Sam Weismuller
WES237B
8/7/2022

Assignment 2 Report

Part 2:

Original Image:

Resized, greyscale image:

OpenCV DCT:

Sam Weismuller
WES237B
8/7/2022

Naïve DCT:

1D DCT:

Time command results:

OpenCV DCT:

real 0m0.899s

user 0m0.551s

sys 0m0.260s

Naïve DCT:

real 0m6.855s

user 0m6.610s

sys 0m0.178s

1D DCT:

real 0m1.032s

user 0m0.787s

sys 0m0.186s

Sam Weismuller
WES237B
8/7/2022

Part 3:

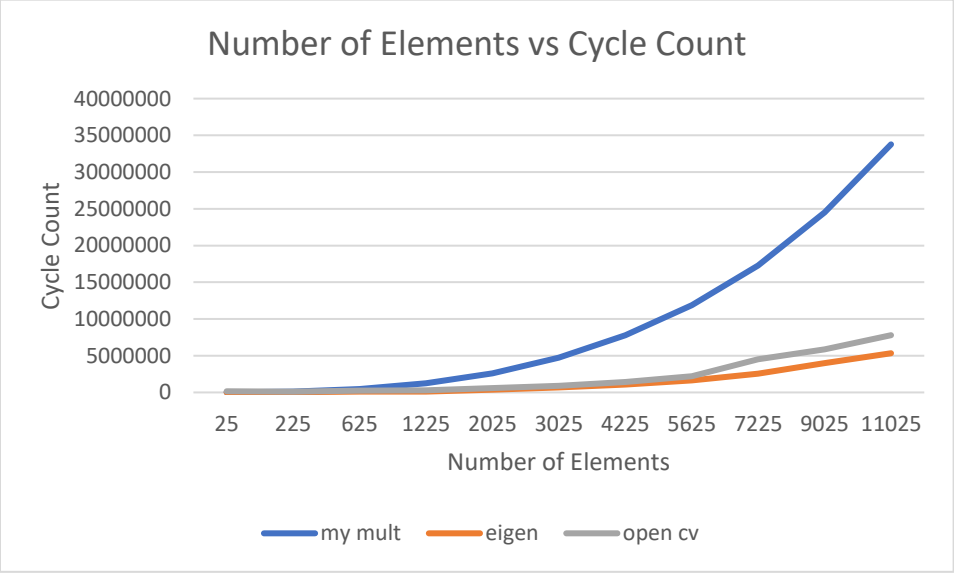
In order to obtain accurate cycle counts using the PMU register, I did the following:

1. Isolated CPU1 with boot args:
   ```
   bootargs = 'console=ttyPS0,115200 root=/dev/mmcblk0p2 rw
   earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1
   uio_pdrv_genirq.of_id="generic-uio" clk_ignore_unused isolcpus=1
   && bootz 0x03000000 - 0x02A00000'
   ```
2. Then set the PMU cycle count register using the kernel_module project.
   It's built with:
   ```
   make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
   ```
   and the module is inserted with:
   ```
   insmod CPUcntr.ko param_cpu_id=1
   ```
3. The matrix multiply project then uses cyclecount.h to read the cycle counts before and after the matrix multiply.
4. Then the matrix multiply project was run on the isolated CPU by prefixing the command with:
   ```
   taskset -c 1
   ```

Results:

Cycle Counts by Number of Matrix Elements:

	25	225	625	1225	2025	3025	4225	5625	7225	9025	11025
my mult	7025	106795	476815	1259370	2576423	4723119	7791584	11890682	17293442	24518742	33770849
eigen	4361	28003	108072	108072	391357	670157	1057182	1643057	2554374	4009090	5333619
open cv	140877	105598	238707	304161	580805	900428	1406048	2196825	4502399	5864897	7792255



Eigen had the best performance overall. OpenCV had a lager initial overhead, so it actually lost to the naïve implementation at low element counts. However, as the number of elements increased it quickly outpaced the naïve solution and got closer to the eigen performance.

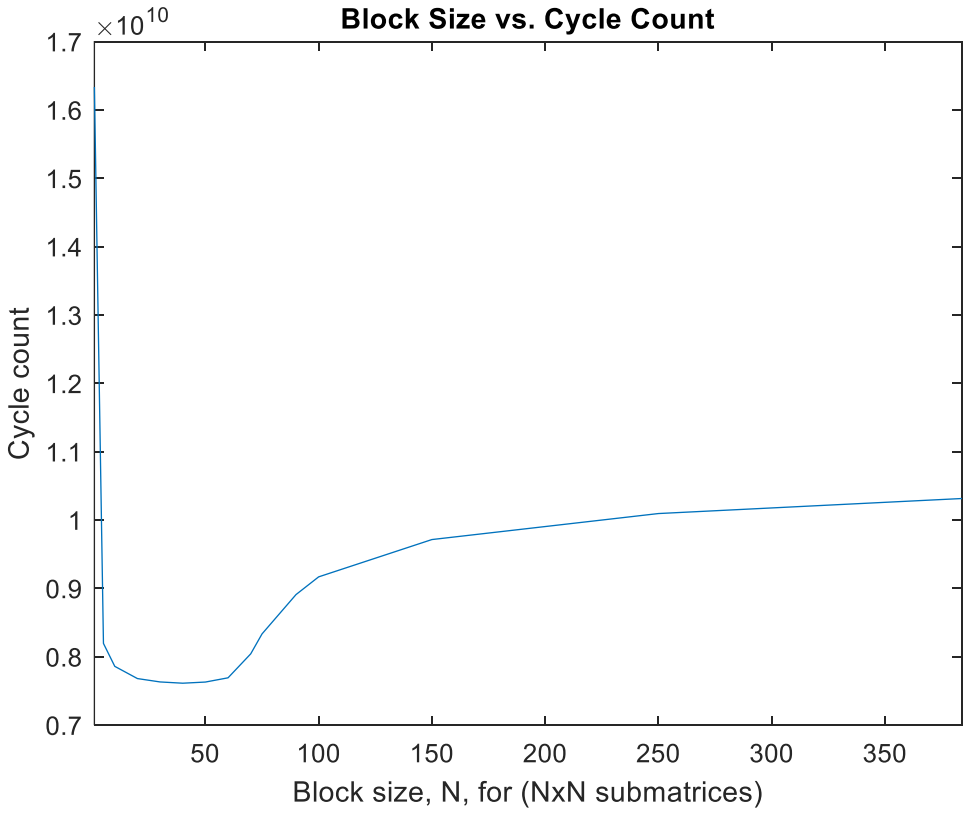
Sam Weismuller
WES237B
8/7/2022

Part 4:

Unfortunately, `perf` did not work on the linux version on my PYNQ board, so I had to obtain metrics just using the PMU register cycle counts.

For using a matrix of size 384x384, the optimal blocks size, N, for the NxN submatrices, was around 40, though the performance was extremely similar for block sizes 20-50.

	1	5	10	20	30	40	50	60	70	75	90	100	150	250	384
cycles	1.63E+10	8.2E+09	7.86E+09	7.68E+09	7.63E+09	7.61E+09	7.63E+09	7.69E+09	8.04E+09	8.34E+09	8.91E+09	9.17E+09	9.72E+09	1.01E+10	1.03E+10

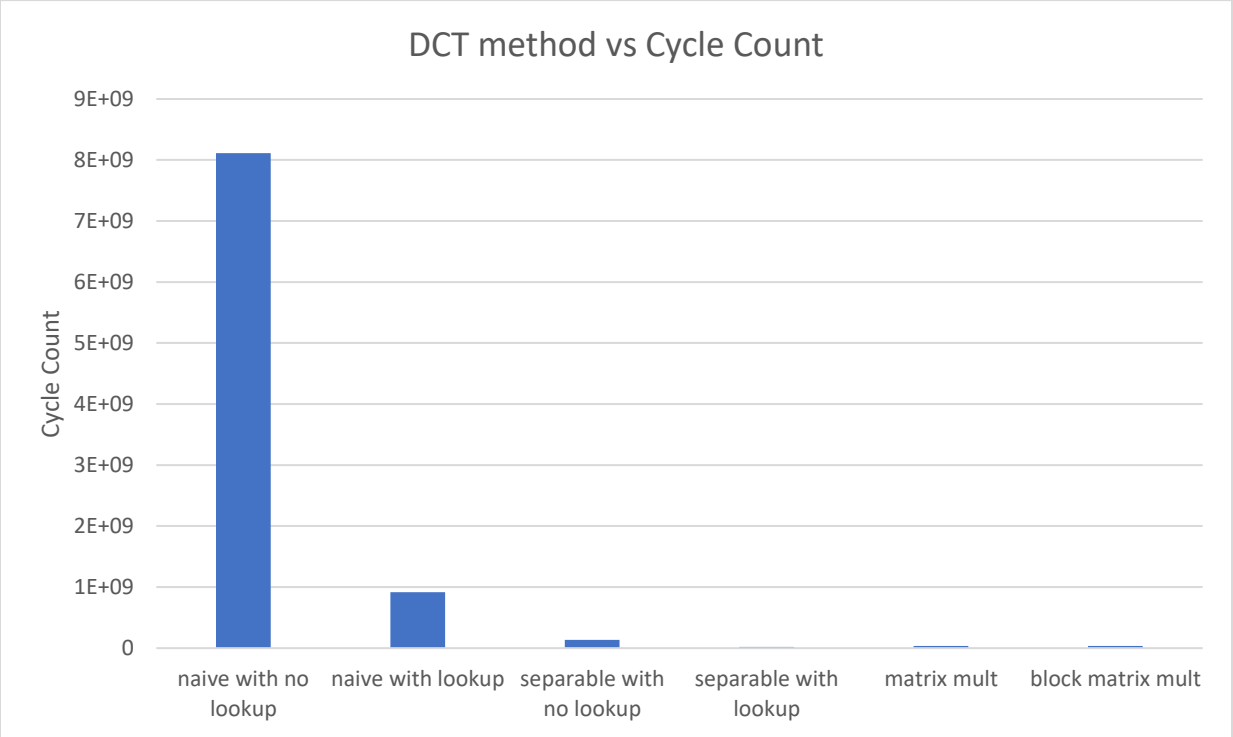


You can see that after a block size of 60 you quickly lose any benefits from the improved locality of the block matrix multiply, indicating we are hitting the limits of our cache size to hold our submatrices.

Sam Weismuller
WES237B
8/7/2022

Cycle counts of various DCT methods on a 64x64 matrix:

	cycles	total time (s)
naive with no lookup	8113971712	14.157
naive with lookup	915822912	3.087
separable with no lookup	134635008	1.879
separable with lookup	21987456	1.704
matrix mult	36970304	1.725
block matrix mult	35355136	1.724



Here it is clear that the naïve implementation is several orders of magnitude slower than the other algorithms.

Sam Weismuller
WES237B
8/7/2022

Experimenting with different matrix sizes, we can see that 1-D separable with the lookup table is the most efficient for matrix is N (for NxN matrices) from 64-624. The block matrix multiplication is the 2nd most efficient with the normal matrix multiplication coming in third.

	64	128	256	384	524	624
naive with no lookup	8113971712					
naive with lookup	915822912	15201712192	2.44932E+11			
separable with no lookup	134635008	1136925632	9235405376	31270418368	75750247616	
separable with lookup	21987456	207353920	1745368896	5974473216	13470756672	24897823936
matrix mult	36970304	362572288	3078944768	10572457856	23354819840	43865204032
block matrix mult	35355136	281342016	2513507072	7617468032	19295694144	32584977088