For instance, this strategy is often used in bundle adjustment or non-rigid deformation frameworks to determine the reliability of correspondences [Triggs et al. 1999; Li et al. 2009; Zach 2014; Zollhöfer et al. 2014]. In Opt, it is easy to add these terms as additional values in the unknown for energy functions on single and mixed domains.

9. FUTURE WORK AND CONCLUSION

From a high-level description of the energy based on stencils and graphs, Opt produces customized GPU solvers for non-linear least squares problems that are faster than state-of-the-art application-specific hand-coded solvers and orders-of-magnitude faster than Ceres. Our in-place solver approach is more efficient than explicit matrix routines, and the generated solver routines out-perform hand-written equivalents.

We believe that Opt's approach of using in-place solvers with automatically generated application-specific routines can be extended to work with more expressive energy functions, more platforms beyond GPUs, and more kinds of solvers.

Currently the Opt language limits what energies can be expressed efficiently. On images, our implementation limits energies to a constant-sized neighboring stencil. However, we can extend Opt to support other neighborhood functions such as affine transformations of indices as long as the neighborhood function is invertible. We also plan to extend our graph language to support the ability to reference a variable number of neighbors (such as the edges around a vertex) to make certain energies easier to express.

While some of our specific optimizations are tailored to GPUs, the overall approach of symbolically calculating and simplifying functions needed by the solver is applicable to other platforms such as multi-core CPUs, or even networked clusters of machines for large problems.

Finally, there are a lot of optimization problems in graphics that are not suited to the Gauss-Newton or Levenberg-Marquardt approach. Many optimization problems in the graphics literature are more efficiently solved using other techniques such as shape deformation with an interior-point optimizer [Levi and Zorin 2014] or mesh parametrization using quadratic programming [Kharevych et al. 2006]. Others require additional features beyond the specification of an energy, such as constraints. We believe these solvers would also benefit from the architecture proposed in Opt, where a general in-place solver library is augmented with automatically derived application-specific routines. Many solvers share the need for the same routines (e.g., the gradient), so as Opt grows to support more solvers, less effort will be needed in the compiler to generate new routines. Eventually, we hope that computer graphics and vision practitioners can put most energy functions from the literature into a system like Opt and automatically get a high-performance solver. We believe that Opt is a significant first step in this direction.

ACKNOWLEDGMENTS

This work has been supported by the DOE Office of Science ASCR in the ExMatEx and ExaCT Exascale Co-Design Centers, program manager Karen Pao; DARPA Contract No. HR0011-11-C-0007; fellowships and grants from NVIDIA, Intel, and Google; the Max Planck Center for Visual Computing and Communications, and the ERC Starting Grant 335545 CapReal; and the Stanford Pervasive Parallelism Lab (supported by Oracle, AMD, Intel, and NVIDIA). We also gratefully acknowledge hardware donations from NVIDIA Corporation.

REFERENCES

AGARWAL, S., MIERLE, K., AND OTHERS. 2010. Ceres solver. `http://ceres-solver.org`.

AGARWAL, S., SNAVELY, N., SIMON, I., SEITZ, S. M., AND SZELISKI, R. 2009. Building rome in a day. In *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, 72–79.

BOCHKANOV, S. 1999. Alglib. `http://www.alglib.net`.

BOYD, S. AND VANDENBERGHE, L. 2004. *Convex Optimization*. Cambridge University Press, New York, NY, USA.

CEBERIO, M. AND KREINOVICH, V. 2004. Greedy algorithms for optimizing multivariate horner schemes. *SIGSAM Bull. 38,* 1 (Mar.), 8–15.

DAI, A., NIESSNER, M., ZOLLÖFER, M., IZADI, S., AND THEOBALT, C. 2016. Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration. *arXiv preprint arXiv:1604.01093*.

DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. H. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. SIGGRAPH '99. New York.

DEVITO, Z., HEGARTY, J., AIKEN, A., HANRAHAN, P., AND VITEK, J. 2013. Terra: A multi-stage language for high-performance computing. PLDI '13. New York, 105–116.

DVOROZNAK, M. 2014. Interactive as-rigid-as-possible image deformation and registration. In *The 18th Central European Seminar on Computer Graphics*.

GRANT, M. AND BOYD, S. 2008. Graph implementations for nonsmooth convex programs. In *Recent Advances in Learning and Control*. Lecture Notes in Control and Information Sciences. Springer-Verlag Limited, 95–110.

GRANT, M. AND BOYD, S. 2014. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`.

GRIEWANK, A. AND WALTHER, A. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Second ed. SIAM, Philadelphia, PA, USA.

GRUND, F. 1982. Automatic differentiation: Techniques and applications. Lecture notes in computer science 120. *ZAMM 62,* 7, 355–355.

GUENNEBAUD, G., JACOB, B., ET AL. 2010. Eigen v3. http://eigen.tuxfamily.org.

GUENTER, B. 2007. Efficient symbolic differentiation for graphics applications. SIGGRAPH '07.

GUENTER, B., RAPP, J., AND FINCH, M. 2011. Symbolic differentiation in GPU shaders. Tech. Rep. MSR-TR-2011-31. March.

HORN, B. K. P. AND SCHUNCK, B. G. 1981. Determining optical flow. *ARTIFICAL INTELLIGENCE 17*, 185–203.

INNMANN, M., ZOLLHÖFER, M., NIESSNER, M., THEOBALT, C., AND STAMMINGER, M. 2016. Volumedeform: Real-time volumetric non-rigid reconstruction. *arXiv preprint arXiv:1603.08161*.

KHAREVYCH, L., SPRINGBORN, B., AND SCHRÖDER, P. 2006. Discrete conformal mappings via circle patterns. *ACM Transactions on Graphics (TOG) 25,* 2, 412–438.

KUMMERLE, R., GRISETTI, G., STRASDAT, H., KONOLIGE, K., AND BURGARD, W. 2011. g2o: A general framework for graph optimization. In *Robotics and Automation (ICRA), IEEE Int. Conf. on*. IEEE, 3607–3613.

LEVI, Z. AND ZORIN, D. 2014. Strict minimizers for geometric optimization. *ACM Transactions on Graphics (TOG) 33,* 6, 185.

LI, H., ADAMS, B., GUIBAS, L. J., AND PAULY, M. 2009. Robust single-view geometry and motion reconstruction. In *ACM Transactions on Graphics (TOG)*. Vol. 28. ACM, 175.

NOCEDAL, J. AND WRIGHT, S. J. 2006. *Numerical Optimization*, 2nd ed. Springer, New York.

NVIDIA 2012. *CUDA CUSPARSE Library*. NVIDIA.

PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. In *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. ACM, New York, NY, USA, 313–318.

RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31,* 4 (July), 32:1–32:12.

SNAVELY, N., SEITZ, S. M., AND SZELISKI, R. 2006. Photo tourism: exploring photo collections in 3D. In *ACM transactions on graphics (TOG)*. Vol. 25. ACM, 835–846.

SORKINE, O. AND ALEXA, M. 2007. As-rigid-as-possible surface modeling. In *Symposium on Geometry processing*. Vol. 4.

SUMNER, R. W., SCHMID, J., AND PAULY, M. 2007. Embedded deformation for shape manipulation. SIGGRAPH '07. New York.

SYMPY DEVELOPMENT TEAM. 2014. *SymPy: Python library for symbolic mathematics*.

THIES, J., ZOLLHÖFER, M., NIESSNER, M., VALGAERTS, L., STAMMINGER, M., AND THEOBALT, C. 2015. Real-time expression transfer for facial reenactment. *ACM Transactions on Graphics (TOG) 34,* 6.

THIES, J., ZOLLHÖFER, M., STAMMINGER, M., THEOBALT, C., AND NIESSNER, M. 2016. Face2face: Real-time face capture and reenactment of rgb videos. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*.

TIEMANN, M. D. 1989. The GNU instruction scheduler. Tech. rep., Cambridge, MA. March.

TRIGGS, B., MCLAUCHLAN, P. F., HARTLEY, R. I., AND FITZGIBBON, A. W. 1999. Bundle adjustmenta modern synthesis. In *Vision algorithms: theory and practice*. Springer, 298–372.

WAND, M., JENKE, P., HUANG, Q., BOKELOH, M., GUIBAS, L., AND SCHILLING, A. 2007. Reconstruction of deforming geometry from time-varying point clouds. In *Symposium on Geometry processing*. Citeseer, 49–58.

WEBER, D., BENDER, J., SCHNOES, M., STORK, A., AND FELLNER, D. 2013. Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. In *Computer Graphics Forum*. Vol. 32. Wiley Online Library, 16–26.

WEFELSCHEID, C. AND HELLWICH, O. 2013. OpenOF: Framework for sparse non-linear least squares optimization on a GPU. In VISAPP.

WOLFRAM RESEARCH. 2000. Mathematica.

WU, C., ZOLLHÖFER, M., NIESSNER, M., STAMMINGER, M., IZADI, S., AND THEOBALT, C. 2014. Real-time shading-based refinement for consumer depth cameras. *ACM Transactions on Graphics (TOG) 33,* 6.

ZACH, C. 2014. Robust bundle adjustment revisited. In *Computer Vision–ECCV 2014*. Springer, 772–787.

ZOLLHÖFER, M., DAI, A., INNMANN, M., WU, C., STAMMINGER, M., THEOBALT, C., AND NIESSNER, M. 2015. Shading-based refinement on volumetric signed distance functions. *ACM Trans. Graph. 34,* 4 (July), 96:1–96:14.

ZOLLHÖFER, M., NIESSNER, M., IZADI, S., RHEMANN, C., ZACH, C., FISHER, M., WU, C., FITZGIBBON, A., LOOP, C., THEOBALT, C., AND STAMMINGER, M. 2014. Real-time non-rigid reconstruction using an RGB-D camera. *ACM Transactions on Graphics (TOG) 33,* 4.