Centaurus: Characterizing the Energy Efficiency of Contemporary CPUs for Running Sparse Problems

Anonymous Author(s)

***Abstract*—Sparse matrix-vector multiplication (SpMV) is a widely used computational kernel in various applications, ranging from traditional scientific computing and graph analytics to modern machine learning applications. Enhancing the efficiency of SpMV – not just their speed – has become increasingly important. To this end, with the growth of machine learning and the rising significance of sparsity within this field, PyTorch, a well-known machine learning library, provides support for SpMV. On the other hand, the ARM ISA offers a promising platform in data centers for executing SpMV operations with low compute intensity, suitable for CPU capabilities. To further investigate the performance efficiency of CPU-based platforms for diverse SpMV kernels, we measure the execution time and power consumption of a PyTorch-based SpMV implementation on three contemporary CPUs including an Ampere Altra, an AMD Epyc, and an Intel Xeon CPU. Demonstrating superior power efficiency, Ampere's Altra processor outperforms x86 processors when performing PyTorch SpMV in scenarios with extremely high sparsity or when power efficiency is prioritized over execution time.**

***Index Terms*—workload characterization, performance evaluation, instruction set architecture, microarchitecture**

I. Introduction

Sparse matrix-vector multiplication (SpMV) has been and continues to be a major computational kernel for applications such as scientific computing, graph analytics, deep neural networks, and modern large language models [1]. However, executing SpMV efficiently remains a challenge for both CPUs and GPUs today due to the inherent sparsity of the data structures involved—even with several optimizations proposed for SpMV in recent years [2]–[10]. SpMV, characterized by low computational intensity, may benefit more from CPUs, especially as modern data centers continue to rely on general-purpose CPUs for day-to-day operations. However, the sparse structure in SpMV leads to low vectorization, which hampers efficient computation by the vector units in common consumer and commercial CPUs [11]–[13].

Challenges related to efficiently running sparse problems in modern data centers are linked to the growing demand for efficient computing, particularly in terms of power efficiency. Power efficiency has historically been crucial for mobile devices dependent on limited power sources. Today, it is also a significant concern for data centers as their energy consumption continues to rise—so much so that companies like Amazon and Google are now investing in nuclear energy [14]. Investing in nuclear energy exemplifies one approach to addressing limited energy: generating more energy. Another approach to meeting the growing power demands of data centers is the adoption of more power-efficient hardware. To this end, many data centers are experimenting with the use of ARM-based processors [15]. The ARM instruction set architecture (ISA) has been gaining popularity partly due to its widespread use in mobile devices and its implementation in Apple's laptops [15]. This popularity is also driven by the reduced complexity of the Reduced Instruction Set Computer (RISC) architecture and its associated power efficiency [11].

As sparsity and SpMV gain popularity and become key kernels in applications beyond traditional domains and into modern machine learning, the software implementations and libraries supporting them become as important as the hardware that executes them. PyTorch is a well-known machine learning library recognized for its flexibility and ease of use [16]. As a machine learning library, it is capable of performing tasks that are computationally intensive in sparse matrix-vector multiplications. Over the past few years, PyTorch has been adding features to support sparse matrix operations. With its sparse API still in beta, it continues to incorporate features such as semi-structured sparsity to further enhance the performance of sparse operations [17]. Additionally, extensive data has been collected on the performance and power efficiency of ARM processors, including in the data center environment [15], [18]. Some studies have even used the ARM Performance Library on ARM processors as a baseline for their own SpMV improvements on CPUs [19]. However, none have examined PyTorch's implementation of single and multi-core SpMV on data center-scale processors to date.

To explore the intersection of power-efficient platforms and popular libraries for running SpMV, this paper **c**haracterizes the **en**ergy efficiency of con**t**empor**a**ry CP**U**s for **ru**nning **s**parse problems (Centaurus). It measures the execution time and power efficiency of a PyTorch-based SpMV implementation on the Ampere Altra, AMD Epyc, and Intel Xeon CPUs across a range of SpMV kernels with varying sparsity levels. These measurements reveal that the Ampere Altra is more power efficient than the other processors across every SpMV kernel in exchange for higher execution times and that the Ampere Altra is more energy efficient than the other processors when sparsity is extremely high.

II. Centaurus: Contributions & Methodology

This paper compares the performance of PyTorch's SpMV operation on the Ampere Altra processor [11] against the AMD Epyc 7313P [12] and Intel Xeon 4216 [13] processors in terms of execution time and power consumption. This comparison not only examines the differences between the Arm and x86 ISAs but also between Arm's Neoverse N1,

AMD's Zen 3, and Intel's Cascade Lake microarchitectures. To make this comparison, this paper varies the number of cores and the input matrices used by the SpMV operation.

A. Hardware Setup

We target three processors including the Ampere Altra, AMD Epyc 7313P, and Intel Xeon 4216. Comparison against two different implementations of the x86 ISA serves to provide context for the reported metrics of the Ampere Altra processor. For the sake of fairness, no processors use simultaneous multithreading (SMT). The Ampere Altra processor is an 80-core 3.0GHz ARM processor with 128-bit vector units [11]. The AMD Epyc 7313P processor is a 16-core 3.0GHz x86 processor with 256-bit vector units supporting the AVX2 extension [12]. The Intel Xeon 4216 is a 16-core 2.10GHz x86 processor with 512-bit vector units supporting the AVX512 extension [13]. For the sake of comparison and visualization, the Intel Xeon 4216 processor is limited to only 256-bits of each vector unit via the AVX2 extension (more details on the method in Section II-B).

Although the Ampere Altra and Intel Xeon 4216 processors are installed in dual-socket systems, only the first socket is used for both the SpMV operation as well as the collection of power consumption. Limiting the SpMV operation to one socket excludes the effect of data transfer between sockets and allows the metrics to reflect a direct processor-to-processor comparison. The method through which this is accomplished is detailed in Section II-B. We also make comparisons between the processors in terms of core counts. The main three groups of comparison are single core, 16 core, and maximum core performance, where maximum core is 16 cores for the AMD and Intel processors and 80 cores for the Ampere processor (more details on the method in Section II-B) .

B. Software Setup

In this paper, any given run of "the program" consists of the following steps:

1) Depending on the given arguments, import or generate the 32-bit float sparse matrices using SciPy and NumPy [20], [21].
2) Generate the 32-bit float dense vector in PyTorch [22].
3) Obtain start time using Python's built-in `time` module.
4) In a `for` loop, perform the SpMV operation using PyTorch's `torch.mm()` function [22].
5) Obtain the end time and subtract the start time to obtain the total time taken by the `for` loop.
6) Output the time taken and matrix metadata with a standard data format using Python's built-in `json` module.

Both the generated matrix and the vector are randomly generated before the `for` loop and do not change between iterations of the loop. The generated matrix is kept constant between iterations just as an imported matrix would be. Keeping the vector constant also prevents random number generation from being measured alongside the SpMV operation.

To focus on the execution time and power consumption caused by SpMV operation, it is performed in a `for` loop with a variable number of iterations. The number of iterations is based on having the loop meet a minimum execution time criterion of 10 seconds. The execution time criterion is meant to mitigate the overhead of Python itself and let the reported metrics primarily represent PyTorch's SpMV operation.

To bring the Intel Xeon 4216 processor closer in line with the other processors, we install PyTorch with no support for AVX512 instructions. Running `torch.__config__.show()` from within Python shows that the `PERF_WITH_AVX2=1` flag is set and the `PERF_WITH_AVX512=1` flag is not. Additionally, the Ampere Altra and Intel Xeon 4216 processors are installed in dual-socket systems. Without consideration, threads may move between sockets and cause unexpected variations in the recorded metrics. We prevent this using `numactl --cpunodebind=0 --membind=0`, which binds the program to the first socket.

When using fewer than the maximum cores, we take two steps to ensure that the correct number of cores are used and that the same cores are used in every run of the program. First, we use the `torch.set_num_threads()` to set the number of threads, N. Second, we use `OMP_PROC_BIND` and `OMP_PLACES` environment variables to place one thread of the SpMV on each core and to make OpenMP use the first N cores. Ampere's version of PyTorch automatically binds N threads to the first N cores in the same manner, but we set the environment variables for all processors regardless.

Running the power collection program on the same processor as the program would always affect the program's execution in some way. To make sure that it affected the program in a consistent manner, we use `taskset -c 0` to bind the power collection program to the first core of the processor. Additionally, we measure the execution time metrics and power metrics in separate runs of the program such that the execution time metrics are not impacted by power collection.

C. Workloads

The sparse matrices used in the SpMV operation are either real-world matrices taken from SuiteSparse [23] matrix collection or synthetic, randomly generated matrices with varying characteristics. All matrices have the same number of rows as columns for simplicity and are stored in memory in the compressed sparse row (CSR) format before being used in the SpMV operation. The dense vectors used in the SpMV operation are randomly generated. The variety in workloads serves two purposes: to examine any potential differences in performance between real-world and synthetic matrices and to cover a wide range of sizes and densities.

1) SuiteSparse: We use two collections of SuiteSparse [23] matrices as follows:

- `as-caida` is a collection of autonomous system (AS) relationship graphs from 2004 to 2007, sourced from the Center for Applied Internet Data Analysis. Each sparse matrix in this collection maintains the same 31,379 by 31,379 size. However, each matrix has a different density, ranging from about 0.007% to 0.01% or about 70,000 to

107,000 non-zero elements. As there are 122 matrices in this collection, every tenth matrix is selected from this collection, bringing the number of used matrices to 12.

- `389000+` is a custom collection of seven sparse matrices obtained from SuiteSparse, ranging in size from 389,874 by 389,874 to 415,863 by 415,863 and ranging in the amount of non-zero elements from 1,216,334 to 19,173,163. Each of these matrices have varying sparsities, come from varying kinds of problems, and maintain a similar overall size. The matrices are detailed in Table I.

2) Synthetic: The synthetic matrices are randomly generated matrices with varying sizes and densities. The sizes in terms of number of rows are 5,000, 10,000, 50,000, 100,000, and 500,000 rows. The densities, the number of non-zero elements over the total number of elements, are 0.00001, 0.00005, 0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, and 0.5. We pair each size with each density, so the parameters of the generated matrices are the Cartesian product of every size and density listed. The exception to this are any size and density combinations which result in greater than 100,000,000 non-zero elements. As a result, we use 35 synthetic matrices, ranging from 250 to 100,000,000 non-zero elements.

While the `as-caida` takes a close examination of a few smaller matrices and the `389000+` takes a wider examination of a few larger matrices, the sizes and densities we used for the synthetic matrices are aimed to encompass both SuiteSparse collections in terms of size and density.

D. Metrics

We report execution time (seconds, s), power consumption (watts, W), and energy consumption (joules, J). Energy consumption is derived from the measured metrics: the power consumption and the execution time of the program. The time and energy consumption metrics are reported per 1,000 iterations (kI) of the SpMV operation. Normalizing for the number of iterations eliminates the effects of the variable iterations noted in Section II-B. The result is that time and energy consumption are relative to the "amount of work done", or 1,000 iterations. The exception to this is power consumption. Watts are defined as joules over seconds or $W = \frac{J}{s}$. As the ratio between joules and seconds, the wattage of the processor stays relatively constant as the number of iterations increases. This can be seen when substituting in joules and seconds per 1,000 iterations, or $\frac{J/kI}{s/kI} = \frac{J}{s} = W$. Effectively, watts already control for the variable iterations in the same manner joules and seconds per 1,000 iterations are.

TABLE I
389000+ COLLECTION

Name	Rows	Non-Zero Elements	Density
mario002	389874	2097566	0.00138%
helm2d03	392257	2741935	0.00178%
test1	392908	9447535	0.00612%
language	399130	1216334	0.000764%
marine1	400320	6226538	0.00389%
amazon0312	400727	3200440	0.00199%
msdoor	415863	19173163	0.0111%

1) Execution Time: We measure execution time for the SpMV operation using Python's `time` module and its `time()` function. As detailed in Section II-B, we record the start and end times just before and after the SpMV's `for` loop.

2) Power Consumption: We measure power consumption using the `sensors` command and the `turbostat` command for the Arm processor and the x86 processors, respectively. We use two different power collection tools because of the lack of support from `turbostat`, as an Intel-created tool, for Arm processors. Moreover, `sensors` does not output processor power consumption on x86 processors. Both tools capture the power consumption of the processor itself and do not capture the power consumed by other components of the system, such as memory. The primary difference between `sensors` and `turbostat` is the power consumption output. `turbostat` outputs the wattage of the processor for a given program's execution. `sensors` outputs the wattage of the processor at the time that `sensors` is run.

To bridge the gap between the outputs of these two programs, we run `sensors` at one second intervals to sample the power consumed during the execution of the program. Linearly interpolating the data between the samples results in a power consumption graph which can be treated as a function $f(s)$ where s is the number of seconds since the program began execution and $f(s)$ is the sampled wattage at time s. To accurately average the wattage, the estimated total energy consumption in joules is calculated and then divided by the total execution time in seconds. This is equivalent to the mean of the function f for some interval $[a, b]$, where the integral is the estimated total energy consumption and $b - a$ is the total execution time. However, power measurement begins at 0 seconds, therefore $a = 0$. Additionally, power measurement ends when the program execution ends, so $b = S, S \in \mathbb{R}$ where S is the total execution time. Substituting in these values results in the following equation:

$$\begin{aligned} J &= \int_a^b f(s)ds \\ &= \int_0^S f(s)ds \end{aligned} \tag{1}$$

As the function $f(s)$ is not a continuous curve, the area under the curve can instead be obtained with the trapezoid rule where sampling every second means $x_k = k$, $x_{k-1} = k - 1$, and $\Delta x_k = 1$:

$$\begin{aligned} \int_0^S f(s)ds &\approx \Sigma_{k=1}^S \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k \\ &= \Sigma_{k=1}^S \frac{f(k-1) + f(k)}{2} \end{aligned} \tag{2}$$

As the execution time of the program is not a whole number, power sampling actually ends before the end of the program by a variable amount. To account for this, the value of the last sample, taken at the last whole second, is assumed to be constant, interpolated to the remaining execution time, and added to the sampled energy consumption. As a result, $S = \lfloor S \rfloor + (S - \lfloor S \rfloor)$ where $S - \lfloor S \rfloor$ is the execution time past $\lfloor S \rfloor$

in excess of a whole number. Then we calculate the estimated total energy consumption in joules as the following:

$$J \approx (S - \lfloor S \rfloor) \cdot f(\lfloor S \rfloor) + \Sigma_{k=1}^{\lfloor S \rfloor} \frac{f(k-1) + f(k)}{2} \quad (3)$$

Finally, we obtain the program's power consumption in watts:

$$\begin{aligned} W &= \frac{J}{S} \\ &\approx \frac{1}{S}((S - \lfloor S \rfloor) \cdot f(\lfloor S \rfloor) + \Sigma_{k=1}^{\lfloor S \rfloor} \frac{f(k-1) + f(k)}{2}) \end{aligned} \quad (4)$$

III. Results & Analyses

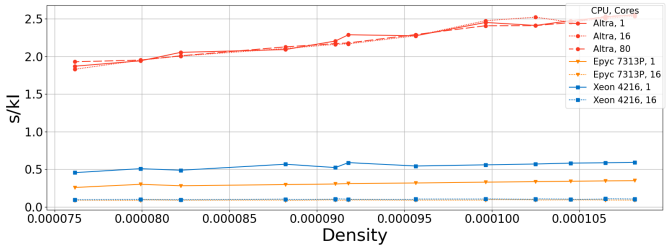
As mentioned in Section II-D, all of the measured metrics are per 1,000 iterations (kI) to eliminate the effect of variable iterations and place the metrics in terms of "amount of work done". Any figure that refers to "Max Core" is stating that the results are obtained using the maximum number of cores available on the processor as specified in Section II-A. Using the maximum number of cores availables on the processor present an opportunity to examine the effect of allowing the Ampere Altra processor to use 5× more cores than the x86 processors to accomplish the same "amount of work" in terms of 1,000 iterations. "s/kI" refers to the seconds taken by the SpMV operation for every 1,000 iterations, "J/kI" refers to the joules used by the computation for every 1,000 iterations, and "W" refers to the watts used by the computation. As these are all measures of execution time, power consumption, and energy consumption, lower is better.

A. as-caida Collection

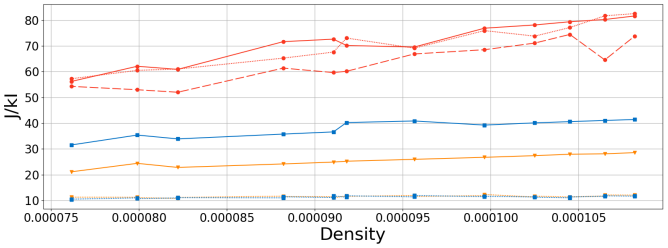
Figure 1 compares the Ampere Altra, AMD Epyc 7313P, and Intel Xeon 4216 processor metrics measured over the `as-caida` collection while using a varying number of cores. The execution time and energy consumption metrics for the Altra are consistently higher than that of the x86 processors, while the power consumption is consistently lower. Moreover, as the core count changes, all metrics for the x86 processors change while only energy consumption changes for the Altra.

1) Time: Figure 1(a) depicts the behaviors of the execution time of the SpMV operation as density increases. The Altra processor has a higher execution time than the x86 processors for every `as-caida` matrix measured, increases in execution time as the density increases much more than the x86 processors, and keeps the execution time constant when the core count increases unlike the x86 processors. Additionally, the Xeon processor takes more time than the Epyc across all densities for the 1-core case, but not the 16-core case.

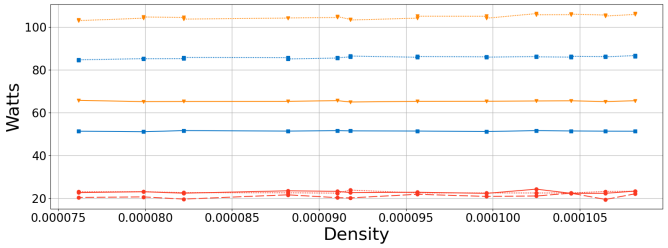
Regarding the Altra's higher execution time, comparing the width of the vector units in the processors offers one explanation. Mentioned in Section II-A, the Altra can use 128-bit wide vector units while the x86 processors can use up to 256 bits. Capable of performing computation on twice as many 32-bit floats at once, the x86 processors can make much shorter work of the same workload as the Altra. Based on this explanation, the x86 processors should complete the same workload in half the execution time of the Altra. However, the



(a) Execution Time per 1,000 Iterations (s/kI) vs. Matrix Density



(b) Energy Consumption per 1,000 Iterations (J/kI) vs. Matrix Density



(c) Power Consumption (W) vs. Matrix Density

Fig. 1. Various Metrics vs. Matrix Density for Different CPUs and Core Counts measured over the as-caida Collection

Epyc processor takes less than 14%. This can be explained by a larger number of vector units in addition to having larger vector units, an idea further explored in Section III-B1.

Regarding the Altra increasing in execution time more than the x86 processors do as density increases, the previous explanation may still apply. As density increases, the number of vector, or SIMD, operations increases. Given a SIMD workload, the Altra will take at least twice as long ($a = 2x$) as the x86 processors (x) to complete it given their respective vector unit widths. However, given twice the previous workload, the x86 processors will only take twice as long ($2x$) while the Altra will take four times as long ($2a = 2(2x)$). Figure 1(a) depicts this effect at a smaller, more discrete scale.

Regarding the Altra keeping execution time constant as the number of cores utilized increases, a bottleneck may be present for the Altra which does not exist for the x86 processors. One possible bottleneck is in the memory speeds, as the Altra supports up to 3,200 MHz RAM [11], the Epyc supports the same [12], and the Intel only supports 2,400 MHz [13]. However, if this were the bottleneck, this would likely affect the Intel processor more. Moreover, all three processors have enough L1, L2, and/or L3 cache combined to completely contain the largest `as-caida` matrix in CSR. This is because the CSR format would require at most $31379 + 1$ elements in the offsets array, based on the size of the matrix, and at

most $2(106510)$ elements for the values and indices array combined [24]. If all integers and floating point values are 32-bit wide, this results in $32(31379 + 1 + 2(106510))$ bits or 0.9776 megabytes. This can be contained within Altra's 64 kB of L1 and and 1 MB L2 cache [11], Epyc's 128 MB of L3 [12], and Intel's 22 MB of cache [25]. Another possible bottleneck is that the Altra is running up against a maximum per-core computational limit. However, if this were the bottleneck, the time would decrease as more cores are used similar to the x86 processors. Ultimately, it is possible that the overhead of PyTorch distributing the SpMV operation across multiple cores offsets any potential reduction in execution time. For the x86 processors, it follows that the time taken would decrease drastically as 15 more cores are used simultaneously than in the 1-core case.

Regarding the difference in the execution time of the x86 processors in the 1-core case, a high-level explanation can be offered by the base clock speeds and boost clock speeds supported by both processors. The Epyc supports a base clock speed of 3.00 GHz and a boost clock speed of 3.70 GHz [12] while the Intel supports 2.10 GHz and 3.20 GHz [13], respectively. Assuming all else is the same, the differences in clock speeds are easily capable of explaining the difference between the processors in the 1-core case. However, as the AMD Epyc 7313P processor was publicly launched two years after the Intel Xeon 4216 was, it is also difficult to suggest that there were no microarchitectural improvements in that time [12], [13]. This is not an entirely fair one-to-one comparison, however, as allowing the Xeon access to AVX512 extension's 512-bit wide vector units would likely serve to fill the gap between the two processors. This is especially so since the 512 bits in the Xeon's vector units are exactly twice the 256 bits in the Epyc's vector units, while the Epyc's execution times are about half that of the Xeon's. Regarding the similarity in the time taken by the x86 processors in the 16-core case, the overhead of splitting the matrix among 16 cores likely outweighed the small amount of time it took each core on either processor to complete the computation.

2) Energy and Power Consumption: Figure 1(b) depicts the behaviors of the energy consumed by the processor as the density of the SpMV matrix increases. Similar to Figure 1(a), Altra's energy consumption increase is higher than that of the x86 processors for every `as-caida` matrix. This suggests that the need to spend more time to complete the SpMV operation is tied to the need to spend more energy for the same reason. Although this may be clear, this need not necessarily be the case. Additionally, the similarity in energy consumption of the x86 processors can be explained as with Section III-A1, where the time spent moving data outweighs the time spent on computation so much so that both processors use the same time and energy to accomplish the SpMV operation.

Unlike Figure 1(a), though, an increase in Altra's core count shows a clearer change in the energy consumed. First, as one core is increased to 16 cores, the energy consumption is similar, if maybe a bit lower. Then, as 16 cores is increased to 80, the energy consumption drops. The reason for this drop can be attributed to strong power efficiency. When 16 cores is increased to 80 cores, the computation assigned to each core is spread out. By parallelizing the work, the computation could complete in a fraction of the time as with the x86 processors in Figure 1(a). Since the amount of time is constant, however, less energy is consumed to accomplish the same task in the same amount of time. The reason the effect is more pronounced when increasing from 16 to 80 cores compared to when increasing from one to 16 cores is a matter of scale where an increase from 1 to 16 is less noticeable than an increase from 1 to 80 – five times the increase.

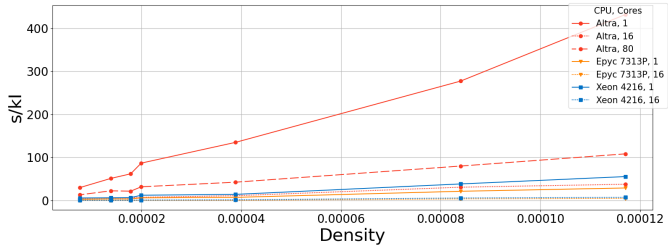
Figure 1(c) depicts the behaviors of the power consumed by the processor as the density increases. Unlike Figures 1(a) and 1(b), the Altra is clearly and consistently below the x86 processors. Unlike the Altra, the x86 processors increase in power consumption as the core count increases, opposite their behavior in Figures 1(a) and 1(b). Finally, all three processors stay at a relatively constant power consumption across all densities. The relatively low power consumption exhibited by the Altra processor is likely an example of the power efficiency touted by Ampere [11]. This power efficiency comes at a cost, however. In exchange for keeping power consumption low, more time is spent on the SpMV operation. The longer time spent on computation means that although power consumption is relatively low, energy consumption is relatively high compared to the x86 processors.

The x86 processors increase in power consumption as the core count increases due to using more cores to accomplish the same amount of work. This alone is nothing of note were it not for the Altra processor which does not exhibit this increase. When moving from one core to 16 cores, and even from 16 cores to 80 cores, the power consumption remains about the same. A plausible explanation involves the Altra processor's baseline power consumption. Assuming all of the Altra's cores are powered equally at all times, then the difference in power consumption between one core doing all of the computation and 80 cores doing one-eightieth of the work would be minimal. This contrasts with the x86 processors, which can achieve better overall power consumption by reducing the power consumption of idle cores and only ramping up overall power consumption when those idle cores are made to perform part of the SpMV operation. The x86 processors could avoid this increase in power consumption by following in the Altra's footsteps, but would use far more energy idly powering 16 cores than the Altra would actively powering 80 cores.

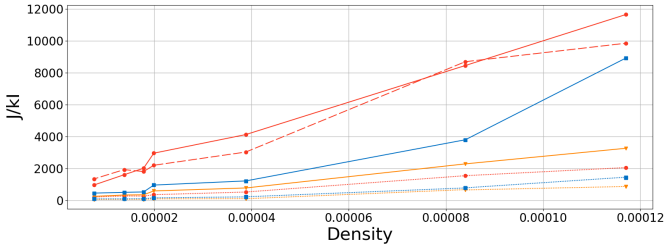
The constant power consumption exhibited by all processors at all core counts is due to a relatively small scale as the `as-caida` matrices vary little in density and not at all in size. The following Sections III-B and III-D address this by examining, respectively, a different set of larger matrices and a superset of both smaller and larger matrices than both the `as-caida` and `389000+` collections.

B. 389000+ Collection

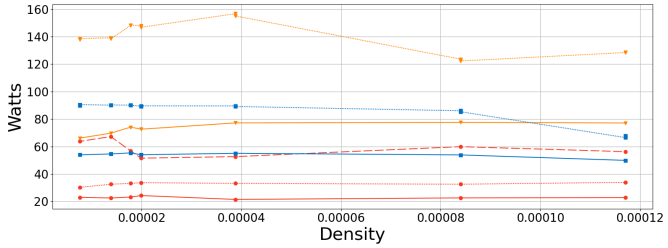
Figure 2 compares the three processor metrics measured over the custom `389000+` collection while using a varying



(a) Execution Time per 1,000 Iterations (s/kI) vs. Matrix Density



(b) Energy Consumption per 1,000 Iterations (J/kI) vs. Matrix Density



(c) Power Consumption (W) vs. Matrix Density

Fig. 2. Various Metrics vs. Matrix Density for Different CPUs and Core Counts measured over the 389000+ Collection

number of cores. Similar to Figure 1, the time and energy consumption metrics for the Altra tend to be higher than that of the x86 processors, while the power consumption tends to be lower. However, there are a few key differences, namely some areas where the Altra's metrics overlap with the x86 processors and an upward curve not previously seen in Figure 1. The main reason for these differences are the difference between the two SuiteSparse collections: matrix size. While the density range remained about the same, the increased size of the `389000+` matrices means their matrices have at least 10 times the amount of non-zero elements as the `as-caida` matrices—from at most 106,510 non-zero elements to at minimum 1,216,334 non-zero elements.

1) Time: Figure 2(a) depicts the behaviors of the time taken by the SpMV operation as density increases. The Altra processor continues to take more time per 1,000 iterations than the x86 processors for every matrix except when examining across core counts, continues to increase in execution time more than the x86 processors except when using 16 cores, and no longer keeps the execution time constant when the core count increases. Additionally, the x86 processors continue to maintain the same behaviors detailed in Section III-A1.

The Altra processor maintains its relatively high execution time compared to the x86 processors in much the same way as in Figure 1(a). This follows the vector unit explanation in Section III-A1 such that this is a hardware limit that the Altra cannot overcome as the density or number of non-zero elements increases regardless of core counts. There is a sort of exception to this, however, when comparing across core counts as opposed to purely within core counts. When comparing the Altra's execution time in the 16-core case to the x86 processors' execution time in the 1-core case, the Altra finds itself in between the x86 processors. Not only did the Altra have an execution time between the x86 processors, it manages to follow the same curve that the x86 processors do. This suggests that, for the `389000+` collection, the Altra processor's SpMV operation using 16 cores is in some way equivalent to the x86 processors' SpMV operation using 1 core. Assuming all else is equivalent and knowing that the Altra has two 128-bit vector units per core [11], this estimates that the x86 processors have around 32 vector units per core. However, this number is fairly high and moreso suggests that there exists other reasons for the difference in performance beyond the vector units themselves.

The Altra processor continues to increase in execution time more than the x86 processors, once again following the explanation offered in Section III-A1, which holds up well when considering the consistency with which the 16-core Altra execution time follows the single-core x86 processor execution times as density increases. Thus, we suggest that the positive curve in execution time as density increases is caused by the limited throughput of the vector units.

Unlike Figure 1(a), a noticeable change in the behavior of the Altra processor's execution time is the difference across core counts. Before, the execution time is about the same when increasing the number of cores. Now, when increasing from one core to 16, the execution time drops. Then, when increasing from 16 cores to 80, the execution time jumps up. This is perhaps surprising until considering the possible overheads involved. Every core must be assigned a division of the SpMV operation in every iteration. It is possible that the time spent by PyTorch dividing up the matrix alongside the active use of more cores with minimal work per core is less time efficient than having PyTorch divide up the matrix into larger chunks and distribute them among fewer cores with an optimal amount of work, especially if PyTorch has to accumulate the results from every core before starting the next iteration of the computation.

As previously mentioned, the x86 processors maintain the same behaviors detailed in Section III-A1 and shown in Figure 1(a) wherein the Epyc is lower than the Xeon in the 1-core case and both achieve a similar execution time in the 16-core case. Again, the overhead of splitting the SpMV operation up across multiple cores likely outweighs the minimal execution time of the operation on each core.

C. Energy and Power Consumption

Figure 2(b) depicts the behaviors of the energy consumed by the processor as density increases. Similarly to Figures 2(a) and 1(a), Altra's energy consumption and energy consumption increase is higher than that of the x86 processors for every

389000+ matrix. This once again suggests that the need to spend more time to complete the SpMV operation is tied to the need to spend more energy for the same reason. This is not one-to-one, however, as Figure 2(c) shows that the ratio of energy consumption to execution time increases as the core count increases. Moreover, the energy consumption of the 1-core and 80-core cases are similar, despite the execution times differing. Additionally, the energy consumption of the x86 processors no longer line up in Figure 2(b) as they did in Figure 1(b). These differences likely existed before in Figure 1(b) but are now made apparent as the larger matrices means time spent moving data no longer outweighs the time spent on computation. This is so much so that the Xeon processor visibly consumes more energy to accomplish the same task as the Epyc processor.

As with Figure 2(a), the energy consumption drops from the 1-core case to the 16-core case but rises again from the 16-core case to the 80-core case. Assuming power consumption between these core counts is constant, then this would be directly attributable to the increased execution time caused by the overheads of PyTorch and using more cores simultaneously. Figure 2(c), however, shows that this assumption cannot be made as the power consumption increases with the core count. As the number of cores used increases, the overhead of splitting the SpMV operation across cores and the overhead of powering more cores to accomplish a diminishing portion of the computation begins to outweigh the benefits of doing so in both time and energy consumption. So much so that the energy consumption of the 80-core case is almost as high as the energy consumption of the 1-core case.

Figure 1(c) depicts the behaviors of the power consumed by the processor as density increases. Once again, the Altra is consistently below the x86 processors when comparing within core counts. Both the Altra and x86 processors increase in power consumption as the core count increases, opposite their behavior in Figure 1(c). As a result, the Altra is no longer consistently below the x86 processors when comparing across core counts. Finally, all three processors no longer stay at a relatively constant power consumption across all densities.

The relatively low power consumption exhibited by the Altra processor is likely another example of the power efficiency touted by Ampere [11]. As a result, when comparing within core counts, the Altra's power consumption is still well below that of the x86 processors in much the same way as Figure 1(c). Unlike Figure 1(c), though, the power consumption of the Altra with different core counts visibly increases. This suggests that the "constant" power consumption in Section III-A2 is the result of smaller sparse matrices. The explanation offered previously, that the difference in power consumption between one core doing all of the computation and 80 cores doing one-eightieth of the work is minimal, no longer holds as the work divided among the cores has increased. The idea, however, still holds as Figure 2(c) depicts the Altra's power consumption difference when one core does all of the work, 16 cores each do one-sixteenth of the work, and 80 cores each do one-eightieth. Even when increasing the Altra's core count from one to 80, the difference in power consumption is minimal compared to when increasing the x86 processor core counts from one to 16. This supports the explanation offered in Section III-A2 where the x86 processors achieve better power efficiency at low utilization by reducing the power used by underutilized cores.

The power consumption exhibited by all processors at all core counts is no longer constant as with Figure 1(c). As we concluded in Section III-A2, the constant power consumption is due to the relatively small scale of the as-caida collection. Although the range in density for the 389000+ collection is similar, the range in size is quite different and the number of non-zero elements for the smallest matrix in the latter collection is over 10 times that of the largest matrix in the former collection. However, a larger picture can be painted with the generation and use of even smaller and even larger matrices as Section III-D.

D. Synthetic Matrices

The synthetic matrices are unlike the SuiteSparse matrices in that they are randomly generated according to a wide variety of sizes and densities up to a limit of 100,000,000 non-zero elements. Being synthetic, the numbers and their positions within the matrix are much unlike the real-world SuiteSparse matrices. Despite this, Figures 4, 5, and 6 depict similar patterns to that of Figures 1 and 2. Additionally, being generated with a wide variety of sizes and densities, the Figures 4, 5, and 6 depict execution time, energy consumption, and power consumption for a much wider range of matrix sizes, densities, and number of non-zero elements than the SuiteSparse collections depicted. As a result of varying both size and density, the volume of information is difficult to parse without being split into a separate figure for each core count, Figure 4 for the 1-core case, Figure 5 for the 16-core case, and Figure 6 for the maximum-core case. For each of these figures, the x-axis is scaled logarithmically. For only the execution time and energy consumption, the y-axis is scaled logarithmically. The legend for these figures is separated into Figure 3 due to its size.

There is one behavior that is unique to the synthetic matrices due to its ability to vary matrix sizes: depicted behaviors in all metrics tend to reappear at lower densities and higher sizes. The shared aspect of both density and matrix size is the number of non-zero elements. The relationship between the number of non-zero elements and the reported metrics is visualized in Figure 7, where energy consumption is placed over the number of non-zero elements. We prefer visualizations using density instead of number of non-zero elements for the sake of clarity.

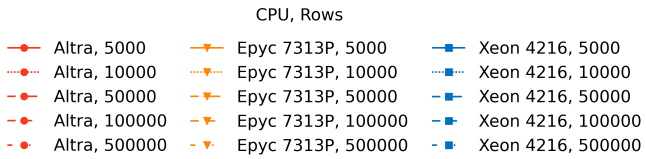
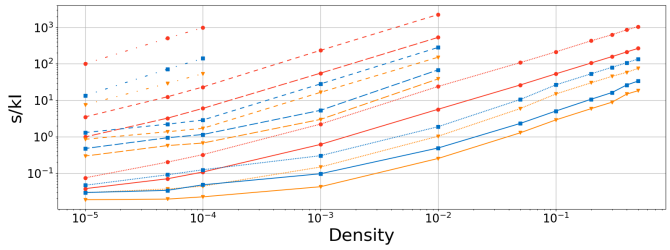
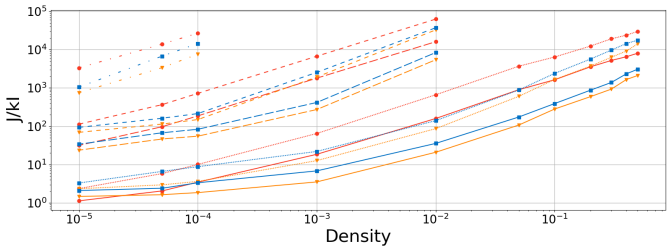


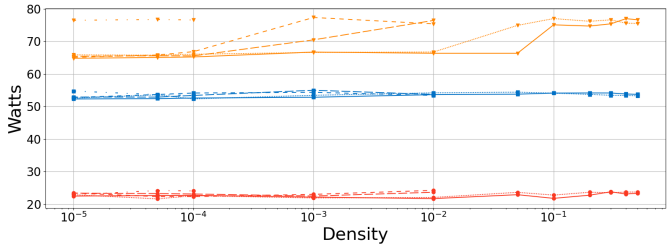
Fig. 3. Legend for Various Metrics vs. Matrix Density for Different CPUs and Matrix Rows measured over Synthetic Matrices



(a) Execution Time per 1,000 Iterations (s/kI) vs. Matrix Density

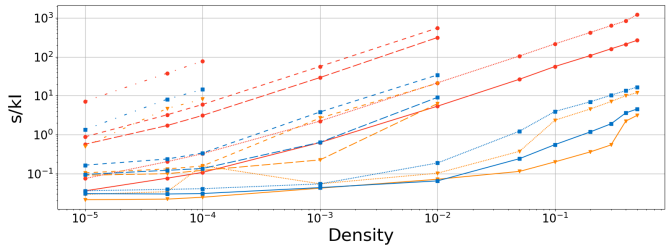


(b) Energy Consumption per 1,000 Iterations (J/kI) vs. Matrix Density

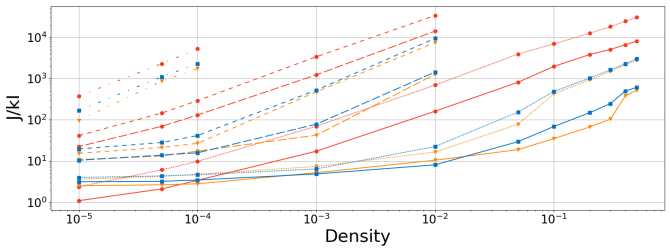


(c) Power Consumption (W) vs. Matrix Density

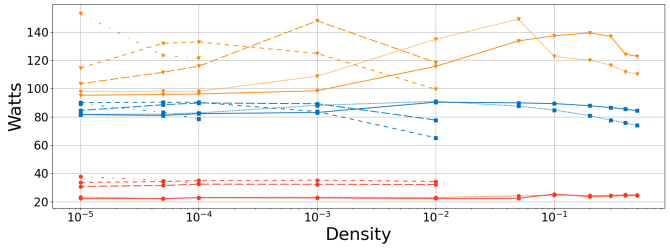
Fig. 4. Various Metrics vs. Matrix Density for Different CPUs and Matrix Rows measured over Synthetic Matrices, 1 Core



(a) Execution Time per 1,000 Iterations (s/kI) vs. Matrix Density



(b) Energy Consumption per 1,000 Iterations (J/kI) vs. Matrix Density



(c) Power Consumption (W) vs. Matrix Density

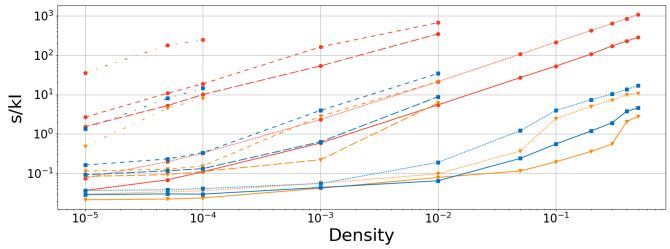
Fig. 5. Various Metrics vs. Matrix Density for Different CPUs and Matrix Rows measured over Synthetic Matrices, 16 Core

Some behaviors in the data that continue to appear when comparing Figures 1 and 2 to Figures 4, 5, and 6. The Altra processor continues to have both higher execution time and higher energy consumption than the x86 processors for every size, density, and core count. The Altra processor also continues to have low power consumption compared to the x86 processors for every size, density, and core count. For all these broad similarities, however, there are also notable differences due to the increased context of larger and smaller matrix sizes and densities. As a result, some explanations of the behavior continue to apply while others gain additional qualifications.

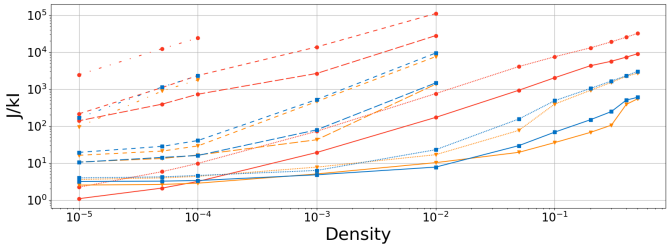
1) Execution Time: The higher execution time of the Altra processor compared to the x86 processors can be attributed to the same explanations offered in Sections III-A1 and III-B1 where the width of the vector units causes a significant difference in computational throughput. A difference which becomes more visible as density increased. This difference is still present in the center of Figures 4(a), 5(a), and 6(a), especially for the 5,000 row synthetic matrix when density is below 0.1. As the density increases, however, the difference diminishes as the execution time of the x86 processors begins to increase at the same rate as that of the Altra processor. In fact, the difference almost appears constant at higher sizes and densities. The diminishing of this difference is slowed by the use of more cores, but still diminishes at higher densities. This suggests that there is a point at which the vector units widths bottleneck begins to be outweighed by a different bottleneck, memory. This diminishing occurs when the matrix no longer fits in any processor's cache. Considering the 100,000 row matrix with a density of 0.01, the amount of data required in the CSR format would be $32(10000+1+2(100000^2 \cdot 0.01)) = 6400320032$ bits or about 800 MB. This does not fit within any processor's cache, meaning their speed of computation is entirely limited by the speed at which data can be read from memory.

2) Energy and Power Consumption: The higher energy consumption of the Altra processor compared to the x86 processors can be attributed to the same explanations offered in Sections III-A2 and III-C, which is that energy consumption is tied to execution time. It is for this reason that the energy consumption of the x86 processors begin to increase at the same rate as that of the Altra processor and is why the aforementioned bottleneck also affects energy consumption.

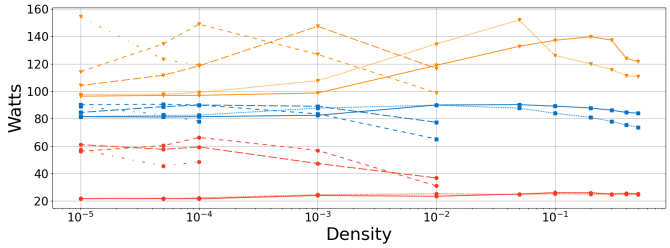
As with Figures 1(c) and 2(c), Figures 4(c), 5(c), and 6(c) depict the Altra processor as being the lowest in power consumption for all sizes and densities of synthetic matrices. With this, we conclude that the Altra processor is consistently the most power efficient processor for PyTorch's SpMV operation on one or more cores. However, in most cases, this comes at the cost of increased execution time and energy consumption. However, at very low densities in both Figure 4(b) and Figure 5(b), it is possible to see some SpMV operations where



(a) Execution Time per 1,000 Iterations (s/kI) vs. Matrix Density



(b) Energy Consumption per 1,000 Iterations (J/kI) vs. Matrix Density



(c) Power Consumption (W) vs. Matrix Density

Fig. 6. Various Metrics vs. Matrix Density for Different CPUs and Matrix Rows measured over Synthetic Matrices, Max Core

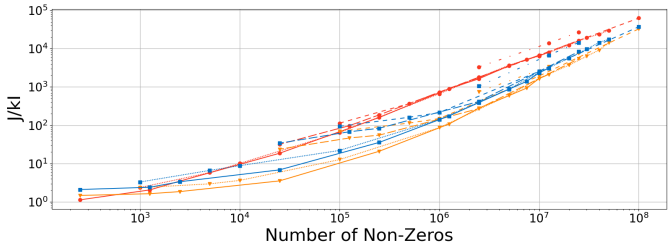


Fig. 7. Energy Consumption vs. Matrix Number of Non-Zeros for Different CPUs and Matrix Rows measured over Synthetic Matrices, 1 Core

the Altra processor is at or below the x86 processors in energy consumption. Although not depicted in previous figures, this behavior follows the explanations in Sections III-A2 and III-C regarding the Altra processor's efficient power consumption.

The impact on energy consumption caused by small workloads is so small for the Altra compared to the x86 processors that the main driver of the Altra's energy consumption is the amount of time it takes to complete the computation. This is not the case for the x86 processors whose main driver of energy consumption is powering the cores assigned to part of the computation. Another way to examine this behavior is from the perspective of power consumption. The power consumption for all processors is constant save for high amounts of non-zero elements. If power consumption is constant, then energy consumption equals execution time ($J = W \cdot s$). In that case, the Altra, with its consistently lower power consumption W, is capable of attaining a lower energy consumption J when execution time s is low. The reason the x86 processors have higher energy consumption despite their lower execution time is depicted in Figures 1(c) and 2(c) as well as between Figures 4(c) and 5(c): the power consumption of the x86 processors is both higher and increases more with core count than that of the Altra. In short, when execution time is low, the execution time advantage of the x86 processors is offset by their increased power consumption. We conclude that when execution time is sufficiently low, such as when the SpMV operation is relatively lightweight, the Altra is more energy efficient. To achieve a relatively lightweight SpMV operation, the matrix either has to be extremely dense, and thus not sparse, or the matrix has to be large with an extremely high sparsity.

Regarding the decrease in power consumption at high amounts of non-zero values, the power consumption reflects a shift in the ratio between energy consumption and execution time. As the workload grows, the speed of computation does not increase as quickly as the energy consumed by the processor does. One explanation for why this may be the case is the same memory bottleneck as with execution time, meaning the speed of computation is again limited by the speed at which data can be read from memory. As a result, the power consumption drops slightly as more energy is spent on moving data than actual computation. This explains why this decrease in power consumption is not present in Figure 4(c): the computational throughput of one core is not higher than the throughput of memory.

IV. Conclusion

In this paper, we characterized the energy efficiency of contemporary CPUs for running sparse problems by comparing the execution time and power efficiency of PyTorch's SpMV operation on the Ampere Altra, AMD Epyc 7313P, and Intel Xeon 4216 processors across a range of sparse matrices. With every matrix, we found that the Ampere Altra is more power efficient than the AMD Epyc and Intel Xeon processors, even when the Ampere Altra uses 5× as many cores. In exchange for this power efficiency, we found that the Ampere Altra either has equivalent or worse execution times than the x86 processors with every matrix. These execution times means the Ampere Altra generally spends more energy completing the same SpMV operation as the x86 processors except for when the SpMV operation has extremely high sparsity. We also found that, for large SpMV operations, the difference in execution time between the Ampere Altra and the x86 processors becomes nearly constant, making the Ampere Altra a very powerful alternative when power efficiency is prioritized over execution time.

REFERENCES

[1] J. Kang, S. Choi, E. Lee, and J. Sim, "Spdram: Efficient in-dram acceleration of sparse matrix-vector multiplication," *IEEE Access*, vol. 12, pp. 176 009–176 021, 2024.

[2] C. Li, T. Xia, W. Zhao, N. Zheng, and P. Ren, "Spv8: Pursuing optimal vectorization and regular computation pattern in spmv," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 661–666.

[3] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding up spmv for power-law graph analytics by enhancing locality & vectorization," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[4] H. Zhao, T. Xia, C. Li, W. Zhao, N. Zheng, and P. Ren, "Exploring better speculation and data locality in sparse matrix-vector multiplication on intel xeon," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 601–609.

[5] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[6] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 149–162.

[7] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 136–145.

[8] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[9] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 273–282.

[10] H. Bian, J. Huang, R. Dong, L. Liu, and X. Wang, "Csr2: a new format for simd-accelerated spmv," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 350–359.

[11] "Ampere Altra Family Product Brief." [Online]. Available: https://amperecomputing.com/briefs/ampere-altra-family-product-brief

[12] "AMD EPYC™ 7313P." [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7313p.html

[13] "Intel® Xeon® Silver 4216 Processor." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/193394/intel-xeon-silver-4216-processor-22m-cache-2-10-ghz/specifications.html

[14] "Amazon, Google make dueling nuclear investments to power data centers with clean energy," Oct. 2024. [Online]. Available: https://apnews.com/article/climate-data-centers-amazon-google-nuclear-energy-e404d52241f965e056a7c53e88abc91a

[15] N. A. Simakov, R. L. Deleon, J. P. White, M. D. Jones, T. R. Furlani, E. Siegmann, and R. J. Harrison, "Are we ready for broader adoption of arm in the hpc community: Performance and energy efficiency analysis of benchmarks and applications executed on high-end arm systems," in *Proceedings of the HPC Asia 2023 Workshops*, ser. HPCAsia '23 Workshops. New York, NY, USA: Association for Computing Machinery, 2023, p. 78–86. [Online]. Available: https://doi.org/10.1145/3581576.3581618

[16] D. Lu and S. Liu, "Real time performance evaluation of deep learning algorithms in image recognition under the pytorch framework," in *2024 International Conference on Intelligent Algorithms for Computational Intelligence Systems (IACIS)*, 2024, pp. 1–6.

[17] "Accelerating Neural Network Training with Semi-Structured (2:4) Sparsity," Jun. 2024. [Online]. Available: https://pytorch.org/blog/accelerating-neural-network-training/

[18] J. Hruska, "The final ISA showdown: Is ARM, x86, or MIPS intrinsically more power efficient?" Aug. 2014. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/the-final-isa-showdown-is-arm-x86-or-mips-intrinsically-more-power-efficient

[19] L. Wang, H. Jia, L. Xu, C. Wei, K. Li, X. Jiang, and Y. Zhang, "Vnec: A vectorized non-empty column format for spmv on cpus," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 14–25.

[20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[21] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[22] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation." [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf

[23] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[24] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, A. Marfatia, and H. Kim, "Copernicus: Characterizing the performance implications of compression formats used in sparse workloads," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 1–12.

[25] "Second Generation Intel® Xeon® Scalable Processors." [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/xeon/2nd-gen-xeon-scalable-processors-brief.html