Course on Hybrid Computing for Signal and Data Processing - Student Project

Parallelization of Image Transformation and Sampling

E4750_2016Fall_PITS_report

Hung Shi Lin hl2997@columbia.edu, Duoying Zhou dz2337@columbia.edu
Columbia University

Abstract

Spatial Transformer Network(STN)[1] is a recently developed Convolutional Neural Network (CNN), which integrated image preprocessing into CNN. It successfully improved the classification accuracy. However, the transformation and sampling in the STN significantly reduce the computation speed, which makes it unscalable to fit the real world data. In the project, we focus on parallelizing transformation and sampling in order to speed up the forward propagation in CNN. The numerical result shows that our parallelized code is much faster than sequential one.

1 Problem in a Nutshell

STN[1] has gained prominence in recent years because of its computation power to improve the classification accuracy. STN applies two preprocessing techniques before feeding the image into CNN, which are transformation and sampling. Transformation takes as input an image, and applies techniques such as affine, projection or thin plate spline to extract the interest region of the image for the following feature extraction task. Since the transformed coordinates are not limited to the integer, STN utilizes bilinear sampling to assign values to the transformed pixels. Because these two steps involve calculating each pixel in the image, the computation speed would benefit if we can parallelize them. In this project, we are going to try different parallelization to make the computation faster than its sequential counterpart. Since different transformation and sampling would require different optimization scheme, we take affine transformation and bilinear sampling as our primary targets in tis project. In particular, because affine transformation involves rotation, scaling, shearing, we assume that the affine transformation is actually scaling and translating the image when we do sampling.

2. Problem Statement and Software Design

In this section, we are going to specify the problem this project is going to address, the math that underlies the problem and the optimization techniques we design to cope with the problem.

2.1. Problem Statement of Affine Transformation and Optimization

A spatial transformation of an image is a geometric transformation of the image coordinate system. In this section, affine transformation is applied to images, creating the mapping between pixel coordinate system of the target image and that of the source image. Since the affine transformation of an image can be represented as the matrix-matrix multiplication, the parallelization of affine transformation can reduce to the parallelization of matrix multiplication. Therefore, the core of optimizing affine transformation is to optimize matrix multiplication.

2.2. Description of Optimization for Affine Transformation

In the affine transformation, the pointwise transformation is

$$\begin{pmatrix} x_i^s \\ y_i^s \end{pmatrix} = A_\theta(G_i) = A_\theta \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix} \quad (1)$$

where $G_i = (x_i^t, y_i^t)$ are the target coordinates of a pixel in the output feature map, (x_i^s, y_i^s) are the source coordinates in the input feature map, and $A_\theta = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix}$ is the affine transformation matrix. And the affine transformation is a composition of translation, scaling, shearing and rotation where scaling is affected by θ_{11} and θ_{22}, translation is affected by θ_{13} and θ_{23}, and shearing and rotation are affected by the combination of thetas.

Then, for an image with multiple pixels, the affine transformation can be considered as the matrix-matrix multiplication, with the target coordinates of multiple pixels forming the input matrix in some way.
We can assume that the shape of the image is (M, N). Then, the image transformation can be written as

$$\begin{bmatrix} x_0^s & x_1^s & \dots & x_{M*N-1}^s \\ y_0^s & y_1^s & \dots & y_{M*N-1}^s \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{bmatrix} x_0^t & x_1^t & \cdots & x_{M*N-1}^t \\ y_0^t & y_1^t & \cdots & y_{M*N-1}^t \\ 1 & 1 & \cdots & 1 \end{bmatrix} \tag{2}$$

Or

$$\begin{bmatrix} s_{vector_{00}} & \cdots & s_{vector_{N-1,0}} \\ \vdots & \ddots & \vdots \\ s_{vector_{0,M-1}} & \cdots & s_{vector_{N-1,M-1}} \end{bmatrix} = \begin{bmatrix} affine & 0 & \cdots & 0 \\ 0 & affine & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & affine \end{bmatrix} \begin{bmatrix} t_{vector_{00}} & \cdots & t_{vector_{N-1,0}} \\ \vdots & \ddots & \vdots \\ t_{vector_{0,M-1}} & \cdots & t_{vector_{N-1,M-1}} \end{bmatrix} \tag{3}$$

where $s_{vector_{ij}} = \begin{pmatrix} x_i^s \\ y_j^s \end{pmatrix}$, $t_{vector_{ij}} = \begin{pmatrix} x_i^t \\ y_j^t \\ 1 \end{pmatrix}$,

$$affine = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix}$$

To accelerate the affine transformation, we just need to focus on accelerating the matrix multiplication by parallelization and explore the optimization of parallel computing.

2.3. Software Design of Optimized Affine Transformation

We used three parallel matrix multiplication algorithms and compared the time taken by them.

Naïve Matrix Multiplication algorithm

The kernel of parallel naïve matrix multiplication algorithm is similar to the sequential code. It just directly describes the process that one element of the product is generated by the inner product of a row of the first input matrix and a column with the second input matrix.

Pseudocode(Pyopencl):

```
int i, j, k
i=get_global_id(0)
j=get_global_id(1)
for ( k=0; k<N; k++){
    C(i, j)=sum (over k) A(i, k) * B(k, j)
}
```

Tiled Matrix Multiplication algorithm

The tiled matrix multiplication algorithm breaks up the execution of the kernel into phases so that the data accesses in each phases are focused on one tile of the first matrix and one tile of the second matrix. In the Pycuda case, the tiles of input matrices are loaded into shared memory to make the main computation running in the shared memory. And the accesses within each warp are coalesced into a DRAM burst.

Pseudocode(Pyopencl):

```
__local float ds_A[TILE_WIDTH][TILE_WIDTH]
__local float ds_B[TILE_WIDTH][TILE_WIDTH]
int bx=get_group_id(1); int by=get_group_id(0)
int tx=get_local_id(1); int ty=get_local_id(0)
int Row=get_global_id(1)
int Col=get_global_id(0)
float Cvalue=0
for ( int t=0; t<(n-1)/TILE_WIDTH+1; t++)
     if (Row <m && t*TILE_WIDTH+tx <n)
          ds_A[ty][tx]=A[Row*n+t*TILE_WIDTH+tx]
    else ds_A[ty][tx]=0
    if (t*TILE_WIDTH+ty<n && Col<k)
          ds_B[ty][tx]=B[(t*TILE_WIDTH+ty)*k+Col]
    else ds_B[ty][tx]=0
barrier(CLK_LOCAL_MEM_FENCE)
for (int i=0; i< TILE_WIDTH; i++)
    Cvalue+=ds_A[ty][i]*ds_B[i][tx]
barrier(CLK_LOCAL_MEM_FENCE)
If (Row<m && Col<k)
    C[Row*k+Col]=Cvalue
```

We also tried to implement tiled matrix multiplication on the input matrix with different shapes according to formula (2) and (3) to see if there is any difference.

Scalable Universal Matrix Multiplication Algorithm (SUMMA)[2]

In SUMMA, formula (3) is used to make multiplication. The first input matrix A, the second input matrix B and their matrix product C are all divided into M*N tiles. That is, the grid dimension of A, B and C are the same, which is the shape of the image. And the block dimension is set as 2*1 because the result matrix consists of the source coordinates $\begin{pmatrix} x_i^s \\ y_i^s \end{pmatrix}$, which is a 2*1 vector. Then as the figure 1 shows, we still break up the execution of the kernel into phases so that the data accesses in each phases are focused on one tile of the first matrix and one tile of the second matrix. However, at this time, we do not need to put any tile into the shared memory.

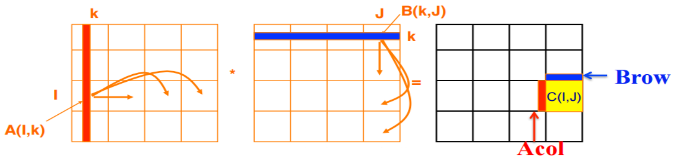

Fig. 1 SUMMA diagram

Pseudocode:

```
int y=get_global_id(0);
int x=get_global_id(1);
float temp=0.0f;
int ty=get_local_id(0); int tx=get_local_id(1);
for (int k=0;k<%(L)d;k+=3)
    float Asub[2][3];
    for (int i=0;i<2;i++)
        for (int j=0;j<3;j++)

Asub[i][j]=A[(get_group_id(0)*2+i)*%(L)d+k+j];
    float Bsub[3][1];
    for (int i=0;i<3;i++)
        for (int j=0;j<1;j++)

Bsub[i][j]=B[(k+i)*%(N)d+get_group_id(1)*1+j];
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int m = 0; m < 3; m++)
            temp+= Asub[ty][m]*Bsub[m][tx];
C[y*%(N)d+x]=temp;
```

2.4. Problem Statement of Bilinear Sampling and Optimization

Bilinear Sampling considers the neighborhood of the transformed pixel coordinate to assign value to the transformed image. Since there are several different affine transformations such as rotation, scaling, shearing and translation, and each of them requires different optimization, we focus on optimizing scaling and translation in this project.

2.5. Description of Optimization for Bilinear Sampling

In this section, we describe three different methods to parallelize bilinear sampling, which are naïve, optimized1 and optimized2. Bilinear sampling equation is defined as

$$V_i^c = \sum_n^H \sum_m^W U_{nm}^c \delta(\lfloor x_i^s + 0.5 \rfloor - m)\delta(\lfloor y_i^s + 0.5 \rfloor - n)$$

Where $(x_i^s, y_i^s$ is the transformed coordinate, $U_{nm}^c$is the value of the original image at location (n, m) and , V_i^c is the pixel value of the image V at location i.

From the formula we can know that bilinear sampling just samples data from the neighborhood of the transformed coordinate, so instead of writing the formula above, we can rewrite it as

$$V_i^c = U_{floor(x_i^s)floor(y_i^s)}^c (ceil(x_i^s) - x_i^s) x (ceil(y_i^s) - y_i^s) + U_{floor(x_i^s)ceil(y_i^s)}^c (ceil(x_i^s) - x_i^s) x (y_i^s - ceil(y_i^s)) + U_{ceil(x_i^s)floor(y_i^s)}^c (x_i^s - floor(x_i^s)) x (ceil(y_i^s) - y_i^s) + U_{ceil(x_i^s)eil(y_i^s)}^c (x_i^s - floor(x_i^s)) x (ceil(y_i^s) - y_i^s) \quad (4)$$

We actually only need four terms and sum up to get the value of the pixel. The basic idea of parallelism of bilinear sampling is therefore to assign one thread to one pixel. However, if we can first cache four boundary values into local or private memory, we can reduce the communication between thread and global memory. The question is, because each pixel might has different boundary values, which values should be cached to the local/private memory? In this project, our optimization is focused on scaling and translation, because their properties ensure that the points that lie on the same line will also be collinear after transformation. As a result, we can cache a column to the local/private memory and each thread samples all the values of that column. In the next section, we show the pseudo codes of one naïve parallelization and two optimized versions.

2.6. Software Design of Optimized Affine Transformation

In this section, we demonstrate the ideas of three parallelization methods and their pseudo code, respectively.

First Parallelization: One Thread One Pixel

This is the naïve version of parallelization where each thread samples one pixel. The global memory access is N, four reads and one write. N is the total number of pixels

Pseudo code

Input: img(original image, 2D array), x and y(transformed coordinates, 2D array), N(number of pixels)
Output: transformed image (2D array)

```
__kernel void BasicBilinearSampling(__global float *original_img, __global float *imgloc, __global float *trans_img, const unsigned int img_w) {
        unsigned int idx = get_global_id(0);
        int m[2];
        int n[2];
        float x_loc = imgloc[idx*2];
        int x = x_loc;
        float y_loc = imgloc[idx*2+1];
        int y = y_loc;
        m[0] = x;
        m[1] = x_loc - x != 0 ? (x + 1):x;
        n[0] = y;
        n[1] = y_loc - y != 0 ? (y + 1):y;
        float c = 0;
        float x_flag = 1.0;
        float y_flag;
        int x_end = m[0] == m[1] ? 1:2;
```

```
    int y_end = n[0] == n[1] ? 1:2;
    for (int i = 0; i < x_end; i++) {
        y_flag = 1.0;
        for (int j = 0; j < y_end; j++) {
            float tmp1 = original_img[n[j]*img_w + m[i]];
            c += original_img[n[j] * img_w + m[i]] * (1 - x_flag * (x_loc - m[i])) * (1 - y_flag * (y_loc - n[j]));
            y_flag = -1.0;
        }
        x_flag = -1.0;
    }
    trans_img[idx] = c;

}
```

First Parallelization: Assign a thread to a column

Because of the property of scaling and translation, we know that points that are on the same line are collinear after transformation. Therefore, we can first cache the boundaries of a column in x direction to the private memory so that we can reduce data transfer times from global memory. The total global memory access is therefore reduced to 2cols + 3N, where we have 2cols for x boundaries reads, 2N for y boundaries reads, and N write.

Pseudo code

Input: img(original image, 2D array), x and y(transformed coordinates, 2D array), N(number of pixels)
Output: transformed image (2D array)

```
__kernel void scalingBilinearSampling(
                    __global float *img,
                    __global float *img_loc,
                    __global float *trans_img
                    ) {

    unsigned int idx = get_global_id(0);
    float U[FETCH_HEIGHT][2];
    float x_loc = img_loc[2*idx];
    float y_loc;
    float xu_diff;
    float xl_diff;
    float yl_diff;
    float yu_diff;
    int k;
    int x_lbound = (int)x_loc;
    int x_ubound = (int)x_loc + 1;
    int y_lbound;
    xl_diff = 1 - (x_loc - x_lbound);
    xu_diff = 1 - xl_diff;

    for (k = 0; k < FETCH_HEIGHT; k++) {
        U[k][0] = img[FETCH_WIDTH * k + x_lbound];
        U[k][1] = img[FETCH_WIDTH * k + x_ubound];
    }

    for (k = 0; k < IMG_HEIGHT; k++) {
        y_loc = img_loc[2*(idx+k*IMG_WIDTH)+1];
        y_lbound = (int)y_loc;
        yl_diff = 1 - (y_loc - y_lbound);
        yu_diff = 1 - yl_diff;
        trans_img[k*IMG_WIDTH+idx] = (U[y_lbound][0] * xl_diff + U[y_lbound][1] * xu_diff) * yl_diff + (U[y_lbound+1][0] * xl_diff + U[y_lbound+1][1] * xu_diff) * yu_diff;

    }
}
```

Third Parallelization: Assign a thread to a column

In the second method, we assign a thread to a column because we know that all values in one column have the same x boundaries. Now we extend this idea that some column would have the same x boundaries and y boundaries. The logic is that when we scale down the image, there are some columns where the x locations have the same floor(x) and ceil(x). Therefore, we can separate an images into cols (number of columns) blocks, then threads in the block read all boundaries into local memory (One thread might read more than one boundary value). Finally, each thread sample all the values of the column (Again, one thread might sample more than one column, which is different from the second method). The following figure shows the idea of the method. The red spots are the transformed coordinates. We can see that they all have the same x boundaries and two of them have the same y boundaries. The difference between 2nd and 3rd is that we assign a thread to a column and cache the boundaries to the private memory while we assign a thread to several columns and cache all boundaries to local memory.

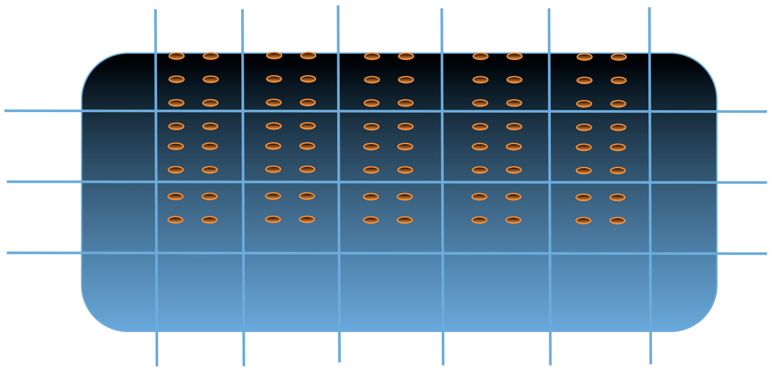

Fig. 2 The idea of the Third Optimization for Scaling

Pseudo code
Input: img(original image, 2D array), x and y(transformed coordinates, 2D array), N(number of pixels)
Output: transformed image (2D array)

```
__kernel void scalingBilinearSampling(
                        __global float *img,
                        __global float *img_loc,
                        __global float *trans_img,
                        __global int* bins
                        ) {
                unsigned int ldx = get_local_id(0);
                unsigned        int        tile_size        =
get_local_size(0);
                unsigned int gid = get_group_id(0);
                __local float U[FETCH_HEIGHT][2];
                float y_loc;
                float xu_diff;
                float xl_diff;
                float yl_diff;
                float yu_diff;
                int k;
                int x_lbound = gid;
                int x_ubound = x_lbound + 1;
                int y_lbound;
                int gdx;
                int tmp = ldx;
                while (tmp < FETCH_HEIGHT) {
                        U[tmp][0]                        =
img[FETCH_WIDTH * tmp + x_lbound];
                        U[tmp][1]                        =
img[FETCH_WIDTH * tmp + x_ubound];
                        tmp += tile_size;
                }
                barrier(CLK_LOCAL_MEM_FENCE);
                tmp = ldx;
                float x_loc;
                int basics = 0;
                for (unsigned int i = 0; i < gid;i++) {
                        basics = basics + bins[i];

                }

                while (tmp < bins[gid]) {

                        gdx = tmp + basics;
                        x_loc = img_loc[2*gdx];
                        xl_diff    =    1    -    (x_loc    -
x_lbound);
                        xu_diff = 1 - xl_diff;
                        for (k = 0; k < IMG_HEIGHT;
k++) {
                                y_loc                =
img_loc[2*(gdx+k*IMG_WIDTH)+1];
                                y_lbound = (int)y_loc;
                                yl_diff =  1 - (y_loc -
y_lbound);
                                yu_diff = 1 - yl_diff;

        trans_img[k*IMG_WIDTH+gdx]                    =
(U[y_lbound][0] * xl_diff + U[y_lbound][1] * xu_diff) *
yl_diff + (U[y_lbound+1][0] * xl_diff + U[y_lbound+1][1]
* xu_diff) * yu_diff;

                        }
                        tmp += tile_size;
                }
}
```

3. Numerical Example

In this Section, we use numerical examples to demonstrate the results and time comparison of different optimization techniques for affine transformation and bilinear sample.

3.1. Affine Transformation

For affine transformation optimization, we separately implement naïve matrix multiplication, tiled algorithm and scalable universal matrix multiplication algorithm(SUMMA), with changing the block size and the shape of the image coordinate matrix, to observe the speed. The time taken by all above-mentioned algorithms is shown in Fig. 3 and Fig. 4.

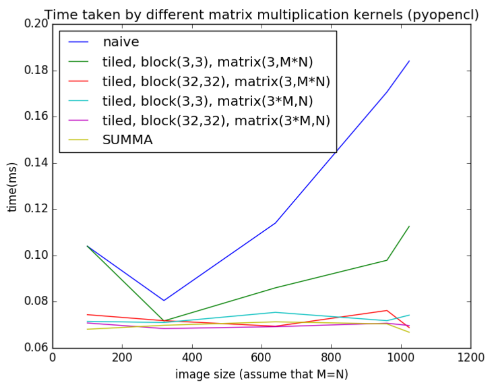

Fig. 3 Time taken by different kernels (pyopencl)

According to Fig. 3, we can compare the performance of the above-mentioned optimization methods. First, compared with the naïve algorithm, other algorithms all effectively optimized the matrix multiplication since they utilize the coalesced access or local memory or the better block size. Then, for tiled matrix multiplication, when the block size is set as (3,3), the speed is relatively slower than that of other optimization methods. As for other optimized kernels, their running time is really similar to each other and keeps stable when the image size increases. When the image size is 1024*1024, the speedup of tiled matrix multiplication with block size (3,3) to the naive algorithm is about 1.64, while the speedup of other methods to the naive algorithm is about 2.57.

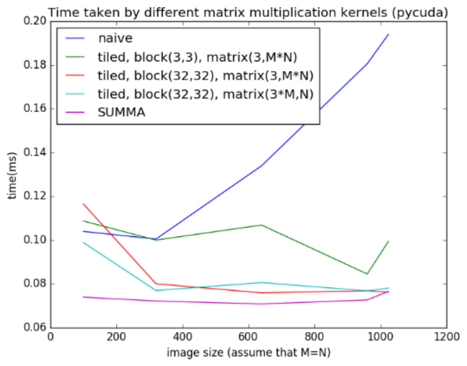

Fig. 4 Time taken by different kernels (pycuda)

The comparison of each optimization methods in pycuda presents the similar result to that in pyopencl. However, when the image size is 1024*1024, the speedup of tiled matrix multiplication with block size (3,3) to the naive algorithm is about 1.95, while the speedup of other methods to the naive algorithm is about 2.62.

And in total, we see that kernels in pyopencl generally have better performance than that in pycuda.

3.2. Bilinear Sampling

For bilinear sampling we assume that the scaling and translation are applied to the image. Two different image sizes with 1024x1024 and 8054x8054 are used to understand how different image sizes and threads affect the performance. In this experiment, the number of threads only has impact on optimized 2. Scaling ratio is 0.3 and the translation coefficient is 200. The original images and the transformed images are shown below:

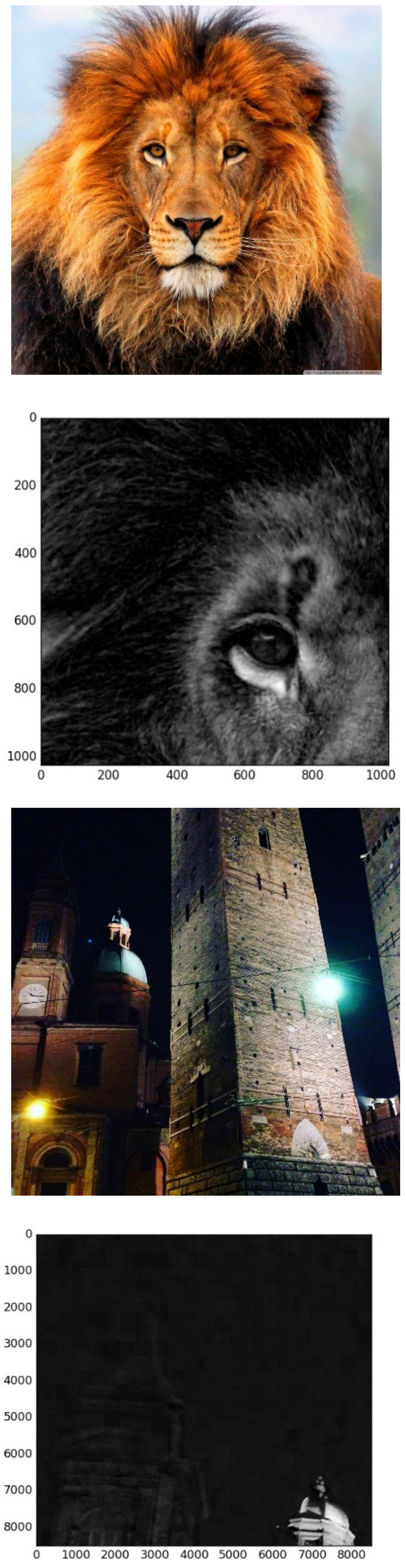

Fig. 5 Original Image and the Image after Transformation and Sampling

The time comparison of different optimization methods and different Implementations (pyopencl and pycuda) for first image (1024x1024) is shown in Fig. 6. It shows that Naïve version is worse than Optimized1 and Optimized2. However, Optimized 1 is better than Optimized 2, which is different from what we expected. Our analysis is that there are some threads idled duringsampling and therefore degrade the performance. Cuda implementation is always unexpectedly slow. We haven't any idea why this happens, but we think this is due to the internal hardware implementation.

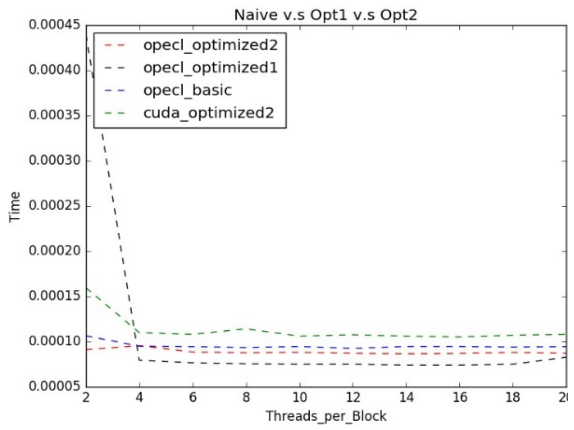

Fig. 6 Time Comparison of Different Techniques and Implementations for Image Size 1024x1024

Fig. 7 shows the time comparison for image size 8054x8054. The result is different from the first case where the Optimized 2 implemented in opencl is the fastest. We think this is due to the number of pixels between boundaries becomes large so that less threads idled during sampling. Optimized 1 becomes the slowest in this case, which we think is that because the image size is too huge to fit in private memory, so the threads have to communicate with global memory, which degrade the computation speed.

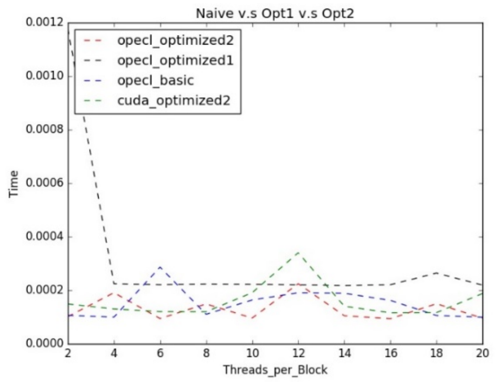

Fig. 7 Time Comparison of Different Techniques and Implementations for Image Size 8054x8054

To summarize, we demonstrate different parallelization for affine transformation and bilinear sampling with scaling and translation in this section. The time comparison of these methods are shown to understand the effect of different factors on the performance.

4. Future Works

Firstly, our future works will focus on the further optimization on affine transformation and bilinear sampling. Even though we improve the performance to some extent, the kernel may be sped up more by tuning the block sizes or shapes and applying other optimization approaches. In addition, we will implement other image transformation methods, such as projection, thin plate spline, etc. to observe the performance. Also the image transformation would be done with experimenting with translation, rotation, shearing and scaling. Furthermore, we will explore different algorithms for image sampling and interpolation and add some filters, both producing better target images and guaranteeing the running performance. Finally, other optimization for bilinear sampling under a more general affine transformation can be further studied to design a strategy for optimizing the parallelism.

5. Acknowledgements

This project is supported by Hybrid Computing for Signal and DATA Processing, Zoran Kostic, Columbia University.

6. References

[1] Mechler, Günther F., and Ramsis S. Girgis. "Calculation of spatial loss distribution in stacked power and distribution transformer cores." *IEEE Transactions on Power Delivery* 13.2 (1998): 532-537.

[2] Geijn, R. A. Van De, and J. Watts. "SUMMA: scalable universal matrix multiplication algorithm." *Concurrency and Computation: Practice and Experience* 9.4(1997):255–274.