Speeding up SIFT for Security Applications

Ben Anglin, Stanislav Bobovych, Robert Brandon

December 9, 2012

1 Introduction

The goal of this design project is to propose a method of improving the performance of existing SIFT implementations. This enhancement is provided in two forms, the introduction of a new ISA-level instruction and the addition of a specialized coprocessor and cache. Specifically, we will allow the most common and processor-intensive pieces of SIFT to be implemented using these specialized structures, rather than relying on a general purpose processor.

2 Background

2.1 Application

For the purposes of this project, the application space of security video surveillance is to be targeted. Our system is designed to allow a security camera's live stream to be analyzed in near-real time, allowing for rapid feature recognition (facial, object, etc). This implementation assumes a fixed input type from a surveillance camera and a reliable processing backend platform.

This usage scenario serves to provide several parameters for the system design. As the input stream from the camera is reliable, we will use the assumption that new data is always available for processing. Because many security systems rely upon custom hardware, we can design a system using some flexibility in terms of power and space requirements. Finally, because we can assume that every frame captured is to be analyzed, we can design our system to explicitly streamline and accelerate image access and processing.

2.2 SIFT

Simply stated, SIFT (Scale-Invariant Feature Transform) is an image processing algorithm. It is used for image recognition and is noted for its ability to recognize elements of an image even when they are rotated or scaled in unusual ways. A full treatment of the algorithm is best left to other papers, but a look at it's core will show what our architecture will impact.

A key part of the SIFT algorithm is scale space construction. This process begins by scaling the image n times (usually shrinking the image). At each scale, a Gaussian blur is applied to the image, generating a new image. This blurring is repeated m times per scale. A set of blurred images at a given scale is known as an octave. In total then, the Gaussian blur is applied to every pixel in n x m images (every image in every octave). After profiling an implementation of SIFT we decided that this part of the algorithm was a good candidate for speedup.

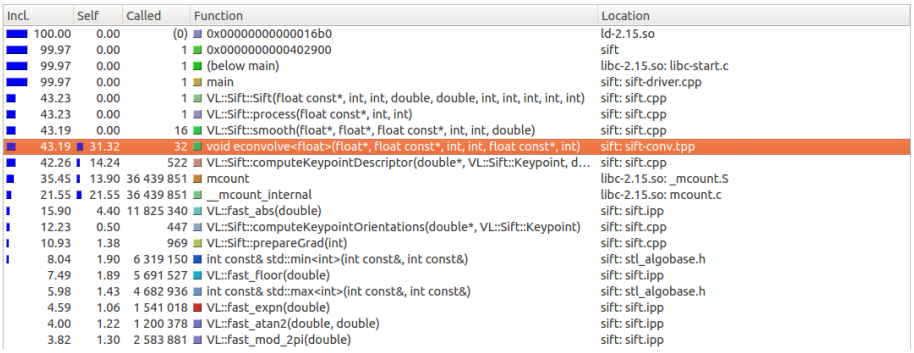
3 Methodology

3.1 Profiling SIFT

Profiling for this project focused on two well known implementations of the SIFT algorithm, SIFT++ and VLFeat. Though initial profiling was conducted on VLFeat, SIFT++ also was used to enable easier analysis as it is less reliant on object oriented structures.

Profiling was conducted by running the sift program included with VLFeat and siftpp included with SIFT++ through Valgrind's Callgrind module. This produced outputs suitable for showing which functions were called most within the SIFT algorithm and how much time was spent in them. In both implementations, the functions handling image convolution proved to take a large portion of the run time. The VLFeat implementation, shown in Figure 1, spends approximately 35% of its running time doing image convolution. Shown in Figure 2 is a similar measurement taken from SIFT++. This shows that convolution a computationally dense piece of code in the two SIFT implementation.

Figure 1



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x00000000000016b0	ld-2.15.so
99.97	0.00	1	0x0000000000402900	sift
99.97	0.00	1	(below main)	libc-2.15.so: libc-start.c
99.97	0.00	1	main	sift: sift-driver.cpp
43.23	0.00	1	VL::Sift::Sift(float const*, int, int, double, double, int, int, int, int, int)	sift: sift.cpp
43.23	0.00	1	VL::Sift::process(float const*, int, int)	sift: sift.cpp
43.19	0.00	16	VL::Sift::smooth(float*, float*, float const*, int, int, double)	sift: sift.cpp
43.19	31.32	32	void econvolve<float>(float*, float const*, int, int, float const*, int)	sift: sift-conv.tpp
42.26	14.24	522	VL::Sift::computeKeypointDescriptor(double*, VL::Sift::Keypoint, d...	sift: sift.cpp
35.45	13.90	36 439 851	mcount	libc-2.15.so: _mcount.S
21.55	21.55	36 439 851	__mcount_internal	libc-2.15.so: mcount.c
15.90	4.40	11 825 340	VL::fast_abs(double)	sift: sift.ipp
12.23	0.50	447	VL::Sift::computeKeypointOrientations(double*, VL::Sift::Keypoint)	sift: sift.cpp
10.93	1.38	969	VL::Sift::prepareGrad(int)	sift: sift.cpp
8.04	1.90	6 319 150	int const& std::min<int>(int const&, int const&)	sift: stl_algobase.h
7.49	1.89	5 691 527	VL::fast_floor(double)	sift: sift.ipp
5.98	1.43	4 682 936	int const& std::max<int>(int const&, int const&)	sift: stl_algobase.h
4.59	1.06	1 541 018	VL::fast_expn(double)	sift: sift.ipp
4.00	1.22	1 200 378	VL::fast_atan2(double, double)	sift: sift.ipp
3.82	1.30	2 583 881	VL::fast_mod_2pi(double)	sift: sift.ipp

Figure 2

A review of both convolution implementations shows two distinct behaviors - looping and memory access. Specifically, convolution steps through each pixel in an image, requiring a nested loop resulting in a O(NM) runtime (for an NxM image). At each stage of this loop, image data is both read from and written to memory. Neither of these attributes is unusual for image processing, but they are nevertheless the most common operations within the SIFT algorithm.

After doing this analysis, it was clear that improving memory performance and parallelizing image convolution was the primary path to achieve the necessary performance in our target application.

A simple calculation of speedup using Amdahl's Law shows that if we are able to fully parallelize the convolution, we can achieve a speedup of 1.54.

$$Speedup = \frac{1}{(1 - 0.35) + \frac{0.35}{\infty}} \approx 1.54$$

3.2 Run time

Running siftpp on a canonical 512x512 image yielded an average wall clock time of 1.36 seconds on an Intel Core 2 Quad Q6600. This is no where near real time performance. Certainly, some of this time is spent doing I/O operations. We tried to profile the I/O operations using strace, but did not have any success. Leaving I/O aside, we accept the fact that we can't get near real time processing of images. Therefore, a 1.54 speedup in processing would still be useful in our application.

4 Results

4.1 High level design

The heart of our system is a Convolution Module(CM). This module is dedicated to performing convolution on input images. It uses a dedicated cache, as well as a new instruction that allows the compiler to take advantage of a massively parallel, specialized convolution pipeline to speed up the convolution process.

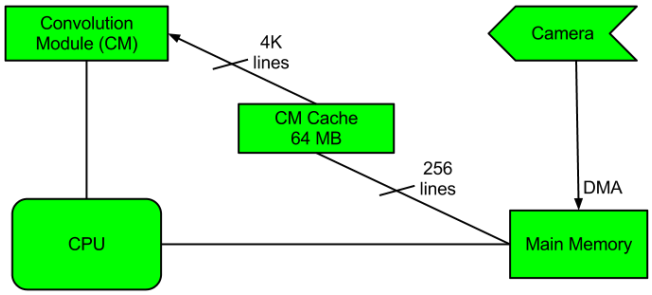


Figure 3

As input, the system uses a grayscale image stream of a camera at a fixed 512 x 512 pixel resolution at any standard frame rate (24, 30, 60 FPS). The camera feed is given DMA access to a fixed region of main memory. This will allow our dedicated cache to easily prefetch images as they arrive. On every hit, the CM cache will prefetch the next set of image data. This ensures a constant flow of data into the CM and ideally eliminates stall cycles caused by cache misses.

During execution of the SIFT algorithm, our new ISA-level instruction will call the Convolution Module (CM) to begin convolving the image stream. Our implementation uses a fixed-size Gaussian kernel to speed up processing. The design we developed allows the CM to deal exclusively with the CM Cache at very high speed. Every cache hit will result in the next image being loaded into the CM module. Once there, every row in the image is convolved in parallel, granting us a massive speedup as discussed later.

4.2 ISA Addition

One of the enhancements in our architecture to improve the performance of the SIFT algorithm is the addition of an instruction to perform the convolution function. Analysis of a profile run of the SIFT process shows that approximately 35% of the time to process an image is spent in the convolution function.

Analyzing a disassembly of the the econvolve function from our reference implementation of SIFT shows that it consists of 162 instructions. While some of these instructions are loops, we will ignore the loop cycles in this analysis since the same loops will be executed on both a traditional CPU and by our Convolution Module. On a traditional CPU this routine must be run for each row of the image being processed. Our architecture adds a single instruction , CONV, to perform a convolve on a row of image data. This function takes 3 arguments, the size of the row in words, a source buffer address, and a destination buffer address. For example, a call to perform a convolve on a 512-byte row of data located at address 0x562B and place the result in address 0xF54A would be notated as:

CONV 512, 0x562B, 0xF54A

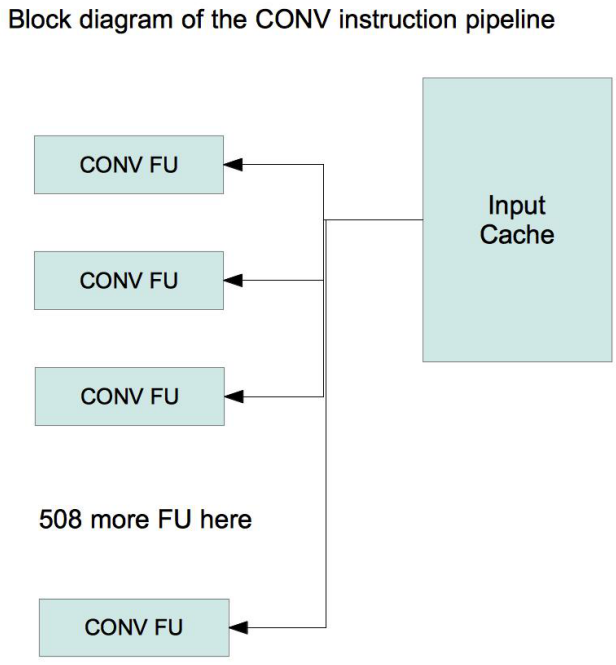


Figure 4

When called, CONV will place the data into a functional unit which performs the actual convolve. This architecture will have 512 of these functional units, with tasking assigned to each assigned by a Tomasulo-type algorithm. For ease of analysis, we will assume that each unit exactly emulates our reference convolve function, so each unit will take around 162 cycles * the average cost-per-instruction to convolute a row of image data. We will assume a CPI of 1 for all calculations, since it cancels out in the comparison. This gives a cost of 162 cycles*512=82,944 cycles on a traditional CPU to perform a convolution for a single image.

In our architecture, the convolutions are performed in parallel, with a single cycle delay to load each row of data into a CONV functional unit. There is also a 1-cycle delay to transfer each result from the functional unit to the destination buffer. This gives a total time of 164 cycles to perform CONV on each row of image data. However, since the operations are parallelized in this architecture, the total time to perform a convolve on an entire 512x512 pixel image is 164 cycles for the first image, plus 512 cycles to load the rest of the rows, for a total time of 676 cycles. This yields a speedup of 82,944/676=122.7 times faster for the convolution function. Using Amdahl's law to calculate speedup, we can see that this implementation yields a speedup that is very close to the theoretical speedup.

$$Speedup = \frac{1}{(1 - 0.35) + \frac{0.35}{122.7}} \approx 1.532$$

4.3 Cache design

4.3.1 Organization

The goal of the special convolution cache is to constantly feed the Convolution Module (CM) with data. We are not concerned with monetary cost or the transistor count of this hardware. We want the CM to execute a single row pass of convolution per call of the CONV instruction. Since our source images are 512x512, we need to give the CM module a cache line that is 512 words, with each word being 8 bits. To accomplish this, the interface to the cash from the CM has to be a 512 word wide bus. To reduce the hit time on the cache, we will use a direct mapped cache. The cache will use virtual addresses to once again keep hit time as low as possible. To avoid capacity misses, we need to use a cache that is at least big enough to keep an entire working set for an image in the cache. To avoid cold start misses, the cache will prefetch data from memory over a dedicated 8 word bus.

Since our application creates 3 octaves with 3 levels per octave per image, the total amount of memory used by one image will be 32x512x512 or 64 MB.

This stems from the fact that we convolve an image across the rows, store that result, and then convolve the output of the row wise computation and store its output in the original image. This saves us a 512x512 chunk of memory per convolution.

Storing the convolution set of a single image in the cache is not enough because we want to keep the CM unit busy convolving images that are coming from the camera. Since a single image working set takes up so many resources, we can only store a few working sets in cache. Without simulation, it is impossible to figure out the optimal number. We choose to store only four working sets, making the total size of the cache 512 MB. This will enable us to do convolutions and other computations on one image while we prefetch the next images to be convolved.

For the convolution results to be usable by the rest of the algorithm, the convolution cache has to be accessible by the regular CPU. To simplify the design, the convolution data's virtual addresses will be fixed. When the application code needs to access the convolution data to do calculations, a normal memory access will occur through a load instruction. Our convolution cache will be connected to the same bus as the level 1 cache and will service the load instructions that use the special virtual address range reserved for convolution data. This necessitates that our cache can handle two read requests in the same cycle and that supporting this does not affect hit time. We assume that the level 1 cache can ignore requests for this special address range and that this does not affect hit time on the level 1 cache.

4.3.2 Performance

With this cache organization, we expect the following performance. Due to prefetching and large capacity, we will have a hit rate from the CM close to 100%. We assume that our system would be turned on once and expected to operate indefinitely, so initially the miss rate would be really high but quickly converge as the system operates. The hit rate from the CPU should also be close to 100% for the same reasons. Thus, a single convolution can be performed in (3*512)*2 CPU cycles. (Load + execute + store) Since a single image requires 16 convolutions, the total number of cycles to process an image would be 49152. Since our interface to main memory is 8 words wide, it will take 64*512 = 32768 cycles to prefetch the next image to be convolved. This is less than the time it takes to convolve an image. We assume that while one image is being convolved and another is being fetched from memory, the application can work on the intermediate results and that there is never a stall in the Convolution Module. This is why we chose to store four working sets in the cache instead of just one or two.

5 Conclusion

An existing SIFT implementation can be recompiled with a compiler that can take advantage of our custom ISA. This eases the burden on software developers. This hardware accelerated design should provide a significant improvement in image throughput for a security system. Also, it leaves the majority of the general purpose processing in the system available for more advanced functions, such as facial and object recognition. An extension of this system would be to have several CPUs with CMs and the special caches to process several frames in parallel. This would bring us closer to achieving near real time performance.