Optimization of Scan Operations Using Explicit Vectorization

Exploiting Intel® AVX-512 SIMD Instructions to Accelerate Prefix Sum Computations

Vamsi Sripathi and Ruchira Sasanka, Senior HPC Application Engineers, Intel Corporation

Introduction

Modern Intel® processors offer instruction-, data-, and thread-level parallelism. The ability to simultaneously execute a single instruction on multiple data (SIMD) operands maximizes utilization of processor arithmetic execution units. In this article, we will focus on SIMD optimizations applied to vector scan operations.

Scan (also known as inclusive/exclusive scan, prefix sum, or cumulative sum) is a common operation in many application domains[1]. As such, it is defined as a standard library function in C++, the OpenMP runtime, and

the Python* NumPy package[2,3]. A scan of a vector is another vector where the result at index i is obtained by summing all the values up to index *i* (for inclusive scan) and *i-1* (for exclusive scan) from the source vector. For instance, the inclusive scan of vector *x* with *n* elements is obtained by:

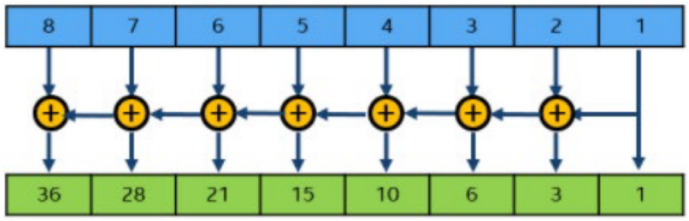
$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

And generally defined as:

$$y_j = \sum_{i=0}^{j} x_i$$



We will be demonstrating inclusive scan, henceforth referred to simply as scan. We'll begin with a brief overview of SIMD, also known as vectorization, followed by a baseline implementation of the scan operation in C and with OpenMP directives. Next, we'll implement an optimized scan algorithm in SIMD operating on 512-bit vector registers. We'll conclude with performance comparisons of the baseline and optimized implementations.

SIMD Overview

SIMD instructions operate on wider vector registers that can hold multiple data operands. The width and the number of vector registers are architecture-dependent, with the latest generation of Intel processors supporting 32, 512-bit-wide vector registers. Intel provides x86 ISA extensions in the form of Advanced Vector Extensions (AVX), AVX2, and AVX-512 that allow various arithmetic and logical operations to be performed on the 256-bit- and 512-bit-wide registers, respectively. Depending on the width of vector register, they are named XMM, YMM, or ZMM (**Table 1**).

Register Width	Register Name	Num. 64b operands/register	x86 ISA	Num. 64b ops per SIMD Add
128-bit	XMM	2	SSE	2
256-bit	YMM	4	AVX/AVX2	4
512-bit	ZMM	8	AVX512	8

Table 1. SIMD register naming

SIMD execution is mainly facilitated through two methods:

1. Implicit vectorization: This relies on the compiler to automatically transform the scalar computational loops into vectorized/SIMD blocks. The compiler does the heavy lifting of identifying which computations to vectorize, how aggressively to unroll loops, etc. based on a cost model. It also handles the data alignment requirements of SIMD instructions by producing the necessary loop prologue and epilogue sections in the generated object code. All the programmer has to do is use the appropriate compiler optimization flags for the target CPU architecture. Intel® compilers also provide pragmas[4] to give explicit hints to control the generation of SIMD instructions.
2. Explicit vectorization:
   a. Vector intrinsics: These are C APIs provided by the compiler that map to the underlying hardware SIMD instructions[5]. They allow the programming to control the type of SIMD instructions (AVX, AVX2, and AVX-512) that are generated in the code and how aggressively they are used. This can be viewed as a hybrid mode wherein the programmer still relies on the compiler for critical aspects of tuning, such as register renaming and instruction scheduling, but at the same time has the ability to control the code generation of non-trivial, data-dependent operations, such as scan.
   b. Assembly language: This completely bypasses compiler optimization in favor of assembly language coding, which requires expert knowledge of underlying microarchitecture. This approach is not recommended.

The tradeoffs of these approaches are summarized in **Table 2**.

Mode	Portability	Effort
Implicit Vectorization	High	Low
Vector Intrinsics	Medium	Medium
Assembly Instructions	Low	High

Table 2. Comparison of SIMD programming approaches

Baseline Code

The baseline code for computing the scan of a vector is shown below, followed by the Intel compiler optimization report (generated using the **-qopt-report=5** compiler option). The scan operation has inherent loop-carried dependencies, so the compiler is unable to fully vectorize the computation.

```
 8 void ref_scan (int *p_n, double *restrict src, double *restrict dst, double *p_init_val)
 9 {
10   int n = *p_n;
11   double tmp = *p_init_val;
12
13   for (int i=0; i<n; i++) {
14     tmp += src[i];
15     dst[i] = tmp;
16   }
17 }
```

```
LOOP BEGIN at kernels.c(13,2)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed ANTI dependence between tmp (14:3) and tmp (14:3)
   remark #15346: vector dependence: assumed FLOW dependence between tmp (14:3) and tmp (14:3)
   remark #25439: unrolled with remainder by 2
LOOP END
```

The same operation with the addition of OpenMP directives is shown below, along with the compiler optimization report. OpenMP is a portable, directive-based parallel programming model that includes SIMD support. The SIMD construct (combined with the reduction clause and scan directive) can be used to perform a scan operation on the vector. (We refer the reader to the OpenMP specification, which provides good documentation[6,7].) Because we are interested in understanding the impact of SIMD on workload performance, we are not parallelizing the loop, but instead limiting the OpenMP directives to vectorization only. The compiler optimization report shows that vectorization was performed using a vector length of eight (i.e., AVX-512) and that the loop was unrolled by a factor of 16 elements.

```
20 void omp_scan (int *p_n, double *restrict src, double *restrict dst, double *p_init_val)
21 {
22   int n = *p_n;
23   double tmp = *p_init_val;
24
25 #pragma omp simd reduction(inscan, +:tmp)
26     for(int i=0; i<n; i++){
27       tmp += src[i];
28 #pragma omp scan inclusive(tmp)
29       dst[i] = tmp;
30     }
31 }
```

```
LOOP BEGIN at kernels.c(26,5)
   remark #15389: vectorization support: reference src[i] has unaligned access   [ kernels.c(27,14) ]
   remark #15388: vectorization support: reference dst[i] has aligned access   [ kernels.c(29,7) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 2
   remark #15309: vectorization support: normalized vectorization overhead 0.342
   remark #15301: SIMD LOOP WAS VECTORIZED
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 9
   remark #15477: vector cost: 2.370
   remark #15478: estimated potential speedup: 3.490
   remark #15488: --- end vector cost summary ---
LOOP END
```

We will compare the baseline and auto-vectorized variants to the explicit AVX-512 SIMD implementation described in the next section.

Explicit Vectorization

In this section, we demonstrate the SIMD techniques to optimize the vector scan operations on Intel processors that support AVX-512 execution units. We will be using the double precision floating point (FP64) datatype for input and output vectors. The AVX-512 SIMD instructions used in our implementation are shown in **Table 3**.

The main idea behind our implementation is to simultaneously perform a series of add operations in the lower and upper 256-bit lanes of the 512-bit register after applying the necessary shuffle sequence. For every eight input elements, we perform a total of five adds and five permutes. All the add operations form a dependency chain because we are accumulating the results in a single register. However, some of the

permutes are independent of the add, so they can be executed simultaneously. **Figure 1** shows a visual representation of the implementation, followed by the code showing the main loop block where we are processing 16 elements per iteration.

Name	AVX-512 Vector Intrinsics API (x86 Instruction)	Description
SIMD Load	__m512d _mm512_loadu_pd (void *mem_address);	Load a set of eight FP64 elements from memory to AVX-512 vector register.
SIMD Add	__m512d _mm512_add_pd (__m512d src1, __m512d src2);	Add FP64 elements in *src1* and *src2* and return the results.
SIMD Permute with mask (128b lanes)	__m512d _mm512_maskz_permute_pd (__mmask8 k, __m512d src, const int imm8);	Shuffle FP64 elements in *src* within 128-bit lanes using the 8-bit control in *imm8*, and store the results using zeromask *k*. (Elements are zeroed out when corresponding mask bit is not set.)
SIMD Permute with mask (256b lanes)	__m512d mm512_maskz_permutex_pd (__mmask8 k, __m512d src, const int imm8);	Shuffle FP64 elements in *src* within 256-bit lanes using the 8-bit control in *imm8*, and store the results using zeromask *k*.(Elements are zeroed out when corresponding mask bit is not set.)
SIMD Store	void _mm512_storeu_pd (void *mem_address, _m512d src);	Store a set of eight FP64 elements to memory from AVX-512 vector register.

Table 3. AVX-512 SIMD instructions

		512-bits				256-bits		128-bits	
Operation	Result Register	511b		383b		255b		127b	0b
_mm512_loadu_pd(p_src)	zmm0	8	7	6	5	4	3	2	1
_mm512_maskz_permute_pd (0xAA, zmm0, 0x00);	zmm1	7	0	5	0	3	0	1	0
_mm512_add_pd (zmm0, zmm1);	zmm2	15	7	11	5	7	3	3	1
_mm512_maskz_permutex_pd (0xCC, zmm0, 0x40);	zmm3	6	5	0	0	2	1	0	0
_mm512_add_pd(zmm2, zmm3);	zmm2	21	12	11	5	9	4	3	1
_mm512_maskz_permute_pd (0xCC, zmm3, 0x44);	zmm3	5	6	0	0	1	2	0	0
_mm512_add_pd(zmm2, zmm3);	zmm2	26	18	11	5	10	6	3	1
_mm512_maskz_permutexvar_pd (0xF0, _mm512_set1_epi64(3), zmm2);	zmm3	10	10	10	10	0	0	0	0
_mm512_add_pd(zmm2, zmm3);	zmm2	36	28	21	15	10	6	3	1

Figure 1. Explicit AVX-512 SIMD scan operation sequence

```
#include "immintrin.h"
#define N_UNROLL          (16)
#define NUM_ELES_IN_ZMM   (8)

void awe_scan (int *p_n, double *restrict src, double *restrict dst, double *p_init_val)
{
  __m512d zmm0, zmm1, zmm2, zmm3, zmm4, zmm_acc, zmm_tmp1 ,zmm_tmp2;
  __m512d zmm5, zmm6, zmm7, zmm8, zmm9, zmm10, zmm11;
  __m512i idx, acc_idx;

  int n = *p_n;

  zmm_acc = _mm512_set1_pd(*p_init_val);
  acc_idx = _mm512_set1_epi64(7);
  idx     = _mm512_set1_epi64(3);


  int n_block = (n/N_UNROLL)*N_UNROLL;
  int n_tail  = n - n_block;

 for (int j=0; j<n_block; j+=N_UNROLL) {
   zmm0 = _mm512_loadu_pd(src);
   zmm5 = _mm512_loadu_pd(src+NUM_ELES_IN_ZMM);

   zmm2  = _mm512_maskz_permute_pd(0xAA, zmm0, 0x00);
   zmm7  = _mm512_maskz_permute_pd(0xAA, zmm5, 0x00);
   zmm10 = _mm512_add_pd(zmm0, zmm2);
   zmm11 = _mm512_add_pd(zmm5, zmm7);

   zmm1  = _mm512_maskz_permutex_pd(0xCC, zmm0, 0x40);
   zmm6  = _mm512_maskz_permutex_pd(0xCC, zmm5, 0x40);
   zmm10 = _mm512_add_pd(zmm10, zmm1);
   zmm11 = _mm512_add_pd(zmm11, zmm6);

   zmm1  = _mm512_maskz_permute_pd(0xCC, zmm1, 0x44);
   zmm6  = _mm512_maskz_permute_pd(0xCC, zmm6, 0x44);
   zmm10 = _mm512_add_pd(zmm10, zmm1);
   zmm11 = _mm512_add_pd(zmm11, zmm6);

   zmm_tmp1 = _mm512_maskz_permutexvar_pd(0xF0, idx, zmm10);
   zmm_tmp2 = _mm512_maskz_permutexvar_pd(0xF0, idx, zmm11);
   zmm10    = _mm512_add_pd(zmm10, zmm_tmp1);
   zmm11    = _mm512_add_pd(zmm11, zmm_tmp2);

   zmm_tmp1 = _mm512_add_pd(zmm10, zmm_acc);
   zmm_acc  = _mm512_add_pd(zmm11, zmm_tmp1);
   zmm_acc  = _mm512_permutexvar_pd(acc_idx, zmm_acc);

   zmm_tmp2 = _mm512_permutexvar_pd(acc_idx, zmm_tmp1);
   _mm512_storeu_pd(dst, zmm_tmp1);

   zmm11 = _mm512_add_pd(zmm11, zmm_tmp2);
   _mm512_storeu_pd(dst+NUM_ELES_IN_ZMM, zmm11);

   src += N_UNROLL;
   dst += N_UNROLL;
 }
```

Performance Evaluation

We evaluate the performance between the three versions of scan implementations on a single core of an Intel® Xeon® Platinum 8260L processor. The GCC (v9.3.0), Clang (v10.0.1), and ICC (Intel® C++ Compiler, v19.1.3.304) compilers were compared for the baseline, OpenMP SIMD, and explicit AVX-512 SIMD implementations. **Table 4** shows the assembly code generated by ICC for the main block of each implementation.

Baseline/SSE	OMP SIMD scan	Explicit AVX512 SIMD

..B1.4:
incq %r8
vaddsd (%rdi,%rsi),
%xmm0, %xmm0
vmovsd %xmm0,
(%rdi,%rdx)
vaddsd 8(%rdi,%rsi),
%xmm0, %xmm0
vmovsd %xmm0,
8(%rdi,%rdx)
addq $16, %rdi
cmpq %rcx, %r8
jb ..B1.4
	..B2.15:
	vmovups (%rsi,%r9,8), %zmm5
	vmovups 64(%rsi,%r9,8), %zmm13
	vbroadcastsd %xmm0, %zmm0
	vpermpd %zmm5, %zmm4, %zmm6{%k3}{z}
	vpermpd %zmm13, %zmm4,
	%zmm14{%k3}{z}
	vaddpd %zmm6, %zmm5, %zmm7
	vaddpd %zmm14, %zmm13, %zmm15
	vpermpd %zmm7, %zmm3, %zmm8{%k2}{z}
	vpermpd %zmm15, %zmm3,
	%zmm16{%k2}{z}
	vaddpd %zmm8, %zmm7, %zmm9
	vaddpd %zmm16, %zmm15, %zmm17
	vpermpd %zmm9, %zmm2, %zmm10{%k1}{z}
	vpermpd %zmm17, zmm2,%zmm18{%k1}{z}
	vaddpd %zmm10, %zmm9, %zmm11
	vaddpd %zmm18, %zmm17, %zmm21
	vaddpd %zmm11, %zmm0, %zmm12
	vpermpd %zmm12, %zmm1, %zmm19
	vmovupd %zmm12, (%rcx,%r9,8)
	vbroadcastsd %xmm19, %zmm20
	vaddpd %zmm21, %zmm20, %zmm22
	vpermpd %zmm22, %zmm1, %zmm0
	vmovupd %zmm22, 64(%rcx,%r9,8)
	addq $16, %r9
	cmpq %rdi, %r9
	jb ..B2.15
		..B2.3:
		vmovups (%rsi), %zmm5
		addl $16, %eax
		vmovups 64(%rsi), %zmm6
		vpermpd $64, %zmm5, %zmm9{%k2}{z}
		vpermpd $64, %zmm6, %zmm10{%k2}{z}
		vpermilpd $0, %zmm5, %zmm3{%k3}{z}
		addq $128, %rsi
		vaddpd %zmm5, %zmm3, %zmm7
		vaddpd %zmm7, %zmm9, %zmm12
		vpermilpd $0, %zmm6, %zmm4{%k3}{z}
		vpermilpd $68, %zmm9, %zmm11{%k2}{z}
		vaddpd %zmm6, %zmm4, %zmm8
		vaddpd %zmm12, %zmm11, %zmm18
		vaddpd %zmm8, %zmm10, %zmm14
		vpermpd %zmm18, %zmm1,
		%zmm17{%k1}{z}
		vpermilpd $68, %zmm10,%zmm13{%k2}{z}
		vaddpd %zmm14, %zmm13, %zmm16
		vaddpd %zmm18, %zmm17, %zmm19
		vpermpd %zmm16, %zmm1,
		%zmm15{%k1}{z}
		vaddpd %zmm2, %zmm19, %zmm20
		vaddpd %zmm16, %zmm15, %zmm22
		vpermpd %zmm20, %zmm0, %zmm21
		vmovups %zmm20, (%rdx)
		vaddpd %zmm22, %zmm21, %zmm23
		vaddpd %zmm20, %zmm22, %zmm2
		vmovups %zmm23, 64(%rdx)
		vpermpd %zmm2, %zmm0, %zmm2
		addq $128, %rdx
		cmpl %edi, %eax
		jl ..B2.3

Table 4. Assembly code generated by ICC for the three scan implementations

We observe that for the baseline code, the loop is unrolled by only two elements, and both scalar add operations form a dependency chain. This is inefficient because the generated code is not exploiting the wider AVX-512 execution units.

The OpenMP SIMD implementation is more interesting because the generated code uses AVX-512 add and permute operations and is unrolled by 16 elements. There are total of eight adds, eight permute, and two broadcast instructions to process 16 input elements. For a set of eight elements, the add and permutes form a dependency chain (highlighted in red). In addition, all the permutes generated in this version of the code shuffle elements within 256b lanes. Permutes that shuffle elements within a 128b lanes have lower latency (one cycle) than permutes that shuffle within 256b lanes (three cycles).

Our explicit AVX-512 SIMD implementation is also unrolled by 16 elements, and uses 11 add and 10 permute instructions to process those 16 elements. Even though we use greater number of add and permute instructions compared to the OpenMP SIMD scan, it has two advantages. First, four out of ten permutes are within a 128b lane (one-cycle latency), and hence are more efficient. These four permute instructions are highlighted in green. Second, some of the permutes are independent of the add instructions. This helps by providing more instruction-level parallelism and more effective utilization of AVX-512 execution units. We do an extra add to compute the accumulation result as soon as possible. This puts more concurrent instructions in flight and mitigates stalls in the execution pipeline for future iterations.

Figure 2 shows the performance for vector sizes ranging from 64 to 1,024 elements in steps of 32 and with all vectors residing in L1 cache. We can make the following observations from the performance data:

1. The explicit AVX-512 SIMD implementation outperforms both the baseline and OpenMP SIMD implementations.
2. GCC and Clang are unable to vectorize the scan computations. Their performance remains unchanged, even with the OpenMP SIMD directives.
3. ICC does a great job of auto-vectorization when OpenMP SIMD directives are used.
4. The average speed-up of the explicit SIMD scan implementation over the baseline and OpenMP SIMD scans is 4.6x (GCC and Clang) and 1.6x (ICC), respectively.

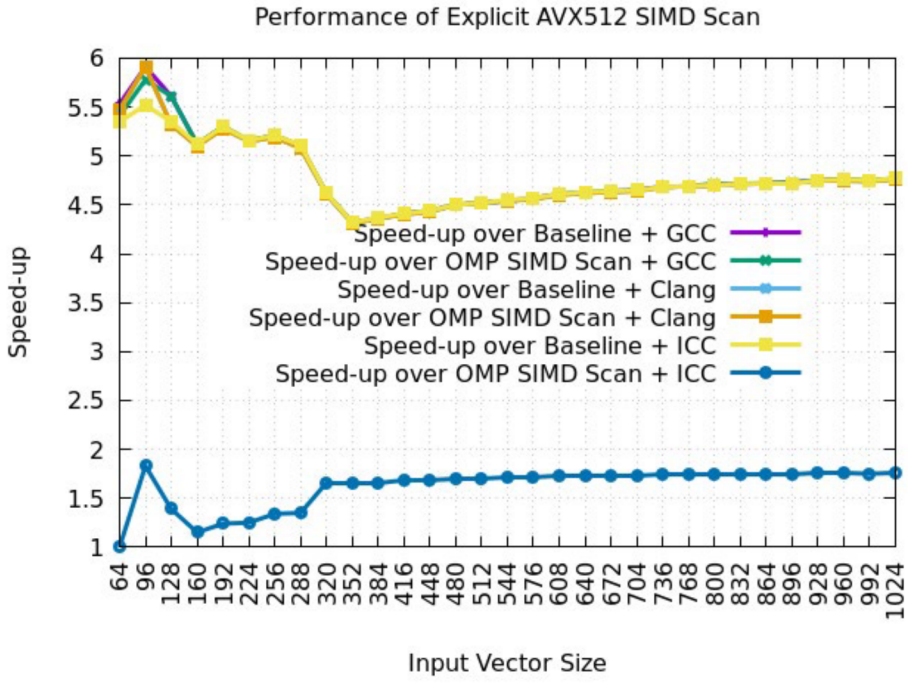


Figure 2. Performance comparison of explicit AVX-512 scan

Final Thoughts

We have briefly introduced explicit SIMD programming and applied it to vector scan computations. While optimizing compilers can often provide good performance, there may be cases, as demonstrated in this article, where there is room for improvement. Therefore, it is useful for developers to understand explicit SIMD programming to achieve maximum performance.

References

1. Guy E Blelloch (2018) Prefix sums and their applications
2. https://en.cppreference.com/w/cpp/algorithm/inclusive_scan
3. https://numpy.org/doc/stable/reference/generated/numpy.cumsum.html
4. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/pragmas/intel-specific-pragma-reference.html#intel-specific-pragma-reference
5. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/details-about-intrinsics.html
6. https://www.openmp.org/spec-html/5.0/openmpsu42.html#x65-1400002.9.3.1
7. https://www.openmp.org/spec-html/5.0/openmpsu45.html#x68-1940002.9.6

intel software

THE PARALLEL UNIVERSE

Printed in USA 421/IH Please Recycle.