

Thallo – Scheduling for High-Performance Large-scale Non-linear Least-Squares Solvers

MICHAEL MARA, Stanford University

FELIX HEIDE, Princeton University

MICHAEL ZOLLHÖFER, Stanford University

MATTHIAS NIESSNER, Technical University of Munich

PAT HANRAHAN, Stanford University

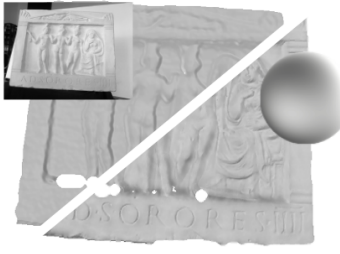

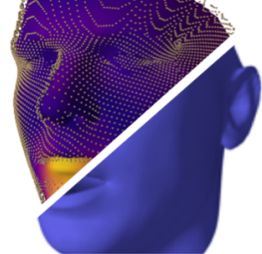

JOINT SHAPE-AND-SHADING	BUNDLEFUSION	BLENDSHAPE FITTING	POISSON DECONVOLUTION
			
Opt: 110 lines 2259.6ms	Handwritten 366 lines CUDA [2017] 5.57ms	Opt: 49 lines 1057.8ms	Opt: 37 lines 195.9ms
Thallo: 99 lines 171.9ms	Thallo: 86 lines 2.96ms	Thallo: 33 lines 46.3ms	Thallo: 27 lines 46.6ms
90% as long. 13.1× faster than Opt.	25% as long. 1.8× faster than handwritten.	59% as long. 22.8× faster than Opt.	38% as long. 4.2× faster than Opt.

Fig. 1. Thallo is a domain-specific language (DSL) that generates tailored high-performance GPU solvers and schedules from a concise, high-level energy description without the hassle of manually constructing and maintaining tedious and error-prone solvers. We compare algorithms in our language, Thallo, to state-of-the-art implementations of four visual computing applications, including hand crafted GPU solvers and solvers generated with Opt [2017]. Thallo code is compact, and exceeds state-of-the-art performance across diverse applications in graphics and vision.

Large-scale optimization problems at the core of many graphics, vision, and imaging applications are often implemented by hand in tedious and error-prone processes in order to achieve high performance (in particular on GPUs), despite recent developments in libraries and DSLs. At the same time, these hand-crafted solver implementations reveal that the key for high performance is a problem-specific schedule that enables efficient usage of the underlying hardware. In this work, we incorporate this insight into Thallo, a domain-specific language for large-scale non-linear least squares optimization problems. We observe various code reorganizations performed by implementers of high-performance solvers in the literature, and then define a set of basic operations that span these scheduling choices, thereby defining a large scheduling space. Users can either specify code transformations in a scheduling language or use an autoscheduler. Thallo takes as input a compact, shader-like representation of an energy function and a

(potentially auto-generated) schedule, translating the combination into high-performance GPU solvers. Since Thallo can generate solvers from a large scheduling space, it can handle a large set of large-scale non-linear and non-smooth problems with various degrees of non-locality and compute-to-memory ratios, including diverse applications such as bundle adjustment, face blendshape fitting, and spatially-varying Poisson deconvolution. Abstracting schedules from the optimization, we outperform state-of-the-art GPU-based optimization DSLs by an average of 16× across all applications introduced in this work, and even some published hand-written GPU solvers by 30%+.

ACM Reference Format:

Michael Mara, Felix Heide, Michael Zollhöfer, Matthias Nießner, and Pat Hanrahan. 2021. Thallo – Scheduling for High-Performance Large-scale Non-linear Least-Squares Solvers. *ACM Trans. Graph.* 1, 1, Article 1 (January 2021), 14 pages. <https://doi.org/10.1145/3453986>

1 INTRODUCTION

Optimization lies at the core of a variety of computer graphics, vision, and imaging problems. Traditionally, high-performance optimizers for image processing and computer vision algorithms have been hand-crafted specifically for these application domains, with little cross-domain use of a method or its implementation.

Authors' addresses: Michael Mara, Stanford University, mmara@cs.stanford.edu; Felix Heide, Princeton University, fheide@cs.princeton.edu; Michael Zollhöfer, Stanford University, michael@zollhoefer.com; Matthias Nießner, Technical University of Munich, niessner@tum.de; Pat Hanrahan, Stanford University, hanrahan@cs.stanford.edu.

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics*, <https://doi.org/10.1145/3453986>.

Expressing tasks instead as formal optimization problems allows researchers to reuse energy functions and problem settings that generalize across problem domains. Development and adoption is accelerated in practice by efficient tools that enable researchers to prototype optimization methods. Solving non-linear optimization problems efficiently is an open problem, though successful frameworks [Abadi et al. 2015; Paszke et al. 2017; Ragan-Kelley et al. 2013] exist for stencil-based homogeneous image and neural network operations.

Specifically, global non-linear least-squares (NLLS) optimization problems are commonly used for a large variety of approaches across multiple problem domains, ranging from bundle adjustment [Agarwal et al. 2011; Dai et al. 2017; Triggs et al. 2000] to 3D reconstruction [Newcombe et al. 2011] to face tracking [Thies et al. 2016]. Furthermore, non-linear least-squares problems form the inner loop of many more complicated optimization problems, such as those in interactive computational imaging [Heide et al. 2014]. In many of these applications, run-time performance is often critical, and real-time performance typically is a strict requirement. As many of these optimization problems are also computationally expensive, researchers are currently required to invest significant effort in their implementation – in sharp contrast to readily available domain-specific deep learning frameworks [Abadi et al. 2015; Paszke et al. 2017].

A common way of developing new algorithms using optimization is to adopt solver libraries that are based on auto-differentiation, such as Ceres [Agarwal et al. 2010a] or g2o [Kümmerle et al. 2011]. These libraries make it easy to try different solver types. However, the resulting performance, in particular on parallel architectures, is relatively slow. On the other end of the spectrum, we see custom solvers that are tailored to a specific problem and implemented by hand, for instance as efficient CPU code [Grant and Boyd 2014] or GPU code [Thies et al. 2016]. Here, skilled programmers achieve impressive run-time performance; however, they require significant implementation effort: deriving derivatives by hand, structuring a solver tailored to the problem, and scheduling options, which makes rapid prototyping infeasible.

Domain-specific languages (DSLs) offer a unique opportunity to combine the benefits of these two approaches. For stencil-based problems, where the solver or individual operations can exploit regularity, we have seen significant progress [Devito et al. 2017; Heide et al. 2016]. These DSLs benefit from encoding matrix structures implicitly in the compiled code, which also allows for matrix-free implementations. While progress has been impressive, these state-of-the-art optimization DSLs often still cannot produce efficient code for a variety of problems.

The issue is that different problem types require massive reorganization of the computation in order to achieve high performance. For example, the handwritten BundleFusion [Dai et al. 2017] (see Section 6) implementation is 15× faster than one generated by the most efficient nonlinear-least squares DSL today.

The solution to this is *scheduling*, explicitly modifying the method by which work is assigned to computing resources in terms of reuse, parallelization and locality. Splitting problems into an algorithm

description and execution schedule has been demonstrated in Halide [Li et al. 2018; Ragan-Kelley et al. 2013] as successful strategy for efficient image processing pipelines.

In order to schedule our computation appropriately we need to define a space of schedules. We do this by first observing various code reorganizations performed by implementers of high-performance solvers in the literature [Dai et al. 2017; Thies et al. 2016], and then defining a set of basic operations that span these scheduling choices. Once this scheduling space is defined, we introduce autoscheduling in this space.

Specifically, we introduce a scheduling language for a NLLS DSL that generates GPU-based solvers. We designed the language, called Thallo, to span the space of schedules that exist in the nonlinear least-square solving literature.

Specifically, this work makes the following contributions:

- We introduce scheduling transforms for nonlinear least square (NLLS) problems that define a space of GPU schedules encompassing prior work.
- We create a realization of these transforms in a scheduling language for a NLLS DSL.
- We introduce a heuristic autoscheduler for this scheduling language. We validate that this autoscheduler achieves high performance schedules for diverse GPU architectures and across problem instances.
- We demonstrate that our framework achieves state-of-the-art results on a variety of non-local optimization problems in graphics, vision and imaging, while producing efficient solver implementations that outperform existing GPU DSLs by an average of 16× and are comparable or even slightly outperform hand-written implementations.
- We release the framework and implemented applications to the open-source community.¹

2 BACKGROUND

Non-linear Least Squares Optimization Problems. Unconstrained non-linear least squares optimization problems are ubiquitous in graphics, vision, and imaging.

At the core of these problems, a solver is used to find the best value \mathbf{x}^* , such that an optimization objective E is minimized

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} E(\mathbf{x}), \text{ with } E(\mathbf{x}) = \sum_{i=1}^R [\mathbf{F}_i(\mathbf{x})]^2 = \|\mathbf{F}(\mathbf{x})\|_2^2 \quad (1)$$

The energy function E is expressed as the sum of squared residual terms \mathbf{F}_i . These residuals can be arbitrary functions of the input, making the problem (potentially) non-linear and non-convex [Boyd and Vandenberghe 2004].

A concrete example is fitting a parametric curve $y = f(x)$, with $f(x) = b_0 \cdot \sin(b_1 \cdot x) + b_1 \cdot \cos(b_0 \cdot x)$, to an array of (x, y) observation pairs of size N . The optimization problem is to find the values of the unknowns, b_0 and b_1 , that produces the curve geometrically closest

¹<https://www.thallo-lang.org/>

to the observations

$$\arg \min_{b_0, b_1} \sum_{n=1}^N (y_n - (b_0 \cdot \sin(b_1 \cdot x_n) + b_1 \cdot \cos(b_0 \cdot x_n)))^2 \quad (2)$$

Problems of this form are often tackled by methods such as Gauss-Newton (GN) or Levenberg-Marquard (LM) [Nocedal and Wright 2006]. GN and LM perform a fixed point iteration by iteratively solving a sequence of linear problems, formed at each step by using a first-order approximation of the Hessian at the solution of the previous iteration step. This approach permits data-parallel implementations, which have been exploited to solve a large variety of sparse [Meka et al. 2016; Wu et al. 2014; Zollhöfer et al. 2014], semi-dense [Dai et al. 2017; Parikh and Boyd 2013], and dense [Izadi et al. 2011; Thies et al. 2016] computer graphics and vision problems at real-time frame rates. In these approaches, each of the linearized problems is solved using a data-parallel preconditioned conjugate gradient (PCG) solver, where each implementation was made by a domain expert tailored to the respective problem.

High-level CPU Solvers. High-level CPU solvers are widely employed to tackle many optimization problems. For example, solvers such as CVX [Grant and Boyd 2008, 2014] construct a specialized CPU solver for each type of problem. Google’s Ceres [Agarwal et al. 2010a] solver combines operator overloading with template meta-programming to implement automatic backwards auto-differentiation to solve general non-linear least squares optimization problems on the CPU. CPU libraries such as Alglib [Shearer and Wolfe 1985], GTSAM [Dellaert 2012], and g2o [Kümmerle et al. 2011] enable the implementation of a problem-specific solver by the manual specification of the objective function and optionally its derivatives. CPU solvers with support for auto-differentiation are easy to use and free programmers from the burden of manual differentiation, but do not scale to real-time performance even for medium-sized real-world optimization problems. Our method aims for the same functionality and ease of programming, only requiring the specification of the optimization objective; however, Thallo automatically generates problem-specific and highly efficient data-parallel GPU optimization code that can use the more parallel GPU hardware to outperform similar CPU solvers.

Hand-written Non-Linear GPU Solvers. One widely employed technique to overcome the performance limitations of high-level CPU solvers is to hand-design problem-specific data-parallel GPU solvers, achieving real-time performance even for large-scale computer graphics and vision problems. Sparse problems that have been accelerated based on data-parallel solvers range from real-time deformable mesh tracking [Innmann et al. 2016; Zollhöfer et al. 2014] to shape-from-shading [Wu et al. 2014; Zollhöfer et al. 2015] and intrinsic video decomposition [Meka et al. 2016]. Semi-dense problems include the real-time GPU bundling approach of [Dai et al. 2017], but such problems also have various applications in computational imaging [Heide et al. 2016; Parikh and Boyd 2013]. Dense problems that have been tackled using handwritten GPU solvers include model-to-frame camera tracking [Izadi et al. 2011; Newcombe et al. 2011; Nießner et al. 2013] and parametric model fitting for real-time face reconstruction [Thies et al. 2015, 2016]. To design an efficient data-parallel solver the implementer must take the low-level problem

structure into account. For example, different hand-designed solvers take advantage of the varying sparsity patterns of the underlying Jacobian, which might be a sparse, semi-dense, or dense matrix. Skilled programmers achieve impressive solver speeds, but hand-designed data-parallel solvers require a significant implementation effort, i.e., manual differentiation and finding a suitable problem-dependent parallelization strategy through trial and profiling. Therefore, writing hand-designed GPU solvers is a tedious and error-prone process, which makes rapid prototyping infeasible.

Domain Specific Languages. Domain Specific Languages (DSLs) are a tool to enable the easier development of domain-specific application code and allow for rapid prototyping, which is especially important for research projects and design space exploration. There are a large variety of DSLs for tackling different tasks in computer graphics and vision. DSLs such as Ebb [Bernstein et al. 2016] and Simit [Kjolstad et al. 2016] allow users to express and abstract linear algebra operations over graphs and other relations on heterogeneous architectures. ProxImaL [Heide et al. 2016] is a DSL for tackling inverse problems in computational imaging that are defined on regular grid-based domains, such as deconvolution and denoising. Indigo [Driscoll et al. 2018] is an embedded DSL for implementing linear operators for computational imaging. It uses an expression tree representation to combine matrix-free and materialized matrix components. PyTorch [Paszke et al. 2017] and TensorFlow [Abadi et al. 2015] are popular DSLs used for solving large-scale machine learning problems, e.g., training deep neural networks using mini-batch gradient descent. Most closely related to our approach is Opt [Devito et al. 2017], an optimization DSL that enables the data-parallel solution of sparse stencil-based and graph-based optimization problems. Followup work has used it to implement fast GPU NLLS solvers for applications such as depth refinement in conjunction with deep learning [Laidlow et al. 2019], 3D avatar reconstruction [Zeitvogel and Laubenheimer 2018], and shading-based refinement [Deng et al. 2018]. In contrast to Opt, Thallo introduces the concept of scheduling to the non-linear solver implementations, allowing massive code transformations that are not available in Opt. As such, Thallo tackles a much wider class of non-linear least squares optimization problems.

Scheduling. Scheduling, or high-level specifications of computation and storage reorganization, has seen a surge of interest in the last few years. Halide [Ragan-Kelley et al. 2013] is a domain specific language that abstracts computations on images and tensors and makes it easier to write high-performance code. It introduces a split between specifying the algorithm and the schedule. The original paper introduces an expensive autotuning process for schedule discovery, but importantly exposes the scheduling language (based around imperfect loop nests) to the user. The Tensor Algebra Compiler [Kjolstad et al. 2017] and Taichi [Hu et al. 2019] decouple *data structures* from computation in tensor algebra and 3D spatial computation, respectively. Users implicitly change the schedule by changing the underlying data structures. Automatic differentiation was later added to Taichi [Hu et al. 2020]. Darkroom [Hegarty et al. 2014] formulates hardware stencil pipelines as linebuffer programs; reducing scheduling to choices in delay buffer sizes, which is solved as an integer linear program. Rigel [Hegarty et al. 2016] extends this

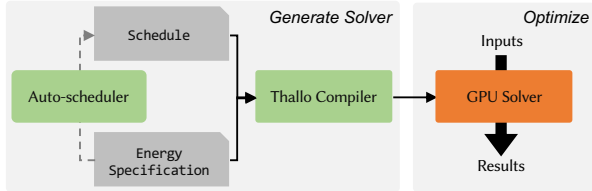


Fig. 2. Thallo at a glance. The core compiler takes a high-level energy specification and a schedule as input. The autoscheduler can automatically generate a good schedule from the energy specification, so novice to intermediate users never have to write their own schedule.

to a synchronous dataflow model to model multi-rate modules, sacrificing automatic scheduling, and exposing scheduling constructs (FIFO sizes) to the user. Manual scheduling in Halide is much easier than hand-writing implementations, but writing efficient schedules is much harder than writing the algorithm. Follow-up work seeks to efficiently and automatically schedule Halide pipelines [Adams et al. 2019; Mullapudi et al. 2016; Sioutas et al. 2019]. Gradient Halide [Li et al. 2018] extends Halide with reverse mode auto-differentiation to compute gradients of scalar functions, and a heuristic autoscheduler targeting GPUs. We take inspiration from these approaches for our own autoscheduler, using a drastically simplified rules cascade that nevertheless produces good schedules. Users can choose to provide their own schedules or use this autoscheduler. See Figure 2 for a high-level overview.

3 SCHEDULING CHOICES

We treat scheduling as a problem of splitting, re-ordering, and parallelizing loop nests over data-parallel pure functions similar to Halide [Ragan-Kelley et al. 2013]. Specifically, we rely on re-ordering and materialization (computing and storing for later reading) as core scheduling approaches, while introducing new constructs tailored to the NLLS-centered matrix calculus domain we operate in.

3.1 The Order of Computation in NLLS Solvers

The core scheduling choices for GN and LM solvers revolve around the computation of the Jacobian of the energy, and related matrix multiplies. In the following, we describe core scheduling aspects for this computation.

GN and LM perform a fixed point iteration by iteratively solving a sequence of linearized problems, where the non-linear problem is always linearized with respect to the solution of the previous iteration step based on a first order Taylor expansion. In each iteration step, a linear system of the form $\mathbf{J}^T \mathbf{J} \delta = \mathbf{J}^T \mathbf{F}$ has to be solved, where \mathbf{F} is the vector of residuals described in Equation 1 evaluated at the previous estimate of the unknowns \mathbf{x}_0 , and \mathbf{J} the Jacobian matrix of \mathbf{F} with respect to the unknowns at \mathbf{x}_0 . It has one row for each residual, and one column for each unknown; each entry is a partial derivative of a residual term in the specification language

$$\mathbf{J} = \left[\frac{\partial \mathbf{F}}{\partial x_1}, \dots, \frac{\partial \mathbf{F}}{\partial x_n} \right]$$

A standard Preconditioned Conjugate Gradient (PCG) solver for this linear system will initialize by computing $\mathbf{J}^T \mathbf{F}$, and then perform repeated iterations of right-multiplying $\mathbf{J}^T \mathbf{J}$ by a data vector, \mathbf{p} ,

with as many elements as the unknowns. That is, it will repeatedly compute $\mathbf{J}^T \mathbf{J} \mathbf{p}$ for different values of \mathbf{p} .

The $\mathbf{J}^T \mathbf{F}$ and $\mathbf{J}^T \mathbf{J} \mathbf{p}$ computations are where much of the scheduling choice lies. The partial derivatives that make up the entries of \mathbf{J} often share intermediates and depend on the particular choice of energy function, much like the sparsity pattern of \mathbf{J} . Furthermore, the computation of each entry of $\mathbf{J}^T \mathbf{J} \mathbf{p}$ consists of the (mathematically commutative and associative) summations

$$\mathbf{J}^T \mathbf{J} \mathbf{p}_i = \sum_j \frac{\partial \mathbf{F}}{\partial x_i} \cdot \frac{\partial \mathbf{F}}{\partial x_j} \mathbf{p}_j = \sum_{r=1}^R \sum_j \frac{\partial \mathbf{F}_r}{\partial x_i} \cdot \frac{\partial \mathbf{F}_r}{\partial x_j} \mathbf{p}_j \quad (3)$$

which provide the opportunity for reorganization.

In this work we restrict ourselves to the Jacobi preconditioner for PCG, which we compute outside of the inner loop.

3.2 Code Reorganization

Code reorganization is a key part of extracting maximum performance in handwritten GPU solvers.

Although there is extensive previous work, no two handwritten solvers in the literature are structured identically. Each one is specialized for the particular energy it was written to solve, with local and global code reorganization done to exploit two avenues for increased performance:

- (1) Increase data-parallelism.
- (2) Tradeoff computation vs. materialization costs for faster linear solve iterations.

The first avenue is straightforward: GPUs are massively data-parallel and so high-performance solver code must be structured to take advantage of this. Code is reorganized such that as many lanes of computation as possible are active at once; this is done by parallelizing over as many dimensions as possible, but also by limiting synchronization stalls (by avoiding atomic collisions).

The second avenue is exploited by hoisting computation from the inner loop of the linear solve outside of the inner iterations. The system matrix ($\mathbf{J}^T \mathbf{J}$) remains constant for the linear solve, so any type of partial evaluation and materialization of its intermediates before the linear solve is valid.

We break the concept of materialization into a couple of related scheduling transforms: materialization of the Jacobian or its products, and fine-grained materialization of its intermediates.

There are three algebraically meaningful intermediates that can be materialized per residual template before computing $\mathbf{J}^T \mathbf{J} \mathbf{p}$:

$$\mathbf{J}, \quad \mathbf{J}^T \mathbf{J}, \quad \mathbf{J} \mathbf{p}$$

We adopt a simple notation to refer to this: surrounding a term with brackets means it is materialized before computation of $\mathbf{J}^T \mathbf{J} \mathbf{p}$. For example, an explicit computation that one might adopt if implementing a solver using library building blocks is $[(\mathbf{J})^T (\mathbf{J})] \mathbf{p}$: first materialize (write out to memory) the Jacobian, then use a matrix-matrix multiply to compute $\mathbf{J}^T \mathbf{J}$ and store that, then finally use a matrix-vector multiply to compute $\mathbf{J}^T \mathbf{J} \mathbf{p}$. This is the strategy adopted by Opt [Devito et al. 2017] for "materialized" solvers. On the other hand, most solvers generated by Opt are $\mathbf{J}^T \mathbf{J} \mathbf{p}$ (no brackets)

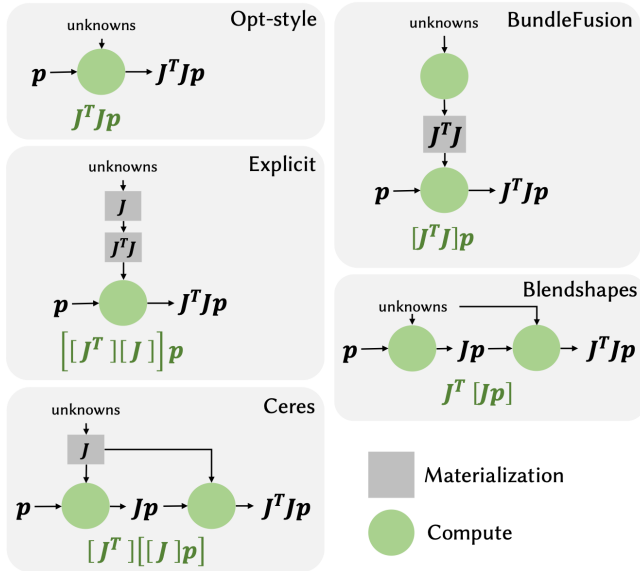


Fig. 3. There are 5 different ways of structuring just the coarse-grained $J^T J p$ evaluation of a residual group in a Levenberg-Marquardt solver, each corresponding to various existing solvers. The top-left is used in Opt [2017], bottom-left in Ceres [2010a], top-right for the dense term in BundleFusion [2017] and the mid-right for blendshape fitting [2016].

solvers; they are computed matrix-free, without ever materializing an intermediate matrix to memory.

We detail the possible combinations and permutations in Figure 3. Reviewing prior work, Wu et al. [2014] and Meka et al. [2016] use $J^T J p$, Zollhöfer et al. [2014], Zollhöfer et al. [2015], and Thies et al. [2016] use $J^T [J p]$, Innmann et al. [2016] use $[J^T][J] p$, and Dai et al. [2017] uses $[J^T J] p$ for the dense term and $J^T [J p]$ for the sparse term.

Each of the solver types have various strengths and weaknesses. For sparse problems, Opt-style matrix-free solves can end up requiring less bandwidth than reading from a materialized matrix. On the other hand, dense problems, or problems with many more residuals than unknowns may save both bandwidth and significant redundant compute by materializing parts of $J^T J p$ before the inner linear iterations.

In addition, intermediate expressions are often computed and stored before computing the full Jacobian. For example, the spherical harmonic shading term in Wu et al. [2014] and the transform matrices in Dai et al. [2017] are computed before the linear solve begins, and are used by the Jacobian product kernels.

4 DESIGNING THE SCHEDULING SPACE

Existing high-performance GPU solvers are structured in radically different ways in terms of how the computation of Jacobian terms are structured and split across data-parallel compute kernels.

We define the scheduling space as the space spanned by various scheduling transforms extracted from hand-crafted solvers. For every conceptual transform, we provide one or more corresponding Thallo scheduling constructs.

```

1 U,N = Dims("U", "N") -- Dimensions for Inputs and Residuals
2 in = Inputs {
3   B = Unknown(float2, {U}) -- Two unknowns
4   data = Array(float2, {N}) -- N (x,y) pairs
5 }
6
7 u,n = U(),N() -- Get Index Domains from Dimensions
8 b,x,y = in.B(u),in.data(n)(0),in.data(n)(1)
9
10 term = y - (b(0)*sin(b(1)*x) + b(1)*cos(b(0)*x))
11 -- Residuals are squared and summed over their Index Domains
12 r = Residuals {
13   fit = term
14 }
    
```

Fig. 4. "curvefit.t": simple Thallo example for curve fitting, implementing the example in Equation 2.

4.1 Energy Functions

In order to discuss scheduling with a concrete syntax, we introduce our energy syntax using the parametric curve fitting example from Equation 2 in Figure 4.

$$\arg \min_{b_0, b_1} \sum_{n=1}^N (y_n - (b_0 \cdot \sin(b_1 \cdot x_n) + b_1 \cdot \cos(b_0 \cdot x_n)))^2$$

Like Opt, we embed the energy specification language in Lua.

Line 10 is a direct transcription of the math inside the summation in Equation 2, and by assigning the result in the *Residual Block* on line 13, we declare that we are trying to find the argmin of that term, squared and summed along the dimensions of the Index Domains used in the expression, here the trivial dimension U of size one, and N , the number of points. Multiple expressions can be assigned to a single name, and multiple names can be used. Each name defines a *residual group*, which can be scheduled independently.

At the top of the specification, we list the problem dimensions ($U, N = \text{Dims}("U", "N")$).

These *abstract dimensions* are used both for specifying the dimensions of input arrays, as well as defining *index domains* for what we are mapping residuals over.

Lines 2-4 are the *Input Block*, where we define multidimensional arrays that we pass in via the C API. Line 7 creates index domain variables from the abstract dimensions, which are used to index into the input arrays on the following line.

The API and programming model are not the focus of the paper, but they resemble those of Opt [Devito et al. 2017], see the supplemental Language Details for a more complete description.

4.2 Intermediate Representation

The energy description creates an IR which is then acted upon by the scheduling constructions in Section 4.3. This IR represents the *Residual Block* of the objective, which is an associative array from names to residual groups. These residual groups are lists of expressions, each of which represents a residual mapped along the dimensions of the Index Domains used in constructing the expression. The expressions themselves are represented as Directed Acyclic Graphs (DAGs) of operators constructed from inputs, mathematical expressions, and Index Domains, which are used to access Arrays and Unknowns. Redundant expressions are shared, thus the IR is not composed of

trees and there is no need for an explicit common subexpression elimination pass.

The computation of a Jacobian product requires computing the partial derivatives of these expressions, mapped along their Index Domains. In order to compute these derivatives, we use the OnePass algorithm, a variant of forward-mode differentiation that was first used in the HLSL shading language compiler [Guenther et al. 2011].

The compiler generates an IR corresponding to a single residual group or output, depending on the kernel being generated, and then emits a GPU kernel that maps (in parallel) across the Index Domains used in the expressions, calling the derivative subroutine to construct the subexpressions it requires for any particular residual-unknown pair.

Materialized intermediate terms are handled similarly, with kernels generated mapping across the Index Domains for an expression to be materialized. These kernels are merged if they map over the same dimensions, which allows expressions to be shared.

The various scheduling constructs modify how the energy IR is transformed into kernels. Split and Merge change the groupings of expressions in the residual groups; for a set of N expressions that map over the same index domain, it's possible to construct schedules that compute the Jacobian terms corresponding to each of the expressions in N separate kernels that each is scheduled separately, or merge them into a single kernel which can share expressions. The Materialize construct on intermediates marks subgraphs to be materialized in their own kernels that are executed before the full Jacobian products. The rest of the constructs (Sum Parallelize, Compute at Output, Reorder, and Materialize of the Jacobian terms) directly annotate the residual group IR node with information consumed by the code generator on what Jacobian terms are to be computed and how to map over them.

4.3 Scheduling Constructs

4.3.1 Materialize. As discussed in Section 3.2, portions of the Jacobian can be computed and stored (materialized) once per non-linear iteration of the solver and then reused for multiple PCG iterations.

The algebraically relevant materialization choice can be made using the scheduling constructs `:materializeJ()`, `:materializeJp()`, and `:materializeJtJ()` on the residual terms. When materializing J or $J^T J$ we use either compressed sparse row (CSR) or a dense matrix. Specifically, we set this behavior by using `:set_sparseJ()` and `:set_sparseJtJ()`. We use cuBLAS and cuSPARSE² for post-materialization matrix-matrix or matrix-vector products.

Additionally, we can mark any intermediate expressions used in a residual for materialization (and their partial derivatives, i.e. their own Jacobians), using the `:materialize()` (and `:materializeJ()`) construct. These values will be computed and stored once per linear solve and used by either later materialization stages or the various $J^T J p$ kernels in the inner iterations. We can visualize these fine-grained materialization choices as cuts in the expression DAG of the residuals, as we do in Figure 5.

4.3.2 Compute At Output. There is a natural data-parallel strategy for materializing any term involving a Jacobian: parallelize along the Index Domains of the residuals within a group. There are in

²<https://developer.nvidia.com/cublas> and <https://developer.nvidia.com/cusparse>

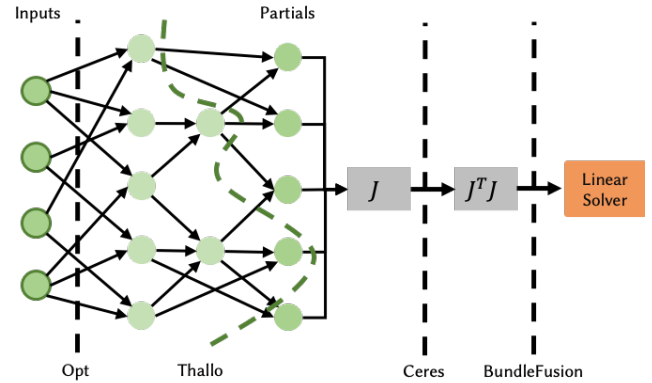


Fig. 5. Thallo is highly flexible and allows materialization of intermediate terms at many points. This is key to exploring tailored solver choices and rapid development while focusing on run-time performance. Here we visualize the expression DAG representing the computation of the Jacobian. Since Thallo controls the compilation process, it is able to materialize intermediate terms before the full computation of the Jacobian or its products.

We show a materialization choice unavailable to alternate solvers or solver frameworks as a green cut in the expression DAG; expressions at nodes immediately above the cut are materialized.

general multiple outputs per residual when computing $J^T J p$ or $J^T J$, requiring an aggregation scheme. We generally rely on fast atomic operations, which can introduce atomic contention and increase write bandwidth load. Wu et al. [2014] and Meka et al. [2016] improve performance significantly by computing the terms "at output", i.e. they compute a single element of $J^T J p$ or $J^T F$ per lane of computation. This involves a non-trivial code transform that inverts the mapping from residuals to unknowns and inserts boundary condition checking; this transform is implemented for "Stencil Residuals" in Opt [Devito et al. 2017], see Figure 6.

This transform increases redundant computation and bounds checking and may decrease parallelism, but decreases write bandwidth load and atomic contention. Additionally, when the residual uses Sparse constructs, this can require an explicit inverse mapping data structure (we use a compressed sparse row analogue). Since this transform has competing effects on performance, Thallo exposes it as a scheduling transform, on both $J^T J p$ and $J^T F$ as `compute_at_output()`.

4.3.3 Sum Parallelize. When multiple partial derivatives of a given residual have identical form (but different data), it's possible to exploit extra data parallelism when computing the Jacobian or Jacobian products, and Thies et al. [2015] does just that. Instead of parallelizing over residuals, they parallelize over both residuals and unknowns, materializing the partial for a given unknown-residual pair on each lane of computation. This extra data-parallelism arises naturally when a residual sums over a dimension corresponding to an unknown read; the partial for a given unknown then corresponds to the derivative of a single term of the sum. This is done in our Blendshape Fitting example, see Figure 7.

The energy language for Thallo contains a Sum construct that sums an expression over one or more dimensions, which allows users to

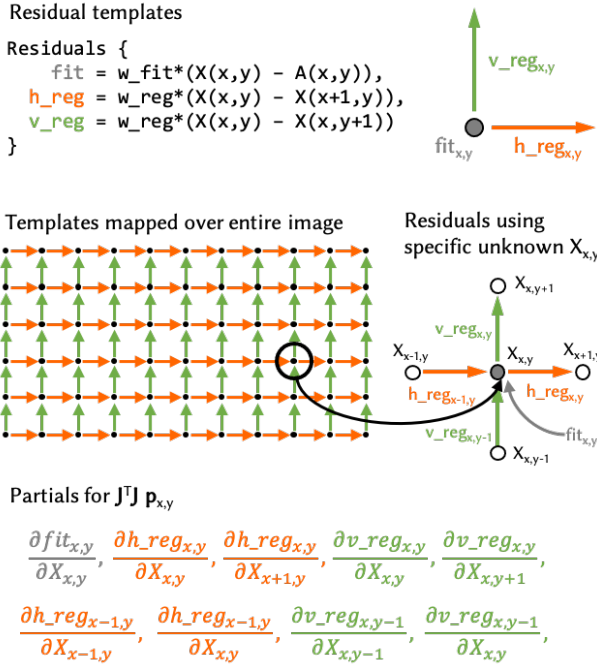


Fig. 6. In order to compute a single entry of $J^T J p$, the compiler needs to invert the mapping from residuals to unknowns. This corresponds to computing the inverse of the stencil for stencil residuals. This transform was done automatically by Opt, we expose it as a scheduling transform. This figure is adapted (with permission) from Fig. 9 in [Devito et al. 2017]

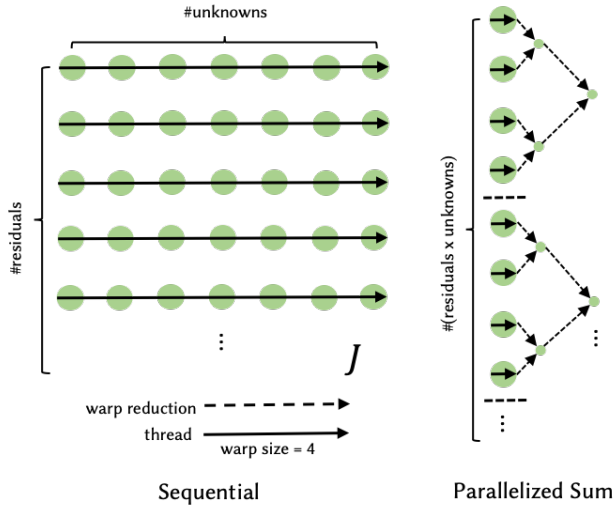


Fig. 7. For associative and commutative reductions (namely `Sum()`), we can expose significantly more parallelism by parallelizing over both unknowns and residuals instead of merely residuals. This is done, for example, in the Blendshape solvers we compare against.

express ubiquitous concepts such as dot product or convolution in our energy language. We can demonstrate this using a simple 5×5 deconvolution example:

```

W, H, K = Dims("W", "H", "K") -- K = 5
in = Inputs {
  X = Unknown(float, {W,H}) -- Image
  Target = Array(float, {W,H})
  kernel = Array(float, {K,K}) -- Convolution Kernel
}
x,y,k_0,k_1 = W(), H(), K(), K()
kernel_weight = in.kernel(k_0,k_1)
pixel = in.X(x-k_0+2, y-k_1+2)
convolved = Sum({k_0,k_1}, kernel_weight*pixel)
Residuals = { conv = in.Target(x,y) - convolved}

```

This energy specification sets up an energy that is minimized when the unknown image (X) convolved with `kernel` produces the `Target` image. The partial derivatives for a given residual differ only in the index of the kernel weight they read, and so can be computed using a data-parallel kernel. We expose this in Thallo using the `:parallelize()` construct, which can be specified on any `Sum()` term (in this example, `convolved`).

4.3.4 Split/Merge. Residual contributions to a particular term are separable due to the additive commutativity in Equation 3. This is used in Dai et al. [2017] to have completely separate schedules for the sparse and dense terms of the optimization problem. We use Residual Groups and `split/merge` constructs to capture this separability. Residual Groups that are defined over the same Index Domains can be merged together or split apart. When executing each Jacobian product, all of the expressions in a single residual group are executed together, which allows for term sharing (and thus common subexpression elimination). In Thallo, each separate Residual Group corresponds to a separate GPU kernel launch for every Jacobian product.

4.3.5 Reorder. The prior scheduling transforms can be viewed as constructing a set of parallelizable loop nests, corresponding to partial evaluation of the Jacobian and various Jacobian product terms. The iterator order for these loop nests in handwritten code was chosen by the implementors. This particular transform is important for scheduling reductions on the GPU, as iterator order can determine whether it is possible to do fast local reductions and decrease global memory bandwidth, see Figure 8.

In future work, we envision researchers could write a backend that lowers these loop nests to Halide pipelines, which would immediately suggest a large set of scheduling transforms that were not used in prior work. For the initial implementation of Thallo, we chose to only implement a transform that transposes the loops, i.e. the `reorder()` construct.

4.4 Complete Schedule Example

We provide an example of a schedule using multiple scheduling constructs in Figure 9. In total, this schedule accelerates the solver by multiple orders of magnitude. The final three lines specify the schedule: `reorder` computation to allow for coherent reductions, force the matrix representation of the camera transformation to be materialized before $J^T J$ computation, and then materialize the full $J^T J$ matrix before the inner iterations of the solver, respectively.

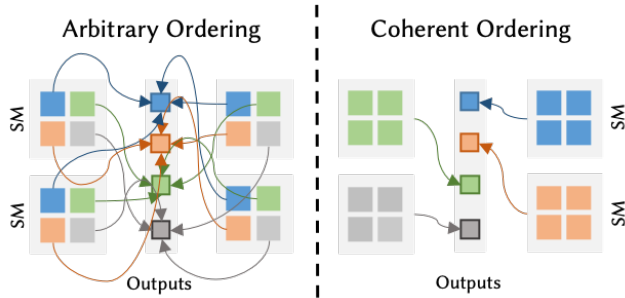


Fig. 8. By reordering the iteration over residuals, we can choose an iteration order that enables us to use warp reductions (SIMD lane shuffles) that bypass expensive memory hierarchies when associatively reducing into unknown-indexed vectors or matrices. In this image, by a simple index reordering (and adding simple warp reduction code), write bandwidth and atomic contention is reduced by a factor of 4 as each separate Streaming Multiprocessor (SM) only issues reductions to one output (in the best case this can be improved by a factor of the SIMD width, 32 on all existing NVIDIA GPUs).

```

1 Pairs,T,W,H = Dims("Pairs","T","W","H")
2 Inputs {
3   CamTranslation = Unknown(float3,{T},0),
4   CamEulerAngles = Unknown(float3,{T},1),
5   Positions      = Array(float4,{W,H,T},2),
6   Normals        = Array(float4,{W,H,T},3),
7   intrinsics     = Param(float4,4),
8   imageDim       = Param(float2,5),
9   TargetT        = Sparse({Pairs},{T},6),
10  SourceT         = Sparse({Pairs},{T},7)
11 }
12
13 t,w,h,p = T(),W(),H(),Pairs()
14 t_s,t_t = SourceT(p),TargetT(p)
15 posSrc, normalSrc = Positions(w,h,t_s), Normals(w,h,t_s)
16
17 xformT = convert_to_matrix(CamRotation(t), CamTranslation(t))
18 srcToTgtXform = matmul(xformT:get(t_t),invert(xformT:get(t_s)))
19
20 normalSrcInTgt = vec3(gemv(srcToTgtXform,normalSrc))
21 posSrcInTgt = rigid_xform(srcToTgtXform,posSrc)
22 tgtScreenPos = camera_to_depth(intrinsics,Constant(posSrcInTgt))
23 inScreen     = rect_contains_point(imageDim,tgtScreenPos)
24 posTgt       = vec3(bilinear_sample(Positions,tgtScreenPos,t_t))
25 pointToPlane = dot(posTgt - posSrcToTgt,normalSrcInTgt)
26
27 r = Residuals {
28   dense = select(inScreen,pointToPlane,0.0)
29 }
30
31 r.dense:reorder({w,h,p})
32 xformT:set_materialize(true)
33 r.dense.JtJ:set_materialize(true)

```

Fig. 9. A simplified dense-only implementation of BundleFusion [Dai et al. 2017]. The final 3 lines specify a high-performance schedule.

5 AUTOSCHEDULER

The amount of scheduling choices for moderately complicated energy specifications can be overwhelming. In Figure 10, we show the distribution of convergence time on thousands of solvers for BundleFusion, generated with different schedules. In order for non-domain experts to achieve high performance without reasoning about the

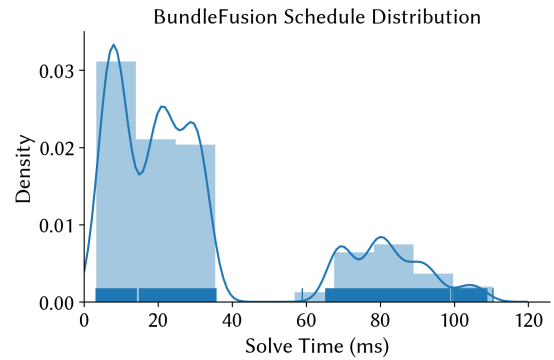


Fig. 10. Distribution of convergence times for solvers generated with various schedules for the BundleFusion example. A randomly selected schedule would perform much worse than a well-chosen one.

scheduling transforms, we developed a heuristic rule-based auto-scheduler that mimics a basic strategy that domain experts use to choose performant schedules for their handwritten solvers. This autoscheduler is controlled by a single configuration flag. Our auto-scheduler executes very quickly (under a millisecond) and requires no user guidance. We empirically demonstrate that solvers generated by our autoscheduler achieve better performance than previous work on all problems detailed in Section 6, despite there being no formal performance guarantees.

We use a cost model that evaluates the total time to perform one nonlinear iteration for a given scheduled solver. We found that taking into account a few cost features provides efficient schedules. Moreover, it makes the cost model interpretable by a human, but elides many details of the underlying hardware that could be used for more accurate comparisons. Alternative cost models, such as Hong and Kim [2009] or the learned cost model for the latest Halide autoscheduler Adams et al. [2019] could be incorporated into Thallo and are left to future work.

We build our cost model on the following features:

- Total Memory
- Memory Accesses
- Compute Operations
- Atomic Collisions

For this cost model, we need to query a few properties of the GPU: the theoretical peak FLOPs (t_c), the peak bandwidth (t_m), the maximum available memory (M), the number of threads in a warp W , and the minimum size of a global memory transaction g_m .

For a given schedule S , concrete dimensions, and an average number of expected linear iterations per nonlinear iterations (ℓ), we can compute a featurization of the schedule.

First, we can separate out the generated compute kernels for S into two disjoint sets $K_N = \{k_{N_i}\}$ for kernels executed once per nonlinear iteration and $K_L = \{k_{L_i}\}$ for kernels executed each linear iteration. For a given kernel k , We can then define a cost C for each kernel k as

$$C(k) = t_k \max \left(\frac{m_k}{t_m}, \frac{o_k}{t_c} \right),$$

where o_k is the number of arithmetic operations in kernel k , t_k is the number of threads launched by kernel k , m_k is the amount of memory accessed by one thread in the kernel, where a memory access is rounded up to g_m if adjacent threads do not access adjacent locations in memory and atomic writes are only counted once per warp if the iteration order (specified by `reorder()`) allows for warp reductions before the atomic write.

Note that the two sides of the max correspond to the bandwidth-bound and compute-bound regimes in the roofline model ([Williams et al. 2009]).

We also compute the amount of global memory necessary to store all inputs/outputs and intermediates ($M(S)$).

We then compute a cost for S as

$$C(S) = \begin{cases} \infty & M(S) > M \\ \left(\sum_{k_{N_i} \in K_N} C(k_{N_i}) \right) + \ell \left(\sum_{k_{L_i} \in K_L} C(k_{L_i}) \right) & M(S) \leq M \end{cases}$$

Our cost model is intentionally simple and elides many architectural features, see the supplemental material for an examination of its predictiveness on our application suite.

The autoscheduler uses the following rules-cascade as a greedy scheme to find a low-cost schedule:

1. The autoscheduler aggressively merges all residual groups that are mapped over the same Index Domains (this monotonically reduces the cost in the cost model if there are any shared terms, otherwise it is cost-neutral).

2. We use breadth-first search on the expression DAG for a residual group starting from array accesses, materializing whenever reading an expression and its Jacobian would lower cost compared to computing the original expression (including the new materialization kernel in the cost calculation).

3. For the merged groups, we choose the high-level $\mathbf{J}^T \mathbf{J} \mathbf{p}$ materialization strategy by evaluating the costs over the five combinations.

4. If the schedule for a group is $\mathbf{J}^T \mathbf{J} \mathbf{p}$ and the residuals are mapped over the dimensions of the unknowns, we map over elements of the output using the `compute_at_output()` transform (this does not require a potentially expensive explicit inverse map, and lowers the number of required memory writes).

5. We sort the indexing order of the residuals such that any Index Domain used in the residual that is not used to access an unknown is brought to the front (so it is the innermost iterator); this is a mechanical process that, combined with coherent reduction code emitted by our compiler, drastically improves performance, see Figure 8.

6 RESULTS

In the following, we demonstrate the benefits of the scheduling constructs in Thallo on a large variety of problems in imaging, vision, and graphics. For extensive descriptions of the problems and details of validation of solvers generated by Thallo on the NIST StrD NLLS dataset³, see the supplementary material.

³https://www.itl.nist.gov/div898/strd/nls/nls_main.shtml

6.1 Applications

We demonstrate the use of Thallo on six least-squares optimization problems in visual computing:

Bundle Adjustment. Bundle Adjustment is at the core of every structure-from-motion framework (SfM). It is used estimate accurate and globally-consistent camera parameters alongside a sparse 3D reconstruction [Agarwal et al. 2011; Jebara et al. 1999; Schönberger and Frahm 2016; Triggs et al. 2000] from a set RGB images. We implement standard Bundle Adjustment, formulated as in Bundle Adjustment in the Large [Agarwal et al. 2010b]. In this section we evaluate the solvers on the second largest problem in the BAL dataset. The optimization is posed over 4585 cameras, each of which has 9 unknown parameters. There are 1324582 correspondences, and twice as many residual terms. See the supplement for a much more comprehensive comparison on all of the BAL problems, with various error thresholds.

BundleFusion. BundleFusion [Dai et al. 2017] formulates the analog problem to Bundle Adjustment for the RGB-D case with known depth. It uses both sparse image correspondences and dense depth maps to achieve highly-accurate loop closure. For a sparse set of inter-frame correspondences C and a set of N images, a global alignment energy is optimized to find the best rigid camera transformations \mathbf{T}_i (using 6 DoF parameterizations with Euler vectors for the rotations) such that the total point-to-point error of a sparse set of detected and matched feature points is minimized

$$\mathbf{T}_i^* = \arg \min_{\mathbf{T}_i} \sum_{i=1}^N \sum_{j=1}^N \sum_{(l,k) \in C(i,j)} \|\mathbf{T}_i \mathbf{c}_i^k - \mathbf{T}_j \mathbf{c}_j^l\|_2^2. \quad (4)$$

We use the sparse term of BundleFusion and the combined sparse and dense energy as separate example problems. For the evaluated dataset, the sparse-only problem has 2000 residuals, while the full problem has > 265000.

Shape-and-Shading. We modify the shape-from-shading problem evaluated in Opt [Devito et al. 2017] to solve not only for refined depth from the output of an RGB-D sensor, but also jointly for the spherical harmonic (SH) lighting conditions, which were fixed in a preprocessing step in the original work by Wu et al. [2014]. We add an explicit lighting residual fitting term as well as modifying the gradient shading terms. This results in a significantly different Jacobian structure, as the 9 SH terms are in every pixel's lighting and shading residuals.

Face Template Fitting. Non-rigid model-based registration is a fundamental building block for a large variety of tracking and reconstruction approaches [Loper et al. 2015; Romero et al. 2017; Thies et al. 2016; Xu et al. 2018]. One prominent example is fitting an affine parametric 3D mesh model \mathcal{P} to 2D observations, e.g., as done in Face2Face [Thies et al. 2016]. For face tracking, the affine parametric model is then defined by a low-dimensional expression subspace of M blendshapes spanned by a matrix $\mathbf{B} \in \mathbb{R}^{3N \times M}$ that models the facial expressions relative to a neutral face template $\mathbf{a} \in \mathbb{R}^{3N}$

$$\mathbf{m} = \mathcal{P}(\mathbf{c}) = \mathbf{a} + \mathbf{B} \mathbf{c}.$$

Each of the M blendshapes represents a different facial expression as per-vertex offsets to the neutral face template and new faces $\mathbf{m} \in \mathbb{R}^{3N}$ are created by interpolation of these M displacement vectors based on the M blendshape weights $\mathbf{c} \in \mathbb{R}^M$. We fit the model to 2D data by finding the best coefficients \mathbf{c}^* , such that $\Theta(\mathcal{P}(\mathbf{c}^*))$ best matches a set of target correspondences $\mathbf{t} \in \mathbb{R}^{2N}$, where Θ is a projective camera transformation (using the 9-dimensional parameterization from BAL [Agarwal et al. 2010b])

$$\mathbf{c}^* = \arg \min_{\mathbf{c}} \|\mathbf{t} - \Theta(\mathcal{P}(\mathbf{c}))\|_2^2. \quad (5)$$

Poisson Deconvolution. We developed a pipeline for using Thallo on a general class of proximal optimization problems [Parikh and Boyd 2013] for imaging applications, scheduling the sub-problems of the ProxImaL optimization compiler [Heide et al. 2016]. As a concrete example, we adopt the Poisson Deconvolution optimization problem from [Heide et al. 2016] with identical regularization weights on the total-variation term.

Spatially-Varying Deconvolution. While existing methods for efficient spatially-varying deconvolution require the blur to be local point-spread-functions (PSFs), i.e., stencil-operations that are at least spatially-invariant in a local neighborhood, Thallo lets us formulate *per-pixel* PSFs with arbitrary support over the image plane. We evaluate this more complicated example as well.

6.2 Run-time Performance Evaluation

For all the examples mentioned in Sec. 6.1, we compare Thallo against state-of-the-art implementations and high-level languages. Our first comparison is against the flexible and mature CPU-based Ceres library [Agarwal et al. 2010a]. Since Ceres is a CPU library, and the other comparisons use (far more parallel) GPUs, this is not an apples-to-apples comparison, and it is not surprising that the GPU-based solvers converge faster in wall-clock time.

Our second point of comparison is the state-of-the-art GPU optimization DSL Opt [Devito et al. 2017], which has a similar programming model to ours sans the scheduling constructs that allow Thallo to generate efficient solvers on non-local problems.

Where available, we also compare against existing handwritten GPU-based solvers, such as the original open-sourced BundleFusion solver [Dai et al. 2017].

We run all our experiments on a Linux machine with an NVIDIA TITAN X (Pascal) and an Intel Core i7-6700K Processor, which has 4 physical and 8 virtual cores. For the autoscheduler evaluation across architectures we also use NVIDIA Jetson Nano Devkit and a Google Compute Engine instance with a NVIDIA Tesla K80 GPU a 2 vCPUs on an Intel Broadwell processor.

6.2.1 Performance Comparisons on Non-Local Problems. Table 1 shows the summary of the run-time results; comparing solvers generated by our system both with no scheduling transforms and with autoscheduling to solvers generated with the Opt DSL. All of these solvers use single-precision arithmetic for highest performance. We additionally compare to Google’s Ceres Solver, which, unlike the other solvers, does not use the GPU. It is a mature library that can use a variety of optimization methods, including solving unconstrained optimization. We use Ceres’ Levenberg-Marquardt implementation,

and use the same hyperparameters for Thallo, Opt, and Ceres. Ceres has multiple choices for inner linear solver (in contrast to Thallo and Opt, which are currently limited to preconditioned conjugate gradient), and so we must choose among the options for a comparison. We chose the inner linear solver for Ceres based on published performance advice⁴ and manual search per-problem in order to minimize convergence time, always including the PCG solver in the search. Since Ceres operates in double precision, it often completes in fewer iterations than the other solvers (while requiring more memory traffic for each iteration). We report convergence time in this section, and provide iteration count breakdowns in the supplement.

Comparing the bottom two rows, it becomes clear that being able to make structural changes via the schedule allows for significant performance improvements. Our autoscheduled solvers consistently outperform solvers with no scheduling transformations.

Our autoscheduled solvers unsurprisingly outperform the CPU-based Ceres across the board, due to a combination of taking advantage of the massive parallelism of the GPU hardware and our improved scheduling. Similarly, we outperform every solver generated by Opt on these problems, as we are able to schedule the problems to take greater advantage of the underlying hardware. We perform on par or better than every handwritten solver, by virtue of being able to express the same schedules or explore better ones.

6.2.2 Performance Comparison on Problems with Small Stencil Structure. In addition to the non-local problems above, we use Thallo to solve ten energy formulations with small stencil structure, where each residual is computed from a small local neighborhood of unknowns, which results in a predictable and very sparse Jacobian. Since these problems have schedules which can be realized by matrix-free stencil constructs, they are efficiently addressed by previous work.

In Table 2, we show comparisons against state-of-the-art baselines realizing the same problems. Although the Opt DSL by DeVito et al. [2017] was centered around these specific problems, we achieve comparable or better run-time performance. This is an expected result: our schedules subsume the limited scheduling choices available in Opt.

6.2.3 Performance Scaling with Problem Size. In Fig. 11, we evaluate the performance of solvers generated with Thallo versus other systems for Sparse BundleFusion as we increase the problem size. Our scheduling gives us a slight edge over the handcrafted CUDA solver of [Dai et al. 2017], despite the latter benefiting from the drastically simplified matrix exponential derivatives.

Our solvers are roughly twice as fast as those generated by Opt throughout the range, and are an order of magnitude faster than solvers built with Ceres at the highest unknown counts, with the performance gap widening rapidly as we saturate the more parallel hardware.

Also note that our autoscheduler changes the schedule based on problem size, which gives it the performance edge against our (single) manual schedule at low unknown counts, where materializing $\mathbf{J}^T \mathbf{J}$ is advantageous. An added benefit of a high-level system

⁴http://ceres-solver.org/solving_faqs.html

Solver	Optimization Problem						
	BundleFusion [2017]	Face Fit [2016]	Shape/Shading	Bundle Adjust [2000]	Deconv. (11 × 11)	Deconv. (15 × 15)	SV Deconv.
Ceres [2010a]	5394.10ms	25612.1ms	21982.3ms	11954.56ms	25304.85ms	55158.87ms	81344.37ms
Opt DSL [2017]	110.39ms	1057.8ms	2259.6ms	114.78ms	195.93ms	∞	∞
Hand-written CUDA	5.57ms	1946.5ms	N/A	N/A	N/A	N/A	N/A
Ours (Unscheduled)	106.3ms	1013.1ms	2235.6ms	105.81ms	187.73ms	253.39ms	452.16ms
Ours (Auto)	2.96ms	46.3ms	171.86ms	89.47ms	46.58ms	93.11ms	137.92ms

Table 1. Runtimes of different solvers on various benchmarks. We report time-to-convergence for the problems described in Section 6.1. From top to bottom, we have solvers generated by the Ceres library [Agarwal et al. 2010a], solvers generated by Opt [Devito et al. 2017], handwritten expert CUDA solvers (using hand-coded derivatives) if existent, solvers generated by Thallo with no scheduling annotations (and no autoscheduling), and solvers generated by Thallo with autoscheduling. The fastest implementation for each problem is bolded. Note that Ceres is a flexible high-level library that executes on the CPU, while the other systems use the GPU. Also, Opt fails to compile for 15 × 15 Deconvolution and Spatially-Varying Deconvolution.

Solvers	Optimization Problem									
	ARAP	COT	EMD	IW	IID	OF	PIE	RNA	SFS	VMD
Ceres [2010a]	74339.97ms	N/A	N/A	132.58ms	N/A	N/A	75404.39ms	N/A	75404.39ms	187.96ms
Opt DSL [2017]	5201.91ms	118.21ms	35.40ms	138.09ms	19.59ms	7.14ms	29.98ms	36.34ms	88.99ms	62.65ms
Hand-written	5355.39ms	N/A	N/A	260.18ms	N/A	N/A	32.43ms	N/A	106.64ms	65.70ms
Ours (Unscheduled)	5122.35ms	83.16ms	29.13ms	317.76ms	31.58ms	6.77ms	28.49ms	34.61ms	456.74ms	37.51ms
Ours (Auto)	5122.35ms	42.11ms	29.13ms	128.65ms	19.11ms	6.77ms	28.49ms	34.61ms	84.13ms	38.01ms

Table 2. Performance comparison on problems with small stencil structure that were the focus of current state-of-the-art DSLs [Devito et al. 2017]. From left to right: as-rigid-as-possible mesh deformation [Sorkine and Alexa 2007] (ARAP), cotangent mesh smoothing [Desbrun et al. 1999] (COT), embedded mesh deformation [Sumner et al. 2007] (EMD), image warping [Dvoroznak 2014] (IW), intrinsic image decomposition [Meka et al. 2016] (IID), optical flow [Horn and Schunck 1981] (OF), Poisson image editing [Pérez et al. 2003] (PIE), robust non-rigid alignment [Zollhöfer et al. 2014] (RNA), shape-from-shading [Wu et al. 2014] (SFS), volumetric mesh deformation [Innmann et al. 2016] (VMD). As expected Thallo is able to match or outperform Opt, despite the latter being designed specifically for these stencil applications, since the scheduling choices in Thallo subsume Opt’s limited scheduling choices.

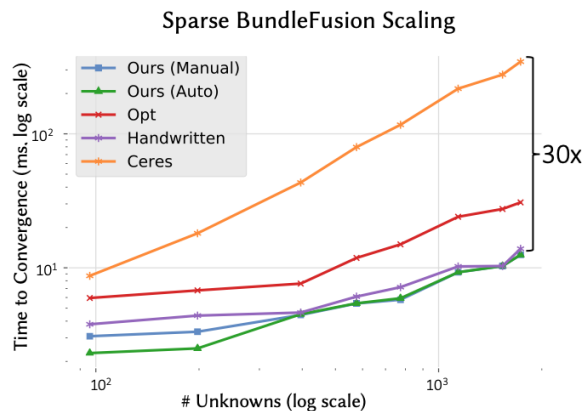


Fig. 11. Time to convergence of various solvers on Sparse BundleFusion as we increase vary the number of unknowns, charted on a log-log plot. We configure Thallo and Opt to generate Gauss-Newton solvers to match the published handwritten solver [Dai et al. 2017] with its preconfigured iteration counts; and configure the Ceres solver to stop once it lowers the cost to within 1% of the converged handwritten solution. Our autoscheduler changes the schedule based on problem size, which gives it the performance edge against our manual schedule at low unknown counts, where materializing $J^T J$ before the linear solve is advantageous.

like Thallo is that it is trivial to generate different solvers tailored to different data distributions.

6.3 Schedule Space Exploration

Our scheduling primitives define a rich space of schedules for any given problem.

6.3.1 Scheduling Case Study. We use Dense+Sparse BundleFusion as a case study of the power of Thallo scheduling constructs, in Table 3. We start with a default schedule and transform the schedule step-by-step into a high-performance schedule, in the process making the resulting schedule significantly faster.

Schedule	Time (in ms)
<no scheduling annotations>	106.34ms
r.dense:reorder{w,h,t}	41.32ms
xformT:materialize(true)	16.23ms
r.dense.JtJ:materialize(true)	4.35ms
r.sparse.JtJ:materialize(true)	2.96ms

Table 3. A breakdown of the BundleFusion problem as we add scheduling constructs to the schedule. The final schedule is over 35× faster than the original schedule.

6.3.2 Quantitative Schedule Space Search. In order to quantify the performance space our scheduling primitives allow users to explore, we exhaustively explored the combinatorial scheduling choices on two problems of different complexities and generated performance distributions, seen in Figure 12. This exhaustive search is effectively

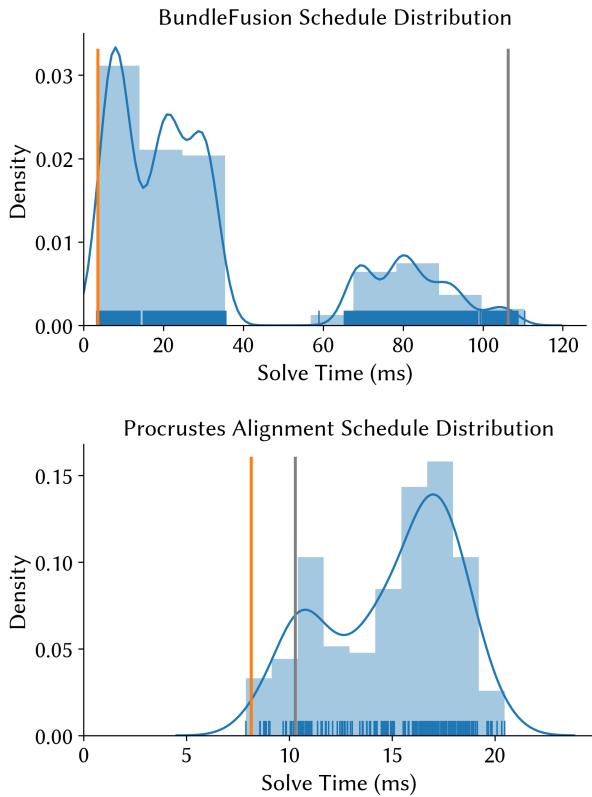


Fig. 12. Distribution of the performance of schedules expressible in our system. The orange line marks the schedule selected by our autoscheduler, the grey line marks the schedule corresponding to Opt [Devito et al. 2017], and the blue ticks represent individual data points. This data is summarized by both a histogram and kernel-density estimated distribution. The top graph shows that on the relatively simple Procrustes Alignment problem, our autoscheduler chooses a decent schedule, only 15% off the optimum, outperforming the Opt schedule slightly and outperforming the median schedule by about a factor of 2. For BundleFusion, our autoscheduler actually chooses the best schedule available, outperforming most other schedules by an order of magnitude, outperforming the Opt schedule by a factor of 35, and almost doubling the performance of the published handwritten solver. The performance distribution spans a very large range.

an autotuning process, however it is impractical for algorithm development that requires rapid prototyping. For example, due to the several second overhead of CUDA compilation, it took more than 48 hours to generate the distribution for BundleFusion.

These results suggest both that our scheduling primitives define a rich scheduling space, and that our heuristic autoscheduler effectively finds good schedules within that space.

6.4 Autoscheduler Evaluation

Using the exhaustive search method from Section 6.3.2, we evaluated our autoscheduler’s performance versus the best schedule possible. The results can be seen in Table 4. Across the suite of problems, our autoscheduler picks a schedule within 30% of optimal, with the majority of schedules within 10% of optimal.

Example	Best Time (ms)	Autoscheduled Time (ms)	% Off Optimum
BundleFusion	2.96	2.96	0.00%
Projective Face Fit	46.26	47.03	1.67%
Shape and Shading*	171.16	171.86	0.41%
Bundle Adjust	83.47	89.47	7.19%
Deconv. (11 × 11)	44.39	46.58	4.93%
Deconv. (15 × 15)	88.63	93.11	5.05%
SV Deconv.	137.92	137.92	0.00%
ARAP	5041.4	5122.35	1.61%
COT	32.44	42.11	29.81%
EMD*	26.83	29.13	8.57%
IW*	128.65	128.65	0.00%
IID*	19.11	19.11	0.00%
OF*	6.36	6.77	6.45%
PIE	27.81	28.49	2.45%
RNA	33.65	34.61	2.85%
SFS*	79.22	84.13	6.20%
VMD*	30.1	38.01	26.28%

Table 4. Comparison between solver convergence time of the best possible schedule, and the schedule selected by our autoscheduler. The rightmost column is the relative increase in solver time from using the autoscheduled solver versus the best possible one. Examples for which the exhaustive search did not complete within a day are marked with *; for these we modified the exhaustive schedule search to exclude the *split()* scheduling primitive, which shrunk the space enough to be tractable.

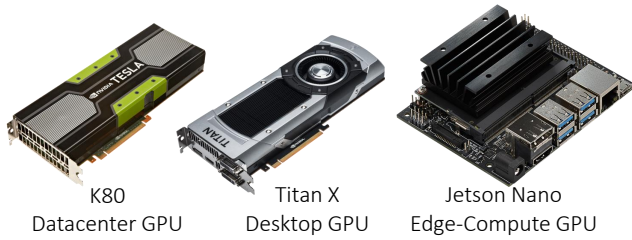
We provide a more thorough evaluation of the cost model used by our autoscheduler in the supplemental material.

6.4.1 Autoscheduler on Different GPU Architectures. We repeated the autoscheduler evaluation on two other GPU platforms of varying performance: an NVIDIA Jetson Nano Devkit which has a low-power Maxwell GPU and a NVIDIA Tesla K80 GPU used in data centers, which has about 40× more compute cores. The results from all 3 platforms can be seen in Table 5. Though absolute performance drastically differs between platforms, the autoscheduler still picked a schedule within 40% of optimal in all cases. This experiment validate the efficacy of the proposed scheduling scheme across platforms and for a wide set of problem instances in visual computing.

7 FUTURE WORK

It is easy to use Thallo in a larger system thanks to its thin shading-language-like API; however, the interoperability between other DSLs such as Ebb, Simit, Halide, or deep learning frameworks is at the coarse granularity of passing data between them. It would be beneficial to jointly compile constructs between high-level languages and thus allow for global scheduling options; this remains an open problem.

Thallo is currently limited to generating Levenberg-Marquardt and Gauss-Newton solvers for unconstrained nonlinear least squares optimization. It is also currently limited to Jacobi preconditioning for compatibility with matrix-free scheduling. Future work could extend the solver types or add support for constraints. Ideally, there



	% Off Optimum		
Example	TITAN X	K80	Nano
BundleFusion	0.00%	1.93%	1.19%
Projective Face Fit	1.67%	0.75%	4.04%
Shape and Shading	15.38%	4.46%	20.72%
Bundle Adjust	7.19%	8.30%	10.19%
Deconv. (11 × 11)	4.93%	5.74%	1.52%
Deconv. (15 × 15)	3.93%	6.95%	1.73%
SV Deconv.	0.00%	0.45%	1.81%
COT	29.81%	1.09%	2.99%

Table 5. Relative increase in solver time from using the autoscheduled solver versus the best possible one on three different platforms. All examples from Table 1 are included, along with the worst performing example from Table 4.

could be a Matlab-like language for writing general solvers where the resulting solvers could perform on par with Thallo's. Thallo could also be extended to work on other backends, such as multicore CPUs or FPGAs.

Promising areas for immediate follow-up work include extending the range of scheduling options by integrating sparsity encoding such as in [Kjolstad et al. 2017] or using more alternative approaches to the autoscheduler, such as the learned cost model in [Adams et al. 2019].

8 CONCLUSIONS

We introduced a scheduling space that encompasses code reorganization choices that span the literature for fast GPU nonlinear least squares solvers in graphics and vision. We proposed Thallo, a domain-specific language that enables rapid exploration of this scheduling space using an energy specification and scheduling specification language design. We introduce a compiler that takes a user's problem description and automatically generates a highly-efficient solver by exploiting problem structure. By default, our new autoscheduler generates tailored schedules based on this structure. In a series of experiments, we have validated the proposed framework on a variety of local and non-local optimization problems in graphics, vision and imaging. Thallo achieves state-of-the-art results on these problems: our run-time is comparable or even outperform hand-written implementations – in a few seconds of automatic code generation in contrast to hours of tedious hand-written implementation. Overall, we believe that Thallo will help to make high performance rapid prototyping much more accessible for practitioners in graphics and vision who want to develop new energy formulations.

REFERENCES

- Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). <http://tensorflow.org/> Software available from tensorflow.org.
- Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. DOI: <http://dx.doi.org/10.1145/3306346.3322967>
- Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M. Seitz, and Richard Szeliski. 2011. Building Rome in a Day. *Commun. ACM* 54, 10 (Oct. 2011), 105–112. DOI: <http://dx.doi.org/10.1145/2001269.2001293>
- Sameer Agarwal, Keir Mierle, and Others. 2010a. Ceres Solver. <http://ceres-solver.org>. (2010).
- Sameer Agarwal, Noah Snavely, Steven M. Seitz, and Richard Szeliski. 2010b. Bundle Adjustment in the Large. In *Proceedings of the 11th European Conference on Computer Vision: Part II (ECCV'10)*. Springer-Verlag, Berlin, Heidelberg, 29–42. <http://dl.acm.org/citation.cfm?id=1888028.1888032>
- Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2, Article 21 (May 2016), 12 pages. DOI: <http://dx.doi.org/10.1145/2892632>
- Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press, New York, NY, USA.
- Angela Dai, Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Christian Theobalt. 2017. BundleFusion: Real-Time Globally Consistent 3D Reconstruction Using On-the-Fly Surface Reintegration. *ACM Trans. Graph.* 36, 3, Article 76a (May 2017). DOI: <http://dx.doi.org/10.1145/3054739>
- Frank Dellaert. 2012. Overview In this document I provide a hands-on introduction to both factor graphs and GTSAM. (2012).
- Teng Deng, Jianmin Zheng, Jianfei Cai, and Tat-Jen Cham. 2018. SubdSH: Subdivision-based Spherical Harmonics Field for Real-time Shading-based Refinement under Challenging Unknown Illumination. In *2018 IEEE Visual Communications and Image Processing (VCIP)*. IEEE, 1–4.
- Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. 1999. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow (*SIGGRAPH '99*). New York.
- Zachary Devito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. 2017. Opt: A Domain Specific Language for Non-Linear Least Squares Optimization in Graphics and Imaging. *ACM Trans. Graph.* 36, 5, Article 171 (Oct. 2017), 27 pages. DOI: <http://dx.doi.org/10.1145/3132188>
- Michael Driscoll, Benjamin Brock, Frank Ong, Jonathan Tamir, Hsiou-Yuan Liu, Michael Lustig, Armando Fox, and Katherine Yelick. 2018. Indigo: A Domain-Specific Language for Fast, Portable Image Reconstruction. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Vancouver, BC, 495–504. DOI: <http://dx.doi.org/10.1109/IPDPS.2018.00059>
- Marek Dvornak. 2014. Interactive As-Rigid-As-Possible Image Deformation and Registration. In *The 18th Central European Seminar on Computer Graphics*.
- Michael Grant and Stephen Boyd. 2008. Graph implementations for nonsmooth convex programs. In *Recent Advances in Learning and Control*, V. Blondel, S. Boyd, and H. Kimura (Eds.). Springer-Verlag Limited, 95–110. http://stanford.edu/~boyd/graph_dcp.html.
- Michael Grant and Stephen Boyd. 2014. CVX: Matlab Software for Disciplined Convex Programming, version 2.1. <http://cvxr.com/cvx>. (March 2014).
- Brian Guenter, John Rapp, and Mark Finch. 2011. *Symbolic Differentiation in GPU Shaders*. Technical Report. <https://www.microsoft.com/en-us/research/publication/symbolic-differentiation-in-gpu-shaders/>
- James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (July 2014), 11 pages. DOI: <http://dx.doi.org/10.1145/2601097.2601174>
- James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible Multi-rate Image Processing Hardware. *ACM Trans. Graph.* 35, 4, Article 85 (July 2016), 11 pages. DOI: <http://dx.doi.org/10.1145/2897824.2925892>
- Felix Heide, Steven Diamond, Matthias Nießner, Jonathan Ragan-Kelley, Wolfgang Heidrich, and Gordon Wetzstein. 2016. Proximal: Efficient image optimization using

- proximal algorithms. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 84.
- Felix Heide, Markus Steinberger, Yun-Ta Tsai, Mushfique Rouf, Dawid Pająk, Dikpal Reddy, Orazio Gallo, Jing Liu, Wolfgang Heidrich, Karen Egiazarian, and others. 2014. FlexISP: A flexible camera image processing framework. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 231.
- Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 152–163. DOI: <http://dx.doi.org/10.1145/1555815.1555775>
- Berthold K. P. Horn and Brian G. Schunck. 1981. Determining Optical Flow. *ARTIFICIAL INTELLIGENCE* 17 (1981), 185–203.
- Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. In *International Conference on Learning Representations*.
- Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.
- Matthias Innmann, Michael Zollhöfer, Matthias Nießner, Christian Theobalt, and Marc Stamminger. 2016. VolumeDeform: Real-time Volumetric Non-rigid Reconstruction. (October 2016), 17.
- Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. 2011. KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 559–568. DOI: <http://dx.doi.org/10.1145/2047196.2047270>
- T. Jebara, A. Azarbayejani, and A. Pentland. 1999. 3D structure from 2D motion. *IEEE Signal Processing Magazine* 16, 3 (May 1999), 66–84. DOI: <http://dx.doi.org/10.1109/79.768574>
- Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.
- Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (March 2016), 21 pages. DOI: <http://dx.doi.org/10.1145/2866569>
- Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. 2011. G2o: A general framework for graph optimization.. In *ICRA. IEEE*, 3607–3613.
- Tristan Laidlow, Jan Czarnowski, and Stefan Leutenegger. 2019. DeepFusion: real-time dense 3D reconstruction for monocular SLAM using single-view depth and gradient predictions. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 4068–4074.
- Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* 37, 4, Article 139 (July 2018), 13 pages. DOI: <http://dx.doi.org/10.1145/3197517.3201383>
- Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. 2015. SMPL: A Skinned Multi-Person Linear Model. *ACM Trans. Graphics (Proc. SIGGRAPH Asia)* 34, 6 (Oct. 2015), 248:1–248:16.
- Abhimitra Meka, Michael Zollhöfer, Christian Richardt, and Christian Theobalt. 2016. Live Intrinsic Video. *ACM Transactions on Graphics (Proceedings SIGGRAPH)* 35, 4 (2016). DOI: <http://dx.doi.org/10.1145/2897824.2925907>
- Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. DOI: <http://dx.doi.org/10.1145/2897824.2925952>
- Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew W. Fitzgibbon. 2011. KinectFusion: Real-time dense surface mapping and tracking. *2011 10th IEEE International Symposium on Mixed and Augmented Reality (2011)*, 127–136.
- M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. 2013. Real-time 3D Reconstruction at Scale using Voxel Hashing. *ACM Transactions on Graphics (TOG)* (2013).
- J. Nocedal and S. J. Wright. 2006. *Numerical Optimization* (2nd ed.). Springer, New York.
- Neal Parikh and Stephen Boyd. 2013. Proximal algorithms. *Foundations and Trends in Optimization* 1, 3 (2013), 123–231.
- Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.
- Patrick Pérez, Michel Gangnet, and Andrew Blake. 2003. Poisson Image Editing. In *ACM SIGGRAPH 2003 Papers (SIGGRAPH '03)*. ACM, New York, NY, USA, 313–318. DOI: <http://dx.doi.org/10.1145/1201775.882269>
- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. DOI: <http://dx.doi.org/10.1145/2491956.2462176>
- Javier Romero, Dimitrios Tzionas, and Michael J. Black. 2017. Embodied Hands: Modeling and Capturing Hands and Bodies Together. *ACM Transactions on Graphics, (Proc. SIGGRAPH Asia)* 36, 6 (Nov. 2017), 245:1–245:17. <http://doi.acm.org/10.1145/3130800.3130883>
- J. L. Schönberger and J. Frahm. 2016. Structure-from-Motion Revisited. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4104–4113. DOI: <http://dx.doi.org/10.1109/CVPR.2016.445>
- J. M. Shearer and M. A. Wolfe. 1985. ALGLIB, a Simple Symbol-manipulation Package. *Commun. ACM* 28, 8 (Aug. 1985), 820–825. DOI: <http://dx.doi.org/10.1145/4021.4023>
- Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. 2019. Schedule Synthesis for Halide Pipelines Through Reuse Analysis. *ACM Trans. Archit. Code Optim.* 16, 2, Article 10 (April 2019), 22 pages. DOI: <http://dx.doi.org/10.1145/3310248>
- Olga Sorkine and Marc Alexa. 2007. As-rigid-as-possible surface modeling. In *Symposium on Geometry processing*, Vol. 4.
- Robert W. Sumner, Johannes Schmid, and Mark Pauly. 2007. Embedded Deformation for Shape Manipulation (*SIGGRAPH '07*). New York, Article 80.
- Justus Thies, Michael Zollhöfer, Matthias Nießner, Levi Valgaerts, Marc Stamminger, and Christian Theobalt. 2015. Real-time expression transfer for facial reenactment. *ACM Trans. Graph.* 34, 6 (2015), 183–1.
- J. Thies, M. Zollhöfer, M. Stamminger, C. Theobalt, and M. Nießner. 2016. Face2Face: Real-time Face Capture and Reenactment of RGB Videos. In *Proc. Computer Vision and Pattern Recognition (CVPR)*, IEEE.
- Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. 2000. Bundle Adjustment - A Modern Synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice (ICCV '99)*. Springer-Verlag, London, UK, UK, 298–372.
- Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. DOI: <http://dx.doi.org/10.1145/1498765.1498785>
- Chenglei Wu, Michael Zollhöfer, Matthias Nießner, Marc Stamminger, Shahram Izadi, and Christian Theobalt. 2014. Real-time Shading-based Refinement for Consumer Depth Cameras. *ACM Transactions on Graphics (TOG)* 33, 6 (2014).
- Weipeng Xu, Avishek Chatterjee, Michael Zollhoefer, Helge Rhodin, Dushyant Mehta, Hans-Peter Seidel, and Christian Theobalt. 2018. MonoPerfCap: Human Performance Capture from Monocular Video. *ACM Transactions on Graphics* (2018).
- Samuel Zeitvogel and Astrid Laubenheimer. 2018. Towards End-to-End 3D Human Avatar Shape Reconstruction from 4D Data. In *2018 International Symposium on Electronics and Telecommunications (ISETC)*. IEEE, 1–4.
- Michael Zollhöfer, Angela Dai, Matthias Innmann, Chenglei Wu, Marc Stamminger, Christian Theobalt, and Matthias Nießner. 2015. Shading-based Refinement on Volumetric Signed Distance Functions. *ACM Transactions on Graphics (TOG)* 34, 4 (2015).
- Michael Zollhöfer, Matthias Nießner, Shahram Izadi, Christoph Rehmann, Christopher Zach, Matthew Fisher, Chenglei Wu, Andrew Fitzgibbon, Charles Loop, Christian Theobalt, and Marc Stamminger. 2014. Real-time Non-rigid Reconstruction using an RGB-D Camera. *ACM Trans. Graph.* 33, 4, Article 156 (July 2014), 12 pages. DOI: <http://dx.doi.org/10.1145/2601097.2601165>