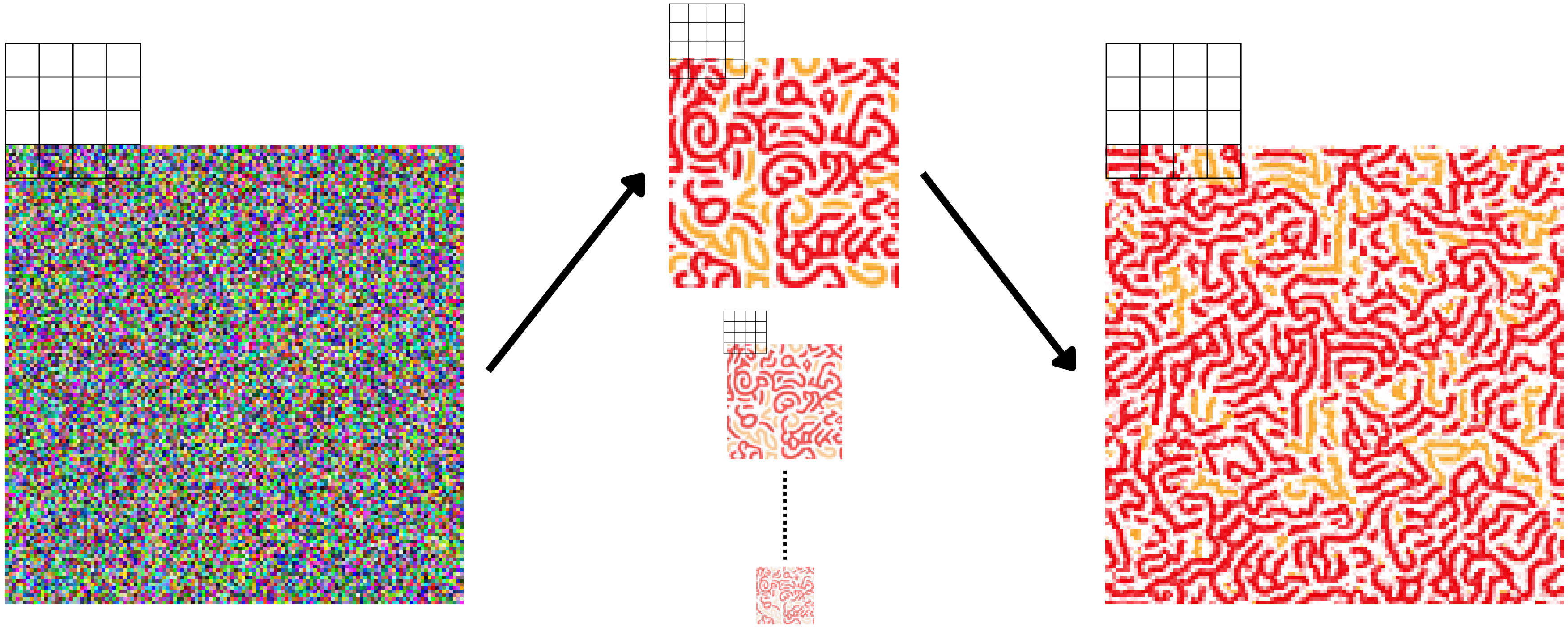
Fast Texture Synthesis

Sofia d’Atri
Arseni Loika
Alessandro Zanzi
Alessandro Cravioglio

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algorithm I - How it Works

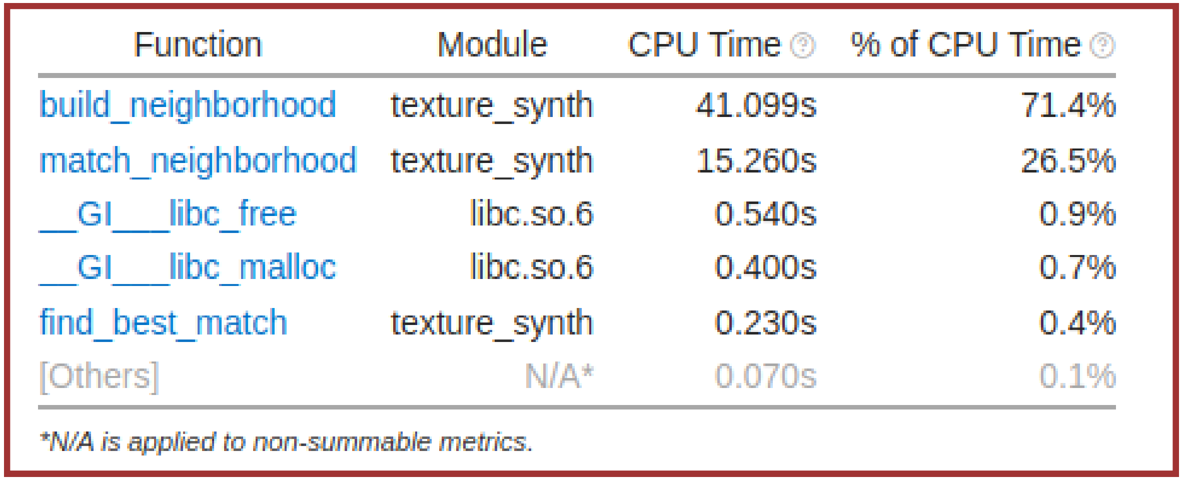
- **Texture Synthesis**
  - **Inputs**: Texture + random noise
  - **Output**: Texture synthesised to the dimensions of the random noise
  - **Possible Applications:** Texture filling, image/video reconstruction, visual motion synthesis, modeling geometric details



Algorithm II - Asymptotic Cost

Build neighborhood	**Gauss Pyramid**	**Find Best Match**	**Texture Synthesis**
O(n0+n1)	$O(w^2)$	$O(w^2 * (n0+n1))$	$\mathbf{O(L*w^4 * (n0+n1))}$

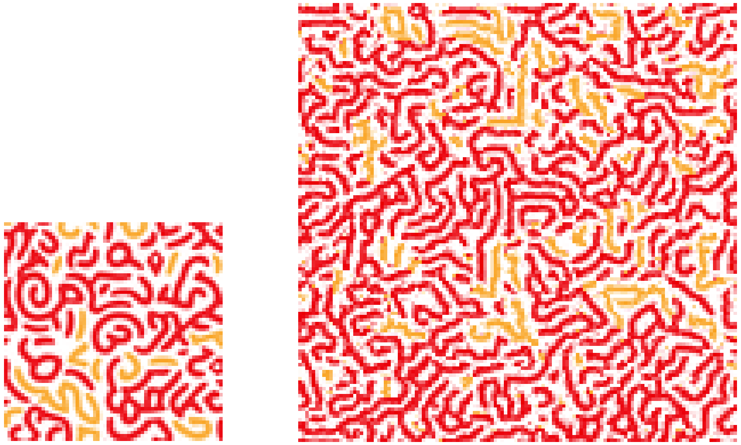
- **n0, n1** = neighborhood sizes, **w** = input image width, **L** = num levels
- **Bottlenecks**: build_neighborhood, match_neighborhood (vtune)



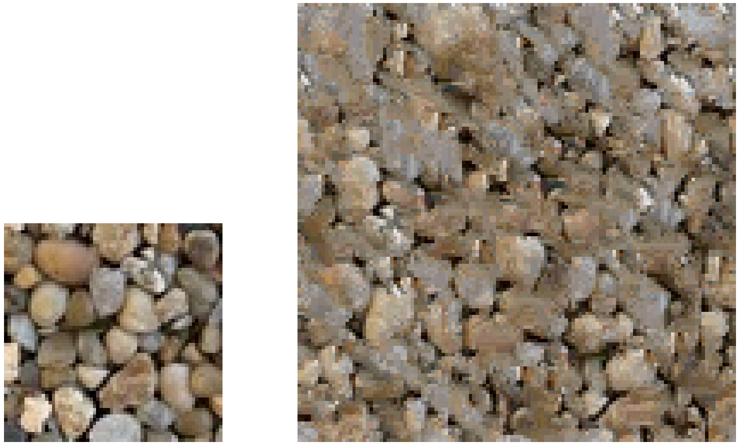
Function	Module	CPU Time	% of CPU Time
build_neighborhood	texture_synth	41.099s	71.4%
match_neighborhood	texture_synth	15.260s	26.5%
__GI___libc_free	libc.so.6	0.540s	0.9%
__GI___libc_malloc	libc.so.6	0.400s	0.7%
find_best_match	texture_synth	0.230s	0.4%
[Others]	N/A*	0.070s	0.1%

**N/A is applied to non-summable metrics.*

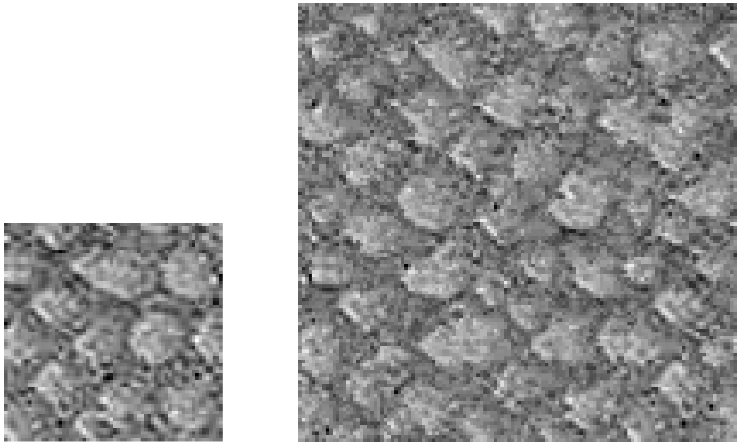
Results Example



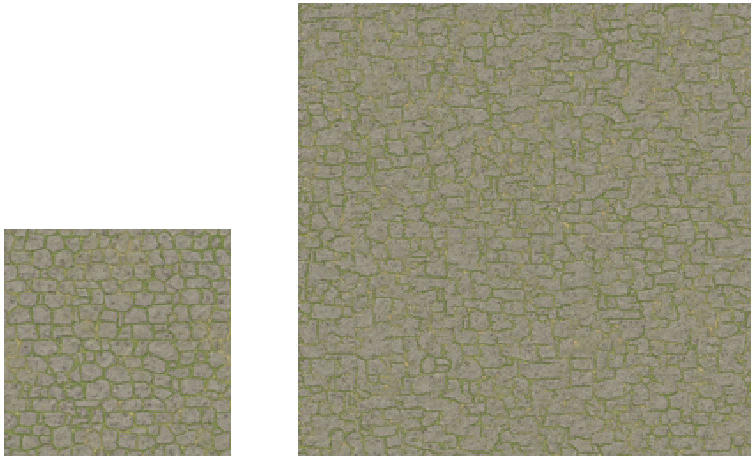
64x64 → 128x128



64x64 → 128x128



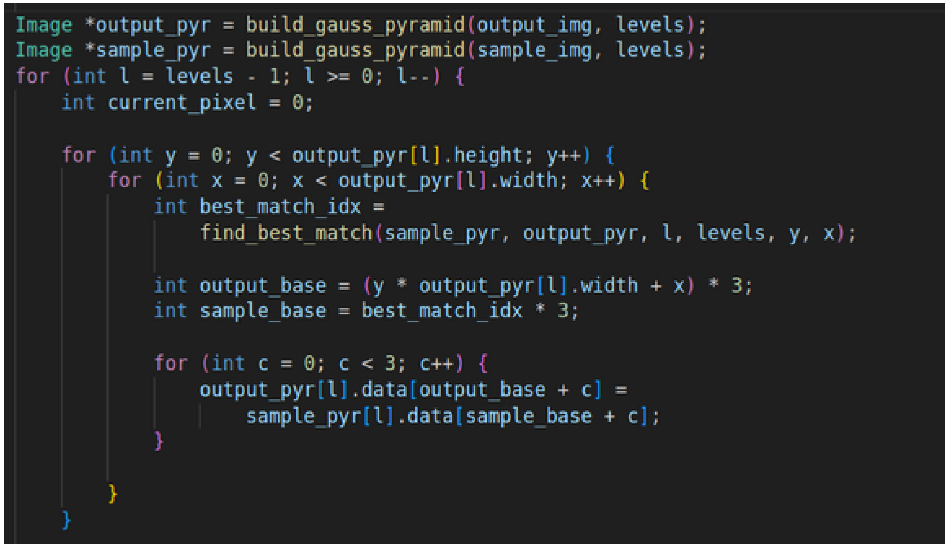
64x64 → 128x128



128x128 → 256x256

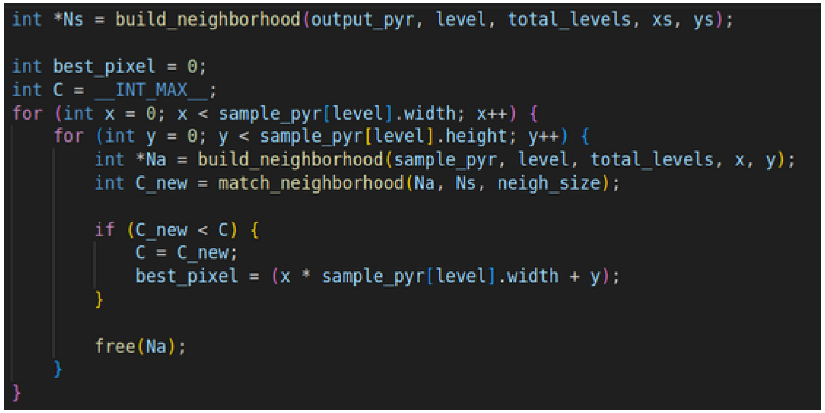
Baseline

- **Naive implementation**: For each output pixel, compare its neighborhood to the every other one on the corresponding pyramid level → then assign
- **Input format**: NxN image, where N is a power of 2
- **Neighborhood size**: 12x9 (from article)
- **Hardware**: Intel Core i7 10850H, 2.7GHz Comet Lake
- **Max theoretical performance** = 4 integer ops / cycle



```
Image *output_pyr = build_gauss_pyramid(output_img, levels);
Image *sample_pyr = build_gauss_pyramid(sample_img, levels);
for (int l = levels - 1; l >= 0; l--) {
    int current_pixel = 0;

    for (int y = 0; y < output_pyr[l].height; y++) {
        for (int x = 0; x < output_pyr[l].width; x++) {
            int best_match_idx =
                find_best_match(sample_pyr, output_pyr, l, levels, y, x);

            int output_base = (y * output_pyr[l].width + x) * 3;
            int sample_base = best_match_idx * 3;

            for (int c = 0; c < 3; c++) {
                output_pyr[l].data[output_base + c] =
                    sample_pyr[l].data[sample_base + c];
            }

        }
    }
```

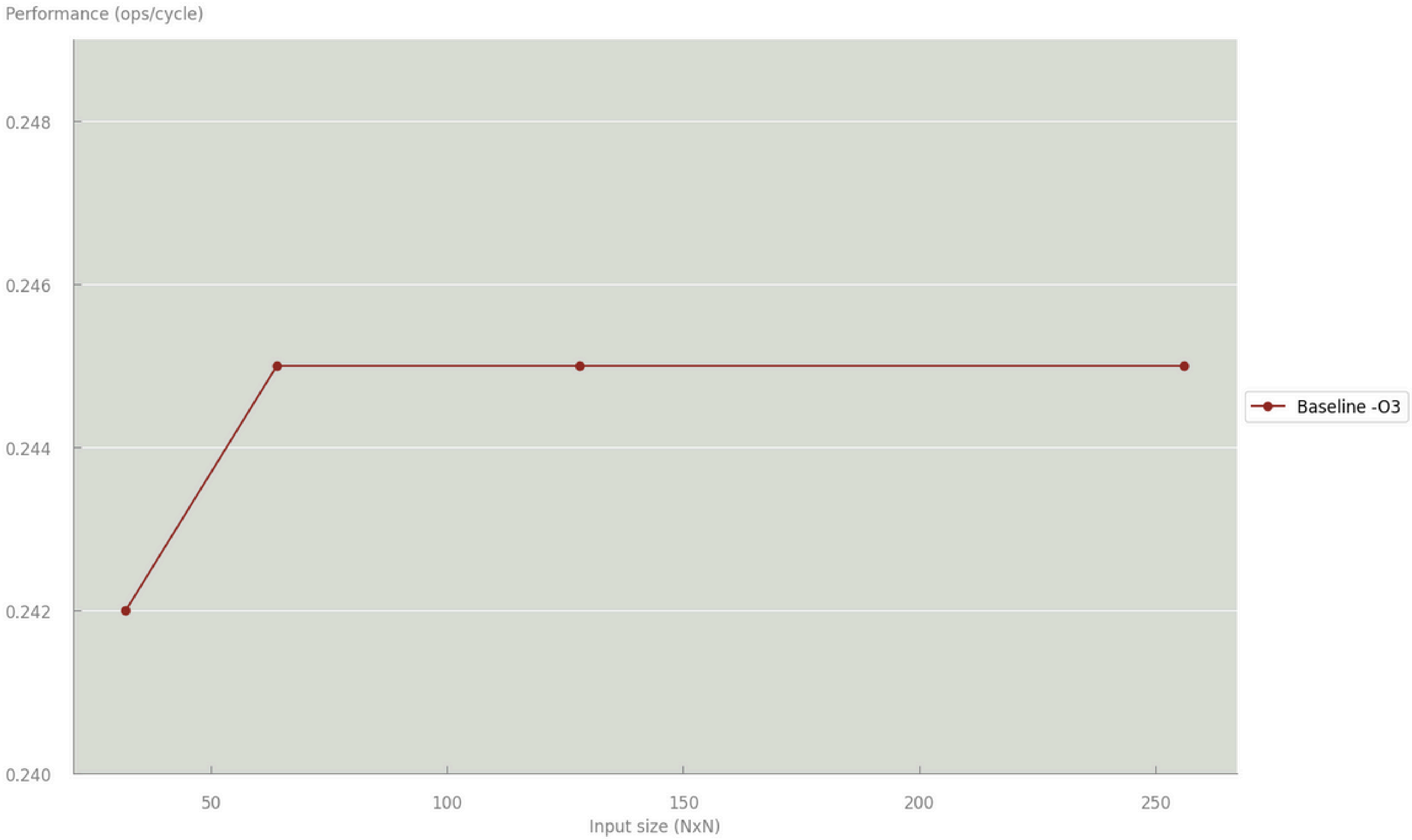
texture_synthesis()



```
int *Ns = build_neighborhood(output_pyr, level, total_levels, xs, ys);

int best_pixel = 0;
int C = __INT_MAX__;
for (int x = 0; x < sample_pyr[level].width; x++) {
    for (int y = 0; y < sample_pyr[level].height; y++) {
        int *Na = build_neighborhood(sample_pyr, level, total_levels, x, y);
        int C_new = match_neighborhood(Na, Ns, neigh_size);

        if (C_new < C) {
            C = C_new;
            best_pixel = (x * sample_pyr[level].width + y);
        }

        free(Na);
    }
}
```

find_best_match()

Baseline Plot



Cost Analysis

- **Measure:** integer operations (Gops)
- Index operations discarded

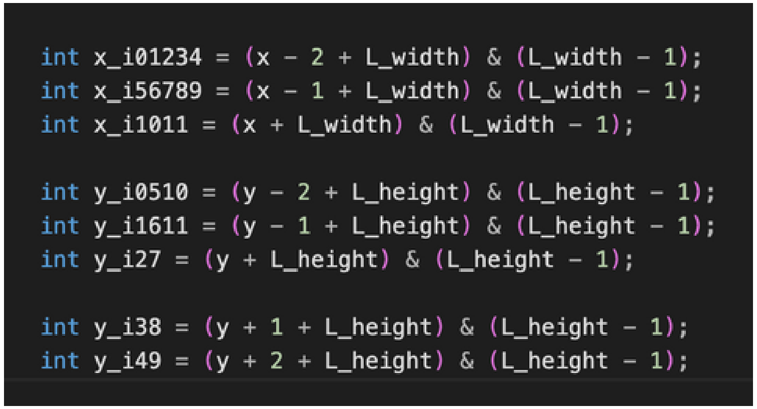
input size	Baseline	Version 1 (v1)	Version 2 (fastest_scalar)	Version 4 (vec)	Version 5 (acceleration)
32 x 32	1.89	1.84	1.49	2.41	1.71
64 x 64	32.29	29.43	23.84	38.52	27.35
128 x 128	484.47	470.47	381.40	616.18	437.52
256 x 256	7,751.16	7,531,27	6,102.09	9,858.40	6,999.99

Optimizations I (scalar)

- **Version 1 (neighborhood_v1) → 1.67x speedup**
    - build_neighborhood: scalar replacement, loop unrolling, strength reduction
    - match_neighborhood: total unrolling, multiple precomputation
    - Didn’t work so well: pointer arithmetic optimizations, precomputing some arithmetic operations

- **Version 2 (neighborhood_fastest_scalar) → 1.72x speedup**
    - The above + memory reuse

- **Version 3 (inlined) → 2.50x speedup**
    - All functions inlined



```
int x_i01234 = (x - 2 + L_width) & (L_width - 1);
int x_i56789 = (x - 1 + L_width) & (L_width - 1);
int x_i1011 = (x + L_width) & (L_width - 1);

int y_i0510 = (y - 2 + L_height) & (L_height - 1);
int y_i1611 = (y - 1 + L_height) & (L_height - 1);
int y_i27 = (y + L_height) & (L_height - 1);

int y_i38 = (y + 1 + L_height) & (L_height - 1);
int y_i49 = (y + 2 + L_height) & (L_height - 1);
```

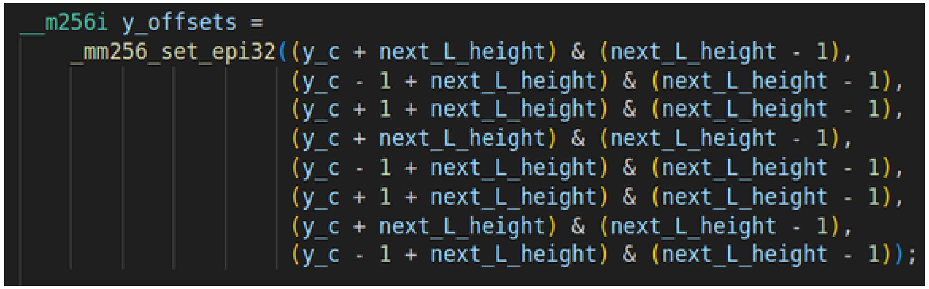
Precomputing of coordinates

Optimizations II (vectorized)

- **Version 4 (neighborhood_vec) → 1.94x speedup**
  - build_neighborhood: vectorized calculation with coordinates of each pixel's neighborhood
  - match_neighborhood: total unrolling of the loop, 8 ints at a time

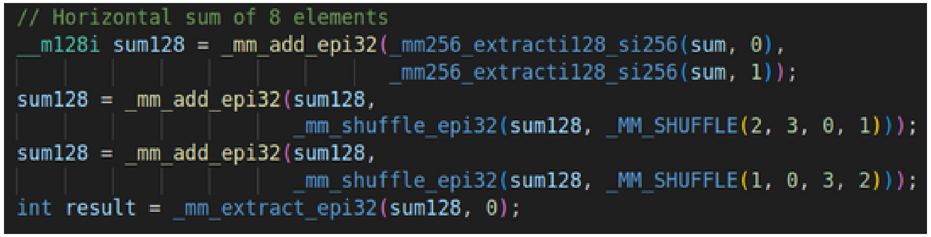
- **Version 5 (acceleration) → 3.14x speedup**
  - match_neighborhood: vectorized loop, 8 ints at a time

- **Version 6 (inlined) → 3.76x speedup**
  - The above inlined



```
__m256i y_offsets =
    _mm256_set_epi32((y_c + next_L_height) & (next_L_height - 1),
                     (y_c - 1 + next_L_height) & (next_L_height - 1),
                     (y_c + 1 + next_L_height) & (next_L_height - 1),
                     (y_c + next_L_height) & (next_L_height - 1),
                     (y_c - 1 + next_L_height) & (next_L_height - 1),
                     (y_c + 1 + next_L_height) & (next_L_height - 1),
                     (y_c + next_L_height) & (next_L_height - 1),
                     (y_c - 1 + next_L_height) & (next_L_height - 1));
```

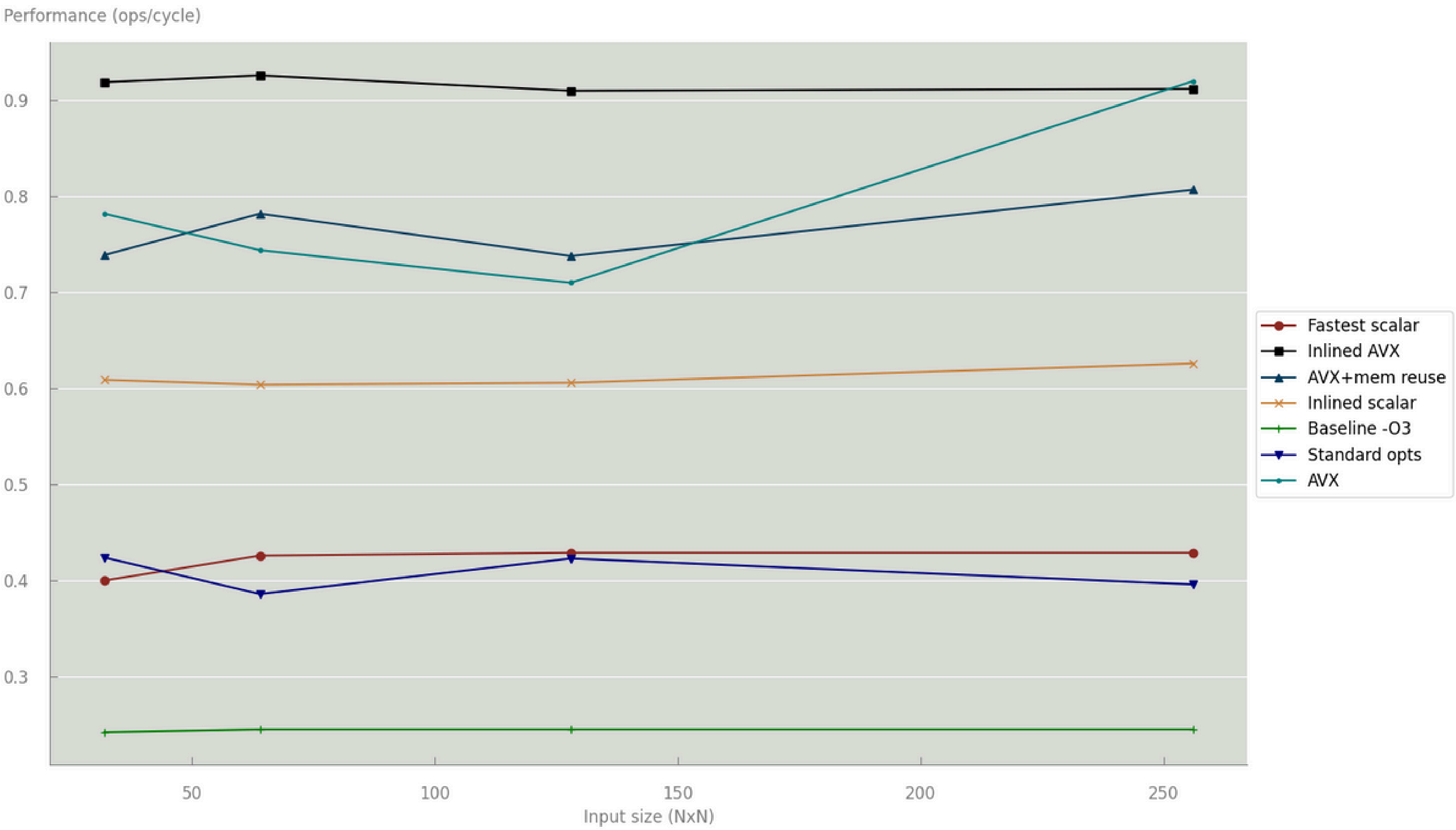
build_neighborhood: vectorization of y-coordinates

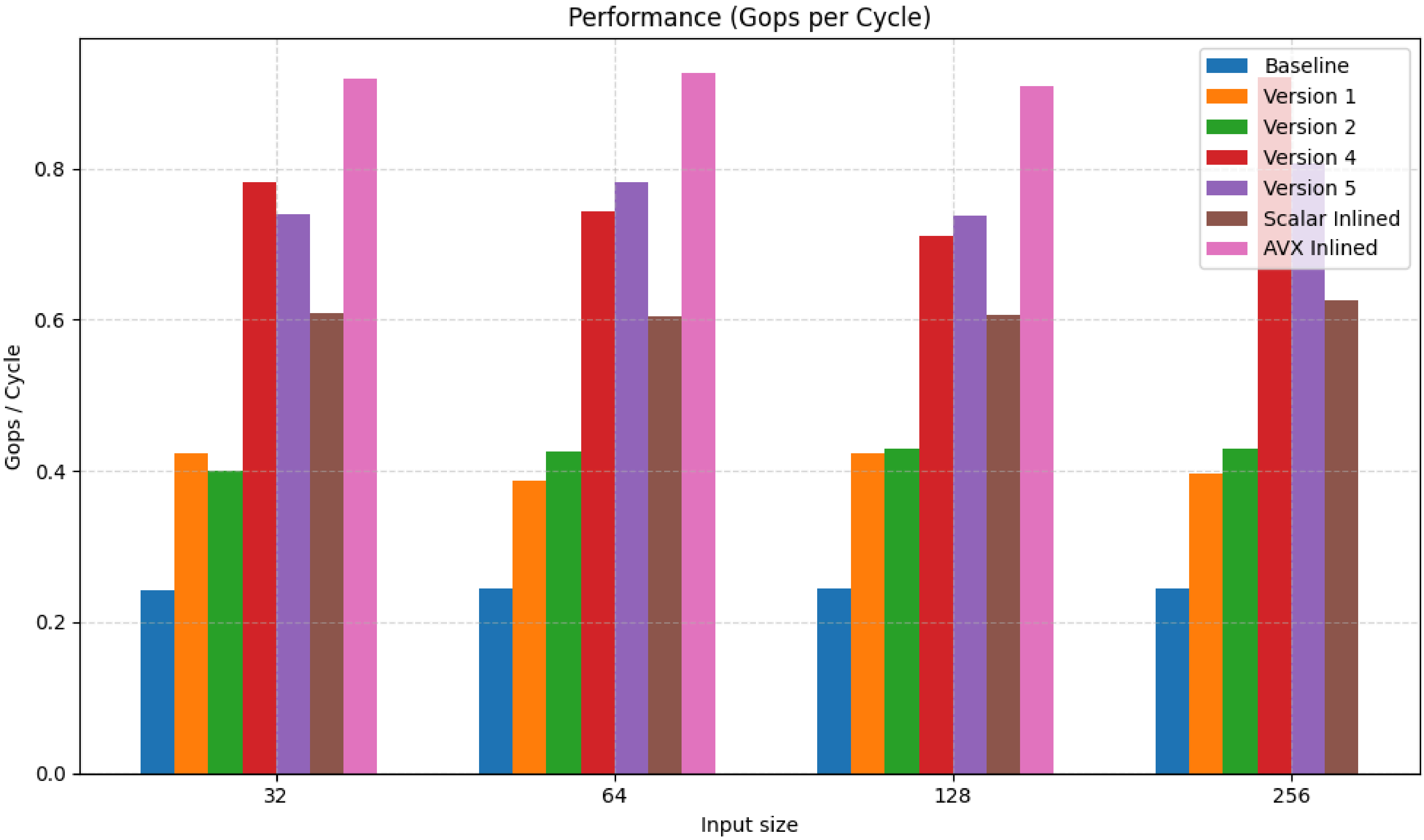


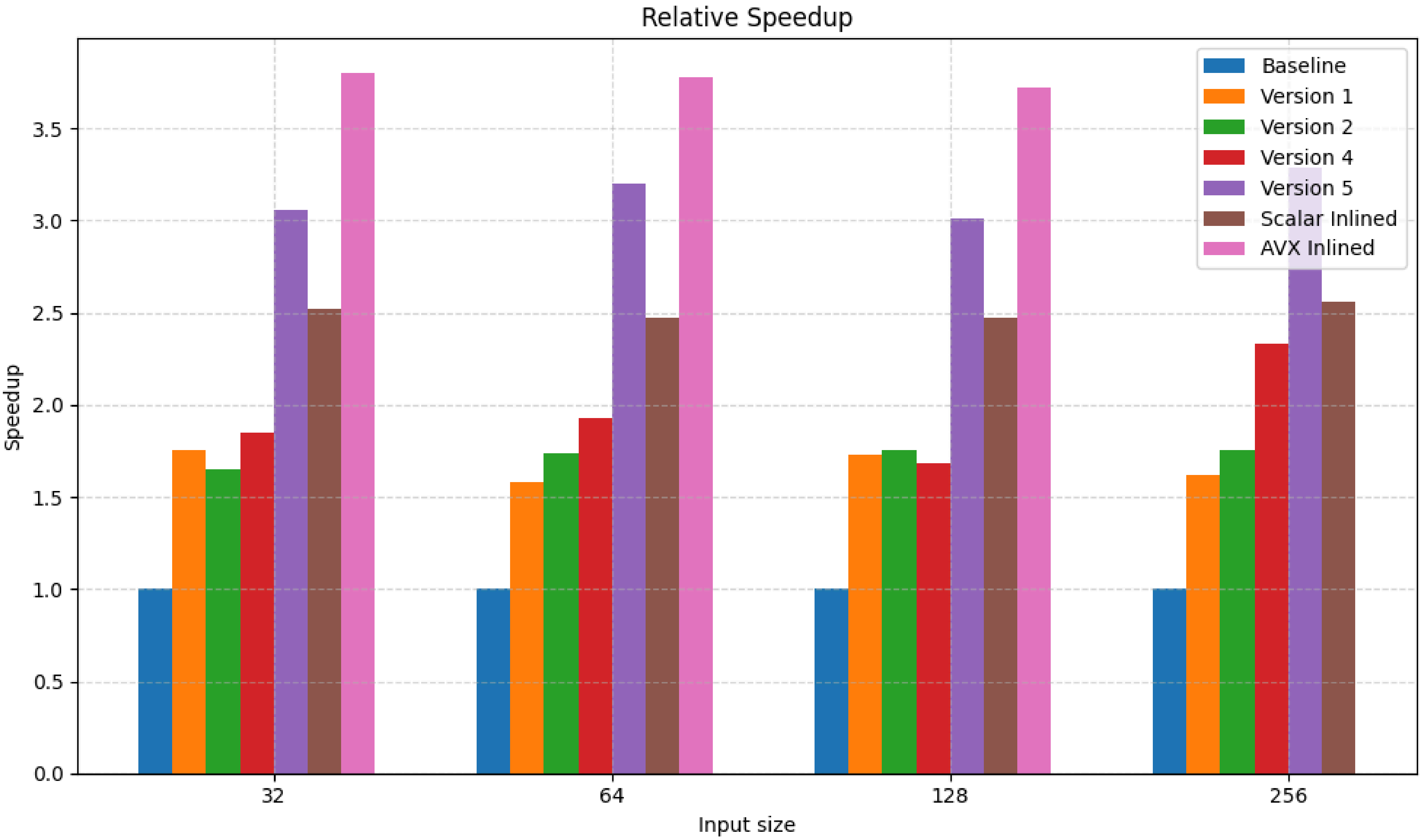
```
// Horizontal sum of 8 elements
__m128i sum128 = _mm_add_epi32(_mm256_extracti128_si256(sum, 0),
                               _mm256_extracti128_si256(sum, 1));
sum128 = _mm_add_epi32(sum128,
                       _mm_shuffle_epi32(sum128, _MM_SHUFFLE(2, 3, 0, 1)));
sum128 = _mm_add_epi32(sum128,
                       _mm_shuffle_epi32(sum128, _MM_SHUFFLE(1, 0, 3, 2)));
int result = _mm_extract_epi32(sum128, 0);
```

match_neighborhood: squared sum vectorization

Optimizations Plot




Performance (Gops per Cycle)
Gops / Cycle
0.0
0.2
0.4
0.6
0.8
32
64
128
256
Input size
Baseline
Version 1
Version 2
Version 4
Version 5
Scalar Inlined
AVX Inlined


Relative Speedup
Speedup
Input size
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
32
64
128
256
Baseline
Version 1
Version 2
Version 4
Version 5
Scalar Inlined
AVX Inlined

Q&A