An Alternative to Intrinsics for Vector Processing

Ross Maloney

June 17, 2024

Abstract

Using inline assembly code in a C program for performing vector processing has been shown to significantly reduce execution time relative to using Intrinsics functions for such processing. Here by using `zmm` registers in place of `ymm` registers, it is shown this exchange in a large number of AVX and AVX2 instructions can extend them to AVX-512 vectors in inline assembly. Exceptions are also covered.

As shown in Jeong et al. (2012) there are three approaches in C program coding to include Intel SIMD registers for vector processing. Of these, *Intrinsics* are most widely used and taught. Intel in Intel (2024*b*) shows the breadth of such functions. But the purpose of vector processing is to reduce the execution time of the code. But as Jeong et al. (2012) show, the use of inline assembly can significantly reduce such execution time relative to using Intrinsic functions.

It would appear logical to process as many data elements in a vector at the one hop to decrease the overall execution time. Since Jeong et al. (2012), AVX-512 has been released by Intel which doubled the vector size to 512 bits over the 256 bits in the AVX (and AVX2) releases. With the AVX-512 release Intel (2024*a*) show a large number of vector processing instructions are available. Where as the AVX-512 instructions can have a complex structure using auxiliary registers, experience reported here indicates the less complex AVX instructions extend to AVX-512 vector size. By using `zmm` registers in place of `ymm` registers in the AVX instruction format, a legal instruction which processes twice as many vector members can result. This substitution approach may not work for every AVX instruction but it does for a variety of useful instructions.

1 Inline assembly

Substitution of registers in assembly instructions requires direct assess to those instruction. Inline assembly provides such assess. Such assembly instructions are processed directly by both `gcc` and `clang` C compilers with no change to the standard command line used except including the `-m64` flag to generate 64-bit executable code.

The *extended* inline assembly was used. This allows easy movement of data between the assembly code portion and the C code in which it is embedded. An outline of the extended inline assembly format is:

```
asm ("       "    code lines
     :            output operands
     :            input operands
     :        )   clobber list
```

A more general form using `__asm__` in place of `asm` with asm qualifier word was not used in this work. This form worked with both `gcc` and `clang` compilers.

The assembler instructions were encoded in a C-link string. This string can contain one or more assembler instructions, with each instruction on a single line. A line is terminated by a C newline character (`\n`). The alignment of the next line in a standard assembly format can be achieved by following the newline character with a tab character (\t).

The next three lines are optional. The output line is denoted by a string containing `=r` and then the name of the C variable in parentheses which will receive the output from the assembly code. The input line is denoted by a string containing `r` followed by the C variable in parentheses for which the assembly code obtains input data. The third line lists registers used in the assembly code which could also be used in the embedding C code. In this work that possibility was ignored and thus the clobber line was not used.

The assembly code employs a number of conventions. Registers are prefixed by double percentage signs (%%). Data movement in each instruction follows the AT&T convention of moving left to right. Connection between the assembly code and the C variables in the output and input lines is via a number preceded by a single percentage sign (%). Those numbers start at zero (0) on the output line and proceed to the input line. For example, if there were one element on the output line it would be represented in the assembly code as %0. If in the same arrangement there were two input elements on the input line, the first would be referenced as %1 and the second as %2. By this convention, variables in the embedding C code are indirectly referenced.

2 SIMD instruction which worked in inline assembly

The instruction set described in Intel (2024*a*) is reasonably complex. With respect to the SIMD instructions this complexity has increased in the AVX-512 release. It appeared instructions in the previous AVX and AVX2 releases using the `ymm` registers had been replaced by more complex instructions to use the longer `zmm` registers. But longer the register the better the efficiency of the instruction.

The program of Figure 1 was used to test whether the `ymm` instructions in Intel (2024*a*) would perform the same using `zmm` registers. Instructions which were thought to have value in relation to undefined future work were testing. All testing was performed on a MacPro 2019 computer with a 28 core Intel Xeon W processor chip governed by a standard Debian Linux operating system. Compiling was performed using both `gcc` and `clang`.

Figure 1: Example template program which was altered for testing inline instructions

```
1  #include <stdio.h>
2
3  int main()
4  {
5    long a1[] = {1, 2, 3, 4, 35, 6, 7, 8, -7, 45, 23, -12, 8,
6                 78, 34, -7};
7    long b1[] = {-25, 16, 27, 8, 9, 30, -15, -32, -34, 8, 90,
8                 100, -57, 23, 12, 87};
9    long b3[] = {4, 1, 5, 2, 6, 3, 4, 4, 0, 0, 0, 9, 0, 0,
10                0, 8};
11   long b4[] = {1, 2, 2, 1, 3, 1, 1, 1, 3, 1, 1, 1, 1, 1,
12                1, 2};
13   long c[16];
14   float g1[] = {1.3, -5.7, 12.3, -6.1, 7.8, 10.0, 13.4, 100.3,
15                 23.1, -23.1, 45.2, 4.2, 9.7, 7.0, 12.0, 5.6};
16   float h1[] = {5.9, -2.4, 7.6, -12.0, 7.9, 13.7, 123.0, 67.8,
17                 5.7, 34.2, -43.8, 0.2, 76.0, -43.7, 6.9, 6.5};
18   float f[16];
19   int   i;
20
21   asm volatile("vmovupd %1, %%zmm0 \n\t"
22                "vmovupd %2, %%zmm1 \n\t"
23          "vpsubb  %%zmm2, %%zmm1, %%zmm0 \n\t"
24          "vmovupd %%zmm2, %0"
25          : "=m" (c[0])
26          : "m" (a1[0]), "m" (b1[0])
27   );
28   for (i = 0; i < 16; i++)  printf("%ld  ", c[i]);
29   printf("\n");
30
31   return 0;
32 }
```

The range of tests required the program of Figure 1 to be adapted. The arrays of data sent to and received from the assembly code was matched to the requirements of the instruction under test. This was done by changing the array in included in lines 23 and 24. This enabled the register load instructions at lines 19 and 20, and the

register read instruction at line 22 to remained unchanged for each test. In the case of and instruction requiring two registers, line 20 was deleted and register `%%zmm1` was removed from line 21 which contained the instruction under test.

Tables 1 and 3 contain the result of the tests. The tests are grouped into the size of data which were contained in the vectors. The `"m" 1` and `"m" 2` columns correspond to the first and second entries on line 24 of Figure 1, respectively. The columns headed `"=m"` correspond to line 23. The `vec` column contains the C data array used in the test, and the `z` is the `zmm` register number used.

Table 1: AVX/AVX2 instructions using `zmm` registers which move data about

Data	Op code	″m″ 1			″m″ 2			″=m″			Description
bits		vec	type	z	vec	type	z	vec	type	z	
32	vmovups	b3	int	2				c	int	1	int copy
32	vmovups	g1	float	2				f	float	1	fp copy
32	vpermd	a1	int	2	b3	int	1	c	int	0	int permute
32	vpermps	a1	int	2	b3	int	1	c	int	0	int permute
32	vpermd	g1	float	2	b3	int	1	f	float	0	fp permute
32	vpermps	g1	float	2	b3	int	1	f	float	0	fp permute
64	vmovups	b3	long	2				c	long	1	long copy
64	vmovups	g1	double	2				f	double	1	dp copy
64	vpermpd	a1	long	2	b3	long	1	c	long	0	long permute
64	vpermq	a1	long	2	b3	long	1	c	long	0	long permute
64	vpermpd	g1	double	2	b3	long	1	f	double	0	dp permute
64	vpermq	g1	double	2	b3	long	1	f	double	0	dp permute

The tables of results are divided into two by the type of operations they contain. Table 3 contains instructions which make changes to the data element of a vector, for example by arithmetic operation. Table 1 by contrast does not change the value of the data elements but move those data elements about.

In some instances in Tables 1 and 3 the `"=m" type"` and `"m" type` are shown as abbreviateion. The correspondence of such entries with the data type of the array in memory as compiled is given in Table 2.

Table 2: Correspondence between SIMD tabulated types and C data types

Table type	Compiler C type
char	signed char
short	short
int	int
long	long
float	float
double	double

Table 3: AVX/AVX2 instructions using `zmm` registers which process data

Data bits	Op code	"m" 1 vec	"m" 1 type	"m" 1 z	"m" 2 vec	"m" 2 type	"m" 2 z	"=m" vec	"=m" type	"=m" z	Description
8	vpaddb	a1	char	1	b1	char	2	c	char	0	add
8	vpsubb	a1	char	1	b1	char	2	c	char	0	subtract
8	vpmaxsb	a1	char	1	b1	char	2	c	char	0	maximum
8	vpminsb	a1	char	1	b1	char	2	c	char	0	minimum
16	vpaddw	a1	short	1	b1	short	2	c	short	0	add
16	vpsubw	a1	short	1	b1	short	2	c	short	0	subtract
16	vpmaxsw	a1	short	1	b1	short	2	c	short	0	maximum
16	vpminsw	a1	short	1	b1	short	2	c	short	0	minimum
32	vpaddd	a1	int	1	b1	int	2	c	int	0	add
32	vpsubd	a1	int	1	b1	int	2	c	int	0	subtract
32	vpmulld	a1	int	1	b1	int	2	c	int	0	multiply
32	vpmaxsd	a1	int	1	b1	int	2	c	int	0	maximum
32	vpminsd	a1	int	1	b1	int	2	c	int	0	minimum
32	vpsllvd	a1	int	1	b4	int	2	c	int	0	int left shift
32	vpsrlvd	b3	int	1	b4	int	2	c	int	0	int right shift
32	vcvtdq2ps	a1	int	2				f	float	1	I32 $\rightarrow$ FP32
64	vpaddq	a1	long	1	b1	long	2	c	long	0	add
64	vpsubq	a1	long	1	b1	long	2	c	long	0	subtract
64	vpmullq	a1	long	1	b1	long	2	c	long	0	multiply
64	vpmaxsd	a1	long	1	b1	long	2	c	long	0	maximum
64	vpminsd	a1	long	1	b1	long	2	c	long	0	minimum
64	vpsllvq	a1	long	1	b4	long	2	c	long	0	long left shift
64	vpsrlvq	b3	long	1	b4	long	2	c	long	0	long right shift
64	vcvtqq2pd	a1	long	2				f	double	1	I64 $\rightarrow$ DP64
32	vaddps	g1	float	1	h1	float	2	f	float	0	add
32	vsubps	g1	float	1	h1	float	2	f	float	0	subtract
32	vmulps	g1	float	1	h1	float	2	f	float	0	multiply
32	vdivps	g1	float	1	h1	float	2	f	float	0	divide
32	vfmadd231ps	g1	float	1	h1	float	2	f	float	0	FMA add
32	vfmsub231ps	g1	float	1	h1	float	2	f	float	0	FMA subtract
32	vmaxps	g1	float	1	h1	float	2	f	float	0	maxiumm
32	vminps	g1	float	1	h1	float	2	f	float	0	minimum
32	vcvttps2dq	g1	float	2				c	int	1	FP32 $\rightarrow$ I32
64	vaddpd	g1	double	1	h1	double	2	f	double	0	add
64	vsubpd	g1	double	1	h1	double	2	f	double	0	subtract
64	vmulpd	g1	double	1	h1	double	2	f	double	0	multiply
64	vdivpd	g1	double	1	h1	double	2	f	double	0	divide
64	vfmadd231pd	g1	double	1	h1	double	2	f	double	0	FMA add
64	vfmsub231pd	g1	double	1	h1	double	2	f	double	0	FMA subtract
64	vmaxps	g1	double	1	h1	double	2	f	double	0	maxiumm
64	vminps	g1	double	1	h1	double	2	f	double	0	minimum
64	vcvttpd2qq	g1	double	2				c	long	1	DP64 $\rightarrow$ I64

With respect to Table 1 the following should be noted. A number of pairs of instructions appear to yield the same result. More significant, right shifting instructions should not be used if negative numbers are present in the vector being operated upon.

3 Cautionaries

As noted above, right shift with negative numbers in the vector requires a different approach. Right shifting divides the absolute value of each vector element by the specified power of 2. But the `vpsrlvd` and `vpsrlvq` instructions do not shilf negative sign bit into vacated element bit positions. Instead the `vpsraw` and `vpsrad` instructions are alternatives to such right shift instructions. However the instruction format is different to that used in Figure 1.

Extending the range of AVX and AVX2 instructions by using `zmm` registers in place of `ymm` registers does not always work. Some of those instances are noted in Table 4. In those instructions, `vdpps` adopts a different format to the others.

Table 4: AVX/AVX2 instructions not extended to `zmm` registers

Data	type	Op code	Description
16-bit	short	vphaddw	horizontal add pairs
32-bit	float	vhaddps	horizontal add pairs
32-bit	integer	vphaddd	horizontal add pairs
32-bit	float	vdpps	dot product
64-bit	double	vhaddpd	horizontal add pairs

What appeared to be missing from both the AVX and AVX-512 instruction sets of Intel (2024*a*) was a horizontal sum along a whole vector, distinct from horizontal sum of pairs. Through a combination of repeated horizontal add pairs and vector permutation this sum can be obtained. By contrast in the SSE instruction set there was the `vdppd` instruction for double prevision and `vdpps` for single precision floating point vectors. These gave the dot product of two vectors. If one vector was a unit vector, then the sum of the elements in the other vector resulted. Never a dot product or vector sum of integers.

Care must be taken in matching the data array type used and the inline assembly code. Arithmetic instructions are sensitive to such match. In the case of operating on integers, a mismatch can result in a negative data value causing the following result values increased by one. By contrast, the instruction `vmovups` was successfully used with all data types to move data between AVX registers and memory.

References

Intel (2024*a*), "Intel 64 and IA-32 Architectures Software Developer's Manual", Intel Corporation (), cdrdv2-public.intel.com, accessed May 2, 2024.

Intel (2024*b*), "Intel Intrinsics Guide", www.intel.com/content/www/us/en/docs/intrinsics-guide/html, accessed May 25, 2024.

Jeong, H., et al. (2012), "Performance of SEE and AVX Instruction Sets" (), arXiv.org/1211.0820v1, accessed Apr. 27, 2024.