**CUDA Convolution**

**- GPGPU Programming -**

Dec, 2008  
Sangyoon Lee (sjames @ evl.uic.edu)  
Electronic Visualization Laboratory  
University of Illinois at Chicago  
<https://www.evl.uic.edu/sjames/cs525/final.html>

# Starting

Each color component of pixel is composed of three values, RGB. To apply convolution filter on image, there are two ways. The first one is simply to map each component as single float and run convolution filter three times for each channel. The second approach is to modify the original code to use uchar4 or int type as dataset so that we can compute separate channel value within CUDA kernel. I implemeted both ways in *convolutionTexuture* and *convolutionSeparable* but later on I only used the first method since it makes kernel code much simpler.

Step 0: the most Naive approach

From the idea of convolution filter itself, the most naive approach is to use global memory to send data to device and each thread accesses this to compute convolution kernel. Our convolution kernel size is radius 8 (total 17x17 multiplication for single pixel value). In image border area, reference value will be set to 0 during computation. This naive approach includes many of conditional statements and this causes very slow execution.

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| Time (msec) | 1749.08 | 6773.38 | 282668.84 |

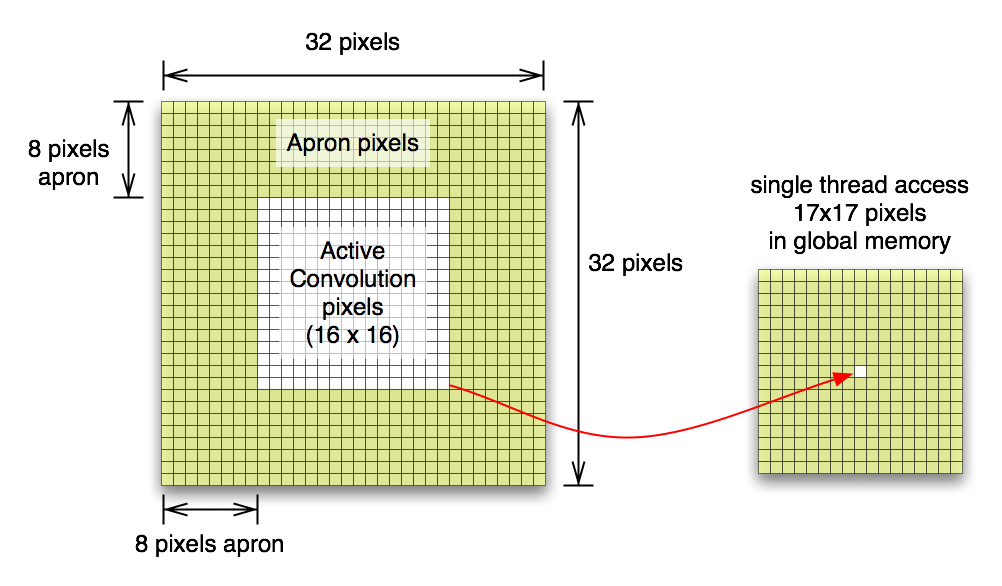


Figure 1. Memory Access Pattern in Naive approach: each threads in block access 17x17 times global memory.

Below is CUDA kernel code.

\_\_global\_\_ **void** **convolutionGPU**(**float** \*d\_Result, **float** \*d\_Data,

**int** dataW, **int** dataH ) {

//////////////////////////////////////////////////////////////////////

// most slowest way to compute convolution

//////////////////////////////////////////////////////////////////////

// global mem address for this thread

**const** **int** gLoc = threadIdx.x + blockIdx.x \* blockDim.x

+ threadIdx.y \* dataW

+ blockIdx.y \* blockDim.y \* dataW;

**float** sum = **0**;

**float** value = **0**;

**for** (**int** i = -KERNEL\_RADIUS; i <= KERNEL\_RADIUS; i++) // row wise

**for** (**int** j = -KERNEL\_RADIUS; j <= KERNEL\_RADIUS; j++) // col wise

{

// check row first

**if** (blockIdx.x == **0** && (threadIdx.x + i) < **0**) // left apron

value = **0**;

**else** **if** ( blockIdx.x == (gridDim.x - **1**) &&

(threadIdx.x + i) > blockDim.x-**1** ) // right apron

value = **0**;

**else**

{

// check col next

**if** (blockIdx.y == **0** && (threadIdx.y + j) < **0**) // top apron

value = **0**;

**else** **if** ( blockIdx.y == (gridDim.y - **1**) &&

(threadIdx.y + j) > blockDim.y-**1** ) // bottom apron

value = **0**;

**else** // safe case

value = d\_Data[gLoc + i + j \* dataW];

}

sum += value \* d\_Kernel[KERNEL\_RADIUS + i] \* d\_Kernel[KERNEL\_RADIUS + j];

}

d\_Result[gLoc] = sum;

}

Step 1: Shared Memory

We all experienced the importance of shared memory throughout project 2. Now, it is time to incorporate with this feature from naive code. When I read nvidia convolution document, I thought that it is OK to invoke many of threads and each thread load data from global mem to shared mem. Then, let some of threads idle. Those are thread loaded apron pixels and do not compute convolution.

The first attempt was to keep active thread size as same as previous and increase block size for apron pixels. This did not work since convolution kernel radius is 8 and it make block size to 32 x 32 (1024). This is bigger than G80 hardware limit (512 threads max per block).

Therefore, I changes scheme as all threads are active and each thread loads four pixels and keep the block size 16x16. Shared Memory size used is 32x32 (this includes all necessary apron pixel values for 16x16 active pixels). Below shows quite a bit of performance improve. This is almost x2.8 speed up over naive approach (in 2048 resolution).

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| Time (msec) | 672.02 | 2421.99 | 9457.56 |

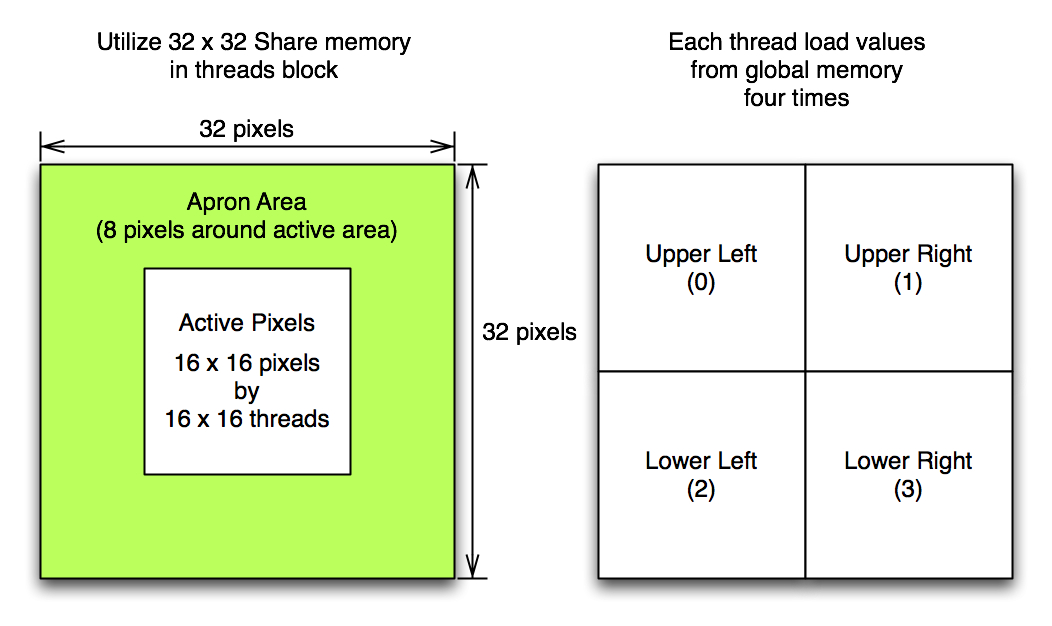


Figure 2. Shared Memory Model for naive approach: each threads in block load 4 values from global memory. Threfore, total shared memory size is 4 times bigger than active convolution pixels to include apron area (kernel radius 8 and block size 16x16. active pixels 256 float vs. shared memory size is 1024 float).

Below codes illustrate the convolution kernel.

\_\_global\_\_ **void** **convolutionGPU**(**float** \*d\_Result, **float** \*d\_Data,

**int** dataW, **int** dataH) {

// Data cache: threadIdx.x , threadIdx.y

\_\_shared\_\_ **float** data[TILE\_W+KERNEL\_RADIUS\***2**][TILE\_W+KERNEL\_RADIUS\***2**];

// global mem address of this thread

**const** **int** gLoc = threadIdx.x +

IMUL(blockIdx.x, blockDim.x) +

IMUL(threadIdx.y, dataW) +

IMUL(blockIdx.y, blockDim.y) \* dataW;

// load cache (32x32 shared memory, 16x16 threads blocks)

// each threads loads four values from global memory into shared mem

// if in image area, get value in global mem, else 0

**int** x, y; // image based coordinate

// original image based coordinate

**const** **int** x0 = threadIdx.x + IMUL(blockIdx.x, blockDim.x);

**const** **int** y0 = threadIdx.y + IMUL(blockIdx.y, blockDim.y);

// case1: upper left

x = x0 - KERNEL\_RADIUS;

y = y0 - KERNEL\_RADIUS;

**if** ( x < **0** || y < **0** )

data[threadIdx.x][threadIdx.y] = **0**;

**else**

data[threadIdx.x][threadIdx.y] = d\_Data[ gLoc - KERNEL\_RADIUS - IMUL(dataW, KERNEL\_RADIUS)];

// case2: upper right

x = x0 + KERNEL\_RADIUS;

y = y0 - KERNEL\_RADIUS;

**if** ( x > dataW-**1** || y < **0** )

data[threadIdx.x + blockDim.x][threadIdx.y] = **0**;

**else**

data[threadIdx.x + blockDim.x][threadIdx.y] = d\_Data[gLoc + KERNEL\_RADIUS - IMUL(dataW, KERNEL\_RADIUS)];

// case3: lower left

x = x0 - KERNEL\_RADIUS;

y = y0 + KERNEL\_RADIUS;

**if** (x < **0** || y > dataH-**1**)

data[threadIdx.x][threadIdx.y + blockDim.y] = **0**;

**else**

data[threadIdx.x][threadIdx.y + blockDim.y] = d\_Data[gLoc - KERNEL\_RADIUS + IMUL(dataW, KERNEL\_RADIUS)];

// case4: lower right

x = x0 + KERNEL\_RADIUS;

y = y0 + KERNEL\_RADIUS;

**if** ( x > dataW-**1** || y > dataH-**1**)

data[threadIdx.x + blockDim.x][threadIdx.y + blockDim.y] = **0**;

**else**

data[threadIdx.x + blockDim.x][threadIdx.y + blockDim.y] = d\_Data[gLoc + KERNEL\_RADIUS + IMUL(dataW, KERNEL\_RADIUS)];

\_\_syncthreads();

// convolution

**float** sum = **0**;

x = KERNEL\_RADIUS + threadIdx.x;

y = KERNEL\_RADIUS + threadIdx.y;

**for** (**int** i = -KERNEL\_RADIUS; i <= KERNEL\_RADIUS; i++)

**for** (**int** j = -KERNEL\_RADIUS; j <= KERNEL\_RADIUS; j++)

sum += data[x + i][y + j] \* d\_Kernel[KERNEL\_RADIUS + j] \* d\_Kernel[KERNEL\_RADIUS + i];

d\_Result[gLoc] = sum;

}

One more optimization tested here. The use of faster integer multiplication instruction (above code already has this change), \_\_mul24. After replacing all integer multiplication with \_\_mul24, I got slight better performance.

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| Time (msec) | 657.59 | 2325.86 | 9301.09 |

Step 2: Filter Separation

Here very important aspect of convolution filter is that it can be separated by row and column. This will reduce computation complexity from m\*m to m+m. Basically we apply two separate convolution. The first one is row-wise and the second one is column-wise from the first result data (apply column convolution over row-wise filtered data). This also reduces some of conditional statement and total number of apron pixel data in each path since we do not need to consider vertical apron in row-convolution kernel and horizontal apron in column-convolution kernel.

This gives me a great improvement over the last shared memory optimized version. This is almost x6.2 speed-up from the last version (in resolution 2048). Code already includes \_\_mul24 instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| Time (msec) | 116.37 | 373.64 | 1492.99 |

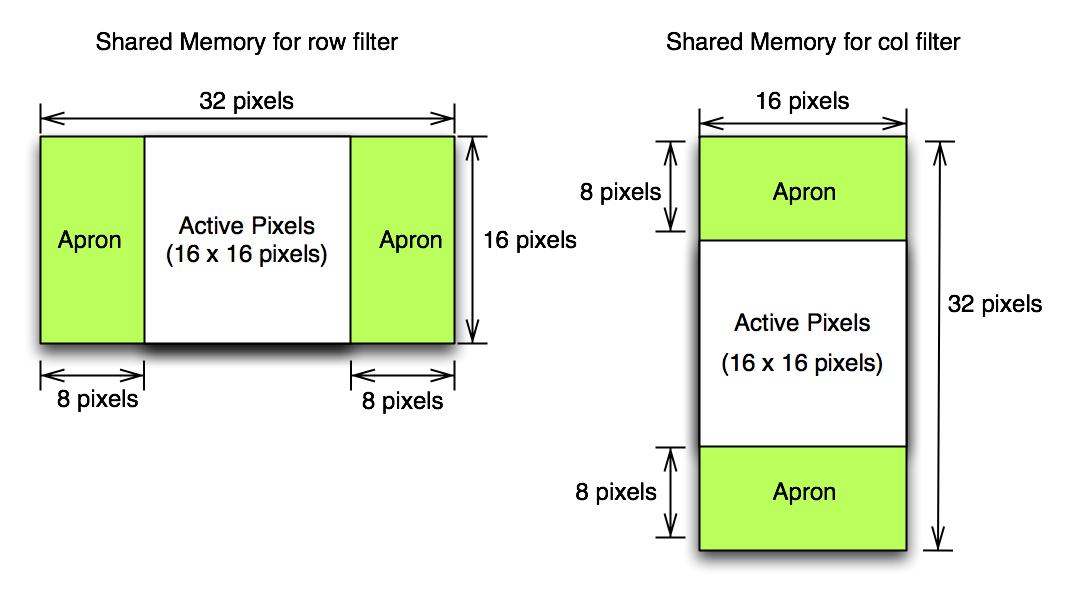


Figure 3. Shared Memory for separate filter: this time only twice bigger memory is necessary for each filter

Unfortunately compiler directive for loopunroll (#pragma unroll 17) does not give any significat improvement. Below shows kernel code for separable convolution filter. Applicatin also modified to run twice of the kernel for row and col convolution.

///////////////////////////////////////////////////////////////////////////

// Row convolution filter

///////////////////////////////////////////////////////////////////////////

\_\_global\_\_ **void** **convolutionRowGPU**(**float** \*d\_Result, **float** \*d\_Data,

**int** dataW, **int** dataH) {

// Data cache: threadIdx.x , threadIdx.y

\_\_shared\_\_ **float** data[TILE\_W + KERNEL\_RADIUS \* **2**][TILE\_H];

// global mem address of this thread

**const** **int** gLoc = threadIdx.x +

IMUL(blockIdx.x, blockDim.x) +

IMUL(threadIdx.y, dataW) +

IMUL(blockIdx.y, blockDim.y) \* dataW;

**int** x; // image based coordinate

// original image based coordinate

**const** **int** x0 = threadIdx.x + IMUL(blockIdx.x, blockDim.x);

// case1: left

x = x0 - KERNEL\_RADIUS;

**if** ( x < **0** )

data[threadIdx.x][threadIdx.y] = **0**;

**else**

data[threadIdx.x][threadIdx.y] = d\_Data[ gLoc - KERNEL\_RADIUS];

// case2: right

x = x0 + KERNEL\_RADIUS;

**if** ( x > dataW-**1** )

data[threadIdx.x + blockDim.x][threadIdx.y] = **0**;

**else**

data[threadIdx.x + blockDim.x][threadIdx.y] = d\_Data[gLoc + KERNEL\_RADIUS];

\_\_syncthreads();

// convolution

**float** sum = **0**;

x = KERNEL\_RADIUS + threadIdx.x;

**for** (**int** i = -KERNEL\_RADIUS; i <= KERNEL\_RADIUS; i++)

sum += data[x + i][threadIdx.y] \* d\_Kernel[KERNEL\_RADIUS + i];

d\_Result[gLoc] = sum;

}

///////////////////////////////////////////////////////////////////////////

// Column convolution filter

///////////////////////////////////////////////////////////////////////////

\_\_global\_\_ **void** **convolutionColGPU**(**float** \*d\_Result, **float** \*d\_Data,

**int** dataW, **int** dataH){

// Data cache: threadIdx.x , threadIdx.y

\_\_shared\_\_ **float** data[TILE\_W][TILE\_H + KERNEL\_RADIUS \* **2**];

// global mem address of this thread

**const** **int** gLoc = threadIdx.x +

IMUL(blockIdx.x, blockDim.x) +

IMUL(threadIdx.y, dataW) +

IMUL(blockIdx.y, blockDim.y) \* dataW;

**int** y; // image based coordinate

// original image based coordinate

**const** **int** y0 = threadIdx.y + IMUL(blockIdx.y, blockDim.y);

// case1: upper

y = y0 - KERNEL\_RADIUS;

**if** ( y < **0** )

data[threadIdx.x][threadIdx.y] = **0**;

**else**

data[threadIdx.x][threadIdx.y] = d\_Data[ gLoc - IMUL(dataW, KERNEL\_RADIUS)];

// case2: lower

y = y0 + KERNEL\_RADIUS;

**if** ( y > dataH-**1** )

data[threadIdx.x][threadIdx.y + blockDim.y] = **0**;

**else**

data[threadIdx.x][threadIdx.y + blockDim.y] = d\_Data[gLoc + IMUL(dataW, KERNEL\_RADIUS)];

\_\_syncthreads();

// convolution

**float** sum = **0**;

y = KERNEL\_RADIUS + threadIdx.y;

**for** (**int** i = -KERNEL\_RADIUS; i <= KERNEL\_RADIUS; i++)

sum += data[threadIdx.x][y + i] \* d\_Kernel[KERNEL\_RADIUS + i];

d\_Result[gLoc] = sum;

}

Step 3: Reorganize Shared Memory

Until step 2, I used 2D array of shared memory to make indexing a bit simpler. Inside computation loop, there is possibility of bank conflict for warp since each thread access first column major memory at the same time. Now, let's re-arrange this shared memory to 1D array so that all threads access data consequently and optimize memory bus here. This only requires changes of indexing in kernel code. Below table shows performance after this re-arrange.

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| Time (msec) | 28.99 | 118.35 | 369.90 |

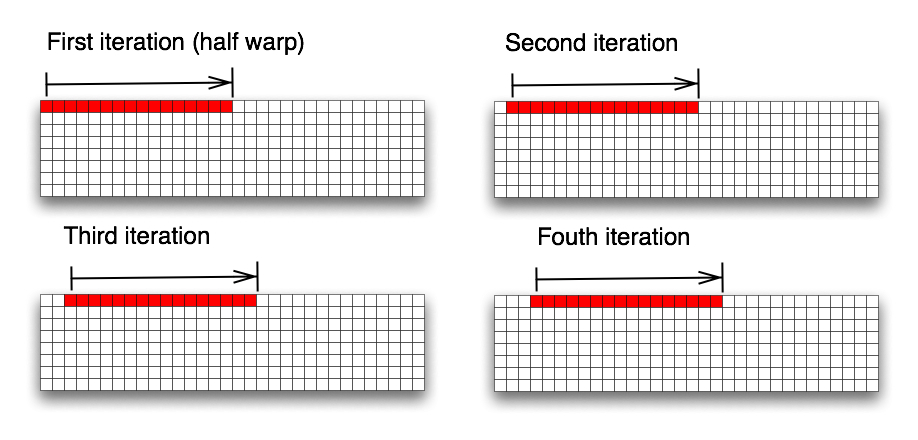


Figure 4. 1D Shared Memory Access Pattern for row filter: shows the first four iteration of convolution computataion. red area is indicating values accessed by half warp threads. Even we obtained fair amount speed up with this re-arrangement, memory access pattern is not aligned well enough to meet the requirement of half-warp alignment for optimal performance.

As you can see above table data, it achived about x3.2 speed-up over the first separable convoluiton implementation (in resolution 2048). Following shows kernel code for this optimization. From step 0 to this point, we made x57 speed-up overall (in resolution 2048).

///////////////////////////////////////////////////////////////////////////

// Row convolution filter

///////////////////////////////////////////////////////////////////////////

\_\_global\_\_ **void** **convolutionRowGPU**(**float** \*d\_Result, **float** \*d\_Data,

**int** dataW, **int** dataH){

// Data cache: threadIdx.x , threadIdx.y

\_\_shared\_\_ **float** data[ TILE\_H \* (TILE\_W + KERNEL\_RADIUS \* **2**) ];

// global mem address of this thread

**const** **int** gLoc = threadIdx.x +

IMUL(blockIdx.x, blockDim.x) +

IMUL(threadIdx.y, dataW) +

IMUL(blockIdx.y, blockDim.y) \* dataW;

**int** x; // image based coordinate

// original image based coordinate

**const** **int** x0 = threadIdx.x + IMUL(blockIdx.x, blockDim.x);

**const** **int** shift = threadIdx.y \* (TILE\_W + KERNEL\_RADIUS \* **2**);

// case1: left

x = x0 - KERNEL\_RADIUS;

**if** ( x < **0** )

data[threadIdx.x + shift] = **0**;

**else**

data[threadIdx.x + shift] = d\_Data[ gLoc - KERNEL\_RADIUS];

// case2: right

x = x0 + KERNEL\_RADIUS;

**if** ( x > dataW-**1** )

data[threadIdx.x + blockDim.x + shift] = **0**;

**else**

data[threadIdx.x + blockDim.x + shift] = d\_Data[gLoc + KERNEL\_RADIUS];

\_\_syncthreads();

// convolution

**float** sum = **0**;

x = KERNEL\_RADIUS + threadIdx.x;

**for** (**int** i = -KERNEL\_RADIUS; i <= KERNEL\_RADIUS; i++)

sum += data[x + i + shift] \* d\_Kernel[KERNEL\_RADIUS + i];

d\_Result[gLoc] = sum;

}

//////////////////////////////////////////////////////////////////////////

// Row convolution filter

//////////////////////////////////////////////////////////////////////////

\_\_global\_\_ **void** **convolutionColGPU**(**float** \*d\_Result, **float** \*d\_Data,

**int** dataW, **int** dataH){

// Data cache: threadIdx.x , threadIdx.y

\_\_shared\_\_ **float** data[TILE\_W \* (TILE\_H + KERNEL\_RADIUS \* **2**)];

// global mem address of this thread

**const** **int** gLoc = threadIdx.x +

IMUL(blockIdx.x, blockDim.x) +

IMUL(threadIdx.y, dataW) +

IMUL(blockIdx.y, blockDim.y) \* dataW;

**int** y; // image based coordinate

// original image based coordinate

**const** **int** y0 = threadIdx.y + IMUL(blockIdx.y, blockDim.y);

**const** **int** shift = threadIdx.y \* (TILE\_W);

// case1: upper

y = y0 - KERNEL\_RADIUS;

**if** ( y < **0** )

data[threadIdx.x + shift] = **0**;

**else**

data[threadIdx.x + shift] = d\_Data[ gLoc - IMUL(dataW, KERNEL\_RADIUS)];

// case2: lower

y = y0 + KERNEL\_RADIUS;

**const** **int** shift1 = shift + IMUL(blockDim.y, TILE\_W);

**if** ( y > dataH-**1** )

data[threadIdx.x + shift1] = **0**;

**else**

data[threadIdx.x + shift1] = d\_Data[gLoc + IMUL(dataW, KERNEL\_RADIUS)];

\_\_syncthreads();

// convolution

**float** sum = **0**;

**for** (**int** i = **0**; i <= KERNEL\_RADIUS\***2**; i++)

sum += data[threadIdx.x + (threadIdx.y + i) \* TILE\_W] \* d\_Kernel[i];

d\_Result[gLoc] = sum;

}

# Step 4: nvidia convolution app

In step 3, we made many of optimizations and improved performance greately. Now, there is a bit of further possible optimization to maximize memory bandwidth by change block organization as we see in nvidia's convolution document. Instead of changing code from step 3, I modified nvidia's original code to use image data to see the difference in performance. As I explained in the very beginning, there are two version of convolution app from nvidia. Following table shows those two application performance (modiifed version).

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| convolutionTexture (msec) | 21.25 | 75.45 | 301.12 |
| convolutionSeparable (msec) | 14.16 | 58.36 | 227.51 |

Compared to the result from step 3, convolutionSeparable optimization shows x2 speed-up (in resolution 2048). This application's kernel code is the same as the original one from nvidia. Only applicaiton side code is modified.

Below table shows a couple of experiments I ran at the beginning of top-down approach with this code (original nvidia convolutionSeparable app).

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **1024** | **2048** | **4096** |
| Base case (no change) (msec) | 14.16 | 58.36 | 227.51 |
| /wo loopunroll (msec) | 14.16 | 58.36 | 227.51 |
| /wo \_\_mul24 (msec) | 23.35 | 78.36 | 302.81 |
| /wo loopunroll & \_\_mul24 (msec) | 24.36 | 77.96 | 300.11 |

As we can see here, loop unrolling does not impact on performance that much but \_\_mul24 intruction gives x1.3 speed-up.

Here is a brief performance chart from step 1 to step 4 (step 0 is excludes due to its huge number).

# Data

- Dataset (Images)

Images used used 1kby1k, 2kby2k and 4kby4k image for performance testing.

- Development platform

Mac OSX 10.5, MacBook Pro 2.5GHz, Geforce 8600M GT 512MB, nvidia CUDA SDK 2.0