Ryan A. Margraf

WES 237B

Assignment 2

8/14/2022

Part 2: Lab DCT

How To Run

1. Open a terminal
2. Navigate to the "Assignment_2/dct" folder
3. Type "make" and press enter
4. The program can be run using a command of the format

```
./dct <mode>
```

Mode 1 uses OpenCV, mode 2 uses the naïve implementation, and mode 3 uses the optimized 1D implementation. So to use the naïve implementation, one would run

```
./dct 2
```

To use timing mode, uncomment the "#define TIMING_MODE" line in main.cxx. This removes writing the image outputs, and is used for timing purposes only.

Images

Figure 1 - Original Image

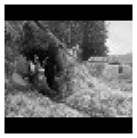


Figure 2 - Resized grayscale image

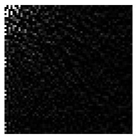


Figure 3 - OpenCV DCT result

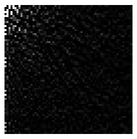


Figure 4 - Naive DCT result

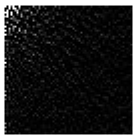


Figure 5 - Optimized (Separable) DCT result

Timing Analysis

	OpenCV	Naïve	Optimized
Real Time (s)	0.717	6.382	0.877
User Time (s)	0.595	6.288	0.763

It is unsurprising OpenCV was the fastest, since it is a widely used library that has had many years to develop. The optimized separable algorithm was a fairly close second, with the naïve implementation being significantly slower than both of the others. Again, this is unsurprising since it is performing many more operations, having an additional nested loop.

Part 3: Matrix Multiplication

How To Run

1. Open a terminal
2. Navigate to the "Assignment_2/matMult" folder
3. Insert the kernel mod to enable access to the PMU counter. To do this, type the following

```
sudo insmod CPUcntr.ko
```

4. Type "make" and press enter
5. The program can be run using a command of the format

```
./matMult <size> <numTrials> <printFlag>
```

   The size parameter indicates the size of the input matrices to test. The numTrials parameter indicates the number of trials to average the cycle counts over. Setting the printFlag to 1 prints the result matrices and whether they match. Setting it to 0 only prints the cycle counts.

   For example, to test 32x32 matrices over 100 trials without printing the result matrices, one would run

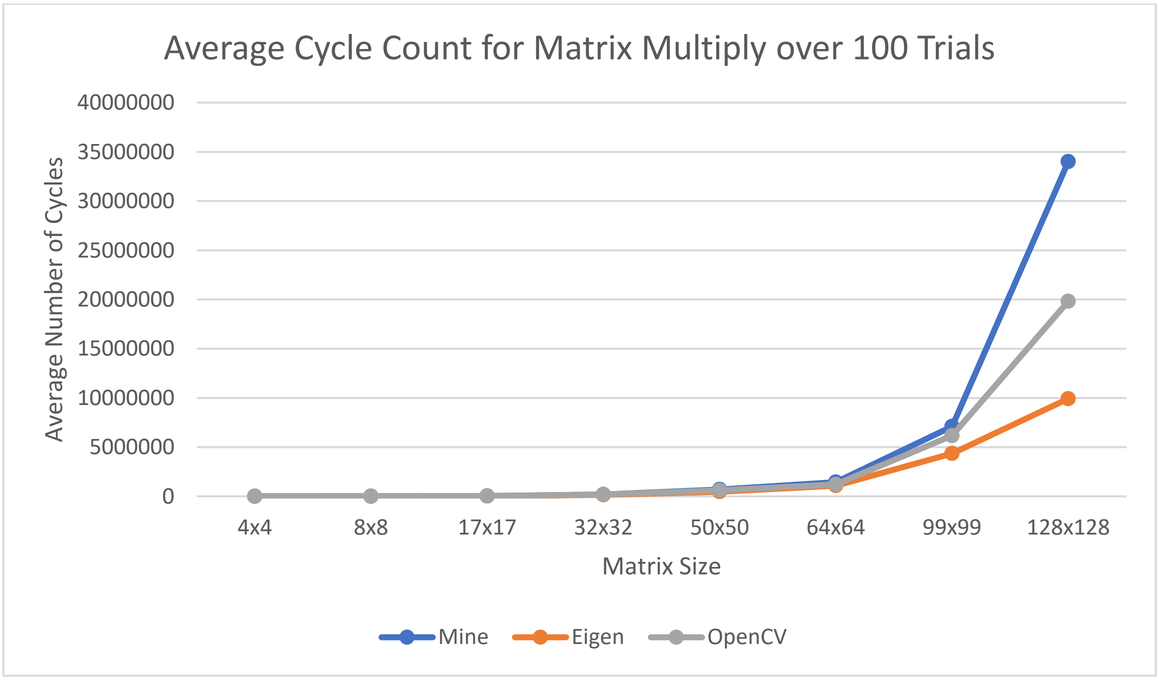
```
./matMult 32 100 0
```

Performance Analysis

The below table and chart show the average cycle count of a matrix multiply operation over 100 trials. I chose some arbitrary values that were not powers of 2 to cover a wider range of scenarios and try to throw off some optimizations the libraries may make for those sizes.

Average Cycle Count for Matrix Multiply over 100 Trials

Size	4x4	8x8	17x17	32x32	50x50	64x64	99x99	128x128
Mine	634	3253	27589	185040	693595	1443821	7099838	34015250
Eigen	1666	5639	29355	149745	466410	1085213	4357867	9925964
OpenCV	11083	12962	37324	194829	605300	1211662	6171709	19822905



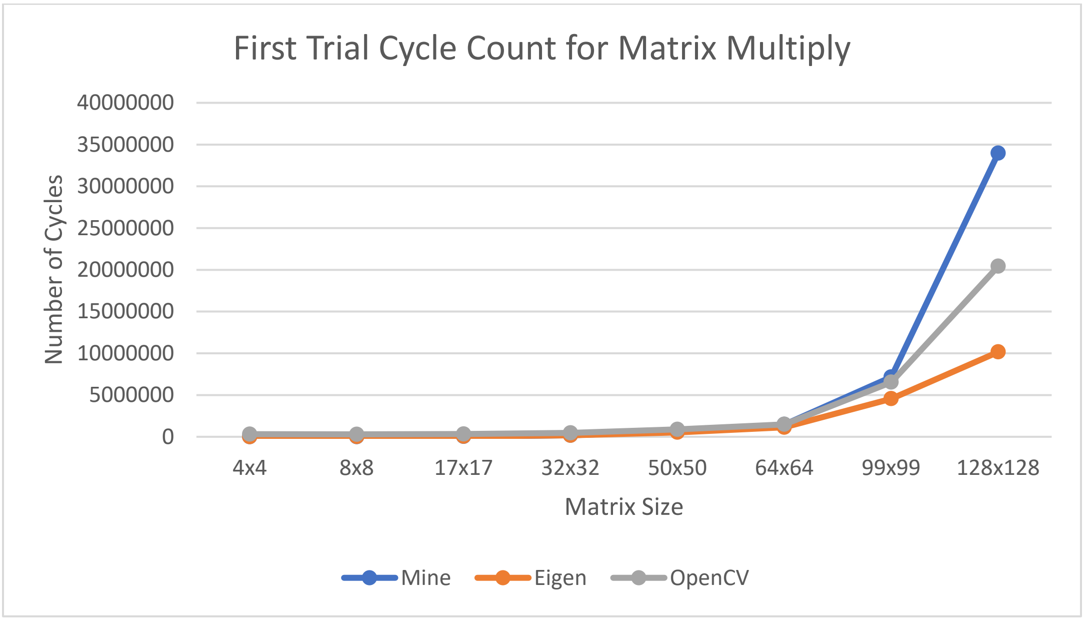
At first, I was a bit surprised my implementation was faster for the smaller sizes, but it makes sense. There is likely some overhead caused by the libraries' data structures, whereas I am using arrays of primitives.

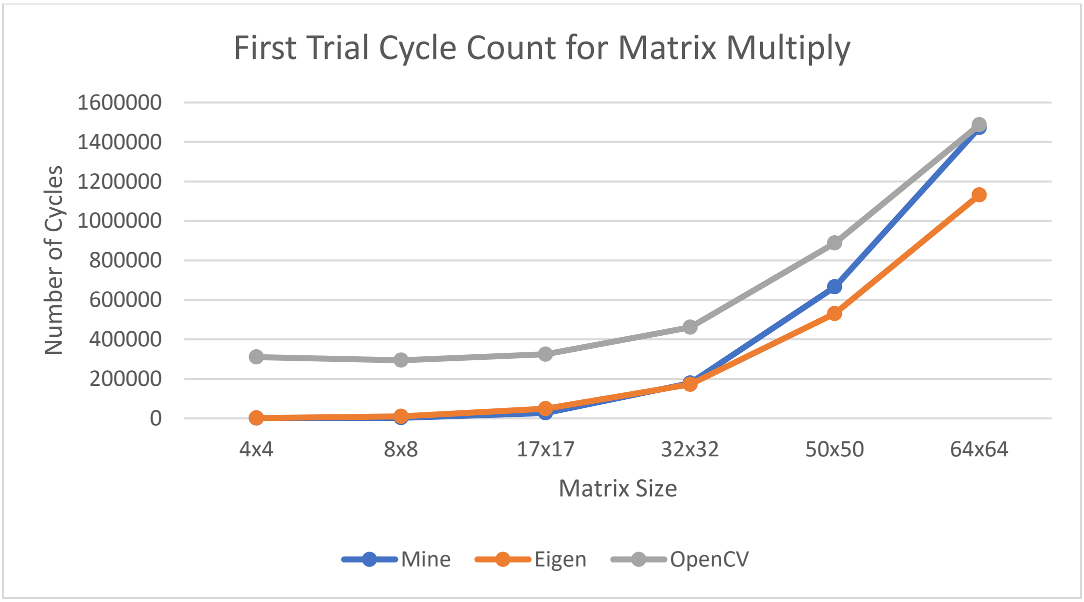
For larger sizes, the libraries pull ahead. I would guess that they use a faster algorithm rather than my naïve O(n^3) implementation, so this is unsurprising. Eigen was by far the fastest for the mid and large sized matrices, which makes sense due to it being a dedicated linear algebra library.

I noticed OpenCV took significantly longer to perform its first trial than its other trials, so I made another table to compare the first trial cycle counts. Indeed, OpenCV is above its average on the first trial for every matrix size, and significantly so in smaller sized matrices.

First Trial Cycle Count for Matrix Multiply

Size	4x4	8x8	17x17	32x32	50x50	64x64	99x99	128x128
Mine	816	3401	27395	178597	665718	1472543	7154051	33969479
Eigen	1693	10182	48960	172297	530816	1131600	4554379	10154460
OpenCV	310459	294321	324653	461238	887972	1486872	6528148	20421549





I thought a bit about why this could be. It can't be the cache because my code takes a new random matrix every time. The only thing I can think of is some initialization the first time an operation/multiply is performed when running OpenCV. This is somewhat supported by the above graph, which shows a relatively constant difference between the Eigen and OpenCV libraries at least up to 64x64.

Part 4: 2D DCT Optimization

How To Run

1. Open a terminal
2. Navigate to the "Assignment_2/2d_dct" folder
3. If not done already, insert the kernel mod to enable access to the PMU counter. To do this, type the following

   ```
   sudo insmod CPUcntr.ko
   ```

4. Type "make" and press enter
5. The program can be run using a command of the format

   ```
   ./hw2 <mode> (<width>)(<height>)
   ```

   Mode 0 uses the naïve algorithm, mode 1 uses the optimized separable algorithm from lab, mode 2 uses the matrix multiply implementation, and mode 3 uses the block matrix multiply implementation. The width and height parameters are optional, and specify the size of the image to use. If only width is specified, the height will be made to be the same. A square image is required for the matrix multiplication options.
   For example, To use the matrix multiply implementation with an image size of 384, type

   ```
   ./hw2 2 384
   ```

   The lookup table is enabled by default. To disable comment out the #define USE_LUT line in student_dct.cxx, and run "make" again.

Performance Analysis

I noticed the provided Makefile used -O0 to compile, so I decided to experiment with -O1 as well to see what differences it would make. The below tables provide the results.

Average Cycle Count over 5 trials		
	64x64 -O0	64x64 -01
Naïve	3838621888	3523518688
Separable	150156252	106453512
Naïve w/ LUT	932493941	137938779
Separable w/ LUT	21532001	2875215
Matrix Multiply	21574294	2925266
Block Matrix Multiply	34943557	4657411

Average Cycle Count over 5 trials				
	128x128 -O0	128x128 -01	384x384 -O0	384x384 -O1
Separable	1260635273	834340233	*	*
Separable w/ LUT	204867140	46218711	1335014421	1502008266
Matrix Multiply	200527133	72146048	1690168586	1932998013
Block Matrix Multiply	279859505	38795771	3284670460	1075474571

For the entries marked with *, I manually stopped the program since it was taking too long, and the counter would overflow anyway.

Unsurprisingly, the Lookup table significantly reduced the number of cycles required for the naïve and separable algorithms since several calculations, such as the cosines, are no longer required every loop iteration. The separable algorithm with the LUT performed similarly to the matrix multiply algorithm in most scenarios. This makes sense since they are both O(n^3) algorithms.

While the block matrix multiplication was slower with -O0 in all cases, it did show an advantage over the regular matrix multiplication when compiled with -O1 and larger input sizes were used.

In most cases, using -O1 instead of -O0 resulted in a significant improvement, up to nearly 10x in some cases. One exception was for the 384x384 input size. I have a suspicion that the counts here are overflowing, since it seemed to take longer than what the timer showed to run them.