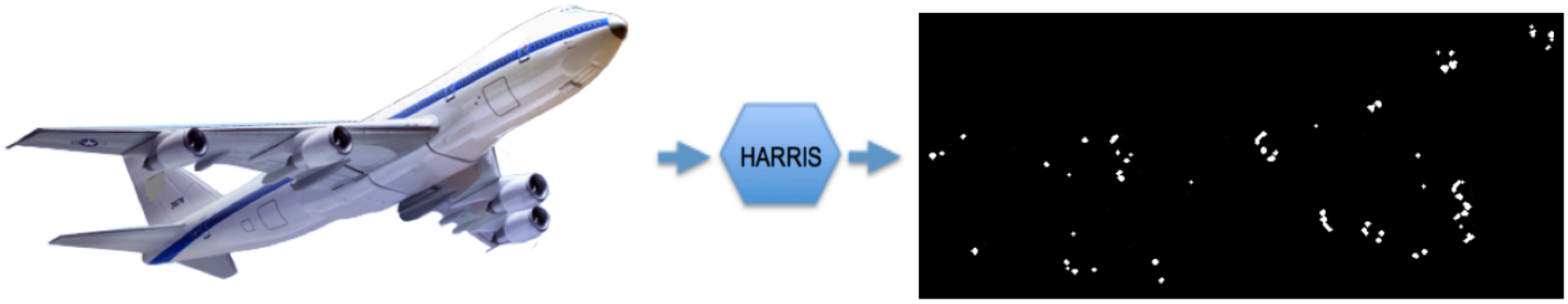
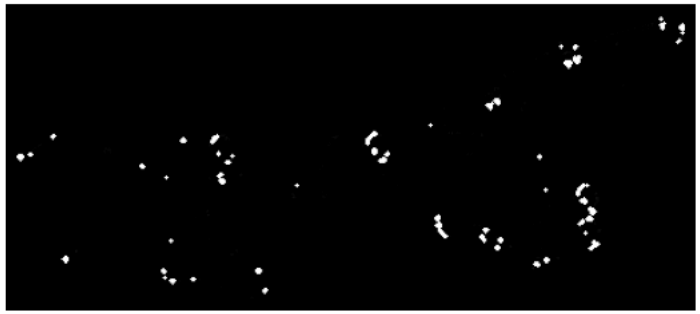
Harris Corner Detection on a NUMA Manycore

Claude TADONKI
Centre de Recherche en Informatique (CRI)

Joint work with
Olfa HAGGUI and **Lionel LACASSAGNE**
Sousse National School of Engineering - Laboratoire d'Informatique de Paris 6 (LIP6)
Mines ParisTech - PSL



Monthly Seminar at Centre de Recherche en Informatique (CRI)
April 16, 2018, FONTAINEBLEAU - FRANCE



Corner points are used for motion detection for instance

1. For each pixel (x, y) in the input image, compute the *Harris matrix* $G = \begin{pmatrix} g_{xx} & g_{xy} \\ g_{xy} & g_{yy} \end{pmatrix}$, with

$$g_{xx} = (\frac{\partial I}{\partial x})^2 \otimes w \qquad g_{xy} = (\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}) \otimes w \qquad g_{yy} = (\frac{\partial I}{\partial y})^2 \otimes w, \tag{1}$$

   where $\otimes$ denotes convolution operator and w is the Gaussian filter.
2. For each pixel (x, y), compute the Harris criterion given by

$$c(x, y) = det(G) - k(trace(G))^2, \tag{2}$$

   where $det(G) = g_{xx}.g_{yy} - g_{xy}^2$, k an empirical constant, and $trace(G) = g_{xx} + g_{yy}$.
3. Set all $c(x, y)$ which are below λ to zero.
4. Extract points (x, y) having the maximum $c(x, y)$ within a window neighborhood. These points represent the corners.

From the intensity I (color not needed), we need to compute (approximated) derivatives and combined them

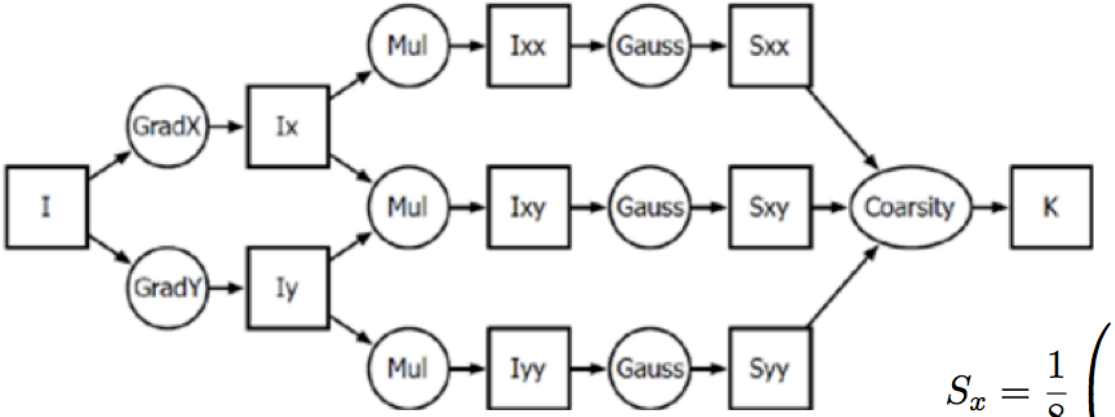


Figure 2: Harris detector workflow

$$S_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad S_y = \frac{1}{8}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

$$G = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

- The procedure applies a series of **convolution kernels** to the input intensity matrix
- Each convolution is a **stencil computation**
- The whole computation can be **fully serial**, **fully pilelined**, or **hybrid**.
- Memory acces patterns are the main focus w.r.t **performances**

- **Stencil computation**: *redundant memory accesses, cache misses*, and *unalignement*
- **Scheduling of the convolutions** : Intermediate reads/writes (*space and access time*)
- **SIMD**: not efficient in its standard form (what we get from the compiler)
- **SM Parallelism**: bus contention, threads synchronization, NUMA

```
void Gauss(float *S, float *X, int tid){
    unsigned int i,j,gi,gj,ti,tj;
    for(i = 1; i < n-1; i++)
        for(j = 1; j < n-1; j++)
            S[z(i,j)] = (X[z(i-1,j-1)]+2*X[z(i-1,j)]+X[z(i-1,j+1)]+
                         2*(X[z(i,j-1)]+2*X[z(i,j)]+X[z(i,j+1)])+
                         X[z(i+1,j-1)]+2*X[z(i+1,j)]+X[z(i+1,j+1)])/16;
}
```

*We are going to explain **our approach** for each of the **aforementioned aspects !!!***

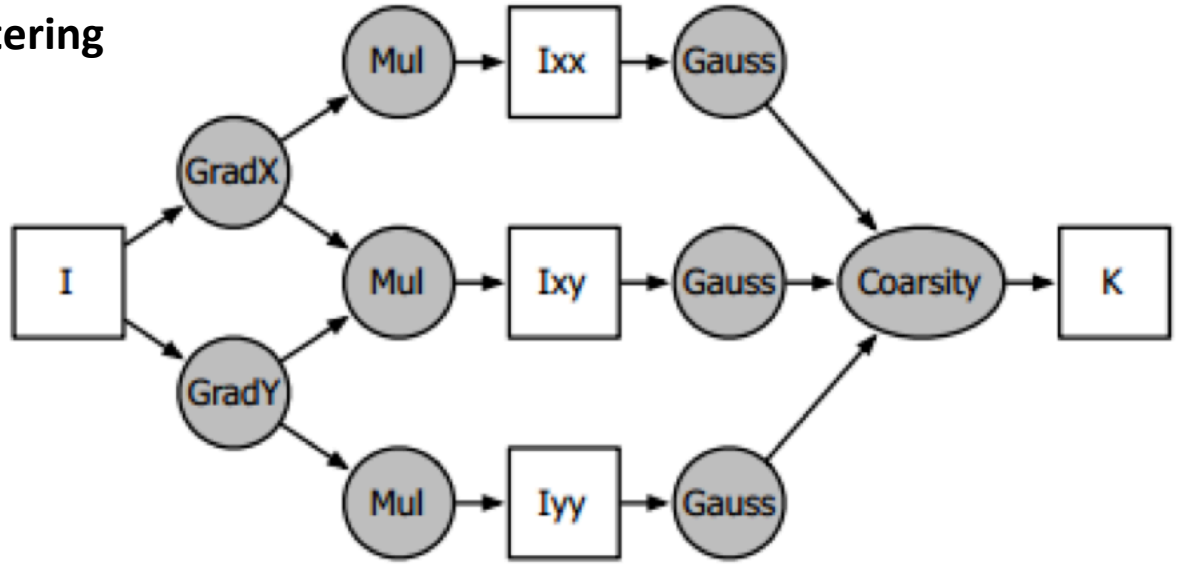
Separability

$$S_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \frac{1}{8}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

$$S_y = \frac{1}{8}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{8}\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$

$$G = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{16}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$

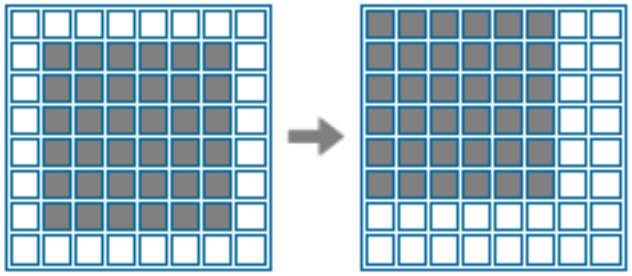
Half-Pipe Clustering



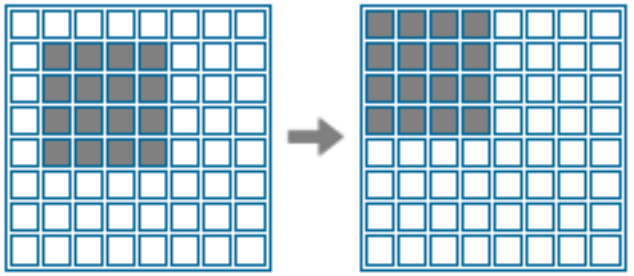
- Vectorization with the original memory access pattern leads to unaligned accesses

We propose a **diagonal shift** to keep all accesses aligned

(i, j) is stored at position $(i - 1, j - 1)$

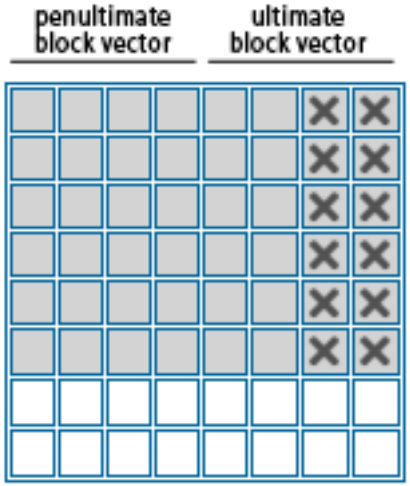


(a) Iteration space (b) Actual storage



(a) Iteration space (b) Actual storage

Storage Layout for GAUSS



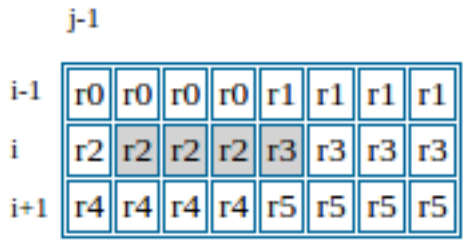
- The last vector register contains 2 dirty values, but the whole vector is stored (4-components vector)

- The goal here is to **load data into vector registers once**, and then perform **all dependent calculations** (*optimal data consumption and memory accesses saving*)

$$A = \begin{pmatrix} a & \alpha a & \beta a \\ b & \alpha b & \beta b \\ c & \alpha c & \beta c \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \begin{pmatrix} 1 & \alpha & \beta \end{pmatrix}$$

$$\begin{aligned} @_{(i,j)} = \; & aI_{i-1,j-1} + \alpha aI_{i-1,j} + \beta aI_{i-1,j+1} \\ & + bI_{i,j-1} + \alpha bI_{i,j} + \beta bI_{i,j+1} \\ & + cI_{i+1,j-1} + \alpha cI_{i+1,j} + \beta cI_{i+1,j+1} \end{aligned}$$

$$\begin{aligned} @_{(i,j)} = \; & (aI_{i-1,j-1} + bI_{i,j-1} + cI_{i+1,j-1}) + \\ & \alpha(aI_{i-1,j} + bI_{i,j} + cI_{i+1,j}) + \\ & \beta(aI_{i-1,j+1} + bI_{i,j+1} + cI_{i+1,j+1}) \; . \end{aligned}$$



$$u = (u_0, u_1, u_2, u_3) \text{ and } v = (v_0, v_1, v_2, v_3)$$
$$\lhd_1(u, v) = (u_1, u_2, u_3, v_0)$$
$$\lhd_2(u, v) = (u_2, u_3, v_0, v_1)$$

$$\begin{aligned} & [a \times r_0 + b \times \lhd_1(r_0, r_1) + c \times \lhd_2(r_0, r_1)] + \\ & \alpha \times [a \times r_2 + b \times \lhd_1(r_2, r_3) + c \times \lhd_2(r_2, r_3)] + \\ & \beta \times [a \times r_4 + b \times \lhd_1(r_4, r_5) + c \times \lhd_2(r_4, r_5)] \end{aligned}$$

- For the computation of the next block, we just need to do $r_0 := r_1$, $r_2 := r_3$, $r_4 := r_5$, and only reload r_1, r_3, and r_5, thereby saving three loads.

$$[a \times r_0 + b \times \lhd_1(r_0, r_1) + c \times \lhd_2(r_0, r_1)] +$$
$$\alpha \times [a \times r_2 + b \times \lhd_1(r_2, r_3) + c \times \lhd_2(r_2, r_3)] +$$
$$\beta \times [a \times r_4 + b \times \lhd_1(r_4, r_5) + c \times \lhd_2(r_4, r_5)]$$

$$a \times [r_0 + \alpha \times r_2 + \beta r_4]$$
$$b \times [\lhd_1(r_0, r_1) + \alpha \lhd_1 (r_2, r_3) + \beta \lhd_1 (r_4, r_5)]$$
$$c \times [\lhd_2(r_0, r_1) + \alpha \lhd_2 (r_2, r_3) + \beta \lhd_2 (r_4, r_5)]$$

$$a \times [r_0 + \alpha \times r_2 + \beta r_4]$$
$$b \times [\lhd_1(r_0 + \alpha \times r_2 + \beta r_4, r_1 + \alpha \times r_3 + \beta r_5)]$$
$$c \times [\lhd_2(r_0 + \alpha \times r_2 + \beta r_4, r_1 + \alpha \times r_3 + \beta r_5)]$$

$$u = r_0 + \alpha \times r_2 + \beta \times r_4 \text{ and } v = r_1 + \alpha \times r_3 + \beta \times r_5$$

$$a \times u + b \times \lhd_1(u, v) + c \times \lhd_2(u, v).$$

Thus, for the computation of an entire row, the typical steps at each iteration are:

1: $u = load(I_{i-1,j})$; $v = load(I_{i,j})$; $w = load(I_{i+1,j})$;
2: $s := u + \alpha \times v + \beta \times w$;
3: $t := a \times r + b \times \lhd_1(r, s) + c \times \lhd_2(r, s)$;
4: $store(t)$;
5: $j := next(j)$; {next block vector}
6: $r := s$;

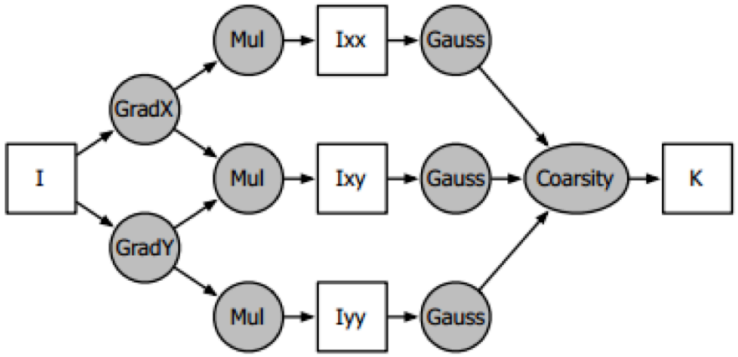
$\lhd_1(v1, v2)$

```
vv  = _mm256_permute_ps(v1, 0b00111001);
u1  = _mm256_permute2f128_ps(vv, vv, 0x81);
u1  = _mm256_blend_ps(vv, u1, 0b10001000);
vv  = _mm256_permute_ps(v2, 0b00111001);
u2  = _mm256_permute2f128_ps(vv, vv, 0);
vv  = _mm256_blend_ps(u1, u2, 0b10000000);
```

$\lhd_2(v1, v2)$

```
vv = _mm256_permute_ps(v1, 0b01001110);
u1 = _mm256_permute2f128_ps(vv, vv, 0x81);
u1 = _mm256_blend_ps(vv, u1, 0b11001100);
vv = _mm256_permute_ps(v2, 0b01001110);
u2 = _mm256_permute2f128_ps(vv, vv, 0);
vv = _mm256_blend_ps(u1, u2, 0b11000000);
```

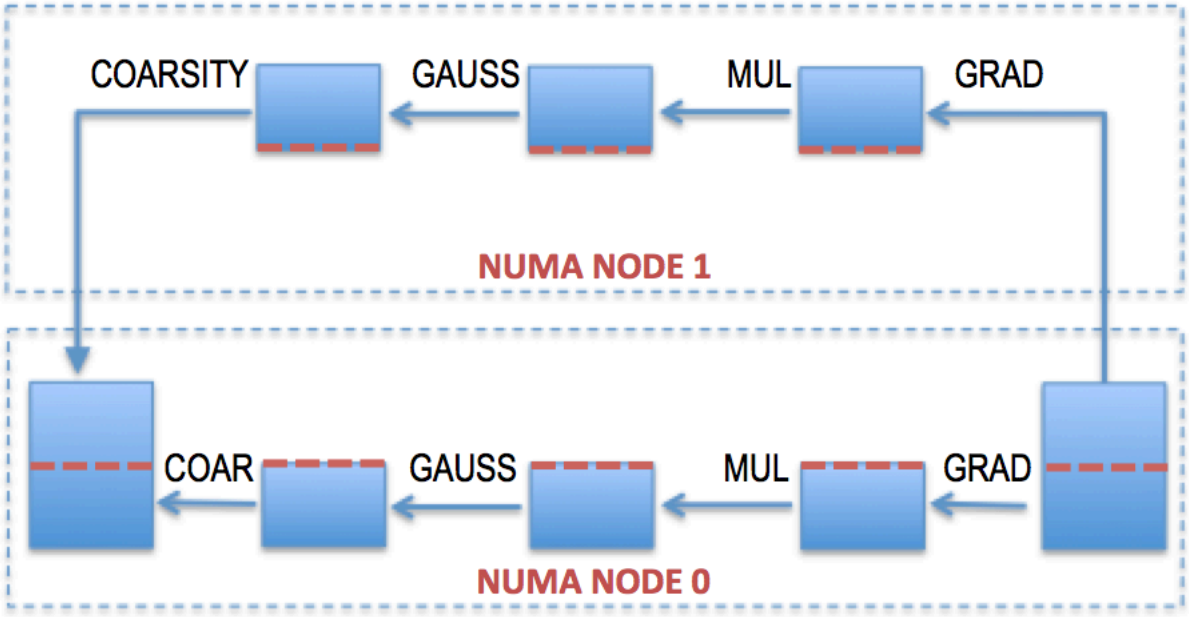
- We consider the **half-pipe clustering**

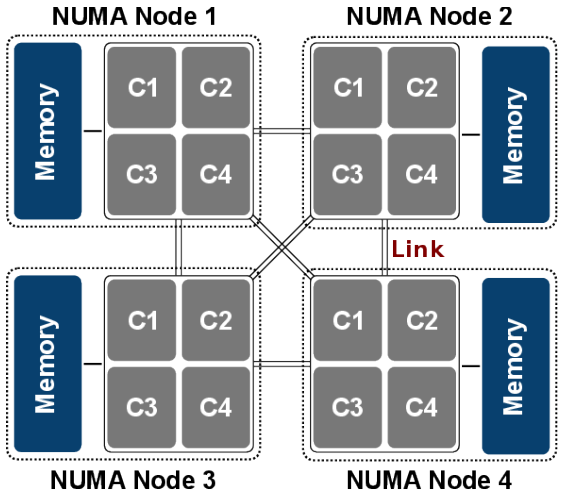


- We **pipeline** the two cluster steps (SOLBEL+MUL and GAUSS+COARSITY) through a **loop fusion**

$$
\begin{array}{ll}
1: & \{l_0^{(I)}, l_1^{(I)}, l_2^{(I)}\} \rightarrow [\text{SOBEL+MUL}] \rightarrow \{l_0^{(S)}, l_1^{(S)}\}\ ; \\
2: & for(i = 2; i < n-2; i++)\{ \\
3: & \quad \{l_{i-1}^{(I)}, l_i^{(I)}, l_{i+1}^{(I)}\} \rightarrow [\text{SOBEL+MUL}] \rightarrow l_i^{(S)}; \\
4: & \quad \{l_{i-2}^{(S)}, l_{i-1}^{(S)}, l_i^{(S)}\} \rightarrow [\text{GAUSS+COARSITY}] \rightarrow l_{i-2}^{(O)}; \\
5: & \}
\end{array}
$$

- We apply an **array contraction** (*mod 3*) for the intermediate storage





- Both input and output images are stored on NUMA node 0
- Each NUMA node locally computes its chunk (block of lines) of the final result
- Within each NUMA node, the work is equally distributed by block to its threads
- Expected memory allocation on the NUMA nodes is done by explicit binding routines

N	GFlops (1)	GFlops (2)	T(s)	GFlops	%Peak
1 024	4.03	10.44	0.00306	12.67	66
2 048	6.57	9.96	0.01160	13.39	69
4 096	5.56	9.56	0.04757	13.05	67
8 192	5.24	9.15	0.22156	11.21	58
16 384	5.25	9.03	0.97172	10.22	53
32 768	-	-	3.71246	10.10	55

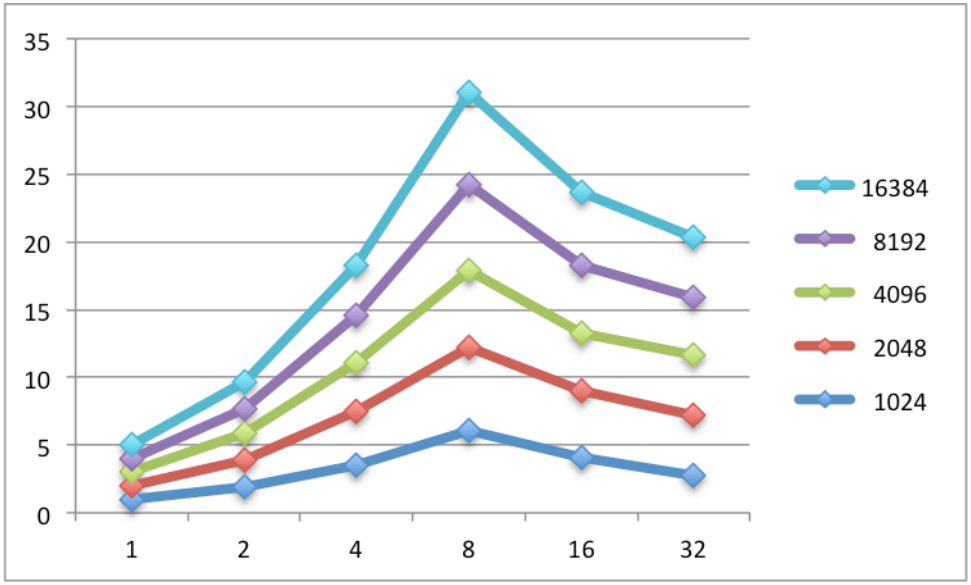
Table 3: Performance results of our sequential implementation

- (1) Original SIMD without the in-registers strategy
- (2) Optimized SIMD with the in-registers strategy

In-register strategy doubles the overall peformances
and
Our sequential implementation outperforms the state-of-the-art absolute performance

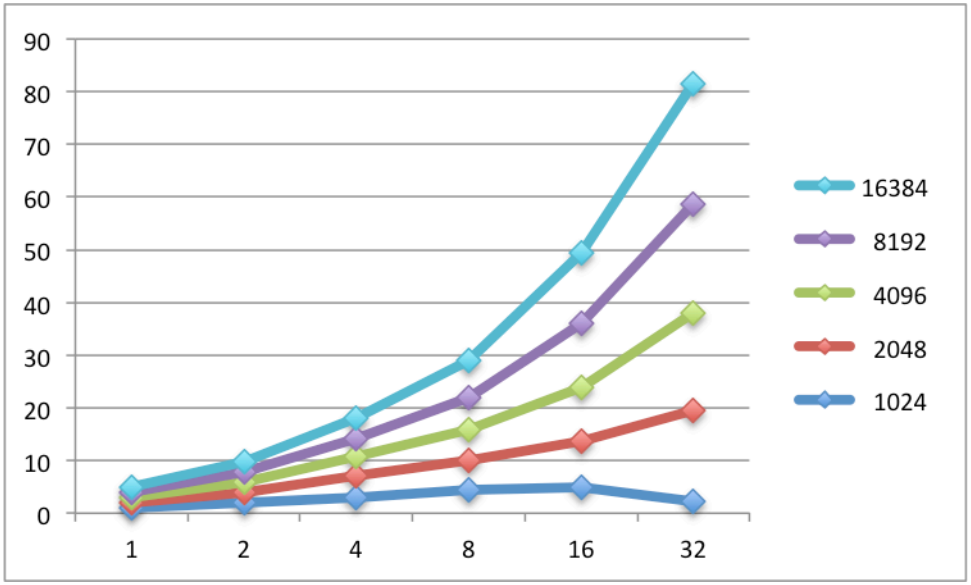
	1024×1024		2048×2048		4096×4096		8192×8192		16384×16384	
#cores	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc
1	0.00303	1.00	0.01166	1.00	0.04756	1.00	0.22143	1.00	0.97425	1.00
2	0.00159	1.90	0.00587	1.99	0.02408	1.98	0.12409	1.78	0.48822	2.00
4	0.00085	3.56	0.00299	3.90	0.01311	3.63	0.06421	3.45	0.25957	3.75
8	0.00050	6.11	0.00191	6.09	0.00835	5.70	0.03493	6.34	0.14260	6.83
16	0.00074	4.10	0.00239	4.88	0.01101	4.32	0.04478	4.94	0.18017	5.41
32	0.00111	2.73	0.00263	4.43	0.01055	4.51	0.05252	4.22	0.21648	4.50

Table 4: Performance results of our NUMA-unaware parallelization



	1024×1024		2048×2048		4096×4096		8192×8192		16384×16384	
#cores	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc
1	0.00306	1.00	0.01176	1.00	0.04777	1.00	0.22193	1.00	0.97619	1.00
2	0.00160	1.92	0.00587	2.00	0.02398	1.99	0.11857	1.87	0.50325	1.94
4	0.00100	3.06	0.00300	3.92	0.01277	3.74	0.06344	3.50	0.25939	3.76
8	0.00071	4.34	0.00203	5.79	0.00847	5.64	0.03549	6.25	0.14251	6.85
16	0.00063	4.90	0.00135	8.68	0.00460	10.38	0.01850	12.00	0.07259	13.45
32	0.00132	2.32	0.00069	17.11	0.00257	18.60	0.01072	20.71	0.04270	22.86

Table 5: Performance results of our NUMA-aware parallelization



THANK YOU SO MUCH FOR YOUR ATTENTION