Using Advanced Vector Extensions AVX-512 for MPI Reductions

Dong Zhong
The University of Tennessee
Knoxville, TN, USA

Qinglei Cao
The University of Tennessee
Knoxville, TN, USA

George Bosilca
The University of Tennessee
Knoxville, TN, USA

Jack Dongarra
The University of Tennessee
Knoxville, TN, USA

ABSTRACT

As the scale of high-performance computing (HPC) systems continues to grow, researchers are devoted themselves to explore increasing levels of parallelism to achieve optimal performance. The modern CPU's design, including its features of hierarchical memory and SIMD/vectorization capability, governs algorithms' efficiency. The recent introduction of wide vector instruction set extensions (AVX and SVE) motivated vectorization to become of critical importance to increase efficiency and close the gap to peak performance.

In this paper, we propose an implementation of predefined MPI reduction operations utilizing AVX, AVX2 and AVX-512 intrinsics to provide vector-based reduction operation and to improve the time-to-solution of these predefined MPI reduction operations. With these optimizations, we achieve higher efficiency for local computations, which directly benefit the overall cost of collective reductions. The evaluation of the resulting software stack under different scenarios demonstrates that the solution is at the same time generic and efficient. Experiments are conducted on an Intel Xeon Gold cluster, which shows our AVX-512 optimized reduction operations achieve 10X performance benefits than Open MPI default for MPI local reduction.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**; *Distributed programming languages*; • **Computer systems organization → Very long instruction word**; **Single instruction, multiple data**; *Heterogeneous (hybrid) systems*; • **Software and its engineering →** *Software libraries and repositories.*

KEYWORDS

Long vector extension, Vector operation, Intel AVX2/AVX-512, Instruction level parallelism, Single instruction multiple data, MPI reduction operation

EuroMPI/USA '20, September 21–24, 2020, Austin, TX, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8880-1/20/09…$15.00
https://doi.org/10.1145/3416315.3416316

ACM Reference Format:
Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. 2020. Using Advanced Vector Extensions AVX-512 for MPI Reductions. In *27th European MPI Users' Group Meeting (EuroMPI/USA '20), September 21–24, 2020, Austin, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3416315.3416316

1 INTRODUCTION

The need to satisfy the scientific computing community's increasing computational demands drives to larger HPC systems with more complex architectures, which provides more opportunities to enhance various levels of parallelism. Instruction-level (ILP) and thread-level parallelism (TLP) has been extensively studied, but data-level parallelism (DLP) is usually underutilized in CPUs, despite its vast potential. While ILP importance subsides DLP becomes a critical factor in improving the efficiency of microprocessors [9, 12, 29, 32, 38]. The most widespread vector implementation is based on single-instruction multiple-data (SIMD) extensions. Vector architectures are designed to improve DLP by processing multiple input data simultaneously with a single instruction, usually applied to vector registers. SIMD instructions have been gradually included in microprocessors, with each new generation providing more sophisticated, powerful and flexible instructions. The higher investment in SIMD resources per core makes extracting the full computational power of these vector units more significant than ever.

A large body of literature has been focusing on employing DLP via vector execution and code vectorization [8, 22, 28], and HPC, with its ever-growing demand for computing capabilities, has been quick to embrace vector processors and harness this additional compute power. Vectorization as an essential factor of processors' capability to apply a single instruction on multiple data, continuously improves from one CPU generation to the next, by using larger vector registers, gather/scatter capabilities and much more. Compared to traditional scalar processors, extension vector processors support SIMD and more powerful instructions operating on vectors with multiples elements and can generate orders of magnitude faster memory accesses and data computations. Over the last decade, the difference between a scalar code and its vectorized equivalent increased from a factor of 4 with SSE, up to a factor of 16 with AVX-512 [35], highlighting the importance of employing vectorized code whenever possible. The conversion of a scalar code into a vectorized equivalent can be rather straightforward for many classes of algorithms and computational kernels, as it can be done transparently by a compiler with auto-vectorization. For more complex codes, the compiler can or might provide a baseline but developers are also encouraged to provide optimized versions using widely available compilers intrinsics.

There are efforts to keep improving the vector processors by increasing the vector length and adding new vector instructions. As an example, Intel's first version of vectorized instruction set, MMX was quickly superseded by more advanced vector integer SSE and AVX instructions [17, 19, 27], then expanded to Haswell instructions as 256 bits (AVX2), and then with the arrival of the Knights Landing processor [35] the more advanced AVX-512 [18] was introduced supporting 512-bit wide SIMD registers (ZMM0-ZMM31) as in Figure 1. The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers, and the lower 128-bit are aliased to the respective 128-bit XMM registers; The AVX-512 features and instructions provide a significant advantage to 512-bit SIMD support. It offers the highest degree of compiler support by including a unique level of richness in designing the instructions. Compared to previous architecture and products, it leverages longer and more powerful registers capable of packing eight double-precision, or sixteen single-precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within a 512-bit vector. It also enables processing twice the amount of data elements than Intel AVX2 and four times than SSE with a single instruction. Furthermore, AVX-512 supports more features such as operations on packed floating-point data or packed integer data, new operations, additional gather/scatter support, high-speed math instructions, and the ability to have optional capabilities beyond the basic instruction set.

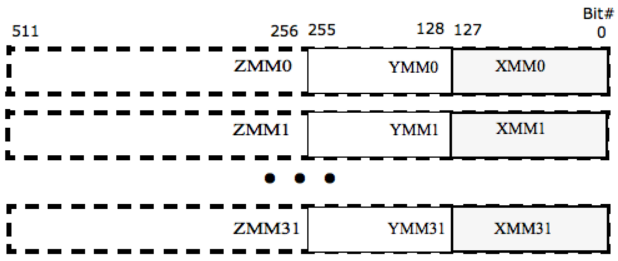

Figure 1: AVX512-Bit Wide Vectors and SIMD Register Set

AVX-512 not only takes advantage of using long vectors but also enables powerful high vectorization features that can achieve significant speedup. Those features include but not limited to:

(1) providing a valuable set of horizontal reduction operations which apply to more types of reducible loop carried dependencies including both logical, integer and floating-point of high-speed math reductions;
(2) and permitting vectorization of loops with more complex loop carried dependencies and more complex control flow.

Similarly, Arm announced the new Armv8 architecture embracing SVE- a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture [2, 13]. Unlike other SIMD architectures, SVE does not define the size of the vector registers. Instead, it provides a range of different values that permit vector code to automatically adapt to the current vector length at runtime with the feature of *Vector Length Agnostic* (VLA) programming [3, 5]. Vector length constrains in the range from a minimum of 128 bits up to a maximum of 2048 bits in increments of 128 bits.

At the other end of the programming spectrum, Message Passing Interface (MPI) [14] is a popular, efficient and portable parallel programming paradigm for distributed memory systems widely used in scientific applications. The MPI standard provides an entire set of communication primitives, between pairs of processes or between entire groups of processes, allowing applications to completely tailor its' use to their needs. Therefore, there is no critical subset of MPI capability in particular, all MPI aspects are critical to some computation domains. However, there is evidence that optimized support for two-sided communications and collective communications, will benefit a large number of parallel applications. As an example, machine learning applications running on distributed systems, critically depend on the performance of an MPI_Allreduce, a reduction operation, for extensive data sets to synchronize updating the weights matrix.

Computation-oriented collective operations such as MPI_Allreduce and MPI_Reduce perform reductions on data along with the communications performed by collectives. These collectives typically encompass a memory-bound operation, which forces the computation to become the main bottleneck and limit the overall performance of the collective implementation. However, the existence of advanced architecture technologies introduced with wide vector extension and specialized arithmetic operations, calls for MPI libraries to provide support for such extensions, providing specialized functions capable to extract most of the computational power of the processor and unconditionally deliver it to the application.

Unlike more traditional HPC applications that embraced MPI long ago, machine learning and data science in general, were more reticent. However, a new trend has arisen lately, certainly in relation to the growth of the size of the problems, toward a wider use of MPI for the distributed training.

As mentioned before, the most expensive step in the learning process, the reduction of the gradients, is entirely dependent on the performance of MPI_Allreduce with a large amount of data (basically all the weights on the layer). Such reduction operations in machine learning applications are commonly seen in synchronous parameter updates of the distributed Stochastic Gradient Descent (SGD) optimization [6], which is used extensively in, for example, neural networks, linear regressions and logistic regressions. Usually, this kind of reduction has two aspects: 1) the number of reduction operations is significant. 2) the data size used by each reduction operation is extremely large (with an extensive training model, the data could be in the hundreds of megabytes). Li's [23] work explores the performance of allreduce algorithms and uses task-based frameworks to improve their performance. Specifically talking about AlexNet on ImageNet [21], it points out that each step needs to perform a weights reduction with an estimated size of 200MB for extensive model training. Similarly, [30] illustrates that with SparkNet, updating the weights of AlexNet, a single reduce operation takes almost 20 seconds, even on five nodes. While it's relatively simple to scale the number of execution nodes to the thousands, the biggest bottleneck remains the allreduce of the gradient values at each step. The size of this reduction is equivalent to the model size itself, and it is not reduced when more nodes are used. When scaling to large numbers of nodes, the full parameter set, commonly hundreds of megabytes, must be summed globally every few microseconds. We can see that, in such cases, reduction operation dominates the overall time-to-solution in distributed neural network training, highlighting the need for a more efficient reduction implementation.

Thus, for many applications it will be crucial to provide a highly optimized version of MPI_Allreduce, and this requires to address the challenge of improving the performance of the predefined MPI operations. We tackle the above challenges and provide designs and implementations for reduction operations, which are most commonly used by the computation collectives - MPI_Reduce, MPI_Allreduce and MPI_Reduce_Local. We propose extensions to multiple MPI reduction methods to fully take advantage of the AVX-512 capabilities such as vector product to efficiently perform these operations.

This paper makes the following contributions:

(1) We investigate and utilize AVX-512 arithmetic instructions/intrinsics to optimize and speed up a variety type MPI reduction operations.
(2) perform experiments using the new reduction operations in the scope of Open MPI on a cluster supporting the Intel AVX-512 extensions. Different types of experiments are conducted with MPI application, performance evaluation tool and deep learning benchmark. Furthermore, our implementation provides useful insight and guidelines on how vector ISA can be used in high-performance computing platforms and software.

The rest of this paper is organized as follows. Section 2 presents related studies on taking advantage of AVX-512 and SVE for specific mathematics applications, together with a survey about optimizations of MPI to take advantage of novel hardware. Section 3 describes the implementation details of our optimized reduction methods in the scope of Open MPI using AVX-512 intrinsics and instructions. Section 4 uses a performance too to evaluate performance by different kinds of instruction counts. Section 5 describes the performance difference between Open MPI and AVX-512 optimized Open MPI and provides a distinct insight on how the new vector instructions can benefit MPI. Section 6 illustrates the performance benefits of our optimized reduction operation in Open MPI using a deep learning application.

2 RELATED WORK

Different techniques can be roughly classified according to the level at which the hardware supports parallelism with multi-core and multi-processor computers having multiple processing elements within a single machine. Different level of parallelization, including bit-level, instruction-level, data-level, and task parallelism, are studied. In this section, we survey related work on techniques taking advantage of advanced hardware or architectures, which mainly focuses on data-level parallelization. Novel processors and hardware architectures from different vendors, such as Intel and Arm, come equipped with long vector extensions, and the usage of those new technologies in high-performance computing has been studied by multiple researchers with various programming models and applications.

2.1 Long vector extension

Lim [24] explored matrix-matrix multiplication based on blocked matrix multiplication improves data reuse. They used data prefetching, loop unrolling, and the Intel AVX-512 to optimize the blocked matrix multiplications, which achieved outstanding performance of GEMM with single and multiple cores. Kim [20] presented an optimal implementation of single-precision and double-precision general matrix-matrix multiplication (GEMM) routines based on an auto-tuning approach with the Intel AVX-512 intrinsic functions. The implementation significantly reduced the search space and derived optimal parameter sets, including the size of submatrices, prefetch distances, loop unrolling depth, and parallelization scheme. Bramas [7] introduced a novel quicksort algorithm with a new Bitonic sort and a new partition algorithm that has been designed for the AVX-512 instruction set, which showed superior performance on Intel SKL in all configurations against two standard reference libraries. A little closer to MPI, Dosanjh et al. [11] proposed and evaluated a novel message matching method Fuzzy-matching to improve the point to point communication performance in MPI with multithreading enabled. The proposed algorithm took advantage of the AVX vector operation to accelerate matches and demonstrated that the benefits of vector operation are not only restricted to computational intensive operations, but can positively impact MPI matching engines. They also presented an optimistic matching scheme that uses partial truth in matching elements to accelerate matches. Intel AVX is not the only ISA to propose vectorized extensions. Similar studies have been done using Arm's new scalable vector SVE. In this work [4], they leveraged the characteristics of SVE to implement and optimize stencil computations, ubiquitous in scientific computing which showed that SVE enabled the easy deployment of optimizations like loop unrolling, loop fusion, load trading or data reuse. Petrogalli's work [16] explored the usage of SVE vector multiple instructions to optimize matrix multiplication in machine learning algorithm. Zhong [41] used SVE load, gather and scatter instructions to optimize MPI datatype packing and unpacking in Open MPI. We can see those work focused on using new instructions to improve a specific application's performance or a specific mathematical algorithm. In our work, we study AVX-512 enabled features more comprehensively for all supported mathematical reduction functions and also provide a detailed analysis of the efficiency achievements of related intrinsics. Furthermore, we aim to accommodate the AVX reduction instructions support in MPI to provide vectorized computations for applications to use.

2.2 MPI reduction operation

Additionally, different techniques and efforts have been studied to optimize MPI reduction operations. Traff [37] proposed a simple implementation of MPI library internal functionality that enabled MPI reduction operations to be performed more efficiently with increasing sparsity of the input vectors. Also [10] analyzed the limitations of the compute oriented CUDA-Aware collectives and proposed alternative designs and schemes by combining the exploitation of the compute capability of GPU and their fast communication path using GPUDirect RDMA feature to alleviate these limitations efficiently. Luo [25] presented a collective communication framework called ADAPT in Open MPI based on an event-driven infrastructure. Through events and callbacks, ADAPT relaxed synchronization dependencies and maintained the minimal data dependencies. This approach provided more tolerance to system noise and also supported fine-grained, multi-level topology-aware collective operations which can exploit the parallelism of heterogeneous architectures. Hofmann [15] presented a pipeline algorithm for MPI Reduce that used a Run Length Encoding scheme to improve the global reduction of sparse floating-point data. Patarasuk's work [31] investigated implementations of the allreduce operation with large data sizes and derived a theoretical lower bound on the communication time of this operation and developed a bandwidth optimal allreduce algorithm on tree topologies. Shan [34] proposed to use idle threads on a many-core node in order to accelerate the local reduction computations, and also used data compression technique to compress sparse input data for reduction. Both approaches (threading and exploitation of sparsity) helped accelerate MPI reductions on large vectors when running on many-core supercomputers.

Most of those work focuses on improving the performance of communication either by relaxing dependencies or hiding the communication latency behind computation. And for the minority of those work that endeavors to strengthen the computation part, they usually have some requirements or limitations of data representation or need extra hardware such as GPUs. Our AVX-512 arithmetic reduction optimizations seek to be more general, and use the newly available vector extensions to provide a straightforward set of predefined MPI operations, with no limitation of data representation or operations. The implementation supports multiple ISA, covering most Intel processors versions either with legacy SSE and AVX or advanced AVX-512.

3 DESIGN AND IMPLEMENTATION

3.1 Intel Advanced Vector Extension

Intel Advanced Vector Extension 2 (Intel AVX2), is a significant improvement to Intel Architecture. It supports the vast majority of previous generations 128-bit SIMD float-point and integer instructions to operate on 256-bit YMM registers to support 256-bit operations. AVX2 also enhances a vibrant mix of broadcast, permute/variable-shift instructions to accelerate numerical computations. The 256-bit AVX2 instructions are supported by the Intel microarchitecture Haswell which implements 256-bit data path with low latency and high throughput. Besides, AVX2 provides enhanced functionalities for broadcast and permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

Moreover, Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions enrich significant supports compared to AVX2. It provides more powerful packing capabilities with longer vector length, such as encapsulating eight double-precision or sixteen single-precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within a vector. The longer vector enables processing of twice the number of data elements than that Intel AVX/Intel AVX2 can process with a single instruction and four times than that of SSE. On the other hand, it contributes to more distinguished performance for the most demanding computational tasks with more vectors(32 vector registers, each 512 bits wide, eight dedicated mask registers), enhanced high-speed math instructions, embedded rounding controls, and compact representation of large displacement value.

Furthermore, Intel AVX-512 instructions offer the highest degree of compiler support by including an unprecedented level of richness in the design of the instructions. Thus, it has better compatibility with Intel AVX that is stronger than prior transitions to new widths for SIMD operations. For SSE and AVX, programs will suffer from performance penalties once mix them. However, the mixing of AVX and Intel AVX-512 instructions is supported without penalty. AVX registers YMM0–YMM15 map into the Intel AVX-512 registers ZMM0–ZMM15, very much like SSE registers map into AVX registers. Therefore, in processors with Intel AVX-512 support, AVX and AVX2 instructions operate on the lower 128 or 256 bits of the first 16 ZMM registers.

3.2 Intrinsics

Intel intrinsics are built-in functions that provide access to the ISA functionality using C/C++ style coding instead of assembly language. Without Intel intrinsic was supported, users had to write assembly code directly to manipulate SIMD instructions arbitrarily. However, Intel has defined several sets of intrinsic functions that are implemented in the Intel Compiler. These types empower the programmer to directly choose the implementation of an algorithm while allowing the compiler to perform register allocation and instruction scheduling wherever possible. The intrinsics are portable among all Intel architecture-based processors supported by a compiler. The use of intrinsic allows developers to obtain performance close to the levels achievable and feasible with assembly. The cost of writing and maintaining programs with intrinsics is considerably less than writing assembly code. In summary, the intrinsic function allows SIMD instructions to be manipulated faster, more accurately, and more effectively than writing lower-level code. We describe the primary AVX-512 intrinsic functions that we are interested in our kernel:

(1) ***__m512i _mm512_loadu_si512 (void const* mem_addr)*** Load 512-bits of integer data from memory into a register. The mem_addr does not need to be aligned on any particular boundary. Generally, this instrinsic is converted into: ***vmovdqu32 zmm, m512***.
(2) ***__m512i _mm512_<op>_epi32 (__m512i a, __m512i b)*** Apply <op> between packed 32-bit integers in "a" and "b", and store the results in destination, here we use 32-bits integer as an example. Generally, this instrinsic is converted into: ***vp<op>d m512, m512, m512***.
(3) ***__m512i _mm512_storeu_si512 (void const* mem_addr, __m512i a)*** Store 512-bits of integer data from "a" into memory. mem_addr does not need to be aligned on any particular boundary. Generally, this instrinsic is converted into: ***vmovdqu32 m512, zmm***.

3.3 Reduction operation in Open MPI

We implement our advanced reduction operation with AVX, AVX2, AVX-512 support in a component in Open MPI, based on a Modular

Component Architecture [40] that facilitates extending or substituting Open MPI core subsystem with new features and innovations. We add our AVX-512 optimization in a specialized component that implements all predefined MPI reduction operations with vector reduction instructions, as in Figure 2. From a practical standpoint, our module will extract the processor feature flag and check related flags, selecting at runtime the set of functions supporting the most advanced ISA (AVX-512, AVX2 or AVX/SSE), or fallback to the default basic module if the processor has no support for such extensions, as shown in Figure 3. To be more specific, we explicitly check CPUID – a processor supplementary instruction allowing software to discover details of the processor, to determine processor type and whether features such as SSE/AVXs are implemented and supported.

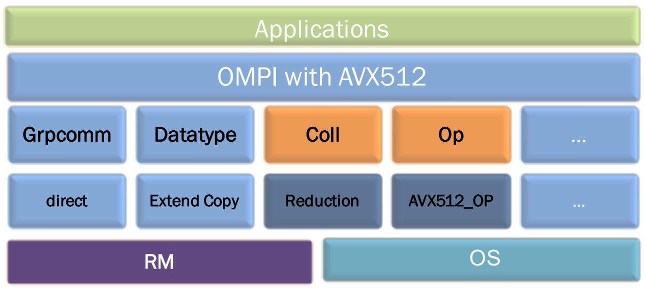

Figure 2: Open MPI architecture. The orange boxes represent components with added AVX-512 reduction features. The dark blue colored boxes are new modules.

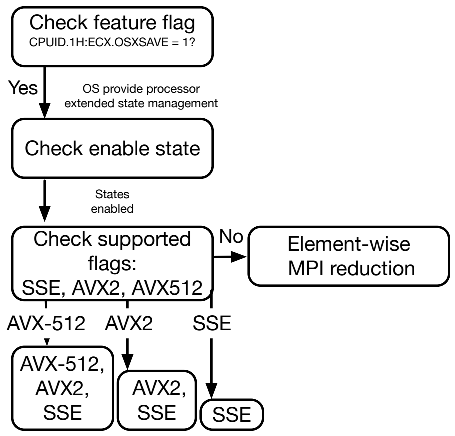
Check feature flag
CPUID.1H:ECX.OSXSAVE = 1?
Yes
OS provide processor extended state management
Check enable state
States enabled
Check supported flags: SSE, AVX2, AVX512
No
Element-wise MPI reduction
AVX-512
AVX2
SSE
AVX-512, AVX2, SSE
AVX2, SSE
SSE

Figure 3: Integrate and automatically activate the AVX component into the Open MPI build system

To be noted, the computational benefit in our component and modules can be extended out the scope of reduction operation to general mathematics and logic operations. This advanced operation module/code-snippet can be easily adapted to other computational intensive software stacks.

To use vector instructions in applications, it can be exploited in several fashions: (a) relying on automatic vectorization support provided by the compiler; (b) explicitly calling vector instructions from assembly or via intrinsic functions; (c) adapting intrinsic functions into programming models or languages for applications to use. The first strategy by using auto-vectorization, is portable and "future-proof", which means that it can quickly adapt code to a future generation of processors, with the only required step being a re-compilation of the code. However, to effectively use automatic vectorization, programmers must follow guidelines and restrictions for vectorizable code, and provide compile-time options largely dependent on the capability and efficiency of a specific compiler. And programmers also need to be aware of the specifics of the instructions that are supported by a processor. Additionally, compilers have strong limitations in the analysis and code transformations phases that prevent an efficient extraction of SIMD parallelism in real applications [26]. The second method allows more control over the very low-level instruction stream, but the use of intrinsics is time-consuming and error-prone for application programmers and users. For our work, to integrate the use of AVX-512 features in the Open MPI stack, we prefer to adopt the second approach – we use intrinsics and compile flags together to guide the compiler in the vectorization phase to maximize performance.

A reduction is a common operation encountered in many scientific applications. Those applications have large amounts of data-level parallelism and should be able to benefit from SIMD support for reduction operation. Especially in deep learning applications, it needs to frequently calculate and update the gradients, which is typically very computation extensive. Traditional reduction operation performs element by element of the input buffers, which executes as a sequential operation or it is possible could be vectorized under particular circumstance or with a specific compiler or constraints. Sometimes it may suffer from dependencies across multiple loop iterations. Figure 4 illustrates the difference between a scalar operation and a vector operation for AVX, AVX2 or AVX-512, respectively. It is an example of a vector instruction processing multiple elements together at the same time, compared to executing the additions sequentially. Where a scalar processor would have to perform a load, an computation and a store instruction for every element, a vector processor perform one load, one computation and one store for multiple elements. An AVX-512 SIMD-vector can process multiple elements at the same time. For example, it can store 8 double-precision floating-point numbers or 16 integer values, also allow the computation of those elements by executing a single instruction. AVX-512 reduction instructions perform arithmetic horizontally across active elements of a single source vector and deliver a scalar result.

Intel AVX-512 intrinsic provides arithmetic reduction operation for integer and float-pointing, also supports logical reduction operations for an integer type. This gives the chance to create AVX-512 intrinsic based reduction support in MPI which will highly increase the performance of MPI local reduction. Additionally, AVX-512 can perform scatter reduction operation with the support of predivector register, which behaves in a vectorized manner. This profoundly expands the limitation of consecutive memory layout for reduction operation to non-contiguous data sets at the same time generic and efficient, but such operations are not needed for the predefined MPI reduction operations.

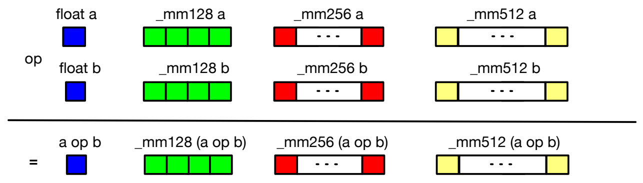

Figure 4: Example of single precision floating-point values using : (🟦) scalar standard C code, (🟩) AVX SIMD vector of 4 values , (🟥) AVX2 SIMD vector of 8 values, (🟨) AVX-512 SIMD vector of 16 values

Algorithm 1 AVX based reduction algorithm

```
types_per_step        ▷ Number of elements in vector
left_over             ▷ Number of elements waiting for reduction
count                 ▷ Total number of elements for reduction operation
in_buf                ▷ Input buffer for reduction operation
inout_buf             ▷ Input and output buffer for reduction operation
sizeof_type           ▷ Number of bytes of the type of the in_buf / inout_buf
1:  procedure ReductionOp( in_buf, inout_buf, count )
2:      types_per_step = vector_length(512) / (8 × sizeof_type)
3:      #pragma unroll
4:      for k ⟵ types_per_step to count do
5:          _mm512_loadu_si512 from in_buf
6:          _mm512_loadu_si512 from inout_buf
7:          _mm512_reduction_op
8:          _mm512_storeu_si512 to inout_buf
9:          Update left_over
10:     if ( left_over ≠ 0 ) then
11:         Update types_per_step >>= 1
12:         if ( types_per_step ≤ left_over) then
13:             _mm256_loadu_si256 from in_buf
14:             _mm256_loadu_si256 from inout_buf
15:             _mm256_reduction_op
16:             _mm256_storeu_si256 to inout_buf
17:             Update left_over
18:     if ( left_over ≠ 0 ) then
19:         Update types_per_step >>= 1
20:         if ( types_per_step ≤ left_over) then
21:             _mm_llddqu_si128 from in_buf
22:             _mm_llddqu_si128 from inout_buf
23:             _mm128_reduction_op
24:             _mm_storeu_si128 to inout_buf
25:             Update left_over
26:     if (left_over ≠ 0 ) then
27:         while ( left_over ≠ 0 ) do
28:             Set case_value
29:             Switch(case_value) : {8 Cases}
30:             Update left_over
```

For our optimized reduction operation, we employ and apply multiple methods to investigate how to achieve the best performance on different processors, as shown in algorithm1. For a better description, in the rest of the paper, we assume that the hardware supports AVX-512. In the algorithm's for-loop section: First of all, we explicitly use 512 bits long vector loads and stores for memory operation rather than using the memory copy (memcpy) function provided by the standard library, because some systems and compilers may not perform the best assembling techniques of using ZMM registers to load and store. After we have the elements loaded in registers, we apply mathematical vector operation to perform a reduction on the whole vector. We repeat this pattern until the remainders cannot fulfill a 512 bits vector, then we fallback to use YMM registers to process elements that fit in the 256 bits registers. And so on, then we execute with 128 bits vectors.

Eventually, we reach the last section of the optimization. We have noticed that depending on the number of elements to apply the operation on, significant execution time is often spent in the prologue, that deals with the remainder, those few elements that cannot fulfill a vector. Intel provides AVX mask intrinsics for mask operations that can vectorize the remainder loop. Still, significant overhead is involved in creating and initializing the mask and executing a separate and additional code path, which can result in low SIMD efficiency. The vectorized remainder loops can be even slower than the scalar executions, because of the overhead of masked operations and hardware. Typically the compiler can determine if the remainder should be vectorized based on an estimate of the potential performance benefit. When trip count information for a loop is unavailable, however, it will be difficult for the compiler to make the right decision. Therefore, for the remainder, we use Duff's device [39] manually implementing a loop unrolling by interleaving two syntactic constructs of C: the do-while loop and a switch statement, which helps the compiler to optimize the device correctly. We benefit from two aspects of Duff's device. First of all, the loop is unrolled, which trades larger code size for more speedup by avoiding some of the overhead involved in checking whether the loop is finished or jump back to the top of the loop. It can run faster when it is executing straight-line code instead of jumping. The second aspect is the switch statement. It allows the code to jump into the middle of the loop the first time through. Execution starts at the calculated case label, and then it falls through to each successive assignment statement, just like any other switch statement. After the last case label, execution reaches the bottom of the loop, at which point it jumps back to the top. The top of the loop is inside the switch statement, so the switch is not re-evaluated anymore. Our Duff's device loop uses eight cases in the switch statement, so the number of iterations is divided by eight. If the remaining elements to be processed aren't multiple of eight, then some elements are left over. Most algorithms first deal with blocks of 8 elements at a time and then handle the remainders (less than eight) at the end, but our Duff's device code processes the remainders (less than eight) at the beginning. The function calculates "count % 8" for the switch statement to figure out what the remainder will be, and jumps to the case label for that many elements. Then the loop continues to deal with blocks of eight elements.

Table 1 shows the variety of MPI_Types and MPI_Ops are supported in our optimized reduction operation module, which matches the combination of types and operations defined by the MPI standard. Table2 lists the supported x86 instruction set architectures and related CPU flags from legacy SSE to the latest AVX-512 instruction sets. To be noted, our work mainly focuses on the "Fundamental"

Table 1: Supported types and operations

Types	uint8 - uint64	float	double
MAX	✓	✓	✓
MIN	✓	✓	✓
SUM	✓	✓	✓
PROD	✓	✓	✓
BOR	✓	–	–
BAND	✓	–	–
BXOR	✓	–	–

Table 2: Supported CPU flags

Instruction Sets	CPU flags			
AVX	AVX512BW	AVX512F	AVX2	AVX
SSE	SSE4	SSE3	SSE2	SSE

feature instruction set with flag AVX512F, available on Knights Landing processors and Intel Xeon processors. It contains vectorized arithmetic operations, comparisons, type conversions, data movement, data permutation, bitwise logical operations on vectors and masks, and miscellaneous math functions. This is similar to the core feature set of the AVX2 instruction set, with the difference of more comprehensive and more extended registers, and more functional supports for float-pointing and integer.

4 PERFORMANCE TOOL EVALUATION

To understand the performance, we analyzed our AVX-512 enabled Open MPI reduction operation using Performance API (PAPI) [36] – a tool that can expose hardware counters, allowing developers to correlate these counters with the application performance. PAPI is a portable and efficient API to access hardware performance monitoring registers/counters found on most modern microprocessors. These counters exist as a small set of registers that count "events", which are occurrences of specific signals and states related to the processor's function. Monitoring these events facilitates correlation between the structure of source or object code and the efficiency of the mapping of that code to the underlying architecture. This correlation has a variety of uses in performance analysis and tuning.

We aim to use hardware performance counters in PAPI to measure two aspects: (1) Memory operation instructions: the total number of load and store instructions. (2) Branching instructions: number of branch execution instructions including branch instructions taken and not-taken, instructions mispredicted and instructions correctly predicted, which have a significant impact on performance. For example, mispredicted branches can disrupt streams of micro-ops or cause the execution engine to waste execution resources on executing streams of micro-ops in the non-architected code path.

Figure 5 shows the total number of instructions, and memory access instructions of load and store, and branch instructions (due to the stability of the results we choose not to clutter the graphs with standard deviation). We can see that for our optimized reduction operation, the total number of instructions is largely reduced. Also, memory access and branch instructions have decreased compared to the default implementation in Open MPI. The explanation

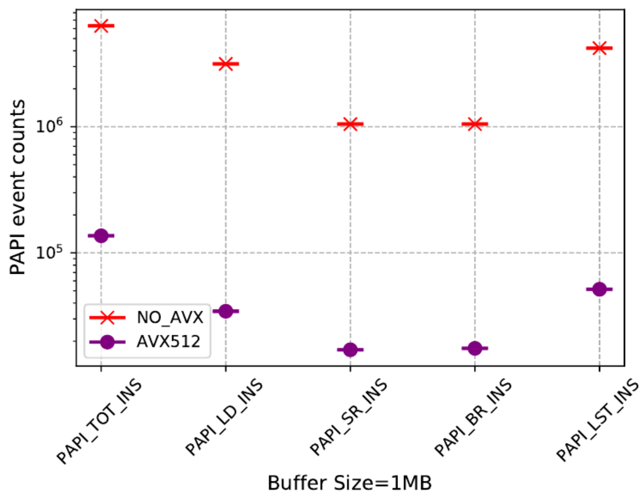

Figure 5: Comparison between AVX-512 optimized OMPI and default OMPI for MPI_SUM reduction with PAPI instruction events overview

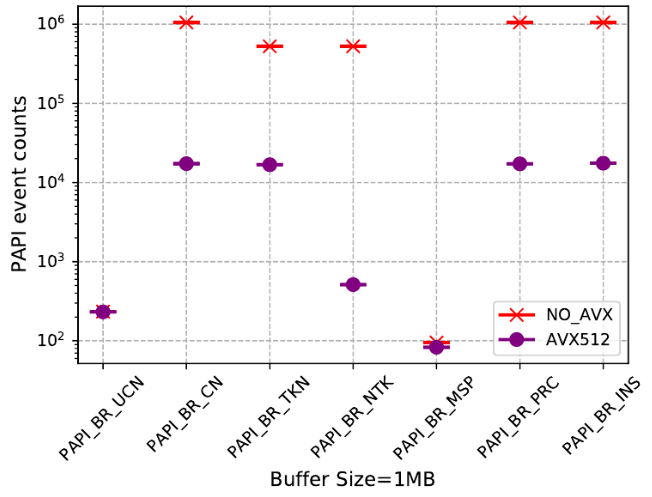

Figure 6: Comparison between AVX-512 optimized OMPI and default OMPI for MPI_SUM reduction with PAPI branch counters

here is straightforward: longer vectors can load and store more elements with each instruction than non-vector loads and stores, which means that we need fewer loads and stores dealing with the same amount of reduction data. Consequently, this will decrease the loop iteration. Our implementation reduced the number of loads and stores instructions by a factor of 90X and 60X, respectively. At the same time, for branching instructions, our optimization decreased by 60X. We also investigated the cache misses of L1 and L2 caches. Because we are dealing with an extensive contiguous data, which means data access patterns are very regular and easy to predict, all predicted accesses will be consumed so that the cache misses are not showing significant variation.

Figure 6 illustrates the instruction count details of branch instructions of both AVX-512 optimized implementation and the default element-wise reduction method. By using long vectors, we largely decreased the "for loop" of the reduction operation. Consequently, the AVX-512 code has much less control and branching instructions. Which means we have less conditional branch instructions. Especially, for conditional branch instructions not taken, we gain more benefits compare to others, which shows conditional branch instructions are being correctly predicted.

5 EXPERIMENTAL EVALUATION

We conduct our experiments on a local cluster which is an Intel(R) Xeon(R) Gold 6254 (AVX512F) based server running at 3.10 GHz. Our work is based upon Open MPI master branch, revision #75a539. Each experiment is repeated 30 times, and we present the average results. For all tests, we use a single node with one process, because our optimization aims to improve the performance of the computation part of reduction operation rather than the communication.

This section compares the performance of the reduction operation with two implementations. For Open MPI default reduction operation base module, it performs element-wise computation across two input buffers. For each loop iteration, it processes two elements. Our new implementation uses AVX-512 vector instruction executing reduction operation on the same inputs, but for each iteration, it deals with two vectors containing all the elements within the vectors which represent a vector-wise operation. For the reduction benchmark, we use the MPI_Reduce_local function call to perform the local reduction for all supported MPI operations using an array of different sizes.

We present to compare arithmetic SUM and logical BAND. For the experiments, we flushed cache to ensure we are not reusing cache for a fair comparison.

Figure 7 and Figure 8 show the result for the MPI_SUM and MPI_BAND. It should be noted for the default Open MPI's compiler, despite the provided optimization flags, did not generate auto-vectorized code. Our optimization uses intrinsics which gives us complete control of the low-level details at the expense of productivity and portability.

Results demonstrate that using AVX-512 enabled operation the performance can be improved by order of magnitude compared with the element-wise operation. To be more specific, when the total size of the reduction elements is small, the performance benefit remains low. However, when the buffer size bigger than 4KB, the performance advantage becomes considerable and stable. We also compare MPI operation together with memcpy, to indicates the peak memory bandwidth for a similar operation. To make a fair comparison, we list the complete execution sequence of reduction operation and memory copy operation. We can see that for a MPI reduction operation, it needs two loads from both input memory, and then an additional computation, eventually followed by one store to save the results into memory. For memcpy it only needs one load from source and one store to destination. The result shows that even with an additional computation included, and our optimized AVX-512 reduction operation achieves a high level of memory bandwidth which is comparable as memcpy. To be remarked, when the reduction buffer size reaches 1 megabyte, our implementation achieves almost the same performance as memcpy which indicates we maximize the memory bandwidth.

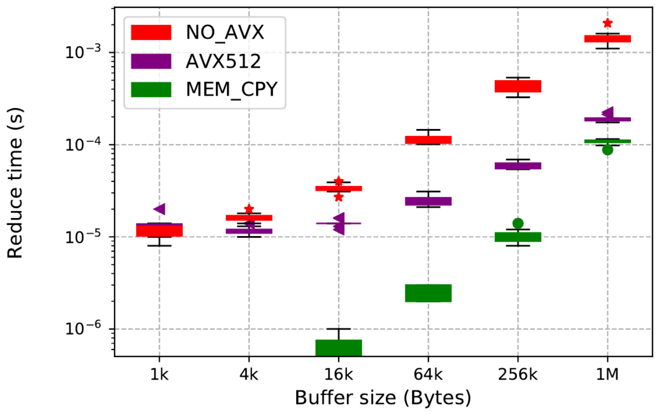

Figure 7: Comparison of MPI_SUM with AVX-512 reduction enable and disable for MPI_UINT8_T together with memcpy

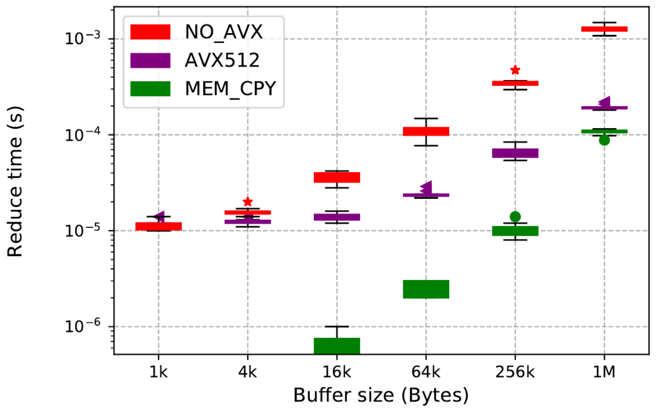

Figure 8: Comparison of MPI_BAND with AVX-512 reduction enable and disable for MPI_UINT8_T together with memcpy

6 DEEP LEARNING APPLICATION EVALUATION

Over the past few years, advancements in deep learning have driven tremendous improvement in image processing, computer vision, speech recognition, robotics and control, natural language processing, and many others. One of the significant challenges of deep learning is to decrease the extremely time-consuming cost of the training process. Designing a deep learning model requires design space exploration of a large number of hyper-parameters and processing big data. Thus, accelerating the training process is critical for research and production. Distributed deep learning is one of the essential technologies in reducing training time. The critical aspect to understand in deep learning is that it needs to calculate and update the gradient to adjust the overall weights. Processes need to prepare and calculate all the gradient data, which is usually very large. When such data and calculations are too extensive, users need to parallelize these calculations and computations. It indicates the training needs to be executed on distributed computing nodes working in parallel, and each node works on a subset of the data. When each of these processing units or workers (CPUs, GPUs, TPUs, etc.) is done calculating the gradient for its subset; they then need to communicate its results to the rest of the processes involved.

In this section, we investigate and experiment on Horovod [33] - an open-source component of Michelangelo's deep learning toolkit makes it easier to start and speed up distributed deep learning projects with TensorFlow. Horovod utilizes Open MPI to launch copies of the TensorFlow program. Open MPI will transparently set up the distributed infrastructure necessary for processes to communicate with each other. All the user needs to do is to modify their program to average gradients using an MPI_Allreduce operation. Conceptually Allreduce has every process to share its data with all other processes and applies a reduction operation. This operation can be any reduction operation, such as sum, max or min. In other words, it reduces the target arrays in all processes to a single array and returns the result array to all processes. Horovod uses a ring-allreduce approach, which is a bandwidth optimal [31] algorithm if the tensors are large enough, but does not work as efficiently for smaller tensors. Horovod can also use a Tensor Fusion - an algorithm that fuses tensors together before it calls ring-allreduce. The fusion method allocates a large fusion buffer and executes the allreduce operation on the fusion buffer. In the ring-allreduce algorithm, each of N nodes communicates with two of its peers $2*(N - 1)$ times. During this communication, a node sends and receives chunks of the data buffer. In the first $N-1$ iterations, received values are added to the values in the node's buffer. In the second $N-1$ iterations, after each process receives the data from the previous process, then it applies the reduction and proceeds to send it again to the next process in the ring. We can see that during the allreduce processing phase, there are $P*(N - 1)$ reduction operations that occurred with big fusion buffer size, which is very computation intensive. Our AVX-512 optimized reduction operations can significantly improve the performance of the computation and reduction part of those collective operations.

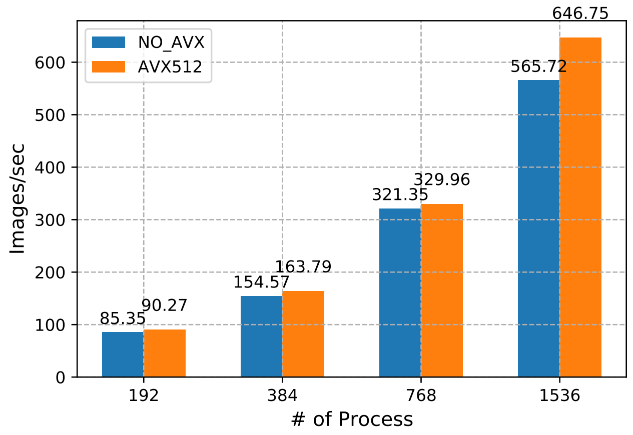

Figure 9: tf_cnn_benchmarks results using Horovod (model: alexnet) on stampede2 with AVX-512 optimized Open MPI and default Open MPI

We conducted our experiments on Stampede2 with Intel Xeon Platinum 8160 ("Skylake" supports AVX512F) nodes; each node has 48 cores with two sockets. For each node, it has 192GB DDR4 memory. For each core, it has 32KB L1 data cache and 1MB L2. The nodes are connected via Intel Omni-Path network. We experimented with TensorFlow CNN benchmarks using Horovod with tensorflow-1.13.1.

Figure 9 shows the performance comparison of our AVX-512 optimized reduction operation and the default reduction operation in Open MPI for Horovod (with synthetic datasets and AlexNet model) to train an application called tf_cnn_benchmarks [1]. Comparing to default element-wise reduction implementations, with the increasing number of processes, our design shows increasing improvements, which start at 5.45% and eventually rise to 12.38% faster than default Open MPI on 192 processes and 1536 processes respectively. We notice that the performance benefit increases with more processes/nodes. It is because with more MPI processes participated in reduction operation, the fact that each one of them is simultaneously using our AVX optimized Open MPI operations drives up the overall application performance.

7 CONCLUSION

In this paper, we pragmatically demonstrated the benefits of Intel AVX, AVX2 and AVX-512 vector operations in the context of MPI reduction operations. We assess the performance advantages of different features introduced by AVX and extended our investigation and analysis to a fully-fledged implementation of all predefined MPI reduction operations. We introduced this new reduction operation module in Open MPI using AVXs' intrinsics supporting different kinds of MPI reduce operations for multiple MPI types. We demonstrated the efficiency of our vector reduction operation using a benchmark calling MPI_Reduce_Local. Experiments are conducted on an Intel Xeon Gold cluster, which shows with AVX-512 enabled reduction operations, we achieve 10X performance benefits. To further validate the performance improvements, experiments are conducted using Skylake processor using a deep learning application using distributed model Horovod, which calculates and updates the gradient to adjust the weights using an MPI_Allreduce. Our new reduction strategy achieved a significant speedup across all ranges of processes, with a 12.38% improvement with 1536 processes. Our analysis and implementation of Open MPI optimization provide useful insights and guidelines on how wide vector operations, in this case, Intel AVX extensions, can be used in actual high-performance computing platforms and software to improve the efficiency of parallel runtimes and applications. Our AVX-512 enabled Open MPI proves that taking advantage of hardware capabilities remains of critical interest to software development, and that even a small improvement in the MPI implementation can have a significant impact on applications.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. (1725692); and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors would also like to thank Texas Advanced Computing Center (TACC). For computer time, this research used

the Stampede2 flagship supercomputer of the Extreme Science and Engineering Discovery Environment (XSEDE) hosted at TACC.

REFERENCES

[1] [n. d.]. A benchmark framework for Tensorflow. https://github.com/tensorflow/benchmarks

[2] ARM. 2018. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[3] Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. 2018. Stencil Codes on a Vector Length Agnostic Architecture. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/3243176.3243192

[4] Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rubén Langarita, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. 2019. Using Arm's scalable vector extension on stencil codes. *The Journal of Supercomputing* (Apr 2019).

[5] M. Boettcher, B. M. Al-Hashimi, M. Eyole, G. Gabrielli, and A. Reid. 2014. Advanced SIMD: Extending the reach of contemporary SIMD architectures. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–4. https://doi.org/10.7873/DATE.2014.037

[6] Léon Bottou. 2010. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of COMPSTAT'2010*, Yves Lechevallier and Gilbert Saporta (Eds.). Physica-Verlag HD, Heidelberg, 177–186.

[7] Berenger Bramas. 2017. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *International Journal of Advanced Computer Science and Applications* 8, 10 (2017). https://doi.org/10.14569/ijacsa.2017.081044

[8] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing '88)*. IEEE Computer Society Press, Washington, DC, USA, 98–105.

[9] Helena Caminal, Diego Caballero, Juan M. Cebrián, Roger Ferrer, Marc Casas, Miquel Moretó, Xavier Martorell, and Mateo Valero. 2018. Performance and energy effects on task-based parallelized applications. *The Journal of Supercomputing* 74, 6 (2018), 2627–2637.

[10] C. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and D. K. Panda. 2016. CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 726–735. https://doi.org/10.1109/CCGrid.2016.111

[11] M. G. F. Dosanjh, W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Gazimirsaeed, and A. Afsahi. 2019. Fuzzy Matching: Hardware Accelerated MPI Communication Middleware. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 210–220. https://doi.org/10.1109/CCGRID.2019.00035

[12] Roger Espasa, Mateo Valero, and James E Smith. 1998. Vector architectures: past, present and future. In *Proceedings of the 12th international conference on Supercomputing*. 425–432.

[13] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 608–621. https://doi.org/10.1145/2837614.2837615

[14] Message Passing Interface Forum. September,2012. MPI: A Message-Passing Interface Standard. https://www.mpi-forum.org

[15] Michael Hofmann and Gudula Rünger. 2008. MPI Reduction Operations for Sparse Floating-point Data. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 94–101.

[16] Dan Andrei Iliescu. 2018. Arm Scalable Vector Extension and application to Machine Learning. Retrieved October, 2018 from https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning

[17] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer Manuals. Retrieved November 11, 2019 from https://software.intel.com/en-us/articles/intel-sdm

[18] Intel. 2019. 64-ia-32-architectures instruction set extensions reference manual. https://software.intel.com/en-us/articles/intel-sdm

[19] Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-1-basic-\architecture

[20] Raehyun Kim, Jaeyoung Choi, and Myungho Lee. 2019. Optimizing Parallel GEMM Routines Using Auto-Tuning with Intel AVX-512. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2019)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/3293320.3293334

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[22] David Levine, David Callahan, and Jack Dongarra. 1991. A comparative study of automatic vectorizing compilers. *Parallel Comput.* 17, 10 (1991), 1223 – 1244. https://doi.org/10.1016/S0167-8191(05)80035-3 Benchmarking of high performance supercomputers.

[23] Zhenyu Li, James Davis, and Stephen Jarvis. 2017. An Efficient Task-based All-Reduce for Machine Learning Applications. 1–8. https://doi.org/10.1145/3146347.3146350

[24] Roktaek Lim, Yeongha Lee, Raehyun Kim, and Jaeyoung Choi. 2018. An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512. *Cluster Computing* 21, 4 (Dec 2018), 1785–1795.

[25] Xi Luo, Wei Wu, George Bosilca, Thananon Patinyasakdikul, Linnan Wang, and Jack Dongarra. 2018. ADAPT: An Event-based Adaptive Collective Communication Framework. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 118–130. https://doi.org/10.1145/3208040.3208054

[26] S. Maleki, Y. Gao, M. J. Garzar'n, T. Wong, and D. A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 372–382.

[27] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. Association for Computing Machinery, New York, NY, USA, 265–274. https://doi.org/10.1145/1995896.1995938

[28] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou. 2013. Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 1107–1116.

[29] Daniel Molka, Daniel Hackenberg, Robert Schöne, Timo Minartz, and Wolfgang E. Nagel. 2012. Flexible workload generation for HPC cluster efficiency benchmarking. *Computer Science - Research and Development* 27, 4 (2012), 235–243.

[30] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. 2015. SparkNet: Training Deep Networks in Spark. arXiv:stat.ML/1511.06051

[31] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations. *J. Parallel Distrib. Comput.* 69, 2 (Feb. 2009), 117–124. https://doi.org/10.1016/j.jpdc.2008.09.002

[32] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2016. Validation of Hardware Events for Successful Performance Pattern Identification in High Performance Computing. In *Tools for High Performance Computing 2015*, Andreas Knüpfer, Tobias Hilbrich, Christoph Niethammer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer International Publishing, Cham, 17–28.

[33] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[34] H. Shan, S. Williams, and C. W. Johnson. 2018. Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 1–11.

[35] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (Mar 2016), 34–46. https://doi.org/10.1109/MM.2016.25

[36] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[37] Jesper Larsson Träff. 2010. Transparent Neutral Element Elimination in MPI Reduction Operations. In *Recent Advances in the Message Passing Interface*, Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–284.

[38] W. J. Watson. 1972. The TI ASC: a highly modular and flexible super computer architecture. In *AFIPS '72 (Fall, part I)*.

[39] Wikipedia contributors. [n. d.]. Duff's device — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Duff%27s_device [Online; accessed 2-May-2020].

[40] Dong Zhong, Aurelien Bouteiller, Xi Luo, and George Bosilca. 2019. Runtime Level Failure Detection and Propagation in HPC Systems. In *Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI '19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 11 pages. https://doi.org/10.1145/3343211.3343225

[41] D. Zhong, P. Shamis, Q. Cao, G. Bosilca, S. Sumimoto, K. Miura, and J. Dongarra. 2020. Using Arm Scalable Vector Extension to Optimize OPEN MPI. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 222–231.