«به نام خدا»

طراحی سیستم های نهفته مبتنی بر هسته

تکلیف کامپیوتری شماره 2: CUDA

پردیس دانشکده‌های فنی دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

استاد: دکتر احمد شعبانی

نیم‌سال دوم سال تحصیلی 1401-1400

نگارش: علی ایمانقلی 810197692

مراحل نصب:

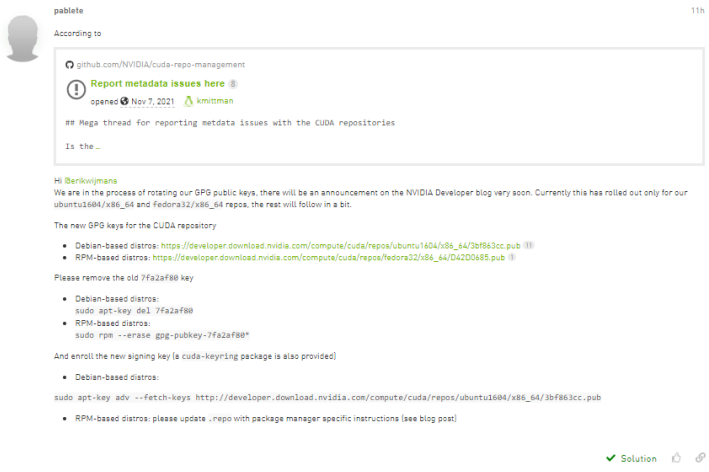
در ابتدا طبق دستورات موجود در صورت پروژه اقدام به نصب شد، ولیکن به علت تغییراتی که در رپوزیتوری مرتبط با NVIDIA به وجود آمده بود در اجرای دستور زیر، ارور ذیل نمایش داده می شد:

```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

```
W: GPG error: https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64  InRelease: The following signatures couldn't be verified because the public key is not available: NO_PUBKEY A4B469963BF863CC
E: The repository 'https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64  InRelease' is no longer signed.
N: Updating from such a repository can't be done securely, and is therefore disabled by default.
N: See apt-secure(8) manpage for repository creation and user configuration details.
```

این ارور در هنگام اجرای دستور apt-get update به وجود می آید.

برای رفع این مشکل طبق بررسی که در اینترنت انجام شد، راهکار زیر یافت شد.



pablete 11h

According to

github.com/NVIDIA/cuda-repo-management

Report metadata issues here 0

opened Nov 7, 2021 kmittman

Mega thread for reporting metdata issues with the CUDA repositories

Is the …

Hi @erikwijmans

We are in the process of rotating our GPG public keys, there will be an announcement on the NVIDIA Developer blog very soon. Currently this has rolled out only for our ubuntu1604/x86_64 and Fedora32/x86_64 repos, the rest will follow in a bit.

The new GPG keys for the CUDA repository

- Debian-based distros: https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/3bf863cc.pub
- RPM-based distros: https://developer.download.nvidia.com/compute/cuda/repos/fedora32/x86_64/D42D0685.pub

Please remove the old 7fa2af80 key

- Debian-based distros:
  sudo apt-key del 7fa2af80
- RPM-based distros:
  sudo rpm --erase gpg-pubkey-7fa2af80*

And enroll the new signing key (a cuda-keyring package is also provided)

- Debian-based distros:

sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/3bf863cc.pub

- RPM-based distros: please update .repo with package manager specific instructions (see blog post)

Solution

در نتیجه با اجرای دستورات زیر، ارور مربوط به رپوزیتوری برطرف شد:

sudo apt-key del 7fa2af80
sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/3bf863cc.pub

```
[13] !sudo apt-key del 7fa2af80

    OK
```

```
!sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/3bf863cc.pub

Executing: /tmp/apt-key-gpghome.wbThX5KNnY/gpg.1.sh --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/3bf863cc.pub
gpg: requesting key from 'http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/3bf863cc.pub'
gpg: key A4B469963BF863CC: public key "cudatools <cudatools@nvidia.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

پس از اجرای دستورات بالا ارور زیر نمایش به وجود می آید که توسط اجرای دستور زیر برطرف می شود:

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys F60F4B3D7FA2AF80

```
ous index files will be used. GPG error: file:/var/cuda-repo-9-2-local  Release: The following signatures couldn't be verified because the public key is not available: NO_PUBKEY F60F4B3D7FA2AF80
ous index files will be used. GPG error: https://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu1804/x86_64  Release: The following signatures couldn't be verified because the public key is not available: NO_PUBKEY F60F4B3D7FA2AF80
verified because the public key is not available: NO_PUBKEY F60F4B3D7FA2AF80
x86_64/Release.gpg  The following signatures couldn't be verified because the public key is not available: NO_PUBKEY F60F4B3D7FA2AF80
```

```
[16] !sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys F60F4B3D7FA2AF80

    Executing: /tmp/apt-key-gpghome.YEpPZ3PDJs/gpg.1.sh --keyserver keyserver.ubuntu.com --recv-keys F60F4B3D7FA2AF80
    gpg: key F60F4B3D7FA2AF80: public key "cudatools <cudatools@nvidia.com>" imported
    gpg: Total number processed: 1
    gpg:               imported: 1
```

و در نهایت دستور apt-get update بدون ارور اجرا می شود.

در ادامه تصاویر مربوط به اجرای کامل مراحل نصب آورده ده است:

```
[19] !apt-get --purge remove cuda nvidia* libnvidia-*
    !dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
    !apt-get remove cuda-*
    !apt autoremove
    !apt-get update
```

```
Hit:1 http://archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64  InRelease
Hit:3 http://archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 https://cloud.r-project.org/bin/linux/ubuntu bionic-cran40/ InRelease
Hit:5 http://archive.ubuntu.com/ubuntu bionic-backports InRelease
Ign:6 https://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu1804/x86_64  InRelease
Hit:7 https://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu1804/x86_64  Release
Hit:8 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:9 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic InRelease
Hit:10 http://ppa.launchpad.net/cran/libgit2/ubuntu bionic InRelease
Hit:11 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu bionic InRelease
Hit:12 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu bionic InRelease
Reading package lists... Done
```

```
[20] !wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64 -O cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
    !dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
    !apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
    !apt-get update
    !apt-get install cuda-9.2
```

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git

Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-no41nh6a
  Running command git clone -q https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-no41nh6a
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-py3-none-any.whl size=4306 sha256=c6abdf7babfeb7d7f307387d6da0d0c22ace45cf007eeb2bb62c6bd2db1d01ba
  Stored in directory: /tmp/pip-ephem-wheel-cache-e1zf539d/wheels/ca/33/8d/3c86eb85e97d2b6169d95c6e8f2c297fdec60db6e84cb56f5e
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

```
%load_ext nvcc_plugin

created output directory at /content/src
Out bin /content/result.out
```

کد قسمت CPU(serial):

در این قسمت کد مربوط به اجرای الگوریتم سوبل توسط CPU آورده شده است.

با اجرای این تابع حلقه‌های for تو در تو در کل به تعداد 227 * 227 مرتبه اجرا می شوند و کانولوشن بین تصویر و ماسک عمودی سوبل را انجام می دهند.

ورودی h_inp ارایه ی تک بعدی است که پیکسل های تصویر را در خود نگه می دارد و در main تعریف شده است.

با وجود اینکه تصویر 2 بعدی می باشد، ولیکن مقادیر آن به صورت تک بعدی و خطی ذخیره شده اند و به همین علت در هنگام دستری به پیکسل های تصویر و ماسک سوبل خانه ی مورد نظر را با انجام عملیات بر روی x و y تعیین می نماییم.

```
void sobel_cpu(int *h_outp, int *h_inp, int *h_mask)
{
  for(int i=1 ; i<ROWS-1 ; i++)
  {
    for(int j=1 ; j<COLS-1 ; j++)
    {
        h_outp[i*ROWS + j] = h_inp[i*ROWS + j - 4] * h_mask[0]  +
                             h_inp[i*ROWS + j - 3] * h_mask[1]  +
                            h_inp[i*ROWS + j - 2] * h_mask[2]  +
                            h_inp[i*ROWS + j - 1] * h_mask[3]  +
                            h_inp[i*ROWS + j    ] * h_mask[4]  +
                            h_inp[i*ROWS + j + 1] * h_mask[5]  +
                            h_inp[i*ROWS + j + 2] * h_mask[6]  +
                            h_inp[i*ROWS + j + 3] * h_mask[7]  +
                            h_inp[i*ROWS + j + 4] * h_mask[8] ;
    }

  }
}
```

کد قسمت(parallel)GPU:

در ای قسمت کد مربوط به الگوریتم سوبل در سمت GPU قرار دارد.

در ابتدای کد اندیس المان را به وسیله‌ی اطلاعات thread مورد نظر می یابیم و سپس با بررسی شرط مشخص میکنیم که این اندیس از مقدار 227*227 بیشتر نباشد تا محاسبات به اشتباه انجام نشود و سپس بررسی می کنیم که ماسک سوبل در حاشیه‌ی تصویر قرار نگیرد. زیر در این صورت باید ضرب هایی در مقادیر نامعلوم که جزء پیکسل های تصاویر نیستند انجام شود.

```
__global__ void sobel_gpu(int *d_outp, int *d_inp, int *d_mask)
{
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  unsigned int idx = y*ROWS + x;

  if(idx < COLS * ROWS)
  {
  if( y!=0 || y!=(ROWS-1) || x!=0 || x!=(COLS-1))
  { d_outp[y*ROWS + x] = d_inp[y*ROWS + x - 4] * d_mask[0] +
              d_inp[y*ROWS + x - 3] * d_mask[1] +
              d_inp[y*ROWS + x - 2] * d_mask[2] +
              d_inp[y*ROWS + x - 1] * d_mask[3] +
              d_inp[y*ROWS + x    ] * d_mask[4] +
              d_inp[y*ROWS + x + 1] * d_mask[5] +
              d_inp[y*ROWS + x + 2] * d_mask[6] +
              d_inp[y*ROWS + x + 3] * d_mask[7] +
              d_inp[y*ROWS + x + 4] * d_mask[8];
  }
  }
}
```

کد قسمت main:

در قطعه کد زیر در ابتدا ماسک عمودی سوبل و آرایه‌ی مربوط به تصویر را تعریف می نماییم، برای مدیریت راحت تر این آرایه ها با اینکه مفهوم دوبعدی دارند، آن ها را به صورت تک بعدی تعریف می نماییم.

سپس آرایه ی مورد نظر را به وسیله‌ی دستور srand با مقدار random پر می نماییم.

و در ادامه حافظه ها را هم به پارمتر های Device و هم به پارامتر های Host تخصیص می دهیم.

در میانه های کد نیز انتقال حافظه های از Host به Device و از Device به Host قرار دارد.

در انتها فراخوانی عملیات sobel برای CPU و GPU قرار دارد، که قسمت CPU آن کامنت شده است.

و در اخر پس از انتقال نتیجه از Device به Host ، حافظه های تخصیص داده شده را حذف می نماییم.

```
57 int main(void)
58 {
59   int *h_inp;
60   int *h_outp;
61   int *h_mask;
62
63   int picSize = ROWS * COLS * sizeof(int);
64   int maskSize = 3 * 3 * sizeof(int);
65
66   h_inp = (int*)malloc(picSize);
67    h_outp = (int*)malloc(picSize);
68    h_mask = (int*)malloc(maskSize);
69
70   srand(time(NULL));
71   for(int i=0 ; i<ROWS ; i++)
72   {
73     for(int j=0 ; j<COLS ; j++)
74     {
75         h_inp[i*ROWS + j] = ((rand() % 10)+1);
76     }
77   }
78   h_mask[0] = -1;
79   h_mask[1] = 0;
80   h_mask[2] = 1;
81   h_mask[3] = -2;
82   h_mask[4] = 0;
83   h_mask[5] = 2;
84   h_mask[6] = -1;
85   h_mask[7] = 0;
86   h_mask[8] = 1;
87
88
89   int *d_inp;
90    int *d_outp;
91    int *d_mask;
92  cudaMalloc((void**)&d_inp, picSize);
93  cudaMalloc((void**)&d_outp, picSize);
94  cudaMalloc((void**)&d_mask, maskSize);
```

```
95
96  cudaMemcpy(d_inp, h_inp, picSize, cudaMemcpyHostToDevice);
97  cudaMemcpy(d_outp, h_outp, picSize, cudaMemcpyHostToDevice);
98  cudaMemcpy(d_mask, h_mask, maskSize, cudaMemcpyHostToDevice);
99
100
101 for(int i=0 ; i<ROWS ; i++)
102 {
103   for(int j=0 ; j<COLS ; j++)
104   {
105       h_outp[i*ROWS + j] = 0;
106   }
107 }
108
109
110 /*
111 clock_t start_serial, end_serial;
112
113 start_serial = clock();
114
115 sobel_cpu(h_outp, h_inp, h_mask);
116
117 end_serial = clock();
118
119 printf("CPU(serial) time: %f s.\n",(end_serial-start_serial)/(float)CLOCKS_PER_SEC);
120
121 for(int i=0 ; i<ROWS ; i++)
122 {
123   for(int j=0 ; j<COLS ; j++)
124   {
125     if( i==0 || i==(ROWS-1) || j==0 || j==(COLS-1))
126     {
127       h_outp[i*ROWS + j] = UNDECLARE;
128     }
129   }
130 }
131 */
132
```

```
135 dim3 dimBlock(32, 32);
136 dim3 dimGrid(  ((COLS+dimBlock.x-1)/dimBlock.x),  ((ROWS+dimBlock.y-1)/dimBlock.y)  );
137
138 /* for part3 */
139 // dim3 dimGrid(2,13);
140
141
142 sobel_gpu<<<dimGrid, dimBlock>>>(d_outp, d_inp, d_mask);
143
144 clock_t start_gpu, end_gpu;
145
146 start_gpu = clock();
147
148 cudaMemcpy(h_outp,d_outp,picSize,cudaMemcpyDeviceToHost);
149
150   end_gpu = clock();
151
152 printf("GPU(parallel) time: %f s.\n",(end_gpu-start_gpu)/(float)CLOCKS_PER_SEC);
153 printf("dimGrid.x %d dimBlock.x %d \n", dimGrid.x, dimBlock.x);
154 printf("dimGrid.y %d dimBlock.y %d \n", dimGrid.y, dimBlock.y);
155 printf("dimGrid.z %d dimBlock.z %d \n", dimGrid.z, dimBlock.z);
156
157
158 for(int i=0 ; i<ROWS ; i++)
159 {
160   for(int j=0 ; j<COLS ; j++)
161   {
162     if( i==0 || i==(ROWS-1) || j==0 || j==(COLS-1))
163     {
164       h_outp[i*ROWS + j] = UNDECLARE;
165     }
166   }
167 }
168
169
```

در این حلقه‌ی for حاشیه های تصویر را که الگوریتم سوبل آن ها را تعیین نکرده است با مقدار UNDECLARE پر می کنیم.

مقدار UNDECLARE را به دلخواه برابر با 999- در نظر گرفته شده است.

```
171  free( h_inp );
172    free( h_outp );
173    free( h_mask );
174
175  cudaFree(d_inp);
176  cudaFree(d_outp);
177  cudaFree(d_mask);
178
179  return 0;
180 }
```

صحت عملیات:

برای بررسی اینکه الگوریتم سوبل به درستی اجرای می شود، تست زیر را انجام شد:

برای راحتی بررسی، ابعاد تصویر را 3*3 در نظر می گیریم:

شکل زیر مربوط به اجرای عملیات سوبل توسط CPU می باشد:

$$(7 * -1) + (1 * 0) + (6 * 1) + (6 * -2) + (9 * 0) + (3 * 2) + (9 * -1) + (6 * 0) + (5 * 1) = -11$$

```
7  1  6
6  9  3
9  6  5


-999  -999  -999
-999  -11  -999
-999  -999  -999
```

شکل زیر مربوط به اجرای عملیات سوبل توسط CPU می باشد:

$$(7 * -1) + (4 * 0) + (7 * 1) + (4 * -2) + (8 * 0) + (5 * 2) + (9 * -1) + (6 * 0) + (2 * 1) = -5$$

```
7  4  7
4  8  5
9  6  2


-999  -999  -999
-999  -5  -999
-999  -999  -999
```

همانطور که مشخص است الگوریتم سوبل به درستی اجرا می شود.

بخش1:

نتیجه اجرای الگوریتم سوبل در حالت CPU:

*کد بخش CPU به همراه توضیح، بالاتر آورده شده است.

```
CPU(serial) time: 0.000697 s.
```

بخش2:

نتیجه اجرای الگوریتم سوبل در حالت GPU:

*کد بخش GPU به همراه توضیح، بالاتر آورده شده است.

```
GPU(parallel) time: 0.000080 s.
```

بخش3:

```
GPU(parallel) time: 0.000081 s.
dimGrid.x 8 dimBlock.x 32
dimGrid.y 8 dimBlock.y 32
dimGrid.z 1 dimBlock.z 1
```

```
GPU(parallel) time: 0.000077 s.
dimGrid.x 8 dimBlock.x 32
dimGrid.y 15 dimBlock.y 16
dimGrid.z 1 dimBlock.z 1
```

```
GPU(parallel) time: 0.000078 s.
dimGrid.x 15 dimBlock.x 16
dimGrid.y 15 dimBlock.y 16
dimGrid.z 1 dimBlock.z 1
```

```
GPU(parallel) time: 0.000080 s.
dimGrid.x 29 dimBlock.x 8
dimGrid.y 15 dimBlock.y 16
dimGrid.z 1 dimBlock.z 1
```

```
GPU(parallel) time: 0.000096 s.
dimGrid.x 29 dimBlock.x 8
dimGrid.y 57 dimBlock.y 4
dimGrid.z 1 dimBlock.z 1
```

در جدول زیر تمامی مشخصات بالا به همراه speed up حاصل نسبت به اجرای سریال آورده شده است.

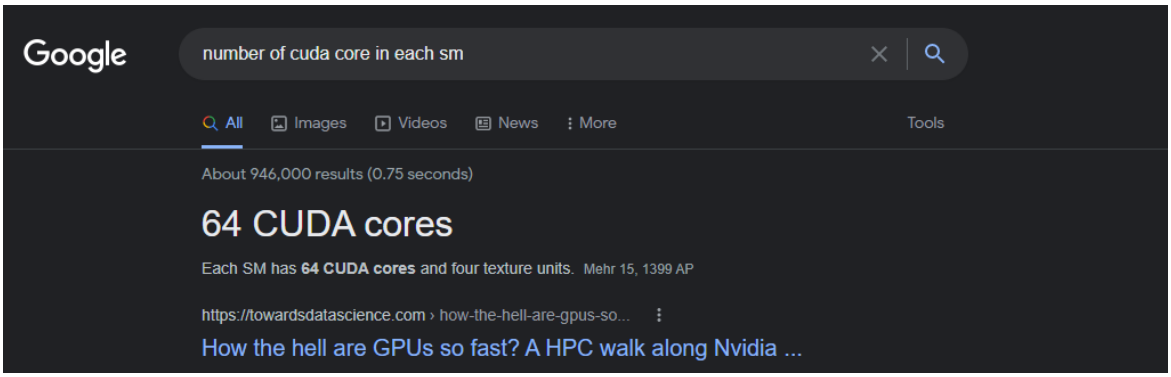
threads per block	Grid size	block size	time	Speed up
1024	(8,8,1)	(32,32,1)	0.000081s	8.6049
512	(8,15,1)	(32,16,1)	0.000077s	9.0519
256	(15,15,1)	(16,16,1)	0.000078s	8.9359
128	(29,15,1)	(8,16,1)	0.000080s	8.7125
32	(29,57,1)	(8,4,1)	0.000096s	7.2604

بخش4:

برای رسید به بیشترین تسریع ممکن باید تعداد thread در هر block را به نحوی تنظیم نماییم که، از یک طرف CUDA core ای بی کار نماند و از طرف دیگری تعداد thread ها به قدری نباشد که load فراتر از حد بر روی CUDA core قرار بگیرد و برای اجرای thread های بایند شده به CUDA core نیازمند به توقف بشویم تا تمامی دستورات توسط CUDA core اجرا شود.

بدین منظور بهترین حالت به نحوی است که حد واسط دو حالت بالا قرار بگیرد. حالت مناسب می تواند به نحوی باشد که چینش thread ها مضربی از warp باشد.

همانطور که می دانیم هر block به یک SMبایند می شود و هر SM به تعداد 64 CUDA core دارا می باشد.



از طرف دیگری می دانیم که هر warp تعداد 32 عدد thread را شامل می شود؛

بنابراین می توان block size را به صورت 32*64 در نظر گرفت.

حال برای grid size به نحوی تعیین می نماییم که تعداد thread های مازاد برای تعداد element کمینه شود که با درنظر گرفتن ابعاد 2*13 برای grid می توان به این امر دست یافت.

در شکل زیر نتیجه‌ی اجرا به ازای ابعاد بالا شده نمایش داده شده است:

```
GPU(parallel) time: 0.000070 s.
dimGrid.x 2 dimBlock.x 32
dimGrid.y 13 dimBlock.y 64
dimGrid.z 1 dimBlock.z 1
```

threads per block	Grid size	block size	time	Speed up
32*64=2048	(2,13,1)	(32,64,1)	0.000070s	9.9571

Speed up حاصل از تمامی Speed up های بدست آمده در بخش 3 بالاتر است.

بخش5:

* در بخش GPU activities بیشترین زمان صرف انتقال داده از Host به Device و بلعکس شده است.

(حدود 68.79% درصد از کل زمان صرف شده توسط Device)

و همانطور که آموختیم این بخش bottleneck برنامه می باشد.

* در قسمت API calls نیز بیشترین زمان صرف شده مربوط به allocate کردن حافظه می باشد که حدود 99.30٪ از کل زمان صرف شده توسط API calls می باشد.

* برای بهبود شرایط فوق می توان از تکنیک هایی برای کاهش دفعات انتقال داده از Host به Device و بلعکس استفاده نمود به طور مثال همزمان که داده ها در حال ارسال می باشند، GPU پردازش انجام دهد.
* می توان برنامه را به صورت طراحی کرد که از یک مرتبه kernel از سمت Host، lunch شود و در مراتب بعدی kernel از سمت Device، lunch شود.
* راه دیگر ذخیره سازی تصویر به نحوی است که چینش پیکسل های مجاور به نحوی باشد که دسترسی به آن ها با سرعت بالاتری انجام شود.

```
!nvprof ./sobel

==18540== NVPROF is profiling process 18540, command: ./sobel
GPU(parallel) time: 0.000116 s.
dimGrid.x 8 dimBlock.x 32
dimGrid.y 8 dimBlock.y 32
dimGrid.z 1 dimBlock.z 1
==18540== Profiling application: ./sobel
==18540== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   47.26%  62.720us         3  20.906us  1.6000us  31.616us  [CUDA memcpy HtoD]
                   31.20%  41.408us         1  41.408us  41.408us  41.408us  sobel_gpu(int*, int*, int*)
                   21.53%  28.576us         1  28.576us  28.576us  28.576us  [CUDA memcpy DtoH]
      API calls:   99.30%  200.32ms         3  66.775ms  3.0380us  200.32ms  cudaMalloc
                    0.27%  537.08us         1  537.08us  537.08us  537.08us  cuDeviceTotalMem
                    0.15%  304.28us         4  76.069us  41.686us  111.76us  cudaMemcpy
                    0.10%  197.08us        96  2.0520us     137ns  81.106us  cuDeviceGetAttribute
                    0.09%  188.51us         1  188.51us  188.51us  188.51us  cudaLaunchKernel
                    0.07%  144.20us         3  48.065us  3.9300us  124.96us  cudaFree
                    0.01%  25.874us         1  25.874us  25.874us  25.874us  cuDeviceGetName
                    0.00%  6.7930us         1  6.7930us  6.7930us  6.7930us  cuDeviceGetPCIBusId
                    0.00%  1.8770us         3     625ns     131ns  1.0250us  cuDeviceGetCount
                    0.00%  1.4530us         2     726ns     212ns  1.2410us  cuDeviceGet
```

```
GPU(parallel) time: 0.000079 s.
dimGrid.x 8 dimBlock.x 32
dimGrid.y 8 dimBlock.y 32
dimGrid.z 1 dimBlock.z 1

real    0m0.161s
user    0m0.009s
sys     0m0.146s
```

با تشکر از توجه شما