Quantifying the Energy Efficiency of Object Recognition and Optical Flow

Michael Anderson
Forrest Iandola
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-184
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-184.html

November 24, 2014

Quantifying the Energy Efficiency of Object Recognition and Optical Flow

Michael Anderson, Forrest Iandola, Kurt Keutzer
UC Berkeley ASPIRE Lab

November 23, 2014

Abstract

In this report, we analyze the computational and performance aspects of current state-of-the-art object recognition and optical flow algorithms. First, we identify important algorithms for object recognition and optical flow, then we perform a pattern decomposition to identify key computations. We include profiles of the runtime and energy efficiency (GFLOPS/W) for our implementation of these applications on a commercial architecture. Finally, we include an analysis of memory-bandwidth boundedness for optical flow to identify opportunities for communication-avoiding algorithms.

Our results were measured on an Intel i7-4770K (Haswell) reference platform. A five-layer convolutional neural network used for object classification achieves 0.70 GFLOPS/W, which is 21% of the theoretical compute bound for this Haswell processor. On the Horn-Schunck, Lucas-Kanade, and Brox optical flow methods our implementations achieve 0.0338, 0.0103, and 0.0203 GFLOPS/W respectively. Our implementation achieves 7.9% of the theoretical bandwidth bound, assuming no cross-iteration memory optimization, for Horn-Schunk optical flow using the Jacobi solver, and 9.7% of the bandwidth bound for the conjugate-gradient solver. To improve performance, we will focus first on increasing bandwidth utilization, then on doing cross-iteration memory optimizations such as blocking and tiling the Jacobi solver and employing communication-avoiding linear solvers.

We also compare the runtime-accuracy tradeoffs for each optical flow method. We find that each method has distinct advantages over the other methods in terms of the runtime-accuracy tradeoff, so we will continue to develop and support all three methods in the future.

1 Introduction

In this report, we examine computations required for on-board unmanned aerial vehicle (UAV) vision processing. Specifically, we focus on object recognition, object tracking, and optical flow. Given that on-board processing is constrained by power, we focus on quantifying the energy efficiency and accuracy of current state-of-the-art methods.

We start by decomposing the application capabilities, for example object recognition using convolutional neural networks, into patterns. This provides a high-level structural and computational understanding of the application. Then we profile performance on a commercial processor (Intel i7-4770K). To contextualize performance, we calculate the number of floating point operations (FLOPs) performed for each computation. This allows us to compute a measure of energy efficiency known as giga-flops per second per watt (GFLOPS/W). For optical flow, we also count the total number of bytes transferred in the inner loop of the algorithm. This

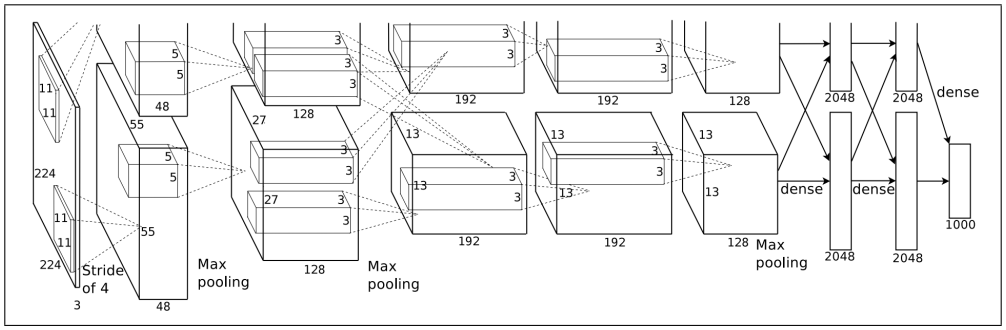


Figure 1: Deep convolutional neural network for object recognition [2]. This network layout achieved the highest object classification accuracy for ImageNet 2012, and a similar network won ImageNet 2013. Big data and efficient implementations have made deep learning accurate and tractable.

allows us to compute the flop-to-byte ratio, which gives an idea of the memory-boundedness of the algorithm and the potential speedup from communication-avoiding algorithms [1].

In Section 2, we analyze object recognition using convolutional neural networks [2]. We find that a five-layer convolutional neural network is able to achieve 0.70 GFLOPS/W. In Section 3, we analyze the Horn-Schunck, Lucas-Kande, and Brox optical flow algorithms [3, 4, 5]. We determine these methods achieve 0.0338, 0.0103, and 0.0203 GFLOPS/W respectively. We also compute the flop-per-byte ratio for Horn-Schunck and Brox methods and report accuracy results for all three methods, and we compute the achieved percentage of peak bandwidth for the Horn-Schunck method. Section 4 concludes our report.

2 Object Recognition

Object recognition is a key enabling technology for a variety of UAV capabilities including navigation, odometry, and reconnaissance.

2.1 State of the art Algorithm

Within the past 18 months, the computer vision community has seen a large improvement in accuracy by designing systems based on deep neural networks instead of hand-engineered descriptors. The key algorithms of the deep learning revolution can be traced back to the late 1980s. However, the rise of big data has led to huge labeled datasets (e.g. ImageNet [6] with >1M labeled images) for training and evaluating object recognition systems. It turns out that large datasets are a lynchpin of high-accuracy neural networks for object recognition. Additionally, extremely efficient deep neural network implementations such as Berkeley's *Caffe* [7] expose enough parallelism to make ImageNet a tractable benchmark for deep neural network object recognition. Today, neural networks such as Alexnet [2] and their ilk (e.g. [8],[9]) provide state-of-the-art object classification accuracy (up to 88% when scored on top-5 categories) on the 1000-category ImageNet dataset. We show an illustration of Alexnet in Figure 1.

In the remainder of this section, we analyze the computational patterns and bottlenecks, GFLOPS/s, and energy for state-of-the-art deep convolutional networks on the Haswell reference architecture.

Application Pattern	Number of Papers
Convolution	30
Histogram Accumulation	29
Vector Distance	22
Quadratic Optimization	15
Graph Traversal	9
Eigen Decomposition	6
K-means Clustering	6
Hough Transform	4
Nonlinear Optimization	4
Meanshift Clustering	2
Fast Fourier Transform	1
Singular Value Decomposition	1
Convex Optimization	1
K-medoids Clustering	1
Agglomerative Clustering	1

Figure 2: **The ASPIRE "Periodic Table" of computer vision computational patterns.** Computer vision algorithms evolve quickly, but these patterns continue to underpin most computer vision mechanisms. "Number of papers" denotes the number of papers in the CVPR 2011 object recognition track that leverage each pattern.

2.2 Computational Patterns

After several years of work on efficient computer vision in the ParLab and ASPIRE Lab at Berkeley, we have codified computer vision computations into a "periodic table" of 15 underlying computational patterns (Figure 2). While computer vision algorithms continue to evolve and advance, these underlying patterns have remained relatively static over many generations of computer vision algorithms. Recently, object recognition algorithms have seen a major shift to deep learning, and it would be easy for computational efficiency researchers to be intimidated by this – how much of what we know about efficient computer vision will transfer to these new deep neural algorithms? Well, as it turns out, it is quite reasonable to map deep neural networks into our periodic table of computer vision patterns (Figure 2). As we will show in the next paragraph, analyzing the performance and energy efficiency of deep neural networks is quite easy, so long as we think in terms of well-understood patterns that are less susceptible to computer vision algorithmic changes.

Broadly, deep neural networks perform feature extraction and recognition by taking an image and feeding it through several layers of filters and dimensionality adjustments. For both training and inference, layers are implemented with primitives such as 3D convolution with multiple kernels, neighborhood max filtering ("max-pooling"), ReLu (removing negative numbers), and dropout (zeroing out a random collection of values to avoid overfitting). In Table 1, we map these deep neural network primitives into our periodic table of computer vision computational patterns.

2.3 Performance and Energy Analysis

We now turn to analyzing the computational complexity, efficiency, and energy of object recognition with convolutional neural networks. As shown in the previous subsection, convolution dominates the overall computation time in this system. Therefore, we use convolution as a lower bound in terms of overall FLOP count, leading to slightly conservative but reasonable

Layer type	Pattern(s)
convolution layer	convolution
dropout layer	–
ReLu layer	convolution (1x1 filter)
max-filtering	convolution-style data access pattern

Table 1: Mapping convolutional neural networks to the ASPIRE periodic table of computer vision patterns.

Layer	Runtime per 50 frames (s)	Input dims	Filter dims	# filters	Complexity (# GFLOP)	GFLOPS/s	Avg Power (W)	Energy per 50 frames (J)	GFLOPS/s/W
conv1	0.0980	224x224x3	11x11x3	96	10.93	111.46			
conv2	0.1806	55x55x48	5x5x48	256	22.39	124.02			
conv3	0.0860	27x27x128	3x3x128	384	14.95	173.81			
conv4	0.0859	13x13x192	3x3x192	384	11.21	130.52			
conv5	0.0756	13x13x192	3x3x192	256	7.48	98.93			
TOTAL conv layers	0.5261				66.97	127.28			
TOTAL all layers	0.71				>=127.28	94.32	135	95.85	**0.70**

Table 2: Performance per convolutional layer and overall for the Berkeley Caffe [7] convolutional neural network, initialized with the Alexnet [2] configuration. Evaluated on the Haswell i7-4770K reference architecture.

efficiency and energy results.

In Table 2, we show the analysis and results in terms of GFLOPS/s for the convolutional layers in the Caffe convolutional neural network, using the same configuration as discussed previously. As you can see in Table 2, the layers funnel down from a 224x224 3-channel input image down to a 13x13 256-channel feature descriptor map. Given our coarse power measurement technology, we are able to obtain the power and energy of the overall system, but not of individual layers. Also, since the computation of neural networks in Caffe is dominated by convolution, we use the number of GFLOPs in convolution as a lower bound for the overall computational complexity of all layers in the neural network. This analysis culminates in finding that object recognition with the Caffe convolutional neural network achieves 0.70 GFLOPS/s/W on the Intel Haswell reference platform (Table 2). This is 21% of the theoretical compute bound for this Haswell processor.

3 Optical Flow

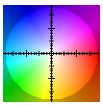
Optical flow is a common computer vision application that computes the apparent motion of each pixel between pairs of images, or between frames in a video. Optical flow information enables point tracking which can be a powerful capability for UAVs.

Optical flow between two images can be visualized (Figure 3b). Colors indicate direction and intensity indicates magnitude of pixel motion. Quality of solution is measured using standard benchmarks such as the Middlebury optical flow benchmark dataset [10] and the KTTI vision benchmark suite [11]. The quality metrics for optical flow are average angular error (AAE) of the flow vectors compared to ground truth provided by the these benchmark datasets, as well as average endpoint error (AEE) of the flow vectors.

There are many different ways to solve optical flow. As of February 2014, the KTTI vision benchmark results webpage for optical flow reports results for 42 different optical flow methods. The Middlebury benchmark results webpage reports results for 95 different optical flow methods.

We choose three methods to focus on: Horn-Schunck [3], Brox [5], and Lucas-Kanade [12]. We focus on Horn-Schunck and Brox due to their popularity, along with the general consensus

(a) Color indicates direction of the flow and intensity indicates the magnitude of the flow.



(b) Colorcode mapping colors to flow directions and magnitudes.

Figure 3: Visualization of optical flow [10].

that the majority of newer methods are simply extensions of these original formulations [13]. We include Lucas-Kanade in our analysis because it considers only local image patches, so it is fundamentally different than Horn-Schunk and Brox. The Lucas-Kanade method is also much cheaper to compute.

We will analyze these three optical flow methods in the following subsections. For each method, as specified in the deliverable text, we will show:

- the high level algorithm description,
- the decomposition into computational and structural patterns,
- profiles of the runtime and energy analysis,
- analysis of computation vs. quality of solution trade-offs, and
- an analysis of memory-boundedness.

We also provide plots comparing all three optical flow methods side-by-side, run with a variety of parameters, in terms of runtime vs. accuracy.

3.1 Horn-Schunck Method

3.1.1 High level algorithm description

The Horn-Schunck method [3] is formulated as a minimization of the following energy functional:

$$E = \iint (I_x u + I_y v + I_t)^2 + \alpha^2 \left(\left(\frac{\partial u}{\partial x} \right)^2 + \left(\frac{\partial u}{\partial y} \right)^2 + \left(\frac{\partial v}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial y} \right)^2 \right) dxdy \qquad (1)$$

I_x is the image gradient in the x dimension, I_y is the image gradient in the y dimension, I_t is the image gradient in the time dimension, u and v are the x and y components of the flow vectors, respectively, and α is a parameter that trades off smoothness of the flow field with the accuracy of the flow nearby a given pixel. The integral is summing this quantity for every pixel in the image. Since each term in this functional represents a cost, we try to minimize the functional over all possible flow functions u and v.

3.1.2 Decomposition into computational and structural patterns

Figure 4 contains code that computes optical flow in Python using the Horn-Schunck method using the variant of the Jacobi solver which was proposed in the Horn-Schunck paper [3]. This solver is also called the iterative 2×2 blockwise linear solver [14]. This code runs in our Hindemith framework. The Hindemith framework analyzes the algorithm description in Python and composes hand-written OpenCL functions to eliminate unnecessary memory traffic that is common array codes such as this. This particular implementation was designed to match the implementation found in the ArrayFire example codes [15]. We've annotated the code with the input types, as well as the computational and structural patterns.

The solver consists mainly of Dense Linear Algebra and Structured Grid operations. This pattern decomposition informs hardware and software implementation choices. For example, these particular patterns are particularly amenable to vectorization and tiling optimizations so we expect this application to compile to efficient code using vectorizing compilers or implicitly parallel languages, and to execute efficiently on vector or SIMD hardware.

We can also solve the Horn-Schunck algorithm using other linear solvers, such as conjugate-gradient (CG), preconditioned conjugate-gradient (PCG), and red-black Gauss Seidel (RB). We have also implemented those linear solvers in our Hindemith framework and will present comparative performance and accuracy results using those linear solvers.

3.1.3 Profiles of runtime and energy analysis

In this section we will analyze the runtime and energy consumption of Horn-Schunck optical flow using a variety of different linear solvers. We run our experiment on Intel Core i7-4770 CPU 3.4 GHz (Haswell) processor. We use the AMD APP OpenCL SDK compiler and runtime. The power was taken using the Watts Up Pro? power logger at one second intervals, then averaged over the duration of multiple executions. The input image pair is RubberWhale from the Middlebury optical flow benchmark set, which is size 588x384. We do not resize the image. We represent the images, the flow, and all intermediate data in grayscale single-precision floating point format. We set the parameter α to 0.1.

Because the Hindemith framework fuses most of these operations together, we do not have line-by-line profiling information. Instead, we record the runtime and power consumption for an execution of 400 linear solver iterations. Then we compute the average runtime and energy *per linear solver iteration*. Some linear solvers converge faster than others. We will consider the trade-off between per-iteration efficiency and convergence rate in the next section.

3.1.4 Analysis of computation vs quality of solution trade-offs

As shown in Table 3, different linear solvers have different runtime and energy costs. However, there is a trade-off between computation and quality of solution that must be explored. More computationally expensive solvers such as conjugate-gradient converge to a good solution faster than cheaper solvers like the Jacobi solver. Figure 5 shows the rate of convergence for each of

```
def hs_oflow    (im1_data, im2_data,  # Input images
                 D,                   # Laplacian stencil
                 Gx, Gy,              # Gradient stencils
                 u, v,                # Flow vectors
                 zero, one, lam2      # Scalars
                 ):
  du = zero * u                       # Dense Linear Algebra
  dv = zero * v                       # Dense Linear Algebra
  Ix = Gx*im1_data                    # Structured Grid
  Iy = Gy*im1_data                    # Structured Grid
  It = im1_data -
       warp_img2d(im2_data, u, v)     # Sparse Linear Algebra
  Ix2 = Ix * Ix                       # Dense Linear Algebra
  IxIy = Ix * Iy                      # Dense Linear Algebra
  Iy2 = Iy * Iy                       # Dense Linear Algebra

  # Application pattern: Linear Solver
  # Structural pattern: Iterator
  for i in range(200):
    ubar = D * du                     # Structured Grid
    vbar = D * dv                     # Structured Grid
    num = Ix * ubar + Iy * vbar + It  # Dense Linear Algebra
    den = Ix2 + Iy2 + lam2            # Dense Linear Algebra
    du = ubar - (Ix * num) / den      # Dense Linear Algebra
    dv = vbar - (Iy * num) / den      # Dense Linear Algebra
  return du, dv
```

Figure 4: Code and pattern decomposition for Horn-Schunck optical flow solved using an Jacobi linear solver

Solver Type	Runtime per Frame (s)	Average Power (W)	Energy per Frame (J)	# Iter.	Runtime per iteration (ms)	Energy per iteration (mJ)	GFLOPS per Watt
Jacobi	0.735	95.16	69.98	400	1.838	174.9	0.0338
CG	1.542	98.39	151.74	400	3.856	379.4	0.0256
PCG	2.193	100.69	220.80	400	5.482	552.0	0.0252
RB	2.539	96.43	244.79	400	6.346	612.0	0.0193

Table 3: Runtime and energy metrics for Horn-Schunck optical flow

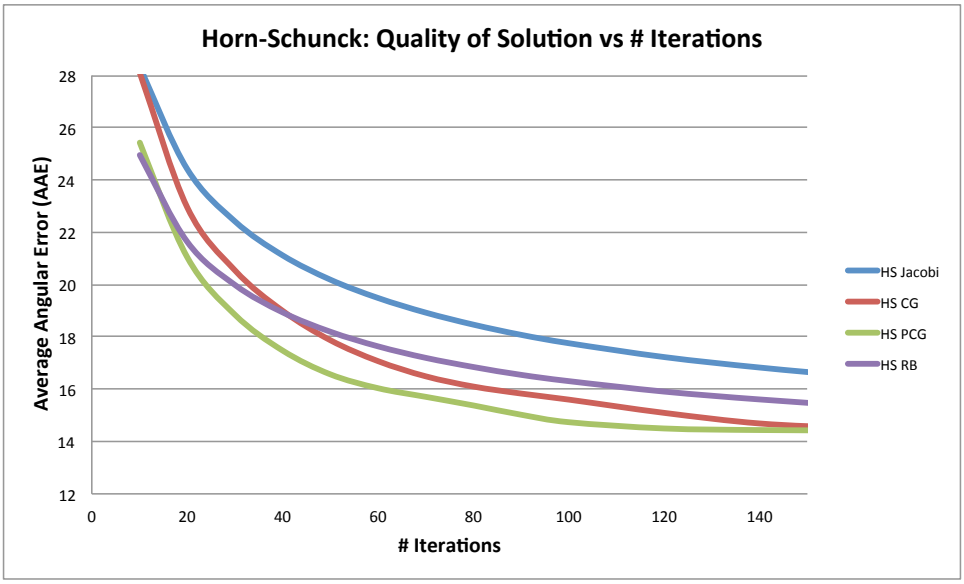


Figure 5: Quality of solution vs number iterations for different Horn-Schunck linear solvers.

the four linear solvers. The best performing solver, in terms of quality per number of iterations, is the preconditioned conjugate gradient solver, followed by the red-black Gauss Seidel, and conjugate gradient solvers.

We are interested in the energy it takes to get a given solution quality. In Figure 6, we plot the quality of solution for each linear solver *per Joule*. This is calculated by multiplying the average number of Joules per iteration with the total number of iterations at each point. As we would expect, Jacobi becomes a more attractive option when we consider Joules instead of iterations. This is because Jacobi iterations are comparatively cheap in terms of Joules. Conversely, our implementation of red-black Gauss Seidel proves to be very inefficient when we consider Joules instead of just iterations.

3.1.5 Analysis of memory-boundedness

For our analysis of memory-boundedness and GFLOPS/Watt, we consider only the inner loop of the linear solver. In Figure 7, we annotate the number of FLOPS performed for every operation in the Jacobi linear solver:

One iteration of the Jacobi linear solver for Horn-Schunck computes 28*h*w FLOPS, where h and w are the height and width of the image respectively. The number of words transferred between memory and the processor is 9*h*w (to read variables Ix, Iy, It, Ix2, Iy2, du, dv, and to write variables du, dv). We can use this information to compute the arithmetic intensity (FLOPS/Byte) of this kernel. Arithmetic intensity is a measure of the memory-boundedness of a particular algorithm or implementation [16]. Depending on the balance of floating-point and bandwidth capabilities of a particular device, we can compute a limit on the achievable performance by multiplying the arithmetic intensity by the peak memory bandwidth, computed by the STREAM benchmark [17].

The arithmetic intensity of the Jacobi optical flow kernel is $\frac{28}{4*9} = 0.778$ FLOPS per byte. We measured a STREAM copy bandwidth of up to 52 GB/Sec on the Haswell reference machine with an array size set to match the working set of the Jacobi solver (compiler flags: gcc -O3 -fopenmp stream.c -o stream_omp -DSTREAM_ARRAY_SIZE=396536, 8 threads). This means

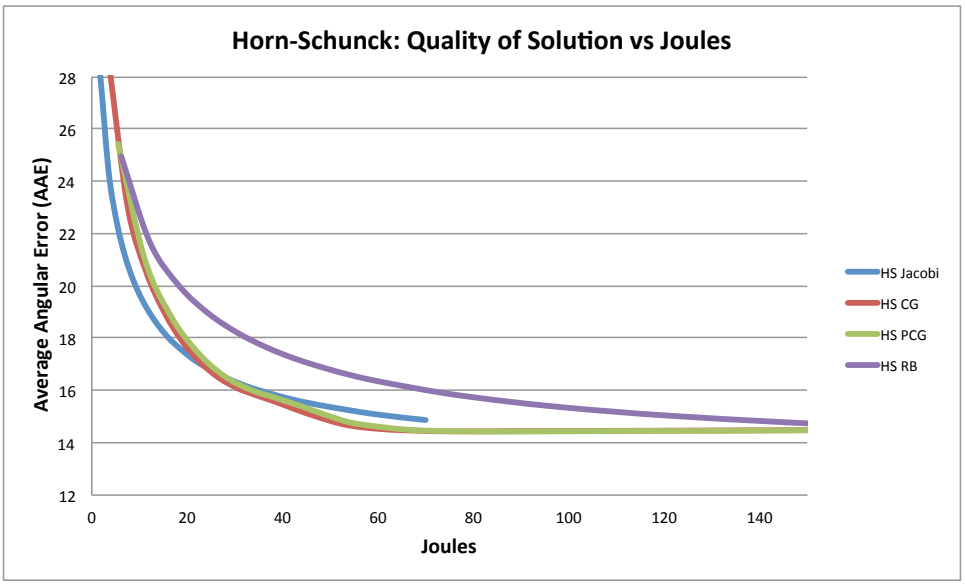


Figure 6: Quality of solution vs number of Joules spent for different Horn-Schunck linear solvers.

```
ubar = D * du                    # 9*h*w FLOPS
vbar = D * dv                    # 9*h*w FLOPS
num = Ix * ubar + Iy * vbar + It # 4*h*w FLOPS
den = Ix2 + Iy2 + lam2           # 2*h*w FLOPS
du = ubar - (Ix * num) / den     # 2*h*w FLOPS
dv = vbar - (Iy * num) / den     # 2*h*w FLOPS
```

Figure 7: Number of FLOPS performed for each operation in the Horn-Schunck Jacobi linear solver

```
Ap0 = (D*p0 + Ix2 * p0 + IxIy * p1)          # 13*h*w FLOPS
Ap1 = (D*p1 + Iy2 * p1 + IxIy * p0)          # 13*h*w FLOPS
alpha = rsold / sum2d(p0 * Ap0 + p1 * Ap1)   # 4*h*w FLOPS
du = du + alpha * p0                         # 2*h*w FLOPS
dv = dv + alpha * p1                         # 2*h*w FLOPS
r0 = r0 - alpha * Ap0                        # 2*h*w FLOPS
r1 = r1 - alpha * Ap1                        # 2*h*w FLOPS
rsnew = sum2d(r0 * r0 + r1 * r1)             # 4*h*w FLOPS
beta = rsnew / rsold
p0 = r0 + beta * p0                          # 2*h*w FLOPS
p1 = r1 + beta * p1                          # 2*h*2 FLOPS
rsold = rsnew
```

Figure 8: Number of FLOPS performed for each operation in the Horn-Schunck conjugate gradient linear solver

the max achievable performance for this algorithm on the Haswell reference platform, without cross-iteration memory optimizations, is 0.778×52, or 40.4 GFLOPS. Our implementation actually performs 13*h*w word transfers instead of the 9*h*w theoretical minimum because the framework breaks the inner loop into two OpenCL kernels. We are currently acheiving 3.21 GFLOPS, which is 8% of the 40.4 GFLOPS limit imposed by the bandwidth-bound, assuming no cross-iteration memory optimizations. In the future, we can work to improve our bandwidth performance. The application can also benefit from the cross-iteration memory optimizations. This includes blocking and tiling the Jacobi solver and employing communication-avoiding linear solvers.

Figure 8 shows the number of FLOPS computed for the Horn-Schunck method using the conjugate-gradient solver. One iteration of the conjugate-gradient linear solver for Horn-Schunck computes 46*h*w FLOPS, where h and w are the height and width of the image respectively. The number of words transferred between memory and the processor is 15*h*w (to read variables du, dv, p0, p1, Ix2, IxIy, Iy2, r0, r1, and to write variables p0, p1, r0, r1, du, dv). This means the arithmetic intensity of this kernel is $\frac{46}{4*15} = 0.766$ FLOPS per byte. We measured a STREAM copy bandwidth of up to 33 GB/Sec on the Haswell reference machine with an array size set to match the working set of the Jacobi solver (compiler flags: gcc -O3 -fopenmp stream.c -o stream_omp -DSTREAM_ARRAY_SIZE=509832, 8 threads). This means the limit imposed by the bandwidth-bound for this algorithm is 0.766×33 or 25.7 GFLOPS. We are currently acheiving 2.51 GFLOPS, or 9.7% of the limit. In the future, we can work to improve our bandwidth performance. After optimizing for bandwidth performance, we can investigate using communication-avoiding linear solvers, which can reduce both the amount of data transferred and the number of synchronizations performed.

We compute the number of GFLOPS for the rest of the Horn-Schunck linear solvers using a similar approach. This allows us to compute the GFLOPS/W values in Table 3.

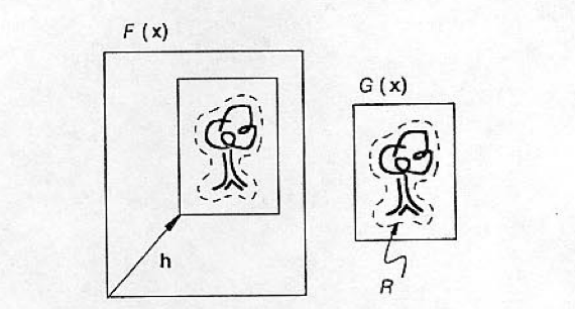


Figure 9: Figure from the original Lucas-Kanade paper (1981) [12]: *"We wish to find the disparity vector h which minimizes some measure of the difference between $F(x+h)$ and $G(x)$ for x in some region of interest R."*[12]

3.2 Lucas-Kanade Method

3.2.1 Problem Formulation

The Lucas-Kanade method is works by examining local regions across a pair of image and computing a displacement vector for each local region [12, 4]. The method can be visualized using a figure from the original 1981 paper [12], shown here as Figure 9. The displacement vector for each local region is computed using a least squares solution to the optical flow equation solved simultaneously for each pixel in the region. The Lucas-Kanade algorithm iterates solving this least squares problem and warping the image until a local minimum is found. This process is summarized and contextualized in a review of Horn-Schunck by Baker and Matthews [4].

3.2.2 Pattern Decomposition

Figure 10 shows our implementation of Horn-Schunck that runs in Python, as well as our Hindemith framework. We've annotated the code with the input types, as well as the computational and structural patterns within the application.

Like Horn-Schunck, this method consists of mainly the Dense Linear Algebra and Structured Grid computational patterns. This tells us that the application will be amenable to optimizations such as tiling and vectorization. However, there is a data-dependent array indexing operation, image warping, that falls under the Sparse Linear Algebra pattern. This means that we won't be able to statically partition the computation given that dependences between operations are not known until runtime.

3.2.3 Profiles of runtime and energy analysis

We evaluate the runtime and energy performance of Lucas-Kanade optical flow using the same machine, methodology, and input data as was used for the Horn-Schunck analysis in Section 3.1.3. In this case, we only run one iteration of Lucas-Kanade and our window size is 4x4. The energy per iteration is much higher than Horn-Schunck, but the energy per problem is much lower.

```
def lk_oflow     (I1, I2,              # Input images
                  u, v,                # Flow vectors
                  Gx, Gy):             # Gradient stencils
  Ix = Gx * I2                        # Structured Grid
  Iy = Gy * I2                        # Structured Grid

  # Structural pattern: Iterator
  for i in range(2):
    WarpedI2 = warp_img2d(I2, u, v)   # Sparse Linear Algebra
    WarpedIx = warp_img2d(Ix, u, v)   # Sparse Linear Algebra
    WarpedIy = warp_img2d(Iy, u, v)   # Sparse Linear Algebra
    ErrorImg =  I1 - WarpedI2         # Dense Linear Algebra

                                      # Structured Grid
    du, dv = lk_least_squares(ErrorImg, WarpedIx, WarpedIy)

    u = u + du                        # Dense Linear Algebra
    v = v + dv                        # Dense Linear Algebra
  return u, v
```

Figure 10: Code and pattern decomposition for Lucas-Kanade optical flow

Runtime per Frame (s)	Average Power (W)	Energy per Frame (J)	# Iter.	Runtime per iteration (ms)	Energy per iteration (mJ)	GFLOPS per Watt
0.02286	59.84	1.368	1	22.86	1368	0.0103

Table 4: Runtime and energy metrics for Lucas-Kanade optical flow using a 4x4 window size

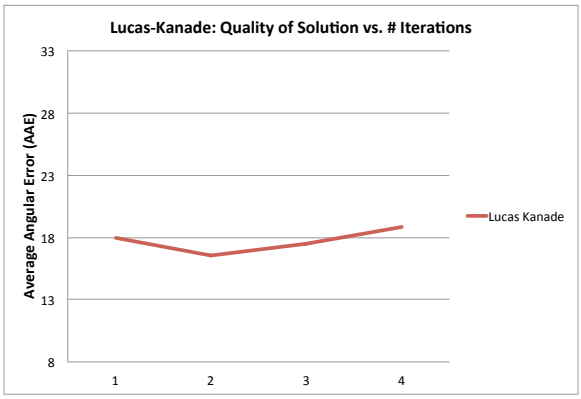


Figure 11: Quality of solution for multiple Lucas Kanade iterations using a 4x4 window size

3.2.4 Analysis of computation vs quality of solution trade-offs

Figure 11 shows the quality of solution (average angular error) of the Lucas-Kanade method for our benchmark image. The average angular error for the Lucas-Kanade method is lower than both the Horn-Schunck and the Brox methods for the same image. However, the Lucas-Kanade solution is much cheaper to compute.

3.2.5 Analysis of memory-boundedness

The Lucas-Kanade algorithm is dominated by the number of FLOPs required for the solution to the local least squares problems. This depends on the size of the window under consideration. For larger windows, the computation will likely be compute-bound, for very small windows the computation will most likely be memory-bound.

We run our tests with a 4x4 window. In this case, the number of FLOPS computed per pixel is 2*2*winsize*winsize, plus 3 FLOPS for computing the error image and adding the displacements to the flow vectors. So the total number of FLOPS for our example is 67*h*w, where h and w are the height and width of the image. This FLOP count is used to compute GFLOPS per Watt in Table 4.

3.3 Brox Method

3.3.1 Problem Formulation

The Brox method is a recent algorithm for optical flow that attains high-quality results but also comparitvely computationally intensive [5]. It is based on an energy-minimization approach similar to the Horn-Schunck method. However, the functional to be minimized has additional terms that add complexity to the formulation but produce a higher-quality result.

The main optimization problem is to minimize an energy functional with two terms. The first term enforces adherence to the gray-value constancy and gradient constancy assumptions, and the second term enforces smoothness in the flow field. In these equations we use the notation from the original paper, where the image coordinates (both x and y) are represented by a vector x, and the flow field (both u and v) are represented by a vector u:

$$E(u, v) = E_{Data} + \alpha E_{Smooth} \tag{2}$$

The E_{Data} term penalizes deviations from the gray-value constancy assumption and the gradient constancy assumption:

$$E_{Data}(u,v) = \int_{\Omega} (|I(x+w) - I(x)|^2 + \gamma|\nabla I(x+w) - \nabla I(x)|^2)dx \tag{3}$$

The E_{Smooth} tern penalizes flow functions that are not smooth. That is, typically pixels that are nearby one another move in similar directions and at similar speeds. So we assume the change in flow should not vary much in either space or time:

$$E_{Smooth}(u,v) = \int \Psi(\|\nabla_3 u\|^2 + \|\nabla_3 v\|^2)dx \tag{4}$$

The Ψ function is added to this penalty function in order to reduce the influence of outliers. ϵ is 1E-3:

$$\Psi(s^2) = \sqrt{s^2 + \epsilon^2} \tag{5}$$

3.3.2 Pattern Decomposition

The following code computes optical flow in Python using the Brox method and the conjugate-gradient linear solver. This particular implementation was designed to match the implementation found in the OpenCV library [18]. However, this code does run in our Hindemith framework. We've annotated the code with the input types, as well as the computational and structural patterns.

```
def brox_oflow  (I1, I2,                        # Input images
                 Gx, Gy,                        # Gradient stencils
                 u, v,                          # Flow vectors
                 pointfive, zero, eps,          # Scalars
                 brox_alpha, brox_beta, gamma, # Parameters
                 A                              # Structured sparse matrix
                 ):

  du = zero * u                          # Dense Linear Algebra
  dv = zero * v                          # Dense Linear Algebra
  tex_Ix0 = Gx * I1                      # Structured Grid
  tex_Iy0 = Gy * I1                      # Structured Grid
  tex_Ix = Gx * I2                       # Structured Grid
  tex_Iy = Gy * I2                       # Structured Grid
  tex_Ixx = Gx * tex_Ix                  # Structured Grid
  tex_Iyy = Gy * tex_Iy                  # Structured Grid
  tex_Ixy = Gx * tex_Iy                  # Structured Grid

  # Application pattern: Non-convex non-linear solver
  # Outer fixed-point iterations
  for outer in range(5):

    # Warp images
    Iz = warp_img2d(I2, u, v) - I1      # Sparse Linear Algebra
    Ix = warp_img2d(tex_Ix, u, v)       # Sparse Linear Algebra
    Ixz = Ix - tex_Ix0                  # Dense Linear Algebra
    Ixy = warp_img2d(tex_Ixy, u, v)     # Sparse Linear Algebra
    Ixx = warp_img2d(tex_Ixx, u, v)     # Sparse Linear Algebra
    Iy = warp_img2d(tex_Iy, u, v)       # Sparse Linear Algebra
```

```
  Iyz = Iy - tex_Iy0                  # Dense Linear Algebra
  Iyy = warp_img2d(tex_Iyy, u, v)     # Sparse Linear Algebra

  pd1 = (Iz + Ix * du + Iy * dv)      # Dense Linear Algebra
  pd2 = (Ixz + Ixx * du + Ixy * dv)   # Dense Linear Algebra
  pd3 = (Iyz + Ixy * du + Iyy * dv)   # Dense Linear Algebra

  PsiData = pointfive /               # Dense Linear Algebra
            sqrt(pd1*pd1 + gamma*(pd2*pd2 + pd3*pd3) + eps)

  # Set up linear system
  gx0 = Gx * (u + du)                 # Structured Grid
  gy0 = Gy * (u + du)                 # Structured Grid
  gx1 = Gx * (v + dv)                 # Structured Grid
  gy1 = Gy * (v + dv)                 # Structured Grid
  PsiSmooth = pointfive /             # Dense linear Algebra
              sqrt(gx0*gx0 + gy0*gy0 + gx1*gx1 + gy1*gy1 + eps)
  set_brox_matrix(A, PsiSmooth, brox_alpha)       # Structured Grid
  du_coef0 = PsiData * (Ix * Ix + gamma * (Ixx * Ixx + Ixy * Ixy))   # DLA
  dv_coef0 = PsiData * (Ix * Iy + gamma * (Ixx * Ixy + Ixy * Iyy))   # DLA
  du_coef1 = PsiData * (Iy * Ix + gamma * (Iyy * Ixy + Ixy * Ixx))   # DLA
  dv_coef1 = PsiData * (Iy * Iy + gamma * (Iyy * Iyy + Ixy * Ixy))   # DLA
  b0 = zero - (PsiData * Ix * Iz + gamma * PsiData *
              (Ixx * Ixz + Ixy * Iyz) + A * (u + du))                # DLA
  b1 = zero - (PsiData * Iy * Iz + gamma * PsiData *
              (Iyy * Iyz + Ixy * Ixz) + A * (v + dv))                # DLA

  # Application pattern: Linear system of equations
  # Solve CG for du, dv
  r0 = b0 - (A*du + du_coef0 * du + dv_coef0 * dv)  # Structured Grid
  r1 = b1 - (A*dv + du_coef1 * du + dv_coef1 * dv)  # Structured Grid
  p0 = b0 - (A*du + du_coef0 * du + dv_coef0 * dv)  # Structured Grid
  p1 = b1 - (A*dv + du_coef1 * du + dv_coef1 * dv)  # Structured Grid
  rsold = sum2d(r0 * r0 + r1 * r1)                  # Dense Linear Algebra
  for inner in range(40):
    Ap0 = (A * p0 + du_coef0 * p0 + dv_coef0 * p1)  # Structured Grid
    Ap1 = (A * p1 + du_coef1 * p0 + dv_coef1 * p1)  # Structured Grid
    alpha = rsold / sum2d(p0 * Ap0 + p1 * Ap1)      # Dense Linear Algebra
    du = du + alpha * p0                            # Dense Linear Algebra
    dv = dv + alpha * p1                            # Dense Linear Algebra
    r0 = r0 - alpha * Ap0                           # Dense Linear Algebra
    r1 = r1 - alpha * Ap1                           # Dense Linear Algebra
    rsnew = sum2d(r0 * r0 + r1 * r1)                # Dense Linear Algebra
    beta = rsnew / rsold                            # Dense Linear Algebra
    p0 = r0 + beta * p0                             # Dense Linear Algebra
    p1 = r1 + beta * p1                             # Dense Linear Algebra
    rsold = rsnew

return du, dv
```

Like Horn-Schunck, Brox is mainly Dense Linear Algebra and Structured Grid operations which means it will run well on architectures that support vector or SIMD execution.

Solver Type	Runtime per Frame (s)	Average Power (W)	Energy per Frame (J)	# Iter.	Runtime per iteration (ms)	Energy per iteration (mJ)	GFLOPS per Watt
CG	6.527	101.4	662.1	1400	4.662	472.9	0.0202
PCG	8.656	104.2	901.8	1400	6.183	644.1	0.0203

Table 5: Runtime and energy metrics for Brox optical flow

We can also solve the Brox algorithm using other linear solvers, such as preconditioned conjugate-gradient (PCG) [19] and red-black Gauss Seidel. We have also implemented preconditioned conjugate-gradient in our Hindemith framework and will present comparative performance and accuracy results using both linear solvers.

3.3.3 Profiles of runtime and energy analysis

We evaluate the runtime and energy performance of Brox optical flow using the same machine, methodology, and input data as was used for the Horn-Schunck analysis in Section 3.1.3. In this case, we do seven warping iterations on the image, five outer iterations and 40 linear solver iterations, for a total of 1400 linear solver iterations. We set the parameter α to 0.197 and the parameter γ to 50.0. The energy per iteration is for Brox is slightly more than the energy per iteration for Horn-Schunk even though it is the same linear solver. This is because Brox uses an explicit sparse matrix, requiring more memory traffic.

3.3.4 Analysis of computation vs quality of solution trade-offs

Currently our Brox implementation achieves an average angular error of 7.33 when solved with the preconditioned conjugate-gradient algorithm using seven warping iterations, five fixed-point iterations, and 40 PCG iterations (1400 total iterations). For traditional CG with the same setup, we are currently achieving an average angular error of 8.89. As shown in Figures 13 and 14, these are among the highest accuracy results we were able to produce on our two benchmarks. It is difficult to contextualize the quality of these solutions compared to published figures because our two images come from the Middlebury training set and most published results report accuracy on the test set. The Middlebury optical flow benchmark webpage lists accuracy results for many published methods on the test set.

3.3.5 Analysis of memory-boundedness

The following analysis is for the Brox method using the conjugate-gradient solver. We consider the inner loop of the conjugate-gradient solver, shown in Figure 12. The only difference between this and the Horn-Schunck conjugate-gradient analysis is that the matrix must be represented explicitly, meaning it requires 5*h*w words to be transferred from memory to the processor. The Horn-Schunck sparse matrix, in contrast, could be represented implicitly.

One iteration of the conjugate-gradient linear solver for Brox computes 46*h*w FLOPS, where h and w are the height and width of the image respectively. The number of words transferred between memory and the processor is 20*h*w (to read variables A, du, dv, p0, p1, Ix2, IxIy, Iy2, r0, r1, and to write variables p0, p1, r0, r1, du, dv). This means the arithmetic

```
Ap0 = (A * p0 + du_coef0 * p0 + dv_coef0 * p1)   # 13*h*w FLOPS
Ap1 = (A * p1 + du_coef1 * p0 + dv_coef1 * p1)   # 13*h*w FLOPS
alpha = rsold / sum2d(p0 * Ap0 + p1 * Ap1)       # 4*h*w FLOPS
du = du + alpha * p0                             # 2*h*w FLOPS
dv = dv + alpha * p1                             # 2*h*w FLOPS
r0 = r0 - alpha * Ap0                            # 2*h*w FLOPS
r1 = r1 - alpha * Ap1                            # 2*h*w FLOPS
rsnew = sum2d(r0 * r0 + r1 * r1)                 # 4*h*w FLOPS
beta = rsnew / rsold
p0 = r0 + beta * p0                              # 2*h*w FLOPS
p1 = r1 + beta * p1                              # 2*h*w FLOPS
rsold = rsnew
```

Figure 12: Number of FLOPS computed for each operation in the Brox conjugate-gradient linear solver

intensity of this kernel is $\frac{46}{4*20} = 0.575$ FLOPS per byte. Since this solver is similar to Horn-Schunck we will take a similar approach to optimizing it: First maximize bandwidth utilization, then apply cross-iteration optimizations and communication-avoiding algorithms [1].

3.4 Comparison of Optical Flow Methods

In Figures 13 and 14, we compare the performance of all three optical flow methods and the linear solvers in terms of both accuracy and runtime on two Middlebury benchmark images [10]. Figure 13 shows the accuracy/speed tradeoff for the Dimetrodon benchmark image, and Figure 14 shows the accuracy/speed tradeoff for the RubberWhale image. These results are on an AMD Radeon 7990 GPU, our fastest platform. We run a number of different parameterizations of each optical flow method. We vary the number of solver iterations, number of warping iterations, number of pyramid levels, and linear solver type. Each point in the plot represents a single run. For Lucas-Kanade, we vary the radius between 3, 5, and 7 pixels wide. For Horn-Schunck we set α to 0.1. For the Brox method, we set α to 0.197 and γ to 50.0. Finally, we apply a 5×5 median filter between warping iterations and pyramid levels to improve accuracy.

The fastest, least accurate points belong to the Lucas Kanade method. As we would expect, the Brox method is generally the most accurate and the slowest. The Horn-Schunck method lies somewhere in between. The best performing linear solver varies depending on both the benchmark image and the error metric.

4 Conclusion

We've explored state-of-the-art algorithms and approaches for object recognition and optical flow. These are two important application capabilities for future embedded vision environments such as on-board unmanned aereal vehicle (UAV) video processing.

For object recognition, we identified convolutional neural networks as a target for analysis, implementation and optimization. We measured an energy efficiency (GFLOPS/W) of 0.70 for a convolutional neural network with five convolutional layers performing an object recognition task. This is 21% of the theoretical compute bound for this Haswell processor.

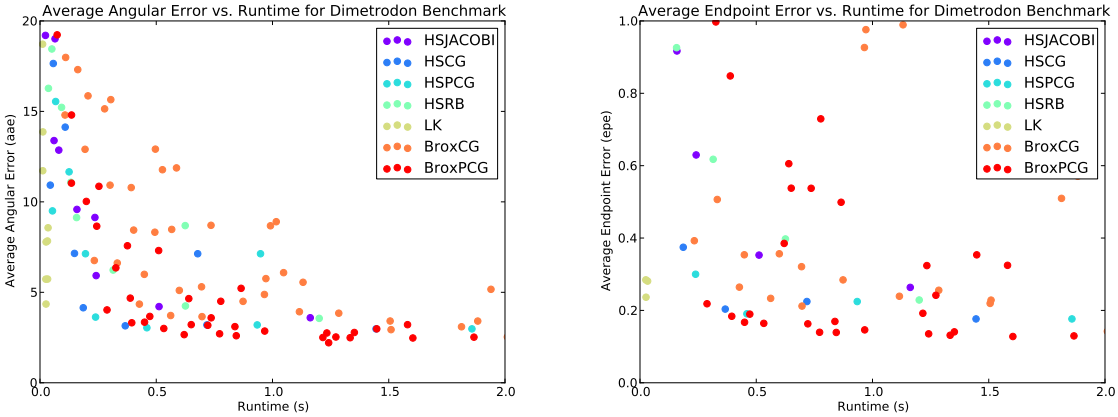


Figure 13: Comparison of optical flow methods. Accuracy vs. runtime on the Dimetrodon benchmark image using many different configurations. Run on AMD Radeon 7990.

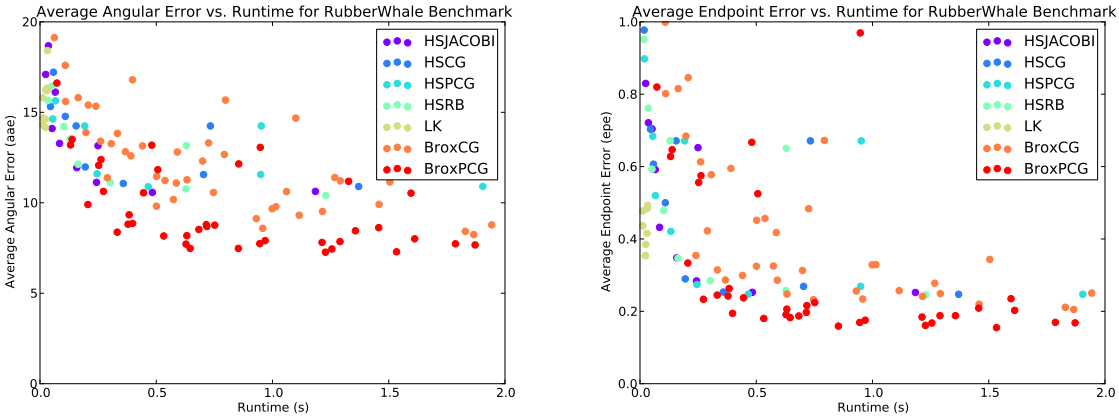


Figure 14: Comparison of optical flow methods. Accuracy vs. runtime on the RubberWhale benchmark image using many different configurations. Run on AMD Radeon 7990

We also identified three important optical flow approaches: Horn-Schunck, Lucas-Kanade, and Brox. We analyzed and implemented optimized versions of these approaches. They achieve 0.0338, 0.0103 and 0.0203 GFLOPS/W respectively. We achieve 7.9% of the theoretical bandwidth bound, assuming no cross-iteration memory optimization, for Horn-Schunk optical flow using an Jacobi solver, and 9.8% of the bandwidth bound for the conjugate-gradient solver. To improve performance on optical flow, we will focus on increasing bandwidth utilization. We were not surprised by the low performance because that the code was previously tuned for GPUs. Also, we expect the Intel OpenCL compiler will generate better results than the AMD compiler, which is our current setup. After improving memory bandwidth utilization, we plan to add cross-iteration memory optimizations such as blocking and tiling the Jacobi solver and communication-avoiding linear solvers.

Finally, we identified a meaningful accuracy vs. runtime tradeoff between the three optical flow approaches. Each method shows distinct advantages in terms of the accuracy-runtime tradeoff compared to the other two. We conclude from this that we should continue to develop all three optical flow methods in the future.

References

[1] Mark Hoemmen. *Communication-Avoiding Krylov Subspace Methods*. PhD thesis, University of California, Berkeley, 2010.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[3] Berthold KP Horn and Brian G Schunck. Determining Optical Flow. *Artificial intelligence*, 17(1):185–203, 1981.

[4] Simon Baker and Iain Matthews. Lucas-Kanade 20 Years On: A Unifying Framework. *International Journal of Computer Vision*, 56(3):221–255, 2004.

[5] Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High accuracy optical flow estimation based on a theory for warping. In *Computer Vision-ECCV 2004*, pages 25–36. Springer, 2004.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition (CVPR)*, 2009.

[7] Yangqing Jia and et al. Caffe. `caffe.berkeleyvision.org`.

[8] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *ArXiV technical report*, 2013.

[9] Matthew Zeiler. Clarifai. `clarifai.com`.

[10] Simon Baker, Daniel Scharstein, JP Lewis, Stefan Roth, Michael J Black, and Richard Szeliski. A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, 92(1):1–31, 2011.

[11] Jannik Fritsch, Tobias Kuehnl, and Andreas Geiger. A New Performance Measure and Evaluation Benchmark for Road Detection Algorithms. In *International Conference on Intelligent Transportation Systems (ITSC)*, 2013.

[12] Bruce D Lucas, Takeo Kanade, et al. An Iterative Image Registration Technique with an Application to Stereo Vision. In *IJCAI*, volume 81, pages 674–679, 1981.

[13] Deqing Sun, Stefan Roth, and Michael J Black. Secrets of optical flow estimation and their principles. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2432–2439. IEEE, 2010.

[14] Louis Le Tarnec, François Destrempes, Guy Cloutier, and Damien Garcia. A proof of convergence of the Horn-Schunck optical flow algorithm in arbitrary dimension. *SIAM Journal on Imaging Sciences*, 7(1):277–293, 2014.

[15] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU acceleration platform. *SPIE Defense, Security, and Sensing*, 2012.

[16] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[17] John D McCalpin. A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE TCCA Newsletter*, pages 19–25, 1995.

[18] Gary Bradski. The OpenCV library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.

[19] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense point trajectories by gpu-accelerated large displacement optical flow. In *11th European Conference on Computer Vision (ECCV 2010)*, pages 438–451. Springer, 2010.