Subject: SSE2 and SVO optimization (Painter, memcpy....)
Posted by Tom1 on Mon, 27 Apr 2020 17:19:48 GMT

View Forum Message <> Reply to Message

Hi,

Here's an optimization for BufferPainter.

BufferPainter::Clear(RGBA) speed is improved by over 30 % with the following change in Painter/Render.cpp:

```
void BufferPainter::ClearOp(const RGBA& color)
{
// UPP::Fill(~*ip, color, ip->GetLength());
 FillRGBA(~*ip, color, ip->GetLength());
 ip->SetKind(color.a == 255 ? IMAGE_OPAQUE : IMAGE_ALPHA);
}
```

And in Painter/Fillers.h:

```
namespace Upp {

// Add the following line:
#define FillRGBA(a,b,c) memsetd((a),*(dword*)&(b),(c))

struct SolidFiller : Rasterizer::Filler {
```

This may be significant in some usage scenarios as it can currently take e.g. 4.5 milliseconds to clear a 4K ImageBuffer before drawing to it. This can now be reduced to 2.8 milliseconds.

Best regards,

Tom

EDIT: Changed code to use the newly optimized FillRGBA() found in Fillers.h. This can be found at:
   https://www.ultimatepp.org/forums/index.php?t=msg&th=110 11&goto=53752&#msg_53752

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 28 Apr 2020 08:12:35 GMT

View Forum Message <> Reply to Message

Tom1 wrote on Mon, 27 April 2020 19:19Hi,

Here's an optimization for BufferPainter.

BufferPainter::Clear(RGBA) speed is improved by over 30 % with the following change in

```
Painter/Render.cpp:
void BufferPainter::ClearOp(const RGBA& color)
{
// UPP::Fill(~*ip, color, ip->GetLength());
 FillRGBA(~*ip, color, ip->GetLength());
 ip->SetKind(color.a == 255 ? IMAGE_OPAQUE : IMAGE_ALPHA);
}
```

```
And in Painter/Fillers.h:
namespace Upp {

// Add the following line:
#define FillRGBA(a,b,c) memsetd((a),*(dword*)&(b),(c))

struct SolidFiller : Rasterizer::Filler {
```

This may be significant in some usage scenarios as it can currently take e.g. 4.5 milliseconds to clear a 4K ImageBuffer before drawing to it. This can now be reduced to 2.8 milliseconds.

Now this is really interesting. Fill for RGBA* is actually one that is optimized for filling huge blocks. I will need to do some benchmarks...

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 28 Apr 2020 08:20:53 GMT

View Forum Message <> Reply to Message

Current Fill(RGBA * assembler code

```
4000EEE0  cmp r8d,byte +0x10
4000EEE4  jl 0x14000ef13
4000EEE6  movd xmm0,edx
4000EEEA  pshufd xmm0,xmm0,0x0
4000EEEF  nop
4000EEF0  mov eax,r8d
4000EEF3  movdqu [rcx],xmm0
4000EEF7  movdqu [rcx+0x10],xmm0
4000EEFC  movdqu [rcx+0x20],xmm0
4000EF01  movdqu [rcx+0x30],xmm0
4000EF06  add rcx,byte +0x40
4000EF0A  lea r8d,[rax-0x10]
4000EF0E  cmp eax,byte +0x1f
4000EF11  jg 0x14000eef0
```

```
4000EF13  add r8d,byte -0x1
4000EF17  cmp r8d,byte +0xe
4000EF1B  ja 0x14000ef59
4000EF1D  lea r9,[rel 0x4000ef5c]
4000EF24  movsxd rax,dword [r9+r8*4]
4000EF28  add rax,r9
4000EF2B  jmp rax
4000EF2D  mov [rcx+0x38],edx
4000EF30  mov [rcx+0x34],edx
4000EF33  mov [rcx+0x30],edx
4000EF36  mov [rcx+0x2c],edx
4000EF39  mov [rcx+0x28],edx
4000EF3C  mov [rcx+0x24],edx
4000EF3F  mov [rcx+0x20],edx
4000EF42  mov [rcx+0x1c],edx
4000EF45  mov [rcx+0x18],edx
4000EF48  mov [rcx+0x14],edx
4000EF4B  mov [rcx+0x10],edx
4000EF4E  mov [rcx+0xc],edx
4000EF51  mov [rcx+0x8],edx
4000EF54  mov [rcx+0x4],edx
4000EF57  mov [rcx],edx
4000EF59  ret
```

and the central snippet from the memsetd variant....

```
40001565  movaps xmm0,[rel 0x402c60a0]
4000156C  nop dword [rax+0x0]
40001570  movups [rsi+rdx*4],xmm0
40001574  movups [rsi+rdx*4+0x10],xmm0
40001579  movups [rsi+rdx*4+0x20],xmm0
4000157E  movups [rsi+rdx*4+0x30],xmm0
40001583  movups [rsi+rdx*4+0x40],xmm0
40001588  movups [rsi+rdx*4+0x50],xmm0
4000158D  movups [rsi+rdx*4+0x60],xmm0
40001592  movups [rsi+rdx*4+0x70],xmm0
40001597  movups [rsi+rdx*4+0x80],xmm0
4000159F  movups [rsi+rdx*4+0x90],xmm0
400015A7  movups [rsi+rdx*4+0xa0],xmm0
400015AF  movups [rsi+rdx*4+0xb0],xmm0
400015B7  movups [rsi+rdx*4+0xc0],xmm0
400015BF  movups [rsi+rdx*4+0xd0],xmm0
400015C7  movups [rsi+rdx*4+0xe0],xmm0
400015CF  movups [rsi+rdx*4+0xf0],xmm0
400015D7  add rdx,byte +0x40
400015DB  add rdi,byte +0x8
```

```
400015DF  jnz 0x140001570
```

Interesting...

Benchmarking code

```
#include <CtrlLib/CtrlLib.h>

using namespace Upp;

GUI_APP_MAIN
{
 Color c = Red();

 int len = 4000 * 2000;

 Buffer<RGBA> b(len);

 for(int i = 0; i < 1000; i++) {
  {
   RTIMING("memsetd");
   memsetd(b, *(dword*)&(c), len);
  }
  {
   RTIMING("Fill");
   Fill(b, c, len);
  }
 }
}
```

CLANGx64, 2700x

```
TIMING Fill          :  2.73 s  -  2.73 ms ( 2.73 s  / 1000 ), min:  2.00 ms, max:  4.00 ms, nesting: 0 - 1000
TIMING memsetd       :  2.78 s  -  2.78 ms ( 2.78 s  / 1000 ), min:  2.00 ms, max:  5.00 ms, nesting: 0 - 1000
```

MSBT19x64

```
TIMING Fill          :  2.89 s  -  2.89 ms ( 2.89 s  / 1000 ), min:  2.00 ms, max:  5.00 ms, nesting: 0 - 1000
TIMING memsetd       :  2.90 s  -  2.90 ms ( 2.90 s  / 1000 ), min:  2.00 ms, max:  5.00 ms, nesting: 0 - 1000
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 28 Apr 2020 08:27:28 GMT
View Forum Message <> Reply to Message

Hi,

Benchmarking and tuning is exactly what I did through yesterday (and beyond). I worked with both CLANGx64 and MSBT19x64. I worked out a bunch of optimized fillers until it turned out that memsetd() wins easily on large blocks and mostly on smaller blocks too. Especially on MSBT19x64 there does not seem to be a way to beat memsetd(). On CLANGx64 small transfer of one or two items was slightly faster, but on larger blocks memsetd() won again. Interestingly, CLANGx64 was a lot faster than MSBT19x64 for any of my own block transfer attempts, but still could not beat memsetd().

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 28 Apr 2020 08:33:30 GMT
View Forum Message <> Reply to Message

I guess it might be CPU related... ?

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 28 Apr 2020 09:10:47 GMT
View Forum Message <> Reply to Message

Hm, MacOS 2,3 GHz Intel Core i5

```
TIMING Fill          :  1.52 s  -  1.52 ms ( 1.52 s  / 1000 ), min:  1.00 ms, max:  2.00 ms, nesting: 0 - 1000
TIMING memsetd       :  1.53 s  -  1.53 ms ( 1.53 s  / 1000 ), min:  1.00 ms, max: 12.00 ms, nesting: 0 - 1000
```

That's quite weird...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 28 Apr 2020 09:17:16 GMT
View Forum Message <> Reply to Message

Hi,

Yes, CPU is likely the major player here. I took the liberty to modify TimingInspector to get finer granularity for timing using usecs(). The modified testcase can be found below. I get the following results on my Core i7 + Windows 10 professional x64. Now we can focus on the best round 'min:' to better avoid other tasks' effect. As you can see memsetd on MSBT19x64 is quite amazing performer.

MSBT19x64, Intel Core i7:

```
TIMING memsetd       :  1.45 s  -  1.45 ms ( 1.45 s  / 1000 ), min:  1.15 ms, max:  5.25 ms, nesting: 0 - 1000
TIMING Fill          :  3.73 s  -  3.73 ms ( 3.73 s  / 1000 ), min:  3.25 ms, max:  9.92 ms, nesting: 0 - 1000
```

CLANGx64, Intel Core i7:

```
TIMING memsetd       :  3.85 s  -  3.85 ms ( 3.85 s  / 1000 ), min:  3.35 ms, max: 10.36 ms, nesting: 0 - 1000
TIMING Fill          :  3.87 s  -  3.87 ms ( 3.87 s  / 1000 ), min:  3.38 ms, max: 11.33 ms, nesting: 0 - 1000
```

I guess that in my larger program the optimizations did not work this well as the Fill would have performed at around 5 ms level for this size of a buffer.

Anyway here's the modified benchmark.

```
#include <CtrlLib/CtrlLib.h>

using namespace Upp;

class UTimingInspector {
protected:
 static bool active;

 const char *name;
 int        call_count;
 int64      total_time;
 int64      min_time;
 int64      max_time;
 int        max_nesting;
 int        all_count;
 StaticMutex mutex;

public:
 UTimingInspector(const char *name = NULL); // Not String !!!
 ~UTimingInspector();

 void   Add(dword time, int nesting);
 String Dump();

 class Routine {
 public:
  Routine(UTimingInspector& stat, int& nesting)
  : nesting(nesting), stat(stat) {
   start_time = usecs();
   nesting++;
  }

  ~Routine() {
   nesting--;
   stat.Add(start_time, nesting);
  }

 protected:
  int64 start_time;
  int& nesting;
  UTimingInspector& stat;
 };

 static void Activate(bool b)                    { active = b; }
};

bool UTimingInspector::active = true;

static UTimingInspector s_zero; // time of Start / End without actual body to measure

UTimingInspector::UTimingInspector(const char *_name) {
 name = _name ? _name : "";
 all_count = call_count = max_nesting = min_time = max_time = total_time = 0;
 static bool init;
 if(!init) {
#if defined(PLATFORM_WIN32) && !defined(PLATFORM_WINCE)
  timeBeginPeriod(1);
#endif
  init = true;
 }
}

UTimingInspector::~UTimingInspector() {
 if(this == &s_zero) return;
 Mutex::Lock __(mutex);
 StdLog() << Dump() << "\r\n";
}

void UTimingInspector::Add(dword time, int nesting)
```

```
{
 time = usecs() - time;
 Mutex::Lock __(mutex);
 if(!active) return;
 all_count++;
 if(nesting > max_nesting)
  max_nesting = nesting;
 if(nesting == 0) {
  total_time += time;
  if(call_count++ == 0)
   min_time = max_time = time;
  else {
   if(time < min_time)
    min_time = time;
   if(time > max_time)
    max_time = time;
  }
 }
}

String UTimingInspector::Dump() {
 Mutex::Lock __(mutex);
 String s = Sprintf("TIMING %-15s: ", name);
 if(call_count == 0)
  return s + "No active hit";
 ONCELOCK {
  int w = GetTickCount();
  while(GetTickCount() - w < 200) { // measure profiling overhead
   thread_local int nesting = 0;
   UTimingInspector::Routine __(s_zero, nesting);
  }
 }
 double tm = max(0.0, double(total_time) / call_count / 1000000 -
                double(s_zero.total_time) / s_zero.call_count / 1000000);
 return s
       + timeFormat(tm * call_count)
       + " - " + timeFormat(tm)
       + " (" + timeFormat((double)total_time  / 1000000) + " / "
       + Sprintf("%d )", call_count)
    + ", min: " + timeFormat((double)min_time / 1000000)
    + ", max: " + timeFormat((double)max_time / 1000000)
    + Sprintf(", nesting: %d - %d", max_nesting, all_count);
}

#define RUTIMING(x) \
 static UTimingInspector COMBINE(sTmStat, __LINE__)(x); \
 static thread_local int COMBINE(sTmStatNesting, __LINE__); \
 UTimingInspector::Routine COMBINE(sTmStatR, __LINE__)(COMBINE(sTmStat, __LINE__),
COMBINE(sTmStatNesting, __LINE__))

GUI_APP_MAIN
{
 Color c = Red();

 int len = 4000 * 2000;

 Buffer<RGBA> b(len);

 for(int i = 0; i < 1000; i++) {
  {
   RUTIMING("Fill");
   Fill(b, c, len);
  }
  {
   RUTIMING("memsetd");
   memsetd(b, *(dword*)&(c), len);
  }
 }
}
```

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Oblivion on Tue, 28 Apr 2020 09:27:56 GMT
View Forum Message <> Reply to Message

Hello,

A quick test on an older AMD FX 6100, six core processor. 3.2 GHZ (naturally, it is slower):

```
// GCC (x64, latest ver.)
TIMING Fill          :  7,53 s  -  7,53 ms ( 7,53 s  / 1000 ), min:  7,00 ms, max:  9,00 ms, nesting: 0 - 1000
TIMING memsetd       :  6,31 s  -  6,31 ms ( 6,31 s  / 1000 ), min:  6,00 ms, max: 18,00 ms, nesting: 0 - 1000
                                                           ----
// CLANG(x64, latest ver.)
 TIMING Fill          :  7,07 s  -  7,07 ms ( 7,07 s  / 1000 ), min:  6,00 ms, max: 10,00 ms, nesting: 0 - 1000
 TIMING memsetd       :  7,07 s  -  7,07 ms ( 7,08 s  / 1000 ), min:  6,00 ms, max: 17,00 ms, nesting: 0 - 1000
                                                           -----
```

Best regards,
Oblivion

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 15 May 2020 07:04:51 GMT
View Forum Message <> Reply to Message

Experimenting with parallel:

```
#include <CtrlLib/CtrlLib.h>

using namespace Upp;

void CoFill(RGBA *t, RGBA c, int len)
{
 const int CHUNK = 1024;
 std::atomic<int> ii(0);
 CoDo([&] {
  for(;;) {
   int pos = CHUNK * ii++;
   if(pos >= len)
    break;
   Fill(t + pos, c, min(CHUNK, len - pos));
  }
 });
}

GUI_APP_MAIN
{
 Color c = Red();

 int len = 4000 * 2000;

 Buffer<RGBA> b(len);

 for(int i = 0; i < 10; i++) {
  {
   RTIMING("memsetd");
   memsetd(b, *(dword*)&(c), len);
  }
  {
   RTIMING("Fill");
   Fill(b, c, len);
  }
  {
   RTIMING("CoFill");
   CoFill(b, c, len);
  }
 }
}
```

```
TIMING CoFill        : 19.00 ms -  1.90 ms (19.00 ms / 10 ), min:  1.00 ms, max:  3.00 ms, nesting: 0 - 10
TIMING Fill          : 31.00 ms -  3.10 ms (31.00 ms / 10 ), min:  3.00 ms, max:  4.00 ms, nesting: 0 - 10
TIMING memsetd       : 30.00 ms -  3.00 ms (30.00 ms / 10 ), min:  2.00 ms, max:  5.00 ms, nesting: 0 - 10
```

To try that on different CPU, Rapsberry PI 4 numbers:

```
TIMING CoFill        : 145.00 ms - 14.50 ms (145.00 ms / 10 ), min: 14.00 ms, max: 15.00 ms, nesting: 0 - 10
TIMING Fill          : 225.00 ms - 22.50 ms (225.00 ms / 10 ), min: 22.00 ms, max: 24.00 ms, nesting: 0 - 10
TIMING memsetd       : 184.00 ms - 18.40 ms (184.00 ms / 10 ), min: 11.00 ms, max: 77.00 ms, nesting: 0 - 10
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 08:18:46 GMT

View Forum Message <> Reply to Message

Hi Mirek,

While interesting, I found that a plain memset() is way faster than memsetd() or Fill(). Just filling with 0xff (as the RGBA is for white) you will get a superior speed. I currently use memset() for a clear white on a ImageBuffer before giving it to BufferPainter. For more complex fill colors, I guess, the apex_memmove / memcpy code could be investigated for a more optimal result. (I posted a link to the apex code here on the forum briefly before release of 2020.1

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization

Posted by mirek on Fri, 15 May 2020 09:33:55 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Fri, 15 May 2020 10:18
While interesting, I found that a plain memset() is way faster than memsetd() or Fill(). Just filling with 0xff (as the RGBA is for white) you will get a superior speed. I currently use memset() for a clear white on a ImageBuffer before giving it to BufferPainter. For more complex fill colors, I guess, the apex_memmove / memcpy code could be investigated for a more optimal result. (I posted a link to the apex code here on the forum briefly before release of 2020.1

Best regards,

Tom

With CLANG, memset performance is about the same. However, with MSVC, it really is pretty damn fast.

I have digged into the code and the key ingredient seems to be MOVNTPS instruction, which means the code could be easily adapted to setting dwords. I just need to understand MT implications mentioned here:

https://www.felixcloutier.com/x86/movntps

It also might be questionable how this will affect the performance down the road (data not being in cache and everything...)

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 09:41:13 GMT
View Forum Message <> Reply to Message

At the time I was testing with the memset -- if I remember correctly -- on Windows + CLANG the memset with zero value was very efficient too, but the rest of the set values were slower. So, there must be some special optimized implementation for zeroing memory on CLANG too.

BR, Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 15 May 2020 09:47:20 GMT
View Forum Message <> Reply to Message

Here we go:

```
void SSEFill2(RGBA *t, RGBA c, int len)
```

```
{
 if(len >= 512) {
  while((uintptr_t)t & 63) { // align to cache line
   *t++ = c;
   len--;
  }
  dword m[4];
  m[0] = m[1] = m[2] = m[3] = *(dword*)&(c);
  __m128d val = _mm_loadu_pd((double *)m);
  while(len >= 16) {
   _mm_stream_pd((double *)t, val);
   _mm_stream_pd((double *)(t + 4), val);
   _mm_stream_pd((double *)(t + 8), val);
   _mm_stream_pd((double *)(t + 12), val);
   t += 16;
   len -= 16;
  }
  _mm_sfence();
 }

 Fill(t, c, len);
}
```

```
TIMING CoFill        : 42.00 ms -  2.10 ms (42.00 ms / 20 ), min:  1.00 ms, max:  3.00 ms, nesting: 0 - 20
TIMING SSEFill2      : 16.00 ms - 799.98 us (16.00 ms / 20 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 20
TIMING SSEFill       : 55.00 ms -  2.75 ms (55.00 ms / 20 ), min:  2.00 ms, max:  3.00 ms, nesting: 0 - 20
TIMING Fill          : 56.00 ms -  2.80 ms (56.00 ms / 20 ), min:  2.00 ms, max:  3.00 ms, nesting: 0 - 20
TIMING memsetd       : 52.00 ms -  2.60 ms (52.00 ms / 20 ), min:  2.00 ms, max:  3.00 ms, nesting: 0 - 20
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 10:08:47 GMT

View Forum Message <> Reply to Message

And we have a winner!!

Also, please take a look at MSBT19 and MSBT19x64 for this too. It looks like this code only works with CLANG and CLANGx64 on Windows. (Have not checked on Linux yet.)
Additionally, plain memset, memsets and memsetd -variants would be useful for various tasks, as their efficiency varies depending on the compiler.

Thanks and best regards,

Tom

EDIT: I mean it does not compile on MSBT...

Subject: Re: BufferPainter::Clear() optimization
Posted by Oblivion on Fri, 15 May 2020 10:16:13 GMT
View Forum Message <> Reply to Message

On linux with the relatively old AMD Athlon FX 6100.

Works with both GCC (9.3) and CLANG (10.0). Requires #include <smmintrin.h>:

GCC:

```
TIMING SSEFill2       : 43,99 ms -  4,40 ms (44,00 ms / 10 ), min:  4,00 ms, max:  5,00 ms, nesting: 0 - 10
TIMING CoFill         : 55,99 ms -  5,60 ms (56,00 ms / 10 ), min:  5,00 ms, max:  6,00 ms, nesting: 0 - 10
TIMING Fill           : 75,99 ms -  7,60 ms (76,00 ms / 10 ), min:  7,00 ms, max:  8,00 ms, nesting: 0 - 10
TIMING memsetd        : 66,99 ms -  6,70 ms (67,00 ms / 10 ), min:  5,00 ms, max: 17,00 ms, nesting: 0 - 10
```

CLANG:

```
TIMING SSEFill2       : 45,99 ms -  4,60 ms (46,00 ms / 10 ), min:  4,00 ms, max:  7,00 ms, nesting: 0 - 10
TIMING CoFill         : 55,99 ms -  5,60 ms (56,00 ms / 10 ), min:  5,00 ms, max:  6,00 ms, nesting: 0 - 10
TIMING Fill           : 65,99 ms -  6,60 ms (66,00 ms / 10 ), min:  6,00 ms, max: 10,00 ms, nesting: 0 - 10
TIMING memsetd        : 78,99 ms -  7,90 ms (79,00 ms / 10 ), min:  5,00 ms, max: 23,00 ms, nesting: 0 - 10
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 10:28:11 GMT
View Forum Message <> Reply to Message

Hi,

Thanks Oblivion; the #include <smmintrin.h> was exactly what was needed on Windows + CLANG too...

Here are the results for the 4k RGBA fill on Windows 10 x64 on Core i9:
MSBT19:
```
TIMING SSEFill2      :  1.30 s  -  1.30 ms ( 1.30 s  / 1000 ), min:  1.03 ms, max:  1.99 ms, nesting: 0 - 1000
TIMING Fill          :  1.13 s  -  1.13 ms ( 1.13 s  / 1000 ), min: 841.00 us, max:  3.04 ms, nesting: 0 - 1000
```

MSBT19x64:
```
TIMING SSEFill2      : 906.90 ms - 906.90 us (906.93 ms / 1000 ), min: 846.00 us, max:  1.67 ms, nesting: 0 - 1000
TIMING Fill          :  2.34 s  -  2.34 ms ( 2.34 s  / 1000 ), min:  2.21 ms, max:  4.69 ms, nesting: 0 - 1000
```

CLANG:
```
TIMING SSEFill2      : 935.97 ms - 935.97 us (936.02 ms / 1000 ), min: 854.00 us, max:  1.67 ms, nesting: 0 - 1000
TIMING Fill          :  2.44 s  -  2.44 ms ( 2.44 s  / 1000 ), min:  2.25 ms, max:  4.74 ms, nesting: 0 - 1000
```

CLANGx64:
```
TIMING SSEFill2      : 934.45 ms - 934.45 us (934.47 ms / 1000 ), min: 854.00 us, max:  1.77 ms, nesting: 0 - 1000
TIMING Fill          :  2.20 s  -  2.20 ms ( 2.20 s  / 1000 ), min:  1.98 ms, max:  5.97 ms, nesting: 0 - 1000
```

Looks very good indeed! MSBT19 on the other hand looks surprising...

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 15 May 2020 11:15:51 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Fri, 15 May 2020 12:08
Additionally, plain memset, memsets and memsetd -variants would be useful for various tasks, as their efficiency varies depending on the compiler.

What about this:

```
void FillCacheLines(void *cache_aligned_ptr, void *data16, int count)
{
 dword *t = (dword *)cache_aligned_ptr;
 __m128d val = _mm_loadu_pd((double *)data16);
 dword *e = t + 16 * count;
 while(t < e) {
  _mm_stream_pd((double *)t, val);
  _mm_stream_pd((double *)(t + 4), val);
  _mm_stream_pd((double *)(t + 8), val);
  _mm_stream_pd((double *)(t + 12), val);
  t += 16;
 }
 _mm_sfence();
}

template <class T>
void MemSet(void *dest, T data, int len)
{
 static_assert(sizeof(T) == 1 || sizeof(T) == 2 || sizeof(T) == 4 || sizeof(T) == 8 || sizeof(T) == 16,
"invalid sizeof");
 T *t = (T *)dest;
 if(len * sizeof(T) > 550) {
  while((uintptr_t)t & 63) { // align to cache line
   *t++ = data;
   len--;
  }
  const int itemn = 16 / sizeof(T);
  const int per_cache_line = 4 * itemn;
  T m[itemn];
  for(int i = 0; i < itemn; i++)
   m[i] = data;
  int count = len / per_cache_line;
  FillCacheLines(t, m, count);
  len -= per_cache_line * count;
 }

 while(len >= 16) {
  t[0] = data; t[1] = data; t[2] = data; t[3] = data;
  t[4] = data; t[5] = data; t[6] = data; t[7] = data;
  t[8] = data; t[9] = data; t[10] = data; t[11] = data;
  t[12] = data; t[13] = data; t[14] = data; t[15] = data;
  t += 16;
  len -= 16;
 }
 switch(len) {
 case 15: t[14] = data;
 case 14: t[13] = data;
 case 13: t[12] = data;
 case 12: t[11] = data;
 case 11: t[10] = data;
 case 10: t[9] = data;
 case 9: t[8] = data;
 case 8: t[7] = data;
 case 7: t[6] = data;
 case 6: t[5] = data;
 case 5: t[4] = data;
 case 4: t[3] = data;
 case 3: t[2] = data;
 case 2: t[1] = data;
 case 1: t[0] = data;
 }
}
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 11:36:13 GMT

View Forum Message <> Reply to Message

Mirek,

Yes, absolutely beautiful!

The results for the set including the new MemSet() on Win10x64 on Core i9 are:

MSBT19:

```
TIMING MemSet        : 831.06 ms - 831.06 us (831.13 ms / 1000 ), min: 779.00 us, max:  1.72 ms, nesting: 0 - 1000
TIMING SSEFill2      :  1.21 s  -  1.21 ms ( 1.21 s  / 1000 ), min:  1.00 ms, max:  2.19 ms, nesting: 0 - 1000
TIMING Fill          : 915.70 ms - 915.70 us (915.76 ms / 1000 ), min: 859.00 us, max:  3.49 ms, nesting: 0 - 1000
```

MSBT19x64:

```
TIMING MemSet        : 818.33 ms - 818.33 us (818.36 ms / 1000 ), min: 777.00 us, max:  1.71 ms, nesting: 0 - 1000
TIMING SSEFill2      : 899.74 ms - 899.74 us (899.77 ms / 1000 ), min: 854.00 us, max:  1.78 ms, nesting: 0 - 1000
TIMING Fill          :  2.29 s  -  2.29 ms ( 2.29 s  / 1000 ), min:  2.21 ms, max:  4.51 ms, nesting: 0 - 1000
```

CLANG:

```
TIMING MemSet        : 835.39 ms - 835.39 us (835.45 ms / 1000 ), min: 790.00 us, max:  1.51
ms, nesting: 0 - 1000
TIMING SSEFill2      : 918.63 ms - 918.63 us (918.68 ms / 1000 ), min: 872.00 us, max:  1.47 ms,
nesting: 0 - 1000
TIMING Fill          :  2.36 s  -  2.36 ms ( 2.36 s  / 1000 ), min:  2.28 ms, max:  5.45 ms, nesting: 0 -
1000
```

CLANGx64:

```
TIMING MemSet        : 838.86 ms - 838.86 us (838.89 ms / 1000 ), min: 787.00 us, max:  1.70
ms, nesting: 0 - 1000
TIMING SSEFill2      : 921.49 ms - 921.49 us (921.51 ms / 1000 ), min: 870.00 us, max:  1.84 ms,
nesting: 0 - 1000
TIMING Fill          :  2.10 s  -  2.10 ms ( 2.10 s  / 1000 ), min:  2.01 ms, max:  5.00 ms, nesting: 0 -
1000
```

I trust you can now make all the different fillers through U++ to use this new code... right?

Thanks and best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 21:13:27 GMT

View Forum Message <> Reply to Message

Hi Mirek,

The game is not over yet, I'm afraid. I did some additional benchmarking with varying buffer lengths to set. It get's more complicated...

```
 RGBA c = Red();

 int bsize=8*1024*1024;
 Buffer<RGBA> b(bsize,(RGBA)Blue());

 String result="\"N\",\"Fill()\",\"memsetd()\",\"MemSet()\"\r\n";
 for(int len=1;len<=bsize;len*=2){
  int maximum=1000000000/len;
  int64 t0=usecs();
  for(int i = 0; i < maximum; i++) Fill(~b, c, len);
  int64 t1=usecs(t0);
  t0=usecs();
  for(int i = 0; i < maximum; i++) memsetd(~b, *(dword*)&(c), len);
  int64 t2=usecs(t0);
  t0=usecs();
  for(int i = 0; i < maximum; i++) MemSet(~b, c, len);
  int64 t3=usecs(t0);
  result.Cat(Format("%d,%f,%f,%f\r\n",len,1000.0*t1/maximum,1000.0*t2/maximum,1000.0*t3/max
imum));
 }

 SaveFile(GetHomeDirFile("Desktop/memset.csv"),result);
```

Now, if you import the resulting memset.csv to your spreadsheet program and create a log-log plot, you will see that the different buffer lengths have a huge impact on the performance of each algorithm. As filling lengths can be quite diverse, I think we need to think about some combination of the different algorithms. Additionally, we need to look at the results on different CPUs. I will keep tinkering on this one for a while here.

(Now I'm running on Core i7 here at home, so this one I can test easily, and also the Core i9 at the office every now and then, as the situation is what it is...)

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Didier on Fri, 15 May 2020 21:45:21 GMT

View Forum Message <> Reply to Message

Here is what I get on my Linux and Ryzen 2700
Du to unstable results with 10 loops, I also placed results for 1000 loops

The new MemSet() is definitly really a good addition

```
==== CLANG X64 ====
TIMING MemSet        : 10.00 ms - 999.98 us (10.00 ms / 10 ), min:  1.00 ms, max:  1.00 ms,
nesting: 0 - 10
TIMING SSEFill2      : 12.00 ms -  1.20 ms (12.00 ms / 10 ), min:  1.00 ms, max:  2.00 ms,
nesting: 0 - 10
TIMING CoFill        : 21.00 ms -  2.10 ms (21.00 ms / 10 ), min:  2.00 ms, max:  3.00 ms, nesting:
0 - 10
TIMING Fill          : 30.00 ms -  3.00 ms (30.00 ms / 10 ), min:  3.00 ms, max:  3.00 ms, nesting: 0
- 10
TIMING memsetd       : 30.00 ms -  3.00 ms (30.00 ms / 10 ), min:  2.00 ms, max:  9.00 ms,
nesting: 0 - 10

TIMING MemSet        : 833.97 ms - 833.97 us (834.00 ms / 1000 ), min:  0.00 ns, max:  2.00 ms,
nesting: 0 - 1000
```

```
TIMING SSEFill2      : 870.97 ms - 870.97 us (871.00 ms / 1000 ), min:  0.00 ns, max:  2.00 ms, nesting: 0 - 1000
TIMING CoFill        :  1.88 s  -  1.88 ms ( 1.88 s  / 1000 ), min:  1.00 ms, max:  3.00 ms, nesting: 0 - 1000
TIMING Fill          :  2.90 s  -  2.90 ms ( 2.90 s  / 1000 ), min:  2.00 ms, max:  4.00 ms, nesting: 0 - 1000
TIMING memsetd       :  2.51 s  -  2.51 ms ( 2.51 s  / 1000 ), min:  2.00 ms, max: 10.00 ms, nesting: 0 - 1000


==== GCC X64 ====
TIMING MemSet        :  7.00 ms - 699.98 us ( 7.00 ms / 10 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 10
TIMING SSEFill2      :  9.00 ms - 899.98 us ( 9.00 ms / 10 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 10
TIMING CoFill        : 23.00 ms -  2.30 ms (23.00 ms / 10 ), min:  2.00 ms, max:  3.00 ms, nesting: 0 - 10
TIMING Fill          : 30.00 ms -  3.00 ms (30.00 ms / 10 ), min:  2.00 ms, max:  4.00 ms, nesting: 0 - 10
TIMING memsetd       : 35.00 ms -  3.50 ms (35.00 ms / 10 ), min:  2.00 ms, max: 10.00 ms, nesting: 0 - 10

TIMING MemSet        : 820.98 ms - 820.98 us (821.00 ms / 1000 ), min:  0.00 ns, max:  2.00 ms, nesting: 0 - 1000
TIMING SSEFill2      : 877.98 ms - 877.98 us (878.00 ms / 1000 ), min:  0.00 ns, max:  2.00 ms, nesting: 0 - 1000
TIMING CoFill        :  1.85 s  -  1.85 ms ( 1.85 s  / 1000 ), min:  1.00 ms, max:  3.00 ms, nesting: 0 - 1000
TIMING Fill          :  2.97 s  -  2.97 ms ( 2.97 s  / 1000 ), min:  2.00 ms, max:  4.00 ms, nesting: 0 - 1000
TIMING memsetd       :  2.52 s  -  2.52 ms ( 2.52 s  / 1000 ), min:  2.00 ms, max:  8.00 ms, nesting: 0 - 1000
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 15 May 2020 23:59:37 GMT

View Forum Message <> Reply to Message

Hi,

I've worked on optimizing the new_memsetd() operation through various buffer sizes up to 8M and here's the best I can come up with (at least this night...). With CLANG it seems to be beneficial to use the Mirek's new MemSet() for buffer sizes above about 1M, but below that and also with MSBT19 / MSBT19x64 the result is better without. (This algorithm is especially efficient with small fills and therefore should work well as a BufferPainter filler too.) For best results, there are separate versions for 32-bit and 64-bit code. (The '#ifdef WIN64' obviously only works on

Windows, but I think there was some other flag on Linux for detecting a 64-bit environment. Please apply that flag, whatever it is, if you test on Linux, etc...)

```
#ifdef WIN64

inline void new_memsetd(dword *t, dword data, int len){
#ifdef COMPILER_CLANG
 if(len>1024*1024){
  MemSet(t,data,len);
  return;
 }
#endif
 if(len&1) *t++=data;
 len>>=1;

 uint64 *w=(uint64*)t;
 uint64 q=data;
 q = (q << 32) | data;

 switch(len) {
  default:{
   uint64 *lim = w + len;
   while(w < lim) *w++ = q;
   break;
  }
  case 16: w[15] = q;
  case 15: w[14] = q;
  case 14: w[13] = q;
  case 13: w[12] = q;
  case 12: w[11] = q;
  case 11: w[10] = q;
  case 10: w[9] = q;
  case 9: w[8] = q;
  case 8: w[7] = q;
  case 7: w[6] = q;
  case 6: w[5] = q;
  case 5: w[4] = q;
  case 4: w[3] = q;
  case 3: w[2] = q;
  case 2: w[1] = q;
  case 1: w[0] = q;
 }
}

#else

inline void new_memsetd(dword *t, dword data, int len){
#ifdef COMPILER_CLANG
```

```
 if(len>1024*1024){
  MemSet(t,data,len);
  return;
 }
#endif
 switch(len) {
  default:{
   dword *lim = t + len;
   while(t < lim) *t++ = data;
   break;
  }
  case 16: t[15] = data;
  case 15: t[14] = data;
  case 14: t[13] = data;
  case 13: t[12] = data;
  case 12: t[11] = data;
  case 11: t[10] = data;
  case 10: t[9] = data;
  case 9: t[8] = data;
  case 8: t[7] = data;
  case 7: t[6] = data;
  case 6: t[5] = data;
  case 5: t[4] = data;
  case 4: t[3] = data;
  case 3: t[2] = data;
  case 2: t[1] = data;
  case 1: t[0] = data;
 }
}

#endif
```

The updated benchmarking code:

```
 RGBA c = Red();

 int bsize=8*1024*1024;
 Buffer<RGBA> b(bsize,(RGBA)Blue());

 String result="\"N\",\"Fill()\",\"new_memsetd()\",\"MemSet()\"\r\n";
 for(int len=1;len<=bsize;){
  int maximum=100000000/len;
  int64 t0=usecs();
  for(int i = 0; i < maximum; i++) Fill(~b, c, len);
  int64 t1=usecs();
  for(int i = 0; i < maximum; i++) new_memsetd((dword*)~b, *(dword*)&(c), len);
  int64 t2=usecs();
  for(int i = 0; i < maximum; i++) MemSet(~b, c, len);
  int64 t3=usecs();
  result.Cat(Format("%d,%f,%f,%f\r\n",len,1000.0*(t1-t0)/maximum,1000.0*(t2-t1)/maximum,1000.
0*(t3-t2)/maximum));
  if(len<32) len++;
  else len*=2;
 }

 SaveFile(GetHomeDirFile("Desktop/memset.csv"),result);
```

Again, I suggest you plot your results using a log-log chart to clearly see the performance with all different block sizes.

If you have some time to spare, please let me know how this works for you.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Sat, 16 May 2020 22:10:57 GMT

View Forum Message <> Reply to Message

Hi,

Interestingly, the FillRGBA() that can be found in the current BufferPainter Fillers, is a real performer. It wins below 1M dwords just about anything else. However, Mireks new MemSet is the winner thereafter. This applies on Windows 10 x64 to CLANG/CLANGx64/MSBT19 on my Core i7. Only MSBT19x64 has a different situation and the following code tries to optimize that, in addition to combining FillRGBA and MemSet for the other compilers:

```
#if defined(WIN64) && defined(COMPILER_MSC)

// for MSBT19x64 only:
inline void new_memsetd(void *b, dword data, int len){
 dword *t=(dword *)b;
 switch(len){
  case 6: t[5] = data;
  case 5: t[4] = data;
  case 4: t[3] = data;
  case 3: t[2] = data;
  case 2: t[1] = data;
  case 1: t[0] = data;
  case 0: return;

  default:{
   if(len&1) *t++=data;
   len>>=1;
   uint64 *w=(uint64*)t;
   uint64 q=*(dword*)&data;
   q |= (q << 32);

   switch(len) {
    default:{
     uint64 *lim = w + len - 32;
     while(w < lim) *w++ = q;
    }
    case 32: w[31] = q;
    case 31: w[30] = q;
    case 30: w[29] = q;
    case 29: w[28] = q;
    case 28: w[27] = q;
    case 27: w[26] = q;
    case 26: w[25] = q;
    case 25: w[24] = q;
    case 24: w[23] = q;
    case 23: w[22] = q;
    case 22: w[21] = q;
    case 21: w[20] = q;
    case 20: w[19] = q;
    case 19: w[18] = q;
    case 18: w[17] = q;
    case 17: w[16] = q;
    case 16: w[15] = q;
    case 15: w[14] = q;
    case 14: w[13] = q;
    case 13: w[12] = q;
    case 12: w[11] = q;
    case 11: w[10] = q;
    case 10: w[9] = q;
    case 9: w[8] = q;
    case 8: w[7] = q;
    case 7: w[6] = q;
    case 6: w[5] = q;
    case 5: w[4] = q;
    case 4: w[3] = q;
    case 3: w[2] = q;
    case 2: w[1] = q;
    case 1: w[0] = q;
   }
  }
 }
}

#else
```

```
inline void new_memsetd(void *b, dword data, int len){
 if(len<=1024*1024) FillRGBA((RGBA*)b,*(RGBA*)&data,len);
 else MemSet(b,data,len);
}

#endif
```

The benchmarking code for various fill sizes now looks like this:

```
 RGBA c = Red();

 int bsize=8*1024*1024;
 Buffer<RGBA> b(bsize,(RGBA)Blue());

 String result="\"N\",\"Fill()\",\"new_memsetd()\",\"MemSet()\",\"FillRGBA()\"\r\n";
 for(int len=1;len<=bsize;){
  int maximum=100000000/len;
  int64 t0=usecs();
  for(int i = 0; i < maximum; i++) Fill(~b, c, len);
  int64 t1=usecs();
  for(int i = 0; i < maximum; i++) new_memsetd(~b, *(dword*)&c, len);
  int64 t2=usecs();
  for(int i = 0; i < maximum; i++) MemSet(~b, c, len);
  int64 t3=usecs();
  for(int i = 0; i < maximum; i++) FillRGBA(~b, c, len);
  int64 t4=usecs();
  result.Cat(Format("%d,%f,%f,%f,%f\r\n",len,1000.0*(t1-t0)/maximum,1000.0*(t2-t1)/maximum,10
00.0*(t3-t2)/maximum,1000.0*(t4-t3)/maximum));
  if(len<64) len++;
  else len*=2;
 }

 SaveFile(GetHomeDirFile("Desktop/memset.csv"),result);
```

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sun, 17 May 2020 06:47:44 GMT

View Forum Message <> Reply to Message

Tom1 wrote on Sat, 16 May 2020 01:59With CLANG it seems to be beneficial to use the Mirek's new MemSet() for buffer sizes above about 1M

I guess L2 cache size plays a role here. The new trick bypasses the cache so kicks in when cache is exhausted...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Sun, 17 May 2020 08:01:51 GMT

View Forum Message <> Reply to Message

mirek wrote on Sun, 17 May 2020 09:47Tom1 wrote on Sat, 16 May 2020 01:59With CLANG it seems to be beneficial to use the Mirek's new MemSet() for buffer sizes above about 1M

I guess L2 cache size plays a role here. The new trick bypasses the cache so kicks in when cache is exhausted...

Mirek

Hi,

Where can I find the new trick?

BR,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sun, 17 May 2020 13:49:50 GMT

View Forum Message <> Reply to Message

Ah, by "trick" I mean using using non-temporal move instruction which we have found in memset....

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sun, 17 May 2020 16:05:52 GMT

View Forum Message <> Reply to Message

What about this:

```
#include <CtrlLib/CtrlLib.h>
#include <smmintrin.h>

using namespace Upp;
```

```
void Fill0(RGBA *t, RGBA c, int len)
{
 while(len >= 16) {
  t[0] = c; t[1] = c; t[2] = c; t[3] = c;
  t[4] = c; t[5] = c; t[6] = c; t[7] = c;
  t[8] = c; t[9] = c; t[10] = c; t[11] = c;
  t[12] = c; t[13] = c; t[14] = c; t[15] = c;
  t += 16;
  len -= 16;
 }
 switch(len & 15) {
 case 15: t[14] = c;
 case 14: t[13] = c;
 case 13: t[12] = c;
 case 12: t[11] = c;
 case 11: t[10] = c;
 case 10: t[9] = c;
 case 9: t[8] = c;
 case 8: t[7] = c;
 case 7: t[6] = c;
 case 6: t[5] = c;
 case 5: t[4] = c;
 case 4: t[3] = c;
 case 3: t[2] = c;
 case 2: t[1] = c;
 case 1: t[0] = c;
 }
}

void Fill2(RGBA *t, RGBA c, int len)
{
 while(len >= 16) {
  t[0] = c; t[1] = c; t[2] = c; t[3] = c;
  t[4] = c; t[5] = c; t[6] = c; t[7] = c;
  t[8] = c; t[9] = c; t[10] = c; t[11] = c;
  t[12] = c; t[13] = c; t[14] = c; t[15] = c;
  t += 16;
  len -= 16;
 }
 if(len & 8) {
  t[0] = t[1] = t[2] = t[3] = t[4] = t[5] = t[6] = t[7] = c;
  t += 8;
 }
 if(len & 4) {
  t[0] = t[1] = t[2] = t[3] = c;
  t += 4;
 }
 if(len & 2) {
  t[0] = t[1] = c;
  t += 2;
 }
 if(len & 1)
  t[0] = c;
}

void Fill3(RGBA *t, RGBA c, int len)
{
 dword m[4];
 m[0] = m[1] = m[2] = m[3] = *(dword*)&(c);
 __m128d val = _mm_loadu_pd((double *)m);
 if(len >= 16) {
  if(len > 1024*1024 / 16 && ((uintptr_t)t & 3) == 0) { // for really huge data, bypass the cache
   while((uintptr_t)t & 15) { // align to 16 bytes for SSE
    *t++ = c;
    len--;
   }
   do {
    _mm_stream_pd((double *)t, val);
    _mm_stream_pd((double *)(t + 4), val);
    _mm_stream_pd((double *)(t + 8), val);
    _mm_stream_pd((double *)(t + 12), val);
    t += 16;
    len -= 16;
   }
   while(len >= 16);
   _mm_sfence();
  }
  else
   do {
    _mm_storeu_pd((double *)t, val);
    _mm_storeu_pd((double *)(t + 4), val);
    _mm_storeu_pd((double *)(t + 8), val);
    _mm_storeu_pd((double *)(t + 12), val);
    t += 16;
    len -= 16;
   }
   while(len >= 16);
 }
 if(len & 8) {
  _mm_storeu_pd((double *)t, val);
  _mm_storeu_pd((double *)(t + 4), val);
  t += 8;
 }
 if(len & 4) {
  _mm_storeu_pd((double *)t, val);
  t += 4;
 }
 if(len & 2) {
  t[0] = t[1] = c;
  t += 2;
 }
 if(len & 1)
  t[0] = c;
}

int len = 2000 * 4000;

GUI_APP_MAIN
{
 Color c = Red();

 Buffer<RGBA> b(2000);

 Vector<int> rnd;
 for(int i = 0; i < 200; i++)
  rnd << Random(100);

 for(int i = 0; i < 1000000; i++) {
  {
   RTIMING("memsetd");
   for(int i = 0; i < rnd.GetCount(); i += 2)
    memsetd(b + rnd[i], *(dword*)&(c), rnd[i + 1]);
  }
  {
   RTIMING("Fill");
   for(int i = 0; i < rnd.GetCount(); i += 2)
    Fill(b + rnd[i], c, rnd[i + 1]);
  }
  {
   RTIMING("Fill0");
   for(int i = 0; i < rnd.GetCount(); i += 2)
    Fill0(b + rnd[i], c, rnd[i + 1]);
  }
  {
   RTIMING("Fill2");
   for(int i = 0; i < rnd.GetCount(); i += 2)
    Fill2(b + rnd[i], c, rnd[i + 1]);
  }
  {
   RTIMING("Fill3");
   for(int i = 0; i < rnd.GetCount(); i += 2)
    Fill3(b + rnd[i], c, rnd[i + 1]);
  }
  {
   RTIMING("memset");
   for(int i = 0; i < rnd.GetCount(); i += 2)
    memset(b + 4 * rnd[i], 255, 4 * rnd[i + 1]);
  }
 }

 b.Alloc(len);

 for(int i = 0; i < 20; i++) {
  memsetd(b, *(dword*)&(c), len);
  {
   RTIMING("HUGE memsetd");
   memsetd(b, *(dword*)&(c), len);
  }
  {
   RTIMING("HUGE Fill");
   Fill(b, c, len);
  }
  {
   RTIMING("HUGE Fill3");
   Fill3(b, c, len);
  }
  {
   RTIMING("HUGE memset");
   memset(b, c, len * 4);
  }
 }

 BeepExclamation();
}
```

I believe Fill3 does not have any weakness here... Actually, CLANG produced almost exactly the same code for Fill2 and memsetd for small fills, but I guess providing SSE2 implementation directly does not hurt anything. Plus we still like to have that cache bypass...

So I would go for Fill3 for X86 and Fill2 for non-X86 (in the hope it gets optimized for neon on ARM...)

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Sun, 17 May 2020 18:56:41 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Here are my results:

CLANG

```
TIMING HUGE memset    : 27.28 ms -  1.36 ms (27.29 ms / 20 ), min:  1.25 ms, max:  1.66 ms,
nesting: 0 - 20
TIMING HUGE Fill3     : 35.01 ms -  1.75 ms (35.01 ms / 20 ), min:  1.63 ms, max:  1.99 ms,
nesting: 0 - 20
TIMING HUGE Fill      : 73.74 ms -  3.69 ms (73.75 ms / 20 ), min:  3.32 ms, max:  7.43 ms,
nesting: 0 - 20
TIMING HUGE memsetd   : 72.88 ms -  3.64 ms (72.88 ms / 20 ), min:  3.40 ms, max:  4.51 ms,
nesting: 0 - 20
TIMING memset         :  1.01 s  -  1.01 us ( 1.07 s  / 1000000 ), min:  1.00 us, max: 28.00 us,
nesting: 0 - 1000000
TIMING Fill3          : 505.44 ms - 505.44 ns (565.98 ms / 1000000 ), min:  0.00 ns, max: 29.00 us,
nesting: 0 - 1000000
TIMING Fill2          : 497.06 ms - 497.06 ns (557.61 ms / 1000000 ), min:  0.00 ns, max: 28.00 us,
nesting: 0 - 1000000
TIMING Fill0          : 772.53 ms - 772.53 ns (833.07 ms / 1000000 ), min:  0.00 ns, max: 63.00 us,
nesting: 0 - 1000000
TIMING Fill           :  1.67 s  -  1.67 us ( 1.73 s  / 1000000 ), min:  1.00 us, max: 58.00 us, nesting:
0 - 1000000
TIMING memsetd        : 495.67 ms - 495.67 ns (556.22 ms / 1000000 ), min:  0.00 ns, max: 28.00
us, nesting: 0 - 1000000
```

CLANGx64

```
TIMING HUGE memset    : 27.76 ms -  1.39 ms (27.76 ms / 20 ), min:  1.28 ms, max:  1.80 ms,
nesting: 0 - 20
TIMING HUGE Fill3     : 36.31 ms -  1.82 ms (36.31 ms / 20 ), min:  1.59 ms, max:  2.27 ms,
nesting: 0 - 20
TIMING HUGE Fill      : 73.42 ms -  3.67 ms (73.42 ms / 20 ), min:  3.41 ms, max:  4.74 ms,
nesting: 0 - 20
TIMING HUGE memsetd   : 74.52 ms -  3.73 ms (74.52 ms / 20 ), min:  3.47 ms, max:  4.22 ms,
nesting: 0 - 20
TIMING memset         : 898.49 ms - 898.49 ns (925.83 ms / 1000000 ), min:  0.00 ns, max: 52.00
us, nesting: 0 - 1000000
TIMING Fill3          : 492.59 ms - 492.59 ns (519.92 ms / 1000000 ), min:  0.00 ns, max: 32.00 us,
nesting: 0 - 1000000
TIMING Fill2          : 495.82 ms - 495.82 ns (523.15 ms / 1000000 ), min:  0.00 ns, max: 28.00 us,
nesting: 0 - 1000000
TIMING Fill0          : 569.61 ms - 569.61 ns (596.95 ms / 1000000 ), min:  0.00 ns, max: 41.00 us,
nesting: 0 - 1000000
TIMING Fill           : 591.56 ms - 591.56 ns (618.90 ms / 1000000 ), min:  0.00 ns, max: 30.00 us,
nesting: 0 - 1000000
TIMING memsetd        : 549.04 ms - 549.04 ns (576.37 ms / 1000000 ), min:  0.00 ns, max: 65.00
us, nesting: 0 - 1000000
```

MSBT19

```
TIMING HUGE memset    : 26.51 ms -  1.33 ms (26.51 ms / 20 ), min:  1.26 ms, max:  1.49 ms,
nesting: 0 - 20
TIMING HUGE Fill3     : 35.42 ms -  1.77 ms (35.42 ms / 20 ), min:  1.58 ms, max:  2.14 ms,
nesting: 0 - 20
TIMING HUGE Fill      : 25.47 ms -  1.27 ms (25.48 ms / 20 ), min:  1.18 ms, max:  1.59 ms,
nesting: 0 - 20
TIMING HUGE memsetd   : 25.12 ms -  1.26 ms (25.12 ms / 20 ), min:  1.15 ms, max:  1.59 ms,
nesting: 0 - 20
TIMING memset         : 978.21 ms - 978.21 ns ( 1.05 s  / 1000000 ), min:  1.00 us, max: 29.00 us,
nesting: 0 - 1000000
TIMING Fill3          :  1.50 s  -  1.50 us ( 1.58 s  / 1000000 ), min:  1.00 us, max: 29.00 us, nesting:
0 - 1000000
TIMING Fill2          :  1.89 s  -  1.89 us ( 1.96 s  / 1000000 ), min:  1.00 us, max: 34.00 us, nesting:
0 - 1000000
TIMING Fill0          :  2.02 s  -  2.02 us ( 2.09 s  / 1000000 ), min:  1.00 us, max: 33.00 us, nesting:
0 - 1000000
TIMING Fill           :  2.06 s  -  2.06 us ( 2.14 s  / 1000000 ), min:  1.00 us, max: 32.00 us, nesting:
0 - 1000000
TIMING memsetd        :  1.62 s  -  1.62 us ( 1.69 s  / 1000000 ), min:  1.00 us, max: 45.00 us,
nesting: 0 - 1000000
```

MSBT19x64

```
TIMING HUGE memset    : 26.96 ms -  1.35 ms (26.96 ms / 20 ), min:  1.27 ms, max:  1.90 ms,
nesting: 0 - 20
TIMING HUGE Fill3     : 35.07 ms -  1.75 ms (35.08 ms / 20 ), min:  1.62 ms, max:  2.02 ms,
nesting: 0 - 20
TIMING HUGE Fill      : 67.09 ms -  3.35 ms (67.09 ms / 20 ), min:  3.17 ms, max:  3.60 ms,
nesting: 0 - 20
TIMING HUGE memsetd   : 25.64 ms -  1.28 ms (25.64 ms / 20 ), min:  1.19 ms, max:  1.48 ms,
nesting: 0 - 20
TIMING memset         : 818.75 ms - 818.75 ns (856.11 ms / 1000000 ), min:  0.00 ns, max: 31.00
us, nesting: 0 - 1000000
TIMING Fill3          :  1.36 s  -  1.36 us ( 1.40 s  / 1000000 ), min:  1.00 us, max: 31.00 us, nesting:
0 - 1000000
TIMING Fill2          :  1.67 s  -  1.67 us ( 1.70 s  / 1000000 ), min:  1.00 us, max: 30.00 us, nesting:
0 - 1000000
TIMING Fill0          :  1.66 s  -  1.66 us ( 1.70 s  / 1000000 ), min:  1.00 us, max: 46.00 us, nesting:
0 - 1000000
TIMING Fill           :  1.68 s  -  1.68 us ( 1.72 s  / 1000000 ), min:  1.00 us, max: 50.00 us, nesting:
0 - 1000000
TIMING memsetd        :  1.50 s  -  1.50 us ( 1.54 s  / 1000000 ), min:  1.00 us, max: 29.00 us,
nesting: 0 - 1000000
```

Fill3 is generally the best, but I experience two issues behind the scenes of this benchmark:

1. On MSBT19 / MSBT19x64 there is a significant penalty for small counts. It results in 5 ns per call, whereas in CLANG it is about 0.8 - 1.0 ns per call.
2. On MSBT19x64 the optimal threshold size is 2M counts on my Core i7. However, interestingly the default threshold value works better with MSBT19 on the same computer.

I will continue to investigate this.

Thanks and best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Sun, 17 May 2020 19:46:30 GMT
View Forum Message <> Reply to Message

Mirek,

Also, please check my previous new_memsetd() above using MSBT19x64 for reference. Preferably also with short transfers (1-64).

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Oblivion on Sun, 17 May 2020 20:00:14 GMT
View Forum Message <> Reply to Message

Here's results on (AMD FX, linux x64):

GCC:

```
TIMING HUGE memset    : 113,98 ms -  5,70 ms (114,00 ms / 20 ), min:  5,00 ms, max:  6,00 ms, nesting: 0 - 20
TIMING HUGE Fill3     : 81,98 ms -  4,10 ms (82,00 ms / 20 ), min:  4,00 ms, max:  5,00 ms, nesting: 0 - 20
TIMING HUGE Fill      : 145,98 ms -  7,30 ms (146,00 ms / 20 ), min:  7,00 ms, max:  8,00 ms, nesting: 0 - 20
TIMING HUGE memsetd   : 125,98 ms -  6,30 ms (126,00 ms / 20 ), min:  6,00 ms, max:  7,00 ms, nesting: 0 - 20
TIMING memset         :  1,24 s  -  1,24 us ( 2,23 s  / 1000000 ), min:  0,00 ns, max:  1,00 ms, nesting: 0 - 1000000
TIMING Fill3          :  2,01 s  -  2,01 us ( 3,01 s  / 1000000 ), min:  0,00 ns, max:  1,00 ms, nesting: 0 - 1000000
```

TIMING Fill2   : 2,43 s - 2,43 us (3,42 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING Fill0   : 2,77 s - 2,77 us (3,76 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING Fill   : 2,72 s - 2,72 us (3,71 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING memsetd   : 1,80 s - 1,80 us (2,80 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000

ClANG:

TIMING HUGE memset   : 120,98 ms - 6,05 ms (121,00 ms / 20), min: 5,00 ms, max: 7,00 ms, nesting: 0 - 20
TIMING HUGE Fill3   : 81,98 ms - 4,10 ms (82,00 ms / 20), min: 4,00 ms, max: 5,00 ms, nesting: 0 - 20
TIMING HUGE Fill   : 130,98 ms - 6,55 ms (131,00 ms / 20), min: 6,00 ms, max: 7,00 ms, nesting: 0 - 20
TIMING HUGE memsetd   : 133,98 ms - 6,70 ms (134,00 ms / 20), min: 6,00 ms, max: 7,00 ms, nesting: 0 - 20
TIMING memset   : 1,49 s - 1,49 us (2,39 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING Fill3   : 1,74 s - 1,74 us (2,64 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING Fill2   : 1,62 s - 1,62 us (2,52 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING Fill0   : 2,00 s - 2,00 us (2,90 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING Fill   : 2,06 s - 2,06 us (2,96 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000
TIMING memsetd   : 2,18 s - 2,18 us (3,08 s / 1000000), min: 0,00 ns, max: 1,00 ms, nesting: 0 - 1000000

Best regards,
Oblivion

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Sun, 17 May 2020 21:25:00 GMT
View Forum Message <> Reply to Message

Mirek,

Please check out this one. It features better performance on MSBT19 / MSBT19x64 with low counts, and works well on CLANG/CLANGx64 too:
inline void new_memset128(void *b, dword data, int len){

```
 switch(len){
  case 4: ((dword *)b)[3] = data;
  case 3: ((dword *)b)[2] = data;
  case 2: ((dword *)b)[1] = data;
  case 1: ((dword *)b)[0] = data;
  case 0: return;
 }

 __m128i q = _mm_set1_epi32(*(int*)&data);
 __m128i *w = (__m128i*)b;

 switch(len>>2){
  default:{
   __m128i *e = (__m128i*)b + (len>>2) - 4;
   if(len <= 2*1024*1024){
    while(w<e){
     _mm_storeu_si128(w++, q);
     _mm_storeu_si128(w++, q);
     _mm_storeu_si128(w++, q);
     _mm_storeu_si128(w++, q);
    }
   }
   else{
    while(w<e){
     _mm_stream_si128(w++, q);
     _mm_stream_si128(w++, q);
     _mm_stream_si128(w++, q);
     _mm_stream_si128(w++, q);
    }
   }
  }
  case 4: _mm_storeu_si128(w++, q);
  case 3: _mm_storeu_si128(w++, q);
  case 2: _mm_storeu_si128(w++, q);
  case 1: _mm_storeu_si128(w++, q);
 }
 switch(len&3){
  case 3: ((dword *)b)[len-3] = data;
  case 2: ((dword *)b)[len-2] = data;
  case 1: ((dword *)b)[len-1] = data;
 }
}
```

Best regards,

Tom

EDIT: Fine tuning...

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 08:16:32 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Sun, 17 May 2020 23:25Mirek,

Please check out this one. It features better performance on MSBT19 / MSBT19x64 with low counts, and works well on CLANG/CLANGx64 too:

I think there are 2 issues:

- Cache bypass starts at 8MB.
- Missing alignment adjustment for cache bypass.
- I might be wrong, but why is there " - 4": __m128i *e = (__m128i*)t + (len>>2) - 4; ?

But yes, it hits something for MSC compiler...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 09:13:55 GMT
View Forum Message <> Reply to Message

Hi,

Yes, you're right: The alignment should be handled. I'll take a look at it... (just need to minimize the code size in order to avoid penalty for short transfers. It is extremely sensitive.)

The cache limit is intentionally 8MB as this is the sweet spot for my Core i7. Probably should get this value from the system to optimize the correct threshold.

The -4 compensates the rest of the samples handled within the leaked default in the switch. (The below cases do the trick).

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 11:31:25 GMT
View Forum Message <> Reply to Message

Hi,

Alignment corrected. (Caused obviously a lot of rearranging things to obtain balance.) Threshold is still at 8M, but feel free to experiment.

```
inline void new_memset128(void *b, dword data, int len){
 switch(len){
  case 5: ((dword *)b)[4] = data;
  case 4: ((dword *)b)[3] = data;
  case 3: ((dword *)b)[2] = data;
  case 2: ((dword *)b)[1] = data;
  case 1: ((dword *)b)[0] = data;
  case 0: return;
 }

 __m128i q = _mm_set1_epi32(*(int*)&data);
 __m128i *w = (__m128i*)b;
 __m128i *e = (__m128i*)b + (len>>2);

 if(len <= 2*1024*1024 || ((uintptr_t)b&3)){
  while(w<e-1){
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
  }
  if(w<e) _mm_storeu_si128(w++, q);
 }
 else{
  int s=(-((int)((uintptr_t)b)>>2))&0x3;
  switch(s){
   case 3: ((dword *)b)[2] = data;
   case 2: ((dword *)b)[1] = data;
   case 1: ((dword *)b)[0] = data;
  }

  w = (__m128i*) ((dword*)b + s);

  while(w<e) _mm_stream_si128(w++, q);
  _mm_sfence();
 }

 switch(len&3){
  case 3: ((dword *)b)[len-3] = data;
  case 2: ((dword *)b)[len-2] = data;
  case 1: ((dword *)b)[len-1] = data;
 }
}
```

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 11:33:20 GMT
View Forum Message <> Reply to Message

So I kept digging and found that the reason why Fill3 performed great with CLANG and less great with MSC was that CLANG understand what I mean with that ugly array based code to fill 4 color values into the xmm register while MSC really created that 'm' array, stored 4 values into memory and then fetched them into xmm...

Fixed Fill3 seems to perform well with MSC too:

```
void Fill3(RGBA *t, RGBA c, int len)
{
 __m128i val4 = _mm_set1_epi32(*(int*)&c);
 auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };
 auto Set4S = [&](int at) { _mm_stream_si128((__m128i *)(t + at), val4); };
 if(len >= 16) {
  if(len > 1024*1024 / 16 && ((uintptr_t)t & 3) == 0) { // for really huge data, bypass the cache
   while((uintptr_t)t & 15) { // align to 16 bytes for SSE
    *t++ = c;
    len--;
   }
   do {
    Set4S(0);
    Set4S(4);
    Set4S(8);
    Set4S(12);
    t += 16;
    len -= 16;
   }
   while(len >= 16);
   _mm_sfence();
  }
  else
   do {
    Set4(0);
    Set4(4);
    Set4(8);
    Set4(12);
    t += 16;
    len -= 16;
   }
   while(len >= 16);
 }
 if(len & 8) {
  Set4(0);
  Set4(4);
  t += 8;
 }
 if(len & 4) {
  Set4(0);
  t += 4;
 }
 if(len & 2) {
  t[0] = t[1] = c;
  t += 2;
 }
 if(len & 1)
  t[0] = c;
}
```

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 11:43:39 GMT
View Forum Message <> Reply to Message

This variation of basically the same thing seems a tiny bit faster:

```
void Fill3a(RGBA *t, RGBA c, int len)
{
 __m128i val4 = _mm_set1_epi32(*(int*)&c);
 auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };
 auto Set4S = [&](int at) { _mm_stream_si128((__m128i *)(t + at), val4); };
 if(len >= 32) {
  if(len > 1024*1024 / 16 && ((uintptr_t)t & 3) == 0) { // for really huge data, bypass the cache
   while((uintptr_t)t & 15) { // align to 16 bytes for SSE
    *t++ = c;
    len--;
   }
   do {
    Set4S(0); Set4S(4); Set4S(8); Set4S(12);
    Set4S(16); Set4S(20); Set4S(24); Set4S(28);
    t += 32;
    len -= 32;
   }
   while(len >= 32);
   _mm_sfence();
  }
  else
   do {
    Set4(0); Set4(4); Set4(8); Set4(12);
    Set4(16); Set4(20); Set4(24); Set4(28);
    t += 32;
    len -= 32;
   }
   while(len >= 32);
 }
 if(len & 16) {
  Set4(0); Set4(4); Set4(8); Set4(12);
  t += 16;
 }
 if(len & 8) {
  Set4(0); Set4(4);
  t += 8;
 }
 if(len & 4) {
  Set4(0);
  t += 4;
 }
 if(len & 2) {
  t[0] = t[1] = c;
  t += 2;
 }
 if(len & 1)
  t[0] = c;
}
```

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 11:53:37 GMT
View Forum Message <> Reply to Message

You can actually do alignment without branching there (that I got from studying memset code . I guess that is the last thing to try now

---

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 14:06:19 GMT
View Forum Message <> Reply to Message

mirek wrote on Mon, 18 May 2020 14:53You can actually do alignment without branching there (that I got from studying memset code . I guess that is the last thing to try now

Hi,

Sounds good, but seems hard to squeeze speed from ... (tried quite a while now).

BR, Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 15:08:11 GMT

View Forum Message <> Reply to Message

Mirek,

Here it is: The unconditional alignment. I took your idea, ditched my own, and modified your Fill3a as follows:

```
void inline Fill3T(void *b, dword data, int len){
 switch(len){
  case 3: ((dword *)b)[2] = data;
  case 2: ((dword *)b)[1] = data;
  case 1: ((dword *)b)[0] = data;
  case 0: return;
 }
 __m128i q = _mm_set1_epi32(*(int*)&data);
 __m128i *w = (__m128i*)b;

 if(len >= 32) {
  __m128i *e = (__m128i*)b + (len>>2) - 8;
  if(len > 1024*1024 / 16 && ((uintptr_t)w & 3) == 0) { // for really huge data, bypass the cache
   _mm_storeu_si128(w, q); // Head align
   int s=(-((int)((uintptr_t)b)>>2))&0x3;
   w = (__m128i*) ((dword*)b) + s;
   do {
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
   }while(w<=e);
   _mm_sfence();
  }
  else
```

```
    do {
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
      _mm_storeu_si128(w++, q);
    }while(w<=e);
  }

  if(len & 16) {
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
  }
  if(len & 8) {
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
  }
  if(len & 4) {
    _mm_storeu_si128(w, q);
  }
  _mm_storeu_si128((__m128i*) (((dword*)b) + len - 4), q); // Tail align
}
```

I made some other changes too and this one is slightly faster on short transfers while equals Fill3a() on longer ones. The improvement is more significant on MSBT19 / MSBT19x64.

In order to get real fast short transfers, the function must be 'inline'. I think this necessitates two variants of the final function. (I have seen that BufferPainter paints most of the time with really short fills, so inlining really makes a difference there.)

Best regards,

Tom

P.S. My cache threshold is still at 8M...

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 16:12:37 GMT
View Forum Message <> Reply to Message

Well, my full idea was to align for len >= 32 always and MAYBE have some benefit from the fact that stores are now aligned (even perhaps use aligned version). Sources diverge on actuall performance, but it might be around 10%. In any case, MSC memset does this...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 16:28:52 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Mon, 18 May 2020 17:08
In order to get real fast short transfers, the function must be 'inline'. I think this necessitates two variants of the final function. (I have seen that BufferPainter paints most of the time with really short fills, so inlining really makes a difference there.)

Well, CLANG inlines all Fill3 variants without me asking him to do it, so I guess I have zero problems to have it in the header...

Quote:P.S. My cache threshold is still at 8M...

What are your CPU L1/L2/L3 caches?

What happens if you move that to 1M, 12M, 16M? (I mean, how much penalty you get?)

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 18:57:20 GMT
View Forum Message <> Reply to Message

Hi,

My CPU here is Intel(R) Core(TM) i7-4790K:

https://ark.intel.com/content/www/us/en/ark/products/80807/i
ntel-core-i7-4790k-processor-8m-cache-up-to-4-40-ghz.html

Not surprisingly, they say it has an 8M 'smart cache'.

Please find attached two CSV files portraying execution time in ns for each call in average. The length is in dwords. Fill3a is there for reference and Fill3T is using 64 dword threshold for streaming in one and 2M dword (8MB) threshold in the other file. While not portrayed here, increasing the threshold above 32MB decreases the performance from 1.5 ms to 3.6 ms for a 32 MB buffer.

Best regards,

Tom

File Attachments

1) Fill3T-cache-comparison.7z, downloaded 283 times

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 19:20:28 GMT
View Forum Message <> Reply to Message

Hi,

It looks that luckily CPUID reveals cache information:

 https://www.intel.com/content/www/us/en/architecture-and-tec
hnology/64-ia-32-architectures-software-developer-vol-2a-man ual.html

Initial value in EAX 80000006H
ECX:
Bits 07 - 00: Cache Line size in bytes.
Bits 11 - 08: Reserved.
Bits 15 - 12: L2 Associativity field.
Bits 31 - 16: Cache size in 1K units.

Could we use this?

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Mon, 18 May 2020 19:40:50 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Something like this, maybe... I'm not quite sure as this method reports 16M cache for me -- although this works quite well for me:

static int cachesize=999;

```
INITBLOCK{
#ifdef COMPILER_MSC
 int cpuInfo[4];
 Zero(cpuInfo);
 __cpuid(cpuInfo, 0x80000006);
#else
 unsigned int cpuInfo[4];
 Zero(cpuInfo);
 __get_cpuid(0x80000006, &cpuInfo[0], &cpuInfo[1], &cpuInfo[2], &cpuInfo[3]);
#endif
 cachesize=1024*(cpuInfo[2]>>16)*(cpuInfo[2]&0xff);
};


void inline Fill3T(void *b, dword data, int len){
 switch(len){
  case 3: ((dword *)b)[2] = data;
  case 2: ((dword *)b)[1] = data;
  case 1: ((dword *)b)[0] = data;
  case 0: return;
 }
 __m128i q = _mm_set1_epi32(*(int*)&data);
 __m128i *w = (__m128i*)b;

 if(len >= 32) {
  __m128i *e = (__m128i*)b + (len>>2) - 8;
  if(len >= (cachesize>>2) && ((uintptr_t)w & 3) == 0) { // for really huge data, bypass the cache
   _mm_storeu_si128(w, q); // Head align
   int s=(-((int)((uintptr_t)b)>>2))&0x3;
   w = (__m128i*) ((dword*)b) + s;
   do {
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
   }while(w<=e);
   _mm_sfence();
  }
  else
   do {
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
   }while(w<=e);
 }

 if(len & 16) {
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
 }
 if(len & 8) {
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
 }
 if(len & 4) {
  _mm_storeu_si128(w, q);
 }
 _mm_storeu_si128((__m128i*) (((dword*)b) + len - 4), q); // Tail align
}
```

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 19:56:57 GMT

View Forum Message <> Reply to Message

Tom1 wrote on Mon, 18 May 2020 20:57Hi,

My CPU here is Intel(R) Core(TM) i7-4790K:

  https://ark.intel.com/content/www/us/en/ark/products/80807/i
ntel-core-i7-4790k-processor-8m-cache-up-to-4-40-ghz.html

Not surprisingly, they say it has an 8M 'smart cache'.

Please find attached two CSV files portraying execution time in ns for each call in average. The length is in dwords. Fill3a is there for reference and Fill3T is using 64 dword threshold for streaming in one and 2M dword (8MB) threshold in the other file. While not portrayed here, increasing the threshold above 32MB decreases the performance from 1.5 ms to 3.6 ms for a 32 MB buffer.

Best regards,

Tom

If I interpret these numbers correctly, it looks like around 4MB potential drop because of cache bypass starts to be diminish, right?

Thing is, I am afraid that making this dynamic will cause a lot of problems, starting with perfromance - it is after all another read from the memory. I would settle for some compromise constant there. Like 4MB...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 18 May 2020 22:02:48 GMT
View Forum Message <> Reply to Message

What about this:

```
never_inline
void HugeFill(dword *t, dword c, int len)
{
 __m128i val4 = _mm_set1_epi32(*(int*)&c);
 auto Set4S = [&](int at) { _mm_stream_si128((__m128i *)(t + at), val4); };
 while((uintptr_t)t & 15) { // align to 16 bytes for SSE
  *t++ = c;
  len--;
 }
 while(len >= 16) {
  Set4S(0);
  Set4S(4);
  Set4S(8);
  Set4S(12);
  t += 16;
  len -= 16;
 }
 while(len--)
  *t++ = c;
 _mm_sfence();
}

void Fill6(dword *t, dword c, int len)
{
 if(len >= 4) {
  __m128i val4 = _mm_set1_epi32(*(int*)&c);
  auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };
  if(len > 4*1024*1024 / 4) {
   HugeFill(t, c, len);
   return;
  }
  while(len >= 16) {
   Set4(0);
   Set4(4);
   Set4(8);
   Set4(12);
   t += 16;
   len -= 16;
  }
  if(len & 8) {
   Set4(0);
   Set4(4);
   t += 8;
  }
  if(len & 4) {
   Set4(0);
   t += 4;
  }
 }
 if(len & 3)
  t[0] = t[(len & 2) >> 1] = t[(len & 2) & ((len & 1) << 1)] = c;
}
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 19 May 2020 06:59:08 GMT

View Forum Message <> Reply to Message

Hi,

Fill6 fails integrity check due to a small indexing glitch here:

```
 if(len & 8) {
  Set4(0);
  Set4(8); // << Should be 4
  t += 8;
 }
```

However, Fill3T is still faster below 64 and mostly on par above that on my i7.

And thanks! I do indeed enjoy the final alignment trick! Very clever!

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 19 May 2020 07:14:34 GMT

View Forum Message <> Reply to Message

Yeah, there was another bug in it, I should test more before posting.

In retrospective, while the trick is nice, I do not think it is worth it. But if you wanted to experiment with this path, I have found the way how to extend / simplify this. The basic idea is

```
int nlen = -len;
t[1 & HIBYTE(nlen)] = c;
nlen++;
t[2 & HIBYTE(nlen)] = c;
nlen++;
t[3 & HIBYTE(nlen)] = c;
....
```

(at some point, nlen will become > 0 and thus HIBYTE goes from 0xff to 0x00, thus "grounding" indices).

Also, I would like to try to explain why I am trying to beat Fill3T. It is about those switches, while

```
switch(len) {
case 0:
case 1:
case 2:
default:
}
```

looks magnificent, it is actually 2 "unstable" branch predictions and quite a bit of code to compute the target address. So

```
if(len & 2) {
}
if(len & 1) {
}
```

should be on par - 2 branch predictions and maybe a bit less of code....

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 19 May 2020 07:49:01 GMT

View Forum Message <> Reply to Message

Also, a little note about your testing code: You loop over the same "len" many times and measure that. The problem is that first pass setups branch prediction so all other passes are predicted. If "len" is changing, prediction fails and you might get different results....

Which explains why my tests, which feeds random lens, shows a bit different picture...

All in all, I think in the end we will just need to test this with Painter....

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 19 May 2020 09:32:39 GMT

View Forum Message <> Reply to Message

Three more variants, based on your FillT. Fill7 is basically identical, with a little trick added (hope you like it). Fill7a has different "frontend". Fill8 is not performing very well, adding that just so that you know I have tested that variant too...

Fill7 and Fill7a seem to be basically equal and maybe just a tiny bit faster than Fill3T....

```
void Fill7(dword *t, dword data, int len){
 switch(len) {
  case 3: t[2] = data;
  case 2: t[1] = data;
  case 1: t[0] = data;
  case 0: return;
 }

 __m128i val4 = _mm_set1_epi32(data);
 auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };

 Set4(len - 4); // fill tail
 if(len >= 32) {
  if(len >= 1024*1024) { // for really huge data, bypass the cache
   HugeFill(t, data, len);
   return;
  }
  const dword *e = t + len - 32;
  do {
   Set4(0); Set4(4); Set4(8); Set4(12);
   Set4(16); Set4(20); Set4(24); Set4(28);
   t += 32;
  }
  while(t <= e);
 }
 if(len & 16) {
  Set4(0); Set4(4); Set4(8); Set4(12);
  t += 16;
 }
 if(len & 8) {
  Set4(0); Set4(4);
  t += 8;
 }
 if(len & 4)
  Set4(0);
}

void Fill7a(dword *t, dword data, int len){
 if(len < 4) {
  if(len & 2) {
   t[0] = t[1] = data;
   t += 2;
  }
  if(len & 1)
   t[0] = data;
  return;
 }

 __m128i val4 = _mm_set1_epi32(data);
 auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };

 Set4(len - 4); // fill tail
 if(len >= 32) {
  if(len >= 1024*1024) { // for really huge data, bypass the cache
   HugeFill(t, data, len);
   return;
  }
  const dword *e = t + len - 32;
  do {
   Set4(0); Set4(4); Set4(8); Set4(12);
   Set4(16); Set4(20); Set4(24); Set4(28);
   t += 32;
  }
  while(t <= e);
 }
 if(len & 16) {
  Set4(0); Set4(4); Set4(8); Set4(12);
  t += 16;
 }
 if(len & 8) {
  Set4(0); Set4(4);
  t += 8;
 }
 if(len & 4)
  Set4(0);
}

void Fill8(dword *t, dword data, int len){
 switch(len) {
  case 3: t[2] = data;
  case 2: t[1] = data;
  case 1: t[0] = data;
  case 0: return;
 }

 __m128i val4 = _mm_set1_epi32(data);
 auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };

 Set4(len - 4); // fill tail
 if(len >= 32) {
  if(len >= 1024*1024) { // for really huge data, bypass the cache
   HugeFill(t, data, len);
   return;
  }
  int cnt = len >> 5;
  do {
   Set4(0); Set4(4); Set4(8); Set4(12);
   len -= 32;
   Set4(16); Set4(20); Set4(24); Set4(28);
   t += 32;
  }
  while(len >= 32);
 }
 switch((len >> 2) & 7) {
 case 7: Set4(24);
 case 6: Set4(20);
 case 5: Set4(16);
 case 4: Set4(12);
 case 3: Set4(8);
 case 2: Set4(4);
 case 1: Set4(0);
 }
}
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 19 May 2020 10:35:28 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 19 May 2020 10:49Also, a little note about your testing code: You loop over the same "len" many times and measure that. The problem is that first pass setups branch prediction so all other passes are predicted. If "len" is changing, prediction fails and you might get different results....

Which explains why my tests, which feeds random lens, shows a bit different picture...

All in all, I think in the end we will just need to test this with Painter....

Mirek

I wish I came to think of this benchmarking pitfall... I mean the branch prediction. Well, I agree that we need to put it in the BufferPainter environment for real test.

Meanwhile, as you worked on 7, 7a and 8, I prepared 3T2, which avoids the switch and uses ifs instead. Funnily, your 7a does the same, but with table offsets.

```
void inline Fill3T2(dword *b, dword data, int len){
 if(len<4){
  if(len&1) *b++ = data;
  if(len&2){ *b++ = data; *b++ = data; }
  return;
 }

 __m128i q = _mm_set1_epi32(*(int*)&data);
 __m128i *w = (__m128i*)b;

 if(len >= 32) {
  __m128i *e = (__m128i*)b + (len>>2) - 8;
  if(len > 4*1024*1024 / 4 && ((uintptr_t)w & 3) == 0) { // for really huge data, bypass the cache
   _mm_storeu_si128(w, q); // Head align
   int s=(-((int)((uintptr_t)b)>>2))&0x3;
   w = (__m128i*) (b + s);
   do {
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
    _mm_stream_si128(w++, q);
   }while(w<=e);
   _mm_sfence();
  }
  else
   do {
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
    _mm_storeu_si128(w++, q);
   }while(w<=e);
 }

 if(len & 16) {
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
 }
 if(len & 8) {
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
 }
 if(len & 4) {
  _mm_storeu_si128(w, q);
 }
 _mm_storeu_si128((__m128i*) (b + len - 4), q); // Tail align
}
```

I really like the w++ incremental pointer logic over the Set4(pointer+offset). This approach seems to give a small improvement on my system.

Next, I will test your 7 + 7a and report against 3T2.

But seriously, we need to put an end to this madness! The bang for the buck is rapidly decreasing as working hours are increasing...

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 19 May 2020 10:45:52 GMT
View Forum Message <> Reply to Message

[quote title=Tom1 wrote on Tue, 19 May 2020 12:35]mirek wrote on Tue, 19 May 2020 10:49
I really like the w++ incremental pointer logic over the Set4(pointer+offset). This approach seems to give a small improvement on my system.

Compiler actually converts that to offsets anyway... (I have checked disassembly).

Quote:
But seriously, we need to put an end to this madness! The bang for the buck is rapidly decreasing as working hours are increasing...

Well, you have started it

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 19 May 2020 11:18:01 GMT
View Forum Message <> Reply to Message

[quote title=mirek wrote on Tue, 19 May 2020 13:45]Tom1 wrote on Tue, 19 May 2020 12:35mirek wrote on Tue, 19 May 2020 10:49
I really like the w++ incremental pointer logic over the Set4(pointer+offset). This approach seems to give a small improvement on my system.

Compiler actually converts that to offsets anyway... (I have checked disassembly).

Quote:
But seriously, we need to put an end to this madness! The bang for the buck is rapidly decreasing as working hours are increasing...

Well, you have started it

Mirek

I admit to it! My fault...

Anyway, pick you choice: 7a or 3T2, but note that MSBT19 (32bit I mean) likes 3T2 better on short transfers. CLANG, CLANGx64 and MSBT19x64 are happy with both. (But, please do your own benchmarks, as this is just my repeated scan through different lengths with the pitfall you pointed out earlier.)

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 19 May 2020 14:22:08 GMT
View Forum Message <> Reply to Message

Hi,

WARNING: Something still wrong in 3T2 alignment code. I will continue to investigate it.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 19 May 2020 23:34:03 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Yes, I'm nuts... still working at this hour.

Anyway, here's a new version - Fill3T3 - that can actually handle all alignment variations (even those not handled by 7a). Please benchmark and check for correctness:

```
never_inline void FillStream(dword *b, dword data, int len){

 while((uintptr_t)b & 15){ // Try to align
  *b++=data;
  len--;
 };
 __m128i *w = (__m128i *)b;
 __m128i q = _mm_set1_epi32((int)data);
 if(len>=16){
  __m128i *e = w + (len>>2) - 3;
  do{
   _mm_stream_si128(w++, q);
   _mm_stream_si128(w++, q);
   _mm_stream_si128(w++, q);
   _mm_stream_si128(w++, q);
  }while(w<e);
 }
 if(len & 8) {
  _mm_stream_si128(w++, q);
  _mm_stream_si128(w++, q);
 }
 if(len & 4) {
  _mm_stream_si128(w++, q);
 }
 _mm_sfence();
 _mm_storeu_si128((__m128i*)(b + len - 4), q); // Tail align
}

void inline Fill3T3(dword *b, dword data, int len){
 if(len<4){
  if(len&1) *b++ = data;
  if(len&2){ *b++ = data; *b++ = data; }
  return;
 }

 __m128i *w = (__m128i *)b;
 __m128i q = _mm_set1_epi32((int)data);

 if(len >= 32) {
  if(len>1024*1024 && (((uintptr_t)b & 3)==0)){
   FillStream(b,data,len);
   return;
  }

  __m128i *e = w + (len>>2) - 7;
  do{
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
   _mm_storeu_si128(w++, q);
  }while(w<e);
 }
 if(len & 16) {
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
  _mm_storeu_si128(w++, q);
 }
 if(len & 8) {
  _mm_storeu_si128(w++, q);
 _mm_storeu_si128(w++, q);
 }
 if(len & 4) {
  _mm_storeu_si128(w++, q);
 }
 _mm_storeu_si128((__m128i*)(b + len - 4), q); // Tail align

}
```

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 19 May 2020 23:52:36 GMT
View Forum Message <> Reply to Message

[quote title=Tom1 wrote on Wed, 20 May 2020 01:34]Hi Mirek,

Yes, I'm nuts... still working at this hour.

Anyway, here's a new version - Fill3T3 - that can actually handle all alignment variations (even those not handled by 7a). Please benchmark and check for correctness:

```
 if(len & 8) {
  _mm_stream_si128(w++, q);
  _mm_stream_si128(w++, q);
 }
 if(len & 4) {
  _mm_stream_si128(w++, q);
 }
```

Yeah, I think that after filling 8MB of data, this will really have impact compared to trivial loop

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 06:22:43 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 20 May 2020 02:52Yeah, I think that after filling 8MB of data, this will really have impact compared to trivial loop

Mirek

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 08:04:35 GMT
View Forum Message <> Reply to Message

Hi,

There must still be something wrong with 3T3 because applying it to BufferPainter (replacing FillRGBA) causes artifacts in drawing. E.g. PainterExamples spiral example at 3x scale clearly shows noise in line edges.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 08:20:56 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Wed, 20 May 2020 10:04Hi,

There must still be something wrong with 3T3 because applying it to BufferPainter (replacing FillRGBA) causes artifacts in drawing. E.g. PainterExamples spiral example at 3x scale clearly shows noise in line edges.

Best regards,

Tom

My guts feeling is either the tail fill, or less likely, "e" computation. I think these are simpler code in Fill7a.... (and actually, these are the only real difference now).

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 08:55:46 GMT
View Forum Message <> Reply to Message

Hi,

No, Sorry... I'll take that alarm back. There is no error in 3T3 after all. My copy of the code inside Painter was faulty... Now I took the correct version and it is all good now.

I'm just too tired after not sleeping too much lately...

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 09:56:41 GMT
View Forum Message <> Reply to Message

So I have replaced memsetd with Fill7a, replaced RGBA fill with (new) memsetd and did benchmarks.

Except the situation where the benchmark involves Clear of large area, numbers have not changed...

EDIT: Bug on my part, retesting...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 10:23:01 GMT
View Forum Message <> Reply to Message

OK, after retesting, I think it might be at most 3% faster. Looking at fillers, I think there is much more time spent in AlphaBlend function - even if it is just for segment start/end pixels. Perhaps that one should be SSE2 optimized?

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 10:41:30 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Two things to consider before you go with 7a:

- 7a crashes on unaligned buffers (t&3) while 3T3 handles them all.

- 3T3 is faster on MSBT19 with short transfers up to 50-60 dwords.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 10:52:53 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 20 May 2020 13:23OK, after retesting, I think it might be at most 3% faster. Looking at fillers, I think there is much more time spent in AlphaBlend function - even if it is just for segment start/end pixels. Perhaps that one should be SSE2 optimized?

Mirek

Hi,

My SSE2 battery is now 'discharged' for a while.... Need to recharge before next use.

I also did some testing on span filler with memcpy. This is based on using IMAGE_OPAQUE of the image being rendered. It does improve the speed somewhat, but the edges cause a problem since the edge is alpha blended even if FILL_FAST is specified. So, this needs some reconsideration and better knowledge on the Painter internals (i.e. beyond my level...):

BufferPainter.h:

```
struct SpanSource {
 int kind;
 SpanSource(){
  kind = IMAGE_OPAQUE;
 }
 virtual void Get(RGBA *span, int x, int y, unsigned len) = 0;
 virtual ~SpanSource() {}
};
```

Fillers.cpp:

```
void SpanFiller::Render(int val, int len)
{
 if(val == 0) {
  t += len;
  s += len;
  return;
 }
 if(alpha != 256)
  val = alpha * val >> 8;
 if(val == 256) {
  if(ss->kind==IMAGE_OPAQUE) memcpy(t,s,len*sizeof(RGBA)); // apex_memcpy() would be
even faster
  else{
   for(int i = 0; i < len; i++) {
    if(s[i].a == 255)
     t[i] = s[i];
    else
     AlphaBlend(t[i], s[i]);
   }
  }
  t += len;
  s += len;
 }
 else {
  const RGBA *e = t + len;
  while(t < e)
   AlphaBlendCover8(*t++, *s++, val);
 }
}
```

Painter/Image.cpp:

```
struct PainterImageSpan : SpanSource, PainterImageSpanData {
 LinearInterpolator interpolator;

 PainterImageSpan(const PainterImageSpanData& f)
 : PainterImageSpanData(f) {
  interpolator.Set(xform);
  kind = image.GetKindNoScan(); // Add this
 }
```

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 10:53:01 GMT

View Forum Message <> Reply to Message

I was aware about unaligned problem, thats fixed in final version. That said, unaligned in general should be considered illegal anyway, because otherwise hell will broke lose with Armv6....

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 11:01:50 GMT
View Forum Message <> Reply to Message

Quote:I was aware about unaligned problem, thats fixed in final version. That said, unaligned in general should be considered illegal anyway, because otherwise hell will broke lose with Armv6....

But that's good to know. In this case we could drop (t&3) code entirely from 3T3 and improve instruction cache locality for even better results on short transfers.

((Is there a way to 'cleanly crash' (whatever that might mean) an application attempting unaligned memset? Now it just disappears from the process list at least on Windows.))

EDIT: Let me rephrase it: Is there a way to check during development that an application will never use unaligned memset?

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 13:18:19 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Wed, 20 May 2020 13:01
EDIT: Let me rephrase it: Is there a way to check during development that an application will never use unaligned memset?

memsetd!

Yes, put ASSERT(((uintptr_t)t & 3) == 0); to memsetd

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 13:58:30 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Wed, 20 May 2020 12:41
- 3T3 is faster on MSBT19 with short transfers up to 50-60 dwords.

Interestingly, adding "inline" to it seems to fix the problem... For some reason, 32-bit MSC does not inline it unless you ask it to do so...

In fact, assembler for both inlined function is virtually the same, the only difference is different tail handling (IMO, mine is 1% esier on eye):

7a:

```
0017940D  cmp ecx,byte +0x4
00179410  jnl 0x17942f
00179412  test cl,0x2
00179415  jz 0x17941f
00179417  mov [eax+0x4],edi
0017941A  mov [eax],edi
0017941C  add eax,byte +0x8
0017941F  test cl,0x1
00179422  jz dword 0x1794b4
00179428  mov [eax],edi
0017942A  jmp dword 0x1794b4
0017942F  movd xmm0,edi
00179433  pshufd xmm0,xmm0,0x0
00179438  movups [eax+ecx*4-0x10],xmm0     <<= tail handling
0017943D  cmp ecx,byte +0x20
00179440  jl 0x179486
00179442  cmp ecx,0x100000
00179448  jl 0x179457
0017944A  push ecx
0017944B  push edi
0017944C  push eax
0017944D  call dword 0x14ff88
00179452  add esp,byte +0xc
00179455  jmp short 0x1794b4
00179457  lea edx,[ecx-0x20]
0017945A  lea edx,[eax+edx*4]
0017945D  nop dword [eax]
00179460  movups [eax],xmm0
00179463  movups [eax+0x10],xmm0
00179467  movups [eax+0x20],xmm0
0017946B  movups [eax+0x30],xmm0
0017946F  movups [eax+0x40],xmm0
00179473  movups [eax+0x50],xmm0
00179477  movups [eax+0x60],xmm0
0017947B  movups [eax+0x70],xmm0
0017947F  sub eax,byte -0x80
00179482  cmp eax,edx
00179484  jna 0x179460
00179486  test cl,0x10
00179489  jz 0x17949d
0017948B  movups [eax],xmm0
0017948E  movups [eax+0x10],xmm0
```

```
00179492  movups [eax+0x20],xmm0
00179496  movups [eax+0x30],xmm0
0017949A  add eax,byte +0x40
0017949D  test cl,0x8
001794A0  jz 0x1794ac
001794A2  movups [eax],xmm0
001794A5  movups [eax+0x10],xmm0
001794A9  add eax,byte +0x20
001794AC  test cl,0x4
001794AF  jz 0x1794b4
001794B1  movups [eax],xmm0
```

3T3

```
00179540  cmp eax,byte +0x4
00179543  jnl 0x179560
00179545  test al,0x1
00179547  jz 0x17954e
00179549  mov [edx],edi
0017954B  add edx,byte +0x4
0017954E  test al,0x2
00179550  jz dword 0x179607
00179556  mov [edx],edi
00179558  mov [edx+0x4],edi
0017955B  jmp dword 0x179607
00179560  movd xmm0,edi
00179564  mov ecx,edx
00179566  pshufd xmm0,xmm0,0x0
0017956B  cmp eax,byte +0x20
0017956E  jl 0x1795c6
00179570  cmp eax,0x100000
00179575  jng 0x179589
00179577  test dl,0x3
0017957A  jnz 0x179589
0017957C  push eax
0017957D  push edi
0017957E  push edx
0017957F  call dword 0x14ff88
00179584  add esp,byte +0xc
00179587  jmp short 0x179604
00179589  mov edi,eax
0017958B  sar edi,0x2
0017958E  sub edi,byte +0x7
00179591  shl edi,0x4
00179594  add edi,edx
00179596  mov eax,ecx
00179598  movups [eax],xmm0
```

```
0017959B  lea eax,[ecx+0x70]
0017959E  movups [ecx+0x10],xmm0
001795A2  movups [ecx+0x20],xmm0
001795A6  movups [ecx+0x30],xmm0
001795AA  movups [ecx+0x40],xmm0
001795AE  movups [ecx+0x50],xmm0
001795B2  movups [ecx+0x60],xmm0
001795B6  sub ecx,byte -0x80
001795B9  movups [eax],xmm0
001795BC  cmp ecx,edi
001795BE  jc 0x179596
001795C0  mov eax,[ebp-0x14]
001795C3  mov edi,[ebp-0x18]
001795C6  test al,0x10
001795C8  jz 0x1795e3
001795CA  mov eax,ecx
001795CC  movups [eax],xmm0
001795CF  lea eax,[ecx+0x30]
001795D2  movups [ecx+0x10],xmm0
001795D6  movups [ecx+0x20],xmm0
001795DA  add ecx,byte +0x40
001795DD  movups [eax],xmm0
001795E0  mov eax,[ebp-0x14]
001795E3  test al,0x8
001795E5  jz 0x1795f8
001795E7  mov eax,ecx
001795E9  movups [eax],xmm0
001795EC  lea eax,[ecx+0x10]
001795EF  add ecx,byte +0x20
001795F2  movups [eax],xmm0
001795F5  mov eax,[ebp-0x14]
001795F8  test al,0x4
001795FA  jz 0x1795ff
001795FC  movups [ecx],xmm0
001795FF  movups [edx+eax*4-0x10],xmm0     <= TAIL
```

EDIT: OK, now rechecking it, it looks like 3T3 has a bit more instructions doing weird things....

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 14:15:59 GMT

View Forum Message <> Reply to Message

Hi,

This is strange, since I immediately added the inline to 7a when I started testing it. (I found out earlier that MSBT19 did not do it for me.) Now I did a new run and the result is in the attached csv.

Can you post the latest 7a if it is any different compared to the one posted here above?

Best regards,

Tom

File Attachments

1) memset.csv, downloaded 323 times

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 15:16:51 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Wed, 20 May 2020 16:15Hi,

This is strange, since I immediately added the inline to 7a when I started testing it. (I found out earlier that MSBT19 did not do it for me.) Now I did a new run and the result is in the attached csv.

Can you post the latest 7a if it is any different compared to the one posted here above?

Best regards,

Tom

It is now in trunk as memsetd....

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 15:31:48 GMT
View Forum Message <> Reply to Message

I am getting quite different picture:

```
 int bsize=8*1024*1024;
 Buffer<dword> b(bsize, 0);
 dword cw = 123;

 String result="\"N\",\"memsetd()\",\"Fill3T3()\"\r\n";
 for(int len=1;len<=bsize;){
  int maximum=100000000/len;
  int64 t0=usecs();
  for(int i = 0; i < maximum; i++)
   memsetd(~b, cw, len);
  int64 t1=usecs();
  for(int i = 0; i < maximum; i++)
   Fill3T3(~b, cw, len);
  int64 t2=usecs();
  String r = Format("%d,%f,%f",len,1000.0*(t1-t0)/maximum,1000.0*(t2-t1)/maximum);
  RLOG(r);
  result.Cat(r + "\r\n");
  if(len<64) len++;
  else len*=2;
 }

 SaveFile(GetHomeDirFile("memset.csv"),result);
```

I am starting to wonder if there is difference between our MSC 32bit compilers...

File Attachments

1) memset.csv, downloaded 301 times

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Wed, 20 May 2020 15:37:16 GMT

View Forum Message <> Reply to Message

Ha, funny. It depends on order of functions tested. If I test memsetd second, I am getting different numbers

---

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Wed, 20 May 2020 17:51:10 GMT

View Forum Message <> Reply to Message

mirek wrote on Wed, 20 May 2020 18:31I am getting quite different picture:

```
int bsize=8*1024*1024;
Buffer<dword> b(bsize, 0);
dword cw = 123;

String result="\"N\",\"memsetd()\",\"Fill3T3()\"\r\n";
for(int len=1;len<=bsize;){
 int maximum=100000000/len;
 int64 t0=usecs();
 for(int i = 0; i < maximum; i++)
  memsetd(~b, cw, len);
 int64 t1=usecs();
```

```
  for(int i = 0; i < maximum; i++)
   Fill3T3(~b, cw, len);
  int64 t2=usecs();
  String r = Format("%d,%f,%f",len,1000.0*(t1-t0)/maximum,1000.0*(t2-t1)/maximum);
  RLOG(r);
  result.Cat(r + "\r\n");
  if(len<64) len++;
  else len*=2;
 }

 SaveFile(GetHomeDirFile("memset.csv"),result);
```

I am starting to wonder if there is difference between our MSC 32bit compilers...

Hi,

No wonder we ended up with (very slightly) different approach... Your results are more or less reversed to what I'm getting. I tried to reorder the calls too, but without any observable difference.

It's either the different CPUs or a different compiler. My compiler is:

Microsoft (R) C/C++ Optimizing Compiler Version 19.21.27702.2 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

Should I downgrade or upgrade?...

Anyway, seriously I'm pleased with the final result here. The filler is now better than anything before and can be used generally for all clearing/presetting of buffers. I use this a lot in signal processing in addition to clearing the ImageBuffer for BufferPainter. After all, the ImageBuffer needs to be cleared or preset to user preference background color once before each display update. It is much better to have a 1.5 ms delay instead of 3.6 ms delay before drawing approximately 10-20 ms worth of vector map data on the screen.

Should this new memsetd() now be deployed all over the u++? I mean e.g. Core/Topt.h :: Fill?

Thank you a lot for your great work on this!

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Thu, 21 May 2020 07:04:01 GMT

View Forum Message <> Reply to Message

[quote title=Tom1 wrote on Wed, 20 May 2020 19:51]mirek wrote on Wed, 20 May 2020 18:31

Should this new memsetd() now be deployed all over the u++? I mean e.g. Core/Topt.h :: Fill?

IDK, maybe as specialisation...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Thu, 21 May 2020 11:28:28 GMT
View Forum Message <> Reply to Message

OK, so I could not stop digging and found last important ingredient: alignment matters!

```
void FillX(void *p, dword data, int len)
{
 dword *t = (dword *)p;
 if(len < 4) {
  if(len & 2) {
   t[0] = t[1] = t[len - 1] = data;
   return;
  }
  if(len & 1)
   t[0] = data;
  return;
 }

 __m128i val4 = _mm_set1_epi32(data);
 auto Set4 = [&](int at) { _mm_storeu_si128((__m128i *)(t + at), val4); };

 Set4(len - 4); // fill tail
 if(len >= 16) {
  Set4(0); // align up on 16 bytes boundary
  const dword *e = t + len;
  t = (dword *)(((uintptr_t)t | 15) + 1);
  len = e - t;
  e -= 16;
  if(len >= 1024*1024) { // for really huge data, bypass the cache
   huge_memsetd(t, data, len);
   return;
  }
  while(t <= e) {
   Set4(0); Set4(4); Set4(8); Set4(12);
   t += 16;
  }
 }
 if(len & 8) {
  Set4(0); Set4(4);
  t += 8;
 }
 if(len & 4)
  Set4(0);
}
```

This is about twice as fast as Fill7a for len > 60 (up to cache bypass limit).

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Thu, 21 May 2020 14:21:30 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 20 May 2020 16:18Tom1 wrote on Wed, 20 May 2020 13:01
EDIT: Let me rephrase it: Is there a way to check during development that an application will never use unaligned memset?

memsetd!

Yes, put ASSERT(((uintptr_t)t & 3) == 0); to memsetd

Mirek

Good point! Please do!

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Thu, 21 May 2020 14:38:20 GMT
View Forum Message <> Reply to Message

Hi,

This new FillX is incredibly elegant! Congratulations Mirek! I really do like your new findings there. You just need to rename it as memsetd() and place in the correct header in Core...

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by koldo on Thu, 21 May 2020 15:51:42 GMT
View Forum Message <> Reply to Message

Thank you all for your job. Although please review this in Redmine.

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Thu, 21 May 2020 17:22:43 GMT
View Forum Message <> Reply to Message

Hi Koldo,

I checked and #include <emmintrin.h> seems to work just fine for what we are working on. Thanks for pointing this out.

Mirek: Agree?

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Thu, 21 May 2020 17:25:51 GMT
View Forum Message <> Reply to Message

Mirek,

I just found that there is a sweet spot at ~0x3f alignment (i.e. 64 bytes) on my CPU. This is presumably the L1 cache line length, if I'm not mistaken.

Best regards,

Tom

EDIT: It just looks that I cannot squeeze the benefit out as re-alignment code tends to eat what would could possibly be achieved here. However, if allocator could allocate large blocks at even 64 byte limits, that could improve performance behind the scenes.

Subject: Re: BufferPainter::Clear() optimization
Posted by Didier on Fri, 22 May 2020 07:32:09 GMT
View Forum Message <> Reply to Message

Hello mirek ans Tom,
Grenat work hère but I have une simple question: what is the point with cache ?

Normally cache speeds things up when you need to reaccess data just After writing it.
So filling a buffer with a constant value that is not read immediatly After in most cases isn't a corresponding use case.
So, I think that having a fill function that doesn't use cache at all will benefit in two points:
Timing stability and more importantly, cache is not touched so it can speed up other functions calls further

---

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 08:04:03 GMT
View Forum Message <> Reply to Message

Didier wrote on Fri, 22 May 2020 09:32Hello mirek ans Tom,
Grenat work hère but I have une simple question: what is the point with cache ?
Normally cache speeds things up when you need to reaccess data just After writing it.
So filling a buffer with a constant value that is not read immediatly After in most cases isn't a corresponding use case.
So, I think that having a fill function that doesn't use cache at all will benefit in two points:
Timing stability and more importantly, cache is not touched so it can speed up other functions calls further

Thing that started this whole issue: If you need to clear buffer for 4K screen, that is about 32MB of data. Thats definitely more than can fit into the cache. So what really happens in that in this case is that at some point cache runs out and you are significantly slowed down by CPU writing data from the cache to main memory. The "fix" is to bypass the cache in this case (we have for now established that the reasonable threshold is somewhere around 4MB).

That said, really a lot of other things were optimised thereafter, mostly on the other size of size spectrum...

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 08:05:49 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Thu, 21 May 2020 19:25
EDIT: It just looks that I cannot squeeze the benefit out as re-alignment code tends to eat what would could possibly be achieved here. However, if allocator could allocate large blocks at even 64 byte limits, that could improve performance behind the scenes.

It cannot as alignment is important part of block information...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 08:28:24 GMT
View Forum Message <> Reply to Message

So I have implemented a bunch of other functions based on info gathered during this session:

memcpyd
svo_memset
svo_memcpy

Now I have hopefully the last problem to tune... I have tried to put svo_memcpy to Vector::Add grow routine and it indeed improved performance a bit. Then tried to improve this even more and put memcpyd (which svo_memcpy is using as backend in some situations) and performance dropped.

I believe that the problem is that memcpyd became too fat and it screws inlining. So the thing to solve now is to find how to remove some if this fat to non-inline.... (svo_memcpy already has such non-inlined part). Probably same should happend to memsetd too....

Subject: Re: BufferPainter::Clear() optimization
Posted by koldo on Fri, 22 May 2020 08:29:25 GMT
View Forum Message <> Reply to Message

One question. To use these new features, is it necessary to set compiler flags, like /arch:AVX in Visual Studio?

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 22 May 2020 09:13:48 GMT
View Forum Message <> Reply to Message

Quote:I believe that the problem is that memcpyd became too fat and it screws inlining. So the thing to solve now is to find how to remove some if this fat to non-inline.... (svo_memcpy already has such non-inlined part). Probably same should happend to memsetd too....

Hi Mirek,

I think this could be the same phenomenon that caused me issues with 32-bit MSC. It was more critical to code length and the short transfers suffered immediately when code size increased. At the same time MSBT19x64 and both CLANG and CLANGx64 did not experience any trouble. Perhaps MSBT19 did not do as good job with code size as the rest and on my CPU the instruction cache was exhausted. I bet the instruction cache on your CPU is larger than what my i7 has.

At some moment I was thinking of offering the functions as two variants: inline and never_inline, in a way that the never_inline is simply calling the inline. An then when the code benefits from it, calling the never_inline variant.

Then I also thought of handling something like <= 16 .. 32 sizes inline and the rest in a deeper never_inline function. This would probably improve the situation without adding so much complexity.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 09:32:12 GMT
View Forum Message <> Reply to Message

koldo wrote on Fri, 22 May 2020 10:29One question. To use these new features, is it necessary to set compiler flags, like /arch:AVX in Visual Studio?

No so far. This is just SSE2, which is enabled by default for ages now...

Of course, the next logical step is to use AVX256

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 22 May 2020 09:32:51 GMT
View Forum Message <> Reply to Message

koldo wrote on Fri, 22 May 2020 11:29One question. To use these new features, is it necessary to set compiler flags, like /arch:AVX in Visual Studio?

Hi Koldo,

Here I do not need /arch:AVX or any other compiler flag added. It's just that include (#include <smmintrin.h> or #include <emmintrin.h>, which works for me, I think.

Best regards,

Tom

EDIT: Mirek was faster to respond!

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 09:39:48 GMT

View Forum Message <> Reply to Message

[quote title=Tom1 wrote on Fri, 22 May 2020 11:13]Quote:
Then I also thought of handling something like <= 16 .. 32 sizes inline and the rest in a deeper never_inline function. This would probably improve the situation without adding so much complexity.

In the trunk now... >=16 now handled by non-inline function. There is impact in your benchmark (the one that runs for all sizes), less impact in my benchmark (with ransom sizes), but I think this is the right move...

Another benefit is that we can now consider using AVX (testing for AVX presence would be clumsy in inline function I think).

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 22 May 2020 09:46:29 GMT
View Forum Message <> Reply to Message

[quote title=mirek wrote on Fri, 22 May 2020 12:39]Tom1 wrote on Fri, 22 May 2020 11:13Quote:
Then I also thought of handling something like <= 16 .. 32 sizes inline and the rest in a deeper never_inline function. This would probably improve the situation without adding so much complexity.

In the trunk now... >=16 now handled by non-inline function. There is impact in your benchmark (the one that runs for all sizes), less impact in my benchmark (with ransom sizes), but I think this is the right move...

Another benefit is that we can now consider using AVX (testing for AVX presence would be clumsy in inline function I think).

Mirek

The apex_memmove() did the architecture checking on startup (or first run) and then initialized function pointers to optimal versions. I think we could do this too in some INITBLOCK.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 22 May 2020 09:59:21 GMT
View Forum Message <> Reply to Message

mirek wrote on Fri, 22 May 2020 12:39
In the trunk now... >=16 now handled by non-inline function. There is impact in your benchmark (the one that runs for all sizes), less impact in my benchmark (with ransom sizes), but I think this is the right move...

It looks like >32 might be better in this case... Not sure though.

BR, Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by koldo on Fri, 22 May 2020 10:47:03 GMT
View Forum Message <> Reply to Message

Dear colleagues

Please consider Sender proposal:
- Remove #include <emmintrin.h> from Blit.h
- Include #include <immintrin.h> in config.h

As now the intrinsics are included inside Upp namespace, they cannot be used later by Eigen.
config.h is included in Core.h before Upp namespace.

Thank you!

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 11:01:25 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Fri, 22 May 2020 11:59mirek wrote on Fri, 22 May 2020 12:39
In the trunk now... >=16 now handled by non-inline function. There is impact in your benchmark (the one that runs for all sizes), less impact in my benchmark (with ransom sizes), but I think this is the right move...

It looks like >32 might be better in this case... Not sure though.

BR, Tom

It in turn makes inlined part bigger.... I would rather be careful there.

OK, for what is worth, I have tried with AVX and I do not see any improvement. Here is the code (for CLANG):

```
__attribute__((target ("avx")))
never_inline
void memsetd_l2(dword *t, dword data, size_t len)
{
 __m128i val4 = _mm_set1_epi32(data);
 __m256i val8 = _mm256_set1_epi32(data);
 auto Set4 = [&](size_t at) { _mm_storeu_si128((__m128i *)(t + at), val4); };
 #define Set8(at) _mm256_storeu_si256((__m256i *)(t + at), val8);
 Set4(len - 4); // fill tail
 if(len >= 32) {
  if(len >= 1024*1024) { // for really huge data, bypass the cache
   huge_memsetd(t, data, len);
   return;
  }
  Set8(0); // align up on 16 bytes boundary
  const dword *e = t + len;
  t = (dword *)(((uintptr_t)t | 31) + 1);
  len = e - t;
  e -= 32;
  while(t <= e) {
   Set8(0); Set8(8); Set8(16); Set8(24);
   t += 32;
  }
 }
 if(len & 16) {
  Set8(0); Set8(8);
  t += 16;
 }
 if(len & 8) {
  Set8(0);
  t += 8;
 }
 if(len & 4)
  Set4(0);
}

inline
void FillX(void *p, dword data, size_t len)
{
 dword *t = (dword *)p;
 if(len < 4) {
  if(len & 2) {
   t[0] = t[1] = t[len - 1] = data;
   return;
  }
  if(len & 1)
   t[0] = data;
  return;
 }

 if(len >= 16) {
  memsetd_l2(t, data, len);
  return;
 }

 __m128i val4 = _mm_set1_epi32(data);
 auto Set4 = [&](size_t at) { _mm_storeu_si128((__m128i *)(t + at), val4); };
 Set4(len - 4); // fill tail
 if(len & 8) {
  Set4(0); Set4(4);
  t += 8;
 }
 if(len & 4)
  Set4(0);
}
```

Frankly I am sort of happy, because GCC/CLANG way of dealing with AVX is really stupid: It declines AVX instrinics, unless you compile whole function for AVX code, but then it starts generating AVX opcodes everywhere and the funciton does not run on non-AVX CPUs anymore.

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 22 May 2020 11:06:14 GMT
View Forum Message <> Reply to Message

Quote: I have tried with AVX and I do not see any improvement.

So, this means SSE2 is enough to saturate the memory bus completely.

Thanks also for the new memcpy optimizations. This is equally important in many areas.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by koldo on Fri, 22 May 2020 14:58:14 GMT
View Forum Message <> Reply to Message

Problem solved. Thank you!

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 22 May 2020 17:03:16 GMT
View Forum Message <> Reply to Message

Added memcpy optimized for sizeof 8 and 16 and this little neat function to make sense from it all:

```
template <class T>
void memcpy_t(T *t, const T *s, size_t count)
{
 if((sizeof(T) & 15) == 0)
  memcpydq((dqword *)t, (const dqword *)s, count * (sizeof(T) >> 4));
 else
 if((sizeof(T) & 7) == 0)
  memcpyq((qword *)t, (const qword *)s, count * (sizeof(T) >> 3));
 else
 if((sizeof(T) & 3) == 0)
  memcpyd((dword *)t, (const dword *)s, count * (sizeof(T) >> 2));
 else
  svo_memcpy((void *)t, (void *)s, count * sizeof(T));
}
```

Vector<String>::ReAlloc(int newalloc)

disassembly now looks magnificent, copying elements to new buffer with SSE2...

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sun, 24 May 2020 08:20:45 GMT
View Forum Message <> Reply to Message

I have the first implementation and test of SSE2 AlphaBlend:

```
TIMING SSE          : 46.95 ms - 46.95 ns (58.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
TIMING Non SSE      : 123.95 ms - 123.95 ns (135.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
```

File Attachments
1) AlphaBlendSSE2.cpp, downloaded 298 times

Subject: Re: BufferPainter::Clear() optimization
Posted by Oblivion on Sun, 24 May 2020 09:56:01 GMT
View Forum Message <> Reply to Message

Hello Mirek,

On Linux 5.4 and 5.6, with CLANG 10.0

```
TIMING SSE          : 119.41 ms - 119.41 ns ( 1.06 s  / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
TIMING Non SSE      : 232.41 ms - 232.41 ns ( 1.18 s  / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
```

On GCC (10.1): apparently _mm_storeu_si32 is yet to be implemented. :

```
'_mm_storeu_si32' was not declared in this scope; did you mean '_mm_storeu_epi32'?
 (): 47 |  _mm_storeu_si32(rgba, PackRGBA(x, _mm_setzero_si128()));
 (): |  ^~~~~~~~~~~~~~~
 (): |  _mm_storeu_epi32
```

Possible workaround is given here:

https://stackoverflow.com/questions/58063933/how-can-a-sse2-function-be-missing-from-the-header-it-is-supposed-to-be-in

Best regards,
Oblivion

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 26 May 2020 11:14:43 GMT
View Forum Message <> Reply to Message

Hi!

Sorry for the delay... I was out of town for a while.

Here are my results for Windows 10 pro x64 on Core i7:

MSBT19x64:

TIMING SSE   : 37.08 ms - 37.08 ns (50.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000
TIMING Non SSE   : 129.08 ms - 129.08 ns (142.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000

MSBT19:

TIMING SSE   : 29.88 ms - 29.88 ns (45.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000
TIMING Non SSE   : 125.88 ms - 125.88 ns (141.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000

CLANG:

TIMING SSE   : 37.41 ms - 37.41 ns (50.00 ms / 1000000), min: 0.00 ns, max: 2.00 ms, nesting: 0 - 1000000
TIMING Non SSE   : 125.41 ms - 125.41 ns (138.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000

CLANGx64:

TIMING SSE   : 37.43 ms - 37.43 ns (47.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000
TIMING Non SSE   : 129.43 ms - 129.43 ns (139.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms, nesting: 0 - 1000000

Impressive numbers Mirek! When is this going to be available on BufferPainter?

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 26 May 2020 12:15:32 GMT

View Forum Message <> Reply to Message

Tom1 wrote on Tue, 26 May 2020 13:14Hi!

Sorry for the delay... I was out of town for a while.

Here are my results for Windows 10 pro x64 on Core i7:

MSBT19x64:

TIMING SSE   : 37.08 ms - 37.08 ns (50.00 ms / 1000000), min: 0.00 ns, max: 1.00 ms,

```
nesting: 0 - 1000000
TIMING Non SSE       : 129.08 ms - 129.08 ns (142.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
```

MSBT19:

```
TIMING SSE           : 29.88 ms - 29.88 ns (45.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
TIMING Non SSE       : 125.88 ms - 125.88 ns (141.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
```

CLANG:

```
TIMING SSE           : 37.41 ms - 37.41 ns (50.00 ms / 1000000 ), min:  0.00 ns, max:  2.00 ms, nesting: 0 - 1000000
TIMING Non SSE       : 125.41 ms - 125.41 ns (138.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
```

CLANGx64:

```
TIMING SSE           : 37.43 ms - 37.43 ns (47.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
TIMING Non SSE       : 129.43 ms - 129.43 ns (139.00 ms / 1000000 ), min:  0.00 ns, max:  1.00 ms, nesting: 0 - 1000000
```

Impressive numbers Mirek! When is this going to be available on BufferPainter?

Best regards,

Tom

I guess by the end of the week. Still fixing bugs + there is like 8 variants to implement...

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sun, 31 May 2020 22:39:13 GMT
View Forum Message <> Reply to Message

While optimizing memcpy and memset, I have tried a new look at othre things, like String comparison and memhash. I think I improved String::operator== a tiny bit and now I am working on memhash function. Decided to introduce "hash_t" and to have hash value 64 bit when CPU_64.

After bit of experimenting, I have found these functions (one for 64 bit, other 32 bit) work best:

```
never_inline
uint64 memhash64(const void *ptr, int len)
{
 const byte *s = (byte *)ptr;
 uint64 val = HASH64_CONST1;
 if(len >= 8) {
  if(len >= 32) {
   uint64 val1, val2, val3, val4;
   val1 = val2 = val3 = val4 = HASH64_CONST1;
   while(len >= 32) {
    val1 = HASH64_CONST2 * val1 + *(qword *)(s);
    val2 = HASH64_CONST2 * val2 + *(qword *)(s + 8);
    val3 = HASH64_CONST2 * val3 + *(qword *)(s + 16);
    val4 = HASH64_CONST2 * val4 + *(qword *)(s + 24);
    s += 32;
    len -= 32;
   }
   val = HASH64_CONST2 * val + val1;
   val = HASH64_CONST2 * val + val2;
   val = HASH64_CONST2 * val + val3;
   val = HASH64_CONST2 * val + val4;
  }
  const byte *e = s + len - 8;
  while(s < e) {
   val = HASH64_CONST2 * val + *(qword *)(s);
   s += 8;
  }
  return HASH64_CONST2 * val + *(qword *)(e);
 }
 if(len > 4) {
  val = HASH64_CONST2 * val + *(dword *)(s);
  val = HASH64_CONST2 * val + *(dword *)(s + len - 4);
  return val;
 }
 if(len >= 2) {
  val = HASH64_CONST2 * val + *(word *)(s);
  val = HASH64_CONST2 * val + *(word *)(s + len - 2);
  return val;
 }
 return len ? HASH64_CONST2 * val + *s : val;
}

never_inline
uint64 memhash32(const void *ptr, int len)
{
 const byte *s = (byte *)ptr;
 uint64 val = HASH32_CONST1;
 if(len >= 4) {
  if(len >= 16) {
   uint64 val1, val2, val3, val4;
   val1 = val2 = val3 = val4 = HASH32_CONST1;
   while(len >= 32) {
    val1 = HASH32_CONST2 * val1 + *(dword *)(s);
    val2 = HASH32_CONST2 * val2 + *(dword *)(s + 4);
    val3 = HASH32_CONST2 * val3 + *(dword *)(s + 8);
    val4 = HASH32_CONST2 * val4 + *(dword *)(s + 12);
    s += 16;
    len -= 16;
   }
   val = HASH32_CONST2 * val + val1;
   val = HASH32_CONST2 * val + val2;
   val = HASH32_CONST2 * val + val3;
   val = HASH32_CONST2 * val + val4;
  }
  const byte *e = s + len - 4;
  while(s < e) {
   val = HASH32_CONST2 * val + *(dword *)(s);
   s += 4;
  }
  return HASH32_CONST2 * val + *(dword *)(e);
 }
 if(len >= 2) {
  val = HASH32_CONST2 * val + *(word *)(s);
  val = HASH32_CONST2 * val + *(word *)(s + len - 2);
  return val;
 }
 return len ? HASH32_CONST2 * val + *s : val;
}
```

While other "mem*" functions are easy to write tests for, hasing is a bit more complicated; can I request some code review here? Basically, I think combination functions are OK, but I would like to be sure it reads exactly len bytes from memory (it is ok if some are read twice...).

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by omari on Mon, 01 Jun 2020 09:24:50 GMT
View Forum Message <> Reply to Message

in uint64 memhash32(const void *ptr, int len)

while(len >= 16) {
instead of
while(len >= 32) {

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Mon, 01 Jun 2020 13:47:18 GMT
View Forum Message <> Reply to Message

Well, that is intentional - it is not worth the effort of final combining unless there is more memory to process.

In the end, 32bit variant is for now:

```
hash_t memhash(const void *ptr, size_t len)
{
 const byte *s = (byte *)ptr;
 dword val = HASH32_CONST1;
 if(len >= 4) {
  if(len >= 16) {
   dword val1, val2;
   val1 = val2 = HASH32_CONST1;
   while(len >= 8) {
    val1 = HASH32_CONST2 * val1 + *(dword *)(s);
    val2 = HASH32_CONST2 * val2 + *(dword *)(s + 4);
    s += 8;
    len -= 8;
   }
   val = HASH32_CONST2 * val + val1;
   val = HASH32_CONST2 * val + val2;
  }
  const byte *e = s + len - 4;
  while(s < e) {
   val = HASH32_CONST2 * val + *(dword *)(s);
   s += 4;
  }
  return HASH32_CONST2 * val + *(dword *)(e);
 }
 if(len >= 2) {
  val = HASH32_CONST2 * val + *(word *)(s);
  val = HASH32_CONST2 * val + *(word *)(s + len - 2);
  return val;
 }
 return len ? HASH32_CONST2 * val + *s : val;
}
```

(I have for now reduced that to 8 bytes being processed as I am afraid about register pressure there - not enough registers in 386 ISA. Perhaps needs more testing...)

Subject: Re: BufferPainter::Clear() optimization

Posted by Tom1 on Tue, 02 Jun 2020 11:59:53 GMT
View Forum Message <> Reply to Message

Hi Mirek,

What's the current status of the new BufferPainter optimizations? More specifically, the AlphaBlend variants. Are they on their way to the BufferPainter?

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Tue, 02 Jun 2020 15:43:21 GMT
View Forum Message <> Reply to Message

Well, somehow I dug myself into more mem* (memeq*, memhash) functions and optimisations (going 64 bit hashes)... Hopefully all is done for now (except in future, I plan to do aarch64 and NEON optimizations too).

I think I will be able to return to AlphaBlend soon.

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Tue, 02 Jun 2020 16:31:54 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Thanks for the update. I'll stay tuned on this channel.

Best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Thu, 04 Jun 2020 15:23:37 GMT
View Forum Message <> Reply to Message

SSE2 alphablending comitted. I see 10% improvements in heavily blended example. Looks like low-hanging fruits are long gone

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Thu, 04 Jun 2020 15:45:18 GMT
View Forum Message <> Reply to Message

OK, that might have been a bit too pesimistic, in some other examples the speedup is noticeable. Somewhat expected thing however is that this is more in single-threaded mode, less in MT.

Note: I have added "NOSIMD" flag to make it possible to turn the new code off.

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Novo on Thu, 04 Jun 2020 16:07:37 GMT
View Forum Message <> Reply to Message

Problem with Mac 10.13:

```
/Users/ssg/.local/soft/bb-worker/worker/m-upp/build/uppsrc/Painter/AlphaBlend.h:57:2: error: use of undeclared identifier '_mm_storeu_si64'
        _mm_storeu_si64(rgba, PackRGBA(x, _mm_setzero_si128()));
        ^
```

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Thu, 04 Jun 2020 16:48:50 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 04 June 2020 18:23SSE2 alphablending comitted. I see 10% improvements in heavily blended example. Looks like low-hanging fruits are long gone

Mirek

Hi Mirek,

Thanks! This is a welcome improvement. When rendering complex maps with MT, I see an overall improvement of 4.. 20 % depending on the contents. None of the geometries are transparent themselves, but the edges of strokes and fills likely do benefit from this.

Having the improvement more on the ST side is nice to have as (soft) real-time processes get less disturbed by the GUI being rendered by the BufferPainter running in ST.

Thanks and best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Thu, 04 Jun 2020 18:20:34 GMT
View Forum Message <> Reply to Message

Novo wrote on Thu, 04 June 2020 18:07Problem with Mac 10.13:

```
/Users/ssg/.local/soft/bb-worker/worker/m-upp/build/uppsrc/Painter/AlphaBlend.h:57:2: error: use of undeclared identifier '_mm_storeu_si64'
        _mm_storeu_si64(rgba, PackRGBA(x, _mm_setzero_si128()));
        ^
```

Should be now, eh... workarounded.

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 12 Jun 2020 10:23:09 GMT
View Forum Message <> Reply to Message

I have finally figured out how to SSE2 optimize ImageSpan code, so we have now about 20% boost when rendering Images in Painter with bilinear interpolation...

Subject: Re: BufferPainter::Clear() optimization
Posted by Tom1 on Fri, 12 Jun 2020 10:55:58 GMT
View Forum Message <> Reply to Message

Hi Mirek,

Thanks! This also seems to improve FILL_FAST speed. Was this expected?

Now when comparing between 2020.1 and this latest enhancement altogether, rendering an ImageBuffer by first clearing it and then adding a large raster image with FILL_FAST is now down at 2.8 ms from 4.4 ms!

Thanks and best regards,

Tom

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Fri, 12 Jun 2020 14:28:04 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Fri, 12 June 2020 12:55Hi Mirek,

Thanks! This also seems to improve FILL_FAST speed. Was this expected?

Was not quite expected, but was noticed... Looks like trivial FP solution beats integer tricks...

Mirek

Subject: Re: BufferPainter::Clear() optimization
Posted by Novo on Fri, 12 Jun 2020 16:45:12 GMT

View Forum Message <> Reply to Message

Could you please fix a compilation error on Mac? It was introduced a couple of days ago.

```
In file included from /Users/ssg/.local/soft/bb-worker/worker/m-upp/build/uppsrc/Core/App.cpp:4:
In file included from /usr/include/mach-o/dyld.h:31:
/usr/include/mach-o/loader.h:56:2: error: unknown type name 'cpu_type_t'; did you mean
'Upp::cpu_type_t'?
        cpu_type_t      cputype;        /* cpu specifier */
        ^
/usr/include/mach/machine.h:70:19: note: 'Upp::cpu_type_t' declared here
typedef integer_t       cpu_type_t;
                  ^
```

TIA

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sat, 13 Jun 2020 08:15:24 GMT

View Forum Message <> Reply to Message

Novo wrote on Fri, 12 June 2020 18:45Could you please fix a compilation error on Mac? It was introduced a couple of days ago.

```
In file included from /Users/ssg/.local/soft/bb-worker/worker/m-upp/build/uppsrc/Core/App.cpp:4:
In file included from /usr/include/mach-o/dyld.h:31:
/usr/include/mach-o/loader.h:56:2: error: unknown type name 'cpu_type_t'; did you mean
'Upp::cpu_type_t'?
        cpu_type_t      cputype;        /* cpu specifier */
        ^
/usr/include/mach/machine.h:70:19: note: 'Upp::cpu_type_t' declared here
typedef integer_t       cpu_type_t;
                  ^
```

TIA

Hopefully fixed, please check.

Subject: Re: BufferPainter::Clear() optimization
Posted by coolman on Sat, 13 Jun 2020 08:33:23 GMT
View Forum Message <> Reply to Message

Hi,

The commit Core: Fixed compilation issue in MacOS created compilation error on Linux

```
lib/libDraw-lib.a(MakeCache.cpp.o): In function `Upp::SysImageRealized(Upp::Image const&)':
MakeCache.cpp:(.text._ZN3Upp16SysImageRealizedERKNS_5ImageE+0xd): undefined
reference to `Upp::IsValueCacheActive()'
MakeCache.cpp:(.text._ZN3Upp16SysImageRealizedERKNS_5ImageE+0x46): undefined
reference to `Upp::ValueCacheMutex'
MakeCache.cpp:(.text._ZN3Upp16SysImageRealizedERKNS_5ImageE+0x5b): undefined
reference to `Upp::TheValueCache()'
lib/libDraw-lib.a(MakeCache.cpp.o): In function `Upp::SysImageReleased(Upp::Image const&)':
MakeCache.cpp:(.text._ZN3Upp16SysImageReleasedERKNS_5ImageE+0xf): undefined
reference to `Upp::IsValueCacheActive()'
MakeCache.cpp:(.text._ZN3Upp16SysImageReleasedERKNS_5ImageE+0x3f): undefined
reference to `Upp::ValueCacheMutex'
MakeCache.cpp:(.text._ZN3Upp16SysImageReleasedERKNS_5ImageE+0x55): undefined
reference to `Upp::TheValueCache()'
lib/libDraw-lib.a(MakeCache.cpp.o): In function `Upp::SetMakeImageCacheMax(int)':
MakeCache.cpp:(.text._ZN3Upp20SetMakeImageCacheMaxEi+0xb): undefined reference to
`Upp::SetupValueCache(int, int, double)'
lib/libDraw-lib.a(MakeCache.cpp.o): In function `Upp::SetMakeImageCacheSize(int)':
MakeCache.cpp:(.text._ZN3Upp21SetMakeImageCacheSizeEi+0xb): undefined reference to
`Upp::SetupValueCache(int, int, double)'
lib/libDraw-lib.a(MakeCache.cpp.o): In function `Upp::SweepMkImageCache()':
MakeCache.cpp:(.text._ZN3Upp17SweepMkImageCacheEv+0x1): undefined reference to
`Upp::AdjustValueCache()'
lib/libDraw-lib.a(MakeCache.cpp.o): In function `Upp::MakeImage__(Upp::ImageMaker const&,
bool)':
MakeCache.cpp:(.text._ZN3Upp11MakeImage__ERKNS_10ImageMakerEb+0x25): undefined
reference to `Upp::MakeValue(Upp::LRUCache<Upp::Value, Upp::String>::Maker&)'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
CMakeFiles/ide-bin.dir/build.make:326: recipe for target 'bin/ide' failed
```

BR, Radek

Subject: Re: BufferPainter::Clear() optimization
Posted by Novo on Sat, 13 Jun 2020 11:07:52 GMT
View Forum Message <> Reply to Message

mirek wrote on Sat, 13 June 2020 04:15

Hopefully fixed, please check.
All three platforms are broken at this time because of linking.

Subject: Re: BufferPainter::Clear() optimization
Posted by coolman on Sat, 13 Jun 2020 12:45:46 GMT
View Forum Message <> Reply to Message

Hi,

The commit Core: Fixed to compile fixed compilation for Linux

Radek

Subject: Re: BufferPainter::Clear() optimization
Posted by Didier on Sun, 14 Jun 2020 10:45:50 GMT
View Forum Message <> Reply to Message

Hello all,

While searching for info on vectorisation techniques I stumbled on this
https://godbolt.org/

this web site proposes to compile small pieces of code (on many compilers) and examine the assembler output: it is dedicated to getting the best performance out the code

This may help to get the best vectorisation code quicker and for many compilers

---

Subject: Re: BufferPainter::Clear() optimization
Posted by mirek on Sun, 14 Jun 2020 12:09:07 GMT
View Forum Message <> Reply to Message

RescaleFilter now SSE2 optimised too...