Streamlining the Intel x86_64 Architecture for Machine Learning and Image Classification

Joe Lagnese, Malik Jackson, Peter Gottleib, Muhammad Islam

I. Introduction

We propose two modifications to the Intel x86_64 architecture in order to improve the speed at which image classification software can be executed. In particular, we propose an Instruction Set Architecture change to improve code that converts the RGB color floating-point values of pixels from the range of 0-255 to a continuous range of 0-1. Our hardware changes aims to improve performance through the introduction hardware to support fused add-divide and divide-add operations.

II. Application Characteristics

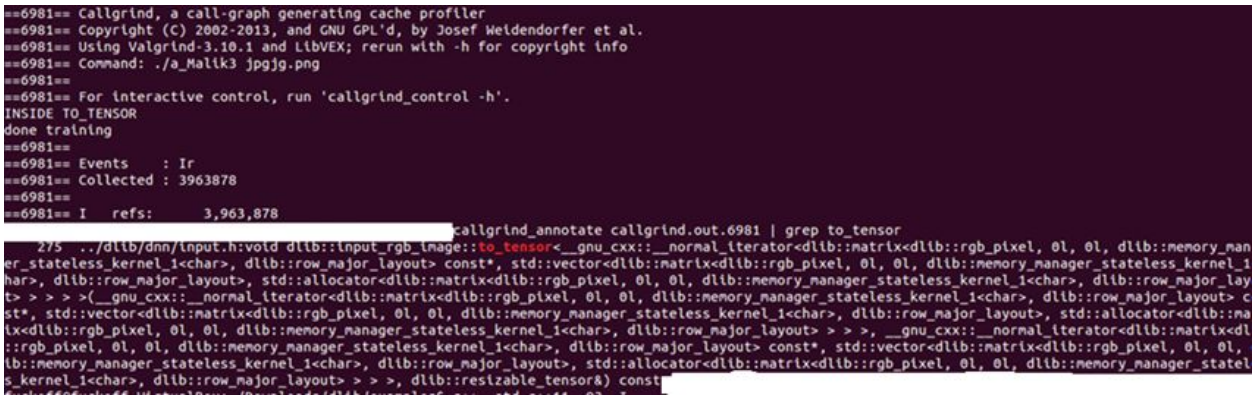
A. Target

Our target application uses supervised machine learning in order to train a classifier to identify images or vectors representing images to a set of labels. We modify code from examples in the Dlib C++ library, which provides various types of data processing tools to build machine learning models, process images and operate on data structures such as arrays, vectors and matrices [9]. The toolkit makes heavy use of vector operations as a lot of machine learning involves the adjustments of weights in a weight vector to minimize error when training a classifier on test data. Classification is notorious for requiring significant time and computational power to perform as it often involves training until convergence which should be a slow process in order to achieve the best accuracy. For our application, we run trivial examples, training datasets on as few as two or four vectors containing as few as 12 to 75 elements in order to keep manual analysis and calculations manageable.

B. Architecture

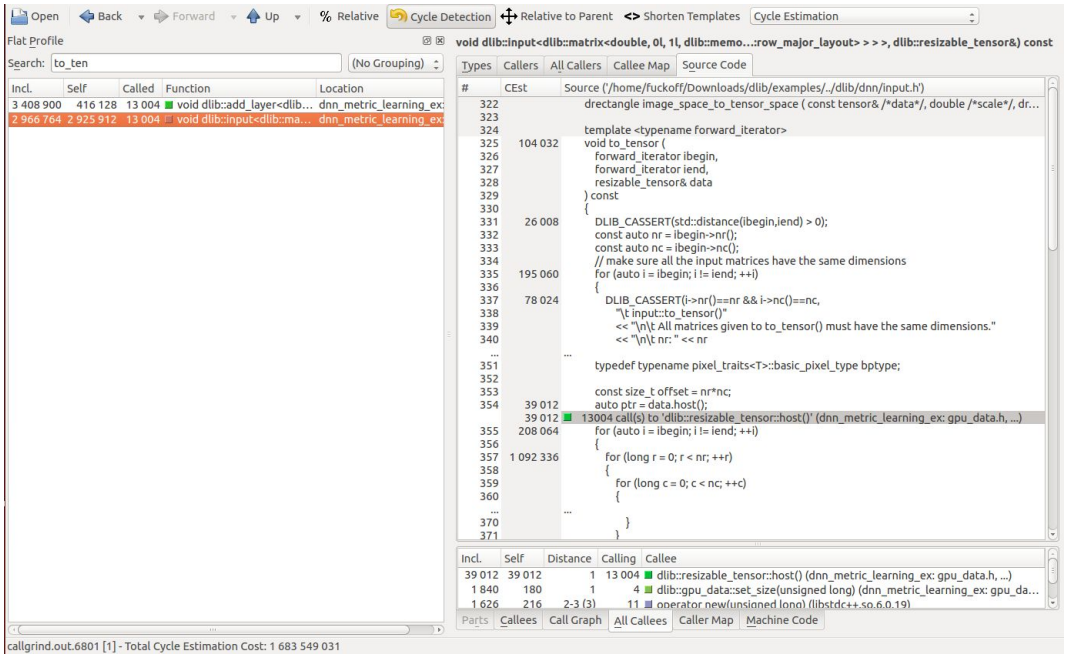
Our modifications are intended to be used with early Intel x86_64 architecture. We decided to use this architecture as a base as our source machine for collecting and processing metrics also ran on x86_64 hardware (Ubuntu 14.12 on an Intel i7). Unfortunately, this choice in architecture meant that we were unable to easily compile our source code directly into MIPS assembly code (MIPS architecture being our first choice), although we did experiment with cross-compilation before our final decision. While we had originally planned to make modifications for a MIPS architecture, the existence of systems for parallel vector operations in the x86_64 architecture was helpful in the construction of our hardware modifications and provided a good foundation to create a more specialized architecture for primarily vector operations.

C. Tools and Metrics



Valgrind and the callgrind profiling tool allowed us to dissect program execution into more into distinct chunks.

We used Valgrind's Callgrind profiler to determine calculations involving the number of cycles a program or function took. We attempted to use kcachegrind as well, but found that it produced results which seemed to conflict with the output of callgrind_annotate, so we decided to exclusively use callgrind and callgrind_annotate for cycle calculations. We also used Godbolt, an online utility that allows one to compile a variety of coding languages using various compilers [1]. While it was helpful in helping us understand the underlying instructions constituting the code we wanted to improve, we did not find much use for its output outside of that, as we still needed the CPI of any code to calculate a speedup which required us to calculate the total cycles anyway using callgrind.



Kcachegrind provides a powerful visual interface for extensively diving into source code, analyzing connectivity between functions and searching sources of cost. All too often however, the application provides too much specificity in its breakdown such that user may be misled from its findings.

III. Design

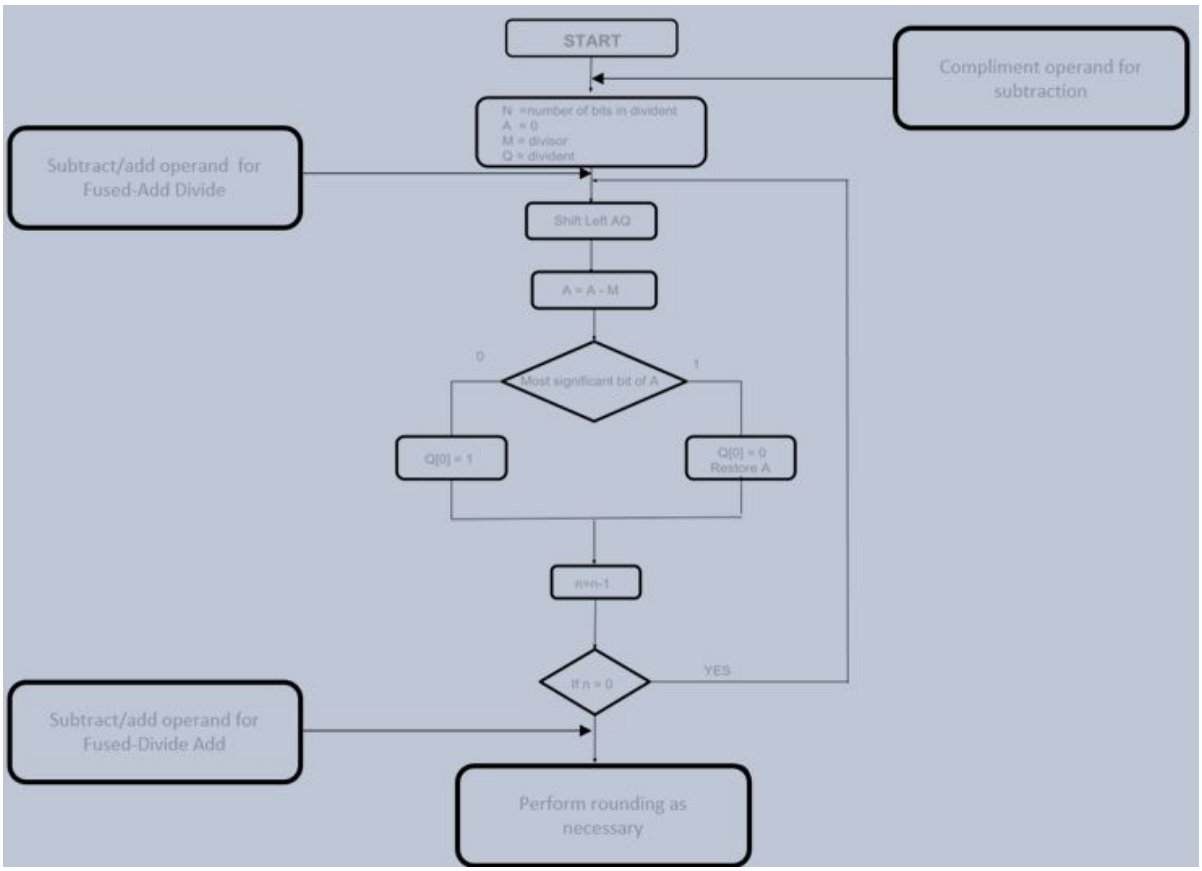
A. Hardware

Intel's x86_64 architecture is a 64-bit mode extension of its original x86 architecture. Dubbed the Intel64 from the original implementation of the x86_64 from AMD, the architecture provided technological advancements in high performance computing, image processing and machine learning. Our hardware design improvement specializes the Intel64 architecture for operations on image processing and machine learning. We propose specialization by extending ALUs with new hardware to support hypothetical Floating Point Fused Divide-Add and Floating Point Fused Add-Divide Instructions.

With their Advanced Vector Extensions, released around 2008, Intel upgraded their Single Instruction Multiple Data vector capacity with 16 new YMM registers, alongside their 16 XMM registers. With Advanced Vector Extensions 2, Intel introduced the Fused Multiply-Add to their instruction set architecture. Addition and multiply instructions are widely useful for matrix operations in areas such as graphics and processors. However, in image processing, division is also valuable. One might use RGB pixel addition/subtraction for digital imaging or filtering, and in most image processing programs, RGB values are divided by 256 to restrict their range between 0-1.

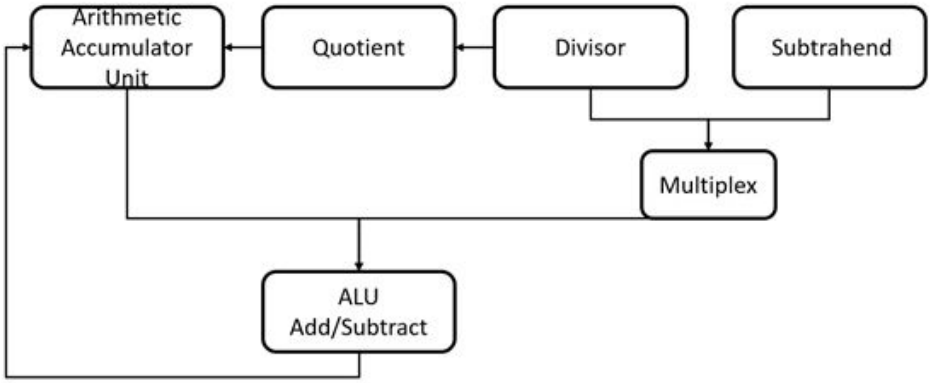
Fused Multiply Add architecture adds the operand to be summed directly to the end of the partial product generation from multiplication. In Floating Point Fused Multiply Add, the

product and sum of the three operands are computed before rounding down to the intended number of bits and then saving to a memory location.



A data flowchart for our proposed Fuseed Add Divide/Fused Divide Add implementation.

Our Fused Divide implementation would similarly provide several opportunities for speedup between individual operations. We use the restoring method of division and propose identical hardware with control for the occurrence of our with our addition- or a complemented operand for subtraction. Similar to Fused Multiply architecture, we will make a very small change in an existing arithmetic architecture to accommodate our fused operation. Save an accumulator, we will not save results to a register before either operation is completed and similar to Floating Point Fused Multiplication, we will round our results after both operations have been completed. The figures above and below illustrate our algorithm and hardware concept for our Fused Divide Architecture.



A basic overview of our FAD/FDA architecture.

The architecture and data flow plath retain most qualities of a restoring divide. Since subtraction already occurs in throughout the division, it is a relatively simple process to add an additional subtraction operation to the architecture with a mux to control when the subtraction takes place. When sent before the division starts, the architecture effectively becomes a Fused Add Divide, and otherwise it is a Fused Divide Add.

With these hardware design changes, we make the following assumptions:

1. The x86_64 and relevant processor implementing has sufficient space to for the hardware to support the extra arithmetic operation.

2. Appropriate measures have been taken to support a Fused-Add Divide and Fused-Divide Add instruction

3. Upon operation selection, a mechanism like a binary a switch is set in the multiplexor of the architecture controlling the order the subtraction and division occur in.

4. Fused Division is made available for all common number types including integer, float and double. Proceeding the operation, the result is rounded to the appropriate value.

5. The architecture of the proposed Fused Dsivision is able to be vectorized.

B. Instruction Set Architecture

The target snippet of code for our ISA modifications was the following nested loop (see Appendix A for the full containing function), written in C++:

```
const size_t offset = nr*nc;
auto ptr = data.host();
for (auto i = ibegin; i != iend; ++i)
{
    for (long r = 0; r < nr; ++r)
    {
        for (long c = 0; c < nc; ++c)
        {
            rgb_pixel temp = (*i)(r,c);
            auto p = ptr++;
            *p = (temp.red-avg_red)/256.0;
            p += offset;
            *p = (temp.green-avg_green)/256.0;
            p += offset;
            *p = (temp.blue-avg_blue)/256.0;
            p += offset;
        }
    }
    ptr += offset*(data.k()-1);
}
```

We recognized that this code could be streamlined by removing the need for branch prediction, using the fact that the loads and stores were easily predictable, using the fact that we are always dividing by 256, and using the fact that we really only needed to load the values of avg_red, avg_green, and avg_blue once to use for all the subtraction operations. To take advantage of these features, we designed the instruction VCSD (short for **V**ector **C**olor **S**ubtract and **D**ivide), the encoding of which is given below (based on the bit/byte breakdown given in the Intel 64 Software Developer's Manual) [6][7]:

01000000	00001111	11111111	11AAALLL
REX	Opcode (0F FF)		ModRM

We expected to only be working with float values, so we did not need any of the 64-bit registers and thus didn't need the REX.R or REX.B bits, nor did we need the REX.W or REX.X bits to be set (in other words, the last 4 bits of the REX portion are 0 for this reason). For the opcode, we found from [4] that the 0F FF opcode was unclaimed. If this opcode is claimed or reserved in a more modern processor, this value could be changed, as we used these bits purely as a unique identifier for our instruction. For the ModRM bits, we assumed that the bits labeled AAA would be replaced by the binary value representing the register which held the starting address of our vector before the instruction was run, and LLL would be the register holding the length of the vector. We calculated that a 32 bit register would be large enough for this value, as

we anticipated a maximum image matrix of 4096x4096x3 = $2^{12}*2^{12}*3 \approx 2^{26}$. Even if the vector length was represented as the actual length of the vector in memory (so that it is 2^{26} * sizeof(float) = $2^{26}*2^5 = 2^{31}$), this should still fit in a 32-bit register. As far as we understood x86 instruction encoding, the Mod bits would be 11 as we wanted to use the values given in the two registers for our operations.

This instructions works by first loading a value from the 3D vector/array/matrix, then subtracting either avg_red, avg_green, or avg_blue from the value (which we assume are preloaded into registers), and then subtracting 8 from the exponent of the result to perform the division by 256. This new value is then stored into the location from which it was loaded, and the address from which to load the next value is incremented by 32 (the size of a float in memory). This new location is compared to the length of the vector to determine if there are more elements in the vector left to scale or if the instruction has finished. Note that in the code snippet above, this instruction will essentially perform the function of the code inside the "for (auto i = …)" loop, but will not replace the i loop itself (which loops across input). It is also worth noting that we decided *not* to include a destination address for our instruction, instead assuming that we would overwrite the values as they are modified.

For this instruction to function ideally, we made the following assumptions about our hardware, software, and pipeline:

(1) Three special-purpose 32-bit floating-point registers exist for storing color float values. Our VCSD instruction will assume that these registers contain the values avg_red, avg_green, and avg_blue, respectively.
(2) A MUX exists that can determine whether avg_red, avg_green, or avg_blue should be used for each subtraction, and can influence the datapath such that the correct corresponding register is used for subtraction.
(3) Our color vectors are stored contiguously in memory, meaning each trio of RGB values representing a pixel is stored sequentially, each pixel in a row of pixels is stored sequentially, and each row of pixels is stored sequentially.
(4) An Adder exists that can increment the memory access location for each load/store pair to the next point in our vector/array. Since we are working with vectors of floats (which should be stored as 32-bit floating point numbers), we should be able to simply increment by 32 bits from the starting location of our vector to access each value.
(5) Another MUX exists that can determine when we have reached the end of our vector using the starting address of the vector and length that was passed in. This MUX should be able to control the datapath such it prevents our instruction from moving any further through memory if it has reached the end of the vector.
(6) A sufficient pipeline exists such that we can maximize how often we perform loads and stores. We still assume that we can only perform *either* a single load or a single store each cycle.

(7) A sufficient pipeline exists such that we can maximize how often we are performing subtraction operations, still assuming only one can be occurring in a given cycle.
(8) An Adder exists that can subtract 8 from the exponent from a floating number to perform division by 256 in a single cycle. We also assume that sufficient systems are in place to handle any underflow.

While it is likely on a modern system that there is sufficient hardware in place to perform multiple load/stores or subtractions in the same cycles, we limit ourselves to operating on a single unit at a time so that other instructions may be completed in parallel.

IV. Justification and Analysis

A. Hardware

While were unable to directly test the potential speedup and performance of our Fused Division on x86_64 hardware, we can take measures to appropriately bound and estimate. In the best case, the hardware executes a floating-point Fused Divide-Add/Add-Divide in the same amount of time it would take for a regular restoring based division to execute. In the worst case, it should take about the same amount of time to sequentially execute a restoring divide and a subtraction.

We were able to find the latency and reciprocal throughput of instructions from the tables provided by Agner Fog [2]. In his documentation, latency is defined as " [The minimum delay] that the instruction generates in a dependency chain" and the reciprocal throughput indicates "... issue latency. This value indicates the average number of clock cycles from the execution of an instruction begins to a subsequent independent instruction of the same kind can begin to execute" [2]. For SkylakeX, one of Intel's more recently released processors, the tables list the latency for vector instructions, add, subtract multiply ***and*** Fused Multiply-Add as 4 cycles and reciprocal throughput as a range between .5 and 1. This means that the documentation backs a 2x speedup for fused multiply add. For the vector divide instruction however, the latency is 5 cycles and the reciprocal throughput is 11. The latency of Fused Divide should then be 9 cycles at its slowest and 5 cycles at its fastest.

Looking out to further research, we discovered similar proposed hardware with some real performance claims. Pande et al. suggest an implementation of a non-interchangeable Divide-Add architecture using SRT division [8]. They measure a 1.3416 speedup between their architecture and a division followed by subtraction [8]. Since SRT division is more closely related to non-restoring division and both usually outperform the restoring method of division, we expect an ideal performance of our architecture somewhere between 1 and 1.3416.

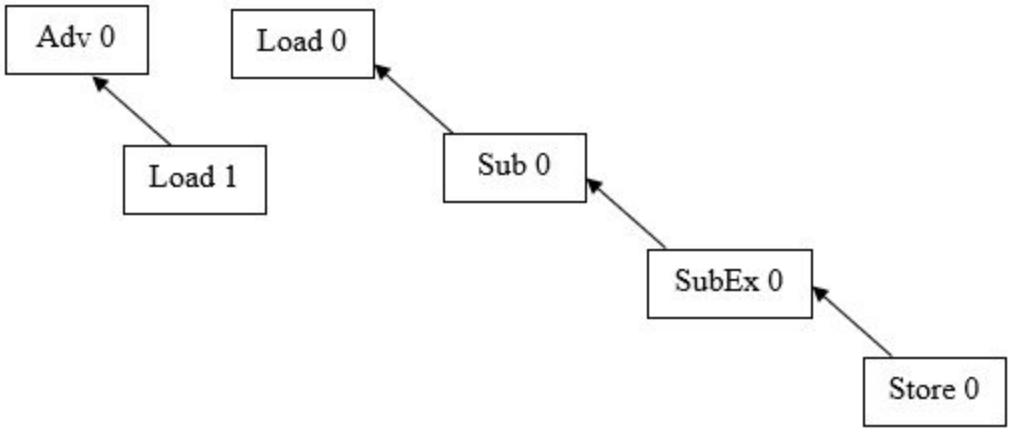
During our work in examining the effects of the ISA changes, we observed that around .2% to 1% of our total cycles used in our example image processing and machine learning applications were related to operations relevant to Fused Division. While this number may seem insignificant, it grows nonlinearly with the size of our sample data. For this experiment, we choose smaller vector sizes to keep values reasonable enough for manual analysis. In a realistic

environment, we expect the amount of input data, and thus the percentage of cycles spent on divide-add operations to rise exponentially. This makes our hardware viable for image processing and certain mathematical machine learning applications. In the following table, to_tensor() is a function which performs a divide-add (or rather, divide-sub), and is the function which contains the code which we used as the basis for our ISA changes.

Input vector size	**Total cycles of example program**	**Total cycles for to_tensor() (performing divide-add)**	**% of total cycles for to_tensor()**
12 (2x2x3)	1,043,786,580	2,963,075	.28387%
27(3x3x3)	969,694,456	5,191,668	.535392%
48(4x4x3)	18,231,876,184	101,757,216	.55812%
75(5x5x3)	41,555,602,109	335,008,112	.80616%

B. ISA Changes

In order to determine the cycles our new instruction would take, we had to first examine its intra-iteration and inter-iteration dependencies. A rough data flow graph for our instruction is shown below:



The bubbles in the diagram represent the following components of our instruction:

- "Load" -- the loading of a value in our vector from memory into a register. "Loads" will mainly be limited by our assumption that we can only load one value per cycle, but technically must also wait for the "Adv" operation to complete as it provides the next memory location to access.
- "Adv" -- both the incrementing of our "pointer" to the next value to load from memory as well as the checking of this value against the end location of our vector (calculated in parallel with the first load using the starting address and vector length). Theoretically, the current memory address to be accessed could be simultaneously used as the location for the current "Load" while also being passed into an Adder to increment for the next "Load" operation, so this operation should not depend on the completion of the current "Load".
- "Sub" -- the subtraction of avg_red, avg_green, or avg_blue from the value we loaded from memory. We can determine which value to subtract in parallel with the "Load" using a MUX, but we cannot determine what we are subtracting **from** until the "Load" has completed.
- "SubEx" -- the subtraction of 8 from the exponent of our floating-point value to replicate division by 256. We cannot execute this operation until the "Sub" operation on our value is complete.
- "Store" -- the storing of the resulting value in memory. This cannot be done until the division by 256 is completed in the "SubEx" operation.

Using the latencies from Agner Fog's instruction tables, we assumed that the "Load" and "Store" operations would take 2 cycles each given the latencies for the MOV operations of fairly recent Intel Processors (Pentium 4 and Skylake) being 1-3 cycles [2]. We also assumed that a "Sub" would take 3 cycles based on the timing for an FADD / FSUB on the Skylake. Since "Adv" and "SubEx" just involve Adders/MUXs, we assumed they would take a cycle each. With these latencies and the above diagram, we reached an execution for our instruction that looked something like the below table, where each "iteration" represents running our set of operations on another float from our vector.

Cycle	**Iteration 0**	**Iteration 1**	**Iteration 2**
0	Load/Adv		
1	Load		
2	Sub		
3	Sub		
4	Sub	Load/Adv	
5	SubEx	Load	
6	Store	Sub	
7	Store	Sub	
8		Sub	Load/Adv
9		SubEx	Load
10		Store	Sub
11		Store	Sub
12			Sub
13			SubEx
14			Store
15			Store

Extrapolating this pattern to N iterations (for a vector with N floating-point elements), we got this formula as our estimation of the cycles for our instruction:

Estimated # of cycles for VCSD = 4N + 4

However, this estimate did not include the moving of avg_red, avg_green, and avg_blue into their special-purpose registers beforehand, nor did it include the loading of the base memory address of the vector or length of said vector into registers. We assumed that these 5 MOVs would have to occur before each execution of VCSD, although we did assume for these instructions that there are enough loading stations (such as those for a Tomasulo implementation) to perform these loads in parallel as they come. That said, we still had to consider the 2-cycle latency of the final MOV, so we had 6 additional cycles added to our cycle estimate, giving us:

Estimated # cycles for VCSD (including pre-loads) = 4N + 10

We planned on using callgrind and callgrind_annotate in tandem with kcachegrind to profile the target code, but encountered the following issues:

- The cycle counts provided by kcachegrind for various functions disagreed with those provided by callgrind_annotate. We chose to use the values provided by callgrind_annotate as our "true" values.
- dlib has different classes for rgb_images and matrices. The code which we had replaced was the main body of the function to_tensor() in the rgb_images class, but the example code used to get our total cycles data utilized the matrix class, whose to_tensor() function differed (it did not subtract anything from its vector values and only acted on one value in the innermost loop) [Appendix B]. When we attempted to replace the vector of matrices using for training with a vector of rgb_images, the program frequently crashed after a single iteration despite our attempts to debug.
- As seen in our attached picture of the full to_tensor() function which contained our target code [Appendix A], the code which our instruction was intended to speed up was not the entirety of the function, which meant that we were not sure we could consider the total cycles for to_tensor() as "the total cycles for the code we were replacing." Upon trying to extract the code into its own function, we ran into a host of dependency and scoping issues, and decided against it given that the double nested loop does constitute the main work of the function.

Despite these concerns, we were able to obtain the data in the following table using callgrind and callgrind_annotate. For each row, we provided the example program with vectors which could feasibly represent the colors for an image (since a vector is still a matrix). For example, in the first record in the table, we provided the program with vectors of length 12, representing a 2 pixel by 2 pixel image with RGB values for each pixel (2*2*3 = 12 values).

The leftmost column thus gives the length of the vector provided and the size of the analogous image (times 3 for the RGB for each pixel). The next two columns show the total cycles taken by the program to execute as well as the cycles for to_tensor() to execute according

to callgrind. We used these values to calculate the % of the total cycles for the execution of the program resulting from to_tensor() calls. We used print statements to figure out how many times to_tensor() was called, as well as how many times the outer i loop occurred (knowing that our instruction was only "replacing" the code inside that loop), the results of which are included in the fourth and sixth columns. We made sure not to include the print statements when calculating the total cycles of the program or to_tensor(). The final column shows the cycles callgrind reported for the single iteration of to_tensor() from the rgb_image class that we were able to run for an image of each size (2x2, 3x3, 4x4, and 5x5 pixels). In each case, we provided four vectors as input samples for the program, expecting the i loop to thus execute four times per call of to_tensor().

To ensure that the single iteration's worth of timing we got from the rgb_image's to_tensor() function was valid, we tested a single call of the matrix class' to_tensor() function with four 4x4x3 input vectors and compared it to the average cycles per to_tensor() call for the same input vectors in our below table. We found these two values to be equal for all tests, and so assume that the cycles for a given call of to_tensor() changes little if at all between calls for either class.

Input vector size (analogous image dimensions)	**Total Cycles of example program**	**Total cycles for to_tensor() in example**	**to_tensor() calls**	**to_tensor() % of total cycles**	**i loop executions**	**Cycles for 1 iteration of to_tensor() with rgb_image**	**Anticipated cycles for one to_tensor() with VCSD**
12 (2x2x3)	1,043,786,580	2,963,075	8,845	.284	35,380	164	(4*12+10)* 4 = 232
27(3x3x3)	969,694,456	5,191,668	8,163	.535	65,304	275	(4*27+10)* 4 = 472
48(4x4x3)	18,231,876,184	101,757,216	96,361	.558	385,444	427	(4*48+10)* 4 = 808
75(5x5x3)	41,555,602,109	335,008,112	207,307	.806	829,228	614	(4*75+10)* 4 =1240

From this table, we were able to obtain the following information:

- Most significantly, our VCSD instruction would actually appear to incur a *slowdown* rather than a *speedup*.
- For the 27-element vector tests, the i loop consistently ran 8 times per to_tensor() in the matrix class, while the i loop only ran the expected 4 times per to_tensor() call for every other input size. We are unsure why this occurred given that we always tested with only four input vectors.

- to_tensor() constitutes a very small percentage of the cycles for the total program, but grows in significance as the image size increases.
- The change in the cycles for each call to to_tensor() does not seem to directly correlate to the change in i loop executions, meaning that while to_tensor() takes more cycles as the loops which our code affects run for longer, the effects of our instruction on the function may not dwarf the effects of the other existing code that we are not changing. This means that our overall speedup calculation is inherently flawed, as we assume that the % of cycles which our new instruction affects is equal to the % of cycles that to_tensor() takes out of the total cycles for the program.
- The changes in any given value (total cycles for program, total cycles for to_tensor(), to_tensor() executions, etc.) seem to vary wildly for different input sizes. The 27-element vector is particularly interesting as the cycles for the program and to_tensor() calls actually *decrease* while the total cycles taken by the to_tensor() calls still *increases*.
- The to_tensor() method from the rgb_image class seems to take significantly less cycles than the matrix class' to_tensor() method does on average. It also seems to increase in how much faster is relative to an implementation with our VCSD instruction as the image size increases.
- We also tested the program with the color values represented as doubles instead of floats, and found that in almost every case the total cycles for the program decreased slightly (on the order of ~ 5%) while the cycles for to_tensor() decreased greatly (on the order of 40%+). The only exception to this was in the 12-element vector case, where the total cycles actually increased by around 20%, while the to_tensor() cycles only decreased by about 20%.

It is hard to glean a realistic speedup for our ISA change from this table due to the high deviation in the data and severe lack of data points. While we wanted to test this program on larger vectors and images, we found that even these current tests took several minutes to compile and run, and that time was doubled as we needed to run each test twice to also see how many times to_tensor() or the "i loop" were executed. That said, if we take the average % of the total cycles that to_tensor() constituted, we get the following as an estimate of the % of all cycles that our ISA change will affect:

$$\textit{\% affected} = (.284\% + .535\% + .558\% + .806\%) / 4 = .54575\%$$

Averaging the speedup (or rather, slowdown) that our instruction achieves for a single iteration when compared to the to_tensor() from the rgb_image class, we get the following speedup for this portion of the program:

$$\text{Speedup}_{\text{Local}} = (164/232 + 275/472 + 427/808 + 614/1240) / 4 \approx .57929$$

Since the clock rate will theoretically not change, we can use Amdahl's Law solely in terms of the change in cycles to calculate the overall speedup:

$$\text{Speedup}_{\text{Overall}} = 1 / ((1 - .0054575) + (.0054575/.57929)) \approx 0.9960$$

Therefore our ISA change will roughly lead to a .4% decrease in speed for this program.

V. Future Work

Overall, we felt that our project went fairly well despite the multitude of issues we ran into and unsureness we felt with attempting some of our changes. While we were disappointed in the slowdown from our ISA change, we felt that we achieved our initial goals in terms of what we wanted to accomplish and did not have any gross errors in the way of erroneous assumptions or clear oversights. The biggest issues we found with the project were that our initial scope was too large, our target codebase was very complex and highly abstracted, and our choice to make modifications to an "Intel" processor (which we did not work with much in class) meant that we were frequently unsure of the validity or originality of our changes.

If we were to attempt a similar set of modifications in the future, our first change would be to greatly simplify our target code. Getting reliable and consistent cycle counts is incredibly difficult when running machine learning training code, since the code essentially runs until the learning rate becomes smaller than a given epsilon and so does not always run for a fixed number of loops. This means that without a large number of runs of the code (which we did not have the time to perform), the cycle count will be an estimate at best. In addition, the library we chose to use for our baseline code was greatly abstracted to the point that functions became almost unreadable in some cases (as shown in the image in section II showing callgrind_annotate's output). Furthermore, the extensiveness of dlib was actually a hindrance at times, as it meant that even small changes resulted in compiles that took minutes long not including the minutes-long runtime of the program.

We also would not try to plan modifications for a CISC architecture in the future, especially an Intel one. It was understandably difficult to find any information on the most recent Intel systems as far as the timing of operations or underlying hardware/pipeline were concerned, and understanding how to encode an instruction in x86-64 was a challenge in and of itself. These issues combined with those stated above forced us to make a multitude of assumptions about the effects of our modifications, such that we felt uncomfortable with the true accuracy of our speedup claims. We are reasonably certain that the reason our ISA change was a significant slowdown rather than a speedup was that either that our latencies were incorrect or our assumption of what could be done in parallel in our instruction (memory accesses or subtraction operations in particular) was a gross underestimate. Our primary improvement in future work in this area would be to eliminate the need for all the estimations and assumptions that we used by working on a processor that we can get more explicit answers for and which is closer to our "ideal" processor both for the ISA modifications and for our baseline for our hardware modifications.

Works Referenced

[1] - Godbolt Compiler Explorer https://godbolt.org/
[2] - Instructions Tables for Intel, AMD and VIA CPUs
https://www.agner.org/optimize/instruction_tables.pdf
[3] - x86 Opcode and Instruction Reference http://ref.x86asm.net/
[4] - List of x86_64 Instructions and Opcodes http://ref.x86asm.net/coder64.html
[5] - Intel 64 Software Developer's Manual Vol. 1
https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf
[6] - Intel 64 Software Developer's Manual Vol. 2
https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
[7] - A Beginner's Guide to x86_84 Instruction Encoding
https://www.systutorials.com/72643/beginners-guide-x86-64-instruction-encoding/
[8] - Pande, Kuldeep, et al. "Design and Implementation of Floating Point Divide-Add Fused Architecture." *2015 Fifth International Conference on Communication Systems and Network Technologies*. IEEE, 2015. https://ieeexplore.ieee.org/document/7280029
[9] - Dlib C++ Library http://dlib.net/

```
template <typename forward_iterator>
    void to_tensor (
        forward_iterator ibegin,
        forward_iterator iend,
        resizable_tensor& data
    ) const
    {
        DLIB_CASSERT(std::distance(ibegin,iend) > 0);
        const auto nr = ibegin->nr();
        const auto nc = ibegin->nc();
        // make sure all the input matrices have the same dimensions
        for (auto i = ibegin; i != iend; ++i)
        {
            DLIB_CASSERT(i->nr()==nr && i->nc()==nc,
                "\t input_rgb_image::to_tensor()"
                << "\n\t All matrices given to to_tensor() must have the same dimensions."
                << "\n\t nr: " << nr
                << "\n\t nc: " << nc
                << "\n\t i->nr(): " << i->nr()
                << "\n\t i->nc(): " << i->nc()
            );
        }

        // initialize data to the right size to contain the stuff in the iterator range.
        data.set_size(std::distance(ibegin,iend), 3, nr, nc);

        const size_t offset = nr*nc;
        auto ptr = data.host();
        for (auto i = ibegin; i != iend; ++i)
        {
            for (long r = 0; r < nr; ++r)
            {
                for (long c = 0; c < nc; ++c)
                {
                    rgb_pixel temp = (*i)(r,c);
                    auto p = ptr++;
                    *p = (temp.red-avg_red)/256.0;
                    p += offset;
                    *p = (temp.green-avg_green)/256.0;
                    p += offset;
                    *p = (temp.blue-avg_blue)/256.0;
                    p += offset;
                }
            }
            ptr += offset*(data.k()-1);
        }

    }
```

Appendix A. The full function to_tensor() which our code resided within.

```
template <typename forward_iterator>
    void to_tensor (
        forward_iterator ibegin,
        forward_iterator iend,
        resizable_tensor& data
    ) const
    {
        DLIB_CASSERT(std::distance(ibegin,iend) > 0);
        const auto nr = ibegin->nr();
        const auto nc = ibegin->nc();
        // make sure all the input matrices have the same dimensions
        for (auto i = ibegin; i != iend; ++i)
        {
            DLIB_CASSERT(i->nr()==nr && i->nc()==nc,
                "\t input::to_tensor()"
                << "\n\t All matrices given to to_tensor() must have the same dimensions."
                << "\n\t nr: " << nr
                << "\n\t nc: " << nc
                << "\n\t i->nr(): " << i->nr()
                << "\n\t i->nc(): " << i->nc()
            );
        }

        // initialize data to the right size to contain the stuff in the iterator range.
        data.set_size(std::distance(ibegin,iend), pixel_traits<T>::num, nr, nc);
        typedef typename pixel_traits<T>::basic_pixel_type bptype;

        const size_t offset = nr*nc;
        auto ptr = data.host();
        for (auto i = ibegin; i != iend; ++i)
        {
            for (long r = 0; r < nr; ++r)
            {
                for (long c = 0; c < nc; ++c)
                {
                    auto temp = pixel_to_vector<float>((*i)(r,c));
                    auto p = ptr++;
                    for (long j = 0; j < temp.size(); ++j)
                    {
                        if (is_same_type<bptype,unsigned char>::value)
                            *p = temp(j)/256.0;
                        else
                            *p = temp(j);
                        p += offset;
                    }
                }
            }
            ptr += offset*(data.k()-1);
        }
    }
```

Appendix B. The actual to_tensor() function from the matrix class that we used for much of our profiling.