Performance Prediction for GPU-based HPC Codes

Identify the Edges in a Given Image As Fast As Possible and model their speedup

Robert Hughes

An Edge detector is the program that is being parlellized. The data will be examined to see if the parallelization techniques proved to work. The parallelization is done using the NVIDIA cuda library and taking advantage of GPU archtiecture. In this particular example we will explore the use of a GPU to expedite the process of detection. We will also look at how we can model and make predictions for GPU speedups and run times

Image Processing, canny edge detection, CUDA, GPU, Performance Prediction

I. Introduction

Every image is is composed up of many different regions. It is between thee boundaries that we can find the edges of an image. The edges of an image are extremely important and they can help to give us a lot of valuable data that we can take and use for further analysis. The detection will isolate the pixels of interest and mark them as edges. This is an important image processing technique because it allows to simplify the image while preserving a lot of the structural elements. As the field of computing is ever evolving it is important that computers continue to increase in speed. And one way that many companies are making faster computer is by adding a Graphical Processing Unit (GPU). GPUs are really good at image manipulation and so we expect to see a great speed up compared to the serial run. The reason for this is due to their anatomy. GPUs have a parallel structure that allows for large data blocks to be quickly computed. They also have levels of caches to make them better and faster at memory access. In all of the top preforming super computers some sort of GPU can be found.

While the previous paper discussed the speedup of the kernels in the Canny Edge detection program this one will be more focused on the cornerness detection that was added and performance prediction. Performance is super important to be able to model and predict in High Performance Computing(HPC) because it allows users to have a guess as the how the program speed up and run time will be without running. In the case of this canny edge detector the time to run is ever more than few minutes (even in serial) so it is not as important but when HPC scientists have more complex and time intensive algorithms to run then the prediction can be key. If the prediction model only shows minimal improvements it may not be worth running. Whereas if a large speed up is seen in the prediction this can indicate an algorithm is a good candidate for the HPC speed ups.

II. Canny Edge Detection

A. Image

The image that we have chosen to manipulate is the Lenna image. It is a well known open source image and has lots of interesting features that we will extract using the Canny Edge detection. This will allow us to have a different, but similar, image that still retains lots of useful information. We are also using many different sizes of the Lenna image this is important because we can compare the speed up vs the size and see how it may behave, either linearly or not. All of this will help us to make better predictions of how different file sizes may behave and how fast they may run

B. Edge Detection

In order to start the Canny Edge the image must first be read into the program. Once this is completed we will apply series of Gaussian filters which will help to eliminate the noise and give us some temporary files that we will then use to gain the magnitude and gradient images. These images are useful because the magnitude is used as a starting place for the edges and the gradient can be used as a way to determine in which way a the pixels are changing in intensity. After these two temporary files are found we need to use Non-maximum Suppression. Non-maximal suppression is an edge linking technique that we will use to use the magnitude image to determine strong edges and to suppress 'weaker' edges in the non-intensity directions. Once this is is done we take the image and preform hysteresis which is another edge linking technique. In hysteresis we are trying to take out all the weak edges and only keep the strong edges. This is a function where we can set some parameters of how low we want the weak edges to be. So by changing the this parameter we can either have more or less edges. Finally the last test that we have to preform and is basically the second half of the hysteresis where we are either turning those strong or weak edges on/off depending on their neighbors. Once all of these steps are complete we could have images with all of the edges taken out of it. The next step that is new to this process is finding all of the corners or features in an image. This is important because these features can give us insight into an image or more importantly image sequences and videos. They can allow us to track certain features and see how they are changing over time. This may not be as important or this particular problem but feature detection and cornerness is very important in image processing.

III. CUDA in C

CUDA, Compute Unified Device Architecture, will be used as the preferred technique for the parallelization because of it ease of use with C, which has already been studied in class, and all the documentation that is offered. CUDA was first developed by Nvidia for developers to take advantage of their GPU architecture. There have since been other variations of CUDA that can be used to be run GPUs. One advantage of CUDA is that it offers many different libraries that we can take advantage of and make the overall coding of the project easier. Another advantage are tools that can be used like cuda-memcheck which can hep us to fix GPU segmentation faults. The other big advantage of CUDA is that it will allow us to take advantage of shared memory programming which we will see plays a role in the time to run out convolutions. In both of the new added CUDA kernels we will be taking advantage of the shared memory to make the kernels faster. CUDA was not supposed to allow us to declare multiple __extern__ memory locations but we found out in class that if we do this the compiler can handle it and figure out how to combine it to make the proper memory allocation. One important component of the utilization of shared memory architecture is that we need to sync up the threads when we are writing or creating parts of the memory. However one issue that I saw was the final edges image had almost of the blocks in the image outlined. This was due to incorrectly place sync thread functions. There was a sync thread in the if statement which, because not every thread would reach, was causing problems. It was found that when these incorrectly placed syncthreads were removed the program would yield the correct edges image. What is interesting about this is that there were no segmentation faults or other errors despite this and the program would run without issue. I think that one debugging tool that could be useful for CUDA would be to look for these mistakes. Since the syncthreads requires all of the threads to bump into it for it to behave properly I think it would be interesting to see an error thrown if not all the threads are going to hit a syncthreads call.

A. *Corner Detection*

The main part of updating the older code was to take the image and get the corners out. To start doing this we allocated a shared memory block as a way to make a faster corner detector. We also utilized a window size of 7 we can split the image up into image blocks so that each threads finds the corners in their image.

B. *Get Corners and Top Corner Cases*

Once the corners are given by the corner detection kernel we want to filter out the bad cases and only preserve the best corners per block. This is done in a separate kernel where we pool all of the corners up sort them by their indices and reduce them to only print the best corners per block. It is during this time that the most important corners of the image are apparent that if we wanted to track movement we would use.

IV. Regression Based Performance Prediction

For our purposes we intend to use the regression based performance prediction, that was described in the reading. In order for us to make a good prediction we need to develop a good model of the data. We will use R as out statistical analysis tool. From R we can take the data and develop a system that we will use to make the predictions. By using R-squared we can make sure that our model is close to accurate and that it will be valid for use in our predictions. Ideally the R-squared value will be close to 1 and the p values close to 0.

A. *How can we make predictions?*

The way we will approach making our predictions is by having a multiple different situations run and timed and then using those numbers to make our prediction. The first step is we will time all of the different kernels in the CUDA program to see about how long they will take and then we will take the memcpy numbers that we got and add them all together to get our final prediction. One big thing that we will need to keep in mind is the number of FLOPs in the program. This means we need to have a rough estimate of how many times the program is going and talking to the memory as this plays a large role in the overall run time.

B. *R as a tool for statistical analysis*

R is a programming language and environment that we will be using to perform the statistical analysis of the program on. Most importantly we can read in the data that we collected through the trials and build models to make predictions with. R will be very useful because it has a command line along with some graphical interfaces that we can use to see how the data is relating to each other. One of the most important tools that we will be using in R is the lm() function. This will be the linear regression tool and is useful as we can use it to find a line of best fit to then make the predictions. We can also make these models for the kernels to determine what the run time will be for different image sizes.

V. Results

Since the goal of this project was more focused on understanding and being able to make performance predictions than it was to have a final image at the end that this what the results section will focus on. One thing that is important to note is that we were trying to get a cornerness image out of the program and we were successful. This cornerness image can be seen in figure 1. We were also tasked with finding the max corners of each of the blocks. This would require doing a reduction of the blocks to find the best cornerness case for the block and corners in question. Another task that we were given as to predict the execution time of 9 different set ups, unfortunately the image of size 7680 will not run because there is a memory issue where not enough space can be allocated. Seeing as there is no memory that I can free before this pint I do not see an easy fix for this problem so I will just make a prediction for these. However we can compare the other 6 to the actual to see how close we are able to predict the run time.

A. *Performance Modeling.*

The first thing that had to be done was to form a couple of different models that we could use to make the equation and solve for the execution time. The there are two parts that we need o do this in to make a good model that we can apply. First we need to look at the communication time and second we need to take the kernel time into consideration. First the communication cost was looked at, this was done by writing a simple benchmark to test an array of different sizes and then looking a the host to the device memory copy times. Since it asked for an array of floats we can use 245 floats to represent every 1kB. This would be the basic calculation that I used to determine how many floats needed to be put in the array. This is a similar case for the MB except it needed to be multiplied by 1000. once this simple test was made a bash script was written that allowed for multiple different KB values to be ran multiple times. We then took the average and calculated the bandwidth. Table 1 shows the average bandwidth for all of the different array sizes and the host to Device vs. the device to the host. Table 2 shows the average time in microseconds that it took for these different communications to occur. Table 4 takes the data and uses R to analyze it. Due to the shape of the graph I decided to run a logarithmic linear regression of the data this yielded good results which we can see from the R-squared value which we would ultimately like to be as as close to 1 as possible. Using these models we can make predictions of how the memory transfers may occur in the other images. In the second part we were tasked with making a model of the different kernels. By looking at table 5 we can see that there was not a good model that was determined for the data set. For some reason my data did not indicate any clear best fit so when I ran the linear regression on the R squared value did indicate that there was not a strong correlation between the block size and the kernel time although by further looking at this table we can see that there is a better R-squared value for the bigger the image size. This leads me to believe that the efficiency and utilization of the GPU plays a role in determining the R squared value. What this indicates to me is that when the block dimensions are not that of a power two it greatly impacts the over all running of the GPU and slows it own making the model poor. When comparing the same block dimension versus different size images we get a much better fit that we can apply to make estimates for the different sized images. By looking at table 5 we can see the different run times of the kernel and do an addition to make the prediction of the what the time will be. We can hen compare it to what we actually ran and see to what accuracy we were able to predict the run time to. In the first image size we can see that the we were had an error of about 10% this is not great but I think that it is within reason. I think one error that needs to be taken into account is that it is on the lower side. I think that if the prediction was on the higher side it would be better. This is because the program may occasionally run a little slower than the average, making the prediction a little closer to correct. It is also nice to be pleasantly surprised when your code finishes sooner than expected. When it came to modeling the 5170 sized image it was found to have an error closer to 35%-40% this is not good and is very bad. We would not want to use these as a model for our system as it would give inaccurate results. I think that the reason for this maybe that there is some kind of over head that is in the program that this does not account for. This maybe in memory copies that are not being accounted for. I think that another reason it may be off is due to the small execution time if the program took longer to run then maybe we could have higher accuracy because then the small changes in the microseconds will make less of a difference. The one thing that this assignment asked was for us to model was the image of size 7680 but we were unable to do so. The reason we were not able to model this was because it would cause a memory issue. The problem was that there was not enough room on the system to allocate enough memory for that much data. When running the CUDA-Memcheck I found the issue was purely on the issue of not enough device space not a memory/segmentation fault for another reason. When looking into the code for some memory that I could potentially free I was not able to find any that was not pertinent until the end. For this reason I did make a prediction of the run time of the image size with these grid dimensions but I did not get any actual run times. If we were to look into the code some more I think we would be able to find some memory that we could clear.

VI. Conclusion

One of the most important things that we need to understand as HPC scientists is how to make the models of predictions. It is important because it can allow us to make guesses at execution time. This is important for programs that will require more time to run than a few minutes or seconds and can be really useful to help decide if it is worth parallelizing programs/algorithms. For this reason it is pertinent to develop good modeling skills. One key component to this is utilizing R as a tool. By Using R we can develop even better and smarter statistical models. Overall I would have liked to see a better accuracy of my own results in the prediction. Looking back on the class as a whole though I think that using the Canny Edge was a sufficient and useful algorithm to learn about the different parallelism techniques and how to apply them.

VII. Acknowledgments

I would like to thank Dr. Pallipuram for all of the psuedo code that he provided and all of the help that he gave in office hours. I would also like to thank him for putting on this class I learned a lot and I think HPC is a growing and important area to learn about.

size	H2d – kilo	D2h – kio	H2d – Mega	D2h – Mega
1	0.090909090909091	0.058823529411765	0.001801477211313	0.000707013574661
2	0.158730158730159	0.123157449101335	0.001795977011494	0.000728066982162
4	0.240963855421687	0.148367952522255	0.002018163471241	0.000726991183414
8	0.392927308447937	0.197530864197531	0.002872428279056	0.000746073786698
16	0.502670436694942	0.218340611353712	0.003100198412698	0.000788959809253
32	0.594464053501765	0.252585050122346	0.002927454029823	0.000784794604537
64	0.770156438026474	0.287821550638604	0.004255885091103	0.000790661882898
128	0.846392911459366	0.317397341797262	0.004416991614617	0.000809386352355
256	1.08428631935621	0.43331076506432	0.004659543874338	0.000830435459594
512	1.6783583557333	0.664762399376785	0.004957493367416	0.000837937626018

Table 1: Modeling of the GPU Time Bandwidth

KILO	h2d	d2h	MEGA	h2d	d2h
1	11	17	1	555.1	1414.4
2	12.6	13.48	2	1113.6	2747
4	16.6	20.25	4	1982	5502.13
8	20.36	36.64	8	2785.1	10722.8
16	31.83	55.59	16	5160.96	20279.867
32	53.83	100.82	32	10931	38534
64	83.1	519.66	64	15038	76378
128	151.23	506.76	128	28979	163100
256	236.1	590.8	256	54941	355229
512	305.06	770.2	512	103278	632578

Table 2: Average Time of the Communication (microseconds)

Function	Min	1Q	Median	3Q	Max	R-squared	P-value	fit
Kilo H2D	-0.1683	-0.09053	-0.03675	0.03614	0.361	0.8991	2.96E-05	lm(kh2d ~log(size))
Kilo D2H	-0.085764	-0.039481	-0.007969	0.023722	0.15524	0.8508	0.0001445	lm(kd2h ~ log(size))
Mega H2D	-5.48E-04	-1.09E-04	1.37E-05	1.73E-04	3.90E-04	0.948	2.04E-06	lm(mh2d ~log(size))
Mega D2H	-1.11E-05	-5.78E-06	-2.07E-06	3.39E-06	2.13E-05	0.961	6.43E-07	lm(md2h ~ log(size))

Table 3: R analysis of the MemCpy Data Points Linear Regression

Image Size	total time	total time – file IO	File IO Read	File IO write	Host to Device	Horizontal Convolution	Vertical Convolution
256	321409.868131868	243752.483516484	349.56043956044	77307.8241758242	479.747252747253	368.758241758242	285.296703296703
256	344716.36	270049.08	351.06	74316.22	377.4	256.84	219.52
256	390158.42	370123.62	327.06	90371.8	287.8	141.96	92.3
256	326962.78	241375.44	317.72	85269.62	581.38	353.86	307.96
256	711156.46	611159.3	345.66	99651.5	738.44	492	460.32
256	468373.36	372760.58	345.66	95267.12	477.8	407.42	416.04
256	700415.36	609965.44	351.56	90098.36	556.9	353.14	328.6
512	290714.511111111	285200.311111111	1630.06666666667	3884.13333333333	576.588888888889	380.288888888889	394.211111111111
512	305116.08	299976.54	1165.22	3974.32	598.92	537.44	438.78
512	437857.2	432595.06	1163.58	4098.56	554.46	367.08	303.14
512	270936.56	261501.16	1146.62	8288.78	629.26	397.8	349.08
512	562502.58	557192.84	1213.08	4096.66	639.82	638.6	547.38
512	522774.3	517464.62	1163.32	4146.36	684.94	580.26	535.12
512	401563.4	396367.06	1183.36	4012.98	661.62	603.14	637.12
1024	315577.943820225	298234.303370787	4127.05617977528	13216.5842696629	1530.50561797753	934.516853932584	857.292134831461
1024	338418.7	320033.74	4314.16	14070.8	1476.22	900.8	877.04
1024	364370.44	345403.84	4350.74	14615.86	1798.18	1045.16	998.2
1024	336201.28	318036.22	3964.26	14200.8	1329.72	833.86	855.76
1024	467556.9	435072.4	4291.22	28193.28	1584.24	950.46	764.62
1024	370967.34	352923.4	4302.54	13741.4	1545.5	845.36	783.94
1024	382090.16	363889.84	4259.5	13940.82	1458.76	955.58	915.2
2048	429691.593023256	368564.406976744	14625.0581395349	46502.1279069767	5051.73255813954	2245.87209302326	2140.53488372093
2048	438592.170212766	376508.978723404	15286.6170212766	46796.5744680851	5154.57446808511	3020.6170212766	2076.48936170213
2048	646475.7	580864.08	14942.32	50669.3	4827.22	3105.4	2367.82
2048	488937.361702128	427252.382978723	14946.3617021277	46738.6170212766	4870.70212765957	2473.21276595745	2462.12765957447
2048	731106.22	670310.1	14659.26	46136.86	5161.2	2641.16	2294.28
2048	783417.36	721633.42	14637.68	47146.26	5229.56	3327.06	3203.36
2048	727459.346938776	666668.918367347	14238.306122449	46552.1224489796	5923.97959183673	3945.81632653061	3988.95918367347
4096	987177.325301205	749678	60473.4819277108	177025.843373494	18888	7389.24096385542	7249.32530120482
4096	919383.681818182	685651.590909091	62724.5909090909	171007.5	18416.1363636364	8770.79545454545	7661.13636363636
4096	1093132.47916667	852888.916666667	62541.6875	177701.875	20173.625	10291.7916666667	9417.3125
4096	934401.083333333	695880.791666667	64859.3541666667	173660.9375	18553.0416666667	8435.97916666667	8607.85416666667
4096	1301086.46808511	1056550.93617021	60689.5744680851	183845.957446809	19322.3404255319	10187.0425531915	9318.89361702128
4096	1245863.14893617	994749.127659575	62415.9361702128	188698.085106383	19126.3617021277	10117.5531914894	9870.51063829787
4096	1391527.25531915	1135606.5106383	63279.2553191489	192641.489361702	19486.5744680851	16476.4042553192	15665
10240	6241392.45	3712505.65	375079.15	2153807.65	118570.275	61997.25	64592.475
10240	5915542.2	3467249.3	364020.3	2084272.6	111421.5	59736.7	57143.4
10240	6924741.5	4512479.61538462	373342.923076923	2038918.96153846	119697.653846154	80842.3846153846	55355.0384615385
10240	5968208.44	3450328.24	369183.68	2148696.52	114602.92	64315.88	58793.16
10240	6846718.63333333	4400281	369282.7	2077154.93333333	120047.6	84647.5	57036.8666666667
10240	6946755.14814815	4520650.92592593	379349.740740741	2046754.48148148	122074.407407407	79838.8518518519	63240.0740740741
10240	7470925.25	5058697.70833333	376772.708333333	2035454.83333333	123318.708333333	115332.708333333	90538.4166666667

Image Size	Magnitude Kernel	Gradient Kernel	Thrust Sorting	Suppression Kernel	Hysteresis Kernel	Edge Linking Kernel	Device to Host	Block Size
256	232.659340659341	249.153846153846	458.967032967033	256.076923076923	275.10989010989	239.186813186813	1066.20879120879	8
256	193.96	193.24	429.48	202.44	190.86	189.84	1112.38	12
256	63.52	64.34	315.26	174.38	107.08	100.46	1057.28	16
256	192.96	384.06	473.44	256.08	198.9	187.36	1045.64	20
256	356.78	439.5	638.74	496.54	436.52	411.48	1110.64	24
256	381.42	377.02	503.34	333.34	318.4	282.62	1103.92	28
256	195.96	209.14	458.74	295.74	286.74	245.76	1127.86	32
512	285.588888888889	320.322222222222	1780.43333333333	264.811111111111	281.177777777778	272.4	4157.78888888889	8
512	357.08	316.56	1207.32	329.9	459.22	425.46	4236.7	12
512	215.3	215.52	2713.52	272.74	174.6	198.24	4311.98	16
512	232.76	230.26	2255.5	296.86	259.36	244.56	4216.78	20
512	477.1	419.82	3127.82	319.06	394.18	324.3	4293.98	24
512	419.48	414.9	3865.78	390.2	413.52	461.38	4339.56	28
512	419.78	471.82	1572.34	315.48	403.86	380.76	4262.66	32
1024	547.269662921348	537.988764044944	3337.02247191011	477.741573033708	560.775280898876	434.325842696629	14432.393258427	8
1024	477.54	1585.06	3154.92	592.88	708.96	513.4	15356.9	12
1024	635.06	657.16	3789.16	968.24	711.02	584.16	14455.72	16
1024	362	391.48	4225.1	309.46	340.42	270.44	14419.4	20
1024	448.04	470.08	3442.26	583.2	440.1	280.02	14744.84	24
1024	476.7	471.92	4734.02	495.36	400.4	786.78	14521.14	28
1024	355.24	404.36	4126.62	432.5	278.22	272.6	14704.22	32
2048	789.46511627907	779.244186046512	5965.22093023256	859.488372093023	1179.8488372093	555.546511627907	55519.9651162791	8
2048	638.617021276596	639.553191489362	6264.57446808511	1101.65957446809	816.170212765957	461.425531914894	56083.7021276596	12
2048	993.72	991.14	5383.58	897.02	609.1	355.72	56282.4	16
2048	1169.06382978723	1158.74468085106	4824.78723404255	991.106382978724	973.510638297872	770.617021276596	56844.4042553192	20
2048	919.74	966.26	7227.7	989.04	595.78	324.28	56304.88	24
2048	1532.26	1575.24	5574.06	1204.34	921.02	660.56	56064.24	28
2048	1198.44897959184	1166.63265306122	6535.28571428572	1151.89795918367	843.714285714286	525.673469387755	56778.8163265306	32
4096	1658.34939759036	1657.60240963855	12548.4096385542	3507.69879518072	1732.89156626506	1172.96385542169	210693.65060241	8
4096	2429.95454545455	2366.90909090909	12821.1818181818	3373.97727272727	1859.79545454545	833.25	209770.886363636	12
4096	2861.33333333333	2865.72916666667	14042.0833333333	3605.9375	2147.75	978.770833333333	212419.5625	16
4096	3194.02083333333	3208.77083333333	12299.125	3782.47916666667	2168.47916666667	999.604166666667	209105.083333333	20
4096	3366.25531914894	3513.34042553191	13577.914893617	4039.29787234043	2373.14893617021	1088.82978723404	208243.382978723	24
4096	3517.40425531915	3586.97872340425	14292.7872340426	4767.04255319149	2792.82978723404	1366.55319148936	212651.638297872	28
4096	3671.23404255319	3806.51063829787	12474.1063829787	4676.57446808511	2673.25531914894	1264.93617021277	215742.170212766	32
10240	28234.4	28815.35	61674.75	39701.225	19949.05	9353.675	1468651.025	8
10240	33970.05	33414.25	63039.3	30282.15	19013	7068.6	1470117.25	12
10240	18553.8076923077	17575.1153846154	59659.4615384615	25365.4615384615	13647.7692307692	6766.11538461539	1452370.76923077	16
10240	25510	26996.28	66759.8	30463.68	16127.64	8715.72	1453783.52	20
10240	24913.9666666667	23566.6666666667	60191.9333333333	28260.2333333333	15399.7333333333	8131.4	1455701.13333333	24
10240	26737.5185185185	30219.7037037037	59481.0740740741	28802.0740740741	16959.962962963	10092.6666666667	1443016.37037037	28
10240	26247.7916666667	26296.25	60415.75	28901.5	17386.3333333333	10410.4583333333	1456024.45833333	32

Table 4: Average GPU Kernel Times (Microseconds)

Figure 1: Cornerness Image

size	kernel	R-squared	P-value	Estimate	STD Error	t value	Pr(>|t|)
256	horizontal	0.174	0.3518	231.22	113.26	2.042	0.0967
	vertical	0.3155	0.1895	142.332	112.883	1.261	0.263
	magnitude	0.1577	0.3778	131.38	110.46	1.184	0.29
	gradient	0.1314	0.4243	162.587	137.72	1.181	0.291
	suppression	0.26	0.2423	162.273	101.98	1.591	0.172
	thrust	0.1427	0.4035	384.26	99.197	3.84	0.0117
	hysteresis	0.2033	0.3099	148.478	105.471	1.408	0.218
	edge	0.1702	0.3577	144.476	98.055	1.473	0.201
512	horizontal	0.4689	0.08963	317.495	93.89	3.382	0.0196
	vertical	0.563	0.05226	249.682	88.41	2.823	0.037
	magnitude	0.3606	0.1539	202.946	90.379	2.245	0.0747
	gradient	0.452	0.09799	188.552	81.012	2.2327	0.0674
	suppression	0.3463	0.1646	255.77	37.692	6.786	0.00106
	thrust	0.1765	0.348	1448.43	948.85	1.572	0.187
	hysteresis	0.1375	0.4128	252.233	106.886	2.36	0.0648
	edge	0.1715	0.3557	236.196	98.871	2.389	0.0625
1024	horizontal	0.02311	0.7449	949.104	79.625	11.919	7.33E-05
	vertical	0.05813	0.602	908.518	85.19	10.665	0.00125
	magnitude	0.355	0.157	608	88.4	6.81	0.000992
	gradient	0.2624	0.2399	1147	405	2.829	0.0367
	suppression	0.071	0.56365	679.162	222.7	3.05	0.0288
	thrust	0.5012	0.07508	2904.89	444.49	6.535	0.000126
	hysteresis	0.5993	0.04105	801.361	122.07	6.565	0.00123
	edge	0.8379	0.009205	492.132	216.45	2.274	0.0721
2048	horizontal	0.4969	0.0769	2028.3	454.21	4.466	0.00661
	vertical	0.7278	0.01466	1268	406.41	3.12	0.0262
	magnitude	0.588	0.04404	509.428	211.327	2.411	0.0608
	gradient	0.5848	0.04521	502.28	218.04	2.304	0.0695
	suppression	0.4913	0.07936	818.041	102.811	7.957	0.000506
	thrust	0.6502	0.04439	5579.8	867.33	6.433	0.00135
	hysteresis	0.5058	0.09311	993.45	217	4.558	0.00607
	edge	0.7743	0.01799	472.473	176.16	2.682	0.0437
4096	horizontal	0.6067	0.03903	4908	2067.23	2.374	0.0636
	vertical	0.6616	0.025	4404.41	1818.63	2.422	0.06
	magnitude	0.8985	0.001156	1400.07	252	5.556	0.0026
	gradient	0.9216	0.000621	1298.25	239.24	5.427	0.00288
	suppression	0.8614	0.002561	2763.62	232.1	11.907	7.36E-05
	thrust	0.6435	0.04619	12747.9	81.82	14.456	2.86E-05
	hysteresis	0.931	0.0004348	1372.493	114.991	11.936	7.28E-05
	edge	0.3775	0.1421	841.311	160.445	5.244	0.00334
10240	horizontal	0.6677	0.2482	41670	12378	3.366	0.02
	vertical	0.3329	0.175	47436.9	11166.1	4.248	0.00811
	magnitude	0.6082	0.05638	28821.2	4949.2	5.823	0.00211
	gradient	0.7967	0.01455	28118.17	5629.94	4.994	0.00412
	suppression	0.3108	0.1935	36051	4158	8.67	0.000337
	thrust	0.5012	0.09504	63453.33	2749.95	23.074	2.84E-06
	hysteresis	0.4243	0.1313	18719.47	2221.27	8.427	0.00036
	edge	0.3334	0.1746	6758.42	1287.15	5.251	0.00332

Table 5: Linear Regression of the kernels

block size	kernel	R-Squared	P-value	Estimate	Std. Error	t value	Pr(>|t|)	3072 Predict	5120 Predict	7680 Predict
8	horizontal	0.9283	0.001978	-6628.52	3982.15	-1.665	0.17133	12484.78	25538.08	41154.74
	vertical	0.9252	0.002153	-7059.916	4245.64	-1.663	0.17167	12863.23	26469.6	42747.96
	magnitude	0.8946	0.004324	-31.95.78	2214.21	-1.44	0.22242	5410.825	11288.64	1830.74
	gradient	0.8935	0.004411	-3265.26	2271.4114	-1.438	0.22393	5515.228	11511.8	18685.97
	thrust	0.9625	0.0005327	-4173.07	2768.44	-1.507	0.206196	14554.24	27343.92	42645.2
	suppression	0.9115	0.003025	-4533.043	2851.3998	-1.59	0.18709	7680.81	16022.17	26001.59
	hysteresis	0.9097	0.003157	-1963.632	1427.41	-1.376	0.24093	4080.421	8208.162	13146.5
	edge	0.9216	0.00237	-763.8074	613.8025	-1.244	0.28129	2043.676	3961.026	6254.901
								64633.21	130343.398	192467.601
16	horizontal	0.9337	0.001685	-8701.139	4994.556	-1.742	0.15645	16313.05	33396.31	53834.33
	vertical	0.9511	0.0009099	-5535.2398	2920.6166	-1.895	0.13096	11661.14	23405.28	37455.71
	magnitude	0.9438	0.001209	-1734.0985	1042.8813	-1.663	0.17169	3966.296	7859.296	12516.89
	gradient	0.9473	0.001061	-1595.0539	954.3295	-1.671	0.16996	3803.142	7489.801	11900.44
	thrust	0.9695	0.000353	-3510.75	2404.1402	-1.46	0.2179	14568.28	26915	41686.8
	suppression	0.935	0.00162	-2459.96	1537.708	-1.6	0.1849	5322.047	10636.71	16995.06
	hysteresis	0.9407	0.001348	-1212.96	785.0169	-1.545	0.1972	2957.476	5805.645	9213.131
	edge	0.9265	0.00208	-498.70955	427.36	-1.167	0.30806	1525.356	2907.677	4561.455
								60116.787	118415.719	188163.816
32	horizontal	0.9396	0.001395	-12365.507	6801.468	-1.818	0.1432	23441.79	47896.79	77152.73
	vertical	0.9251	0.0008747	-9088.527	4733.529	-1.92	0.12767	19069	38300.62	61307.72
	magnitude	0.9365	0.001546	-2617.0611	1576.455	-1.66	0.17223	5460.21	10976.61	17576.24
	gradient	0.9378	0.001483	-2583.55	1561.23	-1.655	0.1733	5504.79	11028.67	17637.31
	thrust	0.9637	0.0004996	-3750.9256	2656.2983	-1.412	0.2308	14517.54	26993.83	41920.23
	suppression	0.9448	0.001162	-2846	1617.3759	-1.76	0.15326	6086.34	12186.75	19485.13
	hysteresis	0.939	0.001426	-1604.26	1017.12	-1.577	0.18987	3719.292	7354.976	11704.63
	edge	0.9193	0.002514	-911.8429	697.1415	-1.308	0.26098	2226.959	4370.581	6935.162
								80025.921	159108.827	253719.152
							mh2d	3157.985	3173.386	3202.144
							md2h	3157.985	3173.144	3202.144
							total:	909491.8	1366899.28	N/A
								864327.57	1247622.49	N/A
								1063418.91	1654553.57	N/A
							actual	1108282	2385665	N/A
								1102468	2402616	N/A
								1182608	2572470	N/A
							difference:	198790.2	1018765.72	N/A
								238140.43	1154993.51	N/A
								119189.09	917916.43	N/A
							error:	0.1793678865	0.4270363693	N/A
								0.2160066596	0.4807233074	N/A
								0.1007849516	0.3568229872	N/A

Table 6: Run times and Predictions (microseconds)