Steven Fu & Christopher Hung (skfu, christoh)
15-418 Final Project, Spring 2017
Helium

SUMMARY

We implemented a math library for vectors and matrices using ARM's Neon SIMD intrinsics and compared the performance to different implementations run on a Raspberry Pi 3. We implemented matrix transposition and multiplication in particular for arbitrary size matrices and used them to more efficiently perform image feature extraction (convolution of an image with multiple filters).

BACKGROUND

We implemented several vector and matrix operations in SIMD using ARM Neon intrinsics. Our vector operations would take in 2 one-dimensional vectors of either ints or floats and compute their sum, difference, or product. The workload here consisted of first loading the entries of the input vectors into a Neon vector and then applying the operation on the two inputs. These operations are inherently data-parallel, which made it an obvious application of the SIMD intrinsics we were using. However, as we discovered, the program was primarily bounded by the cost of loading memory into the Neon vectors, which prevented us from reaching ideal speedups.

Our matrix operations included 2x2 and 4x4 matrix transpose, multiplication, and determinant as well as generalized MxN matrix transpose and multiplication, which took as inputs two-dimensional matrices expressed as a one-dimensional array in row-major order. The matrix transpose operations involved no arithmetic operations and were therefore unable to benefit from SIMD execution. However, calculating the determinant and product involved taking many products, which we could run in parallel.

APPROACH

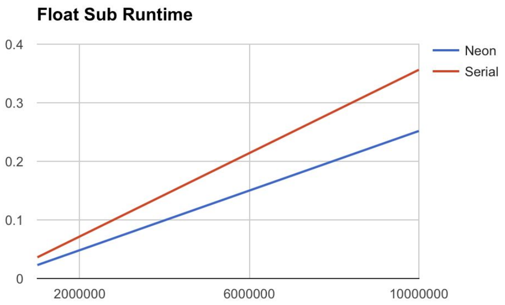
We targeted the Raspberry Pi 3's ARM Cortex A53 processor, which supports ARMv7 and thus the Neon SIMD instruction set. The SIMD vectors are of width 128 bits, and therefore can load and store up to four 32 bit ints or floats at a time. We first wrote 1 x n vector math operations to perform add, subtract, and multiply by loading and performing the operations on four elements at once. We then implemented 2 x 2 and 4 x 4 matrix transposition, determinant, and multiplication because Neon's support for access of stride length 2 and 4 lended itself to these applications. Following that, we began work on matrix transposition and multiplication of arbitrary sizes. For transposition, we tried various techniques such as loop reordering and blocking to optimize access patterns, as well as a cache-oblivious algorithm. This algorithm recursively divided the matrix in half along its largest dimension until a certain cutoff, at which point it would perform the transposition; this was found to be the fastest method of matrix transposition. We then implemented multiplication, similarly starting with a blocking implementation of constant block sizes, a cache-oblivious blocking algorithm similar to transposition, and finally transposing and computing dot products. For the best performance, we ended up using a combination of a cache-oblivious transpose, followed by recursively dividing the matrix into sub-matrices that we would then perform a vectorized dot product on to compute the matrix multiplication result. For the cache-oblivious algorithms, we started with pseudocode from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.5426&rep=rep1&type=pdf.

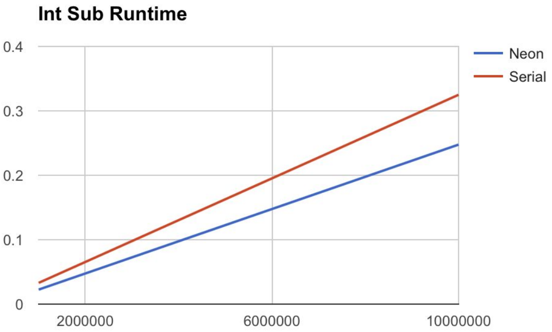
We experimented with varying block sizes and base cases for the recursion to discover what would be most performant in terms of CPU time used. In the case of matrix multiplication, we tried different cutoffs for the total number of elements in the resulting sub-matrices to be multiplied, including 32000, 64000, 128000, and 256000, and analyzed these across different input matrix sizes, ranging from around 500 to 2100. We found that there are interesting results, especially around power-of-2 input matrix sizes where the CPU time used spiked; we suspect that this is due to access patterns that result in thrashing in the cache (described later in the results section). The cutoff of 128000, which provided the best performance, was also the one that we guessed would work best because the L2 cache is of size 512 MiB, which is 128000 32 bit floats.

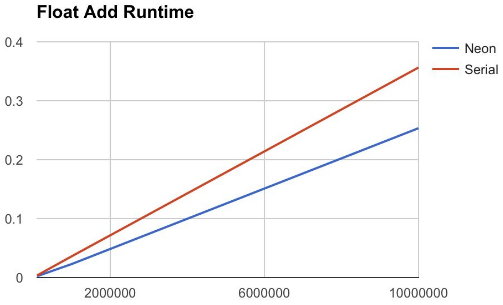
For image feature extraction, which requires the convolution of a single base image with multiple filters, we tried different methods such as implementing convolution and optimizing it, as well as transforming the convolution into a matrix multiplication. For that step, we used the technique described in the “Efficiently Evaluating Deep Networks” lecture, since that would allow us to use our implemented matrix multiplication algorithm. This also made sense, because although serially and naively convolving the image with the filter was faster for 1 filter, as the number of filters was increased to be greater than 1, the transformation became faster relative to the serial version. This is because the precomputation overhead is greater than the speed for serial convolution, but in feature extraction, multiple filters are convolved with a single image, and this resulted in better performance.

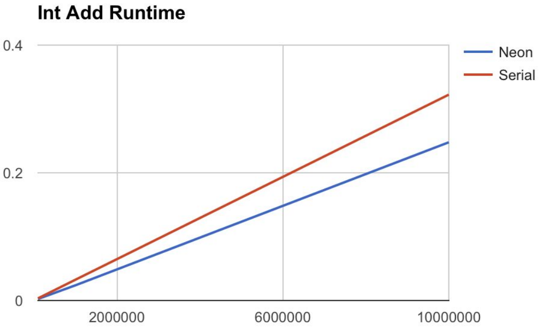
RESULTS

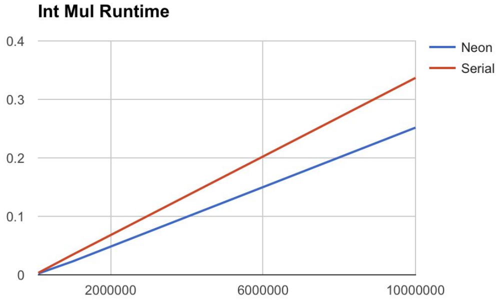
We measured our performance with CPU time used, and measured that across multiple algorithms and input sizes. We compared our performance to serial versions that we implemented and optimized, a past 15418 project (FastAndroid), and openBLAS for matrix multiplication.

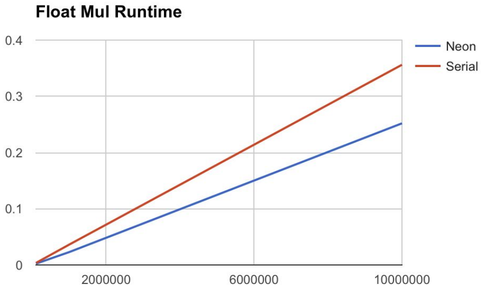

Float Sub Runtime

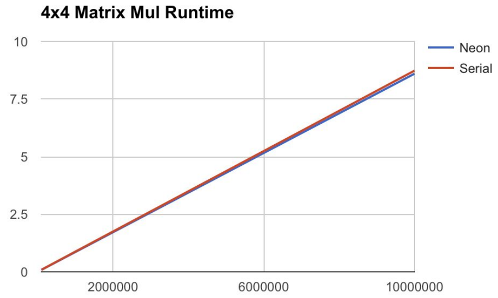

Int Sub Runtime


Float Add Runtime


Int Add Runtime

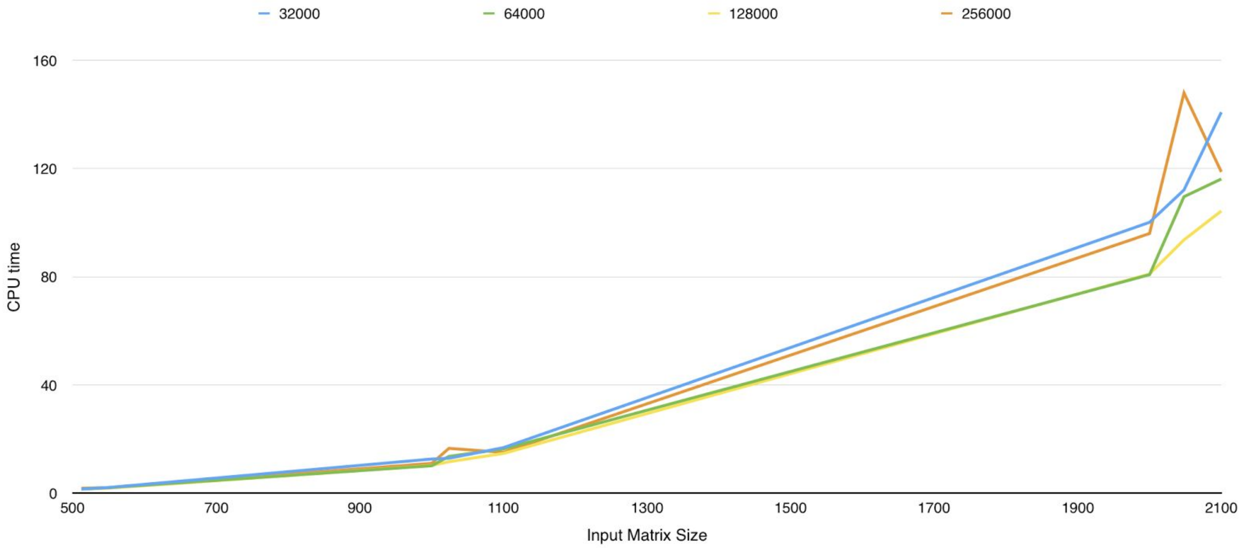

Int Mul Runtime


Float Mul Runtime



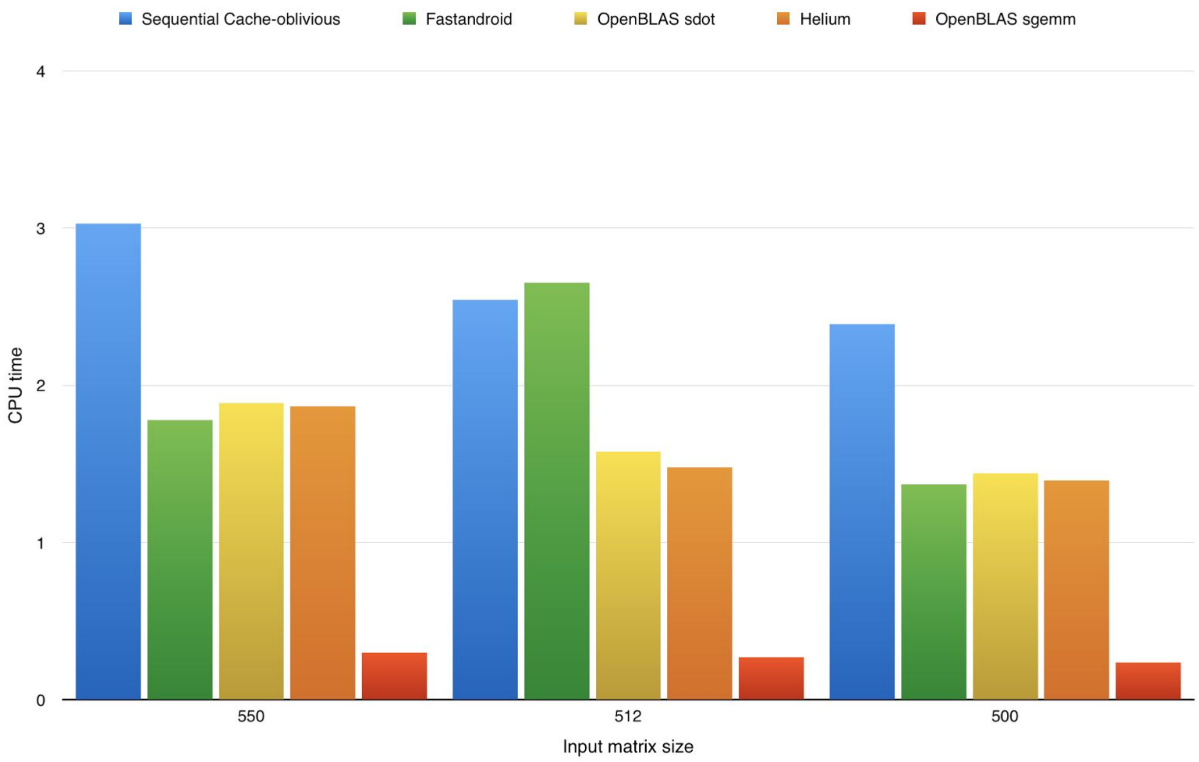
These graphs show the CPU time used by the vectorized and serial algorithms for a variety of input vector sizes.

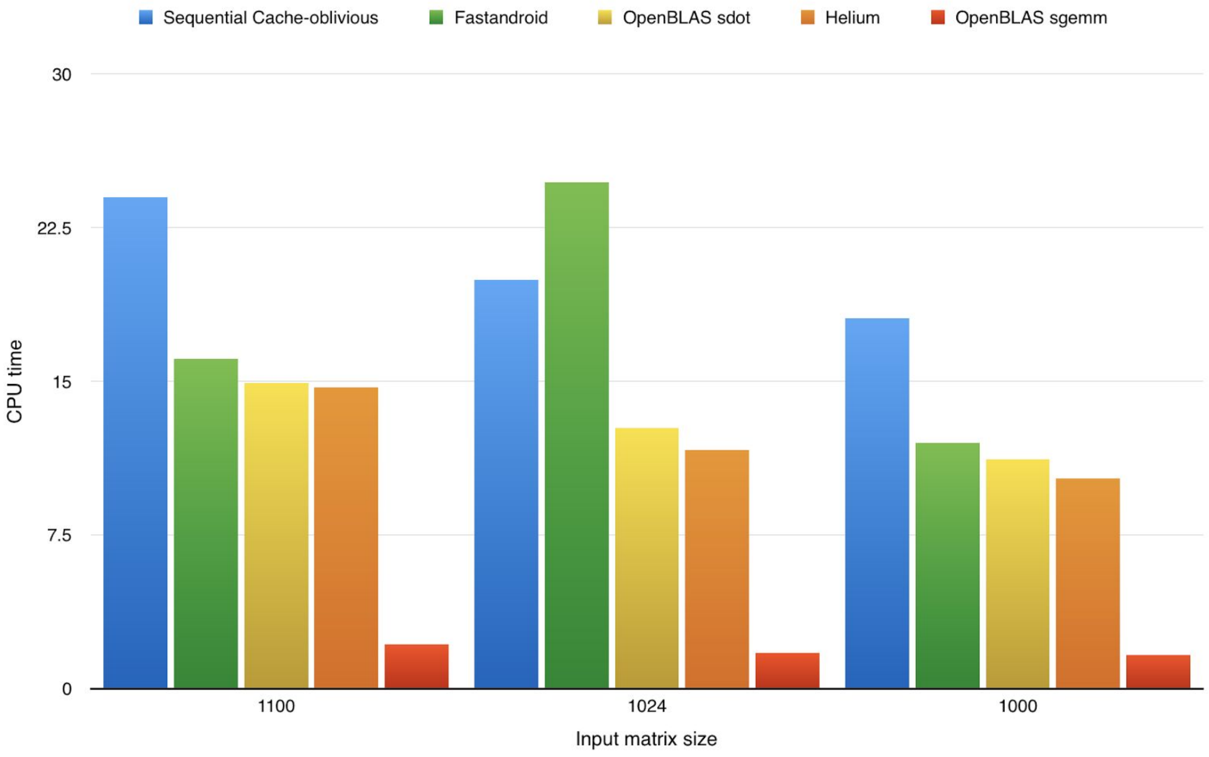
Our vector and matrix operations library was able to get speedups over our sequential baselines and had a runtime competitive with other, similar techniques. However, we were unable to achieve the ideal 4x speedup (given we were using a SIMD width of 4 elements) on even the most parallelizable of our applications, the operations on vectors. This is because operations on vectors already have low arithmetic intensity (1 arithmetic operation for every two pieces of data loaded), which is made worse when the instruction is applied to multiple sets of data (1 arithmetic operation for every eight pieces of data loaded for 4-wide SIMD). Applications like matrix multiplication had higher arithmetic intensity, especially for taking the dot product, which required 3 operations for every 4 elements loaded for the sequential version, but also included some inherently sequential portions, like the memory copies needed for transposing the matrix and the recursive splitting of the input matrices.

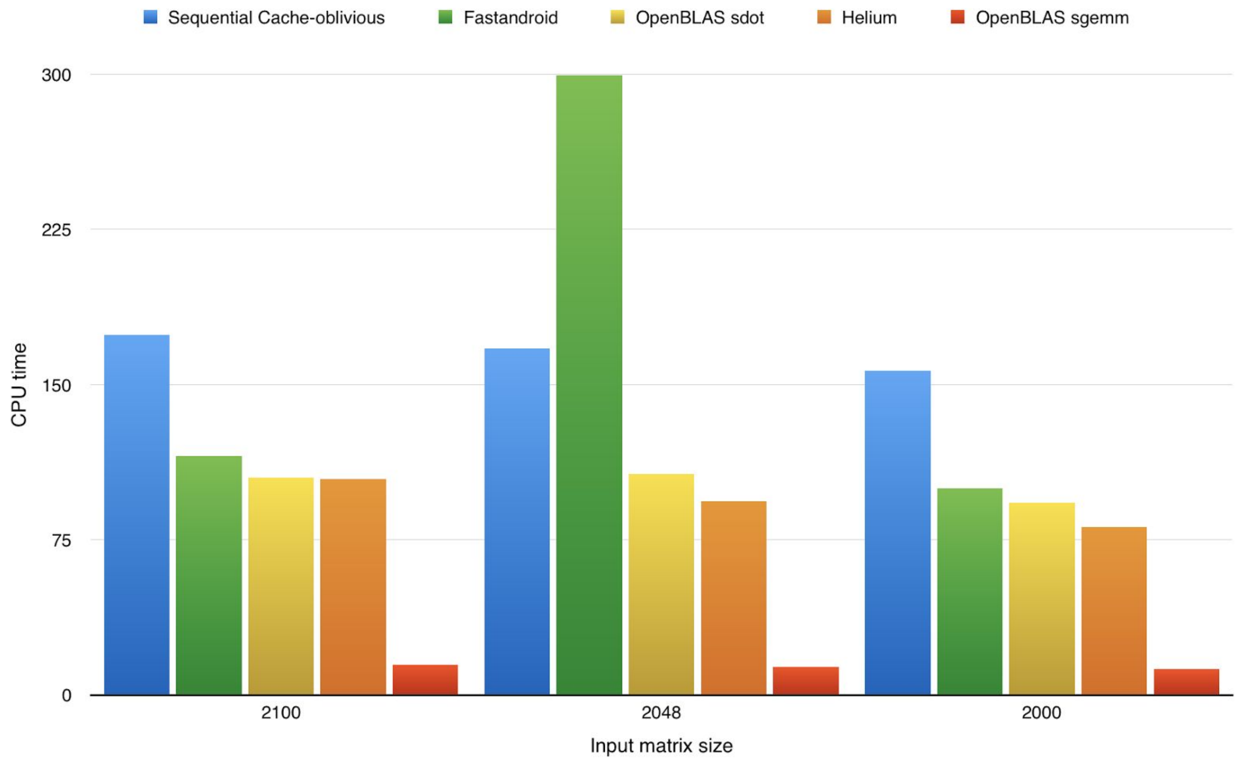


This graph shows the CPU time for matrix multiplication of various matrix sizes with different base case cutoff sizes (32000, 64000, 128000, 256000).

Another interesting factor in our runtime that came up when we were optimizing was the memory access pattern for matrix multiplication. Specifically, when we we raised the cutoff for the base case of our recursive splitting algorithm, we started seeing spikes in our runtime on matrices with power of two dimensions, which we believe to be due to the larger cutoff size causing accesses in the sub-array to thrash the cache and therefore causing more cache misses.


Sequential Cache-oblivious
Fastandroid
OpenBLAS sdot
Helium
OpenBLAS sgemm
4
3
2
1
0
CPU time
550
512
500
Input matrix size


Sequential Cache-oblivious
Fastandroid
OpenBLAS sdot
Helium
OpenBLAS sgemm
30
22.5
15
7.5
0
CPU time
1100
1024
1000
Input matrix size



These graphs show the CPU time usage of the various algorithms we tested against across multiple square matrix sizes.

We analyzed our performance for matrix multiplication particular, and compared it to a serial cache-oblivious algorithm, FastAndroid's SIMD vectorized version, an implementation using openBLAS's transpose and dot product, and openBLAS's general matrix multiplication. We found that for small matrix sizes (around 500x500), our CPU time was slightly longer than that of FastAndroid, but for larger sizes we were consistently faster. For all matrix sizes, our implementation was faster than the corresponding algorithm using openBLAS's transpose and dot product. However, when compared to openBLAS's single precision general matrix multiplication (sgemm) algorithm, we were within a factor of 5 - 7 in terms of CPU time. We discovered that this has to do with both the algorithm as well as the method of implementation. openBLAS's sgemm uses a blocking technique of varying block sizes such as 8 x 4, 4 x 4, 2 x 4, 2 x 2, and 1 x 1; furthermore, they coded in assembly so that loads and stores happened explicitly and used only the floating point registers instead of memory, so for the transposition of matrices, we wrote to memory in an intermediate step whereas they solely used the dedicated floating point registers. Our implementation was written in C and compiled using gcc.

To benchmark our convolution, we compared it to a sequential naive version as well as the same matrix multiplication method with our sequential cache-oblivious matrix multiply instead of a vectorized multiply. Although the sequential naive version was faster when only applying one filter, we found that the matrix multiply version spent a majority of the time converting the image into a matrix that could be multiplied by the filter. When we decoupled that step and ran the precomputed matrix through multiple filters, we were able to consistently beat the naive version, even with only two filters. This use case was ideal for our application, which was to apply different filters onto the same image to extract features that could later be used in image processing.

REFERENCES

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.5426&rep=rep1&type=pdf
http://15418.courses.cs.cmu.edu/spring2017/lecture/dnn/slide_032
http://supertech.csail.mit.edu/papers/Prokop99.pdf

LIST OF WORK BY EACH STUDENT

Equal work was performed by both project members.