Faster Than Memcpy

Mateusz Brzeszcz
Software Engineer @  Adaptive Vision
mbrzeszcz@adaptive-vision.com

Measure, measure, measure

Let’s copy some memory to establish some baseline for benchmarking

```
void copy_plain(uint8_t* dst, const uint8_t* src, int bytes)
{
      for (int i = 0; i < bytes; ++i) dst[i] = src[i];
}
```

Environment:

Compiler: Microsoft (R) C/C++ Optimizing Compiler Version 19.23.28106.4 for x64
CPU: Intel i5-7500 @ 3.4GHz
Datasize: 3MB

Compile and run:

copy_plain: 1.010 ms

Now, let's do some processing!

```
void add_3(uint8_t* dst, const uint8_t* src, int bytes)
{
      for (int i = 0; i < bytes; ++i) dst[i] = src[i] + 3;
}
```

Test and compare to baseline

copy_plain: 1.010 ms
add_3: 0.195 ms

Why **T**hat **F**ast?

5 times faster than just copying memory?

Diving into assembly (only loops)

```
$LL8@copy_plain:
        movzx   eax, BYTE PTR [rdx+rcx]
        mov     BYTE PTR [rcx], al
        lea     rcx, QWORD PTR [rcx+1]
        sub     r9, 1
        jne     SHORT $LL8@copy_plain
```

```
$LL4@add_3:
        movdqu  xmm0, XMMWORD PTR [rdx+rcx]
        add     ebx, 64
        movdqu  xmm1, XMMWORD PTR [rdx+rcx+16]
        paddb   xmm0, xmm2
        movdqu  XMMWORD PTR [rcx], xmm0
        movdqu  xmm0, XMMWORD PTR [rdx+rcx+32]
        paddb   xmm1, xmm2
        movdqu  XMMWORD PTR [rcx+16], xmm1
        movdqu  xmm1, XMMWORD PTR [rdx+rcx+48]
        paddb   xmm0, xmm2
        movdqu  XMMWORD PTR [rcx+32], xmm0
        paddb   xmm1, xmm2
        movdqu  XMMWORD PTR [rcx+48], xmm1
        add     rcx, 64
        mov     rax, rcx
        sub     rax, r10
        cmp     rax, r8
        jl      SHORT $LL4@add_3
...
$LL8@add_3:
        movzx   ecx, BYTE PTR [rax+r9]
        lea     rax, QWORD PTR [rax+1]
        add     cl, 3
        mov     BYTE PTR [rax-1], cl
        sub     rdx, 1
        jne     SHORT $LL8@add_3
```

Why not vectorized?

/Qvec-report:2

FasterThanMemcpy.cpp(49) : info C5002: loop not vectorized due to reason '1300'

...and people complain about gcc error messages when compiling templates.

What does the doc say?

```
void code_1300(int *A, int *B)
{
    // Code 1300 is emitted when the compiler detects that there is
    // no computation in the loop body.

    for (int i=0; i<1000; ++i)
    {
        A[i] = B[i]; // Do not vectorize, instead emit memcpy
    }
}
```

Except that... no std::memcpy was emitted.

Result - the “algorithm” was ok, but the benchmark baseline was flawed.

What about clang?

clang version 8.0.1 (tags/RELEASE_801/final)
Target: i686-pc-windows-msvc

copy_plain: 0.191 ms
add_3: 0.196 ms

Making vectorized loops behave

A simple algorithm - multiplication by 2

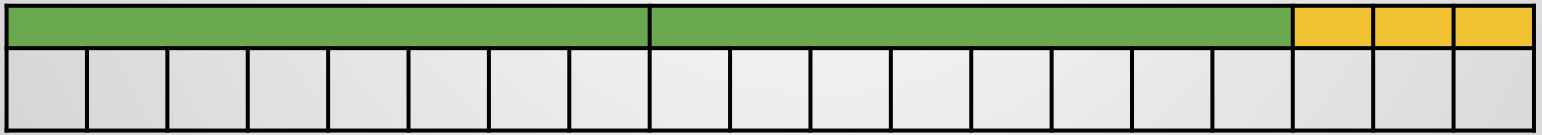
```
void mult2_simple(Image& dst, Image& src)
{
...
      for (int y = 0; y < height; ++y)
      {
            auto srcp = src.row_ptr(y);
            auto dstp = dst.row_ptr(y);

            for (int x = 0; x < pitch; ++x)
            {
                  dstp[x] = 2 * srcp[x];
            }
      }
}
```

Note: Compiler will vectorize this

A standard loop vectorization method

```
template<typename OPER>
void process_buffer_standard(uint8_t* dst, uint8_t* src, int length, OPER op)
{
      const int limit = length - OPER::VECT_SIZE + 1;
      int offset = 0;
      for (; offset < limit; offset += OPER::VECT_SIZE)
      {
            op.process_vector(dst, src, offset);
      }
      for (; offset < length; offset += 1)
      {
            op.process_scalar(dst, src, offset);
      }
}
```



A standard loop vectorization method

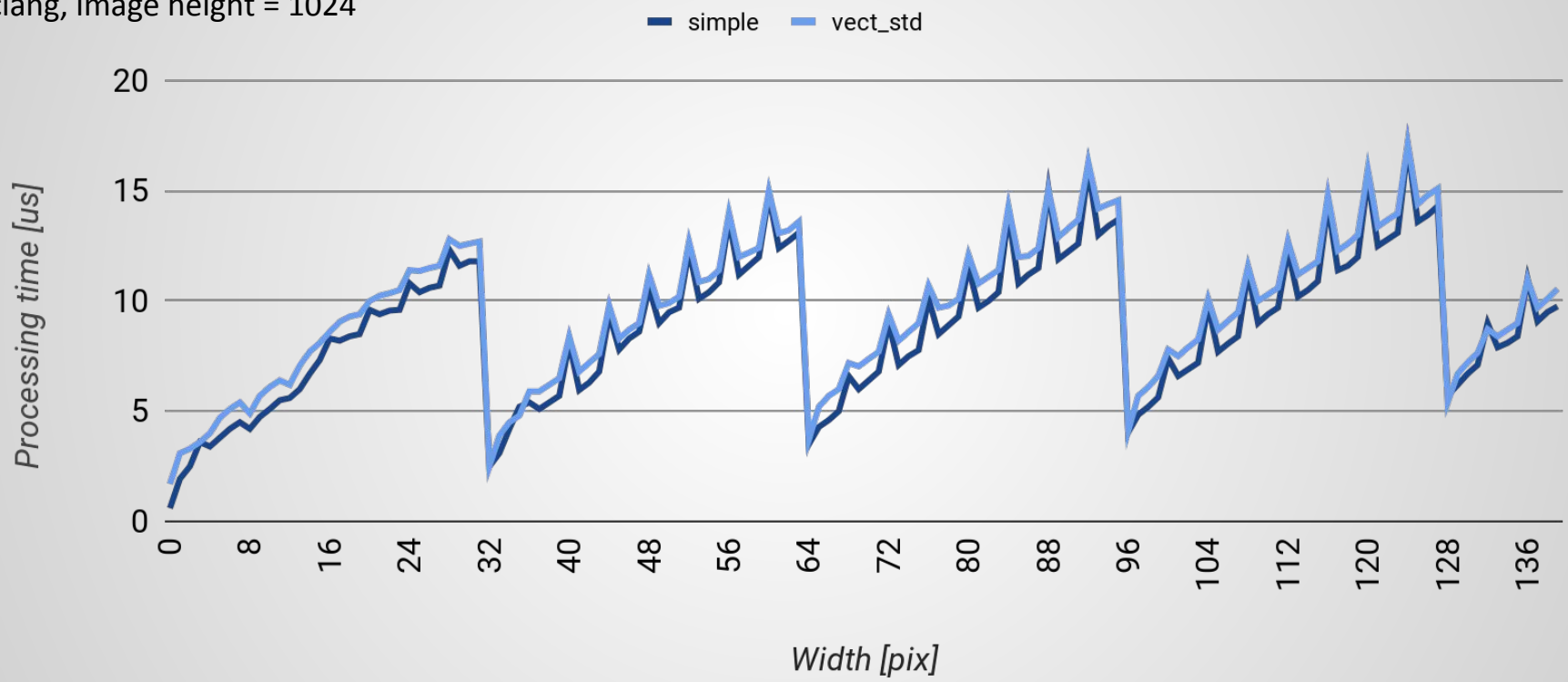
```
struct Mult2Op
{
	const static int VECT_SIZE = 2 * sizeof(__m128i);

	void process_vector(uint8_t* dst, uint8_t* src, int offset) const
	{
		auto val = _mm_loadu_si128((__m128i*)(src + offset));
		auto val2 = _mm_loadu_si128((__m128i*)(src + offset + sizeof(__m128i)));
		val = _mm_add_epi8(val, val);
		val2 = _mm_add_epi8(val2, val2);
		_mm_storeu_si128((__m128i*)(dst + offset), val);
		_mm_storeu_si128((__m128i*)(dst + offset + sizeof(__m128i)), val2);
	}

	void process_scalar(uint8_t* dst, uint8_t* src, int offset) const
	{
		dst[offset] = 2 * src[offset];
	}
};
```

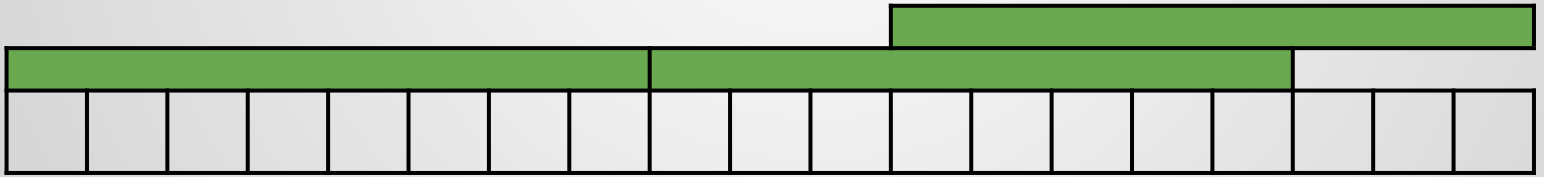
Testing:

clang, image height = 1024



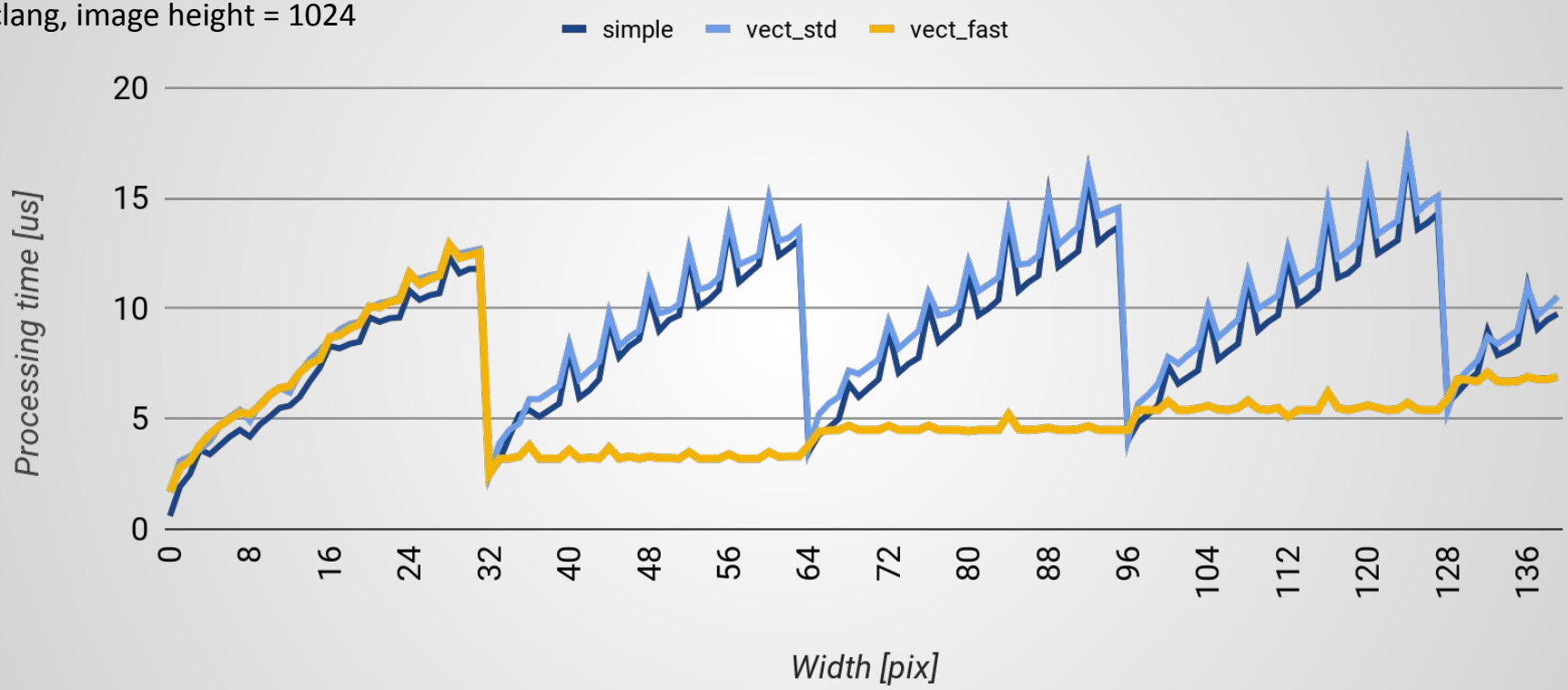
A faster loop vectorization method

```
...
      for (; offset < limit; offset += OPER::VECT_SIZE)
      {
            op.process_vector(dst, src, offset);
      }
      if (offset < length)
      {
            if (length > OPER::VECT_SIZE)
            {
                  op.process_vector(dst, src, length - OPER::VECT_SIZE);
            }
            ...
      }
```



Testing:

clang, image height = 1024



A real-life example - image erosion

Vertical erosion
Image height = 1024
Kernel height = 7

Width: **1020**, std: **350.474 us**, fast: **171.817 us**
Width: **1024**, std: **170.950 us**, fast: **170.670 us**
Width: **1028**, std: **233.753 us**, fast: **174.308 us**

Remarks

- Use case: writing a part of a larger image.
  - If possible, just pad the memory.
  - If possible, treat images as continuous buffer.
- Redundant processing. Won’t work for accumulators.

Curious case of false sharing

Get a list of bright pixels

```
struct Point { int x, y; };

std::vector<Point> get_bright_pixels(Image& img, int threshold)
{
      std::vector<Point> ans;
      ans.reserve(...);
      ...

      for (int y = 0; y < height; y++)
      {
            auto row = img.row_ptr(y);
            for (int x = 0; x < width; x++)
            {
                  if (row[x] >= threshold) ans.push_back({ x,y });
            }
      }
      return ans;
}
```

Accelerate!

```
std::vector<Point> get_bright_pixels_parallel(Image& img, int threshold)
{
      std::array<std::vector<Point>, threads> partial_ans;
      for (auto& a : partial_ans) a.reserve(...);
      ...

#pragma omp parallel for schedule(static) num_threads(threads)
      for (int t = 0; t < threads; t++)
      {
            auto& ans = partial_ans[t];
            int width = img.w;
            for (int y = t * h_chunk; y < (t + 1) * h_chunk; y++)
            {
                  auto row = img.row_ptr(y);
                  for (int x = 0; x < width; x++)
                  {
                        if (row[x] >= threshold) ans.push_back({ x,y });
                  }
            }
      }
      return merge_vectors<Point>(partial_ans);
}
```

Measure...

1024x1024 randomized image
Threshold = 240

Single thread: **1030.598 us**
Parallel (4 threads): **1236.082 us**

Why?

```
std::array<std::vector<Point>, threads> partial_ans;
for (auto& a : partial_ans) a.reserve(...);
...
      auto& ans = partial_ans[t];
      ...
                  if (row[x] >= threshold) ans.push_back({ x,y });
```

- Separate vectors for every thread
- Preallocated memory

Why?

How does the

```
std::array<std::vector<Point>, 4> a;
```

looks like in the memory?

0x00 - 0x39	a[0]._Myfirst	a[0]._Mylast	a[0]._Myend	a[1]._Myfirst	a[1]._Mylast	a[1]._Myend	a[2]._Myfirst	a[2]._Mylast
0x40 - 0x79	a[2]._Myend	a[3]._Myfirst	a[3]._Mylast	a[3]._Myend				

False sharing occured at container metadata, not the data itself.

Accelerate properly

```
template <typename T>
struct alignas(64) aligned_vector : std::vector<T>
{};
...
      std::array<aligned_vector<Point>, threads> partial_ans;
      ...
```

0x00 - 0x39	a[0]. _Myfirst	a[0]. _Mylast	a[0]. _Myend				
0x40 - 0x79	a[1]. _Myfirst	a[1]. _Mylast	a[1]. _Myend				
0x80 - 0xbf	a[2]. _Myfirst	a[2]. _Mylast	a[2]. _Myend				
0xc0 - 0xff	a[3]. _Myfirst	a[3]. _Mylast	a[3]. _Myend				

Measure…

Single thread:	**1030.598 us**
Parallel (4 threads):	**1236.082 us**
Parallel padded (4th):	**306.105 us**

Properly parallelized code gives 3.36x speedup.

Thanks for coming