HOG and Spatial Convolution on SIMD Architecture

Ishan Misra Abhinav Shrivastava Martial Hebert

Robotics Institute, Carnegie Mellon University

{imisra,ashrivas,hebert}@cs.cmu.edu

Abstract

In recent years, Histogram of Oriented Gradients (HOG) has gained tremendous popularity in the Computer Vision community. It is a widely used feature in Object Recognition tasks, and is often followed by a classification step. Popular classifiers like SVMs perform a convolution operation in the classification step. Spatial convolution is a building-block in Computer Vision, being used from filters to classifiers. In this report, we present a SIMD implementation on the most generic processors (CPUs), using the most generic instruction set (`x86` or `x86-64`), in a generic language (C++) and target these two very widely used algorithms - HOG and convolution. These steps form a performance bottleneck for real-time classification pipelines. As a way to test our system, we implement the entire testing pipeline of the Exemplar-SVM [17]. We hope that our implementations serve a large audience in speedy research.

1. Introduction and Motivation

Exemplar-SVMs [17, 20] have introduced a novel way of utilizing large amounts of data for image retrieval and object recognition. Using state-of-the-art techniques from discriminative object detection [8], and a linear SVM for each exemplar image, this technique provides good generalization. The training as well as testing part of the Exemplar-SVM pipeline are slow and not suitable for real-time application [20]. While the training might be done offline (and on more powerful computers or clusters), the actual use of the system i.e. the testing, maybe constrained to use computers with limited processing power (mostly, just a CPU). A great way to speed up this pipeline is to use the Graphics Processing Units (GPUs) which have demonstrated great speed-up in traditional computer vision algorithms [9, 18]. However, the GPU is not as widespread as the `x86` or the `x86-64` ISA based CPUs. A lot of software is run on cloud architecture, like Amazon® Elastic Compute Cloud (EC2), which utilizes virtualized hardware. Using a GPU through such virtualized hardware is not an easy task [6]. GPUs are more energy efficient if one considers the number of flops/watt [11]. However, a GPU also requires significantly more power than a CPU. As an example, the power rating of NVIDIA® GTX 280 GPU is about 236 Watts which is almost equal to the power rating of a compute node [11]. Hence, in spite of being energy efficient, on many platforms like on-board processors on robots, a GPU is not a feasible choice. Thus, to target a wider audience, we decided to use the traditional CPUs as our computational platform.

The two most time consuming parts of the testing phase of the Exemplar-SVM are the Histogram of Oriented Gradient (HOG) [5] pyramid computation, and convolution of the exemplars (weight vectors) at each level of the HOG pyramid [20] (Also see Figure 2). In this report, we present multi-core CPU implementations targeting these two bottlenecks. [7] adopted a frequency domain approach to optimizing the Deformable parts-based model [8] pipeline. They make use of fine-tuned CPU architecture based optimizations like L1 cache size blocks etc. In our approach, we aim to be more portable and leverage the power of language constructs, offloading the optimizations to the compiler.

2. Background

2.1. Architecture and SIMD Instructions

The `x86` and `x86-64` instruction sets have had support for (floating point) vector instructions since Pentium 3 [21]. These instructions allow the processor to handle data in the form of vectors. So, for example, to add 8 pairs of numbers, one can simply define two vectors of 8 numbers and add them in a single CPU clock cycle. These instructions belong to the Single Instruction Multiple Data (SIMD) paradigm. Although these instructions have been around for a while, most modern compilers do not generate code that fully utilizes them [14]. For inherently data parallel tasks, it is important that one uses these instructions for maximum performance. The Intel® Integrated Performance Primitives (IPP) [12] is a highly optimized library which has implementations of many Computer Vision and Image Processing algorithms (but this library is not free) . It is a good example of leveraging SIMD features for improving performance.

The most obvious (but rather painstaking) way for programmers to use these instructions is to use SIMD intrinsics or compiler support (like inline assembly supported by the GNU C Compiler) and write SIMD variables or instructions. We observed that there are two easy ways for modern programs to exploit these instructions - rely on auto-vectorization by the compiler, and/or use high level programming language constructs which help the compiler use vector instructions [15, 19].

2.2. Intel ISPC SPMD Compiler

The Intel ISPC Compiler [19] is an open source project that aims to make SIMD programming more accessible by abstraction of SIMD constructs. It defines a set of extensions to the C programming language which the programmer can use for a declarative style of programming (by specifying which computations are independent) like NVIDIA CUDA. It then generates SIMD optimized code according to the architecture of the CPU. Our experience was that the learning curve of ISPC is easy and it is programmer friendly. In our pipeline we try to migrate all of our computationally intensive tasks onto ISPC. In our code base all of the `.ispc` files contain code for the ISPC compiler. ISPC also allows the programmer to spawn threads using underlying system implementation (like `pthreads`, Intel TBB, etc.). This allows the program to utilize both the multi-threaded and SIMD parallelism.

3. Pipeline Overview

We have implemented the testing part of the Exemplar-SVM pipeline as mentioned in [17]. Figure 1 describes the pipeline. Figure 2 shows a representative split amongst the various sections of this pipeline[1]. It clearly shows that convolution and HOG feature pyramid computation (in that order) are (as expected) the most time consuming parts of the pipeline. Our focus was to implement this pipeline in C++, and use the SIMD architecture to make the convolution and HOG faster. In this section we will discuss the various design decisions we made, and other implementation details of this pipeline. Our code is available at https://github.com/imisra/esvmTestCPP.

3.1. HOG implementation details

We have implemented the Histogram of Oriented Gradients (HOG) as formulated in [8]. Given an input image (of size $M \times N$), the main parts of the HOG computation are as follows

1. Compute image gradient in X and Y directions.
2. Determine the best orientation (among 18 fixed orientations) of the gradient at each pixel.

[1]All subsequent mentions of MATLAB implementation refer to [16], unless stated otherwise

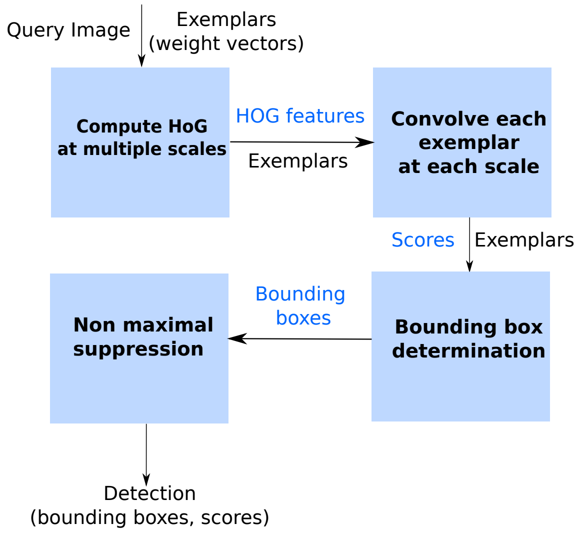

Query Image
Exemplars (weight vectors)
Compute HoG at multiple scales
HOG features
Exemplars
Convolve each exemplar at each scale
Scores
Exemplars
Bounding boxes
Non maximal suppression
Bounding box determination
Detection (bounding boxes, scores)

Figure 1: A flowchart for the Exemplar-SVM detection pipeline.

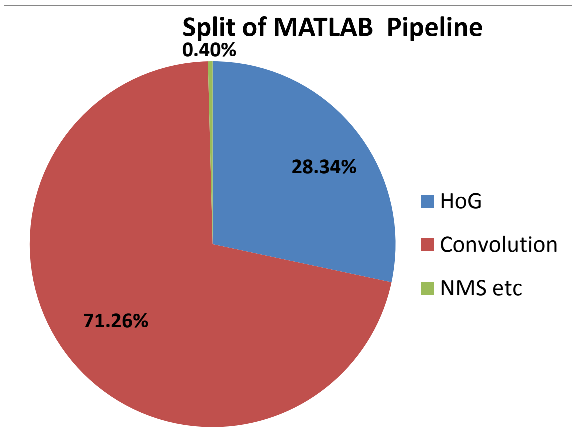


Figure 2: A representative split amongst various functions in the baseline MATLAB® implementation of the testing pipeline [16]. This data was collected for a single image and a single exemplar. Even with one exemplar, convolution takes the maximum percentage of time. As the number of exemplars increase, the number of convolution operations increase linearly, resulting in convolution taking up an even higher share of the total time.

3. Group the pixels of the input image into "cells" based on a cell width (commonly referred to as "sbin" in HOG literature) c. To compute the histogram at each cell we consider contributions from all pixels from the cell. The number of bins in a histogram for one cell is equal to the number of orientations of the gradient (18).

4. The per-cell histogram is normalized by taking into

consideration histograms for neighboring cells [5].

5. The final HOG feature at each cell is now computed as a linear combination the bins of the histogram from Step 4. It is a 31 dimensional feature per cell of the image.

An analysis of the above steps immediately gives us that the steps 1, 2, 4 and 5 are embarrassingly parallel. The main bottleneck in terms of performance is the histogram binning part. This arises due to memory collisions in data parallel architectures [2]. In case of a multi-threaded program, threads should write to the histogram data structure using atomic operations (depicted in Figure 3) Using atomic operations on current hardware architectures uses a significant amount of CPU cycles [13]. Hence we decided to split the "gradient image" among multiple threads and let each thread compute its own histogram. We later merged these histograms from individual threads to form the histogram for the entire image.

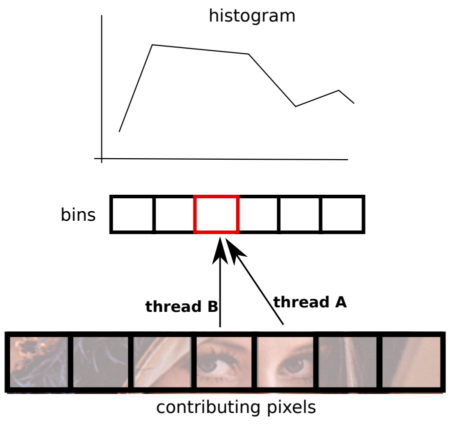


Figure 3: Consider computing a histogram for input data (say pixels of an image). In the histogram binning stage, we can have multiple threads accessing the same bin. Unless precautions are taken, the simultaneous updates will result in undefined results.

The functions

```
file esvm.cpp: binHist(...)
file hog.ispc: combineHist(...)
```

implement the said functionality. `binHist` uses OpenMP threads for computing the histogram. It splits rows of the input amongst OpenMP threads, and these threads compute their local histograms (see Figure 4 for an illustration). We found that implementing `binHist` in ISPC did not give any performance benefit. The SIMD program generated by ISPC for `binHist` actually performed worse than the OpenMP version. We noticed that the data access pattern for the underlying computation was not contiguous. For computing the histogram of one block, we access different orientation bins depending on each contributing pixel's strongest gradient orientation. Current hardware serializes such data loading and hence using SIMD instructions in this case doesn't give higher performance [13].

Once local thread histograms are computed, `combineHist` merges them. It invokes threads which call `mergeHist` to merge chunks of rows. The other steps (1, 2, 4 and 5) are implemented in

```
file esvm.cpp: computeNorm(...)
file hog.ispc: getGradient(...)
               getFeatures(...)
```

The wrapper function

```
file esvm.cpp: computeHogWrapper(...)
```

is intended to abstract the underlying details, and provide an easy way to get the HOG features. The standard pipeline computes HOG features at different scales of the query image (resulting in a "HOG feature pyramid"). The wrapper function

```
file esvm.cpp: computeHogScaleWrapper(...)
```

computes the HOG feature pyramid.

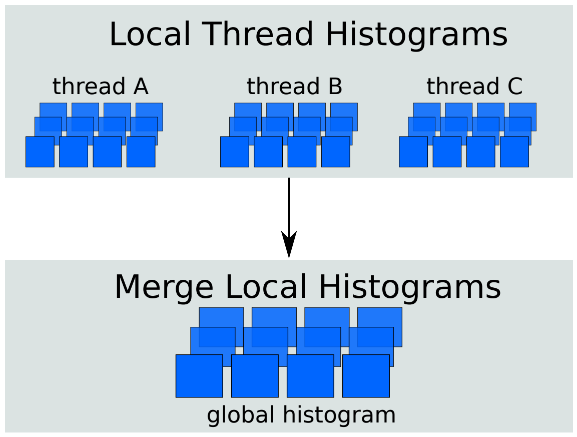


Figure 4: Histogram computation by multiple threads. We divide the gradient of the image into chunks by splitting it on rows. Each thread computes a local histogram for one or more of these chunks. The final global histogram is formed by merging these local histograms together.

3.2. Convolution implementation details

Convolution is a basic operation in image processing. In case of the Exemplar-SVMs, it is used in the detection stage where each exemplar is convolved over the HOG features of the query image. In the standard pipeline for multi-resolution detection, we compute the HOG features at different scales of the image (called the HOG feature pyramid), and convolve the exemplars at each scale. The convolution operation is the major performance bottleneck ([20], Figure 2). We decided to implement spatial convolution using ISPC to leverage SIMD performance boost.

While programming in the SIMD domain, it is crucial to maintain *control flow coherence*. In case of the `x86/x86-64` architecture this implies that within a single thread, one must try to ensure that the control flow is the same for each element in the data being processed. Convolution if implemented naively has edge cases, where one might need to insert `if, else` statements. These have a serious impact on the generated code, since the compiler is not sure of when it may not be able to use vector instructions and cannot easily perform branch prediction optimizations [3]. We implement our basic primitive as a 2D convolution of a matrix with a kernel. In our implementation, we perform only "valid convolution". Hence we do not process elements which lie on the border of $\frac{kernel\_width}{2}$ on all sides of the input data matrix. This removes the need of using any `if` statement. We also use the `reduce_add` function of the ISPC library which improves performance. If desired, the user can get a "full" convolution by padding the input by conventional methods (zero-padding, cyclic-padding). [20] uses zero-padding as a way to handle occluded objects.

In the most general case of the Exemplar-SVM testing phase, one uses multiple exemplars (called *ensemble of Exemplar-SVMs* [17]) on an image. This corresponds to convolution of all the weight vectors associated with the exemplars, at all levels of the HOG feature pyramid. The HOG feature pyramid is computed only once, and hence convolution of multiple exemplars is efficient. In our implementation, we spawn threads, one for each exemplar to be convolved. The core ISPC convolution code performs a series of 2-D convolutions, and merges the result into a 3-D convolution. The corresponding functions are implemented in

```
file conv3D.ispc: convolve2D_single (...)
                  convolve3D (...)
                  mergeConv_single (...)
```

We maintain an array of offsets, which at a point (x, y), determine all the points $(x \pm i, y \pm i)$ which contribute to the convolution sum at this point. This saves writing an inner loop (and can be viewed as a loop unroll for those familiar with the concept).

3.3. Miscellaneous things

The Convolution process gives us the score of an exemplar at each block of the HOG. We use simple thresholding to determine which scores to keep, and sort the blocks according to their scores [20]. Both these operations (thresholding and sorting) can be combined easily. We then determine the bounding boxes (in the scale of the original image) corresponding to the top-ranked blocks. This is followed by Non Maximal Suppression which basically involves discarding boxes (with a lower score), which have considerable overlap with a box of a higher score. The primitive operations (like sorting) needed for handling these bounding boxes (whose number is typically small) have fast implementations in standard C++. We implemented this part of the pipeline in C++ as we did not see any significant advantage of doing this in ISPC.

We have created our own wrappers for memory allocation. We decided to allocate memory on aligned boundaries. Aligned memory allows for faster memory reads/writes. Currently ISPC (version 1.3.0) doesn't support producing optimized code for aligned memory, but we hope future releases will do so. It is also important to mention that this version of ISPC, doesn't have a thread-safe `ISPC_task` implementation. This makes the resulting code base not thread safe.

4. Results

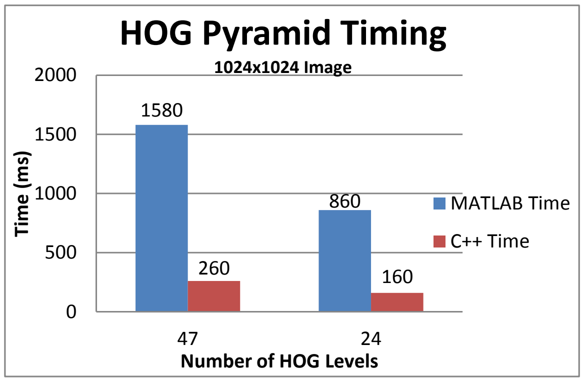
The technologies used for testing and implementing are `gcc` (version 4.6.3), ISPC (version 1.3.0, pre-compiled binary for 64-bit), and the code was written in C++ without the use of any inline assembly or explicit SIMD constructs (apart from those defined by ISPC). We used GNU/Linux (Ubuntu) 64-bit operating system with `pthreads` as our thread implementation. The CPU was Intel Core i-7 2600 (4 physical cores @ 3.4GHz, AVX capable), with 8GB of RAM. We used MATLAB version R2011a (64-bit for Unix) for benchmarking [16]. We also set the number of threads to be equal to the number of physical cores. We did not see any significant advantage in increasing the value beyond this, although it may differ from processor to processor.

We use OpenCV [4] for reading/resizing images[2]. It provides a convenient way to read various image formats like `jpeg`, `png`, `bmp` etc. We also use a C++ template class based on [1] to make accessing image elements easy and efficient.

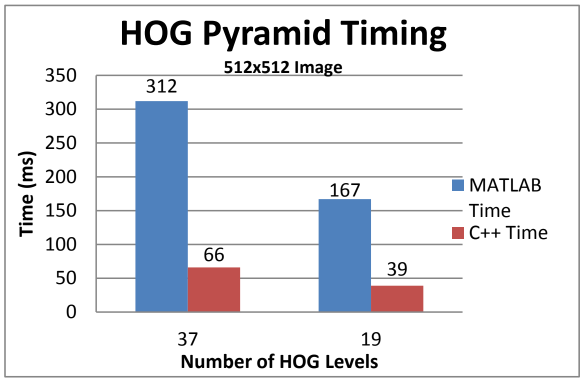
4.1. HOG feature pyramid computation

We tested our implementation of HOG on multiple images by rescaling them multiple times. The rescaling strategy has a parameter called levels per octave (*lpo*). A higher *lpo* implies more levels in the HOG feature pyramid. The Figure 5 show the results for HOG feature pyramid computation. The speed-up of C++ over MATLAB is higher on

[2]The OpenCV image resizing function (`cvResize` with bi-linear interpolation) is roughly 1.3 times faster than the resize function used by [16]



(a) 1024×1024 image at $5, 10$ lpo



(b) 512×512 image at $5, 10$ lpo

Figure 5: Timing results for HOG pyramid computation.

larger image sizes. The computation of HOG feature pyramid about $4 - 7\times$ faster than MATLAB. The split amongst the various HOG subroutines is depicted in Figure 6. Histogram binning (as expected) is the major bottleneck in HOG computation.

4.2. Complete pipeline

The parameters we varied for testing the complete pipeline were the *lpo* and the number of exemplars. These parameters have a direct bearing on the two computationally significant stages of the pipeline - HOG feature pyramid computation, and Convolution.

We observe from Figure 1 that as the number of exemplars increases, the C++ time scales linearly. This is expected since an increase in exemplars corresponds to a linearly proportional increase in the number of convolutions. The MATLAB code [16] implements convolution by setting it up as a matrix multiplication. This approach is faster by at least $2\times$ than the multi-threaded BLAS approach of [8]. All the exemplars are padded and replicated in one large matrix, which is then multiplied with the HOG feature pyramid. Matrix multiplication when tuned to system architecture has been shown to scale sub-linearly [10] (More in Section 4.3). Hence the MATLAB convolution scales sub-linearly with the number of exemplars. The speed-up graphs shown in Figure 2 show that the C++ code is faster than MATLAB for less than 85 exemplars. The number 85 is almost constant across different image sizes and *lpo*. It seems to depend only on the underlying CPU. In the worst case, the C++ code is twice as slow as the MATLAB one. Hence, in the worst case, it is still as fast as [8]. In the best case it is more than 2 times faster. In the case where a user has a large number of exemplars, one can run the C++ code for less than 85 exemplars on different nodes in a cluster and thus benefit from the speed-up.

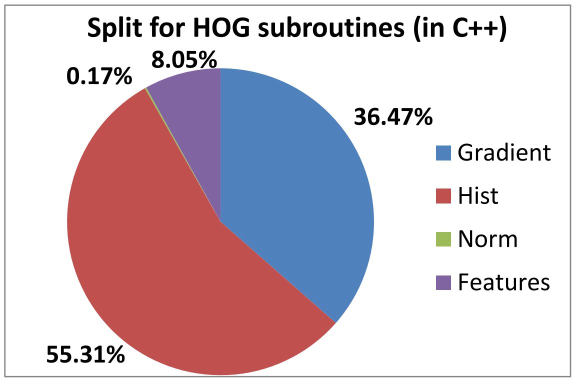


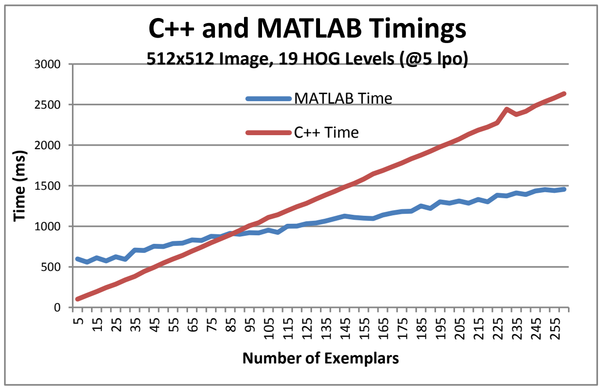
Figure 6: A representative time-split for the subroutines while computing HOG using our C++ implementation. Note that this will vary slightly depending on the cell-width, number of threads and other such parameters. This figure is intended to give an overall idea only.

4.3. Modeling convolution as multiplication

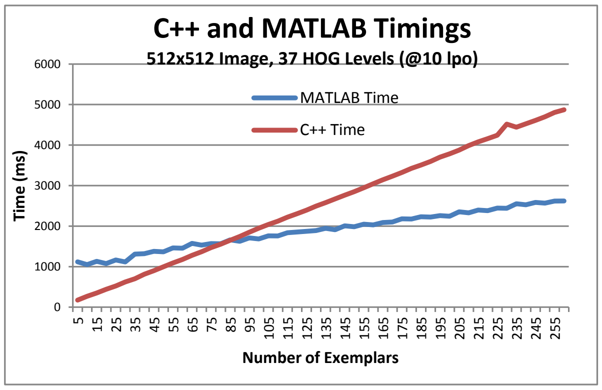
In C++, as the number of exemplars goes up, the number of convolutions to be performed increases linearly. In MATLAB, this is modeled as a matrix multiplication. Although matrix multiplication is inherently an $O(n^3)$ operation, the overall speed of the computation is highly sensitive to the underlying implementation. As discussed in [10], the performance of the current matrix multiplication libraries depends on hand tuning the operations by considering factors such as data locality, packing of data, cache size, Translation Lookaside Buffer (TLB) hit/miss ratio etc. We believe that such fine-tuned optimization of matrix multiplication is the reason MATLAB outperforms the C++ code after certain number of exemplars.

5. Acknowledgements

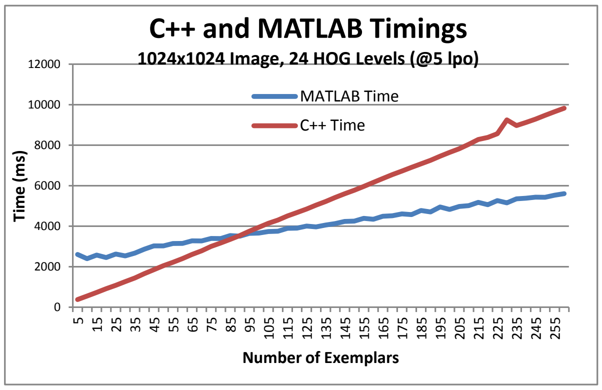
We would like to thank Kayvon Fatahalian for his expertise and valuable inputs throughout the project. His key insights in technologies to use and design decisions helped



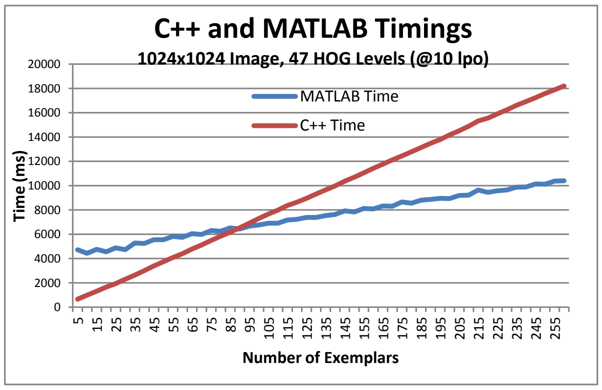
(a) 512 × 512 image at 5 lpo



(b) 512 × 512 image at 10 lpo

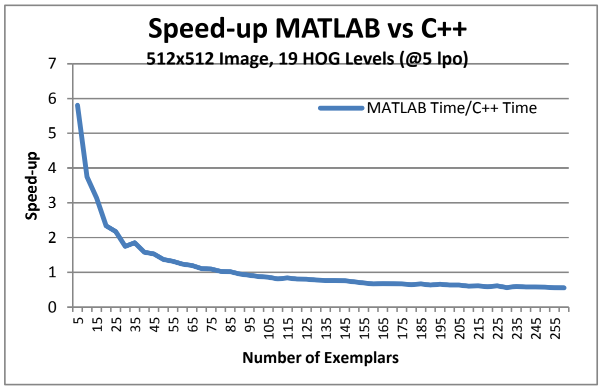


(c) 1024 × 1024 image at 5 lpo

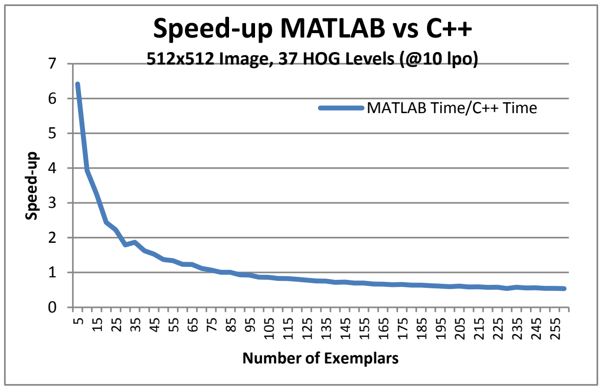


(d) 1024 × 1024 image at 10 lpo

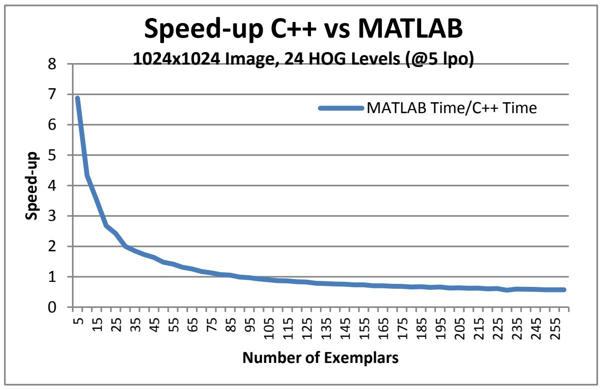
Table 1: Timing results for the complete pipeline. One observes that the number of exemplars at which the MATLAB code is faster than C++, is almost a constant.



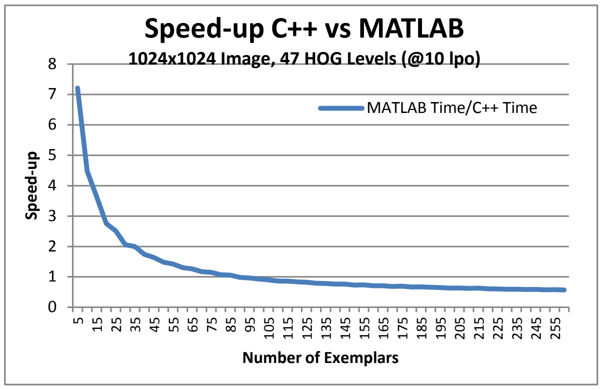
(a) 512 × 512 image at 5 lpo



(b) 512 × 512 image at 10 lpo



(c) 1024 × 1024 image at 5 lpo



(d) 1024 × 1024 image at 10 lpo

Table 2: Speedup results for the complete pipeline. A speedup of less than 1 indicates that MATLAB is faster than C++

us significantly. We also thank the authors of [8, 17] for making their code freely available.

References

[1] G. Agam. Introduction to programming with OpenCV. URL, 2006. 4

[2] J. H. Ahn, M. Erez, and W. J. Dally. Scatter-add in data parallel architectures. In *International Symposium on High-Performance Computer Architecture*, 2005. 3

[3] T. Ball and J. R. Larus. Branch prediction for free. In *SIGPLAN*, 1993. 4

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 4

[5] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005. 1, 3

[6] M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. *SIGOPS Oper. Syst. Rev.*, 43(3), 2009. 1

[7] C. Dubout and F. Fleuret. Exact acceleration of linear object detectors. In *ECCV*, 2012. 1

[8] P.F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part based models. *PAMI*, 32(9), 2010. 1, 2, 5, 7

[9] J. Fung and S. Mann. OpenVIDIA: parallel gpu computer vision. In *International Conference on Multimedia*, 2005. 1

[10] K. Goto and R. A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008. 5

[11] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *International Symposium on Parallel&Distributed Processing*, 2009. 1

[12] Intel Inc. Intel Integrated Performance Primitives, 2002. 1

[13] S. Kumar, D. Kim, M. Smelyanskiy, Y. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. Atomic vector operations on chip multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008. 3

[14] S. Ladra, O. Pedreira, J. Duato, and N. R. Brisaboa. Exploiting SIMD Instructions in Current Processors to Improve Classical String Algorithms. In *Advances in Databases and Information Systems*, volume 7503. Springer Berlin Heidelberg, 2012. 1

[15] R. Leissa, S. Hack, and I. Wald. Extending a C-like language for portable SIMD programming. In *SIGPLAN*, 2012. 2

[16] T. Malisiewicz. https://github.com/quantombone/exemplarsvm, 2011. 2, 4, 5

[17] T. Malisiewicz, A. Gupta, and A. A. Efros. Ensemble of Exemplar-SVMs for object detection and beyond. In *ICCV*, 2011. 1, 2, 4, 7

[18] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. 1

[19] M. Pharr and W. R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing (InPar)*, 2012. 2

[20] A. Shrivastava, T. Malisiewicz, A. Gupta, and A. A. Efros. Data-driven visual similarity for cross-domain image matching. *SIGGRAPH ASIA*, 2011. 1, 4

[21] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions, 1999. 1