IMPLEMENTATION OF WINOGRAD CONVOLUTION

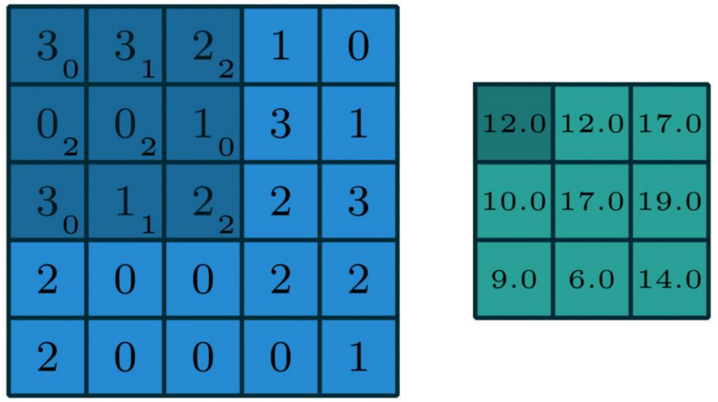
ACA class project @ Politecnico di Milano

by Castiglioni Matteo and Loiacono Tommaso

CONVOLUTION 101

The convolution layer performs the following

- A Kernel (shaded area) slides over input feature map (blue)
- At each kernel position, elementwise product is computed between the kernel and the overlapped input subset
- Result is summed up and constitute the output feature map (cyan)



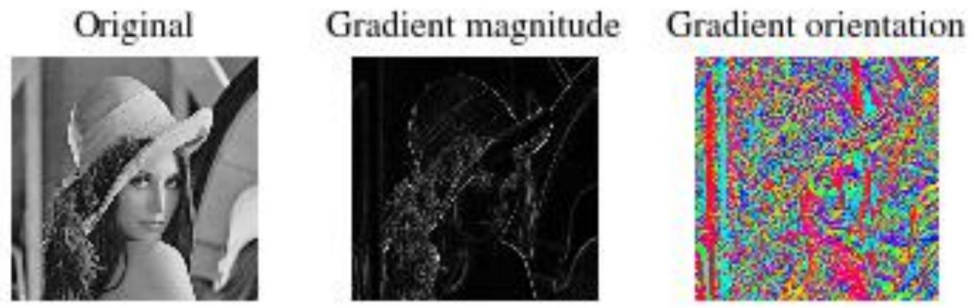
3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

A very well know tool for image processing

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1



Slide by courtesy of professor Matteo Matteucci – Soft Computing Course

WINOGRAD ALGORITHM

Since 1980 is known that a minimal filtering algorithm to compute convolution, $F(m, r)$, where m is the size of the output, and r is the size of the kernel, requires $(m + r - 1)$ multiplications.

Shmuel Winograd documented [1] for 1d convolution an algorithm which was minimal. In matrix form could be written as:

$$Y = A^T\left[(Gg) \odot (B^T b)\right]$$

g = kernel, b = input.

A, B and G should be computed for every kernel and output size.

FROM 1D TO 2D

Bitwise multiplication

Kernel matrix

Input tile*

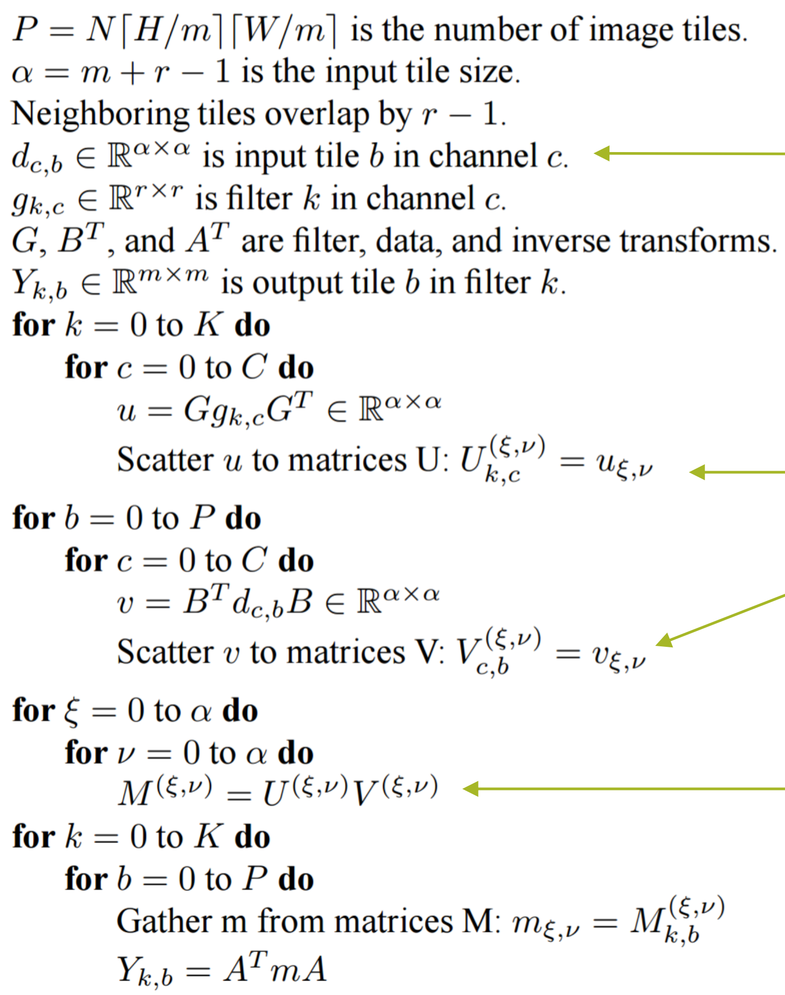
$$Y = A^T \left[\left[G g G^T \right] \odot \left[B^T b B \right] \right] A$$

These matrices are should be computed for every output and kernel size

Output of a single tile*

* Every image $H \times W$ is divided in multiple tiles and then the algorithm is performed on every tile. The number of tiles for a $m \times m$ kernel is $P = \lceil H/m \rceil \lceil W/m \rceil$.

ENTIRE ALGORITHM



```
$P = N\lceil H/m \rceil \lceil W/m \rceil$ is the number of image tiles.
$\alpha = m + r - 1$ is the input tile size.
Neighboring tiles overlap by $r - 1$.
$d_{c,b} \in \mathbb{R}^{\alpha \times \alpha}$ is input tile $b$ in channel $c$.
$g_{k,c} \in \mathbb{R}^{r \times r}$ is filter $k$ in channel $c$.
$G$, $B^T$, and $A^T$ are filter, data, and inverse transforms.
$Y_{k,b} \in \mathbb{R}^{m \times m}$ is output tile $b$ in filter $k$.
for $k = 0$ to $K$ do
    for $c = 0$ to $C$ do
        $u = G g_{k,c} G^T \in \mathbb{R}^{\alpha \times \alpha}$
        Scatter $u$ to matrices U: $U_{k,c}^{(\xi,\nu)} = u_{\xi,\nu}$
for $b = 0$ to $P$ do
    for $c = 0$ to $C$ do
        $v = B^T d_{c,b} B \in \mathbb{R}^{\alpha \times \alpha}$
        Scatter $v$ to matrices V: $V_{c,b}^{(\xi,\nu)} = v_{\xi,\nu}$
for $\xi = 0$ to $\alpha$ do
    for $\nu = 0$ to $\alpha$ do
        $M^{(\xi,\nu)} = U^{(\xi,\nu)} V^{(\xi,\nu)}$
for $k = 0$ to $K$ do
    for $b = 0$ to $P$ do
        Gather m from matrices M: $m_{\xi,\nu} = M_{k,b}^{(\xi,\nu)}$
        $Y_{k,b} = A^T m A$
```

Multichannel is supported.

Intermediate matrices U and V are computed.

Multiple channels are collapsed back to one.

WHAT DO THE SAY?

Lavin and Gray state [2] that theoreticaly the Winograd algorithm yields a speedup over standard convolution of

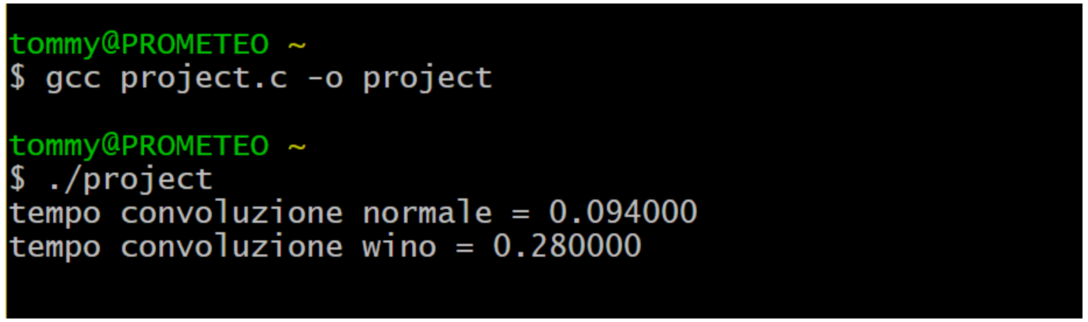
$$\frac{r^2m^2}{(m+r-1)^2}$$

In practice, they say, this results become possibile if we can reduce the number of multiplications we do inside our cpu, in exchange for a greater number of sums, which cost less.

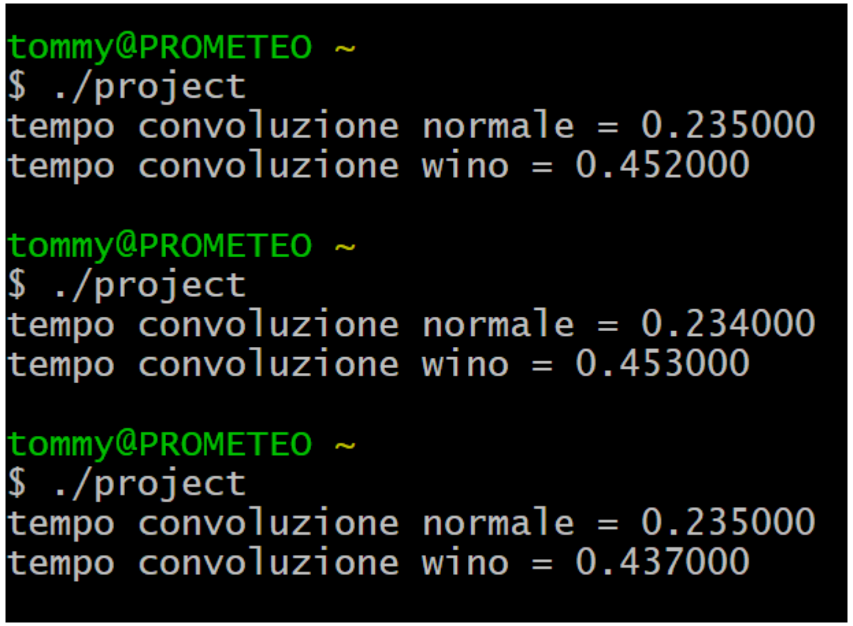
They also state that many CPU and GPU has efficient implementation of matrix multiplication, which means another speedup in performances.

FIRST TRY… POOR RESULTS.

- We tried with 3 x 3 output and kernel, 1 channel, many images.
- Winograd performs much worse than standard convolution.
- Sums are as CPU-intense as multiplication.
- Maybe with some preprocessing would be good?
  - Certain matrices should be calculated only once…
  - We can cancel multiplication with 0 and 1.
  - Other optimizations could be perfomed…



```
tommy@PROMETEO ~
$ gcc project.c -o project

tommy@PROMETEO ~
$ ./project
tempo convoluzione normale = 0.094000
tempo convoluzione wino = 0.280000
```



```
tommy@PROMETEO ~
$ ./project
tempo convoluzione normale = 0.235000
tempo convoluzione wino = 0.452000

tommy@PROMETEO ~
$ ./project
tempo convoluzione normale = 0.234000
tempo convoluzione wino = 0.453000

tommy@PROMETEO ~
$ ./project
tempo convoluzione normale = 0.235000
tempo convoluzione wino = 0.437000
```

IMPLEMENTATION OF NORMAL CONVOLUTION ALGORITHM

```
void conv(float inp[NOI][CHN][IN][IN],float filter[NOK][CHN][F1][F2],float res[NOI][NOK][IN-F1+1][IN-F2+1], int numImg){

    float sum;

    for (int nk=0;nk<NOK;nk++){
        for(int ch=0;ch<CHN;ch++){
            for (int j=0;j<IN-F1+1;j++){
                for (int k=0;k<IN-F2+1;k++) {

                    sum=0;

                    for (int m=0;m<F1;m++){
                        for(int n=0;n<F2;n++){

                            sum=sum+inp[numImg][ch][j+m][k+n]*filter[nk][ch][m][n];

                        }
                    }

                    res[numImg][nk][j][k]=res[numImg][nk][j][k]+sum;
                }
            }
        }
    }

}
```

NOI: Number of Image
CHN: channel
IN: input size
NOK: Number of Kernel (for multiple kernel convolution)
F1-F2: Kernel Size

PARTIAL IMPLEMENTATION OF WINOGRAD ALGORITHM

```
//Winograd convolution
void conv33(float inp[NOI][CHN][IN][IN],float filter[NOK][CHN][F1][F2]){

    calc_u33(filter,u);
    calc_v33(inp,v);

    for (int i=0; i<NOI; i++){
        calc_Elem_wise(u,v,m,i);
        calc_y(m,y,i);
        buildRes(y,i,out);
    }

}
```

```
void calc_u33(float filter[NOK][CHN][F1][F2],float res2[NOK][CHN][F1*2][F1*2]){

    int m=F1*2;
    float sum;


    for (int nk=0;nk<NOK;nk++){
        for(int ch=0;ch<CHN;ch++){
            for (int j=0;j<F1;j++){

                r1[nk][0][j]=filter[nk][ch][0][j]/4;
                r1[nk][1][j]=(filter[nk][ch][0][j]+filter[nk][ch][1][j]+filter[nk][ch][2][j])/(-6);
                r1[nk][2][j]=(-filter[nk][ch][0][j]+filter[nk][ch][1][j]-filter[nk][ch][2][j])/(6);
                r1[nk][3][j]=filter[nk][ch][0][j]/24+filter[nk][ch][1][j]/12+filter[nk][ch][2][j]/6;
                r1[nk][4][j]= filter[nk][ch][0][j]/24-filter[nk][ch][1][j]/12+filter[nk][ch][2][j]/6;
                r1[nk][5][j]=filter[nk][ch][2][j];

            }

            for (int j=0;j<m;j++){

                res2[nk][ch][j][0]=r1[nk][j][0]/4;
                res2[nk][ch][j][1]=(r1[nk][j][0]+r1[nk][j][1]+r1[nk][j][2])/(-6);
                res2[nk][ch][j][2]=(-r1[nk][j][0]+r1[nk][j][1]-r1[nk][j][2])/(6);
                res2[nk][ch][j][3]=r1[nk][j][0]/24+r1[nk][j][1]/12+r1[nk][j][2]/6;
                res2[nk][ch][j][4]=r1[nk][j][0]/24-r1[nk][j][1]/12+r1[nk][j][2]/6;
                res2[nk][ch][j][5]=r1[nk][j][2];

            }
        }
    }

}
```

(MAGICIAN’S) TRICKS USED

- By hand simplifications → speedup matrix multiplications.
- Using multichannelling → collapsing channels during elementwise multiplications means less operations to calculate the final result.
- Utilize same Kernel for different images → *U* is calculated only once for kernel, and used for different images.
- Utilize multiple kernels → another speedup when using multiple images.

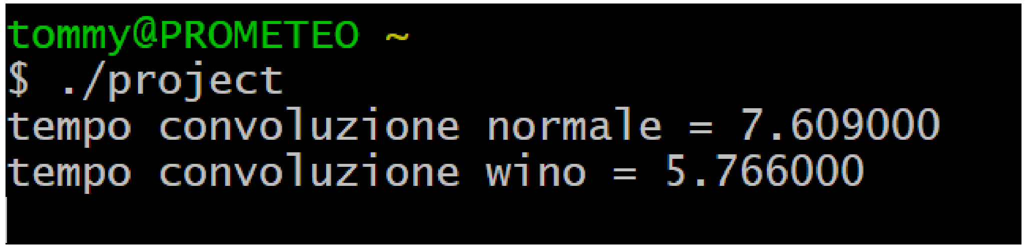
```
r2[ni][0][k]=4*inp[ni][ch][i][j+k]-5*inp[ni][ch][i+2][j+k]+inp[ni][ch][i+4][j+k];
r2[ni][1][k]=-4*(inp[ni][ch][i+1][j+k]+inp[ni][ch][i+2][j+k])+inp[ni][ch][i+3][j+k]+inp[ni][ch][i+4][j+k];
r2[ni][2][k]=4*(inp[ni][ch][i+1][j+k]-inp[ni][ch][i+2][j+k])-inp[ni][ch][i+3][j+k]+inp[ni][ch][i+4][j+k];
r2[ni][3][k]=2*(-inp[ni][ch][i+1][j+k]+inp[ni][ch][i+3][j+k])-inp[ni][ch][i+2][j+k]+inp[ni][ch][i+4][j+k];
r2[ni][4][k]=2*(inp[ni][ch][i+1][j+k]-inp[ni][ch][i+3][j+k])-inp[ni][ch][i+2][j+k]+inp[ni][ch][i+4][j+k];
r2[ni][5][k]=4*inp[ni][ch][i+1][j+k]-5*inp[ni][ch][i+3][j+k]+inp[ni][ch][i+5][j+k];
```

```
calc_u33(filter,u);
calc_v33(inp,v);

for (int i=0; i<NOI; i++){
    calc_Elem_wise(u,v,m,i);
    calc_y(m,y,i);
    buildRes(y,i,out);
}
```

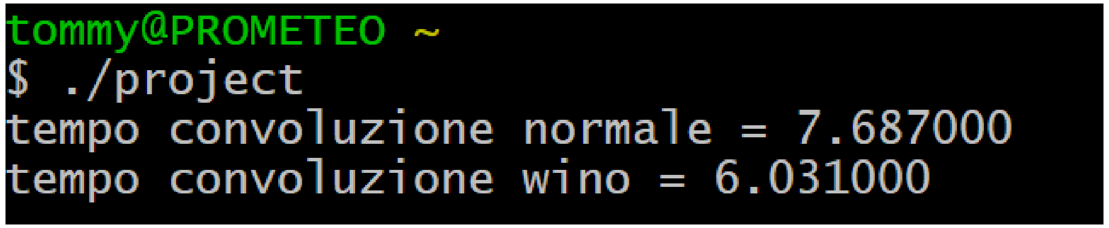
```
#define F1 3
#define F2 3
#define NOI 20
#define CHN 3
#define NOK 10
#define IN 502
```

```
for (int nk=0;nk<NOK;nk++){
    for(int ch=0;ch<CHN;ch++){
        for(int i=0;i<NUM_OF_TILES;i++){
            for(int j=0;j<TD;j++){
                for(int k=0;k<TD;k++){

                    m[numImg][nk][i][j][k]=m[numImg][nk][i][j][k]+u[nk][ch][j][k]*v[numImg][ch][i][j][k];

                }
            }
        }
    }
}
```

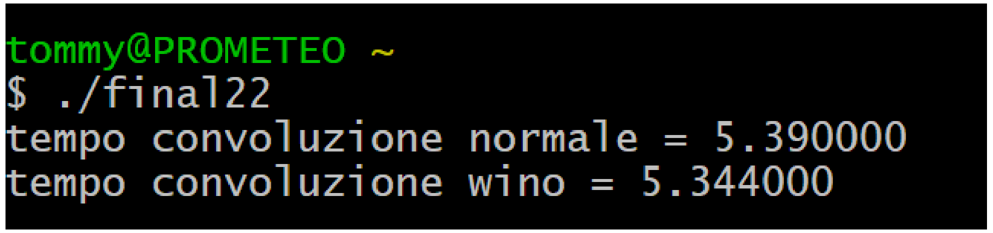
BETTER RESULTS…



```
tommy@PROMETEO ~
$ ./project
tempo convoluzione normale = 7.609000
tempo convoluzione wino = 5.766000
```



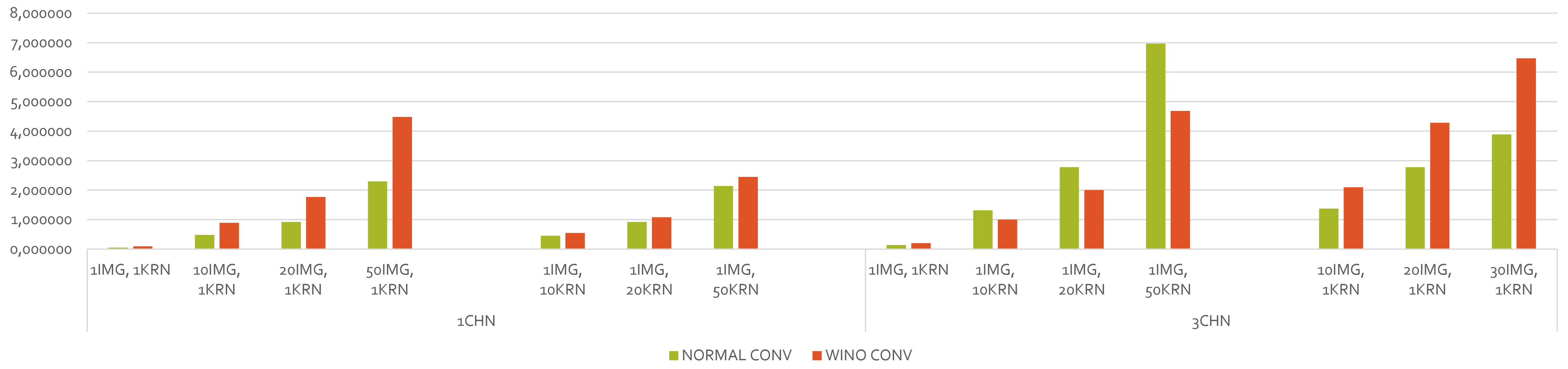
```
tommy@PROMETEO ~
$ ./project
tempo convoluzione normale = 7.687000
tempo convoluzione wino = 6.031000
```

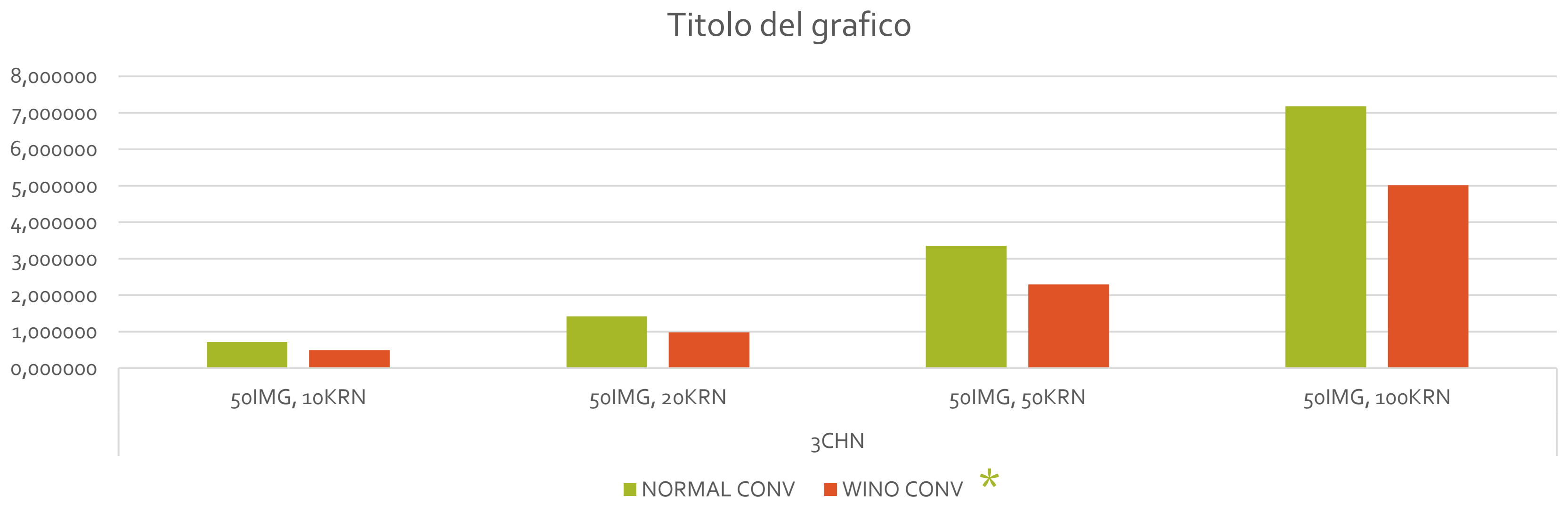


```
tommy@PROMETEO ~
$ ./final22
tempo convoluzione normale = 5.390000
tempo convoluzione wino = 5.344000
```

2x2 implementation… not so good.

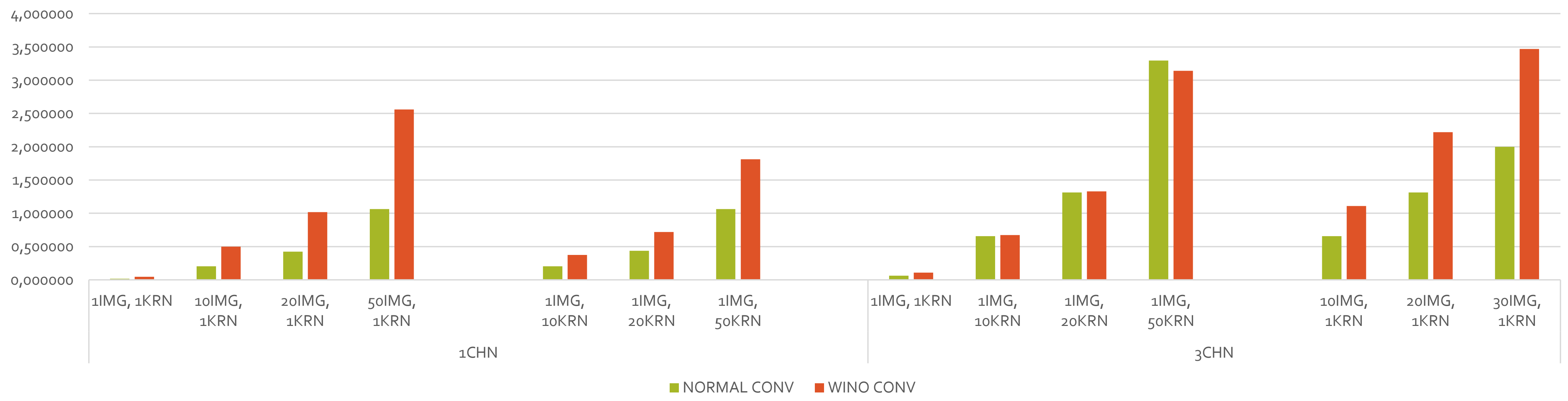
RESULTS FOR A 3 × 3 FILTER

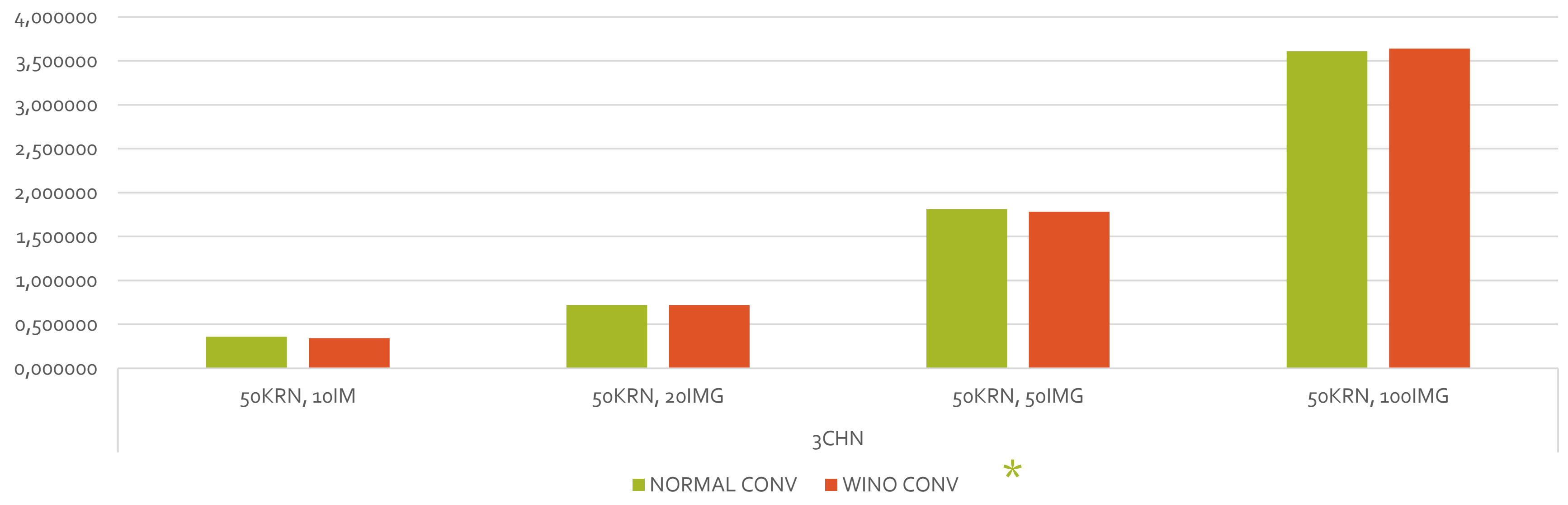




*Tests performed on smaller images in order to compile…

RESULTS FOR A 2 × 2 FILTER





*

*Tests performed on smaller images in order to compile...

WHAT CACHEGRIND SAYS…

Winograd

```
I    refs:      25,800,921,571
I1   misses:             1,262
LLi  misses:             1,258
I1   miss rate:           0.00%
LLi  miss rate:           0.00%

D    refs:       7,171,907,394  (6,700,850,666 rd   + 471,056,728 wr)
D1   misses:        29,729,799  (   26,294,945 rd   +   3,434,854 wr)
LLd  misses:        12,356,781  (    8,921,966 rd   +   3,434,815 wr)
D1   miss rate:            0.4% (          0.4%     +         0.7%  )
LLd  miss rate:            0.2% (          0.1%     +         0.7%  )

LL refs:            29,731,061  (   26,296,207 rd   +   3,434,854 wr)
LL misses:          12,358,039  (    8,923,224 rd   +   3,434,815 wr)
LL miss rate:              0.0% (          0.0%     +         0.7%  )
```

Normal

```
I    refs:      82,604,716,684
I1   misses:             1,099
LLi  misses:             1,094
I1   miss rate:           0.00%
LLi  miss rate:           0.00%

D    refs:      28,840,611,475  (26,585,924,096 rd   + 2,254,687,379 wr)
D1   misses:        19,349,275  (    19,153,532 rd   +       195,743 wr)
LLd  misses:         3,518,174  (     3,322,470 rd   +       195,704 wr)
D1   miss rate:            0.1% (           0.1%     +           0.0%  )
LLd  miss rate:            0.0% (           0.0%     +           0.0%  )

LL refs:            19,350,374  (    19,154,631 rd   +       195,743 wr)
LL misses:           3,519,268  (     3,323,564 rd   +       195,704 wr)
LL miss rate:              0.0% (           0.0%     +           0.0%  )
```

2 X 2 IS EVEN WORSE

```
I   refs:      17,086,439,809
I1  misses:             1,139
LLi misses:             1,134
I1  miss rate:           0.00%
LLi miss rate:           0.00%

D   refs:       5,670,198,917  (5,311,888,595 rd   + 358,310,322 wr)
D1  misses:        24,013,905  (   20,692,998 rd   +   3,320,907 wr)
LLd misses:        10,564,517  (    7,243,649 rd   +   3,320,868 wr)
D1  miss rate:            0.4% (          0.4%     +         0.9%  )
LLd miss rate:            0.2% (          0.1%     +         0.9%  )

LL refs:           24,015,044  (   20,694,137 rd   +   3,320,907 wr)
LL misses:         10,565,651  (    7,244,783 rd   +   3,320,868 wr)
LL miss rate:             0.0% (          0.0%     +         0.9%  )
```

CONCLUSIONS

- More misses in Winograd convolution, but it is a better algorithm under certain conditions.
- The misses are related to temporal and spatial locality principle not exploited by the Winograd algorithm (the matrix is not accessed sequentialy, like in the normal convolution).
- Usage of parallelization libraries (C++ AMP, SSE2, ecc…) could lead to better results, along the usage of GPUs to handle matrix multiplications (but these ones also improve normal convolution).
- Other libraries, like Openblas, have been used with no percievable effect on the result, sometimes worsening the computation time.

QUESTIONS?

REFERENCES

- [1] Shmuel Winograd. Arithmetic complexity of computations, volume 33. Siam, 1980
- [2] Andrew Lavin, Scott Gray, Fast Algorithms for Convolutional Neural Networks, https://arxiv.org/abs/1509.09308