An Attempt of CUDA Implementation of PCA-SIFT

Masakazu IWAMURA†, Takayuki HONDO†, Kazuto NOGUCHI†, and Koichi KISE†

† Graduate School of Engineering, Osaka Prefecture University
1-1 Gakuencho, Naka, Sakai, 599-8531 Japan
E-mail: †{masa,kise}@cs.osakafu-u.ac.jp, {hondo,noguchi}@m.cs.osakafu-u.ac.jp

Abstract GPGPU (General-Purpose computation on GPUs) is a paradigm to use GPUs (graphics processing units) for general computation. Due to recent remarkable improvement of GPU, GPU outperforms CPU in computation ability. However, most people could not use the ability for general computation because existing programming languages require knowledge about GPU hardware architectures and computer graphics for GPGPU computing. Recently, a new GPU language CUDA (Compute Unified Device Architecture) has been released from NVIDIA. The CUDA code is C language style and has less computational restriction. Thus, usual operations of C language can run on GPU without much special knowledge. In this report, we briefly introduce CUDA language programming and report a CUDA implemented of PCA-SIFT. Compared to a CPU implementation, our CUDA implementation reduced the processing time to around 1/4. In addition, we also report an interesting phenomenon results useful for practical use of CUDA.

Key words PCA-SIFT, CUDA, GPU, GPGPU

1. Introduction

GPGPU (General-Purpose computation on GPUs) is a paradigm to use GPUs (graphics processing units) for general computation. Due to recent remarkable improvement of GPU, GPU outperforms CPU in computation ability, and has been evolving faster than CPUs (i.e., Moore's Law). However, most people could not use the ability for general computation because existing shading language such as Cg, HLSL and GLSL require knowledge about GPU hardware architecture and computer graphics. There are some GPGPU-oriented programming languages such as Brook [4], however, it is known to be very slow.

Recently, a new GPU language CUDA (Compute Unified Device Architecture) has been released from NVIDIA. Though CUDA runs only on some restricted GPUs, the CUDA code is C language style and has less computational restriction. Thus, usual operations of C language can run on GPU without much special knowledge.

In this report, we briefly introduce CUDA language programming and report a CUDA implementation of PCA-SIFT algorithm. The implemented PCA-SIFT was used for a demonstration on MIRU2007 [7]. In experiments, our CUDA implementation is compared to a CPU implementation. In addition, we report an interesting phenomenon useful for practical use of CUDA.

2. CUDA

2.1 Overview

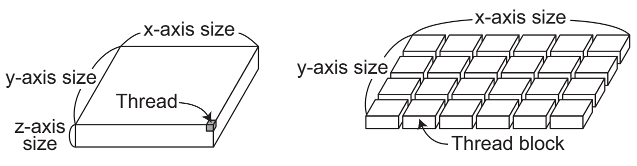
CUDA (Compute Unified Device Architecture) is a new[1] GPU programming language which allow us to program an algorithms executed on GPU in the C programming language. CUDA works only on relatively new NVIDIA graphics cards including GeForce 8000 series, a part of Quadro FX series and Tesla series.

A CUDA code is written in the standard C language[2] with some extensions related to GPU computation. It is compiled with the CUDA compiler *nvcc*, and can be linked with C++ code also. Algorithms executed on GPU have some limitations which include

- functions executed on GPU cannot call functions executed on CPU;
- recursive functions are not supported;
- static variables are not supported;
- double-precision floating-point numbers are not sup-

[1] The initial CUDA SDK was made public in February 2007 and the version 1.0 was released in June 2007.

[2] To be exact, it is noted in Sec. 4.2.5 of [2]: "The front end of the compiler processes CUDA source files according to C++ syntax rules. However, only the C subset of C++ is supported. This means that C++ specific features such as classes, inheritance, or declaration of variables within basic blocks are not supported."



(a) A thread block. (b) A grid of thread blocks.

Figure 1 (a) Threads form a thread block. The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512 and 64, respectively. The maximum number of threads per thread block is 512. Each thread in a thread block is identified by thread ID which represents the 3D coordinate in the block. The threads in a thread block shares fast *shared memory* mentioned in Sec. 2.3. (b) Thread blocks form a grid. The maximum size of each dimension of a grid of thread blocks is 65535. Each block is identified by block ID which represents the 2D coordinate of the block in the grid. All the threads in a grid execute a common function. In the current implementation, CUDA execute thread of one grid at the same time.

ported in the current devices (supported in future devices);

- single-precision floating-point arithmetic are deviated from the IEEE 754 standard.

2.2 Basic things

We explain some basic things required to read this report. For the sake of efficient calculation on GPU, many threads execute a function with different data in parallel. Thus each thread has IDs to identify data. As shown in Fig. 1(a), threads form a thread block. Each thread in a thread block is identified by thread ID which represents the 3D coordinate in the block. The threads in a thread block shares fast *shared memory* mentioned in Sec. 2.3. As shown in Fig. 1(b), thread blocks form a grid. Each block is identified by block ID which represents the 2D coordinate of the block in the grid. A grid is a unit to execute a function.

Then, we explain how to execute an algorithm on GPU with a simple CUDA code and the corresponding C code shown in Listing 1 and Listing 2. The program subtract a *width*×*height* gray-scale image *img2* from another gray-scale image *img1* of the same size for each pixel. For both codes, each gray-scale image is represented by a one-dimensional float array in the range of $[0, 1]$. In the C code, a double-for-loop executes the calculation for each pixel. To the contrary, the CUDA code has no "for loop". This is because "for loops" are expanded for parallelism.

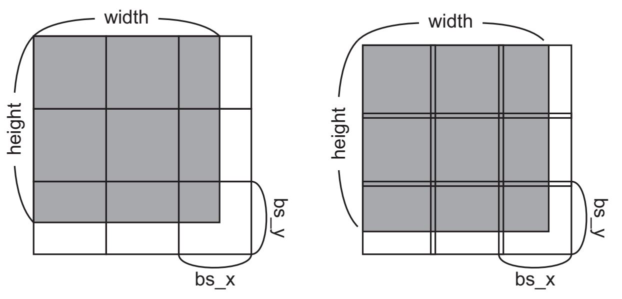
Before describing the for-loop expansion, we first explain the structure of the CUDA code. In the CUDA code, there are two functions: *sub_kernel* beginning at a line 6 is executed on GPU and *CUDA_sub* beginning at a line 26 is executed on CPU. The former executes the subtraction in actual, and the latter configures and calls the former. The configuration may contain GPU memory allocation because it cannot be

Listing 1 CUDA program to subtract two images.

```
 1  #include <math.h>
 2  #define BS_X 16 // x-axis thread block size
 3  #define BS_Y 16 // y-axis thread block size
 4
 5  // function on GPU; executed by each thread
 6  __global__ void
 7  sub_kernel(float *img1, const float *img2,
 8          const int width, const int height)
 9  {
10    int bx = blockIdx.x;   // Block index
11    int by = blockIdx.y;
12
13    int tx = threadIdx.x;  // Thread index
14    int ty = threadIdx.y;
15
16    int x = tx + BS_X * bx;  // The coordinate of the thread
17    int y = ty + BS_Y * by;
18
19    // Execute only inside the image
20    if (y<height && x<width) {
21      img1[y*width + x] -= img2[y*width + x];
22    }
23  }
24
25  // function processed on CPU
26  extern "C"
27  void CUDA_sub(float *img1, const float *img2,
28          const int width, const int height)
29   {
30     // x- and y-axis grid sizes
31     int blk_x = (int)ceil((float)width/BS_X);
32     int blk_y = (int)ceil((float)height/BS_Y);
33
34     // Execution configuration
35     dim3 threads(BS_X, BS_Y); // the size of a block
36     dim3 grid(blk_x, blk_y); // the size of the grid
37
38     // call the kernel
39     sub_kernel<<< grid, threads >>>(img1, img2 width, height);
40  }
```

Listing 2 C program to subtract two images.

```
 1  void sub(float *img1, const float *img2,
 2          const int width, const int height)
 3  {
 4    int x, y;
 5    for (y=0; y<height; y++) {
 6      for (x=0; x<width; x++) {
 7        img1[y*width + x] -= img2[y*width + x];
 8      }
 9    }
10  }
```



(a) Non-overlapping assignment. (b) Overlapping assignment.

Figure 2 Assignment example of threads to a *width*× *height* image. The image is covered by a grid containing 3 × 3 thread blocks without or with overlapping. Each thread block contains *bs_x*× *bs_y* threads. As the result, each thread is assigned to each pixel. Note that we ignore z-axis here. Thus, to be exact, *bs_x*× *bs_y*×1 threads per each block are used in both cases.

done in a GPU function. Thus the most important thing of the configuration is determination of sizes of a thread block and a grid. For example, in this case, an image is covered by thread blocks without overlapping as shown in Fig. 2(a). Thus, each thread is assigned to each pixel. Let us back to the for-loop expansion. Since each thread is assigned to each pixel, "for loop" is not needed any more.

Let us follow the function *sub_kernel* beginning at a line 6 in the CUDA code. At lines 10-14 of the CUDA code, each thread knows the coordinate as thread ID and block ID. Then, at lines 16-17, the position of the pixel where the thread is assigned is calculated. Namely, the coordinates in a thread block and in a grid are tied with the coordinate in an image. Finally, at lines 20-22, the subtraction is carried out. Note that as shown in Fig. 2(a), there exists some threads which are not assigned to any pixels. They should not be executed to avoid memory access violation unless enough memory is allocated for the images.

Finally, we explain briefly on the function *CUDA_sub* begging at a line 26. At lines 35-36, thread block and grid sizes are set. The calculation at lines 31-32 determine the sizes. BS_X and BS_Y are predefined thread block sizes in x- and y-axes, which correspond to *bs_x* and *bs_y* in Fig. 2(a). Finally, at a line 39, the GPU function *sub_kernel* is called. Two variables after <<< are the thread block and grid sizes, and variables in the parentheses after >>> are arguments of the GPU function.

2.3 Example with shared memory

As mentioned in Sec. 2.2, the threads in a thread block shares faster *shared memory* than global memory or texture memory [2]. Using shared memory is very important to reduce processing time. Thus, we show a simple example of CUDA code and the corresponding C code in Listing 3 and Listing 4.[3] The program differentiates an image along x- and y-axes. The big difference in the configuration precess in Listing 3 from Listing 1 is that BS_X and BS_Y are replaced by (BS_X − 2) and (BS_Y − 2) as shown in Fig. 2(b). This overlapping enables us to differentiate an image efficiently. We omit the detail.

3. Implementation

3.1 PCA-SIFT

PCA-SIFT [5] is an improvement of SIFT [6]. SIFT (Scale-invariant feature transform) is one of the most popular feature extraction algorithms in computer vision. SIFT descriptors have some good properties including scale and rotation invariance, robustness against change of viewpoints and that in illumination. The SIFT algorithm can be separated into two stages: (a) calculation of keypoints, and (b) calculation of SIFT descriptors. At the stage (a), feature points (keypoints) stable and robust to change of view angles and noises. At the stage (b), 128-dimensional SIFT descriptors are calculated for the keypoints. PCA-SIFT replaces the stage (b). Instead of 128-dimensional SIFT descriptors, PCA-SIFT calculates 36-dimensional PCA-SIFT descriptors.

Extracting SIFT and PCA-SIFT descriptors are very time consuming. It takes a few seconds for a VGA size image. Thus GPU implementation of SIFT exists [3], [8]. However, that of PCA-SIFT does not exist. Therefore, we implemented PCA-SIFT on CUDA.

3.2 Details

Since translating into a CUDA code from a C code is relatively easier than other languages, we prepare an original code of PCA-SIFT written on C and C++ languages. The original source code of PCA-SIFT was downloaded from `http://www.cs.cmu.edu/~yke/pcasift/`. Since the source code required SIFT descriptors, we downloaded a SIFT implementation from `http://web.engr.oregonstate.edu/~hess/` and merged them.

The overview of our implementation is shown in Fig. 3. The overall process is as follows.

(1) An image to be processed is loaded and transferred from the host to the GPU. We did not use texture memory but global memory.

(2) A Gaussian scale-space pyramid is created on the GPU. As the Gaussian convolution, we used a CUDA code sample "convolution." [4] Using the Gaussian pyramid,

[3] Note that this example is too simple to reduce processing time because the number of memory access is small.

[4] The code sample was downloaded from `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html`. Now, "convolu-

Listing 3 CUDA program to differentiate an image

```
#include <math.h>
#define BS_X 16 // x−axis thread block size
#define BS_Y 16 // y−axis thread block size

// function on GPU; executed by each thread
__global__ void
DXDY_kernel(float *dx, float *dy, const float *img,
        const int width, const int height)
{
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // The coordinate of the thread
  int x = tx + (BLOCK_SIZE_X−2) * bx;
  int y = ty + (BLOCK_SIZE_Y−2) * by;

  // Declare a variable on shared memory
  __shared__ float img_sh[BS_X][BS_Y];

  // copy to shared memory
  if (y<height && x<width) {
    img_sh[tx][ty] = img_in[y*width + x];
  }

  // Synchronize to make sure the image is copied
  __syncthreads();

  // Execute only inside the image
  // if it's not overlapped position of the thread block
  if (y>=1 && y<height−1 && x>=1 && x<width−1 &&
      tx!=0 && ty!=0 && tx!=BS_X−1 && ty!=BS_Y−1) {
    dx[y*width + x] = img_sh[tx+1][ty] − img_sh[tx−1][ty];
    dy[y*width + x] = img_sh[tx][ty+1] − img_sh[tx][ty−1];
  }
}


// function processed on CPU
extern "C"
void CUDA_DXDY(float *dx, float *dy, const float *img,
        const int width, const int height)
 {
   // x− and y−axis grid sizes
   int blk_x = (int)ceil((float)width/(BS_X−2));
   int blk_y = (int)ceil((float)height/(BS_Y−2));

   // Execution configuration
   dim3 threads(BS_X, BS_Y); // the size of a block
   dim3 grid(blk_x, blk_y); // the size of the grid

   // call the kernel
   DXDY_kernel<<< grid, threads >>>
        (dx, dy, img, width, height);
}
```

Listing 4 C program to differentiate an image

```
void DXDY(float *dx, float *dy, const float *img,
        const int width, const int height)
 {
  int x, y;
  for (y=1; y<height−1; y++) {
    for (x=1; x<width−1; x++) {
      dx[y*width + x] =
            img[y*width + x+1] − img[y*width + x−1];
      dy[y*width + x] =
            img[(y+1)*width + x] − img[(y−1)*width + x];
    }
  }
}
```

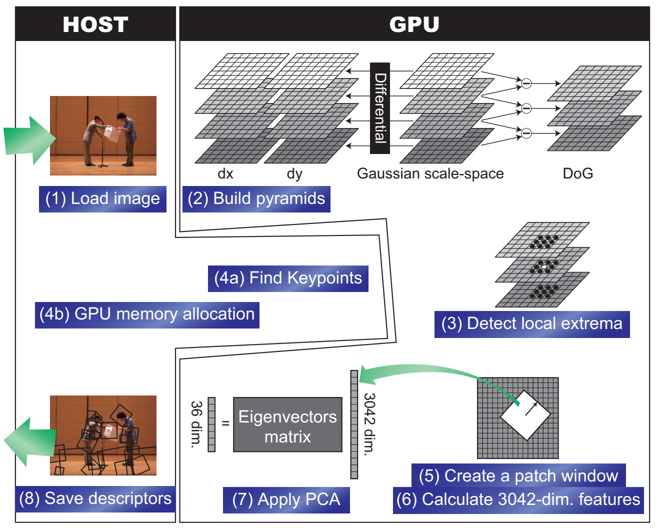


Figure 3 Overview of our CUDA implementation of PCA-SIFT.

Difference-of-Gaussian (DoG) images, the image gradients along x- and y-axes for each pixels are calculated.

(3) Local extrema are detected in parallel on GPU. Definition of the local extremum is that the pixel value of a sample point of DoG pyramid is larger or smaller than those of 26 neighbors. 26 neighbors include eight neighbors in the same scale and nine neighbors in both adjacent scales. Local extrema are candidates of keypoints. After the detection, candidate locations and scales are recalculated at the sub-pixel level, and inaccurate candidates are removed.

(4) Keypoints are found on CPU.[5] According to the number of keypoints, GPU memory is allocated.

(5) For each keypoint, a patch window in a Gaussian scale-space is created according to the position, scale and orientation of the keypoint.

(6) For each keypoint, a 3042-dimensional feature vector

tion” is not available because it was replaced by “convolutionSeparable” when CUDA SDK version 0.9 was released.

[5] In the current implementation, keypoints are found on CPU. However, it can be on GPU in future.

is calculated by differentiating the window patch.

(7) For each keypoint, a 36-dimensional PCA-SIFT descriptor is acquired by applying PCA. For the multiplication of a matrix and a vector, CUDA BLAS (CUBLAS) [1], where is an implementation of BLAS (Basic Linear Algebra Subprograms) on CUDA, is used. The eigenvectors matrix included in the source package of PCA-SIFT was used.

(8) PCA-SIFT descriptors are transferred from the GPU to the host, and saved.

4. Experiments

We performed two experiments: (1) comparison of processing time of PCA-SIFT on CPU and GPUs, (2) the overhead of calling a GPU function.

All the experiments were carried out on an Athlon 64 X2 6000+ machine with 4GB memory. For GPU, three GPUs were examined: GeForce 8800GTX, 8800GTS and 8600GTS. Each of them has 16, 12 and 4 multiprocessors, respectively. A multiprocessor contains eight processors. The processors of a multiprocessor execute an instruction simultaneously.

4.1 Processing time of PCA-SIFT on CPU and GPUs

In the first experiment, we compared processing times of CPU and GPU implementations of PCA-SIFT for evaluating the effect of parallel processing. For the evaluation, we divide the processes described in Sec. 3.2 and Fig. 3 into three parts: (2), (3)–(4) and (5)–(7). Each of them corresponds to building pyramids, calculation of keypoints, and calculation of PCA-SIFT descriptors, respectively. The processing times of the whole process and the three partial processes are shown in Table 1. In the experiments, 64 images were used. Average image size was 514.9 × 438.8 pixels. 1411.1 keypoints were found on GPU and 1398.8 on CPU in average. Although processing time highly depends on the number of detected keypoints, the numbers are almost same. The reason that they were not exactly same was not investigated. The data transfer rate between the host and the GPU was around 2MB per millisecond for both directions. The experimental result shows that our GPU implementation achieved around 2/5, 8/7 and 1/10 of CPU processing time for each partial process in the case of GeForce 8800GTX. In total, it reduced the processing time to around 1/4.

We consider the reasons of the bad performance. Firstly, the reason of little reduction in processing time for (2) seems that the degree of parallelism is low. In order to obtain better performance on CUDA, executing as many threads as possible simultaneously is better. This means that creating images of the Gaussian pyramid simultaneously as many as possible achieves better performance. However, in the current implementation, only one image is processed simultaneously. This can be improved relatively easier. Secondly, the reason of increase in processing time for (3)–(4) seems that (i) sparsity of local extrema and (ii) calling a GPU function many times. For (i), the number of local extrema is essentially very few for the number of pixels. Since threads execute a function, one-on-one assignment between a pixel and a thread can increase waiting threads and reduce efficiency. Thus, finding a balanced point of the assignment, e.g., four pixels per thread, seems to be required. For (ii), this is the same problem to the low degree of parallelism discussed above. The problem is also related the overhead problem discussed in Sec. 4.2.

Table 1 Average processing times of PCA-SIFT executed on different devices are shown. The top row represents the name of device, and the number in the parentheses represents the number of multiprocessors of the GPU. The numbers in the rightmost column correspond to the process numbers described in Sec. 3.2 and Fig. 3. Note that calculation of the image gradients (dx and dy) was contained in the process (2) on GPU and the processes (3)–(4) on GPU.

	GeForce 8800GTX (16)	GeForce 8800GTS (12)	GeForce 8600GTS (4)	CPU
(2)	90.27	107.7	169.4	224.0
(3)–(4)	252.7	283.7	414.1	222.1
(5)–(7)	209.3	218.4	299.7	1932
Total	572.2	628.7	899.0	2383

Note that the total time does not match the sum of processing time of the three partial processes because only the total time contains the processing time of some processes such as releasing memory.

4.2 Overhead of calling a GPU function

As mentioned above, there exists the overhead of calling a GPU function. We found an interesting phenomenon on the overhead. It was found when we examined the computational ability of a processor on GPU because if the ability of a processor is high, almost all computation can be executed on GPU without switching to CPU. However, the calculation on GPU was unreasonably low regardless to the computational task. The reason was the overhead of calling a GPU function.

Thus, we investigated the overhead. The result is shown in Fig. 4. In the figure, we can confirm that there exists relatively small overheads and large overheads for the number of threads in a thread block in GeForce 8800GTX and GeForce 8800GTS. However, it did not appear in GeForce 8600GTS. This seems to be caused by the number of multiprocessors. Though Fig. 4 shows only when the thread block size was $x \times 1$, we confirmed that the result did not change when the thread block size was $x/2 \times 2$. Similarly, we confirmed the phenomenon appeared the grid size was 128×128. However, the phenomenon disappeared when the number of thread block in the grid was less than about 100.

Though we do not know the reason, we think this information is useful to execute a relatively large grid. For example, we obtained 10 milliseconds gain by just changing a thread block size from 16 × 16 to 16 × 12 with GeForce 8800GTX.

5. Conclusions

In this report, we introduced CUDA, a newer GPU programming language, and an implementation of PCA-SIFT on CUDA. The CUDA code is C language style and has less computational restriction. Thus, usual operations of C language can run on GPU without much special knowledge.

In the experiments, our CUDA implementation reduced the processing time to around 1/4 compared to a CPU implementation. In addition, we experimentally examined an interesting phenomenon useful for practical use of CUDA.

Future work includes performance improvement of the implementation in two points: (1) creating images of the Gaussian pyramid simultaneously as many as possible, and (2) finding keypoints, numbered (4a) in Fig. 3, on GPU.

Acknowledgment

This research was supported in part by the Grant-in-Aid for Scientific Research (B) (19300062) from Japan Society for Promotion of Science.

References

[1] *CUDA CUBLAS Library Version 1.0*, June 2007. Downloadable at `http://developer.nvidia.com/object/cuda.html`.

[2] *CUDA Programming Guide Version 1.0*, June 2007. Downloadable at `http://developer.nvidia.com/object/cuda.html`.

[3] `http://cs.unc.edu/~ccwu/siftgpu/`.

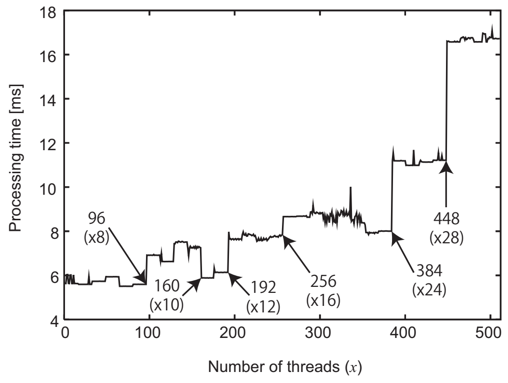
[4] `http://graphics.stanford.edu/projects/brookgpu/`.

[5] Yan Ke and Rahul Sukthankar. PCA-SIFT: A more distinctive representation for local image descriptors. In *Proc. CVPR'04*, volume 2, pages 506–513, 2004.

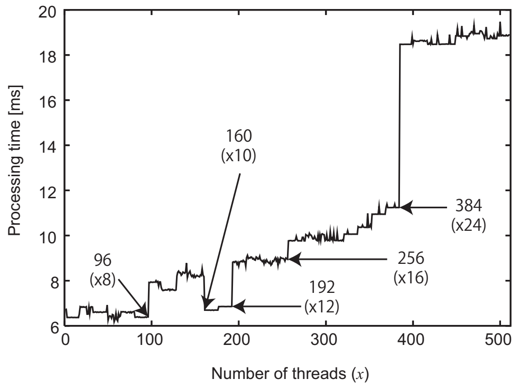
[6] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Proc. ICCV'04*, 60(2):91–110, 2004.

[7] Kazuto Noguchi, Takayuki Hondo, Masakazu Iwamura, and Koichi Kise. Real-time recognition of objectsby cascading approximate nearest neighbor searchers. *Proceedings of MIRU 2007*, pages 467–468, July 2007.

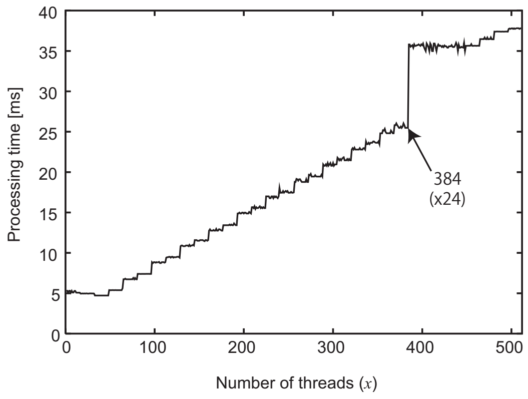
[8] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Gpu-based video feature tracking and matching. Technical Report TR 06-012, Department of Computer Science, UNC Chapel Hill, May 2006.



(a) GeForce 8800GTX.



(b) GeForce 8800GTS.



(c) GeForce 8600GTS.

Figure 4 The overhead of calling an empty GPU function. The thread block size was $x \times 1$ and the grid size was 1024×1024. The results show that the overhead was changed in every 16 increase of x. In the parentheses, $x/16$ is shown as, e.g., $\times 8$ and $\times 10$. There exists relatively small overheads and large overheads for the number of threads in a thread block in GeForce 8800GTX and 8800GTS. $\times 10$, $\times 12$ and $\times 24$ had relatively small overhead. However, it did not appear in 8600GTS. This seems to be caused by the number of multiprocessors.