# Performance Prediction for GPU-based HPC codes

Marco Lin
University of the Pacific

Hight Performance Computing
m_lin23@u.pacific.edu

***Abstract*—The purpose of this report is to performance prediction for GPU-based HPC code. The prediction of execution time is divided into two part. One is Kernel execution time. Another is memory copy of a randomly filled floating-point array from CPU host to GPU device (H2D) and from GPU device to CPU host (D2H). We will use R programming language to explore the correlation between different values and find the prediction model.**

***Keywords—C, Canny Edge Detector, CUDA, GPU, Prediction.***

## I. Introduction

The purpose of this report is to analyze the accelerating by designing a serial code and then using CUDA method to speedup. In Canny Edge Detector, there are several kernels, such as convolution, Suppressed, and Hysteresis. In those kernels, the program needs to get and calculate the value of each pixels. They take up execute time the most, thus by executing on GPUs device, we can effectively to observe the reducing of execution time. In addition, in order to prediction, we divided our execution time into three parts: kernel execution time, memory copy from host to device and memory copy from device to host. So, first we implemented the features detection. This kernel will detect different numbers of features along with different block sizes and image sizes. We tested program with several image sizes which are 256, 512, 1024, 2048, 4096, 8192, 10240. This time we only use one sigma 0.6. And, the most important part is block size. We tested with block sizes: 8,12,16,20,24,28,32. We have run 50 times to get a solid result. Moreover, for each kernel time, we considered that float point computation times and global memory access times to predict our kernel time.

## II. Implementation

### A. Canny edge detector with CUDA

We used this function to detect the edge of image. In order to effectively reduce our execution time. We used GPUs to implement our accelerating. On GPUs device, we can use global memory and shared memory to speed up our program. In convolution part, especially, we used shared memory. Other kernels used only global memory. We only used shared memory for convolution because during the calculation, some data were out of the bound that the threads need to access global memory to get the data what it needs. We can restore our data to shared memory to enhance their speed. For other part of functions, like Suppressed and Hysteresis, we only use global memory for implementation. This is due to there is not calculation which needs the data from different block.

### B. Corner Detection on GPU

In the Corner features detection kernel, the first step is to evaluate cornerness values for each pixel by computing the Z matrix over the neighborhood window (7x7) which is a similar step to convolution. After computing the cornerness values, we need to implement an algorithm which helps us to find the location of the maximum cornerness in a window of BLOCKSIZExBLOCKSIZE.
The following is how to implement:
1. Set the thread block dimensions as dimBlock(BLOCKSIZE,BLOCKSIZE)

2. Using shared memory, have threads in a thread block find the location of the maximum cornerness value. Have thread-0 of the block write the location of the maximum corner to the list of corners.
The following figure is an example of corners output with image-size is 256x256 and Block-Size is 8.

corneress

| Index:771 | I:3 | I:4 |
|---|---|---|
| Index:800 | I:3 | J:32 |
| Index:8016 | I:31 | J:80 |
| Index:6512 | I:25 | J:112 |
| Index:1183 | I:4 | J:159 |
| Index:931 | I:3 | J:163 |
| Index:978 | I:3 | J:210 |
| Index:1021 | I:3 | J:253 |
| Index:11267 | I:44 | J:3 |
| Index:13119 | I:51 | J:63 |
| Index:13890 | I:54 | J:66 |
| Index:15212 | I:59 | J:108 |
| Index:10132 | I:39 | J:148 |
| Index:16312 | I:63 | J:184 |
| Index:15055 | I:58 | J:207 |
| Index:10237 | I:39 | J:253 |
| Index:16650 | I:65 | J:10 |
| Index:21055 | I:82 | J:63 |
| Index:24389 | I:95 | J:69 |
| Index:22905 | I:89 | J:121 |
| Index:22928 | I:89 | J:144 |
| Index:16568 | I:64 | J:184 |
| Index:20427 | I:79 | J:203 |
| Index:19936 | I:77 | J:224 |
| Index:32522 | I:127 | J:10 |
| Index:31807 | I:124 | J:63 |

It appears that the indexes and the threads location I and J.

### C. Testing

After We implemented the code. We tested the program on GPU in several sizes of images and Block-Sizes. Image sizes {256, 512, 1024, 2048, 4096, 8192, and 10240} x {BLOCKSIZES 8, 12, 16, 20, 24 ,28, and 32} x {sigma 0.6}. In addition, in order to get a solid execution time, we executed the program in 30 times for each parameter.

In the figure 2, it shows the average execution time in each parameter.

execution_time(MicroSecond)

| BlockSize | Image-Hight | Image_Width | Sigma | i_o_time | kernel_time | communication_time | endToEndNoIandO | endToEndwithIandO |
|---|---|---|---|---|---|---|---|---|
| 8 | 256 | 256 | 0.6 | 1664 | 317 | 795 | 394535 | 391065 |
| 8 | 512 | 512 | 0.6 | 3683 | 942 | 2739 | 371915 | 362081 |
| 8 | 1024 | 1024 | 0.6 | 11553 | 2531 | 5873 | 387782 | 383232 |
| 8 | 2048 | 2048 | 0.6 | 31420 | 7165 | 17651 | 371036 | 301018 |
| 8 | 4096 | 4096 | 0.6 | 83210 | 54308 | 57252 | 603029 | 453749 |
| 8 | 8192 | 8192 | 0.6 | 277440 | 244661 | 214003 | 1466363 | 1022242 |
| 8 | 10240 | 10240 | 0.6 | 483865 | 295274 | 359343 | 2166675 | 1381534 |
| 12 | 256 | 256 | 0.6 | 716 | 272 | 470 | 222455 | 221153 |
| 12 | 512 | 512 | 0.6 | 1665 | 755 | 3095 | 277933 | 159817 |
| 12 | 1024 | 1024 | 0.6 | 8737 | 2662 | 4903 | 235379 | 244986 |
| 12 | 2048 | 2048 | 0.6 | 29571 | 10194 | 15384 | 285331 | 240212 |
| 12 | 4096 | 4096 | 0.6 | 32416 | 49322 | 23141 | 392203 | 352184 |
| 12 | 8192 | 8192 | 0.6 | 264705 | 267556 | 209200 | 1644761 | 1219253 |
| 12 | 10240 | 10240 | 0.6 | 424019 | 383380 | 294074 | 2053407 | 1372857 |
| 16 | 256 | 256 | 0.6 | 1527 | 435 | 662 | 436495 | 433750 |
| 16 | 512 | 512 | 0.6 | 3419 | 1192 | 2356 | 486852 | 469087 |
| 16 | 1024 | 1024 | 0.6 | 9275 | 4246 | 5791 | 511293 | 490480 |
| 16 | 2048 | 2048 | 0.6 | 39828 | 26777 | 20528 | 554129 | 458458 |
| 16 | 4096 | 4096 | 0.6 | 84981 | 43184 | 59393 | 618851 | 469431 |
| 16 | 8192 | 8192 | 0.6 | 266443 | 174774 | 252485 | 1642102 | 1203434 |
| 16 | 10240 | 10240 | 0.6 | 385434 | 304631 | 311565 | 2055560 | 1389587 |
| 20 | 256 | 256 | 0.6 | 974 | 15566 | 3538 | 294464 | 251342 |
| 20 | 512 | 512 | 0.6 | 3119 | 15721 | 3022 | 381569 | 275737 |
| 20 | 1024 | 1024 | 0.6 | 22475 | 17297 | 7533 | 375599 | 333727 |
| 20 | 2048 | 2048 | 0.6 | 45750 | 11165 | 13242 | 435272 | 362727 |
| 20 | 4096 | 4096 | 0.6 | 89172 | 44742 | 62721 | 663469 | 480882 |
| 20 | 8192 | 8192 | 0.6 | 258094 | 195301 | 214106 | 1529829 | 1120852 |
| 20 | 10240 | 10240 | 0.6 | 398766 | 330364 | 339009 | 2139809 | 1483318 |
| 24 | 256 | 256 | 0.6 | 852 | 7718 | 3382 | 314090 | 321495 |
| 24 | 512 | 512 | 0.6 | 2977 | 1241 | 2625 | 420554 | 411548 |
| 24 | 1024 | 1024 | 0.6 | 7096 | 4154 | 5629 | 427279 | 408947 |
| 24 | 2048 | 2048 | 0.6 | 24824 | 17371 | 15143 | 392919 | 392020 |
| 24 | 4096 | 4096 | 0.6 | 78888 | 45960 | 53370 | 661098 | 507868 |
| 24 | 8192 | 8192 | 0.6 | 277558 | 202817 | 208377 | 1525508 | 1069597 |
| 24 | 10240 | 10240 | 0.6 | 425547 | 373274 | 325903 | 2346146 | 1563821 |
| 28 | 256 | 256 | 0.6 | 5097 | 290 | 426 | 238649 | 228785 |
| 28 | 512 | 512 | 0.6 | 3215 | 984 | 2555 | 315648 | 308675 |
| 28 | 1024 | 1024 | 0.6 | 31176 | 3216 | 7234 | 302939 | 239293 |
| 28 | 2048 | 2048 | 0.6 | 39916 | 12372 | 16593 | 429956 | 359655 |
| 28 | 4096 | 4096 | 0.6 | 85308 | 49486 | 67409 | 542277 | 391162 |
| 28 | 8192 | 8192 | 0.6 | 287593 | 270414 | 215687 | 1467699 | 1008288 |
| 28 | 10240 | 10240 | 0.6 | 429577 | 386462 | 319922 | 2063922 | 1403417 |
| 32 | 256 | 256 | 0.6 | 971 | 342 | 495 | 277804 | 276682 |
| 32 | 512 | 512 | 0.6 | 3007 | 988 | 1612 | 238307 | 233120 |
| 32 | 1024 | 1024 | 0.6 | 9113 | 18387 | 9179 | 412478 | 356625 |
| 32 | 2048 | 2048 | 0.6 | 10140 | 4850 | 6087 | 426546 | 406763 |
| 32 | 4096 | 4096 | 0.6 | 58321 | 52017 | 61797 | 627108 | 497768 |
| 32 | 8192 | 8192 | 0.6 | 283911 | 213268 | 276564 | 1949619 | 1476606 |
| 32 | 10240 | 10240 | 0.6 | 449970 | 427565 | 361132 | 2359228 | 1617397 |

Figure 2.

The above executions will help us to develop performance prediction models for execution time prediction.

*D. Performance Modeling*

Our application's runtime is governed by the time taken by GPU kernels, H2D communications, D2H communications, and any CPU computations. Especially, entire code is on GPUs, Gaussian kernel evaluation which is on CPU is way too small for any meaningful modeling. So, we will not consider CPU computations time. For performance modeling, we have several modeling.

1. Modeling CPU-GPU Communications

In this part, we have to implement a simple benchmark that performs memory copy of a randomly filled floating-point array from CPU host to GPU device (H2D) and from GPU device to CPU host (D2H). For a given array size, we had performed 30 statistical executions to compute the average H2D time and D2H time for these array size: 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB.

And, Calculate the bandwidth as (data size/time), plot H2D bandwidth vs. array-size (KB) and observe the trend. (Dependent variable: bandwidth, independent variable: data size). The figures 3 to 5 are H2D time and the figures 6 to 8 are D2H time. The unit of time is microsecond.

| arraySize_KB | time_ms | bandwidth |
|---|---|---|
| 1 | 11.70000 | 0.08547009 |
| 2 | 12.60000 | 0.15873016 |
| 4 | 13.46667 | 0.29702972 |
| 8 | 19.66667 | 0.40677967 |
| 16 | 56.50000 | 0.28318584 |
| 32 | 67.70000 | 0.47267358 |
| 64 | 86.46667 | 0.74016962 |
| 128 | 162.16667 | 0.78931138 |
| 256 | 304.89999 | 0.83961956 |
| 512 | 645.43335 | 0.79326549 |
| 1024 | 1013.29999 | 1.01055957 |
| 2048 | 2083.86670 | 0.98278839 |
| 4096 | 4047.03345 | 1.01209937 |
| 8196 | 8887.46680 | 0.92219754 |
| 16384 | 18273.09961 | 0.89661855 |
| 32768 | 29746.73242 | 1.10156637 |
| 65536 | 73312.42969 | 0.89392754 |
| 131072 | 137917.32810 | 0.95036644 |
| 262144 | 261403.17190 | 1.00283404 |
| 524288 | 543245.25000 | 0.96510370 |

Figure 3.

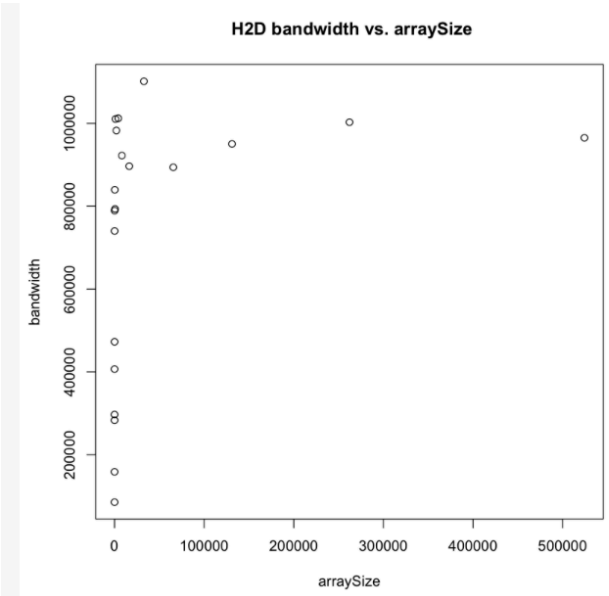


Figure 4.

```
Call:
lm(formula = bandwidth ~ arraySize)

Residuals:
    Min      1Q  Median      3Q     Max
-602222 -231512  103842  203694  387298

Coefficients:
             Estimate  Std. Error t value     Pr(>|t|)
(Intercept) 687690.7993  75502.0145   9.108 0.0000000368 ***
arraySize        0.8111      0.5577   1.454        0.163
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 311300 on 18 degrees of freedom
Multiple R-squared:  0.1051, Adjusted R-squared:  0.05542
F-statistic: 2.115 on 1 and 18 DF,  p-value: 0.1631
```

Figure 5.

The formula for H2D bandwidth is:

H2D_bandwidth = 687690.8+0.8111*array Size

| arraySize_KB | time_ms | bandwidth |
|---|---|---|
| 1 | 15.86667 | 0.06302521 |
| 2 | 15.06667 | 0.13274336 |
| 4 | 19.96667 | 0.20033389 |
| 8 | 25.30000 | 0.31620555 |
| 16 | 48.43333 | 0.33035099 |
| 32 | 89.13333 | 0.35901272 |
| 64 | 179.70000 | 0.35614914 |
| 128 | 262.43332 | 0.48774295 |
| 256 | 434.60001 | 0.58904739 |
| 512 | 906.09998 | 0.56505906 |
| 1024 | 1220.00000 | 0.83934426 |
| 2048 | 2955.00000 | 0.69306261 |
| 4096 | 4789.39990 | 0.85522197 |
| 8196 | 13376.66699 | 0.61270868 |
| 16384 | 18093.43359 | 0.90552188 |
| 32768 | 33889.73438 | 0.96690047 |
| 65536 | 64856.69922 | 1.01047387 |
| 131072 | 138962.00000 | 0.94322189 |
| 262144 | 309439.34380 | 0.84715795 |
| 524288 | 540365.93750 | 0.97024620 |

Figure 6.

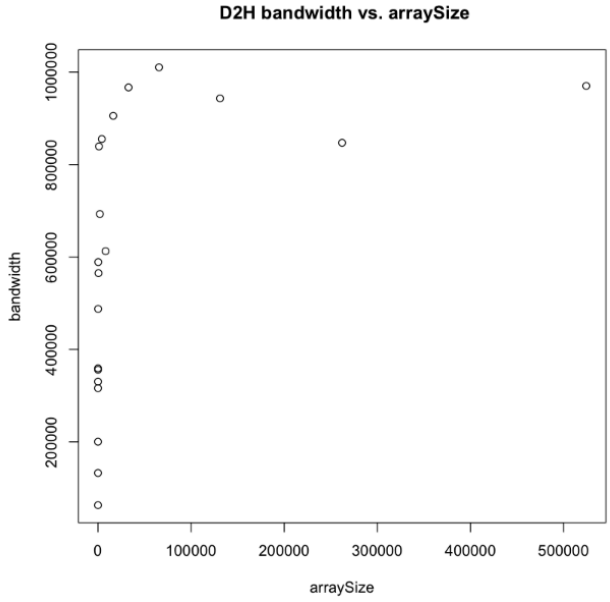


Figure 7.

```
Call:
lm(formula = bandwidth ~ arraySize)

Residuals:
    Min      1Q  Median      3Q     Max
-480744 -194126   16035  263684  393695

Coefficients:
             Estimate  Std. Error t value    Pr(>|t|)
(Intercept) 543768.0945  67834.5476   8.016 0.000000238 ***
arraySize        1.1140      0.5011   2.223      0.0392 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 279700 on 18 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared:  0.2154,        Adjusted R-squared:  0.1718
F-statistic: 4.943 on 1 and 18 DF,  p-value: 0.03925
```

Figure 8.

The formula for D2H bandwidth is:

D2H_bandwidth = 543768.1+1.1140*array Size

2. Modeling Kernel Times

For each of the Canny edge kernels and the cornerness kernel, we need to manually compute the number of floating-point computations and memory accesses per thread. When we multiply these with the total number of threads for that kernel, we can get total floating-point operations for the kernel (FLOPS_kernel) and total memory accesses (BYTES_kernel). Then, we explored the correlation between these two variables and the kernel runtime to decide a candidate mathematical model. (Independent variable: FLOPS_kernel, BYTES_kernel; Dependent variable: kernel time). For the approximation kernel, because we are testing for different values of BLOCKSIZE, the independent variables are FLOPS_kernel, BYTES_kernel, and BLOCKSIZE.

In the figure 9, it shows the FLOPS_kernel, BYTES_kernel, and kernel time in different image size and BLOCKSIZE. And the time unit is microsecond.

kernel_time(Microsecond)

| FLOPS_kernel | BYTES_kernel | Kernel time | imagesize | blocksize |
|---|---|---|---|---|
| 15104486 | 4539716 | 300 | 256 | 8 |
| 60963808 | 18575684 | 563 | 512 | 8 |
| 245433008 | 73971976 | 1776 | 1024 | 8 |
| 926885888 | 289379552 | 7468 | 2048 | 8 |
| 2261574400 | 661418656 | 44103 | 3072 | 8 |
| 4583325696 | 1800351360 | 56821 | 5120 | 8 |
| 4943773696 | 3209166848 | 158718 | 7680 | 8 |
| 15104486 | 2985540 | 297 | 256 | 16 |
| 60963808 | 12162372 | 820 | 512 | 16 |
| 245433008 | 49089028 | 2719 | 1024 | 16 |
| 926885888 | 189815488 | 10334 | 2048 | 16 |
| 2261574400 | 425587472 | 24727 | 3072 | 16 |
| 4583325696 | 529999408 | 67310 | 5120 | 16 |
| 4943773696 | 1984255872 | 176570 | 7680 | 16 |
| 15104486 | 2146436 | 467 | 256 | 32 |
| 60963808 | 8805956 | 1397 | 512 | 32 |
| 245433008 | 22295844 | 11021 | 1024 | 32 |
| 926885888 | 142522872 | 18643 | 2048 | 32 |
| 2261574400 | 302274928 | 39525 | 3072 | 32 |
| 4583325696 | 490202112 | 97976 | 5120 | 32 |
| 4943773696 | 1830021568 | 262450 | 7680 | 32 |

Figure 9.

```
Call:
lm(formula = Kernel.time ~ BYTES_kernel + FLOPS_kernel + blocksize)

Residuals:
   Min     1Q Median     3Q    Max
-66315 -18646   1315  12497  93630

Coefficients:
                Estimate       Std. Error t value Pr(>|t|)
(Intercept)  -22996.576997398  12733.361375750  -1.806   0.0887 .
BYTES_kernel      0.000042797      0.000015612   2.741   0.0139 *
FLOPS_kernel      0.000014503      0.000006649   2.181   0.0435 *
blocksize        40.819234136     18.390685366   2.220   0.0403 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 33210 on 17 degrees of freedom
Multiple R-squared:  0.8148,        Adjusted R-squared:  0.7822
F-statistic: 24.94 on 3 and 17 DF,  p-value: 0.000001864
```

Figure 10.

Kernel time formula is

kernel_time = 22996.576997398+

BYTES_kernel*0.000042797+

FLOPS_kernel*0.000014503+

BLOCKSIZE*40.81923

In figure 10, it shows the coefficients, p-values, and R^2 values and we can make the kernel time formula.

## III. Results and Analysis

By our prediction modeling, we predict the execution time as:

Execution Predict= Predicted H2D time + Predicted D2H time + Predicted kernel runtimes.

Predict the execution times for these configurations (all with Gaussian sigma=0.6):

1. Image Size 3072x 3072 with approximation algorithm BLOCKSIZE=8, 16, 32.

2.Image Size 5120x5120 with approximation algorithm BLOCKSIZE=8, 16, 32.

3.Image Size 7680x7680 with approximation algorithm BLOCKSIZE=8, 16, 32.

For the above 9 cases, we have a table called "Predicted H2D, D2H, and kernel times" (figure 11.) that gives the predicted H2D time, D2H time, kernel times, and CPU time. Also, another table called "Actual H2D, D2H, and kernel times for the actual times for H2D, D2H, kernels (figure 12.). These are obtained by executing code with the above configurations. The table (figure 13.) called "prediction errors for the tested configuration" that gives the prediction error for each of the above 9 configurations as:

Error = (predicted execution time - actual execution time)/actual execution time * 100%

Predicted H2D, D2H, kernel, and CPU times(second)

| | H2D | D2H | Kernel | Execution_predict |
|---|---|---|---|---|
| 3072x3072+ BLOCKSIZE = 8 | 0.01325 | 0.0166343 | 0.055320 | 0.0852043 |
| 3072x3072+ BLOCKSIZE = 16 | 0.01325 | 0.0166343 | 0.067483 | 0.0973673 |
| 3072x3072+ BLOCKSIZE = 32 | 0.01325 | 0.0166343 | 0.192938 | 0.2228223 |
| 5120x5120+ BLOCKSIZE = 8 | 0.03613 | 0.044732 | 0.028388 | 0.10925 |
| 5120x5120+ BLOCKSIZE = 16 | 0.03613 | 0.044732 | 0.075838 | 0.1567 |
| 5120x5120+ BLOCKSIZE = 32 | 0.03613 | 0.044732 | 0.198382 | 0.279244 |
| 7680x7680+ BLOCKSIZE = 8 | 0.0784 | 0.09474 | 0.044932 | 0.218072 |
| 7680x7680+ BLOCKSIZE = 16 | 0.0784 | 0.09474 | 0.123012 | 0.296152 |
| 7680x7680+ BLOCKSIZE = 32 | 0.0784 | 0.09474 | 0.309238 | 0.482378 |

Figure 11.

Actual H2D, D2H, kernel, and CPU times(second)

| | H2D | D2H | Kernel | Execution_predict |
|---|---|---|---|---|
| 3072x3072+ BLOCKSIZE = 8 | 0.009657566 | 0.012346 | 0.044103 | 0.066106566 |
| 3072x3072+ BLOCKSIZE = 16 | 0.009657566 | 0.012346 | 0.056821 | 0.078824566 |
| 3072x3072+ BLOCKSIZE = 32 | 0.009657566 | 0.012346 | 0.158718 | 0.180721566 |
| 5120x5120+ BLOCKSIZE = 8 | 0.0300756 | 0.047185 | 0.024727 | 0.1019876 |
| 5120x5120+ BLOCKSIZE = 16 | 0.0300756 | 0.047185 | 0.067310 | 0.1445706 |
| 5120x5120+ BLOCKSIZE = 32 | 0.0300756 | 0.047185 | 0.176570 | 0.2538306 |
| 7680x7680+ BLOCKSIZE = 8 | 0.05595713 | 0.07510 | 0.039525 | 0.17058213 |
| 7680x7680+ BLOCKSIZE = 16 | 0.05595713 | 0.07510 | 0.097976 | 0.22903313 |
| 7680x7680+ BLOCKSIZE = 32 | 0.05595713 | 0.07510 | 0.262450 | 0.39350713 |

Figure 12.

prediction errors for the tested configuration

| | H2D% | D2H% | Kernel% | Execution_predict |
|---|---|---|---|---|
| 3072x3072+ BLOCKSIZE = 8 | 37 | 35 | 25 | 29 |
| 3072x3072+ BLOCKSIZE = 16 | 37 | 35 | 19 | 24 |
| 3072x3072+ BLOCKSIZE = 32 | 37 | 35 | 22 | 23 |
| 5120x5120+ BLOCKSIZE = 8 | 20 | -5 | 15 | 7 |
| 5120x5120+ BLOCKSIZE = 16 | 20 | -5 | 13 | 8 |
| 5120x5120+ BLOCKSIZE = 32 | 20 | -5 | 12 | 10 |
| 7680x7680+ BLOCKSIZE = 8 | 40 | 26 | 14 | 28 |
| 7680x7680+ BLOCKSIZE = 16 | 40 | 26 | 26 | 29 |
| 7680x7680+ BLOCKSIZE = 32 | 40 | 26 | 18 | 23 |

Figure 13.

As we seen, H2D and D2H execution time will not change by different BLOCKSIZE because two execution time only execute the memory copy. And CPU time are too small to predict. So, kernel time will be the important part focus on. In the tables, for kernel execution time, we got 12% to 26% prediction error which is not too bad due to the formula of kernel time whose multiple R-squared value is up to 0.8148 and adjusted R-squared also up to 0.783. For H2D and D2H formula, multiple R-squared is only around 0.1 to 0.2. And the way to enhance our accuracy of H2D and D2H, we may need more sample data. In addition, in prediction error table, the range of percentage is from -5 to 40 because the accuracy of prediction is affected by many conditions. Sometimes, the execution time is slower than our expectations due to the computing resources are competed.

## IV. Conclusion

One of the strengths of statistics machine learning method is that allows us to do large scale of prediction which is not able to practically test on any devices due to limitation of computing resource. But on the other hand, this is also one of weakness of statistics machine meaning method. In order to getting an accurate prediction, we need to collect as much as possible data or sample for our prediction. If we are not able to collect enough data, it may affect our accuracy of result. The most challenging thing is calculation of global access in the kernel which also uses shared memory. Because not each thread access global memory, we need to apply software method of counting instead of manual counting that helps us to understand more about how threads work .