ORNL/TM-2016/391

Effective Vectorization with OpenMP 4.5

Joseph Huber
Oscar Hernandez
Graham Lopez

March 2017

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

Study was performed as a part of the SULI program sponsored by the Department of Energy at Oak Ridge National Laboratory, managed by UT-Battelle for DOE. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

ORNL/TM-2016/391

Comuter Science and Mathematics Devision

Effective Vectorization with OpenMP 4.5

Usage of OpenMP 4.5 SIMD Directives

Joseph Huber
Oscar Hernandez
Graham Lopez

March 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

- Special thanks to Michael Klemm and Xinmin Tian from Intel for reviewing and providing us valuable feedback on the report. Their comments and corrections helped us understand the rationale and possible implementations of several of the SIMD directives and clauses.
- HACCmk Compiler benchmark https://asc.llnl.gov/CORAL-benchmarks/#haccmk

ABSTRACT

This paper describes how the Single Instruction Multiple Data (SIMD) model and its extensions in OpenMP work, and how these are implemented in different compilers. Modern processors are highly parallel computational machines which often include multiple processors capable of executing several instructions in parallel. Understanding SIMD and executing instructions in parallel allows the processor to achieve higher performance without increasing the power required to run it. SIMD instructions can significantly reduce the runtime of code by executing a single operation on large groups of data. The SIMD model is so integral to the processor's potential performance that, if SIMD is not utilized, less than half of the processor is ever actually used. Unfortunately, using SIMD instructions is a challenge in higher level languages because most programming languages do not have a way to describe them. Most compilers are capable of vectorizing code by using the SIMD instructions, but there are many code features important for SIMD vectorization that the compiler cannot determine at compile time. OpenMP attempts to solve this by extending the C++/C and Fortran programming languages with compiler directives that express SIMD parallelism. OpenMP is used to pass hints to the compiler about the code to be executed in SIMD. This is a key resource for making optimized code, but it does not change whether or not the code can use SIMD operations. However, in many cases critical functions are limited by a poor understanding of how SIMD instructions are actually implemented, as SIMD can be implemented through vector instructions or simultaneous multi-threading (SMT).* We have found that it is often the case that code cannot be vectorized, or is vectorized poorly, because the programmer does not have sufficient knowledge of how SIMD instructions work.

1. Introduction

SIMD hardware units in the processor provides the functionality to process multiple data elements simultaneously. The SIMD model [13, 12, 6] is becoming much more essential to unleash the power of modern processors and achieve higher performance. When a section of code makes efficient use of SIMD instructions, it is said to be *vectorized*. The amount of data that a processor needs to handle is continuously increasing from a wide range of applications, and in some cases that data can be processed simultaneously through the SIMD execution units.

Using SIMD instructions is challenging in higher level languages [5], and many languages do not have built-in mechanisms to allow the programmer to explicitly specify which sections of code should make use of SIMD instructions. The only alternatives are to either write assembly-level code, use compiler-specific low-level SIMD intrinsics, or rely on the compiler to vectorize the code automatically.

Auto-vectorization [1, 4, 7] can make use of the SIMD instructions if the compiler can detect loops or blocks of codes that can be vectorized. However, auto-vectorization is only consistently successful for limited usage on simple cases. The main challenge is that compiler's auto-vectorization relies on static analysis (e.g. with limited inter-procedural analysis, no dynamic information, etc.) to generate efficient code with no runtime information. This causes problems for the performance consistency of automatic vectorization across compilers and platforms, since the quality of static analysis on different compilers varies and additionally when targeting multiple platforms.

*In this TR we will not cover how OpenMP SIMD is implemented using SMT on GPUs. Implementations using this approach are a work in progress

In order to fully exploit the potential of SIMD hardware units, the OpenMP standard [8] and compiler vendors provide high-level SIMD programming extensions [10, 11, 3] to help users better utilize hardware vector units. OpenMP tries to solve these problems by providing a standardized way for programmers to specify information about the code in order to improve the compiler's analysis. These hints can include information about vectorization-safe loop structure, as well as user-defined functions that are suitable for vectorization. To more effectively use these directives, it is important to understand how the SIMD model works and what extensions [9] are needed for future OpenMP specifications.

2. Vectorization

The term "vectorization" refers to when multiple pieces of data are packed into a single, larger data type. This can be thought of as a one-dimensional array of fixed size. The boundaries of the array are determined by the data type packed in the array. SIMD instructions work on this array of packed data by modifying the entire array with a single processor instruction.

This can be illustrated with a simple example. Consider the set of eight integer numbers $\{34, 19, 23, 8, 43, 23, 4, 30\}$. If we want the sum of these numbers, it would require seven additions, namely $34 + 19 + 23 + 8 + 43 + 23 + 4 + 30 = 184$. Alternatively, consider the case where these numbers are grouped together to form two vectors: $\{34, 19, 23, 8\}$ and $\{43, 23, 4, 30\}$. Each element is then vertically added to form a new vector, $\{77, 42, 27, 38\}$. Then this vector is reduced by horizontally adding its first two and last two elements, $\{77 + 42\}$ and $\{27 + 38\}$ resulting in the vector $\{119, 65\}$. We then horizontally add the two vector components $119 + 65$ to come up with the final sum of 184. If each of these vector operations are supported by a vector instructions set, this method will only require *four* SIMD instructions as opposed to seven individual additions. This example highlights the potential savings in terms of number of instructions when an operation can be aggregated using vectorization.

In hardware, SIMD instructions use a large register in the processor called a vector register. A register is simply a small on-chip storage location used by the computer processor to do calculations. There are currently four sizes of vector registers available on modern x86 processors: the ZMM, YMM, XMM, and MM registers.* Other computer architectures that support SIMD instructions such as PowerPC or ARM have their own set of vector registers. The vector instruction set supported by the processor determines which of these registers can be used. Table 1† lists the available x86 vector instruction sets along with their main functionality. The operating system must also have explicit support for the vector registers, otherwise their contents could be changed during context switches. On machines that support multiple vector registers, the smaller vector registers are the lower-order bits of the larger registers, shown in Figure 1.‡

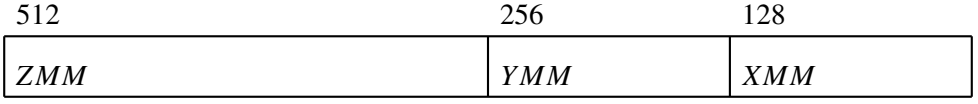

Figure 1. Layout of x86 vector registers

*The MM registers use the same register space as the x87 floating point unit

†AMD's 3DNow! instruction set is obsolete and unsupported on modern processors

‡On x86 architectures, These flags can be checked using the cpuid instruction or reading */proc/cpuid* on Linux machines

Instruction Set	Name	Functionality	Year
MMX	Multimedia Extensions	64 bit MMX for packed integers	1997
SSE	Streaming SIMD Extensions	128 bit XMM for floating point.	1999
SSE2	Steaming SIMD Extensions 2	XMM supports doubles and integers	2001
SSE3	Streaming SIMD Extensions 3	Horizontal operations added	2004
SSSE3	Supplemental SSE3	Horizontal and data movement	2006
SSE4.1	Streaming SIMD Extensions 4.1	Extra functionality	2007
SSE4.2	Streaming SIMD Extensions 4.2	Vector string instructions	2008
AVX	Advanced Vector Extensions	256 bit YMM for floating point.	2011
FMA	Fused Multiply Add	Fused multiply add instructions	2011
AVX2	Advanced Vector Extensions 2	YMM supports packed integers	2013
AVX512	Advanced Vector Extensions 512	512 bit ZMM registers	2016

Table 1. Available x86 vector instruction sets

The amount of data that can be operated on simultaneously depends on the size of the data as well as the size of the vector register holding it. The number of elements that can fit into a vector register is called the *vector length*, these values are given in Table 2 in terms of C data types. The processor has a different set of instructions for working with each packed data type. This means the type of data that can be vectorized depends on the instructions the processor supports. For example, the SSE2 and AVX2 instruction sets allow the XMM and YMM registers to work on integers respectively. The MM registers cannot perform floating point instructions. Additionally, long doubles currently cannot be used with any vector register.[§]

Register	long double	double	float	long	int	short	char
64 bit MM	–	–	–	1	2	4	8
128 bit XMM	–	2	4	2	4	8	16
256 bit YMM	–	4	8	4	8	16	32
512 bit ZMM	–	8	16	8	16	32	64

Table 2. Vector lengths for different vector registers and data types

SIMD instructions allow the same operation to be performed simultaneously on multiple pieces of data. This allows a large set of independent operations to be broken down into a smaller set of packed data that can be handled in parallel. This parallelism allows large loops to execute in a fraction of the time. The effect vectorization has on a loop is similar to unrolling the loop by a certain factor. Consider a loop where each element of two arrays, *A* and *B*, are added to each other.

```
float A[SIZE], B[SIZE];
for (int i = 0; i < SIZE; i++){
    A[i] += B[i];
}
```

Each iteration of this loop performs the same independent addition operation between multiple pieces of data. These qualities allow this loop to be easily vectorized. The benefits of vectorization can be seen if the loop is unrolled by a certain factor. If the processor's architecture supports the SSE instruction set, the processor can make use of 128-bit XMM vector registers. Referring to Table 2, an XMM register is able to operate on four floats simultaneously. So, the loop can be unrolled by a factor of four. A remainder loop is

[§]Some compilers alias long doubles as standard doubles

required to calculate the final iterations of the loop if the number of iterations is not divisible by the unrolling factor.

```
float A[SIZE], B[SIZE];
int i;
for (i = 0; i < SIZE - 4; i += 4)
    A[i] += B[i];
    A[i+1] += B[i+1];
    A[i+2] += B[i+2];
    A[i+3] += B[i+3];
}
for (i = i; i < SIZE; i++){
    A[i] += B[i]
}
```

By unrolling the loop by a factor of four, the number of iterations of the loop has been cut by one fourth. However, the number of instructions has not really changed. The loop can be vectorized by packing the set of values $\{A[i], A[i+1], A[i+2], A[i+3]\}$ and $\{B[i], B[i+1], B[i+2], B[i+3]\}$ each into a XMM vector register. A vertical floating point addition can then be performed between the two XMM vector registers and stored in another vector register. The destination vector register now contains the values $\{A[i]+B[i], A[i+1]+B[i+1], A[i+2]+B[i+2], A[i+3]+B[i+3]\}$. The entire vector register is then written back to memory with a single instruction.

```
float A[SIZE], B[SIZE];
int i;
for (i = 0; i < SIZE - 4; i += 4){
    simd_add_ps(&A[i], &B[i]);
}
for (i = i; i < SIZE; i++){
    A[i] += B[i];
}
```

Now, only a single instruction is in each iteration of the unrolled loop. This effectively cut the number of instruction the original loop executed by one fourth. The vectorized version of this loop will now execute approximated four times faster than the scalar (non-vector) loop. Unfortunately, many high level languages do not have built-in support for SIMD instructions. To use these instructions, the programmer must use intrinsic functions, like the one used in this example, to execute assembly code. Because of this, many programmers rely on the compiler's auto-vectorization optimization pass. The goal of the compiler's auto-vectorization pass is to transform the original loop into the vectorized loop automatically.

Vectorization is essential to high-performance applications. The speedup achieved in the previous example can easily be increased by using wider vector registers. If YMM vector registers are used the vectorized loop would execute eight times faster than the scalar loop. Furthermore, if ZMM vector registers are used, the vectorized loop would then be sixteen times faster than the scalar loop. This effect is increased further when multi-threading is considered. If the example loop was executed using sixteen threads, the loop would ideally execute sixteen times faster. If each of these threads uses 512-bit ZMM registers, the vectorized and multi-threaded loop would then execute 256 times faster than the scalar, single-threaded loop. This is an impressive speedup. However, a comprehensive understanding of how the SIMD model is

implemented is crucial to fully utilize the performance capabilities of SIMD instructions. The following sections will discuss SIMD instructions on x86 processors in detail. Even though the following sections focus on the x86 architecture most of the concepts are common between multiple architectures.

2.1 SIMD Instructions

SIMD instructions are hardware instructions that modify the vector registers. Integer SIMD instructions can work on 8 bits, 16 bits, 32 bits, or 64 bit operands. In the C language, this will correspond to the char, short, int, and long data types respectively. Most of the basic arithmetic operations can be performed on packed integer data: addition, subtraction, multiplication, and, or, xor, bit-shifting, and negation, among others. There is no instruction for packed integer division or modulus. SIMD division or modulus by a constant can, however, be implemented as a series of multiplications and bit-shifting instructions. Additionally, packed bytes have no shifting instruction or multiplication instruction. The vector instruction set supported by the processor determines if these instructions are available. Most integer instructions are available from the SSE2 instruction set for XMM registers and the AVX2 instruction set for YMM registers.

Vector registers can be operated on *vertically* or *horizontally*. A vertical operation works by performing the instruction between each of the corresponding elements of the vector register; this can be thought of as $X[i] + Y[i]$. These operations are called vertical because if both vector registers are placed above each other, the two elements operated on are covered by a vertical line. This can be seen in Figure 2. Conversely, horizontal operations are performed between a vector register and itself; this can instead be thought of as $X[i] + X[i + 1]$. Vertical operations are most common in vectorized code because they are similar to unrolling a loop. Horizontal vector operations are typically used for vector reductions where the vector register is reduced to a single value.

X_0	X_1	X_2	X_3
+	+	+	+
Y_0	Y_1	Y_2	Y_3
↓	↓	↓	↓
$X_0 + Y_0$	$X_1 + Y_1$	$X_2 + Y_2$	$X_3 + Y_3$

Figure 2. Vertical SIMD addition between four packed elements

SIMD floating point instructions can work on 32 bit or 64 bit floating point values, corresponding to the float and double C data types. [¶] Floating point instructions are more complicated than integer instructions. They can be up to *3–4x slower* than their integer equivalents. Functionally, they behave similarly to integer instructions, operations are performed vertically between each element individually, or horizontally within the vector register itself. Additionally, there are floating point instructions for square roots and division, among others. Integer SIMD instructions and floating point SIMD instructions should not be mixed because integers and floating point numbers use different data formats. Packed floating point values can be converted to packed integers if needed.

Floating point instructions can also work on floating point data by performing so-called fused multiply add instructions. These allow the processor to calculate the result of a multiplication and an addition in a single

[¶]There is a single instruction that converts packed 16 bit floats to 32 bit floats

instruction. There are several different versions of these instructions to calculate $(X * Y + Z)$, $(X * Y - Z)$, $(-X * Y + Z)$, and $(-X * Y - Z)$. Additionally, there are other instructions for alternating additions or subtractions. These instructions are available with the FMA instruction set. Some future instruction sets may extend this instruction so it may be used on integers as well.

The floating point instructions also support scalar instructions that only modify the first 32 or 64 bits of the vector register. This is a common way for floating point instructions to be done on x86 machines, and it is now handled in a much more straightforward way than previously with the old x87 floating point stack. This way of handling floating point instructions is more similar to other computer architectures like PowerPC or ARM where floating point values are stored in separate registers but treated like integer registers. The x87 stack is now only used for handling long doubles. Although these instructions are not SIMD instructions, they modify only the least significant bits of the vector register and leave the rest unchanged. Therefore, these instructions can sometimes be used within vectorized code.

2.1.1 Data Movement

The amount of time spent moving data between vector registers is often what determines if using SIMD instructions results in a performance increase or not; if more time is spent moving data between the vector registers than spent on calculations, scalar code will usually out-perform vectorized code. Data can be moved into a vector register from memory, a general purpose register, or another vector register. When loading from a general purpose register, only the low 32 or 64 bits are used, whereas loads from memory can load the entire vector, the low 32 bits, or the low 64 bits of the vector. For example, if a packed load from memory was performed on an array of 32 bit integers into a 128 bit register starting at $A[j]$, the vector register would contain $[A[j], A[j+1], A[j+2], A[j+3]]$. Loading a vector register with a single 32 bit value can be seen in Figure 3.

X	–	–	–
Move	*Move*	*Move*	*Move*
Y_0	Y_1	Y_2	Y_3
↓	↓	↓	↓
X	Y_1	Y_2	Y_3

Figure 3. Moving a single value into a vector register

Movements between other vector registers or memory can commonly be *shuffled*, *blended*, or *unpacked* among others. A shuffle instruction uses the source vector, rearranges it, and stores it back. This is controlled by an immediate operand that selects which element of the source vector register is placed into which element of the destination vector register. For a 32 bit shuffle on a 128 bit register, this is done with an 8 bit immediate value. A set of two bits ranging from zero to three chooses which element in the source vector register to place in the destination vector register. A shuffle with $imm = 0$ copies the first element in the source to every location in the destination. This is called broadcasting and is commonly used for storing constants in vector registers. The AVX instruction set introduced an explicit broadcasting instruction that behaves identically. A shuffling instruction between four packed elements requires an 8 bit immediate value to choose between them. This can be seen in Figure 4. The AVX instruction set introduced a *permute* instruction that is effectively a shuffling instruction controlled by a vector mask rather than an immediate value. An issue with data movement instructions is that often when the size of the vector

register is increased, the immediate value becomes insufficient to describe each element. The result is several instructions that do essentially the same operation with slightly different operands.

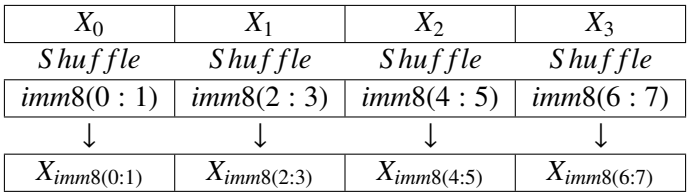

Figure 4. Shuffle operation on four packed elements

Blending instructions are used to choose between the values contained in two different vector registers. This can either be done with an immediate value or a bit-mask contained in a separate vector register. For a 32 bit blend with a 128 bit vector register, the first four bits of the immediate value determine which values are changed. An example blending operation can be seen in Figure 5

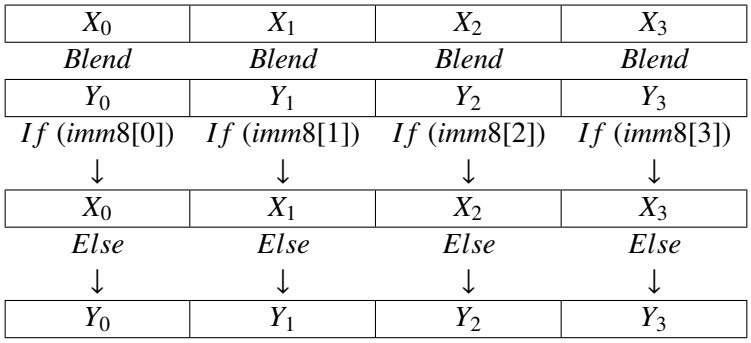

Figure 5. Blend operation on four packed elements

Unpacking instructions work by taking the lower or upper halves of a vector register and interleaving them. This operation does not require an immediate value. Unpacking instructions are typically used to combine the low bits of two vector registers into a single vector register. This allows data from multiple locations to be collected into a single vector register. This operation is shown in Figure 6.

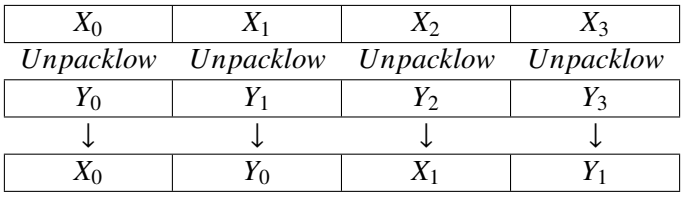

Figure 6. Unpacking the low elements on four packed elements

Data can be moved from the high half of the vector register to the low half or vice-versa. This operation only applies to floating point values. Because this operation is limited to floating point values it is less common than the other data movement instructions. However, since it is simply moving data it can theoretically be used on packed 32 bit or 64 bit integers. An example of this operation is shown in Figure 7. There is another set of instructions that behaves similarly to this instruction that is mostly used for exchanging data between the high 128 bits and low 128 bits of a YMM register.

Finally, there is some support for an insertion instruction. The insertion instruction places a value or vector register into an element of another vector register. This is controlled by an immediate operand that indicated which element of the destination vector register will be replaced. This instruction is commonly used to insert smaller vector registers into larger vector registers.

X_0	X_1	X_2	X_3
MoveLowHigh	*MoveLowHigh*	*MoveLowHigh*	*MoveLowHigh*
Y_0	Y_1	Y_2	Y_3
↓	↓	↓	↓
Y_0	Y_1	X_1	X_2

Figure 7. Movement from the low elements to the high elements

The performance of SIMD instructions depends on how quickly data can be moved into the vector registers and stored back to memory. Situations where all of the data is stored in a completely continuous one-dimensional array are ideal for SIMD instructions, but this is not always feasible in real-world applications. Data can be loaded or stored with a given stride in memory or completely irregularly. When data is loaded from disjoint memory locations and collected into continuous memory, it is called *gathering*. Likewise, when data is stored into disjoint memory locations from continuous memory it is called *scattering*.

These instructions are essential to vectorization because vector registers can be seen as a continuous array of fixed size. If data is stored in disjoint memory, it must be gathered into a vector register and then scattered back to memory. The AVX2 instruction set introduced a gathering instruction capable of loading a vector register with a set of memory locations stored in a vector register.

```
// Gathering
for i from 0 to 3
    xmm[i] = X[index[i]];
// Scattering
for i from 0 to 3
    X[index[i]] = xmm[i];
```

Gathering and scattering is an important part of vectorized code because not all applications access memory in a linear fashion. However, there is no way to explicitly express these instructions when relying on the compiler's auto-vectorization. The compiler will attempt to gather and scatter the memory, but it can be inefficient in cases where the memory can be reused. The only way around this issue would be to use vector intrinsics, template classes, or assembly language. Intrinsics are essentially functions that call inline assembly code. Template classes work in a similar fashion but attempt to make programming simpler by making a vector object and overloading operators between them to call assembly code.

2.1.2 Vector Conditional Masking

Conditional operations are very common in code. This is an issue for vectorized loops. Scalar code performs conditional statements by performing a comparison and conditionally jumping to another section of code depending on the result of the comparison. Vectorized loops execute multiple iterations of the loop

simultaneously. In this case, one iteration of the loop may require a conditional jump while another iteration of the loop will not require a jump. Because both of these iterations are happening at the same time the processor is unable to execute two different sections of code simultaneously. This problem is solved by instead unconditionally executing the entire loop using SIMD comparison instructions.

SIMD comparisons instructions create a bit-mask in the vector register. A vector comparison will vertically compare each element in the vector register. If the result evaluates to true, then all the bits for that element in the vector register are set; otherwise, all the bits are cleared. This mask is then used to unconditionally calculate the result of the conditional statement for each iteration of the loop. To illustrate this, consider a loop with a simple if, else statement.

```
float X[SIZE], Y[SIZE], Z[SIZE];
for (int i = 0; i < SIZE; i++){
    if (X[i] > Y[i]){
        Z[i] = X[i];
    }
    else {
        Z[i] = 0.0f;
    }
}
```

The scalar version of this loop would perform a comparison on the values stored in $X[i]$ and $Y[i]$. This comparison would set the flags to the result of the comparison. The processor would then check these flags and jump to one of the conditional sections of code. To vectorize this loop, the result of the conditional statement will need to be calculated without the use of conditional jumps. This can be done by generating a bit-mask from a SIMD comparison instruction. The bit-mask is then used to move the data into the vector register. This is done with a logical AND instruction because of the identity $X\&1 = X$ and $X\&0 = 0$. An example of operation can be seen in Figure 8.

X_0	X_1	X_2	X_3
$<$	$<$	$<$	$<$
Y_0	Y_1	Y_2	Y_3
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$111\ldots111$	$000\ldots000$	$111\ldots111$	$000\ldots000$
AND	AND	AND	AND
X_0	X_1	X_2	X_3
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
X_0	0	X_2	0

Figure 8. A conditional statement that returns X if true and 0 if false

2.1.3 Vector Constants

Vector registers cannot load constant values like general purpose registers can. Vector registers can only be loaded with values from a general purpose register or a memory location. If a constant is known at compile-time it will be placed in static memory and loaded into the vector register from that memory

Data Type	Alignment
char	1
short	2
int	4
long	8
float	4
double	8
long double	16
XMM register	16
YMM register	32
ZMM register	64

Table 3. memory alignment for each data type

address. However, if a constant is only known at runtime, it will need to be broadcasted into the vector register. This is done by loading the low bits of the vector register from a general purpose register and then performing a shuffle operation with the immediate set to 0 or with a broadcast instruction (cf. Sec. 2.1.1). This causes every location in the destination register to contain the element loaded from the general purpose register.

Some SIMD instructions can be used to create constants without using memory and wasting cache space. Performing a logical XOR between a vector register and itself will cause the entire vector register to be set to zero. A vector comparison for equality performed between a vector register and itself will produce a vector register with all bits set. This value can then be shifted to create other constants.

2.2 Memory Alignment

Memory alignment is an important consideration for writing high performance code. Memory alignment refers to the divisibility of the memory location of a value stored in memory. If a variable's memory address is divisible by n, it is said to be aligned by n or on a n byte boundary. So, for example, if a variable x was stored at the memory address $0x37FD10$, it would be aligned on a 1, 2, 4, 8, and 16 byte boundary. Each data type should ideally be stored at a memory location aligned by the smallest power of two that can contain the size of the data type. These values are given in Table 3. Reading or writing from unaligned memory is slower than aligned memory and on many computer architectures will result in a segmentation fault, so memory should always be aligned, if possible.

Accessing unaligned memory is slower because of how memory is read from the cache. A cache is organized as a set of cache lines that store a set of continuous memory. The x86 processor can read or write to unaligned memory addresses without any penalty if it is within the same cache line. However, if the memory access crosses this cache line boundary, the processor is actually accessing the memory from two cache lines along with a shifting operation to combine the two accesses. When a cache line is split by a memory access it is usually about *three times* slower than a standard memory access. Memory is aligned by a power of two so it evenly divides a cache line. Additionally, if the memory address is aligned, some processors are capable forwarding memory writes directly to a subsequent read without using the cache.

Memory alignment is especially important when accessing memory with SIMD instructions. SIMD

instructions can read or write a large amount of data in a single instruction. If the base memory address is unaligned, then a greater number of memory accesses will split a cache line. When the vector register becomes larger, the ratio of standard reads to cache line splits decreases and the penalty becomes more severe. For example, if data was read sequentially from a misaligned memory address with a 64 byte cache line size, there would be a cache line split every four reads for a XMM register, every other read for a YMM register, and every read for a ZMM register. If a cache line split is three times slower than an access contained in the cache line, the average speed penalty will be *50%* for a XMM register, *100%* for a YMM register, and *200%* for a ZMM register.

Memory alignment is also a requirement for several SIMD instructions. Prior to the AVX instruction set, almost every SIMD instruction that used a memory location required memory alignment. If the memory location was not aligned it would cause a segmentation fault. If the data is guaranteed to be aligned, these instructions can be safely used. The AVX instruction set relaxed this requirement for several instructions but others still require the address to be aligned. However, this does not change the fact that unaligned memory accesses are slow.

The compiler will automatically align most data known at compile time on a 32 or 16 byte boundary. Dynamically allocated memory is not guaranteed to return aligned addresses so they should be aligned by the memory allocator or the programmer. Some compilers offer the ability to specify memory alignment as well as dynamic memory allocation that returns aligned pointers. Aligning memory to a power of two can be done by setting the least significant bits to zero. This can be accomplished with a logical AND instruction with a constant where every bit is set except the last n bytes to align it to 2^n. This constant is most easily generated with a negative number. To ensure that there is enough memory after the alignment, the programmer should allocate $n - 1$ extra bytes when aligning by n. An example of aligning a memory location by 16 bytes is given in Figure 9. A simple function can be made to align dynamically allocated memory. If this function is used, a copy of the original pointer should be saved so the memory can be deallocated.

Binary	Hexadecimal
$10111110011101010010001010011110_2$	$5F3A914E_{16}$
AND	
$11111111111111111111111111110000_2$	-10_{16}
↓	
$10111110011101010010001010000000_2$	$5F3A9140_{16}$

Figure 9. Aligning a memory location by 16

2.3 AVX512 Additions

The AVX512 instruction set is the newest vector instruction set for the x86 architecture. The AVX512 instruction set extends the 256 bit YMM registers into 512 bit ZMM registers, allowing twice as much data to be handled in parallel when compared with YMM registers and four times as much when compared with XMM registers. Additionally, much of the functionality of the previous vector instruction sets has been changed. These changes will further extend the potential performance of vectorized code. The AVX512 instruction set consists of several smaller instruction sets, shown in Table 4. Currently, only the Intel Knights Landing (KNL) supports any of the AVX512 instruction sets. The Knights Corner uses a different

Instruction Set	Name	Functionality
AVX512F	Foundation	Fundamental additions with AVX512
AVX512CD	Conflict Detection	Conflict detection for vector stores
AVX512ER	Exponential and Reciprocal	Fast reciprocal approximations
AVX512PF	Prefetch Instructions	Prefetching for gather and scatter
AVX512BW	Byte and Word	Instructions for packed 8 bit and 16 bit
AVX512DQ	Doubleword and Quadword	Additional instructions
AVX512VL	Vector Length	Support for smaller AVX512 instructions
AVX512IFMA	Integer Fused Multiply Add	Integer FMA instructions
AVX512VBMI	Vector Bit Manipulation	Additional bitwise instructions

Table 4. Available AVX512 vector instruction sets

instruction set that also uses 512 bit vector registers, but this instruction set is likely going to be replaced by the AVX512 instruction set.

2.3.1 Mask Registers

Previously, vector instructions were masked by performing bitwise operations between vector registers that have been loaded with bit-masks (cf. Sec. 2.1.2). The AVX512F instruction set adds a new set of separate mask registers that now perform this operation. Additionally, these mask registers can be used in many more ways than a traditional bit-mask. Mask registers allow the result of the SIMD operation to be controlled on an element by element basis. Mask registers are loaded with a set of bits that correspond to each element of a vector register. For example, the first bit of the mask will correspond to the first element of the vector register, and so forth. This mask register can then be applied to most SIMD operations. When a mask register is applied to a destination register, it determines whether or an element will be updated. This is shown in Figure 10.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
+	+	+	+	+	+	+	+
Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
↓	↓	↓	↓	↓	↓	↓	↓
1	1	0	0	0	0	1	1
$X_0 + Y_0$	$X_1 + Y_1$	X_2	X_3	X_4	X_5	$X_6 + Y_6$	$X_7 + Y_7$

Figure 10. Masked vector addition

The use of mask registers allows for several performance improvements in vectorized code. Mask registers completely block the corresponding element of the vector register from being used. This saves power and prevents exceptions from being thrown, such as segmentation faults or division by zero. Mask registers also allow remainder loops to be removed. The remaining number of iterations can now be executed simultaneously using a mask register. Additionally, the use of mask registers allows more registers to be used for calculations rather than holding bit-masks.

2.3.2 Embedded Broadcasting

Broadcasting instructions are important for vectorized code. Often a constant value must be placed into each element of a vector register. This is traditionally done with a shuffling instruction or a broadcasting instruction. The AVX512 instruction set allows broadcasting instructions to be encoded in the instruction itself. 32 bit, 64 bit, or 128 bit values can be broadcasted from a general purpose register or memory. Embedded broadcasting removes the need for the additional broadcasting instruction as well as the need for additional vector registers to hold the broadcasted value.

2.3.3 Compress and Expand

Compression and expansion is important for reducing the memory requirements for a given program. Compression allows the programmer to select disjoint elements in an array and store them in continuous memory. Expansion has the opposite effect, a continuous set of memory can be stored in disjoint elements of another array. Previously, both compression and expansion have been difficult to implement with SIMD instructions because it requires conditionally gathering or scattering data. The AVX512 instruction set introduced two instructions for compression and expansion. These instructions use the mask registers to select the disjoint elements that the data will be placed into or taken from. This allows for compression and expansion to be easily performed with a few instructions using the AVX512 instruction set

```
int j = 0;
for (int i = 0; i < SIZE; i++){
    if (A[i] > 0){
        B[j] = A[i];
        j++;
    }
}
```

2.3.4 Conflict Detection

The AVX512 instruction set includes a conflict detection instruction. This instruction returns a bit-mask indicating if there are any duplicate elements inside the vector register. The returned bit-mask can then be used to handle the conflict. Conflict detection is used to check if a scattering instruction is going to perform a write to a memory location more than once. If a write is performed to the same memory location the result could be incorrect. Consider a simple histogram that counts the number of times the values stored in an array *A* occurs

```
for (int i = 0; i < SIZE; i++)
    hist[A[i]]++;
```

If this loop was vectorized it would require the values in *A* to be gathered, incremented, and scattered back to memory. However, if there are any duplicate values within the vector register the histogram will only be incremented a single time. Conflict detection allows many previously problematic loops to be efficiently vectorized.

2.3.5 Reciprocals and Exponentials

Some reciprocal calculations can be done faster than other calculations. Previously there were only a few instructions that used this approximation that only applied to floats. The AVX512 instruction set introduces several new approximated reciprocal instructions that can be used on both floats and doubles. Additionally, the level of approximation can be chosen. These instructions allow more instructions to use fast approximations to complex mathematical operations, increasing overall performance at the cost of precision.

2.4 x86 Vector Instructions

Vectorization is accomplished by using a set of SIMD processor instructions with vector registers. Source code can be vectorized, if and only if, the compiler can manage to use these instructions and registers. Because of this restriction, SIMD parallelism is often more difficult to achieve than other forms of parallelism. These restrictions must be kept in mind when relying on the compiler's auto-vectorization. The compiler is only able to efficiently vectorize code if it is already written with vectorization in mind. This will require some knowledge of assembly language. This section will describe how some loops to be vectorized in a high level languages can be vectorized by the compiler in assembly language.

These examples will be given using Intel's x86 assembly syntax.[‖] The instructions are given in the form *opcode dest, src* or *opcode dest, src1, src2* and memory accesses are enclosed in brackets, similar to using the '*' operator in C. All memory accesses are byte-addressed. The registers used with the name XMM, YMM, or ZMM are vector registers and registers with the name k0-k7 are mask registers used in AVX512. The other registers such as *ax*, *bx*, or *cx*, are general purpose registers. The prefix determines the size of the general purpose register, a 'r' indicates a 64 bit value and an 'e' indicates a 32 bit value. All sizes given in these examples will be assumed to be divisible by the vector length and all pointers will be aligned to simplify the code. Additionally, some variable names or memory locations in the C code will be preserved in the assembly code.

2.4.1 SAXPY

The first example will be a simple SAXPY loop. The SAXPY loop performs a single precision multiplication on *X* and adds it to *Y*.

```
float X[SIZE], Y[SIZE], A;
for (int i = 0; i < SIZE; i++){
    X[i] = A*X[i] + Y[i];
}
```

This loop can be easily vectorized using a SIMD floating point multiplication and addition. If the SSE instruction set is supported on the processor, then referring to Table 2, the loop can process four floating point values simultaneously. To vectorize this loop, the compiler will likely generate assembly code similar to this example.

[‖] Compilers on UNIX machines typically use AT&T syntax by default

```
    movss   xmm0, [A]            ; Move single float A into XMM0
    shufps  xmm0, xmm0, 0        ; Broadcast A into XMM0
    xor     rcx, rcx             ; i = 0
saxpy_loop:
    movaps  xmm1, [X + rcx*4]    ; XMM1 = X[i],X[i+1],X[i+2],X[i+3]
    mulps   xmm1, xmm0           ; XMM1 = A*X[i],A*X[i+1],A*X[i+2],A*X[i+3]
    addps   xmm1, [Y + rcx*4]    ; XMM1 = A*X[i]+Y[i],A*X[i+1]+Y[i+1],...
    movaps  [X + rcx*4], xmm1    ; Store XMM1
    add     rcx, 4               ; i += 4
    cmp     rcx, SIZE            ; Compare i and SIZE
    jl      saxpy_loop           ; Jump if i < SIZE
```

Today, most all Intel processors support the SSE instruction set. However, newer processors can also use the AVX instruction set. If the AVX instruction set is used then eight floating point values can be calculated simultaneously. The AVX instruction set also supports an explicit broadcasting instruction, so the load and shuffle used in the SSE version of the loop can be replaced with a single instruction. Additionally, if the processor also supports the FMA instructions set the processor can use fused multiply add instructions to further reduce the instruction count.

```
    vbroadcastss    ymm0, [A]    ; Broadcast A into YMM0
    xor     rcx, rcx             ; i = 0
saxpy_loop:
    vmovaps ymm1, [X + rcx*4]    ; Load 8 packed floats into YMM1
    vfmadd213ps ymm1, ymm0, [Y + rcx*4] ; FMA by source numbers: 2*1 + 3
    vmovaps [X + rcx*4], ymm1    ; Store 8 packed floats from YMM1
    add     rcx, 8               ; i += 8
    cmp     rcx, SIZE            ; Compare i and SIZE
    jl      saxpy_loop           ; Jump if i < SIZE
```

2.4.2 Gather

When data is gathered it is taken from disjoint locations in memory and stored in continuous memory. This operation is common when working with look-up tables or histograms.

```
int A[SIZE], tab[TABLE_SIZE];
for (int i = 0; i < SIZE; i++){
    A[i] = tab[A[i]];
}
```

This operation is troublesome for vectorization because it requires a large amount of data movement. If the SSE instruction set is used then the data must be loaded individually and then combined into a vector register. This is almost certainly slower than a scalar implementation of this loop.

```
    xor     rcx, rcx             ; i = 0
gather_loop:
    movsx   rax, [A + rcx*4]     ; Sign extend 32 bit A[i] to 64 bit
    movsx   rbx, [A + rcx*4 + 4]; rbx = A[i + 1]
    movsx   rdx, [A + rcx*4 + 8]; rdx = A[i + 2]
    movsx   rsi, [A + rcx*4 + 12] ; rsi = A[i + 3]
```

```
    movd     xmm0, [tab + rax*4] ; xmm0 = [1, 0, 0, 0]
    movd     xmm1, [tab + rbx*4] ; xmm1 = [2, 0, 0, 0]
    movd     xmm2, [tab + rdx*4] ; xmm2 = [3, 0, 0, 0]
    movd     xmm3, [tab + rsi*4] ; xmm3 = [4, 0, 0, 0]
    punpckldq    xmm0, xmm2      ; xmm0 = [1, 3, 0, 0]
    punpckldq    xmm1, xmm3      ; xmm1 = [2, 4, 0, 0]
    punpckldq    xmm0, xmm1      ; xmm0 = [1, 2, 3, 4]
    movdqa   [A + rcx*4], xmm0   ; Store 4 packed ints from xmm0
    add      rcx, 4              ; i += 4
    cmp      rcx, SIZE           ; Compare i and SIZE
    jl       gather_loop         ; Jump if i < SIZE
```

Because this operation is so slow, loops involving took-up tables were typically not vectorized. The AVX2 instruction set introduced a gather instruction to solve this problem. The gather instruction uses a pointer offset in a general purpose register and a vector register containing packed indices. These memory locations are then gathered into the destination vector register. Additionally, a mask register is used to conditionally gather indices, if needed.

```
    xor      rcx, rcx            ; i = 0
    lea      rsi, [Tab]          ; rsi = &Tab[0]
gather_loop:
    vmovdqa ymm0, [A + rcx*4]    ; Get 8 packed indices from A
    vpcmpeqd    ymm1, ymm1, ymm1; Get mask of all '1'
    vpxor   ymm2, ymm2, ymm2     ; Zero ymm2
    vpgatherdd  ymm2, [rsi + ymm0*4], ymm1  ; Gather from Tab into ymm2
    vmodqa  [A + rcx*4], ymm0    ; Store 8 ints from ymm0
    add      rcx, 8              ; i += 8
    cmp      rcx, SIZE           ; Compare i and SIZE
    jl       gather_loop         ; Jump if i < SIZE
```

2.4.3 Reduction

A reduction loop is common in many parallel applications. Reduction is performed by collecting several values into a single value. This is commonly done with addition or subtraction.

```
int A[SIZE], sum;
for (int i = 0; i < SIZE; i++){
    sum += A[i];
}
```

Vector reduction is performed by using the vector register to perform several partial sums concurrently. Then after the loop is completed the partial sums are aggregated into a final sum. This is done either using horizontal additions or shifting and addition.

```
    pxor     xmm0, xmm0          ; Zero vector register
    xor      rcx, rcx            ; i = 0
reduction_loop:
    paddd    xmm0, [A + rcx*4]   ; Collect partial sums
    add      rcx, 4              ; i += 4
    cmp      rcx, SIZE           ; Compare i and SIZE
```

```
    jl      reduction_loop      ; Jump if i < SIZE

    phaddd  xmm0, xmm0          ; xmm0 = [1+2,3+4,1+2,3+4]
    phaddd  xmm0, xmm0          ; xmm0 = [1+2+3+4, ...]
    movd    eax, xmm0           ; eax = 1+2+3+4
```

If the loop is implemented using wider vector registers some additional instructions will be required to aggregate the larger value. However, this is unsubstantial compared to the benefits of performing more iterations of the loop in parallel.

```
    vpxor   ymm0, ymm0, ymm0    ; Zero vector register
    xor     rcx, rcx            ; i = 0
reduction_loop:
    vpaddd  ymm0, [A + rcx*4]   ; Collect partial sums
    add     rcx, 4              ; i += 4
    cmp     rcx, SIZE           ; Compare i and SIZE
    jl      reduction_loop      ; Jump if i < SIZE

    vphaddd ymm0, ymm0, ymm0    ; Reduce 128 bit vectors
    vphaddd ymm0, ymm0, ymm0    ; [1+2+3+4,.,.,.,5+6+7+8,.,.,.]
    vpermi128   ymm1, ymm0, 1   ; Move high 128 bits to ymm1
    vpaddd  ymm0, ymm1          ; (1+2+3+4) + (5+6+7+8)
    vmovd    eax, ymm0          ; eax = 1+2+3+4+5+6+7+8
```

2.4.4 Linear

A variable with a linear relation is simply a variable that is increased by a certain value each iteration of the loop.

```
int A[SIZE];
for (int i = 0; i < SIZE; i++){
    A[i] = i;
}
```

This can be vectorized by storing two values in a vector register, the linear step of the variable, and set of initial values. If this loop was vectorized using the SSE2 instruction set, according to Table 2, four iterations of the loop can be performed in parallel. In this case, the set of initial values stored in the vector register will be $\{0, 1, 2, 3\}$ and the linear step will be $\{4, 4, 4, 4\}$. The linear step can either be stored in memory or broadcasted.

```
    movdqa  xmm0, [INITIAL]     ; xmm0 = {0, 1, 2, 3}
    movdqa  xmm1, [STEP]        ; xmm1 = {4, 4, 4, 4}
    xor     rcx, rcx            ; i = 0
reduction_loop:
    movdqa  [A + rcx*4], xmm0   ; Store 4 packed ints from xmm0
    paddd   xmm0, xmm1          ; Increment each value by the step
    add     rcx, 4              ; i += 4
    cmp     rcx, SIZE           ; Compare i and SIZE
    jl      reduction_loop      ; Jump if i < SIZE
```

2.4.5 Conditional Statement

Conditional statements are common inside loops and can be difficult to vectorize. This loop will simply conditionally assign some values.

```
float A[SIZE], B[SIZE], X[SIZE], Y[SIZE], Z[SIZE];
for (int i = 0; i < SIZE; i++){
    if (A[i] < B[i]){
        Z[i] = X[i];
    }
    else {
        Z[i] = Y[i];
    }
}
```

This conditional statement can be vectorized with the use of masking operations. A vector comparison will vertically compare each value and create a bit-mask. This mask will be used to unconditionally calculate the entire loop. If the comparison is true, then the value will be saved in the vector register, otherwise it is discarded. The result of both conditional statements are then combined at the end.

```
    xor     rcx, rcx            ; i = 0
conditional_loop:
    movaps  xmm0, [A + rcx*4]   ; Move A[i] into xmm0
    cmpltps xmm0, [B + rcx*4]   ; get mask for (A[i] < B[i])
    movaps  xmm1, [X + rcx*4]   ; Move X[i] into xmm1
    andps   xmm1, xmm0          ; X[i] if true 0 if false
    andnps  xmm0, [Y + rcx*4]   ; Y[i] if false 0 if true
    orps    xmm1, xmm0          ; Combine both results
    movaps  [Z + rcx*4], xmm1   ; Store 4 floats from ymm1
    add     rcx, 4              ; i += 4
    cmp     rcx, SIZE           ; Compare i with SIZE
    jl      conditional_loop    ; Jump if i < SIZE
```

This same conditional statement can be used with the AVX512 instruction set's masking registers. The result of the vector comparison will instead be placed inside a mask register. The mask register will then dictate which elements of the destination register get updated. This method massively simplifies the process.

```
    xor     rcx, rcx
conditional_loop:
    vmovaps zmm0, [A + rcx*4]       ; Move A[i] into zmm0
    vcmpltps    k1, zmm0, [B + rcx*4]   ; Get mask into mask register
    vmovaps zmm0{k1}, [X + rdx*4]   ; Move true values into zmm0
    knotd   k1                      ; Invert mask
    vmovaps zmm0{k1}, [Y + rdx*4]   ; Move false values into zmm0
    vmovaps [Z + rcx*4], zmm0       ; store 16 floats from zmm0
    add     rcx, 16                 ; i += 16
    cmp     rcx, SIZE               ; Compare i with SIZE
    jl      conditional_loop        ; Jump if i < SIZE
```

3. OpenMP SIMD

Expressing SIMD parallelism is difficult outside of assembly code. Most compilers have auto-vectorization optimization passes that attempt to make use of the SIMD instructions but will often fail to vectorize code or cannot safely vectorize the code without additional information from the programmer. OpenMP is a directive-based programming application programming interface for shared memory and accelerator based systems. In OpenMP 4.0, SIMD directives were added to help compilers generate efficient vector code. The SIMD directives explicitly enable vectorization in the compiler and act as hints or instructions sent to the compiler's vectorization pass to improve its analysis and the quality of the vector code being generated. These directives also help to scope which SIMD statements in the code the compiler should attempt to vectorize. OpenMP's SIMD directives can be placed before a for loop or a function declaration.

3.1 SIMD loop directives

OpenMP SIMD directives can be placed above for loops with the syntax *#pragma omp simd [clause[[,] clause] ...]*, which marks the loop as a SIMD enabled loop or SIMD region. The additional clauses then pass hints to the compiler to improve the quality of the code or otherwise indicate its eligibility to be vectorized. This allows the programmer to specify certain SIMD instructions and provide information to the compiler that it cannot determine through static-analysis alone, such as specifying that a loop has no dependences across iterations or a specific dependence distance. OpenMP loop directives only apply to for or do loops that are in a canonical form, where the number of iterations is known when entering the loop. If these conditions are met, then the loop is in a form that can be vectorized. Loops that contain flow control such as return, break, or continue statements cannot be vectorized. OpenMP SIMD loop directives guarantee that the loop will execute multiple iterations of the loop using vector registers when possible. However, this vectorization can be hindered when the loop requires scalar instructions. In this case, the compiler will perform the scalar instruction multiple times and pack the results into a vector register.

3.1.1 SIMD aligned

#pragma omp simd aligned([ptr] : [alignment], . . .)

Data alignment is important for SIMD instructions. Unaligned memory accesses are always slower than aligned memory accesses if they cross a cache-line boundary. Additionally, some SIMD instructions can only be used with aligned memory addresses. However, the compiler often cannot determine the alignment properties of data that is linked from other files or when they are dynamically allocated. The *aligned* clause asserts to the compiler that a variable is aligned. Each pointer in the aligned clause can have a positive integer alignment applied to it. If no alignment value is given to the compiler, an implementation defined default value is assumed. Using this clause allows the compiler to safely use SIMD instructions that have strict alignment requirements. If this clause is used, the programmer is responsible for ensuring that the data is in fact aligned. Otherwise, the attempted use of aligned memory accesses on unaligned memory may result in segmentation faults.*

*We noticed that GCC implements this clause differently. The pointers are given in a comma separated list and have the alignment applied to all of them

3.1.2 SIMD reduction

#pragma omp simd reduction([operation] : [variable],...)

The *reduction* clause instructs the compiler to perform a vector reduction on a variable. A reduction operation is performed by computing a partial value inside the parallel region. When the parallel region ends, the partial values are then aggregated into the final value. The reduction clause does this by creating a private vector copy of the variable inside the SIMD loop which is used to store the partial values. When the SIMD region ends, the vector copy of the original variable is horizontally aggregated. The final value is then moved from the vector copy to the original variable. The reduction clause takes a character representing the type of reduction performed in the loop and a variable to be reduced inside the loop. The variable specified in the reduction clause is made private to the loop. Some compilers have difficulties detecting reductions automatically, so specifying them with the OpenMP reduction clause can give the compiler the information it needs to vectorize a loop.

3.1.3 SIMD safelen

#pragma omp simd safelen([value])

Loop carried data dependencies hinder parallel programming. If an iteration of the loop requires data that is only calculated in a previous iteration of the loop, it has a data dependency. Because vectorized loops perform multiple iterations of the loop concurrently, the value will change if a data dependency is present in the vectorized loop. The *safelen* clause guarantees that no iterations of the loop will execute simultaneously inside the SIMD loop if the distance between the iterations is smaller than the value specified in the safelen clause. The actual number of loop iterations that will be performed in a single iteration of the SIMD loop is implementation defined, but it will not exceed the value specified in the safelen clause. In some cases, this clause is required for correctness. The compiler may sometimes choose to use a vector length that violates the data dependency. If this value is specified, the compiler may be more aggressive when unrolling loops because it is guaranteed to not change the result.

3.1.4 SIMD collapse

#pragma omp simd collapse([value])

OpenMP SIMD directives can normally only be applied to the innermost loop in a chain of nested loops. The *collapse* clause causes the compiler to attempt to collapse the number of loops specified by the collapse clause into a single loop with a universal address space. This collapsed loop then has the SIMD directives applied to it. The compiler will often collapse loops by creating a single loop that iterates over the total number of iterations of the originally nested loops. If the loop indices were used to address memory, the indices used in the collapsed loop will need to be transformed using division and remainder operations which may not be efficient for vectorization because it does not guarantees contiguous data access. However, compiler optimizations are possible to collapse the iteration space and the data accesses to make vectorization profitable. In this case, the memory does not need to be scattered or gathered into the vector registers and the memory can simply be accessed in a single memory access operation (see Section 5.4).

3.1.5 SIMD linear

#pragma omp simd linear([variable] : [step], . . .)

When used in a SIMD loop context, the *linear* clause will perform a linear incrementation on a variable using SIMD instructions. The *linear* clause takes an integer variable and adds the linear step to the variable each iteration of the loop. This is performed by creating two private vectors inside the SIMD loop. The first vector holds the linear sequence created by adding the step value to the initial value of the variable. The second vector contains the linear step that increments the previous vector. For example, if the *linear* clause has a linear variable $N = 1$ with a linear step of 2 and a vector width of four is used, then the first private vector would contain $1, 3, 5, 7$. After another iteration in the SIMD loop the vector would then be $9, 11, 13, 15$. This is done by adding a vector containing $8, 8, 8, 8$ after each SIMD lane.

3.1.6 SIMD private / lastprivate

#pragma omp simd private([variable], . . .)

The *private* and *lastprivate* clauses control data privatization and sharing of variables for a SIMD Loop. The private clause creates an uninitialized vector inside the SIMD loop for the given variables. SIMD function arguments are private by default because they must correspond to a vector register in the hardware. The *lastprivate* clause provides the same semantics but also copies out the values produced from the last iteration to outside the loop. This clause allows the programmer to guarantee that no iterations of the SIMD loop overlap by making them private to each SIMD lane.

3.2 SIMD declare directives

SIMD enabled functions can be declared by placing *#pragma omp declare simd [clause[[,] clause] ...]* above the function declaration. When a function is declared as a SIMD function, the compiler will generate multiple versions of the function that can be used depending on the context of the function call. If the function was called from a scalar loop, it will use a scalar version of that function; likewise, if the function was called from within an SIMD region it will use a vectorized version of the function. Additionally, there are versions of the function that will be called if the function call is inside of a conditional statement. Finally, when the vectorized versions of a function are created, they have their own set of vectorized arguments. The types are *uniform*, *vector*, and *linear*. These types describe the vector arguments passed to the function. This distinction is important when calling a SIMD declare function within another SIMD declare function. If the arguments are incompatible, the function cannot be vectorized.

When a function is called in a SIMD enabled loop, the compiler will unroll the concurrent calls to that function. The compiler will then check if there are any valid SIMD enabled functions; a SIMD function is valid if it has a matching number of concurrent arguments and the uniform, vector, and linear argument types of the function are identical. The function itself is vectorized according to the uniform, linear, and vector arguments, and the compiler will unroll the SIMD function by the number of concurrent arguments it is expected to handle. If the argument is uniform or linear it will be able to know these values at compile time and use them in the function. Functions called within conditional statements will take an additional argument of the bit-mask generated from the conditional statement.

3.2.1 SIMD declare aligned

#pragma omp declare simd aligned([argument] : [alignment],...)

When used in a SIMD declare context, the *aligned* clause instructs the compiler that the pointers passed as function arguments are always going to aligned by the given alignment value. The compiler will use this information when it creates the vectorized versions of the function by instead using the aligned versions of the SIMD instructions. The alignment value is a positive integer. When the aligned clause is absent, the default alignment is implementation defined. This clause should only be used if the pointers are known to be aligned, otherwise segmentation faults may occur.

3.2.2 SIMD declare simdlen

#pragma omp declare simd simdlen([value])

The *simdlen* clause specifies the number of packed arguments the vectorized function will execute concurrently. If this value is not specified, the compiler will use a default value. The value given to the simdlen clause should correspond to the vector length of a hardware vector register. For x86 processors, these values are given in Table 2. If the simdlen is given as a multiple of a hardware vector register size, the compiler may fuse multiple vector registers into a single logical vector. The importance of the simdlen clause can be seen in the case where nested function calls return different data sizes. For example, if a function that returns a double calls a function that returns an integer it cannot be vectorized because the double function cannot provide enough data to the integer function to satisfy the simdlen requirements.

3.2.3 SIMD declare uniform

#pragma omp declare simd uniform([argument],...)

The *uniform* clause indicates that the given function argument will not change between any of the concurrent function calls in a SIMD loop. This indicates that the value is shared between the SIMD lanes of the loop. When a function argument is specified as uniform, the compiler can use this information to create more efficient code for the vectorized function. If the uniform clause is used for a pointer in conjunction with the *linear* clause, the compiler is able to use faster unit-stride memory instructions rather than gathering or scattering the data. This is because the combination of the linear and uniform clauses allows the compiler to know how the memory will be accessed when the function is executed concurrently.

3.2.4 SIMD declare linear

#pragma omp declare simd linear([argument] : [linearstep],...)

When used in a SIMD declare context, the *linear* clause indicates what the value of the function argument will be if the function is called multiple times concurrently. The argument placed in the *linear* clause will be increased by the linear step value between each successive function call. If a pointer is declared uniform and is accessed with a linear function argument, then the compiler can generate efficient vector memory accesses using the linear stride rather than gathering or scattering the data.

3.2.5 SIMD declare inbranch / notinbranch

#pragma omp declare simd inbranch / notinbranch

When a function is declared as a SIMD function, the compiler will create multiple versions of that function to be called under certain circumstances. If the function is called outside of a SIMD region, it will use a scalar version of that function. Likewise, if the function is called within an SIMD region, it will use a vectorized version of that function. Additionally, there are versions of the function that are used when the function is called within a conditional loop. Specifying that the SIMD declared function is *notinbranch* or *inbranch* simply instructs the compiler whether or not to create the versions of the function that handle calls from within a conditional branch. This allows for an improvement in the size of the code rather than performance.

3.3 SIMD Block-Level directives

Block-level SIMD directives can be placed inside of a SIMD loop and affect a specified section of code. Currently, the OpenMP 4.5 standard only supports a single block-level directive for SIMD regions. All other OpenMP directives cannot be used inside of a SIMD region. Block level directives are used by enclosing a structured block of code following the OpenMP directive.

3.3.1 ordered SIMD

#pragma omp ordered simd
structured block

When used inside of a SIMD region, the *ordered* clause causes the structured block of code to be executed using scalar instructions while maintaining the the parallel execution order. This allows for the use of instructions that cannot be vectorized or operations that provide incorrect results under certain conditions to be used inside of a SIMD region. However, even with the use of the *ordered* clause, flow control instructions such as break, continue, or return cannot be placed inside of a SIMD region. The *ordered* clause can drastically reduce the speed of a SIMD loop by preventing it from executing in parallel. This clause should only be used when scalar execution is required to produce a correct result and if the loop still benefits from being vectorized.

3.4 Vendor Specific OpenMP SIMD directives

Some vendors have provided extensions to the OpenMP SIMD construct that are not in the OpenMP 4.5 standard. These directives are not portable between compilers but can be important for architecture or compiler specific optimizations. Currently, the Intel compiler supports a single clause for OpenMP SIMD declare directives.

3.4.1 SIMD declare processor

#pragma omp declare simd processor(string)

By default, the compiler will often create binaries that are compatible with much older computer architectures. Compiler options such as *-x* for Intel and *-m* for GCC instruct the compiler to target a specific computer architecture, allowing the compiler to generate code that is optimal for that processor. However, this does not affect the code generated inside of SIMD-enabled functions that have not been inlined by the compiler. The *processor* clause controls the instructions and vector registers used inside SIMD-enabled functions. Without this clause, the compiler will often default to 128-bit wide XMM registers rather than 256-bit wide YMM registers or 512-bit wide ZMM registers. The processor clause takes its input as a string that corresponds to a computer architecture; these strings are given in Table 5. This extensions is only provided by the Intel compilers for x86 processors[†].

Processor ID	ISA	Registers
pentium_4	SSE2	XMM
pentium_4_sse3	SSE3	XMM
core_2_duo_ssse3	SSSE3	XMM
core_2_duo_sse4_1	SSE4.1	XMM
core_i7_sse4_2	SSE4.2	XMM
core_2nd_gen_avx	AVX	YMM
core_3rd_gen_avx	AVX	YMM
core_4th_gen_avx	AVX2	YMM
mic	KNC	ZMM
mic_avx512	AVX512	ZMM

Table 5. Supported target architectures for the *processor* clause

3.5 SAXPY example

Consider an example of a simple SAXPY loop. We will start with a scalar version of this program with no OpenMP directives. The SAXPY operation is placed in a separate function and called with each iteration of the loop. For this example, the saxpy function is not allowed to be inlined by the compiler.

```
void saxpy(float *X, float *Y, int i, float SA){
    Y[i] = SA*X[i] + Y[i];
}

int main(){
    float X[SIZE], Y[SIZE], SA;
    for (int i = 0; i < SIZE; i++){
        saxpy(X, Y, i, SA);
    }
}
```

Now let's try to vectorize this code. The constant SA can be broadcasted to each line in the vector register. Then the packed floats are loaded, multiplied, added, and finally stored back into Y. The problem is that the compiler cannot vectorize the SAXPY operation because it is in a function call. So we can declare the

[†]Support for different strings depends on the version of the compiler

SAXPY function as a SIMD function and place the main loop inside a SIMD region to tell the compiler to use the vectorized version of the SAXPY function. These examples were compiled using the Intel C Compiler version 16.0.3 and targeted the AVX2 instruction set. The code was run and timed on a Knights Landing processor.

```
#pragma omp declare simd
void saxpy(float *X, float *Y, int i, float SA){
    Y[i] = SA*X[i] + Y[i];
}

int main(){
    float X[SIZE], Y[SIZE], SA;
    #pragma omp simd
    for (int i =  i < SIZE; i++){
        saxpy(X, Y, i, SA);
    }
}
```

The speed has not increased, and it has actually gotten significantly slower with a $i.64x$ slowdown. To make matters worse, the Intel compiler gave a vectorization report indicating that both the SIMD loop and function were vectorized. So what is making this function slower? The declare clause does not actually give enough information to the compiler for it to to be effectively vectorized. Without enough information, the compiler can either not vectorize the loop or resorts to gathering and scattering to access the memory. This problem can be fixed by using SIMD declare to pass more hints about the function to the compiler. The compiler must have an idea of what the function is going to look like if it were called multiple times in series. Currently, the compiler does not know that the pointers given will not change between the function calls or that they are accessed sequentially. The *uniform* and *linear* clauses can pass this necessary information to the compiler.

```
#pragma omp declare simd uniform(X, Y) linear(i : 1)
void saxpy(float *X, float *Y, int i, float SA){
    Y[i] = SA*X[i] + Y[i];
}

int main(){
    float X[SIZE], Y[SIZE], SA;
    #pragma omp simd
    for (int i =  i < SIZE; i++){
        saxpy(X, Y, i, SA);
    }
}
```

Now the desired speed-up is achieved with the vectorized loop performing roughly $3.36x$ faster using the AVX2 instruction set. But for these examples, the addresses of X and Y have always been misaligned. The memory addresses are aligned and this information is passed to the compiler through the aligned clause. Alignment should make the memory accesses quicker and allow the code to use the aligned versions of the vector instructions.

```
#pragma omp declare simd uniform(X, Y) linear(i : 1) \
aligned(X, Y : 32) notinbranch
void saxpy(float *X, float *Y, int i, float SA){
    Y[i] = SA*X[i] + Y[i];
}

int main(){
    float X[SIZE], Y[SIZE], SA;
    #pragma omp simd aligned(X, Y : 32)
    for (int i =  i < SIZE; i++){
        saxpy(X, Y, i, SA);
    }
}
```

Now the aligned version of this function performs 7.47x faster than the original scalar loop. This code has successfully been vectorized with the use of OpenMP's SIMD directives. But this code could have been easily vectorized by most compilers' auto-vectorizer if the compiler inlined the *saxpy* function. However, if the *saxpy* function was declared in some other file that was linked in by the compiler, it would be impossible for the function to be inlined. This function can still be vectorized if OpenMP is used; declaring the function as *extern* will cause the loop to call the vectorized version of the function once the files are linked.

```
#pragma omp declare simd uniform(X, Y) linear(i : 1) \
aligned(X, Y : 32) notinbranch
extern void saxpy(float *X, float *Y, int i, float SA);

int main(){
    float X[SIZE], Y[SIZE], SA;
    #pragma omp simd aligned(X, Y : 32)
    for (int i =  i < SIZE; i++){
        saxpy(X, Y, i, SA);
    }
}
```

Becuase we are using the Intel C Compiler, we can also make use of the *processor* clause to better target the AVX2 and FMA vector instruction sets. This allows for the vectorized function to fully utilize the 256 bit YMM registers. After this change is made, the resulting code finally runs 24.12x faster than the original misaligned scalar loop without OpenMP directives. If the original code is also aligned it results in a 9.34x speedup. This is faster than the expected 8.00x speedup because of the use of the FMA instruction set.

```
#pragma omp declare simd uniform(X, Y) linear(i : 1) \
aligned(X, Y : 32) processor(core_4th_gen_avx)
extern void saxpy(float *X, float *Y, int i, float SA);

int main(){
    float X[SIZE], Y[SIZE], SA;
    #pragma omp simd aligned(X, Y : 32)
    for (int i =  i < SIZE; i++){
        saxpy(X, Y, i, SA);
```

```
    }
}
```

4. OpenMP SIMD Programming Guidelines

This section provides several programming guidelines for OpenMP programmers to develop correct and high performance SIMD programs using SIMD extensions in the OpenMP 4.5 specifications.

4.1 Ensure SIMD execution legality

When the OpenMP SIMD construct is applied to a for loop or do loop, the programmer guarantees that the loop can be partitioned into smaller chunks of SIMD execution. This chunk is created by executing several logical iterations of the loop in parallel using SIMD instructions. The size of this chunk is determined by the computer architecture that the compiler is targeting and the data types used inside the loop. To provide this guarantee, the programmer must preserve the original data dependencies in the loop using the *safelen* clause and remove potential data dependencies that may hinder SIMD execution using the *private*, *lastprivate*, *reduction*, or *linear* clauses. This is important when backward-carried loop dependencies are present.

```
#pragma omp simd safelen(4)
float A[SIZE], B[SIZE];
for (int i = 4; i < SIZE; i++){
    A[i] = A[i - 4] + B[i];
}
```

Normally, the compiler should be able to detect this backward-carried loop dependency. However, the OpenMP SIMD construct guarantees that the loop can be executed using partitioned chunks of SIMD execution. If the compiler is targeting a computer architecture that supports the AVX instruction set, then according to Table 2, this loop will execute using chunks of eight logical loop iterations. This is an issue because the loop cannot execute more than four logical loop iterations concurrently without violating the backward-carried loop dependency. To preserve this dependency, the *safelen* clause is required to limit the number of concurrent loop iterations. The *safelen* clause indicates that more than four iterations of the loop cannot be executed without violating a data dependency.

Data sharing clauses can be used to break false data dependencies that may hinder SIMD execution. The private clause guarantees that *x* is private to each SIMD chunk of the loop. This indicates that the values in *x* will not cross between the partitioned chunks of the loop. Otherwise, the compiler may think that the variable x contains a write-write or write-read conflict. The private clause is used to remove this false dependency. The *linear* and *reduction* clauses both provide privatization as well.

```
float A[SIZE], B[SIZE], x;
#pragma omp simd private(x)
for (int i = 0; i < SIZE - 1; i++){
    x = A[i];
    B[i] = foo(x*A[i+1]);
```

```
}
```

4.2 Vector Length and Alignment

The vector length indicates how many logical iterations of the loop will be handled in parallel. This depends on the size of the vector register selected by the compiler and the size of the type of data stored inside it. For x86 processors, the vector lengths supported by the hardware for each data type and vector register are given in Table 2. When using OpenMP, the vector length is determined by the compiler's target architecture for SIMD loops and the *simdlen* clause for SIMD-enabled functions. The compiler can simulate larger vector lengths than the computer architecture supports by combining several vector registers into a single logical vector register. In some cases, using a larger vector length can be beneficial due to improved instruction-level parallelism and amortization of various loop overheads. For some other cases, using a larger vector length can be detrimental if the limited number of available vector registers is exhausted.

Memory alignment is important for SIMD applications; this is discussed in Section 2.2. The compiler is unable to determine the alignment properties of data that is linked from other files or is dynamically allocated. Additionally, the compiler by itself cannot assume that function arguments will be aligned. OpenMP provides the *aligned* clause so the programmer can assert memory alignment properties. The ideal alignment value for the *aligned* clause is determined by the size of the vector register used. These values listed in Table 3 provide the ideal alignment for the x86 vector registers.

```
float X[SIZE], Y[SIZE], A;
#pragma omp simd aligned(X : 32, Y : 32)
for (int i = 0; i < SIZE; i++){
    X[i] = A*X[i] + Y[i];
}
```

4.3 OpenMP SIMD Functions

OpenMP allows users to create SIMD-enabled functions that can be called from SIMD regions. These functions allow for *simdlen* partitioned chunks of the logical loop iteration space to be executed in parallel. This is a large benefit for functions that could not be inlined by the compiler. SIMD-enabled functions allow the programmer to create SIMD functions that can be linked from independent vector libraries. However, effectively vectorizing functions with OpenMP can be difficult. These examples will show how to correctly create SIMD-enabled functions with OpenMP SIMD.

```
float rsqrtf(float A){
    return 1.0f / sqrtf(A);
}

double rsqrt_mul(float *A, float *B, int i){
    return B[i]*rsqrtf(A[i]);
}
```

```
int main(){
    float A[SIZE], B[SIZE];
    double C[SIZE];

    #pragma omp simd
    for (int i = 0; i < SIZE; i++){
        C[i] = rsqrt_mul(A, B, i);
    }
}
```

For this loop, the use of the OpenMP SIMD construct causes the loop to be partitioned into chunks of SIMD execution. However, if the function calls to *rsqrt_mul* and *rsqrtf* cannot be inlined by the compiler, such as if they were stored in a separate library or linked from another file, the loop cannot execute the function calls in parallel. The compiler provides the SIMD execution by calling the scalar function multiple times according to the size of the partitioned chunk. Then, each of these returned values are packed into a vector register to preserve the SIMD execution. This method will typically perform slower than a scalar version of the loop without the OpenMP SIMD construct. To solve this problem, the OpenMP declare construct can be used to create SIMD-enabled versions of these functions to call inside of a SIMD region.

```
#pragma omp declare simd
float rsqrtf(float A){
    return 1.0f / sqrtf(A);
}

#pragma omp declare simd uniform(A, B) linear(i : 1)
double rsqrt_mul(float *A, float *B, int i){
    return B[i]*rsqrtf(A[i]);
}

int main(){
    float A[SIZE], B[SIZE];
    double C[SIZE];

    #pragma omp simd
    for (int i = 0; i < SIZE; i++){
        C[i] = rsqrt_mul(A, B, i);
    }
}
```

The OpenMP declare construct allows for SIMD-enabled functions to be created for both functions. The use of the *uniform* clause indicates that the values of *A* and *B* will not change between each SIMD execution of the function. This is implemented by replacing the function argument to the vectorized function with a parameter passed in a general purpose register rather than a vector register that is shared between each SIMD lane of the function. The *linear* clause indicates that the value *i* is increasing by 1 between each concurrent iteration of the loop. This also allows the compiler to replace its function argument with a general purpose register. The combination of these two clauses allows the compiler to unroll the *rsqrt_mul* and perform loads from memory with linear strides. If these clauses were not present, the function would instead perform a gathering operation from a set of indices and pointers passed as arguments to the function using a vector register.

The SIMD-enabled functions created with the OpenMP declare construct each have their own arguments and qualities. The *rsqrt_mul* function has arguments (uniform, uniform, linear:1) and the *rsqrtf* function has arguments (vector). Additionally, each function has its own *simdlen*. For x86 processors, the default target architecture is SSE2. So, according to Table 2, the *rsqrt_mul* function has a default *simdlen* of two while the *rsqrtf* function has a default *simdlen* of four because these functions return doubles and floats respectively. This conflicting *simdlen* is a problem that prevents these functions from being vectorized. When the *rsqrt_mul* function is called, it operates on two arguments concurrently. However, the vectorized *rsqrtf* function requires four arguments to be called. This conflict means that the *rsqrt_mul* function cannot provide enough input to the *rsqrtf* function for it to be vectorized. In order for a SIMD-enabled function to call another SIMD-enabled function the calling function must have a *simdlen* that is a multiple of the called function's *simdlen*. This incompatibility can be solved by the *simdlen* clause. Declaring the *rsqrt_mul* function with a *simdlen* of four causes the compiler to apply double-pumping and treat two vector registers as a single logical vector register.

```
#pragma omp declare simd simdlen(4)
float rsqrtf(float A){
    return 1.0f / sqrtf(A);
}

#pragma omp declare simd uniform(A,B) linear(i:1) simdlen(4)
double rsqrt_mul(float *A, float *B, int i){
    return B[i]*rsqrtf(A[i]);
}

int main(){
    float A[SIZE], B[SIZE];
    double C[SIZE];

    #pragma omp simd
    for (int i = 0; i < SIZE; i++){
        C[i] = rsqrt_mul(A, B, i);
    }
}
```

This code allows the loop to be successfully vectorized. However, improvements can still be made to this function. Currently, the default architecture of SIMD-enabled functions on x86 processors is SSE2. This default currently cannot be changed by compiler flags if the function is not inlined. If the Intel compiler is used on an x86 processor, the *processor* clause can be used to specify a target architecture for the code generated inside the SIMD-enabled functions. If this loop is to target the AVX2 instruction set, then according to Table 5, the string used with the processor clause will be *core_4th_gen_avx*. If the AVX2 instruction set is used, then the *simdlen* should be changed to eight to fully utilize the width of the YMM registers. Additionally, if the *rsqrt_mul* function is never called within a conditional branch the *notinbranch* clause can safely be used. This will reduce the size of the generated code. These functions can again be further be improved if the pointers *A* and *B* are guaranteed to be aligned. In this case the *aligned* clause can be used to provide the compiler with the alignment information.

```
#pragma omp declare simd simdlen(8) notinbranch \
    processor(core_4th_gen_avx)
```

```
float rsqrtf(float A){
    return 1.0f / sqrtf(A);
}

#pragma omp declare simd uniform(A,B) linear(i:1) simdlen(8) \
    aligned(A:32,B:32) notinbranch processor(core_4th_gen_avx)
double rsqrt_mul(float *A, float *B, int i){
    return B[i]*rsqrtf(A[i]);
}

int main(){
    float A[SIZE], B[SIZE];
    double C[SIZE];

    #pragma omp simd
    for (int i = 0; i < SIZE; i++){
        C[i] = rsqrt_mul(A, B, i);
    }
}
```

4.4 Memory Access Collapsing

The OpenMP SIMD construct works by partitioning the logical iterations of the loop into SIMD chunks. This partitioning requires a single logical iteration space to be valid. Because of this, the OpenMP SIMD construct can normally only be correctly applied to the innermost loop in a set of nested loops. The *collapse* clause allows for the OpenMP SIMD construct to be applied to multiple loops by collapsing the set of nested loops into a single logical iteration space.

It is important to understand how the *collapse* clause works to use it correctly. When used incorrectly, the collapse clause can cause some nested loops to execute several times slower than if the OpenMP SIMD construct was applied to the innermost loop alone. This can be seen by observing the process the compiler uses to collapse a loop. For example, consider a simple set of two nested loops.

```
float A[4][4];
#pragma omp simd collapse(2)
for (int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
        A[i][j] = A[i][j] + 1;
    }
}
```

This loop will be collapsed by creating a global iteration space from the total number of logical iterations made by the nested loops. Then the values of i and j will need to be calculated using the new global iteration space. This set of nested loops will be transformed into the following single loop by the compiler.

```
float A[4][4];
#pragma omp simd
for (int k = 0; k < 16; k++){
```

```
    int i = k / 4;
    int j = k % 4;
    A[i][j] = A[i][j] + 1;
}
```

This collapsed loop will perform poorly if it is vectorized. This is because the division or modulus operations must be simulated with software and are fundamentally slow. Additionally, the memory access to *A* is not linear with respect to the collapsed loop so the memory must be gathered and then scattered. However, this transformation may be profitable for SIMD directives that target SMT multithreading (e.g. for accelerators like GPUs). In order to make loop collapsing profitable for vectorization, the data accesses must be collapsed as well as the iteration space.

Memory access collapsing can be done by observing the layout of the original loop. In the original loop, the inner loop performs four iterations. If the loop is vectorized with a vector length that is a multiple of four, then the inner loop can be performed in a single iteration of the loop. This removes the need for the modulus operation on *j* because the remainder will always be zero after performing four iterations of the loop simultaneously. Now, the memory access is linear so the division can be removed. After performing memory access collapsing, the original set of nested loops will be transformed into the following single loop by the compiler.

```
float A[4][4];
#pragma omp simd
for (int k = 0; k < 16; k++){
// Conceptually equivalent to A[0][k] = A[0][k] + 1;
    A[k] = A[k] + 1;
}
```

After collapsing the data accesses, there is no need to re-calculate the *i* and *j* iterators with expensive division and modulus operations. The resulting code also generates contiguous memory accesses for *A*. In this situation, the collapsed loop will perform well if it is vectorized. The compiler must support both memory access collapsing and loop iteration collapsing for the collapse clause to be profitable.

4.5 Scalar Execution Inside SIMD Regions

SIMD parallelism requires that multiple pieces be modifiesd with a single instruction. However, this is not possible in some cases. The compiler provides effective SIMD execution by individually performing each operation using scalar instructions. This situation can occur when a certain operation has no SIMD equivalent or when the result would be incorrect under SIMD execution. While the former can be handled by the compiler, the latter under certain conditions must be specified by the user. The *ordered* clause can be used to specify scalar execution for a block of code.

The *ordered* clause is beneficial if a certain condition prevents the code from correctly executing using SIMD instructions. The histogram problem is a good example of this. A gather instruction can be used to compute the result of a histogram operation if the indices are distinct. If the indices are not distinct, another method must be used. The *ordered* clause can be used to describe this.

```
#pragma omp simd
for (int i = 0; i < SIZE; i++){
    if (has_conflict)
#pragma omp ordered simd
        {
        hist[A[i]]++;
        }
    else
        hist[A[i]]++;
}
```

5. General SIMD Programming Guidelines

Efficiently exploiting SIMD parallelism in code can be more difficult than other forms of parallelism. This is because SIMD parallelism must use a restricted set of hardware registers and SIMD instructions. Because of these restrictions, understanding how the SIMD model works is essential to maximize the potential performance of the SIMD model. Additionally, it is common for a vectorized program to run *slower* than a scalar program. This drop in performance is usually caused by attempting to vectorize operations that have no SIMD instruction equivalent, or performing operations that are relatively inexpensive when done with scalar instructions but quite slow when SIMD instructions are used. This section provides several general programming guidelines for programmers to obtain high performance SIMD programs while relying on auto-vectorization.

5.1 Data Size and Conversion

The choice of data size has a large impact on the performance of SIMD applications. Vector registers can only hold a fixed amount of data, so the size of the data packed inside the vector register determines how much can be processed in parallel. Additionally, the processor must be able to handle each different data type. Because of this, some loops may not be vectorized or may be vectorized poorly because of the data they operate on. For example, the x86 architecture currently does not support multiplication operations between packed bytes.

For high performance SIMD applications, the smaller data types should be used whenever possible. Referring to Table 2, If 16-bit values are used instead of 32-bit values, then twice as much data can be handled in parallel. So, a vectorized loop that processes shorts will perform roughly twice as fast as a loop that processes ints. The same speedup is achieved when floats are used over doubles. So, for integers, the programmer should use the smallest data type that can hold the largest expected value. For floating point values, the programmer should decide how much floating point precision is necessary to produce a correct result.*

Data size is important when operations are performed between two vector registers. Although smaller data types allow more data to be handled in parallel, different vector lengths should not be mixed if at all possible. If two different vector lengths are used then the amount of data that can be handled in parallel is

*Long doubles cannot be vectorized.

limited by the smallest vector length. Additionally, there will be considerable overhead involved with converting from one vector length to another. This can be shown with the following loop.

```
float X[SIZE]; double Y[SIZE];
#pragma omp simd aligned(X : 32, Y : 32)
for(int i = 0; i < SIZE; i++){
    X[i] = Y[i]*X[i] + 1.0f;
}
```

This loop mixes two different vector lengths. The vector length when using doubles is half the vector length when using floats. The vector lengths must match before this loop can be executed using SIMD instructions. Two vector registers must be combined into a single logical vector register so that the vector length of the doubles is compatible with the vector length required by loop. Then the packed floats must be converted to doubles by placing each half of the vector register into a separate vector register. Finally, the multiplication is performed and then the packed doubles are converted back into floats and stored. This is dramatically slower than if only doubles or floats were used inside this loop.

The programmer should be careful to not call library functions with conflicting vector lengths. If the programmer calls a library function that returns conflicting data types, there will be a severe performance penalty.

```
float X[SIZE];
#pragma omp simd aligned(X : 16)
for (int i = 0; i < SIZE; i++){
    X[i] = pow(X[i], -1.5);
}
```

Some compilers can vectorize this loop by using a vectorized math library. However, the *pow* function takes doubles as its arguments and returns a double. This requires a costly conversion operation like the one detailed in the previous loop. This can be avoided if the call to the *pow* function is replaced with the *powf* function. The *powf* function uses floats as its arguments and returns floats. This removes the need for a slow conversion operation.

Finally, the programmer should be careful to specify whether or not a floating point constant is a double or a float. Constants given in the form “x.xx” are usually assumed the be doubles. If this constant is multiplied by a float it may also require a costly conversion operation. Floating point constants should always be specified by using the form “x.xxf” to avoid this.

5.2 Memory Access

The layout of the data is often what determines whether or not vectorization is profitable. Data movement is detailed in Section 2.1.1. Memory should ideally be completely continuous for SIMD applications. This is not always possible, but, in some cases, the programmer can alter the memory layout to be more continuous. This can be seem with the following example that uses a struct to represent a three-dimensional vector.

```
struct {float x, y, z;} v[SIZE];
#pragma omp simd aligned(v : 16)
for (int i = 0; i < SIZE; i++){
    int n=sqrtf(v[i].x*v[i].x+v[i].y*v[i].y+v[i].z*v[i].z);
    v[i].x /= n;
    v[i].y /= n;
    v[i].z /= n;
}
```

This loop represents the vector as an *array of structs*. But in this example, the memory in the array V is not contiguous. When the struct is placed in memory, the data will be arranged in this pattern, $\{x_0, y_0, z_0, x_1, y_1, z_1, \ldots\}$. The computation can easily be vectorized, but moving the data into the vector register will require several data movement instructions that will slow down this loop. It would be much faster if the data was continuously in this pattern, $\{\{x_0, x_1, \ldots\}, \{y_0, y_1, \ldots\}, \{z_0, z_1, \ldots\}\}$. Arranging data in this way is called a *struct of arrays* and will greatly reduce the amount of time spent moving data into the vector registers.

```
struct {float x[SIZE], y[SIZE], z[SIZE];} v;
#pragma omp simd aligned(v.x : 16, v.y : 16, v.z : 16)
for (int i = 0; i < size; i++){
    int n=sqrtf(v.x[i]*v.x[i]+v.y[u]*v.y[i]+v.z[i]*v.z[i]);
    v.x[i] /= n;
    v.y[i] /= n;
    v.z[i] /= n;
}
```

A struct of arrays has the downside that it may fragment the cache. To keep the data more local, a combination of a struct of arrays and an array of structs can be used where the struct of arrays only contains enough elements to completely fill a vector register. Then this struct is used as an array of structs.

Multidimensional arrays can be accessed with a linear stride. An example of this would be a loop that iterates over the columns of a two dimensional array. In this case, the memory accesses can be effectively disjoint if the linear stride is greater than the vector length. However, data that has already been loaded can sometimes be used later. For example, if memory from a column is read into a vector register, data from the next column is also in the vector register. Unfortunately, this is difficult to take advantage of when relying on auto-vectorization.

Memory can also sometimes be accessed at completely disjoint locations. This is common in applications involving look-up tables or histograms. When the accesses are disjoint, the memory must be gathered or scattered. These operations are extremely costly on computer architectures that do not support a gathering or scattering instruction. If gathering and scattering instructions are supported, then loops involving look-up tables can sometimes be vectorized. However, these instructions are still several times slower than a continuous memory access. Because of this, look-up tables should not be used unless the time saved on computation is greater than the time wasted on gathering the memory.

5.3 Integer Multiplication and Division by Constants

Packed integer multiplication can be slow. For constants, the speed can be improved by performing a shift-and-add multiplication. This is done by factoring the constant multiplicand into a sum of powers of two. Thus, the multiplication $10x$ is equivalent to $(8 + 2)x$. This can be accomplished by shifting x to the left three times and adding that to x shifted left once. This approach is typically faster for constants that generate fewer than five factors. Some compilers, such as the GCC compiler, will automatically perform shift-and-add multiplication. Others can see an improvement if this is done explicitly by the programmer. The table shows the improvement between multiplying by 10 and doing shift-and-add. This method provided a $1.66x$ speedup on the Intel compiler, while the GCC compiler used this method automatically.

```
short X[SIZE];
#pragma omp simd aligned(X : 16)
for (int i = 0; i < SIZE; i++){
    X[i] = (X[i] << 3) + (X[i] << 1);
}
```

The x86 architecture does not support SIMD instructions for packed integer division or modulus. However, this operation can be simulated with multiplication and bit-shifting operations. This follows from the idea that for floating point values, division is equivalent to multiplication by the reciprocal. To do this with integers, the reciprocal will be scaled by 2^n and then the product will be shifted to the right n times. There are several algorithms to find a suitable n value covered by other authors[2].

Some compilers may convert integer division by a constant with a multiplication and shift in this way if it is known at compile time. The Intel compiler has a set of functions that can calculate the value of n at runtime if the divisor is only known at runtime. However, this is still an expensive process, so integer division and modulus should be completely avoided whenever possible.

5.4 Conditional Statements

Conditional statements can be vectorized by calculating the values unconditionally and then using a bit-mask or separate mask register to choose between them. The downside to this is that the entire loop must be calculated unconditionally. If a loop contains a relatively costly calculation that is rarely required, the result of this calculation will be calculated each iteration of the SIMD loop only to be discarded at the end. This can sometimes cause a vectorized loop to perform slower than a scalar loop. In some cases this can be avoided by jumping past a code segment if its associated mask is completely empty. Some compilers are capable of doing this.

Generally, branch-free algorithms should be favored for SIMD applications. This removes the need for SIMD comparisons to create bit-masks. Branch-free algorithms will also typically remove the problem of conditional statements with disproportionate costs that can slow down vectorized loops. Additionally, branch-free algorithms typically use operations that are easily performed using SIMD instructions, such as bit-shifting or other bit-logic instructions.

6. HACCmk

HACCmk is a compiler benchmark whose performance relies on how well the compiler can vectorize the critical loop. The loop contains a function call, reduction, and a conditional assignment.

```
void Step10_orig(int count1, float xxi, float yyi, float zzi,
        float fsrrmax2, float mp_rsm2, float *xx1, float *yy1,
        float *zz1, float *mass1, float *dxi, float *dyi, float *dzi ){

    const float ma1, ma2, ma3, ma4, ma5;
    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;

    for (int j = 0; j < count1; j++){
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;

        r2 = dxc * dxc + dyc * dyc + dzc * dzc;

        m = ( r2 < fsrrmax2 ) ? mass1[j] : 0.0f;

        f =  pow(r2 + mp_rsm2, -1.5) - (ma0 + r2*
             (ma1 + r2*(ma2 + r2*(ma3 + r2*(ma4 +r2*ma5)))));

        f = ( r2 > 0.0f ) ? m*f : 0.0f;

        xi = xi + f * dxc;
        yi = yi + f * dyc;
        zi = zi + f * dzc;
    }
}
```

A critical mistake was that the ”pow“ function was used instead of ”powf“ This results in the code running about *three times* slower using the Intel compiler. There are several reason for this dramatic slowdown. The Intel compiler recognized that the exponent could easily be calculated as $\sqrt{x^3}^{-1}$ so it replaced the function call with this calculation. Because the pow function was used, there was a costly conversion performed to convert the floats to doubles and then back to floats. Additionally, there is a packed inverse square root function that only applies to floats. This square root is *faster* than a regular square root and division because it is an approximation of the square root using the Newton-Raphson method. Finally, the use of doubles confused the compiler and caused it to turn the conditional assignment into scalar code that then packed the results back into the vector register at the end.

While the Intel compiler had no problems vectorizing this loop once the function call was corrected, the GCC and Clang 3.8 compilers could not vectorize it. It seems that both Clang and GCC lack a set of vectorized math functions. There are hardware instructions that calculate square roots. So the “powf” function should be replaced with the ’sqrtf’ instruction. However, the GCC and Clang compilers still did not replace the ’pow’ function call with these instructions like the Intel compiler did. We would like OpenMP 4.5 compilers to provide vectorized versions of “math.h” functions.

When the function calls were removed, the two compilers still struggled with the conditional assignments in the code. Although the GCC compiler is capable of vectorizing some conditional loops, it could not vectorize them in this loop; either the loop is too complex, or the compiler is confused because the variables being compared are not arrays. It would be very useful if OpenMP supported conditional assignments to provide this functionality between all compilers that support OpenMP.

Even after the function calls and the conditional statements were removed, this loop was still not vectorized. Most likely, the presence of multiple reductions confused the compilers. This, however, could be solved with OpenMP 4.5. Using the reduction clause and specifying the variables causes the loop to finally be vectorized by GCC. Clang still had issues and only vectorized once the loop was nothing but the first three and last three instructions.

Potentially, the function could be vectorized by writing a vectorized square root function using SIMD declare, but the SIMD declare functionality seems to be troublesome when it is not applied to arrays. A vectorized version could easily be written with a single line of assembly, but the compiler cannot vectorize around assembly code.

7. Conclusion

The performance gained from SIMD execution capabilities is becoming more integral to the overall performance of modern processors. SIMD instructions allow for certain sections of code to be executed many times faster than a typical scalar implementation. This can be used to considerably increase the performance yield from traditional multi-threaded parallel applications without increasing the number of processors.

Effectively making use of SIMD instructions is difficult because of limited static compiler analysis, architecture-specific limitations, and the restrictions of the SIMD execution model itself. Many computer architectures implement SIMD instructions in fundamentally different ways. Because of this, any program explicitly vectorized through the use of intrinsic functions or assembly code could not run on other computer architectures or SIMD models, such as an SMT implementation targeting a GPU. Because of this, many programs rely upon the compiler's auto-vectorization to vectorize the code on many different architectures.

Auto-vectorization allows much of the burden of vectorizing a program to be left to the compiler. However, the SIMD architecture requires several constraints that the compiler cannot rectify through static-analysis alone such as data alignment, data dependencies, aliasing, irregular memory accesses, and generally the fixed-length nature of vector registers. Also, the quality of auto-vectorization varies across compilers. Because of these restrictions, the OpenMP standard API for exploiting parallelism from shared memory processors was expanded to provide powerful and portable directives to assist with auto-vectorization.

OpenMP provides directives to improve the capabilities of the compiler's auto-vectorization pass by providing it with information that cannot be determined through compile-time static-analysis. This allows the programmer to effectively vectorize previously problematic sections of code and have it run efficiently on several computer architectures and accelerators. However, because of the nature of SIMD instructions, an understanding of the SIMD model of execution is essential in order to create code that can be easily and efficiently be vectorized by the compiler. The OpenMP SIMD parallelism is not limited to architectures

with vectorization units as it can be simulated using threads (e.g. GPU SMT threads) on different architectures as long as the SIMD parallel semantics are kept.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[2] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, 1994.

[3] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. *Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures*, pages 59–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] D. J. Kuck, Y. Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. Comput.*, 21(12):1293–1310, December 1972.

[5] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, February 1974.

[6] Leslie Lamport. *The coordinate method for the parallel execution of iterative loops*. SRI International, 1981.

[7] Yoichi Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1971. AAI7121189.

[8] OpenMP Website. http://www.openmp.org.

[9] Hideki Saito, Serge Preis, Nikolay Panchenko, and Xinmin Tian. *Reducing the Functionality Gap Between Auto-Vectorization and Explicit Vectorization*, pages 173–186. Springer International Publishing, Cham, 2016.

[10] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko. Practical simd vectorization techniques for intel xeon phi coprocessors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1149–1158, May 2013.

[11] Xinmin Tian and Bronis R de Supinski. Explicit vector programming with openmp 4.0 simd extensions. *HPCTODAY*, 2014.

[12] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982. AAI8303027.

[13] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM, New York, NY, USA, 1991.