Vectorizing your code using Universal Intrinsics

Prev Tutorial: How to use the OpenCV parallel_for_ to parallelize your code

Compatibility	OpenCV >= 3.0

Goal

The goal of this tutorial is to provide a guide to using the **Universal intrinsics** feature to vectorize your C++ code for a faster runtime. We'll briefly look into *SIMD intrinsics* and how to work with wide *registers*, followed by a tutorial on the basic operations using wide registers.

Theory

In this section, we will briefly look into a few concepts to better help understand the functionality.

Intrinsics

Intrinsics are functions which are separately handled by the compiler. These functions are often optimized to perform in the most efficient ways possible and hence run faster than normal implementations. However, since these functions depend on the compiler, it makes it difficult to write portable applications.

SIMD

SIMD stands for **Single Instruction, Multiple Data**. SIMD Intrinsics allow the processor to vectorize calculations. The data is stored in what are known as *registers*. A *register* may be *128-bits*, *256-bits* or *512-bits* wide. Each *register* stores **multiple values** of the **same data type**. The size of the register and the size of each value determines the number of values stored in total.

Depending on what *Instruction Sets* your CPU supports, you may be able to use the different registers. To learn more, look here

Universal Intrinsics

OpenCVs universal intrinsics provides an abstraction to SIMD vectorization methods and allows the user to use intrinsics without the need to write system specific code.

OpenCV Universal Intrinsics support the following instruction sets:

- *128 bit* registers of various types support is implemented for a wide range of architectures including
  - x86(SSE/SSE2/SSE4.2),
  - ARM(NEON),
  - PowerPC(VSX),
  - MIPS(MSA).
- *256 bit* registers are supported on x86(AVX2) and
- *512 bit* registers are supported on x86(AVX512)

We will now introduce the available structures and functions:

- Register structures
- Load and store
- Mathematical Operations
- Reduce and Mask

Register Structures

The Universal Intrinsics set implements every register as a structure based on the particular SIMD register. All types contain the `nlanes` enumeration which gives the exact number of values that the type can hold. This eliminates the need to hardcode the number of values during implementations.

Note

Each register structure is under the `cv` namespace.

There are **two types** of registers:

- **Variable sized registers**: These structures do not have a fixed size and their exact bit length is deduced during compilation, based on the available SIMD capabilities. Consequently, the value of the `nlanes` enum is determined in compile time.

  Each structure follows the following convention:

  ```
  v_[type of value][size of each value in bits]
  ```

  For instance, **v_uint8 holds 8-bit unsigned integers** and **v_float32 holds 32-bit floating point values**. We then declare a register like we would declare any object in C++

Based on the available SIMD instruction set, a particular register will hold different number of values. For example: If your computer supports a maximum of 256bit registers,

- *v_uint8* will hold 32 8-bit unsigned integers
- *v_float64* will hold 4 64-bit floats (doubles)

```
v_uint8 a;                              // a is a register supporting uint8(char) data
int n = a.nlanes;                       // n holds 32
```

Available data type and sizes:

Type	Size in bits
uint	8, 16, 32, 64
int	8, 16, 32, 64
float	32, 64

- **Constant sized registers**: These structures have a fixed bit size and hold a constant number of values. We need to know what SIMD instruction set is supported by the system and select compatible registers. Use these only if exact bit length is necessary.

  Each structure follows the convention:

```
v_[type of value][size of each value in bits]x[number of values]
```

  Suppose we want to store

  - 32-bit(*size in bits*) signed integers in a **128 bit register**. Since the register size is already known, we can find out the *number of data points in register* (*128/32 = 4*):

```
v_int32x8 reg1                         // holds 8 32-bit signed integers.
```

  - 64-bit floats in 512 bit register:

```
v_float64x8 reg2                       // reg2.nlanes = 8
```

Load and Store operations

Now that we know how registers work, let us look at the functions used for filling these registers with values.

- **Load**: Load functions allow you to *load* values into a register.
  - *Constructors* - When declaring a register structure, we can either provide a memory address from where the register will pick up contiguous values, or provide the values explicitly as multiple arguments (Explicit multiple arguments is available only for Constant Sized Registers):

```
float ptr[32] = {1, 2, 3 ..., 32};   // ptr is a pointer to a contiguous memory block of 32 floats

// Variable Sized Registers //
int x = v_float32().nlanes;          // set x as the number of values the register can hold

v_float32 reg1(ptr);                 // reg1 stores first x values according to the maximum register size ava
v_float32 reg2(ptr + x);             // reg stores the next x values

// Constant Sized Registers //
v_float32x4 reg1(ptr);               // reg1 stores the first 4 floats (1, 2, 3, 4)
v_float32x4 reg2(ptr + 4);           // reg2 stores the next 4 floats (5, 6, 7, 8)

// Or we can explicitly write down the values.
v_float32x4(1, 2, 3, 4);
```

  - *Load Function* - We can use the load method and provide the memory address of the data:

```
float ptr[32] = {1, 2, 3, ..., 32};
v_float32 reg_var;
reg_var = vx_load(ptr);              // loads values from ptr[0] upto ptr[reg_var.nlanes - 1]

v_float32x4 reg_128;
reg_128 = v_load(ptr);               // loads values from ptr[0] upto ptr[3]

v_float32x8 reg_256;
reg_256 = v256_load(ptr);            // loads values from ptr[0] upto ptr[7]

v_float32x16 reg_512;
reg_512 = v512_load(ptr);            // loads values from ptr[0] upto ptr[15]
```

Note

The load function assumes data is unaligned. If your data is aligned, you may use the `vx_load_aligned()` function.

- **Store**: Store functions allow you to *store* the values from a register into a particular memory location.
    - To store values from a register into a memory location, you may use the ***v_store()*** function:

```
float ptr[4];
v_store(ptr, reg); // store the first 128 bits(interpreted as 4x32-bit floats) of reg into ptr.
```

Note

Ensure **ptr** has the same type as register. You can also cast the register into the proper type before carrying out operations. Simply typecasting the pointer to a particular type will lead wrong interpretation of data.

Binary and Unary Operators

The universal intrinsics set provides element wise binary and unary operations.

- **Arithmetics**: We can add, subtract, multiply and divide two registers element-wise. The registers must be of the same width and hold the same type. To multiply two registers, for example:

```
v_float32 a, b;                          // {a1, ..., an}, {b1, ..., bn}
v_float32 c;
c = a + b                                // {a1 + b1, ..., an + bn}
c = a * b;                               // {a1 * b1, ..., an * bn}
```

- **Bitwise Logic and Shifts**: We can left shift or right shift the bits of each element of the register. We can also apply bitwise &, |, ^ and ~ operators between two registers element-wise:

```
v_int32 as;                              // {a1, ..., an}
v_int32 al = as << 2;                    // {a1 << 2, ..., an << 2}
v_int32 bl = as >> 2;                    // {a1 >> 2, ..., an >> 2}

v_int32 a, b;
v_int32 a_and_b = a & b;                 // {a1 & b1, ..., an & bn}
```

- **Comparison Operators**: We can compare values between two registers using the <, >, <= , >=, == and != operators. Since each register contains multiple values, we don't get a single bool for these operations. Instead, for true values, all bits are converted to one (0xff for 8 bits, 0xffff for 16 bits, etc), while false values return bits converted to zero.

```
// let us consider the following code is run in a 128-bit register
v_uint8 a;                               // a = {0, 1, 2, ..., 15}
v_uint8 b;                               // b = {15, 14, 13, ..., 0}

v_uint8 c = a < b;

/*
    let us look at the first 4 values in binary

    a = |00000000|00000001|00000010|00000011|
    b = |00001111|00001110|00001101|00001100|
    c = |11111111|11111111|11111111|11111111|

    If we store the values of c and print them as integers, we will get 255 for true values and 0 for false value
*/
---
// In a computer supporting 256-bit registers
v_int32 a;                               // a = {1, 2, 3, 4, 5, 6, 7, 8}
v_int32 b;                               // b = {8, 7, 6, 5, 4, 3, 2, 1}

v_int32 c = (a < b);                     // c = {-1, -1, -1, -1, 0, 0, 0, 0}

/*
    The true values are 0xffffffff, which in signed 32-bit integer representation is equal to -1.
*/
```

- **Min/Max operations**: We can use the ***v_min()*** and ***v_max()*** functions to return registers containing element-wise min, or max, of the two registers:

```
v_int32 a;                               // {a1, ..., an}
v_int32 b;                               // {b1, ..., bn}

v_int32 mn = v_min(a, b);                // {min(a1, b1), ..., min(an, bn)}
v_int32 mx = v_max(a, b);                // {max(a1, b1), ..., max(an, bn)}
```

Note

Comparison and Min/Max operators are not available for 64 bit integers. Bitwise shift and logic operators are available only for integer values. Bitwise shift is available only for 16, 32 and 64 bit registers.

Reduce and Mask

- **Reduce Operations**: The ***v_reduce_min()***, ***v_reduce_max()*** and ***v_reduce_sum()*** return a single value denoting the min, max or sum of the entire register:

```
v_int32 a;                              //  a = {a1, ..., a4}
int mn = v_reduce_min(a);               // mn = min(a1, ..., an)
int sum = v_reduce_sum(a);              // sum = a1 + ... + an
```

- **Mask Operations**: Mask operations allow us to replicate conditionals in wide registers. These include:
  - ***v_check_all()*** - Returns a bool, which is true if all the values in the register are less than zero.
  - ***v_check_any()*** - Returns a bool, which is true if any value in the register is less than zero.
  - ***v_select()*** - Returns a register, which blends two registers, based on a mask.

```
v_uint8 a;                           // {a1, .., an}
v_uint8 b;                           // {b1, ..., bn}

v_int32x4 mask:                      // {0xff, 0, 0, 0xff, ..., 0xff, 0}

v_uint8 Res = v_select(mask, a, b)   // {a1, b2, b3, a4, ..., an-1, bn}

/*
    "Res" will contain the value from "a" if mask is true (all bits set to 1),
    and value from "b" if mask is false (all bits set to 0)

    We can use comparison operators to generate mask and v_select to obtain results based on conditionals.
    It is common to set all values of b to 0. Thus, v_select will give values of "a" or 0 based on the mask.
*/
```

Demonstration

In the following section, we will vectorize a simple convolution function for single channel and compare the results to a scalar implementation.

Note

Not all algorithms are improved by manual vectorization. In fact, in certain cases, the compiler may *autovectorize* the code, thus producing faster results for scalar implementations.

You may learn more about convolution from the previous tutorial. We use the same naive implementation from the previous tutorial and compare it to the vectorized version.

The full tutorial code is here.

Vectorizing Convolution

We will first implement a 1-D convolution and then vectorize it. The 2-D vectorized convolution will perform 1-D convolution across the rows to produce the correct results.

1-D Convolution: Scalar

```
void conv1d(Mat src, Mat &dst, Mat kernel)
{

    int len = src.cols;
    dst = Mat(1, len, CV_8UC1);

    int sz = kernel.cols / 2;
    copyMakeBorder(src, src, 0, 0, sz, sz, BORDER_REPLICATE);

    for (int i = 0; i < len; i++)
    {
        double value = 0;
        for (int k = -sz; k <= sz; k++)
            value += src.ptr<uchar>(0)[i + k + sz] * kernel.ptr<float>(0)[k + sz];

        dst.ptr<uchar>(0)[i] = saturate_cast<uchar>(value);
    }
}
```

1. We first set up variables and make a border on both sides of the src matrix, to take care of edge cases.

```
    int len = src.cols;
    dst = Mat(1, len, CV_8UC1);

    int sz = kernel.cols / 2;
    copyMakeBorder(src, src, 0, 0, sz, sz, BORDER_REPLICATE);
```

2. For the main loop, we select an index *i* and offset it on both sides along with the kernel, using the k variable. We store the value in *value* and add it to the *dst* matrix.

```
    for (int i = 0; i < len; i++)
    {
        double value = 0;
        for (int k = -sz; k <= sz; k++)
            value += src.ptr<uchar>(0)[i + k + sz] * kernel.ptr<float>(0)[k + sz];

        dst.ptr<uchar>(0)[i] = saturate_cast<uchar>(value);
    }
```

1-D Convolution: Vector

We will now look at the vectorized version of 1-D convolution.

```
void conv1dsimd(Mat src, Mat kernel, float *ans, int row = 0, int rowk = 0, int len = -1)
{
    if (len == -1)
        len = src.cols;

    Mat src_32, kernel_32;

    const int alpha = 1;
    src.convertTo(src_32, CV_32FC1, alpha);

    int ksize = kernel.cols, sz = kernel.cols / 2;
    copyMakeBorder(src_32, src_32, 0, 0, sz, sz, BORDER_REPLICATE);


    int step = v_float32().nlanes;
    float *sptr = src_32.ptr<float>(row), *kptr = kernel.ptr<float>(rowk);
    for (int k = 0; k < ksize; k++)
    {
        v_float32 kernel_wide = vx_setall_f32(kptr[k]);
        int i;
        for (i = 0; i + step < len; i += step)
        {
            v_float32 window = vx_load(sptr + i + k);
            v_float32 sum = vx_load(ans + i) + kernel_wide * window;
            v_store(ans + i, sum);
        }

        for (; i < len; i++)
        {
            *(ans + i) += sptr[i + k]*kptr[k];
        }
    }
}
```

1. In our case, the kernel is a float. Since the kernel's datatype is the largest, we convert src to float32, forming *src_32*. We also make a border like we did for the naive case.

```
    Mat src_32, kernel_32;

    const int alpha = 1;
    src.convertTo(src_32, CV_32FC1, alpha);

    int ksize = kernel.cols, sz = kernel.cols / 2;
    copyMakeBorder(src_32, src_32, 0, 0, sz, sz, BORDER_REPLICATE);
```

2. Now, for each column in the *kernel*, we calculate the scalar product of the value with all *window* vectors of length `step`. We add these values to the already stored values in ans

```
    int step = v_float32().nlanes;
    float *sptr = src_32.ptr<float>(row), *kptr = kernel.ptr<float>(rowk);
    for (int k = 0; k < ksize; k++)
    {
        v_float32 kernel_wide = vx_setall_f32(kptr[k]);
        int i;
        for (i = 0; i + step < len; i += step)
        {
            v_float32 window = vx_load(sptr + i + k);
            v_float32 sum = vx_load(ans + i) + kernel_wide * window;
            v_store(ans + i, sum);
        }

        for (; i < len; i++)
        {
            *(ans + i) += sptr[i + k]*kptr[k];
        }
    }
```

- We declare a pointer to the src_32 and kernel and run a loop for each kernel element

```
    int step = v_float32().nlanes;
    float *sptr = src_32.ptr<float>(row), *kptr = kernel.ptr<float>(rowk);
    for (int k = 0; k < ksize; k++)
    {
```

- We load a register with the current kernel element. A window is shifted from *0* to *len - step* and its product with the kernel_wide array is added to the values stored in *ans*. We store the values back into *ans*

```
        v_float32 kernel_wide = vx_setall_f32(kptr[k]);
        int i;
        for (i = 0; i + step < len; i += step)
        {
            v_float32 window = vx_load(sptr + i + k);
            v_float32 sum = vx_load(ans + i) + kernel_wide * window;
            v_store(ans + i, sum);
        }
```

- Since the length might not be divisible by steps, we take care of the remaining values directly. The number of *tail* values will always be less than *step* and will not affect the performance significantly. We store all the values to *ans* which is a float pointer. We can also directly store them in a `Mat` object

```
        for (; i < len; i++)
        {
            *(ans + i) += sptr[i + k]*kptr[k];
        }
```

- Here is an iterative example:

```
For example:
kernel: {k1, k2, k3}
src:           ...|a1|a2|a3|a4|...


iter1:
for each idx i in (0, len), 'step' idx at a time
    kernel_wide:          |k1|k1|k1|k1|
    window:               |a0|a1|a2|a3|
    ans:               ...| 0| 0| 0| 0|...
    sum =  ans + window * kernel_wide
        =  |a0 * k1|a1 * k1|a2 * k1|a3 * k1|

iter2:
    kernel_wide:          |k2|k2|k2|k2|
    window:               |a1|a2|a3|a4|
    ans:               ...|a0 * k1|a1 * k1|a2 * k1|a3 * k1|...
    sum =  ans + window * kernel_wide
        =  |a0 * k1 + a1 * k2|a1 * k1 + a2 * k2|a2 * k1 + a3 * k2|a3 * k1 + a4 * k2|

iter3:
    kernel_wide:          |k3|k3|k3|k3|
    window:               |a2|a3|a4|a5|
    ans:               ...|a0 * k1 + a1 * k2|a1 * k1 + a2 * k2|a2 * k1 + a3 * k2|a3 * k1 + a4 * k2|...
    sum =  sum + window * kernel_wide
        =  |a0*k1 + a1*k2 + a2*k3|a1*k1 + a2*k2 + a3*k3|a2*k1 + a3*k2 + a4*k3|a3*k1 + a4*k2 + a5*k3|
```

Note

The function parameters also include *row*, *rowk* and *len*. These values are used when using the function as an intermediate step of 2-D convolution

2-D Convolution

Suppose our kernel has *ksize* rows. To compute the values for a particular row, we compute the 1-D convolution of the previous *ksize/2* and the next *ksize/2* rows, with the corresponding kernel row. The final values is simply the sum of the individual 1-D convolutions

```
void convolute_simd(Mat src, Mat &dst, Mat kernel)
{
    int rows = src.rows, cols = src.cols;
    int ksize = kernel.rows, sz = ksize / 2;
    dst = Mat(rows, cols, CV_32FC1);

    copyMakeBorder(src, src, sz, sz, 0, 0, BORDER_REPLICATE);

    int step = v_float32().nlanes;

    for (int i = 0; i < rows; i++)
    {
        for (int k = 0; k < ksize; k++)
        {
            float ans[N] = {0};
            conv1dsimd(src, kernel, ans, i + k, k, cols);
            int j;
            for (j = 0; j + step < cols; j += step)
            {
                v_float32 sum = vx_load(&dst.ptr<float>(i)[j]) + vx_load(&ans[j]);
                v_store(&dst.ptr<float>(i)[j], sum);
            }

            for (; j < cols; j++)
                dst.ptr<float>(i)[j] += ans[j];
        }
    }
    const int alpha = 1;
    dst.convertTo(dst, CV_8UC1, alpha);
}
```

1. We first initialize variables and make a border above and below the *src* matrix. The left and right sides are handled by the 1-D convolution function.

```
    int rows = src.rows, cols = src.cols;
    int ksize = kernel.rows, sz = ksize / 2;
    dst = Mat(rows, cols, CV_32FC1);

    copyMakeBorder(src, src, sz, sz, 0, 0, BORDER_REPLICATE);

    int step = v_float32().nlanes;
```

2. For each row, we calculate the 1-D convolution of the rows above and below it. we then add the values to the *dst* matrix.

```
    for (int i = 0; i < rows; i++)
    {
        for (int k = 0; k < ksize; k++)
        {
            float ans[N] = {0};
            conv1dsimd(src, kernel, ans, i + k, k, cols);
            int j;
            for (j = 0; j + step < cols; j += step)
            {
                v_float32 sum = vx_load(&dst.ptr<float>(i)[j]) + vx_load(&ans[j]);
                v_store(&dst.ptr<float>(i)[j], sum);
            }

            for (; j < cols; j++)
                dst.ptr<float>(i)[j] += ans[j];
        }
    }
```

3. We finally convert the *dst* matrix to a *8-bit* `unsigned char` matrix

```
    const int alpha = 1;
    dst.convertTo(dst, CV_8UC1, alpha);
```

Results

In the tutorial, we used a horizontal gradient kernel. We obtain the same output image for both methods.

Improvement in runtime varies and will depend on the SIMD capabilities available in your CPU.

Generated on Wed Aug 10 2022 03:53:17 for OpenCV by doxygen 1.8.13