Mapping brain activity at scale with cluster computing

Jeremy Freeman[1], Nikita Vladimirov[1], Takashi Kawashima[1], Yu Mu[1], Nicholas J Sofroniew[1], Davis V Bennett[1], Joshua Rosen[2], Chao-Tsung Yang[1], Loren L Looger[1] & Misha B Ahrens[1]

Understanding brain function requires monitoring and interpreting the activity of large networks of neurons during behavior. Advances in recording technology are greatly increasing the size and complexity of neural data. Analyzing such data will pose a fundamental bottleneck for neuroscience. We present a library of analytical tools called Thunder built on the open-source Apache Spark platform for large-scale distributed computing. The library implements a variety of univariate and multivariate analyses with a modular, extendable structure well-suited to interactive exploration and analysis development. We demonstrate how these analyses find structure in large-scale neural data, including whole-brain light-sheet imaging data from fictively behaving larval zebrafish, and two-photon imaging data from behaving mouse. The analyses relate neuronal responses to sensory input and behavior, run in minutes or less and can be used on a private cluster or in the cloud. Our open-source framework thus holds promise for turning brain activity mapping efforts into biological insights.

New technologies[1–9] based on imaging and multielectrode arrays are making it possible to record simultaneously from hundreds or thousands of neurons and in some cases, such as the larval zebrafish[7–9], nearly the entire brain. Given the growing size and complexity of neural recordings[10], analyzing and interpreting the data will be a fundamental bottleneck for neuroscience[11–14]. For example, an hour of two-photon imaging in mouse can yield 50–100 gigabytes (GB) of spatiotemporal data, and recording from nearly the entire brain of a larval zebrafish using light-sheet microscopy[7,8] can yield 1 TB or more. At this scale, even simple calculations can take hours to run on a single workstation, let alone more complex analyses examining joint dynamical patterns across the brain.

Neural data pose unique challenges for analytics. The data are complex, and the 'right' analysis is rarely obvious. Every analysis provides a lens through which to see the data, and it is often necessary to try different analyses interactively, whether by varying parameter choices or developing entirely new algorithms (**Fig. 1a**). The need for flexible analytics is especially crucial for large data sets; the more complex and heterogenous the response properties and dynamics, the wider the variety of analyses needed to reveal their structure. Prototyping analyses for small data sets is straightforward on a workstation using existing tools, but for large data sets, especially those that exceed the memory of one machine, this becomes intractable. Large-scale neuroscience thus demands a flexible platform for creating analyses and inspecting results.

Over the last several years, the private technology sector has invested heavily in 'big data' approaches that leverage the power of distributed computing (networks or 'clusters' of interconnected compute nodes) to analyze large data sets[15–17]. MapReduce[15] is a widely adopted programming model that divides a large computation into two steps: a 'map' step, in which data are partitioned and analyzed in parallel, and a 'reduce' step, in which intermediate results are combined or summarized. Many analyses can be expressed in this model[18], but conventional systems that implement MapReduce, such as the open-source Hadoop MapReduce engine[17], have key weaknesses. In particular, data must be loaded from disk for each operation, which can slow iterative computations (including many machine-learning algorithms), and makes interactive, exploratory analysis difficult. The recent, open-source Apache Spark platform extends and generalizes the MapReduce model while addressing this weakness, by introducing a primitive for data sharing called a resilient distributed data set (RDD). With Spark, a user can cache a data set, or an intermediate result, in the memory (random-access memory; RAM) across cluster nodes, performing iterative computations faster than with Hadoop MapReduce[19] and allowing for interactive analyses. Spark's application programming interfaces (APIs) also allow a user to express distributed computations—not only map and reduce but also filtering for subsets of data, joining multiple data sets together and others—all as operations over RDDs. This abstraction enables simpler, more concise implementations and performance improvements, especially for complex sequences of operations[19]. Other scripting languages and abstractions simplify MapReduce jobs (e.g., Pig and Cascading) but are built on the Hadoop MapReduce engine and thus lack the advantages of in-memory data sharing. Another advantage of Spark's APIs, in particular the Python API, is compatibility with existing libraries for scientific computing and visualization. Spark has primarily been used in industry, but these properties make it well-suited for neuroscience.

[1]Howard Hughes Medical Institute, Janelia Farm Research Campus, Ashburn, Virginia, USA. [2]University of California Berkeley, Computer Science Division, Berkeley, California, USA. Correspondence should be addressed to J.F. (freemanj11@janelia.hhmi.org) and M.B.A. (ahrensm@janelia.hhmi.org).

RECEIVED 21 MARCH; ACCEPTED 23 JUNE; PUBLISHED ONLINE 27 JULY 2014; DOI:10.1038/NMETH.3041

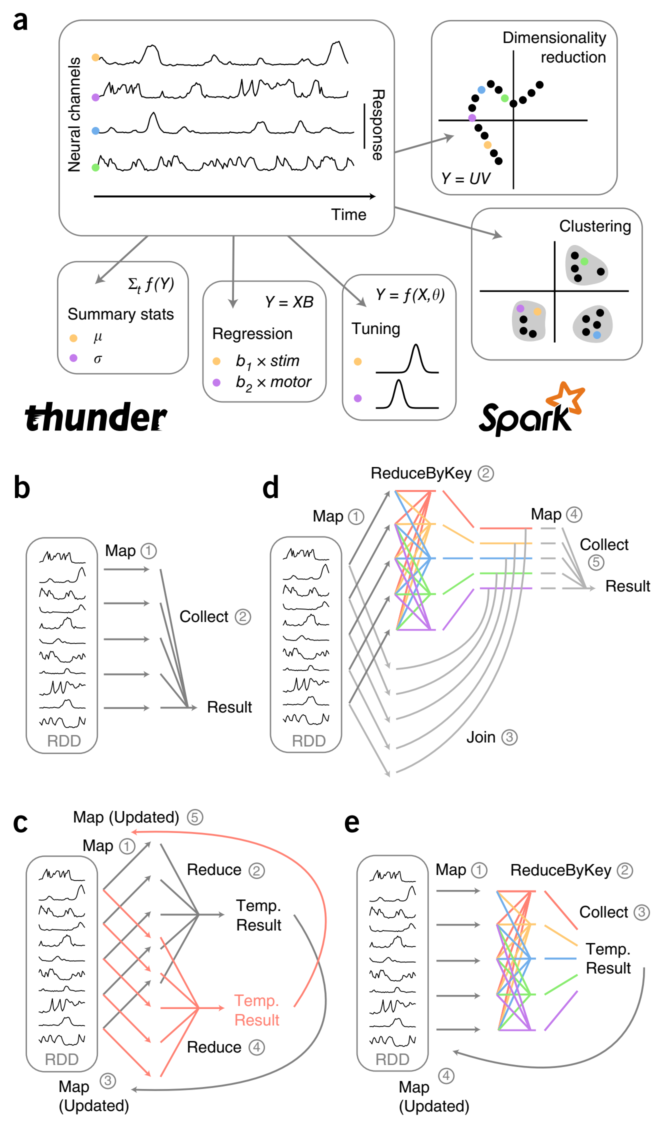

Figure 1 | A platform for large-scale neural analytics. (**a**) Example analyses in Thunder. Input data are time series from different neural channels (e.g., voxels for imaging data). Univariate analyses (summary statistics, regression and tuning) apply the same operation to every channel. Multivariate analyses (dimensionality reduction and clustering) examine joint structure across channels. Mathematical notation (for illustration only): Y, data matrix, f, arbitrary nonlinear function, μ and σ, example summary statistics, X, design matrix for regression, or stimulus parameters, B, regression coefficients, θ, model parameters, U, V, factors from a matrix decomposition. (**b**–**e**) Implementing different neuroscience analysis workflows in Spark by applying distributed operations to an RDD. Mass-univariate analyses implemented through a map operation, which performs computations (e.g., fitting a regression model) on partitions of the data in parallel, followed by collecting the results (**b**). An iterative implementation of the singular value decomposition (**c**) uses repeated map and reduce operations to distribute a sequence of matrix computations, each using the result of the previous computation. The map operations perform local matrix multiplications, and the reduce steps perform addition. Caching the RDD means that data do not need to be loaded from disk for each iteration. The correlation between a time series and its local average (**d**) uses a map and reduceByKey to compute averages of voxels belonging to a spatial neighborhood and then joins these local averages to the original data to compute correlations in a final map operation. k-means clustering (**e**) uses a map to find the center closest to each point, and a reduceByKey to average the data points from each cluster and thus update the centers.

Here we describe an open-source library of analytical tools for neuroscience built on Spark. Our library, called Thunder, implements a variety of algorithms in a modular and extendable user-oriented library (**Fig. 1**). The tools we developed enable the rapid analysis and interpretation of large neural data sets and will be relevant to a wide variety of data. To demonstrate their capability, we applied them to high-resolution calcium imaging data. We analyzed data from a paradigm, described in a companion paper[20], that combines whole-brain light-sheet imaging[7] with visual stimulation and behavior in paralyzed larval zebrafish[12]. These data include recordings from nearly all neurons in the brain and consist of time series from ~10^8 voxels, reflecting activity of ~10^5 neurons[21] and large areas of neuropil. We also analyzed two-photon imaging data from behaving head-fixed mice, including responses from ~10^3 neurons (~10^6 pixels).

Our analyses quickly find patterns in neural data, revealing neuronal populations involved in both stimulus encoding and motor behavior. Although only demonstrated on calcium imaging data, our tools could also be applied to imaging data from other indicators (e.g., voltage, neurotransmitter), high-resolution functional magnetic resonance imaging (fMRI), or electrophysiological data and thus provide a general analytics platform for large-scale neuroscience.

RESULTS

Analysis framework

Large-scale neural data analysis should be fast, interactive, extendable and accessible to the neuroscience community. To achieve these goals, we built a library, Thunder (http://freeman-lab.github.io/thunder), on top of the open-source computing platform Spark[19], using its Python API. Here we describe the library in detail: the input data it assumes, the analyses it performs, how they are implemented, how they can be extended and benchmarks of their performance.

Architecture and organization

In raw form, nearly all functional neural data sets (calcium imaging, fMRI, electrophysiology) are collections of time series from different neural channels, such as voxels. Many computations can be parallelized over channels. Thus, a natural input format is a collection of key-value pairs. The key is an identifier, and the value is a response time series. Imaging data, for example, are key-value pairs where each key is an x-y-z coordinate and each value is the fluorescence time series. If records are stored on disk in a networked or distributed file system, platforms such as Spark can read them in parallel and perform a rich variety of computations. The keys can similarly index other kinds of neural channels, for example, electrodes in a multielectrode array. The key-value format is thus a general representation for neural recordings.

In practice, neural data are acquired and saved in a temporal sequence of the form (all channels at time 0), (all channels at time 1), etc., where the channels are, for example, voxels in imaging data. Before analysis, data must be converted into the common input format by collecting across time and writing key-value records to disk, i.e., rearranging the data as (all time points for channel 0), (all time points for channel 1), etc. Some version of this one-time operation would be necessary in any platform for analyzing how

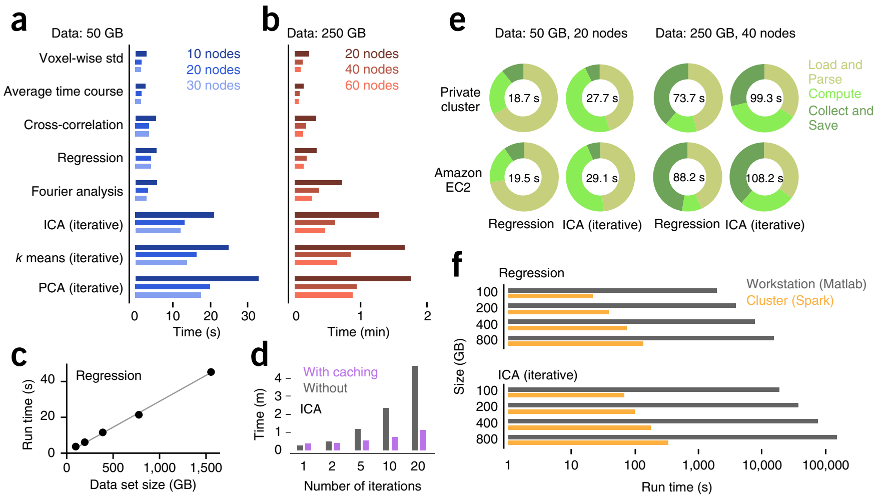

Figure 2 | Performance benchmarks for Thunder. (**a**,**b**) Run times for a 50 GB data set (512 × 512 × 4 voxels, 6,300 time points), with 10, 20 and 30 nodes (**a**) and for a 250 GB data set (2,048 × 1,024 × 18, 894 time points), with 20, 40 and 60 nodes (**b**). std refers to standard deviation. Each node had 16 cores and 128 GB RAM (100 allocated to Spark), see Online Methods for additional specifications. All computations were performed on cached data and only include duration of computation (not loading or saving results, see **e**). For iterative computations, five iterations were used. Times reflect the minimum from three runs. (**c**) Run time as a function of data set size, using 60 nodes. Data dimensions were (2,048 × 1,024 × *z*, 1,500 time points), where *z* was varied from 2 to 32 to vary total size. Gray line, linear fit with intercept fixed at 0, slope = 0.029. (**d**) Run times for varying iterations of independent component analysis, applied to the same 50 GB data set from **a**, with and without data caching (run times for cached data include the initial time to load and cache). The slight increase in run time for caching with one iteration likely reflects a small serialization penalty for populating the cache. (**e**) Run-time breakdown on two cluster environments for the same two data sets from **a**,**b**; durations are shown for loading, parsing and caching data, performing the computation (identical to times reported in **a**,**b** for the private cluster) and collecting and saving the result. Numbers in center give combined time. Amazon EC2 cluster used 'compute optimized' instances, each having 32 vCPUS (with hyper-threading, so this is roughly matched to the 16 real cores per node on the private cluster) and 60.5 GB RAM (53.3 allocated to Spark). (**f**) Run times for performing two analyses on a private cluster (with Spark) or on a single 12-core Linux workstation with 64 GB RAM (with Matlab), for the same data analyzed in **c**. Input data were identical in content (single files containing response time series from all voxels of a single plane) but a text file for Spark and a MAT file for Matlab. In both cases times include loading and parsing, computing, and (for Spark) collecting results. Ten iterations were used for ICA. For convenience, times here were extrapolated where possible (e.g., timing one iteration and multiplying by the number of iterations), and extrapolation was verified in a subset of cases.

the data vary over both space and time. Assuming a common input format additionally supports data sharing and domain-general analysis tools.

The first analysis step in Thunder uses the load function to turn input data into an RDD in Spark and apply any desired preprocessing, as in: data = load(sc, 'dataset', 'dff').cache(), where dataset is the location of the data, sc is the Spark context (a class that acts as the main entry point for Spark functionality) and dff specifies one of several preprocessing options. The optional .cache() marks the RDD for caching into RAM, which is important if it will be queried multiple times (e.g., during an iterative computation or during interactive analysis). A variety of RDD operations can then be performed. The core of Thunder is expressing different neuroscience analyses in the language of RDD operations (**Fig. 1b**–**e**). Spark exposes RDD operations through APIs in the programming languages Scala, Java and Python. Thunder is primarily written in the Python API (PySpark) because it enables the use of robust numerical and scientific computing libraries (e.g., NumPy and SciPy), and provides the simplest front end for new users. Finally, Thunder is designed so that analyses can be run either as standalone scripts or interactively within the Python shell (or an iPython notebook), enabling immediate inspection and visualization of results.

Analyses and implementation

Thunder includes several univariate and multivariate analyses, organized into regression, factorization, clustering and time-series statistics (**Fig. 1**). The components of the analyses are modular, making it easy to combine or extend them.

Univariate analyses (sometimes called 'mass-univariate') perform a computation on each neural channel, for example, voxel-wise summary statistics or voxel-wise regression. These analyses are naturally expressed through a map operation that distributes the computation across the worker nodes of the cluster, followed by a 'collect' to return the result to the driver (**Fig. 1b**). We separate the computation performed on each channel from the RDD operation itself; for example, by using a high-level RegressionModel class that implements the mapping and using subclasses, each for particular models, that implement the fitting. This design enables users with only minimal knowledge of Python to modify the underlying computations (e.g., the form of regression) or add new ones.

Multivariate analyses examine multiple channels, in some cases the entire data set. These analyses typically require (at least) a combination of map and reduce operations, where the map step distributes some computation across partitions of the data, and the reduce step uses a commutative and associative function to combine the results (**Fig. 1c**). Such analyses can be more involved to implement, but the algorithms we include provide a foundation for future work. For example, many multivariate analyses use the singular value decomposition (SVD), which seeks to approximate an $n \times t$ matrix as the product of $n \times k$ and $k \times t$ matrices, where n is the number of channels and t the number of time points. We provide two large-scale implementations of the SVD. The first is suitable for 'tall-and-skinny' matrices (e.g., large n, small t); it uses a map to compute rank 1 outer products, a reduce (with addition) to aggregate the results and another map to project data into the recovered subspace. For the case of many time points ($t > 1{,}000$), we implemented an alternative large-scale iterative algorithm based on expectation maximization[22], which expresses a sequence of matrix updates as map and reduce steps (**Fig. 1c**), and can be faster than the direct method when a small number of singular vectors are required (see below).

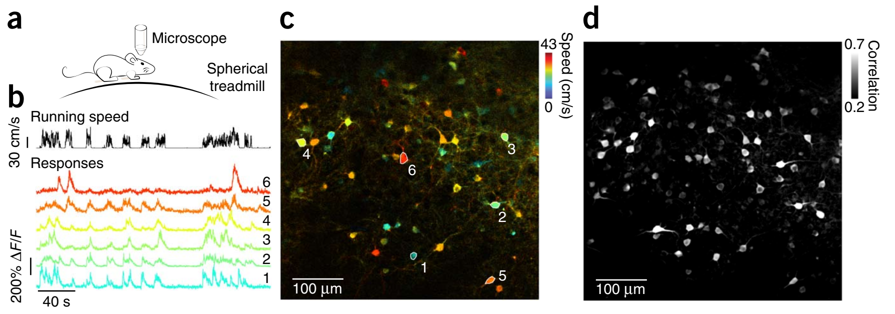

Figure 3 | Maps of *in vivo* two-photon calcium imaging data from mouse cortex revealing response modulation by locomotion. (**a**) Schematic of experimental preparation for two-photon calcium imaging of neural activity in mouse somatosensory cortex during locomotion. (**b**) Animal running speed and calcium responses ($\Delta F/F$) of example cells that showed modulation owing to running speed. Each response is the average of small groups of ~50 voxels. (**c**) Map of running speed modulation from one imaging plane. Numbers indicate example cells from **b**. Color indicates preferred speed and brightness indicates strength of modulation (assessed as R^2 from linear fit, maximum of 0.0125). (**d**) Map of local correlations. Brightness, correlation coefficient between calcium fluorescence of each voxel and the average of a local 7 × 7 neighborhood.

Many other analyses require RDD operations beyond map and reduce. For example, the filter operation can be used to extract spatial subregions during an analysis. Correlating each voxel's time series to the average of a local neighborhood involves a flatMap and reduceByKey to average local groups of voxels, a join to combine the raw data with those averages and a final map to compute the correlation coefficients (**Fig. 1d**). *k*-means clustering involves a map step (to compute, for each point, the closest of the *k* centers), followed by a reduceByKey (to average the data points belonging to each cluster; **Fig. 1e**). With a working knowledge of these operations, and of Python, users can easily extend the analyses or create new ones. The components of Thunder can also be fluidly combined. For example, we could perform regression on every voxel and cluster the resulting regression coefficients or compute residuals from a model fit and use dimensionality reduction to examine structure in the noise. Both can be expressed in a few lines of code.

Setup and installation

Running Thunder on a cluster requires first deploying Spark as well as installing Python and the necessary Python libraries, all available as open source. Spark can be deployed on a private cluster (e.g., at a university or research facility) or on Amazon's EC2 cloud computing services. For EC2 usage, we provide a customized Spark EC2 launch script that creates a Spark cluster on EC2 and preinstalls Thunder (and its dependencies). For a university cluster running the Univa Grid Engine, we provide instructions for setting up Spark in standalone mode (Online Methods). The advantages of EC2 are ease of deployment and scalability (the number of nodes can be tailored to the data set and desired performance, see below). The disadvantage is that data must be transferred to cloud storage, but when many analyses are performed on the same data, this one-time cost may be insignificant.

Benchmarks

To characterize performance, we first measured run times for two data sets, three cluster sizes and nine analyses, all implemented on a private cluster running Spark. The test data sets were two-photon imaging data (**Fig. 2a**, 512 × 512 × 4 voxels, 6,300 time points, ~15 min at 7.5 Hz, 50 GB) and light-sheet imaging data (**Fig. 2b**, 2,048 × 1,024 × 18 voxels, 900 time points, ~15 min at 1 Hz, 250 GB). We analyzed cached data, and the reported run times only include durations required to perform the computation, not time for loading and parsing input data or collecting and saving results (see below for run times of those steps; Online Methods). In general, run times increased with algorithm complexity and decreased with cluster size (**Fig. 2a**,**b**). All voxel-wise statistics completed in under 30 s, and iterative algorithms completed within a couple minutes; with 40 nodes, we were able to query the average time series of the larger data set in less than 5 s and estimate three singular vectors in under a minute. Performance on both data sets increased with cluster size. For the larger data set, performance nearly doubled from 20 to 40 nodes (**Fig. 2b**; geometric mean improvement across analyses, 1.89); on the smaller data set, it also increased from 10 to 20 nodes (**Fig. 2a**, geometric mean improvement across analyses, 1.58). In both cases, further increasing cluster size led to smaller improvements. This ceiling effect likely reflects communication overheads: as the number of nodes increases, less data are processed per node, and computation times are eventually dominated by the fixed cost per task of serializing and transferring data and code. In a separate test we examined performance for one analysis on a larger range of data set sizes; run times scaled linearly with data size, and analysis of 1.5 TB finished in under a minute (**Fig. 2c**). Together, these results suggest that the framework can efficiently process a variety of data sets and will scale well to more complex analyses.

To demonstrate the advantage of data caching, a capability unique to Spark, we computed run times on one data set for different numbers of independent component analysis (ICA) iterations; in one case, the data were reloaded during every iteration and in the other, the data were loaded once and cached (**Fig. 2d**). As reported previously[19], caching improves performance for iterative algorithms because the cost to load data is incurred only once; the speedup resulting from caching is thus greater for more iterations. Fast iterative algorithms can be especially useful for large-scale problems; for example, we compared run times for iterative and direct implementations of the singular value decomposition (see above), each factoring an $n \times t$ matrix into $n \times k$ and $k \times t$ matrices, with $k = 3$. For a data set with dimensions $n = {\sim}10^8$, $t = {\sim}10^2$ (150 GB), using 40 nodes, the direct method took 92 s, and one iteration of the iterative method took 12 s. For a data set with $n = {\sim}10^8$ and $t = {\sim}10^3$ (750 GB), the direct method took 1,983 s, whereas one iteration of the iterative method took only 27 s. Thus, for larger t, the cost of even 10 iterations is about 0.1× the cost of the direct method.

In real use, analysis includes not only computation but loading (and parsing) data and saving results to disk. For the two data sets compared above, we provide these durations under two

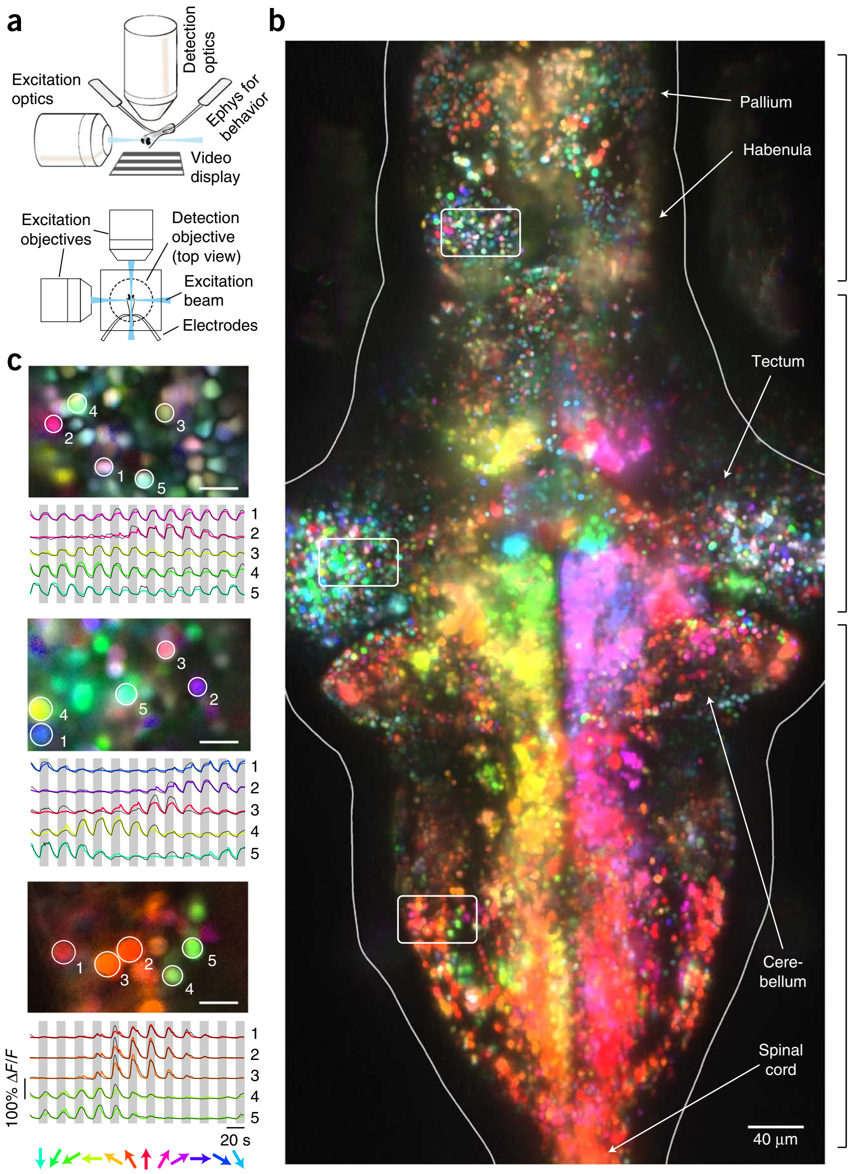

Figure 4 | Direction tuning maps of whole-brain neural activity measured in a larval zebrafish with light-sheet microscopy while zebrafish were presented visual stimuli. (**a**) Schematic of experimental preparation. Ephys refers to electrophysiological recordings for monitoring behavior (Online Methods). (**b**) Maps of direction tuning across the brain derived by fitting every voxel with a tuning-curve model that separately describes the temporal response profile and the tuning to direction. Color, preferred direction (see legend in **c**); saturation, tuning width (i.e., circular variance); brightness, response strength (Online Methods). White means responsive but without unidirectional tuning. Image shows maximum intensity projection through 39 planes covering 195 µm. (**c**) Magnified regions of the habenula, tectum and hindbrain, each from a single imaging plane. Scale bars, 10 µm. Time series show responses of example cells. Black traces, Δ*F*/*F* averaged from small groups of ~100 voxels; responses averaged across five presentations. Color traces, prediction of best-fitting tuning curve model (Online Methods). Gray vertical bars, stimulus on; white vertical bars, stimulus off.

environments: a private cluster with a networked file system and an Amazon EC2 cluster with 'compute optimized' instance types running a Hadoop distributed file system (**Fig. 2e** and Online Methods). For regression, the run time is dominated by loading and parsing. For a more complex analysis (ICA), relatively more time is spent on computation. Performance patterns were similar between the private cluster and EC2, as were overall run times (**Fig. 2e**). We also performed the same tests using the 'general purpose' instance type (data not shown). These instances have a quarter the cores and half the RAM of the compute-optimized instances but are also a quarter of the cost; in that case run times were ~3× slower for these instances, with longer load times in particular.

Finally, we compared a cluster running Spark to a single powerful workstation running Matlab, across four data sizes (from the same data set used in **Fig. 2c**). Of course, the cluster will outperform the workstation but for an end user deciding between alternatives, the magnitude of improvement is of relevance. Spark's performance increase was as large or larger than expected based on the number of nodes (40×), achieving an improvement of at least 200× for an iterative analysis (**Fig. 2f**). The key limitation of any single workstation solution is that, owing to memory limitations, portions of the data must be loaded and processed sequentially, leading to slow run time despite fast core computations (e.g., matrix multiplication). This is especially problematic for iterative analyses; in contrast, Spark can cache the entire data in RAM. Also of practical interest, the price of one such workstation could pay for ~300 h of 40-node cluster usage on EC2, with 1.2 TB total RAM and 320 virtual CPUs (vCPUs).

Two-photon imaging example

For an initial demonstration of our framework, we analyzed two-photon calcium imaging data from layer 2/3 somatosensory cortex neurons in head-fixed mice running on a spherical treadmill in a tactile virtual reality environment[23] (**Fig. 3a**). The genetically encoded calcium indicator GCaMP6s was delivered via infection with adenoassociated virus (AAV2/1, synapsin-1 promoter)[24]. In the visual cortex, neurons exhibit tuning to running speed[25] and locomotion-induced modulation[26]. We used Thunder to perform a voxel-wise interrogation of locomotion-induced modulation in the somatosensory cortex (**Fig. 3b**,**c**). Analysis of these data, which are typically 50–100 GB, at the level of voxels is possible but burdensome with ordinary methods (e.g., Matlab; **Fig. 2e**). We used mass-univariate tuning analyses to compute for each voxel how well its response was predicted by running speed, and the running speed to which it responded best. We combined these two properties into a map in which brightness indicates the reliability of prediction and hue indicates tuning (**Fig. 3c**). We also used local correlation[27] to estimate reliably responsive neurons independent of covariates (**Fig. 3d**). Both the tuning and local

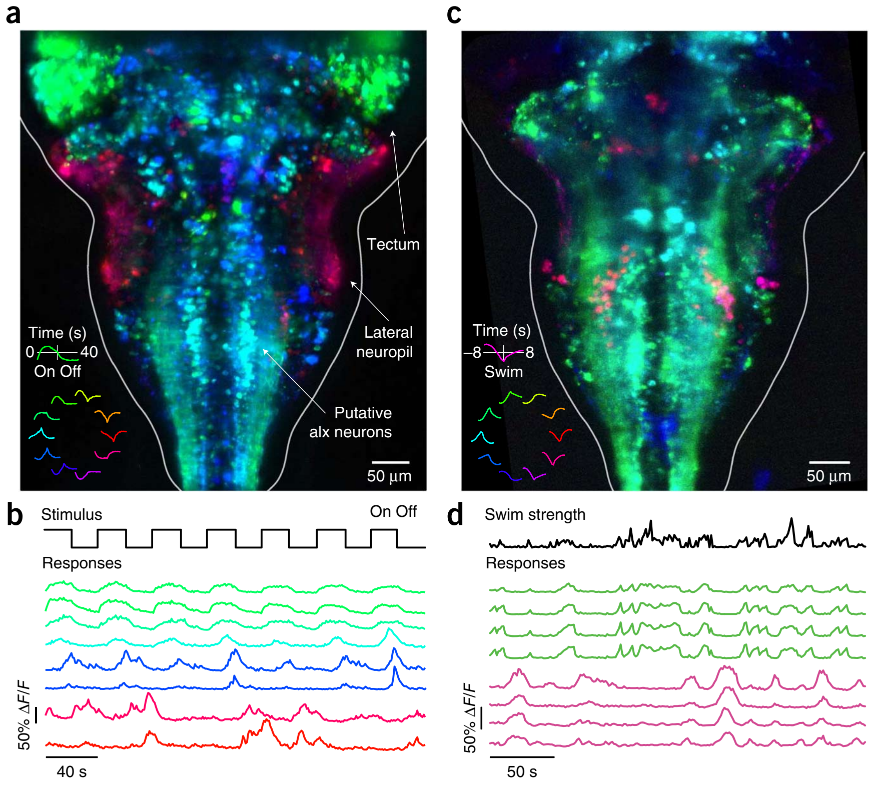

Figure 5 | Maps of sensorimotor responses in larval zebrafish. (**a**) Maps of response dynamics obtained by reducing each voxel to a pair of numbers (weights on the first two principal components) using PCA. In the color wheel (bottom left), the first two principal components are the red and yellow-green traces, and different linear combinations describe a family of dynamics: angle (hue) indicates response shape, and radius (brightness) indicates response strength. These values determine the hue and brightness for each voxel in the map. Shown is maximum intensity projection through 15 planes (each 5 µm apart). (**b**) Stimulus sequence and calcium responses ($\Delta F/F$) of individual neurons; examples highlight different response types. (**c**) Map derived from an experiment in which the fish swam sporadically in response to a constantly slowly moving stimulus. Lagged cross-correlation and PCA were used to reduce each voxel to a pair of numbers, capturing the timing of response relative to swimming and visualized as in **a**. Shown is maximum intensity projection through 47 planes (each 5 µm apart). (**d**) Swimming strength (from electrophysiological recordings) and calcium responses ($\Delta F/F$) of individual neurons during self-driven swimming.

correlation maps can be used to help identify and segment individual neurons. A comparison of the two maps suggests a large fraction of responsive neurons were modulated by locomotion. The ability to quickly and flexibly generate such maps will be of immediate use for the wide variety of two-photon imaging data currently being collected.

Large-scale applications

The capabilities of our framework are well illustrated through applications to data generated by techniques such as light-sheet[7,8,20] and light-field[2,4] microscopy. A recently developed combination of light-sheet imaging in the larval zebrafish with visual stimulation and behavioral monitoring[20] enables simultaneous recording from virtually all neurons in the brain during sensorimotor behavior. The resulting data are typically hundreds of gigabytes per experiment and thus demand tools such as the ones we developed. Here we show how Thunder reveals patterns of biological importance in these data. Detailed examples of implementation and sample data sets are available via http://research.janelia.org/zebrafish/.

Direction selectivity

Across the animal kingdom, organisms have both sensory neurons and behaviors tuned to the direction of visual motion[28,29]. In the zebrafish optic tectum, mechanisms of direction selectivity have been characterized both at the single cell and the network level[30–33]. Light-sheet imaging with visual stimulation[20], combined with appropriate analyses, can reveal patterns of direction selectivity across virtually the entire brain. We generated a transgenic fish Tg(*elavl3:H2B-GCaMP6s*)[jf5] that expresses nuclear-localized GCaMP6s[24] in almost all neurons (Online Methods). We measured neural responses while presenting fish with a whole-field moving stimulus that changed direction every 20 s (**Fig. 4a**), using a light-sheet configuration that avoided stimulation of the eyes[20] (Online Methods).

Direction selectivity is commonly characterized by measuring responses to moving patterns and fitting the responses with a suitable model[9,29,31,34]. In Thunder, a combination of mass-univariate tuning (as described above) and regression can model ~10^8 time series in parallel, by first estimating the response to each direction and then fitting a circular tuning curve. For visualization, we constructed a map in which every voxel is colored by its preferred direction (hue), tuning width (saturation) and response strength (brightness) (**Fig. 4b**, and **Supplementary Videos 1** and **2**). The tectum showed heterogeneous tuning of neighboring cells, consistent with previous reports[35]. Medial and ventral parts of the midbrain and hindbrain showed strong and coarse preferences for direction (left, right or forward) (**Fig. 4b**). Parts of the forebrain, including the habenula[36], also showed heterogeneity, including many cells that responded to the moving stimulus but nearly equally to all directions (**Fig. 4c**). Differences in tuning heterogeneity may signify qualitative differences in the types of computations that these areas perform. In particular, the coarse biases, being in the hindbrain, likely reflect circuits underlying motor coordination, whereas the heterogeneity in visually responsive areas may reflect fine-scale visual computation. Note that this particular analysis assumes unimodal tuning to direction, so responses tuned to orientation, but not direction, will appear untuned; by design, extending Thunder with bimodal tuning would be straightforward.

Maps of sensorimotor responses

Whereas mass-univariate analyses characterize voxels independently, many response properties, especially those involved in motor control, involve the joint dynamics of neural populations[11,13], and demand appropriate multivariate analyses. We characterized

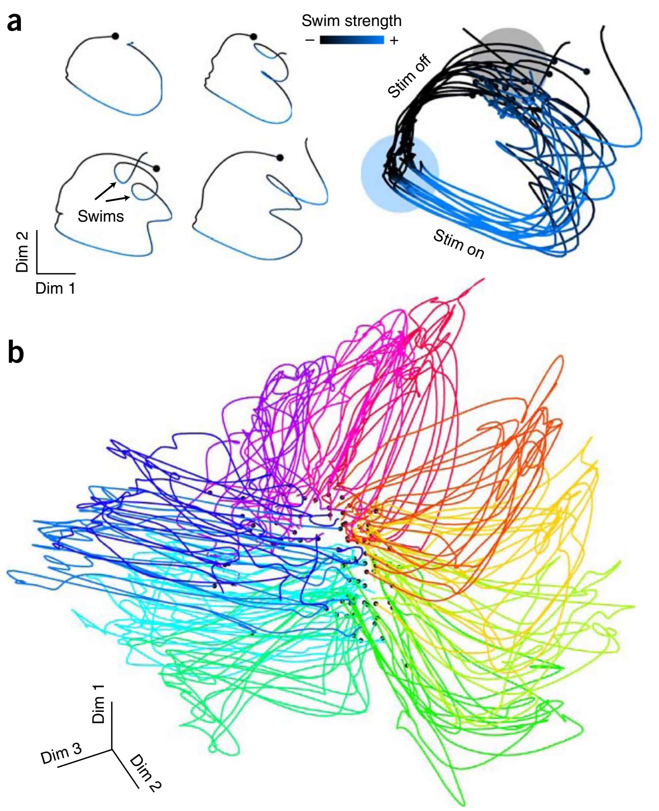

Figure 6 | Dynamical portraits of larval zebrafish whole-brain activity via dimensionality reduction. (**a**) Trial-by-trial trajectories through a neural state space, derived using PCA (Online Methods) from a data set like that reported in **Figure 5a**,**b**; each trace corresponds to one presentation of the stimulus (12 s motion, 12 s stationary). Trace color (black to blue), strength of swimming derived from electrophysiology. Black dots and gray shaded region, trial onset. Blue shaded region, stimulus onset. (**b**) Trial-by-trial trajectories related to stimulus direction, derived using regression and PCA (Online Methods), from a data set similar to that reported in **Figure 4**. Each trace corresponds to one presentation of the stimulus (10 s motion, 10 s stationary). Color, stimulus direction, as in **Figure 4c**. Black dots indicate trial (and stimulus) onset.

motor dynamics during stimulus-driven behavior using a simple visual stimulus alternating between forward motion and stationary pattern, in a fish expressing cytoplasmic GCaMP5G[7,37] pan-neuronally under the *elavl3* promoter[38].

Neurons across the hindbrain and midbrain exhibited a variety of dynamics in response to the stimulus. Principal component analysis (PCA) is a method for characterizing common dynamical patterns in such heterogeneous assemblies[11,13]. Thunder performs PCA through large-scale implementations of the singular value decomposition (SVD) (see above and Online Methods), which we applied here to the stimulus-cycle averaged neural responses. This analysis revealed a coordinated sequence of events after the onset of stimulus across the brain (**Fig. 5a**,**b**). For spatial visualization, we applied a polar transform to recode every voxel's projection onto the first two components into an angle and an amplitude. This representation is sensible because it depicts separately the temporal shape of the response (as a color) and the strength (as brightness), yielding a visualization of response dynamics across the brain (**Fig. 5a**, and **Supplementary Videos 3** and **4**). Early responses appeared in the optic tectum (calcium signals peaking within the first 3–5 s, green), followed by the hindbrain (blue-green), including what appear to be thick axonal bundles in the ventral hindbrain, and columns of neurons that, based on their location, may correspond to a stripe of glutamatergic neurons (specifically, alx neurons) known to be involved in motor signaling[39,40]. We found delayed responses (blue) in the anterior hindbrain as well as in parts of the cerebellum, which were also more variable across stimulus presentations (**Fig. 5b**). Notably, two regions exhibited 'off' responses (i.e., larger responses during the period of no stimulus motion): areas of neuropil in the lateral hindbrain (magenta), as well as two small clusters of cells located far laterally.

More subtle aspects of motor response dynamics can be revealed through more refined experimental designs, which in turn demand more targeted dimensionality reduction. In one such experiment, we isolated aspects of motor generation using a continuous, slow stimulus (Online Methods), which elicited 'self-driven' swims. Because the stimulus was constant, the swim durations were determined by the fish, not the stimulus, and the instantaneous strength of the animal's swimming was captured with an independent electrophysiological recording[12,20].

Response patterns were characterized by correlating the response of each voxel with the animal's swimming at different temporal lags, and embedding and visualizing the ensemble of lagged-correlations in a low-dimensional space (**Fig. 5c**). Thunder's modularity implements this analysis by combining mass-univariate cross-correlation with PCA. In the resulting map (**Fig. 5c**, and **Supplementary Videos 5** and **6**), there were fewer responses in the tectum compared to the block-alternation experiment, consistent with a primarily sensory role for those signals. Two functional networks were instead dominant (**Fig. 5c**). The first (cyan-green) exhibited responses locked to the animal's swimming, in similar populations to those described above (**Fig. 5a**). But a second, sparser population (magenta-red) showed elevated activity preceding swimming and lower activity during swimming (**Fig. 5c**,**d**). Inspection of individual neuron traces (**Fig. 5d**) showed that responses followed the offset of swimming, after which they increased their responses (toward a plateau), apparently until initiation of the subsequent swim, at which point their activity abruptly dropped. Although the dynamics of the calcium indicator may be partially responsible for the ramping nature of the responses, we believe this is unlikely to be the cause, owing to the slow timescale (5–10 s) of the signal increase (relative to the rise and decay time constants of GCaMP5G[37]). These neurons were primarily localized to two regions, a horseshoe-like structure in the dorsal hindbrain and two far lateral clusters of neurons in the medial part of the hindbrain (**Fig. 5c**,**d**), and show how analyses of whole-brain data can reveal small neuronal populations with distinctive dynamics.

Dynamical portraits

Dimensionality reduction reveals the spatial organization of dynamics (**Fig. 5**), but can also reveal how patterns of neural activity evolve over time on a trial-by-trial basis. We repeated the optomotor experiment during simultaneous recording of behavior. We used PCA to estimate a low-dimensional space and then project the complete time course of neural responses (across individual trials) into the low-dimensional state space

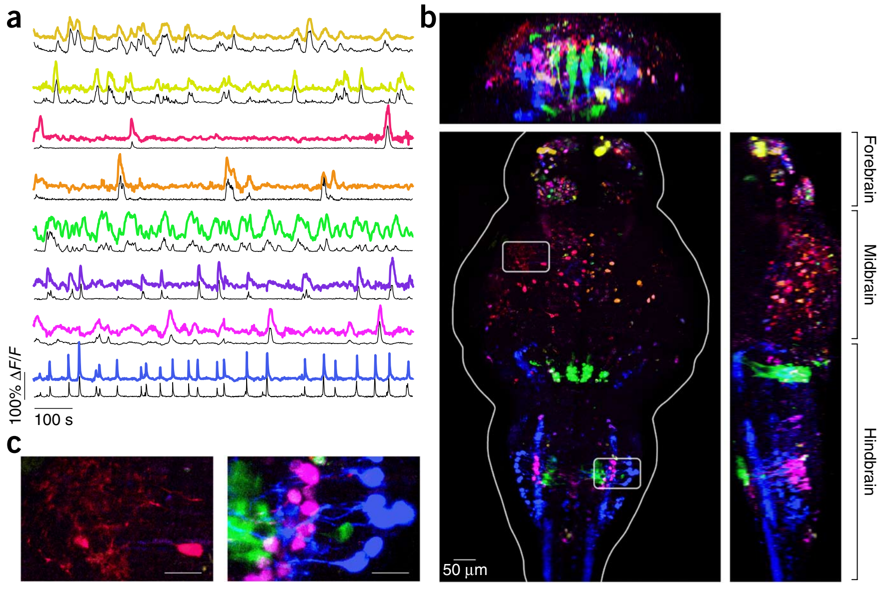

Figure 7 | Analysis of whole-brain spontaneous activity by ICA reveals functionally defined networks. (**a**) Colored lines, temporal traces associated with eight recovered independent components. Components scaled to have similar norm, units arbitrary. Black lines, calcium responses ($\Delta F/F$) of example neurons within each of the eight components. (**b**) Map derived from ICA showing 8 of 20 components with the most well-defined spatial structure (chosen by eye). Absolute value of each component was used to generate a separate color map; maps for the eight components across 41 planes were combined (after scaling each for contrast) with maximum intensity projection through x, y or z. (**c**) Magnifications of maximum intensity projection (see boundaries in **b**) indicating response patterns in tectal neuropil (left) and in the hindbrain (right). Scale bars, 25 μm.

(Online Methods). The resulting trajectories (**Fig. 6a** and **Supplementary Video 7**) show how neural activity evolves through time. Trajectories before and immediately after the onset of stimulus were stereotyped across trials but exhibited variability later, likely reflecting variability in behavior. Indeed, on individual trials, successive swims after the onset of stimulus corresponded to deflections in state space. Unlike voxel-wise tuning (**Figs. 3** and **4**), these results demand simultaneous measurement across the brain and multivariate analyses that can examine the entire data set, especially to link single-trial dynamics to behavior[13].

As a second example, we estimated trial-by-trial trajectories from responses to moving, oriented gratings (similar to the experiment reported in **Fig. 4**). For this analysis, we combined regression (to extract variability related to the different stimulus directions) with dimensionality reduction (**Fig. 6b**, **Supplementary Video 8** and Online Methods). Whereas before trajectory shape was stereotyped across trials, here they diverged depending on the direction of the stimulus.

Modes of spontaneous activity

The analyses described thus far examined an explicit stimulus or behavior. But large-scale analytics are equally powerful in the 'spontaneous' regime. In human functional imaging data, for example, ICA reveals common patterns of activity during rest[41]. Thunder includes a large-scale implementation of ICA. The first step is to reduce dimensionality and whiten the data using SVD. The data are then subjected to a sequence of iterative updates to estimate the unmixing matrix[42], efficiently implemented using map-reduce operations (Online Methods) and benefiting from caching (**Fig. 2d**).

We applied ICA to spontaneous activity in the zebrafish measured while the animal was immobilized in agar with no visual stimuli other than the laser (same data set has been reported previously[7]; the eyes were not excluded from laser stimulation; Online Methods). We visualized the spatial modes recovered by ICA by assigning each a color and composing them into a map (**Fig. 7**, and **Supplementary Videos 9** and **10**). Two functional networks were dominant: the 'hindbrain oscillator' consisting of four clusters that exhibit slow lateralized oscillations in the hindbrain and an area close by the inferior olive as well as the 'hindbrain-spinal network'. These analyses corroborate a correlation-based measure[7] but also find additional populations, including lateralized, temporally sparse 'explosions' of activity in both the tectum and the forebrain, and a family of functionally defined networks in the hindbrain with distinct dynamics. Many of these functional networks include structures other than cell bodies (i.e., axons and dendrites), which may have been missed by cell body region of interest (ROI)-based analyses. The ability to perform such analyses on whole-brain, subcellular-resolution imaging data, especially when coupled with anatomical methods[43–45], will make it possible to link functional networks to anatomy and connectivity.

DISCUSSION

We focused our examples on analysis of calcium imaging data and performed all analyses on voxel-by-voxel data. Some calcium imaging studies have taken this approach[31], whereas others have first defined ROIs, typically centered on somata, either manually[14] or with semiautomated methods using anatomy[46] or activity[47]. Thunder both complements and facilitates ROI-based approaches. The computations underlying any ROI-detection algorithm are time-consuming, and Thunder can be used to efficiently implement them. However, most automated methods still effectively require some manual inspection, which may become impractical for data sets consisting of many neurons; applying analyses directly to voxels is particularly useful for exploratory analyses. Cell bodies are readily identifiable from the resulting voxel-wise maps. In some cases, so are signals of interest in single axons or dendrites and in areas of neuropil. Finally, recordings from larger regions of tissue in other animals may yield data sets with ROI counts equal to or exceeding the voxel counts handled here; Thunder could readily handle such data and is thus built to scale alongside the coming technological advances in neuroscience.

Example analyses showed how our tools can reveal patterns of biological importance in neural data. The zebrafish whole-brain recordings would have been difficult to analyze without the tools we introduced. Whole-brain maps of direction selectivity revealed differences in the organization of stimulus selectivity across brain areas. Dimensionality reduction of responses during optomotor and self-driven swimming behavior identified groups of neurons with distinct dynamics, including previously unknown signals such as 'off' responses in the dorsal and lateral hindbrain. Combining additional experimental assays with more complex model fitting in this system is likely to yield rich insights into network dynamics during behavior. In the future, current and new techniques[2–4] will generate ever larger data sets in a variety of organisms and will benefit from the tools introduced here.

As in other computational fields where large data sizes are becoming the norm, neuroscience will likely increasingly rely on some form of distributed computing. Several factors motivated our building a library on Spark in particular. First, Spark's data caching addresses the bottleneck of data loading, which fundamentally limits single-workstation solutions as well as distributed computing solutions built on the Hadoop MapReduce engine, which Spark matches or outperforms for many computations[19]. Caching is particularly important for iterative computations, which arise frequently in machine-learning algorithms and statistical methods. Spark also addresses the issue of ease of use, by lowering requirements on users' prior experience with distributed systems. Through its APIs, the abstraction of RDDs and optimized job scheduling, writing analyses in Spark is more concise than in other platforms and faster, while requiring minimal control over the distribution and execution of work[19]. We designed Thunder itself so that anyone with minimal Python experience can use all of the analyses. Finally, because we developed Thunder in Spark's Python API, analyses in Thunder can be applied interactively in a Python shell (or an iPython notebook), and results can immediately be inspected, visualized and shared.

Modern distributed computing has the potential to change analytics in neuroscience. Thunder and Spark, however, are a notable departure from more common analysis approaches, for example, running Matlab on a single workstation. An investment of time, and possibly money, will be required to use the tools advocated here, but we took steps to make it easier to start. First, we provided an Amazon EC2 launch script that creates a Spark cluster with Thunder preinstalled. We also provided instructions that facilitate installation of Spark on at least one common university cluster environment. The provided benchmarks can guide selection among cluster sizes for particular use cases.

A key advantage of our approach is that it makes large-scale neural analytics accessible to a broad community. Spark is open-source, as is our library. It relies on the scientific computing routines in Python (through Numpy and Scipy), which are also freely available and increasingly popular tools for data science. A cluster is required, but the cloud computing resources of Amazon (EC2) are available to all. The domain-general specification of both input data and stimulus or behavioral variables makes it straightforward to apply the analyses to imaging data from other species and modalities or to other neural data, such as recordings from large multielectrode arrays. In particular, our library and related onoing efforts to use Spark for analyzing light-field imaging data (M. Broxton, L. Grosenick, B. Poole and K. Deisseroth; personal communication) are complementary and can be integrated. With a working knowledge of Python, and of basic distributed operations, members of the neuroscience community will be able to extend Thunder with new, more complex analyses[48,49].

This work stands at the intersection of two of the most exciting frontiers of modern neuroscience: recording from very large populations of neurons and using distributed computing to find patterns in very large data sets. By bridging these domains, we offer an analytical framework for large-scale neuroscience.

METHODS

Methods and any associated references are available in the online version of the paper.

Note: Any Supplementary Information and Source Data files are available in the online version of the paper.

ACKNOWLEDGMENTS

We thank K. Carlisle and R. Lines for help installing and running Spark on the Janelia Farm Research Campus Compute Cluster, D. Ganguli and M. Zaharia for advice on using Spark, G. Merlino for advice on benchmarking, C. Ziemba, C. Stock and T.J. Florence for help testing EC2 installation procedures, P. Keller for his help and advice in building the light-sheet microscope, B. Coop and T. Tabachnik for their help with hardware design, M. Coleman for writing the light-sheet microscope control software Zebrascope and continuing support, S. Narayan for help with zebrafish experiments, K. Svoboda and S. Peron for help setting up the mouse two-photon imaging, B. MacLennan for help with mouse surgeries, D.G.C. Hildebrand and M. Koyama for discussions, the Janelia Farm Research Campus vivarium staff for fish and mouse husbandry, and V. Jayaraman, G. Murphy, K. Svoboda and P. Keller for comments on an earlier draft of the manuscript. This work was supported by the Howard Hughes Medical Institute.

AUTHOR CONTRIBUTIONS

J.F. and M.B.A. conceived of the project. J.F. developed the analysis library and analyzed the data. N.V., M.B.A. and T.K. developed the zebrafish light-sheet imaging experimental preparation. N.V., M.B.A., Y.M., T.K. and J.F. collected the zebrafish data. N.J.S. developed the mouse experimental preparation, collected the data reported in **Figure 3** and helped develop the analysis of those data. D.V.B. contributed to zebrafish experiments. J.R. contributed code to the analysis library. C.-T.Y. and L.L.L. developed the Tg(*elavl3:H2B-GCaMP6s*)^{jf5} transgenic fish line. J.F. and M.B.A. wrote the paper, with input from all authors.

COMPETING FINANCIAL INTERESTS

The authors declare no competing financial interests.

1. Grewe, B.F., Langer, D., Kasper, H., Kampa, B.M. & Helmchen, F. High-speed *in vivo* calcium imaging reveals neuronal network activity with near-millisecond precision. *Nat. Methods* **7**, 399–405 (2010).
2. Broxton, M. *et al.* Wave optics theory and 3-D deconvolution for the light field microscope. *Opt. Express* **21**, 25418–25439 (2013).
3. Quirin, S., Peterka, D.S. & Yuste, R. Instantaneous three-dimensional sensing using spatial light modulator illumination with extended depth of field imaging. *Opt. Express* **21**, 16007–16021 (2013).
4. Prevedel, R. *et al.* Simultaneous whole-animal 3D imaging of neuronal activity using light-field microscopy. *Nat. Methods* **11**, 727–730 (2014).
5. Churchland, M.M. *et al.* Neural population dynamics during reaching. *Nature* **487**, 51–56 (2012).
6. Holekamp, T.F., Turaga, D. & Holy, T.E. Fast three-dimensional fluorescence imaging of activity in neural populations by objective-coupled planar illumination microscopy. *Neuron* **57**, 661–672 (2008).
7. Ahrens, M.B., Orger, M.B., Robson, D.N., Li, J.M. & Keller, P.J. Whole-brain functional imaging at cellular resolution using light-sheet microscopy. *Nat. Methods* **10**, 413–420 (2013).
8. Panier, T. *et al.* Fast functional imaging of multiple brain regions in intact zebrafish larvae using Selective Plane Illumination Microscopy. *Front. Neural Circuits* **7**, 65 (2013).
9. Portugues, R., Feierstein, C.E., Engert, F. & Orger, M.B. Whole-brain activity maps reveal stereotyped, distributed networks for visuomotor behavior. *Neuron* **81**, 1328–1343 (2014).
10. Alivisatos, A.P. *et al.* Neuroscience. The brain activity map. *Science* **339**, 1284–1285 (2013).

11. Briggman, K.L., Abarbanel, H. & Kristan, W.B. Optical imaging of neuronal populations during decision-making. *Science* **307**, 896–901 (2005).
12. Ahrens, M.B. *et al.* Brain-wide neuronal dynamics during motor adaptation in zebrafish. *Nature* **485**, 471–477 (2012).
13. Churchland, M.M., Yu, B.M., Sahani, M. & Shenoy, K.V. Techniques for extracting single-trial activity patterns from large-scale neural recordings. *Curr. Opin. Neurobiol.* **17**, 609–618 (2007).
14. Huber, D. *et al.* Multiple dynamic representations in the motor cortex during sensorimotor learning. *Nature* **484**, 473–478 (2012).
15. Dean, J. & Ghemawat, S. MapReduce: simplified data processing on large clusters. *Commun. ACM* **51.1**, 107–113 (2008).
16. Isard, M. *et al.* Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review* **41**, 59–72 (2007).
17. Shvachko, K., Kuang, H., Radia, S. & Chansler, R. The Hadoop Distributed File System. in *IEEE 26th Symposium on Mass Storage Systems and Technologies* 1–10 (2010).
18. Chu, C.-T. *et al.* Map-reduce for machine learning on multicore. *Adv. Neural Inf. Process. Syst.* **19**, 281 (2007).
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A. & Ma, J. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* 2–15 (2012).
20. Vladimirov, N. *et al.* Light-sheet functional imaging in behaving zebrafish. *Nat. Methods* http://www.nature.com/doifinder/10.1038/nmeth.3040 (27 July 2014).
21. Naumann, E.A., Kampff, A.R., Prober, D.A., Schier, A.F. & Engert, F. Monitoring neural activity with bioluminescence during natural behavior. *Nat. Neurosci.* **13**, 513–520 (2010).
22. Roweis, S. EM algorithms for PCA and SPCA. *Adv. Neural Inf. Process. Syst.* 626–632 (1998).
23. Sofroniew, N.J., Cohen, J.D., Lee, A.K. & Svoboda, K. Natural whisker-guided behavior by head-fixed mice in tactile virtual reality. *J. Neurosci.* **34**, 9537–9550 (2014).
24. Chen, T.-W. *et al.* Ultrasensitive fluorescent proteins for imaging neuronal activity. *Nature* **499**, 295–300 (2013).
25. Keller, G.B., Bonhoeffer, T. & Hübener, M. Sensorimotor mismatch signals in primary visual cortex of the behaving mouse. *Neuron* **74**, 809–815 (2012).
26. Niell, C.M. & Stryker, M.P. Modulation of visual responses by behavioral state in mouse visual cortex. *Neuron* **65**, 472–479 (2010).
27. Valmianski, I. *et al.* Automatic identification of fluorescently labeled brain cells for rapid functional imaging. *J. Neurophysiol.* **104**, 1803–1811 (2010).
28. Borst, A., Haag, J. & Reiff, D.F. Fly motion vision. *Annu. Rev. Neurosci.* **33**, 49–70 (2010).
29. Niell, C.M. & Stryker, M.P. Highly selective receptive fields in mouse visual cortex. *J. Neurosci.* **28**, 7520–7536 (2008).
30. Grama, A. & Engert, F. Direction selectivity in the larval zebrafish tectum is mediated by asymmetric inhibition. *Front. Neural Circuits* **6**, 59 (2012).
31. Nikolaou, N. *et al.* Parametric functional maps of visual inputs to the tectum. *Neuron* **76**, 317–324 (2012).
32. Gabriel, J.P., Trivedi, C.A., Maurer, C.M., Ryu, S. & Bollmann, J.H. Layer-specific targeting of direction-selective neurons in the zebrafish optic tectum. *Neuron* **76**, 1147–1160 (2012).
33. Del Bene, F. *et al.* Filtering of visual information in the tectum by an identified neural circuit. *Science* **330**, 669–673 (2010).
34. Kubo, F. *et al.* Functional architecture of an optic flow-responsive area that drives horizontal eye movements in zebrafish. *Neuron* **81**, 1344–1359 (2014).
35. Niell, C.M. & Smith, S.J. Functional imaging reveals rapid development of visual response properties in the zebrafish tectum. *Neuron* **45**, 941–951 (2005).
36. Dreosti, E., Vendrell Llopis, N., Carl, M., Yaksi, E. & Wilson, S.W. Left-right asymmetry is required for the habenulae to respond to both visual and olfactory stimuli. *Curr. Biol.* **24**, 440–445 (2014).
37. Akerboom, J. *et al.* Optimization of a GCaMP calcium indicator for neural activity imaging. *J. Neurosci.* **32**, 13819–13840 (2012).
38. Park, H.C. *et al.* Structural comparison of zebrafish Elav/Hu and their differential expressions during neurogenesis. *Neurosci. Lett.* **279**, 81–84 (2000).
39. Kimura, Y. *et al.* Hindbrain V2a neurons in the excitation of spinal locomotor circuits during zebrafish swimming. *Curr. Biol.* **23**, 843–849 (2013).
40. Kinkhabwala, A. *et al.* A structural and functional ground plan for neurons in the hindbrain of zebrafish. *Proc. Natl. Acad. Sci. USA* **108**, 1164–1169 (2011).
41. Fox, M.D. & Raichle, M.E. Spontaneous fluctuations in brain activity observed with functional magnetic resonance imaging. *Nat. Rev. Neurosci.* **8**, 700–711 (2007).
42. Hyvärinen, A. Fast and robust fixed-point algorithms for independent component analysis. *IEEE Trans. Neural Netw.* **10**, 626–634 (1999).
43. Satou, C. *et al.* Transgenic tools to characterize neuronal properties of discrete populations of zebrafish neurons. *Development* **140**, 3927–3931 (2013).
44. Bock, D.D. *et al.* Network anatomy and *in vivo* physiology of visual cortical neurons. *Nature* **471**, 177–182 (2011).
45. Briggman, K.L., Helmstaedter, M. & Denk, W. Wiring specificity in the direction-selectivity circuit of the retina. *Nature* **471**, 183–188 (2011).
46. Oberlaender, M. *et al.* Automated three-dimensional detection and counting of neuron somata. *J. Neurosci. Methods* **180**, 147–160 (2009).
47. Mukamel, E.A., Nimmerjahn, A. & Schnitzer, M.J. Automated analysis of cellular signals from large-scale calcium imaging data. *Neuron* **63**, 747–760 (2009).
48. Pillow, J.W. *et al.* Spatio-temporal correlations and visual signalling in a complete neuronal population. *Nature* **454**, 995–999 (2008).
49. Paninski, L. *et al.* A new look at state-space models for neural data. *J. Comput. Neurosci.* **29**, 107–126 (2010).

ONLINE METHODS

Summary. Details of the Thunder library, and of the reported example analyses and experiments, are provided here. Additional material is available at a webpage about Thunder (http://freeman-lab.github.io/thunder/) and a web page highlighting example analyses and providing access to example data (http://research.janelia.org/zebrafish).

Deployment options. Spark can be deployed in 'standalone mode' on a private cluster or on Amazon's EC2 cloud computing services; it can also be deployed on top of the Apache Mesos or Hadoop YARN resource managers, but we do not consider those use cases here.

Private cluster. For private cluster usage, we deployed Spark in standalone mode on a general purpose compute cluster scheduled by Univa grid engine (UGE), containing 256 nodes running Scientific Linux 6.3. Here we provide details on the deployment process and how we integrated Spark with the UGE scheduler, so administrators of other private research clusters can replicate our work. A custom qsub job class was created so that Spark could be spun up dynamically on a user-designated quantity of nodes. After the qsub is issued, with the number of nodes specified, the grid engine scheduler designates a set of nodes that will be used to run the Spark job. Once all of the nodes are available, a driver (master) is started on one of them (using the start-master.sh script included with the Spark distribution), then sends that driver the addresses of the remaining nodes. The driver then uses the start-all.sh script to start those workers. After starting a job, a user can ssh into the driver and run Spark. When finished, the user exits the ssh session and issues a qdel to stop the Spark job. This triggers the stop-all.sh script on the driver, which takes down the Spark cluster in an organized fashion. For accessing data, Spark is often run on top of a Hadoop Distributed File System, but it can also access data directly from a networked-filed system, so long as it is available to all the nodes. We used a NFS served by several EMC Isilon clusters.

EC2. We created a custom EC2 launch script that extends the spark-ec2.py script (included with the Spark distribution) for launching an EC2 cluster in standalone mode with the desired number of nodes and instance types as well as several other configuration options. The launch scripts deploy Spark on a cluster, create an HDFS and preconfigure the cluster by installing Thunder and its dependencies. Once the cluster has launched, it can be logged into, and analyses can be performed.

Data format. All analyses in Thunder operate on an RDD of records, where each record is a (key, value) tuple, where key is either an integer or a tuple of integers and value is a numpy array. Only a subset of analyses use keys explicitly, but all analyses assume this format for consistency. Additionally, some saving operations (see below) use keys to format results. These key-value records can, in principle, be stored in a variety of Spark-accessible formats in a cluster-accessible file system (for examplet, HDFS or NFS). The core functionality of Thunder does not depend on the file format, only that raw data are parsed appropriately into an RDD of key-value records. The primary loading method we used assumes a text file input, where the rows are neural channels and the columns are the keys and values, each number separated by a space (with the first three numbers providing the *x*-*y*-*z* key in the case of imaging data). Different planes are stored in separate files, but this is optional. Future work can make use of alternative and more efficient file formats, for example, flat binary files. For analyses that require covariates (for example, behavioral or stimulus variables), they can be provided directly as numpy arrays (if working entirely in Python), or alternatively loaded from text files or MAT files.

Components. Thunder currently includes four core packages: clustering, factorization, regression and time series statistics, as well as an io package for loading and saving, and a util package with common utilities. Here we describe the key components of each package and provide examples of code. More detailed code examples are available in the **Supplementary Protocol** and online (http://freeman-lab.github.io/thunder/ and http://research.janelia.org/zebrafish).

Loading. Thunder's load function takes raw data (as described above) and parses it into an RDD of (key, value) tuples. The number of keys can be specified (for example, 3 keys for *x*-*y*-*z* coordinates in imaging data, 1 key for simple indexing of channels). There are also a variety of preprocessing options available, including mean subtraction and conversion to $\Delta F/F$ (by subtracting and dividing by a baseline). These are easily extended by writing Python functions that accept a time series as input and return the preprocessed time series. We also include in the load package convenience functions for determining data set dimensionality (from the keys) through a single reduce operation, and switching between coordinate-wise and linear key indexing.

Saving. Results of analyses (for example, images, time series) can be returned directly in the form of numpy arrays. If working in Python, these can be examined or visualized immediately (for example, using matplotlib). For external compatibility, we provide the ability to save to alternative formats, including MAT files, images and text. The save function takes an RDD as input and collects the results followed by writing to disk. In the case of MAT files or images, if requested, the dimensions of the data are derived from the keys, and outputs are automatically reshaped (for example, into a two-dimensional image or a three-dimensional volume).

Factorization. Factorization methods separate or decompose a data matrix into smaller, lower-rank matrices by optimizing an objective function. PCA finds a decomposition that minimizes the squared error between the true data matrix and the matrix reconstructed from the low-rank matrices. We implement PCA through the singular value decomposition (SVD), and provide two implementations of the SVD. The first is suitable for a 'tall and skinny' matrix $\mathbf{Y}$ of size $n \times d$, where n is the number of channels and d is the dimensionality (for example, number of time points), $n >> d$, and d^2 fits comfortably in the memory of a single machine. This algorithm first computes a $d \times d$ gramian matrix using a distributed implementation of the equation

$$\text{gramian} = \sum_{i=1}^{n} \mathbf{y}_i^T \mathbf{y}_i$$

where $\mathbf{y}_i$ is the mean-subtracted response time course of channel i. A local eigenvector decomposition of the resulting matrix provides the right singular vectors (a small $d \times k$ matrix), and another distributed operation computes the left singular vectors, only for the desired number of vectors. In Thunder, if data is an RDD of (key, value) pairs, with the response time course of each channel as the value, this sequence of operations can be expressed as

```
mat = RowMatrix(data)
cov = mat.gramian() / mat.nrows
eigw, eigv = eigh(cov)
inds = argsort(eigw)[::-1]
S = sqrt(eigw[inds[0:k]]) * sqrt(n)
V = eigv[:, inds[0:k]].T
U = mat.times(V.T / S)
```

where S and V are small arrays represented locally, and U is another RDD of n (key, value) pairs, with the projection of each data point into the low-dimensional space. The initial step creates a RowMatrix (a class from the util package). A RowMatrix is backed by an RDD of (key, numpy array) pairs, and a variety of common matrix operations can be performed on it using distributed operations. For example, the gramian method computes the outer product of a matrix with itself, which can be expressed as a map-reduce operation:

```
mat.rdd.map(lambda (k, y): outer(y, y)).reduce(add)
```

where the map step calculates the outer product of each array with itself, and the reduce step uses the addition operator, which is commutative and associative, to add these outer products together. (In practice, this step can be made more efficient with methods that support in-place addition, for example, an accumulator, or aggregator, but the simpler version is given here for clarity). The times method uses a map to multiply a large matrix (represented by an RDD) with a smaller one:

```
mat.rdd.map(lambda (k, y): (k, inner(y, other)))
```

To compute the SVD when d is large (for example, >1,000), an iterative computation based on expectation maximization[22] is more efficient, especially when only a small number of eigenvectors are required, because in Spark iterations do not require reloading data from disk. The principal subspace is estimated by iterating the following two update equations:

$$\mathbf{X} = (\mathbf{C}^T\mathbf{C})^{-1}\mathbf{C}^T\mathbf{Y}$$

$$\mathbf{C}_{new} = \mathbf{Y}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}$$

These steps can be implemented using the following sequence of two distributed matrix operations (where times and gramian are as described above),

```
iter = 0
C = random.rand(k, d)
while (iter < maxiter) & (error > tol):
        C_inv = dot(C.T, inv(dot(C, C.T)))
        YY = mat.times(C_inv).gramian()
        YY_inv = inv(YY)
        CY = dot(C_inv, YY_inv)
        C = mat.rows().map(lambda y: outer(y, dot(y, CY))).
        reduce(add)
        C = C.T
```

where mat is a RowMatrix containing the data. After convergence, the rows of the $k \times d$ matrix C span the subspace of the leading eigenvectors. Projecting the raw data into this (much smaller) subspace and repeating the gramian calculation (and subsequent map step) described above yields an orthonormal basis (the singular vectors). Options for PCA (and for the SVD) include the number of recovered singular vectors, which algorithm to use, and tolerance and maximum number of iterations for the iterative algorithm.

The source model for PCA is that the data is Gaussian-distributed with a low-rank covariance determined by the principal components. ICA, in contrast, assumes that the components are non-Gaussian and statistically independent from one another, and estimates them by iteratively maximizing an objective function that computes the non-Gaussianity of the components. As is common, Thunder's ICA first reduces the dimensionality of the data and whitens it using SVD. The analysis then uses an iterative sequence of map and reduce operations (as in the iterative SVD algorithm) to compute an unmixing matrix via a widely used fixed-point algorithm[42] (see mathematical details below). Options for ICA include the number of eigenvectors to retain, the number of independent components to estimate, maximum number of iterations, and tolerance.

Regression. Regression analyses, sometimes called 'mass-univariate', describe the time series from each channel as a function of some underlying covariates. These analyses are implemented through a map operation (to perform the regression) followed by a collect. Classes are used to separate the underlying regression functions from the map operation, so they can more easily be extended or modified. For example, the LinearRegressionModel class has the following get method, which performs a regression on a single time series:

```
def get(self, y):
        b = dot(self.X_pinv, y)
        predic = dot(b, self.X)
        resid = y - predic
        sse = sum((predic - y) ** 2)
        sst = sum((y - mean(y)) ** 2)
        r2 = 1 - sse / sst
        return b, r2, resid
```

where X is the design matrix and X_pinv is its pseudoinverse. The distributed operation itself is implemented through a fit method that uses a map to apply the regression to each time series,

```
def fit(self, data):
        result = data.mapValues(lambda y: self.get(y))
```

Thunder includes both linear and bilinear regression. "linear" regression fits each time series as a linear combination of some variables, plus a constant. "bilinear" uses two design matrices;

it uses the first to estimate a basis function that is common to each of several conditions, and uses the second to describe each condition as a scaled version of that basis function (similar to the GLM approach commonly used in fMRI data analysis[50]). These analyses compute the regression coefficients, an R^2 for each voxel, and residuals from the fitted model. These values can easily be passed to other algorithms. For example, a large number of regression coefficients per data point can be further simplified using PCA or *k*-means. Tuning analyses estimate the parameters of a tuning curve that relates an input value to a response. Thunder includes both Gaussian and circular tuning curves, in each case yielding a parameter for the central tendency and the dispersion. The user can provide an input value associated with each data point, or use a regression matrix to estimate a coefficient for each of a smaller set of input values, and then compute tuning from those coefficients.

Clustering. Clustering analyses try to associate data points with particular categories based on a distance metric. The included algorithm, *k*-means, assigns each data point to one of *k* clusters using an iterative algorithm. The implementation of the algorithm proceeds in two primary steps. The first step uses a map operation to compute, for each record, the closest of the current *k* cluster centers. The second step uses a reduceByKey (followed by a map) to identify the average of each cluster based on the current assignment. The cluster centers are then updated, and the procedure is repeated until convergence. Parameters include the number of cluster centers, the maximum number of iterations, and the tolerance for convergence.

Time series. Time series analyses compute statistics on each channel, either simple summary statistics, or more complex time series calculations. As with regression, most of these operations are implemented through a single map and collect, with class-specific methods that handle specific time series opera tions. Stats computes simple summary statistics, including mean, median, std, or norm. Fourier estimates the statistics of the Fourier transform, specifically, the coherence and phase of the time series at a specified frequency (coherence is Fourier amplitude normalized by the amplitude of all frequencies, yielding a number between 0 and 1 that is more interpretable than raw amplitude.) CrossCor computes the cross correlation between data and a signal of interest at multiple time lags. If the maximum lag is large, the resulting cross-correlation functions can be further simplified using PCA or *k*-means. LocalCorr estimates the correlation between each time series and the average time series of its neighbors, useful for detecting structures with reliable activity independent of any covariates. Query extracts local averages from spatio-temporal data. The user specifies a set of indices to include in the average, corresponding to a set of linearized coordinates.

Data sets for benchmarking. Four different data sets were used for benchmarking. Data were represented as floats; this is assumed for all sizes reported in all benchmarks, and yields sizes that correspond closely to the size of data in RAM after loading, as provided by Spark's performance monitoring metrics. On disk, data were stored as unsigned integers, and thus had smaller file sizes.The first data set (**Fig. 2a**,**d**,**e**) was two-photon imaging data, with dimensions of 512 × 512 × 4 (*x*-*y*-*z*) × 6300 (time), and size of approximately 50 GB. The second (**Fig. 2b**,**e**) was light-sheet imaging data, with dimensions of 2,048 × 1,024 × 18 (*x*-*y*-*z*) × 900 (time), and size of approximately 250 GB. A different light-sheet imaging data set was used to test a wider range of sizes (**Fig. 2c**,**f**); these data had *x*-*y* dimensions of 2,048 × 1,024, with 1,500 time points, and the number of planes was varied from 2 to 32 to adjust the total size from ~100 to 1,600 GB (sizes in **Fig. 2f** were rounded to the nearest 100 GB). A final light-sheet imaging data set was used to report comparisons between the iterative and direct SVD for different numbers of time points; these data had *x*-*y*-*z* dimensions of 972 × 2,048 × 40, and the number of time points was either 240 or 1200 (with sizes of 150 and 750 GB).

Cluster environments for benchmarking. Tests were run using Spark 0.9.1 and its associated deploy scripts for EC2 tests (which used the Amazon Machine Image ami-35b1885c). Benchmarks for the private cluster used Spark running on a general purpose high performance compute cluster at the Janelia Farm Research Campus, containing 256 nodes running Scientific Linux 6.3 and scheduled by Univa Grid Engine. The nodes are equipped with 16 cores (2.7 GHz Intel Sandy Bridge, E5-2680), 128 GB RAM, 10 Gb ethernet, and single hard drives. The environment also contains a DDN GridScaler running GPFS and several EMC Isilon clusters serving NFS to the compute cluster (see above for how Spark was deployed). Benchmarks on Amazon EC2 (**Fig. 2e**) were run using a 20 or 40 node cluster (including the driver) with the cc2.8xlarge ('compute optimized') instance type, each node having 32 vCPUs (Intel Sandy Bridge, E5-2670), 60.5 GB RAM, and 4 × 840 GB storage, at a cost of $2.00 per hour per node. This instance type was chosen to match the capabilities of the private cluster; EC2 cores use hyperthreading whereas the cores of the private cluster do not, so the choice of 32 vCPUS meant that the two clusters were matched for numbers of actual cores. In an additional test reported above, the m3.2xlarge ('general purpose') instance type was used, each node having 8 vCPUs (Intel Sandy Bridge, E5-2670), 30 GB RAM, and 2 × 80 GB drives, at a cost of $0.56 per hour per node. EC2 clusters were launched with Thunder preinstalled using a custom launch script (see above). The cluster ran HDFS, and raw data were transferred from Amazon's S3 into HDFS using a distributed copy before analyses (taking approximately 5 min for the 50 GB data set with 20 nodes, and 10 min for the 250 GB data set with 40 nodes). Number of input partitions when running on EC2 was set manually to match the automatic partitioning determined when running on the private cluster.

RAM configuration for benchmarking. For private cluster usage each node had 16 cores and 128 GB RAM. For cc2.8xlarge ('compute-optimized') EC2 instances, each node had 32 vCPUs and 60.5 GB RAM. For m3.2xlarge ('general purpose') EC2 instances, each node had 8 vCPUs and 30 GB RAM. On the private cluster, 100 GB were made available to Spark; on EC2, 53.3 or 26.2 GB; the remaining memory was allocated to the operating system. In both environments, 60% of Java heap space was used for Spark's memory cache (the default), so total RAM available for data caching per node was approximately 60 GB, 32 GB and 16 GB, respectively.

File formats for benchmarking. Data were stored as integers. For cluster-based analyses with Thunder, data were stored as text files containing key-value records, and separate planes were stored in separate files. For analysis on a workstation in Matlab, data were saved to MAT files as matrices of unsigned integers, with rows as voxels and columns as time points, also with one file per plane.

Benchmarking test details. Code for performing all Thunder benchmarks are included in the helper functions thundertestrunner.py and thunderdatatest.py. Important details of the tests, and how they were performed, are provided here. Cached data were used for the tests reported in **Figure 2a–c**. This means that data were initially loaded and parsed (using Thunder's load function) and cached, and a count() operation was performed to force computation and cache the data. Run times were measured for performing the corresponding analysis, using count to force the computation. In the case of iterative computations, five iterations were used. For PCA and ICA, 3 components were estimated, and for *k*-means, 3 clusters were used. No results were collected to the driver or saved to disk. For the tests reported in **Figure 2d** comparing times with and without caching, the tests without caching were run by simply never caching, thus requiring that data were loaded and parsed on every iteration. The tests with caching included the time to load and cache on the first iteration only. For the tests reported in **Figure 2e**, different stages were timed separately. For loading and parsing, time was computed to complete a count operation on the data loaded and parsed with the load function, which incorporates costs due to file IO as well as file content parsing and object creation. The compute time was the time to perform the computation, as induced by a count(), on cached data, as reported in **Figure 2a**,**b**. The saving time was the time to collect a single number per channel from a cached data set to the driver and save the result to disk in MAT format. In normal use, the total time to perform all operations would be slightly lower than the times reported in **Figure 2e** because some components would be executed at the same time. For all tests in **Figure 2a**,**b**,**e**, run times reported are the minimum of three successive runs. For the remaining tests, a single run time was computed.

Workstation comparison. For the single workstation implementation, planes of data were loaded sequentially from MAT files. For each plane, to avoid performing computations on the entire plane (which was impossible due to memory usage), subsets of each plane were analyzed in a loop, with parallelization enabled by parfor from the Matlab Parallel Computing toolbox. For regression, matrix computations were used to estimate the same parameters as in the Thunder implementation for every voxel. For ICA, the appropriate matrix computations were performed to compute and maintain partial updates to the component matrix **B** (see above) using only the data from that plane. In Thunder, results were collected to the driver, but no results were saved for either implementation. For the iterative analysis, 10 iterations were used. For convenience, total run times were extrapolated where appropriate (for example computing the run time on one plane or one iteration, and then multiplying by the total number). The validity of these extrapolations were confirmed in a small number of test cases.

Mouse two-photon imaging. All experiments were conducted according to protocols approved by the Institutional Animal Care and Use Committee of the Howard Hughes Medical Institute, Janelia Farm Research Campus. Mouse surgeries and imaging were performed in a manner similar to previous reports[14]. Male adult (>3 month) C57BL/6Crl mice (Jackson Labs: 000664) were infected with the genetically encoded calcium indicator GCaMP6s[24] under control of the human synapsin-1 promoter using the recombinant adeno-associated virus (serotype 2/1; produced by the University of Pennsylvania Gene Therapy Program Vector Core). A 3 mm window was inserted in a craniotomy made over left barrel cortex (3.4 mm lateral and 1.5 mm posterior to Bregma). A titanium headpost was attached to the skull to permit head fixation. Experiments were performed 2 to 4 weeks post-infection. GCaMP6s was excited at 1,000 nm with a Ti-Sapphire laser using a custom two-photon microscope with a resonant scanner and fast z-focusing. Images were acquired continuously at 7.5 Hz using ScanImage software[51], with dimensions of 512 pixels × 512 pixels × 4 planes.

Mouse behavior apparatus. Head-fixed mice ran on a spherical treadmill in a tactile virtual reality environment as described elsewhere[23]. Briefly, the position of two motorized walls was coupled to the movement of the ball in closed-loop, allowing for the simulation of a winding corridor. Wall position was not considered in the analyses reported here, only running speed. Instantaneous speed was derived by tracking motion of the ball at 500 Hz, and smoothed with a Savitzky–Golay filter with window of 400 ms. The entire apparatus was mounted underneath the two-photon microscope. Typical experiments lasted 30–60 min, generating ~50–100 GB of imaging data per experiment (stored as unsigned integers).

Zebrafish animals and trangenesis. All experiments were conducted according to protocols approved by the Institutional Animal Care and Use Committee of the Howard Hughes Medical Institute, Janelia Farm Research Campus. Zebrafish lines used in this study were Tg(*elavl3:GCaMP5G*) previously described[7,37,38] in the *nacre* background[52], and a nuclear-localized GCaMP6s line *Tg(elavl3:H2B-GCaMP6s)*^{jf5} in the *casper* background[53]. The nuclear-localized GCaMP6s fish was generated using the Tol2 transposon system[54], in which the calcium indicator *GCaMP6s*[24], was fused with sequence encoding human histone H2B at the 5′ end to restrict its expression to the nucleus[55]. The gene encoding the H2B-GCaMP6s fusion was subcloned into a Tol2 vector that contained zebrafish *elavl3* promoter[56]. The transgene construct and transposase RNA were injected into 1–4-cell-stage embryos, and the transgenic lines were isolated by the high expression of bright green fluorescence in the central nervous system in the next generation[57]. Embryos were reared according to standard protocols at 28.5 °C (ref. 58), and experiments were performed on animals 5–7 d post fertilization at room temperature. Fish lines and DNA constructs for *elavl3*:GCaMP6s and *elavl3*:H2B-GCaMP6s are available upon request.

Zebrafish imaging. Larval zebrafish were paralyzed by brief immersion into 1 mg/ml alpha-bungarotoxin solution, and embedded in a custom-made chamber[20]. A custom light-sheet microscope, described in detail elsewhere[20], was used to image GCaMP fluorescence while larval zebrafish were placed horizontally and intended motor output recorded electrically and visual stimuli presented from below. Most of the optics were similar to those described in ref. 7. The key differences[20] were as follows: the detection arm was rotated 90 degrees to vertical orientation, so that the zebrafish could be positioned horizontally, an orthogonal excitation arm was added to scan the tissue between the eyes, and the side excitation arm was dynamically switched off when positioned over the eye to avoid laser stimulation of the eyes. Volumes of images were acquired continuously at 1 Hz (800 ms imaging time and 200 ms piezo reset time), and streamed in binary format to a RAID6 disk array in real time, using a custom LabView (64-bit) program Zebrascope (Coleman Technologies). Image stack dimensions were typically 2,048 × 1,024 pixels (*x*-*y*) by 41 planes (*z*). Typical experiments lasted ~15–30 min, generating ~250–500 GB of imaging data per experiment (stored as unsigned integers). The data reported in **Figure 7** were identical to those reported previously[7], and thereby differed in the imaging details: zebrafish were embedded in a column of agarose and positioned vertically in a light-sheet microscope with only one excitation arm, and eyes were not excluded from stimulation.

Zebrafish stimuli and behavior, general. A diffusive plastic screen was attached to the bottom of the chamber, to allow red monochrome images to be projected from below using a mini projector. Electrical recordings from the ventral root of the spinal cord, or motor neuron axons, were made and processed according to previously described techniques[12,59]. For comparisons to neural data (for example, **Figs. 5** and **6**) electrophysiological signals were downsampled by taking the maximum signal within each temporal window of volume acquisition.

Direction selectivity. Moving gratings of spatial period 1 cm were presented at 12 angular directions, 30° apart, in clockwise sequence. The gratings moved at 1 cm/s for a period of 10 s, after which a gray screen appeared for 10 s. For the data shown in **Figure 4**, the sequence of 12 angles was repeated 5 times (in identical order). For the data shown in **Figure 6b**, the sequence was repeated 8 times.

Optomotor response. A red/black grating of spatial period 1.2 cm moving forward, from head to tail, at a speed of 0.24 cm/s was presented to the fish, alternating between periods of motion and static grating. For the data shown in **Figure 5a**,**b**, the period was 40 s (20 s moving, 20 s static), and the sequence was repeated 9 times. For the data shown in **Figure 6a**, the period was 20 s (10 s moving, 10 s static), and the sequence was repeated 14 times. During visual motion, vigorous fictive swimming was observed, while during no motion, the fish swam little.

Self-driven swimming. A red/black grating of period 1.2 cm moving forward, from tail to head, at a speed of 0.05 cm/s was presented continuously to the fish, while the entire hindbrain and parts of the midbrain were imaged using one light sheet from the side. Motor activity was monitored as explained above. Stimulus speed was titrated online to elicit motor output ~50% of the time.

Analysis details. Here we detail the computations and algorithms underlying the example analyses presented in **Figures 3**–**7**. Each was implemented using routines in Thunder (separately or combined), but below we focus on the analyses themselves, rather than implementation (**Supplementary Protocol**).

Motion correction. For all analyses and data sets, sample motion in raw imaging data was corrected using plane-by-plane cross-correlation based image registration. Specifically, for every plane, the two-dimensional Fourier transform was used to compute the cross-correlation between the image at time t_n and the image at t_0, and the peak of the cross-correlation was used to determine an optimal displacement in x and y, which was then applied to the image at t_n. For the mouse two-photon imaging data, an additional line-by-line motion compensation procedure was performed, as described elsewhere[14]. Image registration was performed using Matlab; all other computations were performed in Spark and Thunder.

Preprocessing. The input to analyses was the fluorescence of every voxel from motion-registered images. As described above, these data were represented and stored as records, each a key-value pair, where the key is the *x*-*y*-*z* coordinate of each voxel, and the value is an array with the response time series. Preprocessing of the time series was handled by Thunder's load function. For all example analyses (except for local correlation, which used raw fluorescence), fluorescence for every voxel was converted into $\Delta F/F$ (yielding what is hereafter referred to as a response time course) as:

$$y(t) = (F(t) - \bar{F})/(\bar{F} + c)$$

Where $F(t)$ is the fluorescence image at time t for each pixel, $\bar{F}$ is the mean, and a small constant c prevents division by 0.

Speed tuning. Speed tuning was characterized using regression followed by tuning curve estimation. Regression was used to estimate, for every voxel, the relative responses to different speeds. The instantaneous speed was binned into 12 equally spaced bins, yielding a binary matrix $\mathbf{X}_1$ of dimension $12 \times t$ describing the speed at each moment in time (where t is the duration of the experiment). Least-squares was used to estimate the response to each speed:

$$\mathbf{b} = (\mathbf{X}_1\mathbf{X}_1^T)^{-1}\mathbf{X}_1\mathbf{y}$$

where $\mathbf{y}$ is the response time course of a single voxel. R^2 was computed from this linear fit as a measure of response predictability. The speed-specific responses were then fit with a Gaussian tuning curve. Rather than perform a nonlinear fit, the best-fitting parameters were approximated by computing a weighted mean and variance. A composite map (**Fig. 3c**) was generated by using the tuning parameter (the weighted mean) as the hue and the R^2 as the brightness. In Thunder, these analyses were implemented using the RegressionModel and TuningModel classes from the regression package.

Local correlations. As a covariate-independent measure of response reliability, local correlations[60] were computed for every voxel by correlating the fluorescence time course of each voxel with the average of a local 7 × 7 pixel neighborhood:

$$localcorr_{i,j} = \frac{\mathbf{y}_{i,j}}{\|\mathbf{y}_{i,j}\|} \bullet \frac{\bar{\mathbf{y}}_{i,j}}{\|\bar{\mathbf{y}}_{i,j}\|}$$

where $\mathbf{y}_{i,j}$ is the response of a voxel at position i,j, and $\bar{\mathbf{y}}_{ij}$ is the average response of the neighborhood defined by the interval

$(i-3, i+3)$ and the interval $(j-3, j+3)$, $\|\ \|$ denotes the vector norm. In Thunder, this analysis was implemented using the LocalCorr class from the timeseries package.

Direction tuning. Direction selectivity was characterized using regression followed by tuning curve estimation, except here additionally incorporating the temporal profile of the response. Responses were first averaged across repeated presentations. Regression was used to estimate, for every voxel, a single temporal basis function, and a scalar associated with each presented direction. Let $\mathbf{X}_1$ be the $s \times t$ binary design matrix specifying the timing of stimulus presentation, where s is the number of time points per stimulus, and t is the total duration ($t = sd$ where d is the number of directions). Let $\mathbf{X}_2$ be the $d \times t$ design matrix specifying which of the d directions were presented at each time point. For every voxel, least-squares was used to estimate the average temporal response profile pooled across directions.

$$\mathbf{b} = (\mathbf{X}_1\mathbf{X}_1^T)^{-1}\mathbf{X}_1\mathbf{y}$$

where $\mathbf{y}$ is the response time course of a single voxel. For non-overlapping conditions, this is identical to computing the average time course to stimulus presentation (ignoring direction). The voxel-specific temporal profile $\mathbf{b}$ was then used to construct, for each voxel, a new design matrix through element-wise multiplication with the nonzero elements of $\mathbf{X}_2$, and used for a second least-squares procedure to estimate the direction-specific weights. Finally, the direction-specific weights were fit with a circular Von Mises function. Rather than perform a nonlinear fit, the parameters of the best-fitting Von Mises were estimated directly by computing the circular mean and circular variance of the directions weighted by the responses[61]. This yielded, for each voxel, a preferred direction and a tuning width. Response strength was computed as the norm of the response time course. The resulting three numbers determined the hue, saturation, and brightness (in HSV color space) of a composite map (**Fig. 4b**). Contrast was enhanced by scaling and thresholding the brightness map. In Thunder, this analysis was performed using the RegressionModel and TuningModel classes from the regression package.

Principal component analysis. Dimensionality reduction was used to characterize the dynamics of neural activity in both space (**Fig. 4**) and time (**Fig. 5**). The input to the analysis was the $n \times t$ matrix $\mathbf{Y}$, containing the response time course of every voxel to each of several repeated presentations of the stimulus (n is the number of voxels; t is the number of time points). To identify a low-dimensional subspace for the data, repeated presentations were first averaged to yield a $n \times \hat{t}$ matrix $\hat{\mathbf{Y}}$, where $\hat{t}$ is the duration of a single stimulus presentation (the same analysis can be performed on the full, non-averaged data matrix). Row means were subtracted, and the singular value decomposition (SVD) was computed:

$$\hat{\mathbf{Y}} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

The principal components (the columns of $\mathbf{V}$) are vectors of length $\hat{t}$ describing the time course of each component, and the scores (the columns of $\mathbf{U}$) are vectors of length n, describing the projection of each voxel on the direction given by the corresponding component, forming the projections onto the volume, i.e., whole-brain maps. As described above, Thunder provides two-large scale implementations of the SVD, both in the factorization package. In either case, scores were used to generate response maps (for example, **Fig. 5a**,**c**). Specifically, for each voxel, the first two scores were converted from Cartesian to polar coordinates; the angle was used to determine the hue, and the magnitude was used to determine the brightness (the saturation was set to 1). Every angle corresponds to a different linear combination of the first two components; the time courses of each of these linear combinations were used to construct a 'color wheel' (**Fig. 5a**) mapping color to time course shape. As for direction selectivity maps, brightness was scaled and thresholded to increase contrast.

For trial-averaged data, the principal components correspond to trial-averaged trajectories in time through a state-space. To recover trial-by-trial trajectories through the same space (as shown in **Fig. 6**), regression followed by PCA was used to recover the principal subspace, and the complete time courses were then projected into the principal subspace,

$$\mathbf{J} = \mathbf{U}^T\mathbf{Y}$$

yielding a $k \times t$ matrix of trajectories $\mathbf{J}$, where k is the dimensionality of the subspace (either 2 or 3). These analyses were performed using the entire data set (i.e., including the responses of all recorded neurons). For the optomotor response (**Fig. 6a**), the initial regression was simply trial-averaging; for direction selectivity (**Fig. 6b**), the initial regression marginalized across time the responses to the different directions. State space trajectories were smoothed with linear interpolation (separately for each trial), for visualization only (**Fig. 6**). In Thunder, these analyses were implemented using a combination of the RegressionModel and PCA classes from the regression and factorization packages.

Cross-correlation. Cross-correlation followed by dimensionality reduction (as above) was used to relate neuronal dynamics to swimming behavior. First, electrophysiological recordings were filtered using a running s.d.[12], then downsampled to the frequency of the volumetric brain scans by computing the maximum within the temporal window during which each imaging volume was acquired. Cross-correlations (with a maximum lag of $l = 17$) were computed between each voxel's mean-subtracted response time course and this downsampled signal. Specifically, the cross-correlation for voxel n at a lag of m was given by

$$crosscorr_{m,n} = \frac{\mathbf{y}_n}{\|\mathbf{y}_n\|} \bullet \frac{\mathbf{e}_m}{\|\mathbf{e}_m\|}$$

where $\mathbf{y}_n$ is the response time course of voxel n, $\mathbf{e}_m$ is the electrophysiological signal shifted in time by m (and padded with 0 s). The resulting $n \times (2l + 1)$ collection of data was used as input to PCA to embed the ensemble of cross-correlation kernels in a low-dimensional space; visualization of this space (**Fig. 4c**,**d**) was performed as described above (**Fig. 4a**,**b**). In Thunder, this analysis was performed using the CrossCorr class from the timeseries package and the PCA class from the factorization package for dimensionality reduction.

Independent component analysis. ICA is an unsupervised learning algorithm that aims to decompose a multivariate data set into underlying, additive signals, assuming they are non-Gaussian and statistically independent. The generative model is:

$$\mathbf{Y} = \mathbf{PA}$$

where $\mathbf{Y}$ is the $n \times t$ data matrix, $\mathbf{P}$ is the $n \times c$ matrix of source signals or components (columns are spatial modes), and $\mathbf{A}$ is the $c \times t$ matrix of mixing coefficients (rows are temporal modes). ICA seeks a matrix $\mathbf{W}$ that unmixes the data

$$\mathbf{P} = \mathbf{YW}$$

such that a function of the non-Gaussianity of the spatial sources is maximized. Before estimating this matrix, data are typically preprocessed through whitening and dimensionality reduction[42,47]. Whitening was implemented by computing an SVD on the data matrix $\mathbf{Y}$ as described above, and projecting the response time course of each voxel onto the leading eigenvectors and scaling by inverse eigenvalues:

$$\hat{\mathbf{Y}} = \mathbf{YVS}^{-1}$$

yielding the whitened, lower-dimensional $n \times k$ data matrix $\hat{\mathbf{Y}}$, where $\mathbf{V}$ and $\mathbf{S}$ are restricted to the desired number of leading eigenvectors k. The unmixing matrix $\mathbf{B}$ in the space of the whitened data was estimated using a fixed-point algorithm[42], iteratively applying the update equation:

$$\mathbf{B}_{new} = (\hat{\mathbf{Y}}(\hat{\mathbf{Y}}\mathbf{B})^3)/n - 3\mathbf{B}$$

(Raising to the power of 3 is element-wise.) On each iteration, $\mathbf{B}$ was additionally orthogonalized:

$$\mathbf{B}_{orth} = \mathbf{B}(\mathbf{B}^T\mathbf{B})^{-1/2}$$

These updates were repeated until convergence. For the analysis shown in **Figure 7**, k = 100 dimensions and c = 20 components were used, and ~20 iterations were required. Finally, the mixing and demixing matrices were transformed to the original data space by undoing the whitening transform

$$\mathbf{W} = \mathbf{VS}^{-1}\mathbf{B}$$
$$\mathbf{A} = \mathbf{B}^T\mathbf{SV}^T$$

The matrix $\mathbf{A}$ contains in its rows the mixing coefficients—the time courses associated with each spatial independent component—and the spatial components are given by $\mathbf{YW}$. In Thunder, this analysis was performed using ica from the factorization package, which makes use of svd for initial dimensionality reduction.

50. Dale, A.M. Optimal experimental design for event-related fMRI. *Hum. Brain Mapp.* **8**, 109–114 (1999).
51. Pologruto, T.A., Sabatini, B.L. & Svoboda, K. ScanImage: flexible software for operating laser scanning microscopes. *Biomed. Eng. Online* **2**, 13 (2003).
52. Lister, J.A., Robertson, C.P., Lepage, T., Johnson, S.L. & Raible, D.W. nacre encodes a zebrafish microphthalmia-related protein that regulates neural-crest-derived pigment cell fate. *Development* **126**, 3757–3767 (1999).
53. White, R.M. *et al.* Transparent adult zebrafish as a tool for *in vivo* transplantation analysis. *Cell Stem Cell* **2**, 183–189 (2008).
54. Urasaki, A., Morvan, G. & Kawakami, K. Functional dissection of the Tol2 transposable element identified the minimal cis-sequence and a highly repetitive sequence in the subterminal region essential for transposition. *Genetics* **174**, 639–649 (2006).
55. Kanda, T., Sullivan, K.F. & Wahl, G.M. Histone-GFP fusion protein enables sensitive analysis of chromosome dynamics in living mammalian cells. *Curr. Biol.* **8**, 377–385 (1998).
56. Sato, T., Takahoko, M. & Okamoto, H. HuC:Kaede, a useful tool to label neural morphologies in networks in vivo. *Genesis* **44**, 136–142 (2006).
57. Fisher, S., Grice, E.A., Vinton, R.M. & Bessling, S.L. Evaluating the biological relevance of putative enhancers using Tol2 transposon-mediated transgenesis in zebrafish. *Nat. Protoc.* **1**, 1297–1305 (2006).
58. Westerfield, M. *The Zebrafish* (University of Oregon Press, 1993).
59. Ahrens, M.B., Huang, K.H., Narayan, S., Mensh, B.D. & Engert, F. Two-photon calcium imaging during fictive navigation in virtual environments. *Front. Neural Circuits* **7**, 104 (2013).
60. Cheng, A., Gonçalves, J.T., Golshani, P., Arisaka, K. & Portera-Cailliau, C. Simultaneous two-photon calcium imaging at different depths with spatiotemporal multiplexing. *Nat. Methods* **8**, 139–142 (2011).
61. Fisher, N.I. *Statistical Analysis of Circular Data* (Cambridge University Press, 1995).