Low Precision and Efficient Programming Languages for Sustainable AI: Final Report for the Summer Project of 2024

João Vitor de Oliveira Silva,[1,3] Tokey Tahmid,[2,3] and Weslley da Silva Pereira[3]

1 University of Colorado Denver
2 University of Tennessee Knoxville
3 National Renewable Energy Laboratory

NREL is a national laboratory of the U.S. Department of Energy
Office of Energy Efficiency & Renewable Energy
Operated by the Alliance for Sustainable Energy, LLC

This report is available at no cost from the National Renewable Energy Laboratory (NREL) at www.nrel.gov/publications.

Contract No. DE-AC36-08GO28308

Technical Report
NREL/TP-2C00-90910
September 2024

Low Precision and Efficient Programming Languages for Sustainable AI: Final Report for the Summer Project of 2024

João Vitor de Oliveira Silva,[1,3] Tokey Tahmid,[2,3] and Weslley da Silva Pereira[3]

1 University of Colorado Denver
2 University of Tennessee Knoxville
3 National Renewable Energy Laboratory

Suggested Citation
de Oliveira Silva, João Vitor, Tokey Tahmid, and Weslley da Silva Pereira. 2024. *Low Precision and Efficient Programming Languages for Sustainable AI: Final Report for the Summer Project of 2024*. Golden, CO: National Renewable Energy Laboratory. NREL/TP-2C00-90910. https://www.nrel.gov/docs/fy24osti/90910.pdf.

Technical Report
NREL/TP-2C00-90910
September 2024

Low Precision and Efficient Programming Languages for Sustainable AI

Final report for the summer project of 2024

João Vitor de Oliveira Silva[1,2]   Tokey Tahmid[1,3]
Mentor: Weslley da Silva Pereira[1]

[1]Computational Science Center, National Renewable Energy Laboratory
[2]Department of Mathematical and Statistical Sciences, University of Colorado Denver
[3]Electrical Engineering and Computer Science, University of Tennessee Knoxville

1 Introduction

This document contains all relevant material generated during the authors' summer internship at NREL in 2024. This report shows how to improve the energy efficiency of a few code samples by using low-precision data types combined with mixed-precision algorithms. The main applications considered here are (i) linear system solvers using mixed precision, and (ii) neural networks using mixed precision. This report also discusses how programming languages affect the energy consumption of algorithms, energy metrics for a code and energy measurement tools, and the currently available software and hardware infrastructure that support low- and mixed-precision computation.

The remainder of this report is organized as follows. Section 2 defines low and mixed precision. Section 3 defines energy metrics and tools to measure energy, and discusses the effects of programming languages in the energy consumption of code. Section 4 presents strategies, experiments, results, and analyses of mixed-precision training of neural networks. Section 5 discusses and presents experiments of mixed-precision strategies to solve symmetric positive definite linear systems, as well as its use in Gaussian Process Regression. Section 6 presents the final discussion and suggestions for future work.

2 Mixed and Low precision

Due to finite memory, all computer software run at finite precision. The most well-known numeric types are the integers and the IEEE 754 single-precision (FP32) and double-precision floating-point (FP64) types. Integers are represented with 32 or 64 bits in modern 64-bit architectures, while FP32 and FP64 always have 32 and 64 bits, respectively. Based on this usual scenario, we consider low-precision types to be the ones whose size varies between 1 bit and 31 bits. Some examples of low-precision floating-point types are: TF32 (19 bits) [1], FP16, BF16 [2], E4M3 (8 bits) and E5M2 (8 bits) [10].

Two of the most used floating-point low-precision types are the BF16 and the FP16. Those are 16-bit data types where the mantissa has 7 bits in the former and 10 bits in the latter. They both use 1 bit to control the sign in front of the number. One attractive aspect of the BF16 type is that it can be easily converted to the FP32 data type since they both have the same number of bits in the exponent. Here is an example of a `C++` class that emulates a BF16 and the cast operations to and from FP32. The cast operations use bit rotation, which is a common instruction implemented in hardware.

```cpp
struct bfloat16
{
```

[1]https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/
[2]https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/

```
    uint16_t data;

public:
    bfloat16()
    {
        data = 0;
    }
    // cast to float
    operator float() const
    {
        uint32_t proc = ((uint32_t) data) << 16;
        return *reinterpret_cast<float *>(&proc);
    }
    // cast to bfloat16
    bfloat16 &operator=(float float_val)
    {
        data = (*reinterpret_cast<uint32_t *>(&float_val)) >> 16;
        return *this;
    }
};
```

Hardware operations, like moving bits and adding numbers, naturally use less bits when applied to smaller types. Thus, hardware instructions operating on low-precision types consume less or the same energy and time than their high-precision equivalents, and may even halve memory requirements in recent GPUs [13]. Mixed-precision operations and algorithms use two or more arithmetic precision types. In combination with low-precision types, mixed-precision computations can achieve high efficiency with minor or no accuracy loss for sufficiently large problems. One example is the training of neural networks using FP16 or BF16, where mixed-precision approaches have been shown to retain high accuracy through the following techniques: (I) maintaining a separate copy of weights in FP32 for weight and gradient updates, (II) scaling up the loss function to prevent underflow when using FP16, and (III) performing FP16 or BF16 arithmetic for computations while accumulating values in FP32 (see details in [13]). Another example is the solution of linear systems, in which a common mixed-precision approach is to first solve the system in lower precision, and then use iterative refinements to recover a highly-accurate solution [7]. Sections 4 and 5 discuss these examples in details.

The efficiency of the two examples mentioned above, and many others, rely on hardware support for mixed-precision matrix-matrix multiplication instructions. In this work, we considered the Intel AMX [3] technology available on Intel CPUs with the Sapphire Rapids architecture, and the Nvidia Tensor Cores [14] available since the Volta architecture on Nvidia GPUs.

2.1 Intel CPUs (Intel AMX) Overview

Intel Advanced Matrix Extensions (AMX) is a processing unit designed to accelerate matrix-matrix multiplications using low-precision data types on input, while converting outputs to higher precision for improved accuracy. Intel AMX accelerators feature:

1. Tiles: New, expandable 2D register files, each 1kB in size.
2. TMUL (Tile Matrix Multiply): Instructions operating on these tiles to perform matrix-multiply computations in mixed precision.

Note that one may easily check if the AMX technology is available on an Intel CPU by using the command '`lscpu | grep amx`' on Unix. If enabled, the output should include the flags: `amx_bf16`, `amx_tile`, and `amx_int8` [4].

[3]`https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html`

[4]We performed several experiments with linear algebra operations on Kestrel with Intel AMX, reported on `https://github.nrel.gov/wdasilv/test_lowprecision`

2.2 Nvidia GPUs (Tensor Cores) Overview

Nvidia's Tensor Cores are specialized for accelerating mixed-precision neural network training by performing matrix-matrix multiplications in lower precision and accumulations in low or high precision [14]. They operate on four 4x4 matrices concurrently, enhancing throughput significantly as follows:

- 16-bit multiplication: Multiplies two 16-bit 4-by-4 matrices to produce a 16-bit 4-by-4 matrix.
- 32-bit Accumulation: Accumulates the resulting 16-bit 4-by-4 matrix into an 32-bit 4-by-4 matrix.

The Tensor Cores can perform up to 64 floating-point operations per clock cycle, providing substantial throughput benefits over FP32 computations (up to 8x more) [14]. The main mathematical operation in the Tensor Cores is the matrix-matrix multiplication with accumulation, i.e., $D = A \times B + C$ where A and B are half-precision 4-by-4 matrices C and D can be half or single precision 4-by-4 matrices. Tensor Cores are significantly faster than double precision (FP64) and single precision (FP32). For instance, the theoretical peak performance of V100 Tensor Cores is approximately 120 TFLOPS which is 10x faster than double precision (FP64) and 4x faster than single precision (FP32) [14].

In order to maximize performance for 16-bit floating-point types, there are recommendations:

- matrix multiplication: dimensions (M, N, K) multiples of 8.
- convolution: input and output channels multiples of 8.
- mixed-precision training: mini-batch size, linear layer dimensions, convolution layer channel counts, vocabulary size, and sequence length multiples of 8.

Moreover, it is recommended to decrease non-Tensor Core Work by optimizing it. Further details on Nvidia's Mixed Precision Tenchnology using the Nvidia Tensor Cores can be found in their user's guide, "Train with Mixed Precision" [14].

2.3 Hardware specification

For the experiments that follow, we used the HPC infrastructure here at NREL, which features CPU and GPU nodes and supports mixed-precision computations, allowing us to handle large-scale problems. Below, we list the configurations used on the two machines, Kestrel and ALIS Server.

1. Kestrel:
   - OS: Red Hat Enterprise Linux version 8.8 (Ootpa, Kernel Linux 4.18.0-477.10.1.el8_8.x86_64)
   - CPU allocation:
     - Intel® Xeon® Platinum 8470QL (104 Cores).
     - No GPUs.
     - RAM: 256 GB
   - GPU allocation:
     - 1x AMD™ EPYC 9554 (64 Cores, up to 3.75 GHz).
     - 1x NVIDIA H100 HBM3 (80GB, 528 Tensor Cores with a theoretical peak performance of approximately 1000 TFLOPS [5].)
     - RAM: 80 GB
2. ALIS Server:
   - OS: Rocky Linux 9.4 (Blue Onyx, Kernel Linux 5.14.0-427.22.1.el9_4.x86_64)
   - RAM: 1.0TB
   - CPU runs: 1× AMD EPYC™ 7443 (24 cores, up to 4 GHz)

- GPU runs: 1× NVIDIA A100-SXM4 (40GB, 432 Tensor Cores with a peak performance of up to 312 TFLOPS [5])

The detailed specification for the Kestrel HPC machine is in `https://www.nrel.gov/hpc/kestrel-computing-system.html`.

Running some experiments on this machine provided insights into how architectural differences can impact overall energy consumption and processing time, as well as reveal memory limitations when handling very large problems.

3 Energy measurements

Computer components all spend energy. The specification of how much energy is spent is usually the range of power (W) the component operates. To measure energy, one must record all the history of power during the time of execution, and then compute the area behind the curve. This is most accurate (and ideal) way of measuring the energy, which is approximated by software like Intel RAPL (Running Average Power Limit) [6] and NVML (NVIDIA Management Library) [7]. In another extreme, one may rely on the formula below for total energy consumption (E):

$$E = \sum_{e \in \mathcal{E}} \sum_{d \in \mathcal{D}} P_{\text{avg},d,e} T_e \tag{1}$$

where $\mathcal{E}$ represents different components, $\mathcal{D}$ denotes different time intervals, $P_{\text{avg},d,e}$ is the average power consumed by component e during interval d, and T_e is the total time for which component e is active. Most tools listed in the following section use both approaches.

Besides the energy measured in kWh, we may also want to consider additional information to evaluate how green (environment friendly) a deployed application is. The Power Usage Effectiveness (PUE) is a common metric that reflects how efficiently a computer data center uses energy. Specifically, it is the ratio between the total energy spent in the infrastructure over the energy used by the computer itself. A lower PUE indicates a more efficient data center. Additionally, considering the carbon dioxide equivalent (CO2) emissions is crucial for evaluating the carbon footprint of the energy consumed. Incorporating these metrics ensures a comprehensive assessment of both the energy efficiency and the environmental friendliness of a deployed application.

Other relevant metrics include the Energy-Delay Product (EDP) [11], which is the product of the total energy consumed times the execution time of an application. EDP is particularly useful in balancing performance and energy efficiency, as it helps identify solutions that minimize energy consumption without significantly compromising performance.

3.1 Available Energy Measurement Tools and Their Comparison

In the review paper [2], the authors describe several energy measurement tools used to estimate the energy consumption and carbon footprint of training deep learning models. These tools provide various levels of detail and accuracy in estimating the energy consumption and carbon emissions of training deep learning models. Based on that reference and our evaluation, we opted to use the tool CodeCarbon. Regardless, we provide a summary of the characteristics of each of the considered tools because we understand that each application may have its appropriate corresponding tool. The tools described in the paper are:

1. **Green-Algorithms (GA)**

   - Type: Online calculator and server-side tool.
   - Methodology: Uses the model of CPU/GPU to pull the corresponding TDP from a database or allows manual input. Memory consumption is estimated based on available memory.

[5] `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`

[6] `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html`

[7] `https://developer.nvidia.com/management-library-nvml`

- Strengths: Provides detailed breakdowns by CPU, GPU, and memory.
- Limitations: Default usage factor is 100% if not provided by the user, which may overestimate consumption.
- Hardware Compatibility: Compatible with various CPUs and GPUs.
- Real-time Carbon Intensity: Not supported.
- Overhead and Accuracy: Lower overhead due to not using real-time data.

2. **CodeCarbon (CC)**
   - Type: Embedded package.
   - Methodology: Uses RAPL files or Power Gadget for CPU energy consumption (Intel CPUs) and pynvml for GPU (NVIDIA GPUs). Also supports manual TDP input.
   - Strengths: Good accuracy when using direct measurements.
   - Limitations: Requires root access for RAPL files, and does not measure memory consumption by default.
   - Usage Factor: Automatically tracked through sensors.
   - Hardware Compatibility: Limited to Intel CPUs and NVIDIA GPUs.
   - Real-time Carbon Intensity: Not supported.
   - Overhead and Accuracy: Moderate overhead with accurate measurements when sensors are available.
3. **Eco2AI (E2)**
   - Type: Embedded package.
   - Methodology: Uses the model of CPU/GPU to pull TDP values from a list. For memory, it uses psutil to measure usage.
   - Strengths: Flexible with process-specific tracking.
   - Limitations: May have higher overhead compared to other tools.
   - Usage Factor: Uses os and psutil python modules.
   - Hardware Compatibility: Compatible with various CPUs and GPUs.
   - Real-time Carbon Intensity: Not supported.
   - Overhead and Accuracy: Higher overhead due to extensive querying of system resources.
4. **CarbonTracker (CT)**
   - Type: Embedded package.
   - Methodology: Uses RAPL files for CPU energy consumption (Intel CPUs) and pynvml for GPU (NVIDIA GPUs).
   - Strengths: Supports real-time fetching of carbon intensity for specific regions.
   - Limitations: Limited to Intel CPUs and NVIDIA GPUs.
   - Usage Factor: Automatically tracked through sensors.
   - Hardware Compatibility: Limited to Intel CPUs and NVIDIA GPUs.

- Real-time Carbon Intensity: Supported for specific regions.
- Overhead and Accuracy: Moderate overhead with accurate real-time data.

5. **Experiment-Impact-Tracker (EIT)**
    - Type: Embedded package.
    - Methodology: Uses RAPL files for CPU energy consumption and nvidia-smi for GPU (NVIDIA GPUs).
    - Strengths: Provides detailed usage factors.
    - Limitations: Limited to Intel CPUs and NVIDIA GPUs, and requires root access.
    - Usage Factor: Uses psutil python module.
    - Hardware Compatibility: Limited to Intel CPUs and NVIDIA GPUs.
    - Real-time Carbon Intensity: Supported for specific regions.
    - Overhead and Accuracy: Higher overhead due to detailed usage tracking.
6. **MLCO2**
    - Type: Online calculator.
    - Methodology: Uses the model of CPU/GPU to pull TDP values from a list. Assumes maximum load for GPU.
    - Strengths: Simplified approach, easy to use.
    - Limitations: May overestimate energy consumption due to assuming maximum load.
    - Usage Factor: Assumes 100% usage.
    - Hardware Compatibility: Compatible with various CPUs and GPUs.
    - Real-time Carbon Intensity: Not supported.
    - Overhead and Accuracy: Lower overhead with potential overestimation.
7. **Cumulator (CMLTRs)**
    - Type: Embedded package.
    - Methodology: Uses the model of CPU/GPU to pull TDP values from a list. Only measures CPU or GPU at a time.
    - Strengths: Simple setup.
    - Limitations: Does not measure memory consumption and assumes maximum load for GPUs.
    - Usage Factor: Assumes 100% usage.
    - Hardware Compatibility: Compatible with various CPUs and GPUs.
    - Real-time Carbon Intensity: Not supported.
    - Overhead and Accuracy: Lower overhead with potential overestimation.

3.2 Energy consumption and programming languages

In software development, the choice of a programming language is often influenced by factors such as ease of use, the availability of libraries, and compatibility with other technologies. For the scientific community, ease of use is especially motivating as it enables researchers to quickly prototype and test complex algorithms without having difficulties with complicated syntax or steep learning curves. While the convenience of a language may accelerate certain scientific discoveries, overlooking energy efficiency can lead to missed opportunities for optimizing performance and reducing the environmental impact of computational research.

The energy efficiency of distinct programming languages is outlined in a comparison along a set of problems [15]. This set of problems is available at a Github repository [8] originally known as Computer Language Benchmark Game (CLBG). The authors show that the choice of compiled languages usually consume less energy and require less computation time compared to using interpreted languages such as Python or Lua. Moreover, the relationship between energy and RAM memory storage is discussed, although, unlike the relationship between energy and time consumption which are more related, it has more to do with how it is represented and used during the application compared to the amount of memory used. The applications we will detail in the next sections are written in Python, and use packages like Pytorch and TensorFlow. The backends for those packages are usually libraries written in C and C++. Therefore, one should expect high energy efficiency even by only relying on scripts using those high-level Python packages.

Compiled pointer-based languages, such as C and C++, offer a high degree of control over system memory and execution, which often results in faster performance and lower energy consumption compared to memory-safe languages. This efficiency is due to the ability to perform direct memory access and manage resources with minimal overhead. However, this fine-grained control comes with significant security risks. Unlike memory-safe languages like Python or Java, which automatically handle memory management and enforce strict bounds checking, pointer-based languages allow direct manipulation of memory addresses. As addressed in a 2024 White House report [9], this may lead to vulnerabilities such as buffer overflows, where data written beyond the allocated memory buffer can corrupt adjacent memory or execute arbitrary code. Thus, while pointer-based languages provide powerful capabilities and are more energy efficient, they also require careful management and thorough testing to avoid security failures. The discussion about the qualities of different programming languages is always ongoing, and the scenario changes as the languages evolve and new languages appear.

4 Application on neural networks

4.1 Examples of mixed precision results on neural networks

4.1.1 Experimental Setup

We tested mixed precision (`mixed_bfloat16`) for training and inference on multiple AI applications at NREL. We did these tests on both CPUs and GPUs in the Kestrel HPC Cluster. For the CPU runs, we utilized the Intel AMX technology in the Intel® Xeon® Sapphire Rapids CPUs. And for the GPU runs, we utilized Nvidia's Tensor Cores in the Nvidia H100 GPUs. Detailed hardware specifications for Kestrel CPU and GPU nodes is given in section 2.3. The mixed-precision technology was tested for both TensorFlow and PyTorch backends. All tests use the BF16 type described in Section 2.

In our methodology, we run each test multiple times with a set seed for reproducible accuracy results across all runs. Then we take the mean and standard deviation of the different performance metrics of these runs. We used the CodeCarbon [3] tool to measure the different energy efficiency metrics such as total energy consumption, carbon emission, and EDP which is computed based on energy consumption and execution time. Finally, we compared the results between mixed precision and single precision for all our test runs.

For TensorFlow models, we tested the TF-MELT (v0.4.1) [10] application using Python (v3.11.9) and TensorFlow (v2.15.1), and compared between mixed precision (`mixed_bfloat16`) and single precision (float32). For PyTorch, we tested the PT-MELT (v0.1.1) [11] application with Python (v3.11.9) and PyTorch (v2.3.1+cpu)

[8] https://github.com/greensoftwarelab/Energy-Languages
[9] https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
[10] `https://github.com/NREL/tf-melt`
[11] `https://github.com/nrel/pt-melt`

for CPU and PyTorch (v2.4.0) for GPUs for mixed-precision tests. The python scripts, notebooks, result outputs, job scripts, and modules configuration scripts for all these tests can be found in the mixed-precision-tests [12] repository. In the notebooks, we particularly tested the mixed-precision training of Artificial Neural Network (ANN), Residual Neural Network (ResNET), and Bayesian Neural Network (BNN) models for TF-MELT repository and ANN, and ResNET models for PT-MELT repository. We only tested inference for the BNN models in TF-MELT as of now. These trained models from our experiments can be found in the mixed-precision-models [13] repository at the NREL organization on Hugging Face.

We also ran tests with PINNs [14] and MMBO [15] (BNNs in Pytorch), although we did not develop too much in those directions. The few examples we ran with PINNs showed a moderate advantage of using mixed precision in the training process ($1 \sim 1.4$X speedup) and a proportional reduction in energy consumption. The tests with MMBO were inconclusive and need due to the lack of time to dedicate to them during the internship.

4.2 Experimental results

We tested each model with multiple mixed-precision training runs on both TensorFlow and PyTorch on CPUs and GPUs to compare the speedup performance and energy efficiency. The tables (1 to 8) below show a side by side comparison of mixed-precision vs single-precision on both CPU and GPUs for both TensorFlow and PyTorch runs. We give hints about the model architectures in the sequence.

Table 1: ResNet-50 Training Summary (TensorFlow)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time (s)	437.48	243.16	93.26	53.50
Loss	1.9178	1.9149	1.8958	1.9124
Consumed Energy (kWh)	0.03391	0.01903	0.01438	0.00799
Emissions (kgCO2)	0.03391	0.01903	0.00528	0.00294
EDP (kWh*s)	27.487	8.614	1.243	0.388
Speedup	(1.80 ± 0.01)X		(1.74 ± 0.11)X	
Emission Reduction	(43.82 ± 0.22)%		(44.30 ± 2.62)%	
EDP Reduction	(68.78 ± 0.24)%		(67.80 ± 3.30)%	
Accuracy Loss	-0.01%		-0.01%	

Table 2: ANN Training Summary (TensorFlow)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time (s)	4245.92	2686.40	571.69	245.62
Loss	1.0520	1.0569	1.0470	1.0469
Consumed Energy (kWh)	0.92489	0.58251	0.13171	0.04412
Emissions (kgCO2)	0.34046	0.21442	0.04848	0.01624
EDP (kWh*s)	3927.169	1565.666	75.297	10.960
Speedup	(1.58 ± 0.04)X		(2.37 ± 0.30)X	
Emission Reduction	(37.01 ± 1.83)%		(66.52 ± 2.89)%	
EDP Reduction	(60.11 ± 2.32)%		(85.46 ± 3.04)%	
Accuracy Loss	-0.01%		0.00%	

[12]https://github.nrel.gov/AI/mixed-precision-tests
[13]https://huggingface.co/NREL/mixed-precision-models
[14]https://github.com/NREL/PINNSTRIPES
[15]https://github.nrel.gov/itaylor/mmbo

Table 3: ResNet Training Summary (TensorFlow)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time (s)	4290.91	2725.49	571.83	245.03
Loss	0.6399	0.7515	0.6030	1.0930
Consumed Energy (kWh)	0.93432	0.58256	0.13444	0.04406
Emissions (kgCO2)	0.34392	0.21444	0.04949	0.01622
EDP (kWh*s)	4012.340	1590.043	76.881	10.911
Speedup	(1.58 ± 0.02)X		(2.37 ± 0.29)X	
Emission Reduction	(37.66 ± 0.59)%		(67.23 ± 2.78)%	
EDP Reduction	(60.41 ± 0.75)%		(85.81 ± 2.91)%	
Accuracy Loss	-0.26%		-0.81%	

Table 4: BNN Training Summary (TensorFlow)

Metric	CPU	
	FP32	BF16
Training Time (s)	1032.293	763.965
Loss	3397.716	3404.362
Consumed Energy (kWh)	0.22603	0.16657
Emissions (kgCO2)	0.08320	0.06132
EDP (kWh*s)	233.327	127.254
Speedup	(1.48 ± 0.20)X	
Emission Reduction	(25.85 ± 2.11)%	
EDP Reduction	(46.63 ± 2.25)%	
Accuracy Loss	-0.00%	

Table 5: BNN Inference Summary (TensorFlow)

Metric	CPU	
	FP32	BF16
Inference Time (s)	43.542	27.094
Consumed Energy (kWh)	0.00892	0.00528
Emissions (kgCO2)	0.003	0.002
EDP (kWh*s)	0.388	0.143
Speedup	(1.83 ± 0.16)X	
Emission Reduction	(45.40 ± 4.25)%	
EDP Reduction	(63.83 ± 0.00)%	

In Table 9, we show the difference in training times of the ANN model training in TensorFlow to observe the overhead of the CodeCarbon tool. We can see that in CPUs, the overhead is up to 2.74% which is not that significant. However, on the GPUs, the overhead is up to 43.77% which is a significant increase in training time due to overhead by the CodeCarbon tool.

4.2.1 Model Architectures Used in Experiments

In our multiple test runs we see some speedup and efficiency gain across all the models. For multiple runs of a particular model, we present the (mean ± standard deviation) of the speedup, emission reduction, and EDP reduction. For example, the speedup for the BNN models in Table 4, were (1.48 ± 0.20)X faster with mixed-precision using BF16 than single precision on CPUs. In this case, mixed-precision reduced carbon emissions by (25.85 ± 2.11)% and EDP by (46.63 ± 2.25)%.

These speedup results are due to the use of a relatively wider network for these models with a similarly

Table 6: ResNet-50 Training Summary (PyTorch)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time (s)	502.29	290.81	96.50	88.06
Loss	2.1860	2.1549	2.1719	2.1574
Consumed Energy (kWh)	0.10384	0.06049	0.01490	0.01296
Emissions (kgCO2)	0.03823	0.02227	0.00549	0.00477
EDP (kWh*s)	52.160	17.591	1.438	1.141
Speedup	(1.73 ± 0.02)X		(1.10 ± 0.01)X	
Emission Reduction	$(41.74 \pm 0.09)\%$		$(13.01 \pm 0.62)\%$	
EDP Reduction	$(66.27 \pm 0.41)\%$		$(20.61 \pm 1.10)\%$	
Accuracy Loss	0.01%		0.01%	

Table 7: ANN Training Summary (PyTorch)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time (s)	2892.84	1290.42	437.93	102.49
Loss	1.0039	1.0126	1.0039	1.0039
Consumed Energy (kWh)	0.63355	0.27331	0.08913	0.01809
Emissions (kgCO2)	0.23321	0.10060	0.03281	0.00665
EDP (kWh*s)	1832.751	352.711	39.034	1.853
Speedup	(2.25 ± 0.02)X		(4.27 ± 0.02)X	
Emission Reduction	$(56.86 \pm 0.40)\%$		$(79.70 \pm 0.05)\%$	
EDP Reduction	$(80.75 \pm 0.41)\%$		$(95.25 \pm 0.04)\%$	
Accuracy Loss	-0.01%		-0.00%	

Table 8: ResNet Training Summary (PyTorch)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time (s)	4965.53	3036.30	560.16	200.27
Loss	0.6199	0.6563	0.7290	0.7471
Consumed Energy (kWh)	1.07446	0.63315	0.13514	0.04113
Emissions (kgCO2)	0.39551	0.23306	0.04974	0.01514
EDP (kWh*s)	5335.283	1925.091	75.698	8.238
Speedup	(1.64 ± 0.06)X		(2.79 ± 0.02)X	
Emission Reduction	$(41.06 \pm 2.29)\%$		$(69.56 \pm 0.03)\%$	
EDP Reduction	$(63.91 \pm 2.88)\%$		$(89.12 \pm 0.07)\%$	
Accuracy Loss	-0.06%		-0.02%	

Table 9: ANN Training Time with and without Tracker (TensorFlow)

Metric	CPU		GPU	
	FP32	BF16	FP32	BF16
Training Time with Tracker (s)	4273.236	2625.727	571.717	276.399
Training Time without Tracker (s)	4324.790	2555.590	537.939	192.262
Extra Time Taken by Tracker (%)	-1.19%	2.74%	6.28%	43.77%

wide number of features in the dataset. For example, for the ANN and ResNET models we are using 4000 for width (Neurons) with 50 for depth (layers) and for the BNN models we are using 4000 for width and 6 for depth. For the synthetic dataset we are using 4000 features and 4000 for batch size. One must use expensive matrix-matrix multiplications to be able to obtain speedup up out of Intel AMX mixed-precision technology.

4.2.2 Benefits in terms of Energy and Performance

With the use of mixed precision in Neural Network training, we observed mean speedup 2.05X with a standard deviation of 0.79 compared to single precision. With this speedup we see a reduced total energy consumption which leads to a significant amount of energy efficiency in comparison. For example, for a 1.35X speedup in training we see 25.40% carbon emission reduction and 44.66% EDP reduction. For a 1.81X speedup in training we see 44.04% carbon emission reduction and 69.02% EDP reduction. The gain in energy efficiency and speedup do not compromise accuracy in most cases. Across all our tests, we observed a mean accuracy loss of -0.19% with a standard deviation of 0.29. The results are shown in details in section 4.2.8.

4.2.3 Predictions After Training

We also tested inference/predictions after training the models in TF-MELT. For mixed-precision inference, we set the same flags and options as training in both TensorFlow and Pytorch. In mixed-precision inference on Intel CPUs, we see slightly better speedup performance and efficiency. For example, the speedup for training one of the BNN models was 1.35X, compared to 1.65X speedup for inference on the trained BNN model. You can find the rest of the inference results at [16].

4.3 Configurations to Use

To attain this kind energy efficiency and performance, mixed precision needs to be used with care because it depends on a couple of factors. These are:

(I) First, the mixed-precision technology needs to be enabled appropriately with the correct flags/options for the correct hardware and software backend. For example, when training the model on an Intel CPU with Intel AMX, the following environment variable must be set to enable mixed precision with the AMX technology:

```
os.environ["ONEDNN_MAX_CPU_ISA"] = "AVX512_CORE_AMX"
```

For the TensorFlow backend, mixed-precision (`mixed_bfloat16`) can be used using the following code:

```
tf.config.optimizer.set_experimental_options({
'auto_mixed_precision_onednn_bfloat16':True
})
```

There are five different approaches for enabling mixed precision with TensorFlow that can be found in detail at [17].

For PyTorch, similar instruction can be found at this tutorial by Intel [18]. So, for PyTorch we use:

```
ipex.optimize(model, optimizer=optimizer, dtype=torch.bfloat16)
# Inside training loop
with torch.cpu.amp.autocast():
```

For Nvidia GPUs, mixed precision is automatically enabled when the code utilizes tensor cores. In the software backend, with TensorFlow, mixed-precision is enabled by setting:

```
tf.keras.mixed_precision.set_global_policy('mixed_bfloat16')
```

[16] `https://github.nrel.gov/AI/mixed-precision-tests`

[17] `https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-automixedprecisionmkl.html`

[18] `https://www.intel.com/content/www/us/en/developer/articles/technical/accelerate-with-intel-extension-for-pytorch.html`

For PyTorch, one must use:

```
with torch.cpu.amp.autocast():
```

(II) Then, the neural network configurations need to set/adjusted appropriately to get the performance. For example, with the Intel AMX technology, relevant speedup is usually observed with big matrices, e.g., 4000-by-4000 and above. So, ideally the neural network width and training batch size need to be equivalently large to get the speedup with neural network training.

Detailed configuration options for both CPU and GPU for TensorFlow and PyTorch are described in section 4.2.7.

4.4 Mixed-Precision Under the Hood

The underlying mechanisms of mixed-precision computing can be elucidated through two main perspectives: hardware and software. The hardware perspective is already explained in sections 2.1 and 2.2. Here, we explain the software side of how mixed precision works under the hood based on the literature [13].

I. Maintaining a Separate Copy of Weights in FP32

During mixed-precision training, weights, activations, and gradients are stored in FP16, but a separate copy of the weights is maintained in FP32. This FP32 copy is updated during the optimizer step, while an FP16 copy is used for forward and backward passes. This approach prevents the loss of small updates and maintains accuracy.

II. Scaling of the Loss Function to Prevent Underflow (FP16 only)

Loss scaling is employed to prevent small gradient values from becoming zeros due to the limited range of FP16. By scaling up the loss before backpropagation, gradient values are shifted to a representable range for FP16. Gradients are then unscaled before the weight update to maintain correct update magnitudes, preserving relevant gradient values and preventing training divergence.

III. Performing 16-bit Arithmetic with FP32 Accumulation

During training, matrix multiplications and large reductions (e.g., in batch normalization and softmax layers) are accumulated into FP32 values before conversion back to 16-bit format. This ensures numerical stability and accuracy, as low precision is used for point-wise operations that are typically less sensitive to arithmetic precision and are memory-bandwidth limited.

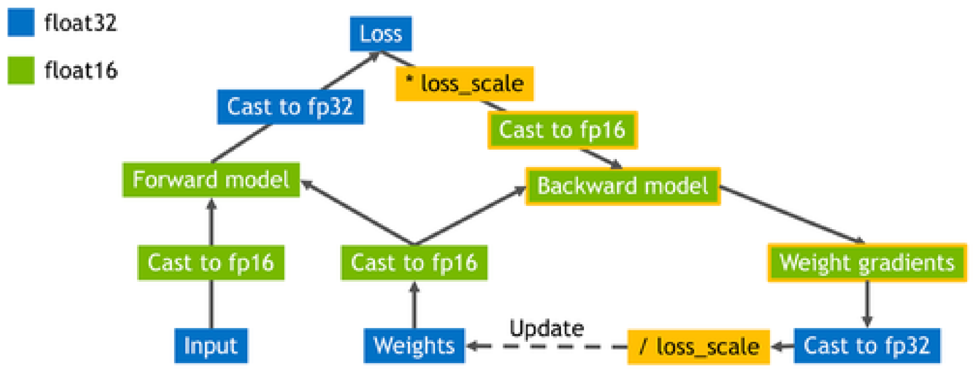


Figure 1: Neural network training loop in mixed precision. [Source: `https://www.youtube.com/watch?v=i1fIBtdhjIg`]

Figure 1 shows the under-the-hood pipeline of automatic mixed precision from Nvidia [19] that utilize these three techniques discussed above to maintain accuracy while optimizing performance. Yellow boxes are not necessary when using the BF16 data type.

4.5 What package should I use? PyTorch vs TensorFlow

Though both PyTorch and TensorFlow have mixed-precision support in their backend, the implementation is vastly different and, in some cases, it is not just the difference in syntax of code for implementation but also how it is implemented under the hood. In terms of the difficulty in the implementation, TensorFlow seems to have more ways to do mixed-precision and it is difficult to figure out which approach would be appropriate for the use case other than testing out all the approaches. They have five different approaches as mentioned in section 4.2.3.

These approaches also depend on the hardware (CPU/GPU) for both TensorFlow and PyTorch. For TensorFlow the difference in CPU and GPU implementation are completely two different approaches (under the hood) and for PyTorch this difference is one line of code between the CPU and GPU implementation.

Below are the differences of CPU and GPU implementation for both TensorFlow and PyTorch:

TensorFlow CPU:

```python
os.environ["ONEDNN_MAX_CPU_ISA"] = "AVX512_CORE_AMX"
if dataType == "bf16":
    tf.config.optimizer.set_experimental_options({
    'auto_mixed_precision_onednn_bfloat16':True
    })
else:
    tf.config.optimizer.set_experimental_options({
    'auto_mixed_precision_onednn_bfloat16':False
    })
    tf.keras.mixed_precision.set_global_policy('float32')
```

TensorFlow GPU:

```python
if dataType == "bf16":

    tf.keras.mixed_precision.set_global_policy('mixed_bfloat16')
else:

    tf.keras.mixed_precision.set_global_policy('float32')
```

PyTorch CPU:

```python
# Set environment at the start
os.environ["ONEDNN_MAX_CPU_ISA"] = "AVX512_CORE_AMX"
import intel_extension_for_pytorch as ipex
if "bf16" == dataType:
    model, optimizer = ipex.optimize(
    model, optimizer=optimizer, dtype=torch.bfloat16
    )
else:
    model, optimizer = ipex.optimize(model, optimizer=optimizer)

scaler = torch.amp.GradScaler(device.type, enabled=True)

# Inside main training loop
if "bf16" == dataType:
    with torch.autocast(
```

[19]https://www.youtube.com/watch?v=i1fIBtdhjIg

```
    device_type=device.type, dtype=torch.bfloat16, enabled=True
    ):
        output = model(data)
        loss = criterion(output, target)
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
else:
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
```

PyTorch GPU:

```
scaler = torch.amp.GradScaler(device.type, enabled=True)
if "bf16" == dataType:
    with torch.autocast(
    device_type=device.type, dtype=torch.bfloat16, enabled=True
    ):
        output = model(data)
        loss = criterion(output, target)
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
else:
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
```

The TensorFlow's mixed-precision GPU implementation does not work with the TensorFlow-probability library [20], so we could not test TF-MELT BNNs on GPUs.

4.5.1 PyTorch Vs. TensorFlow, Which is Greener?

In this section, we compare our results from the tables presented in section 4.2. We compare the results between PyTorch and TensorFlow for each of the models (ResNET50, ANN, and ResNet).

ResNET-50 Model: If we compare the energy efficiency of TensorFlow and PyTorch for training the ResNET-50 model in Table 1 and 6, TensorFlow shows a clear advantage in most metrics. On CPUs, TensorFlow demonstrates lower energy consumption and better EDP values. For instance, in FP32 precision, TensorFlow consumes 0.03391 kWh compared to PyTorch's 0.10384 kWh, and, in BF16 precision, TensorFlow consumes 0.01903 kWh compared to PyTorch's 0.06049 kWh. The EDP for TensorFlow in FP32 precision is 27.487 kWh*s, whereas PyTorch's is 52.160 kWh*s, and, in BF16 precision, TensorFlow's EDP is 8.614 kWh*s compared to PyTorch's 17.591 kWh*s. When training on GPUs, TensorFlow continues to outperform PyTorch in energy consumption and EDP. For FP32 precision, TensorFlow consumes 0.01438 kWh compared to PyTorch's 0.01490 kWh, and, in BF16 precision, TensorFlow consumes 0.00799 kWh compared to PyTorch's 0.01296 kWh. Similarly, TensorFlow's EDP in FP32 precision is 1.243 kWh*s, significantly lower than PyTorch's 1.438 kWh*s, and, in BF16 precision, TensorFlow's EDP is 0.388 kWh*s compared to PyTorch's 1.141 kWh*s. Additionally, TensorFlow achieves greater emission reductions and EDP reductions across both CPU and GPU settings. Also, TensorFlow achieves an EDP reduction of 68.78% in FP32 precision and 67.80% in BF16 precision on GPUs, compared to PyTorch's 20.61% and 11.10% respectively. Overall, TensorFlow demonstrates a clear advantage over PyTorch in terms of energy efficiency for ResNet-50 model training, making it the greener choice in this comparison.

[20] `https://github.com/TensorFlow/probability/issues/1315`

ANN Model: If we compare the energy efficiency for training the ANN model in Table 2 and 7, and now the clear choice is PyTorch which demonstrates superior performance in terms of energy efficiency. Specifically, when training on CPUs, PyTorch consumes significantly less energy than TensorFlow. For example, in FP32 precision, PyTorch's energy consumption is 0.63355 kWh compared to TensorFlow's 0.92489 kWh, and, in BF16 precision, it is 0.27331 kWh for PyTorch compared to 0.58251 kWh for TensorFlow. This trend is consistent when using GPUs as well, where PyTorch again outperforms TensorFlow. In FP32 precision, PyTorch consumes 0.08913 kWh compared to TensorFlow's 0.13171 kWh, and, in BF16 precision, PyTorch consumes 0.01809 kWh compared to TensorFlow's 0.04412 kWh. Furthermore, PyTorch also shows a significant advantage in terms of EDP, which combines both energy consumption and training time. On CPUs, PyTorch's EDP in FP32 precision is 1832.751 kWh*s, much lower than TensorFlow's 3927.169 kWh*s. In BF16 precision, PyTorch's EDP is 352.711 kWh*s, compared to TensorFlow's 1565.666 kWh*s. On GPUs, the EDP for PyTorch in FP32 precision is 39.034 kWh*s, significantly lower than TensorFlow's 75.297 kWh*s, and, in BF16 precision, PyTorch's EDP is 1.853 kWh*s, while TensorFlow's is 10.960 kWh*s. Additionally, PyTorch achieves greater emission reductions and EDP reductions across both CPU and GPU settings. Overall, PyTorch demonstrates a clear advantage over TensorFlow in terms of both energy efficiency and EDP, making it the greener choice for ANN model training.

ResNet Model: Finally, we compare the energy efficiency for the ResNet model in Table 3 and Table 8. PyTorch again shows a significant advantage in both energy efficiency and EDP. On CPUs, PyTorch demonstrates lower energy consumption and better EDP values. For instance, in FP32 precision, PyTorch consumes 1.07446 kWh compared to TensorFlow's 0.93432 kWh, and, in BF16 precision, it consumes 0.63315 kWh compared to TensorFlow's 0.58256 kWh. The EDP for PyTorch in FP32 precision is 5335.283 kWh*s, whereas TensorFlow's is 4012.340 kWh*s, and, in BF16 precision, PyTorch's EDP is 1925.091 kWh*s compared to TensorFlow's 1590.043 kWh*s. When training on GPUs, PyTorch continues to outperform TensorFlow in energy consumption and EDP. For FP32 precision, PyTorch consumes 0.13514 kWh compared to TensorFlow's 0.13444 kWh, and, in BF16 precision, it consumes 0.04113 kWh compared to TensorFlow's 0.04406 kWh. PyTorch's EDP in FP32 precision is 75.698 kWh*s, compared to TensorFlow's 76.881 kWh*s, and, in BF16 precision, PyTorch's EDP is 8.238 kWh*s compared to TensorFlow's 10.911 kWh*s. Furthermore, PyTorch achieves greater emission reductions and EDP reductions across both CPU and GPU settings. Specifically, PyTorch achieves an EDP reduction of 63.91% in FP32 precision and 89.12% in BF16 precision on GPUs, compared to TensorFlow's 60.41% and 85.81% respectively. Overall, PyTorch demonstrates a clear advantage over TensorFlow in terms of energy efficiency and EDP for ResNet model training, making it the greener choice in this case.

Thus, out of the three models we compared, PyTorch was the more greener application in the case of two of those three models.

5 Application on linear system solvers

5.1 SPD linear system solvers

The solution of linear systems $Ax = b$, where A is a symmetric positive definite (SPD) matrix, is crucial for numerous scientific applications. Examples include the Numerical Solution of PDEs, Quadratic Programming, Image Processing, and Gaussian Process Regression (GPR) [9, 6, 17, 16]. As the size of the matrix increases, solving such systems becomes more computationally demanding, particularly when A lacks a special structure such as sparsity patterns. To address this challenge for large-scale problems, various methods have been developed to achieve efficient solutions within feasible time frames. These methods often leverage the properties of SPD matrices, such as their positive definiteness and symmetry, to optimize algorithms like Cholesky decomposition or iterative solvers tailored for SPD matrices.

A promising approach involves a mixed-precision solver specialized for SPD dense matrices [8]. This method operates in two distinct stages: initially solving the linear system using FP16 Cholesky decomposition, and subsequently refining the solution through iterative refinement using the GMRES (Generalized Minimal RESidual) solver, which is iterative and Krylov subspace-based, optimized for GPUs. This new procedure is named Cholesky-based GMRES-IR. By harnessing the inherent capabilities of GPUs for FP16 computations, this algorithm demonstrates superior performance compared to conventional methods, particularly when dealing with exceptionally large matrices. This iterative refinement step is generally successful in recovering the original problem's accuracy, given that the initial low-precision solution is closer to the actual solution, thereby enhancing convergence and final accuracy of the computed solution. Figures 2 and 3 outline the computations performed on this new solver, when the target precision is FP64. It is worth mentioning that there is no accuracy loss in the final solution, since the iterative refinement step only stops when a FP64 accurate solution is obtained.

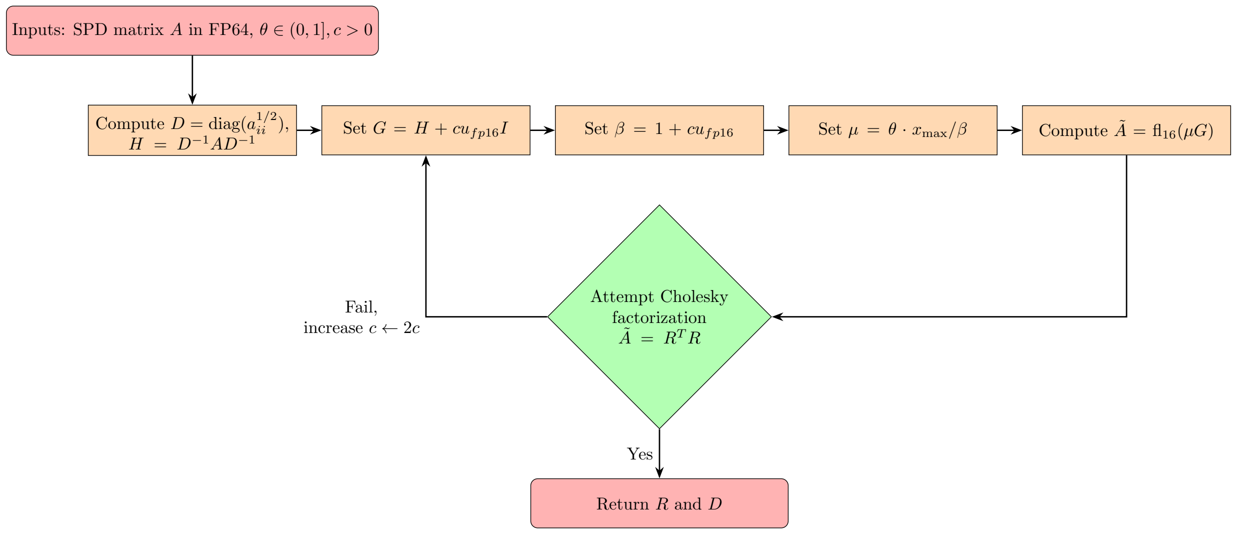

Inputs: SPD matrix A in FP64, $\theta \in (0,1], c > 0$
Compute $D = \mathrm{diag}(a_{ii}^{1/2})$, $H = D^{-1}AD^{-1}$
Set $G = H + cu_{fp16}I$
Set $\beta = 1 + cu_{fp16}$
Set $\mu = \theta \cdot x_{\max}/\beta$
Compute $\tilde{A} = \mathrm{fl}_{16}(\mu G)$
Attempt Cholesky factorization $\tilde{A} = R^T R$
Fail, increase $c \leftarrow 2c$
Yes
Return R and D

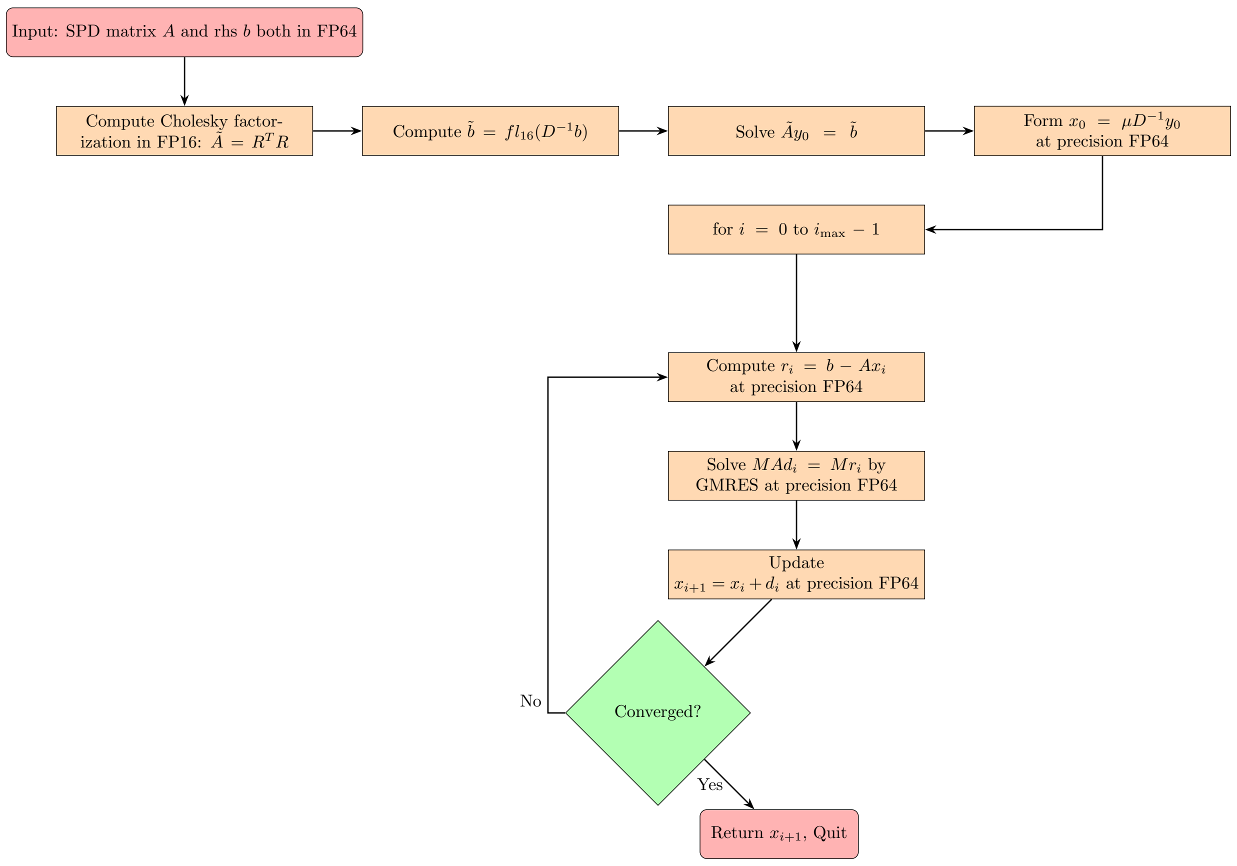
Figure 2: First stage of the SPD system mixed precision solver. It first preprocesses the matrix in order to reduce numerical rounding issues introduced by representing the system in lower precision and then finds its FP16 Cholesky factor.

A recent addition to the C++ MAGMA 2.7.2 library includes a SPD mixed-precision GPU solver based on this presented approach (`dshpov_gmres`). The difference consists on the First Stage, in which the factorization is performed in mixed precision by performing only matrix multiplications in FP16 and saving the final result in FP32. However, it is important to note that while MAGMA includes this advanced solver, the current implementation lacks the iterative selection of the parameter c. As of the latest updates, this solver has not been integrated into the PyTorch 2.5.0a0 MAGMA backend, thus limiting its availability within the PyTorch ecosystem. It is worth noting that the usual PyTorch backend for Nvidia GPUs, cuSOLVER, has a similar mixed-precision solver, although it is based on LU factorization (thus not specialized for SPD systems) and it is still not integrated into its PyTorch backend as well.

After modifying PyTorch source code so as to use the aforementioned solver from the MAGMA library, we performed experiments of solving linear systems to verify its performance gains. The code for these experiments are available at `https://github.nrel.gov/jdeoliv/Cholesky_magma_test`. First, we performed experiments similar to the ones performed in [1], i.e., we generated random SPD matrices with distinct spectrum profiles and 2-norm condition number $\kappa(A) = 10^5$. The experiments were executed in both ALIS (using 1x NVIDIA A100) and Kestrel (using NVIDIA H100) HPC machines. As depicted in Figure 4 benchmark, a performance increase is noticeable for $n \geq 16384$ and $n \geq 19483$ on ALIS and Kestrel, respectively. The difference in their corresponding GPU architectures relates to the matrix size threshold where the classical routines are surpassed. Due to memory limitations, we only performed the experiments on ALIS for sizes up to $n = 32768$.

We also performed the same experiments with a lower condition number on Kestrel to check its influence over the new solver. Since the GMRES iterative solver is used in it, the lower condition number could positive influence the computation time. Figure 5 indicate that there is not a noticeable difference compared to the results in Figure 4, suggesting that the method is somewhat robust to changes in its condition number.

As for energy consumption, we monitored with the CodeCarbon tracker if the new solver is capable of reducing it. Interestingly, even for matrix sizes where the mixed precision solver required more computation



Input: SPD matrix A and rhs b both in FP64
Compute Cholesky factorization in FP16: $\tilde{A} = R^T R$
Compute $\tilde{b} = fl_{16}(D^{-1}b)$
Solve $\tilde{A}y_0 = \tilde{b}$
Form $x_0 = \mu D^{-1}y_0$ at precision FP64
for $i = 0$ to $i_{\max} - 1$
Compute $r_i = b - Ax_i$ at precision FP64
Solve $MAd_i = Mr_i$ by GMRES at precision FP64
Update $x_{i+1} = x_i + d_i$ at precision FP64
Converged?
No
Yes
Return x_{i+1}, Quit

Figure 3: SPD system mixed precision solver. Using the FP16 Cholesky factor, a FP16 solution is computed and then refined up to the desired FP64 precision using iterative refinement with the GMRES method and the lower precision Cholesky factor as a preconditioner.

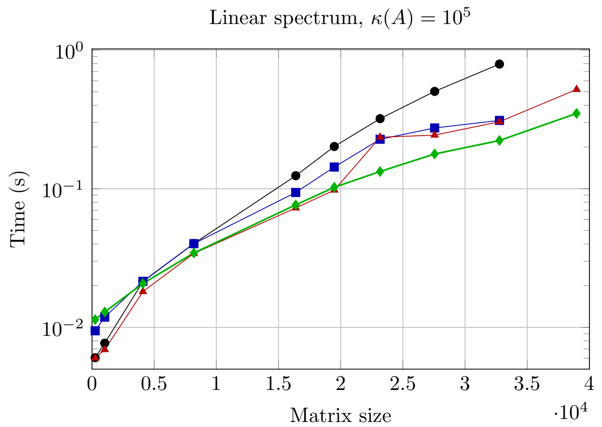
time, the total energy consumed was lower than the single precision equivalent. In order to consider both of these aspects simultaneously, we also considered how the EDP metric changes with respect to the linear system size. These results are detailed on Figure 6. It is possible to notice that for $n \geq 19483$, the new solver outperforms the original FP64 solver.

A downside of the implementation of this new mixed precision solver is that it is still not capable of performing GMRES-IR step for linear systems with multiple right-hand sides, i.e, when $b \in \mathbb{R}^{n\times k}$ with $k \geq 2$. This creates a layer of difficulty to use it at some applications such as GPR. There are plans on updating the Cholesky-based GMRES-IR to support multiple-right sides in the future according to MAGMA library developers. A more in deep investigation regarding the number of GMRES-IR iterations in the multiple right-hand sides systems will be vital to determine if the capabilities of the mixed-precision approach surpass the standard double precision solvers.

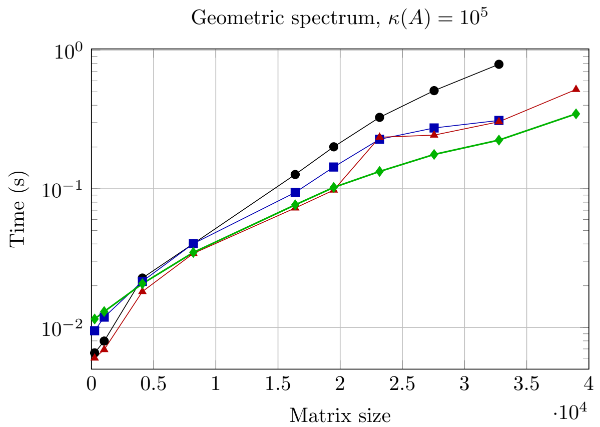
5.1.1 Application to Gaussian Process Regression

Even with the limitation of not being able to solve a SPD linear system with multiple right-hand sides, we applied the first stage of the solver routine to obtain a mixed-precision Cholesky decomposition $A = R^T R$ with a single precision input matrix. For that we used a routine on MAGMA that only performs the factorization using FP32 accumulation of FP16 matrix multiplications (`shpotrf`), in which a modification on PyTorch source code allowed its use on Python. We verified there was a speedup without a considerable loss of accuracy when solving the linear system, as depicted in Figure 7. These results motivated us to use this approach inside a Gaussian Process Regression problem.

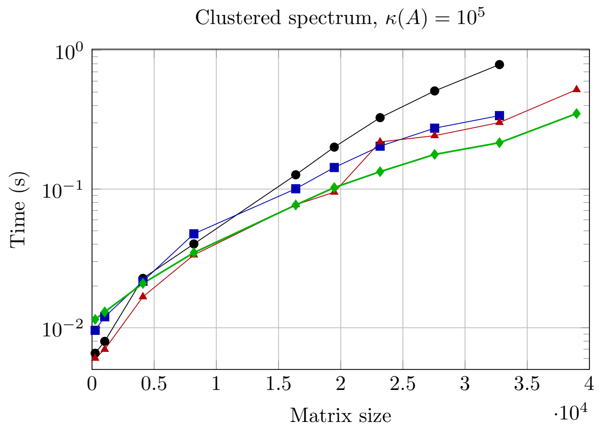
Assuming we have n training samples, it is known from the literature that the Cholesky factorization of



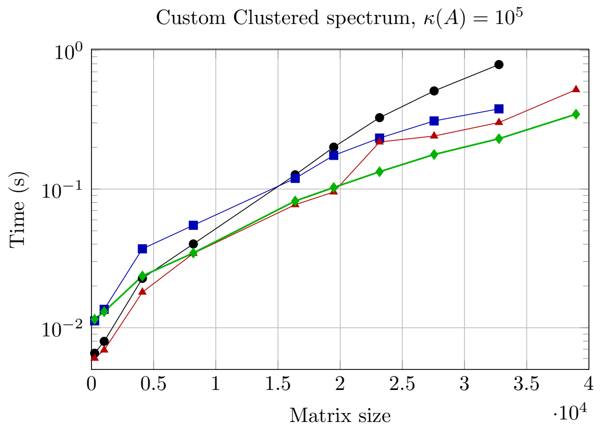
(a) Linear Spectrum



(b) Geometric Spectrum



(c) Clustered Spectrum

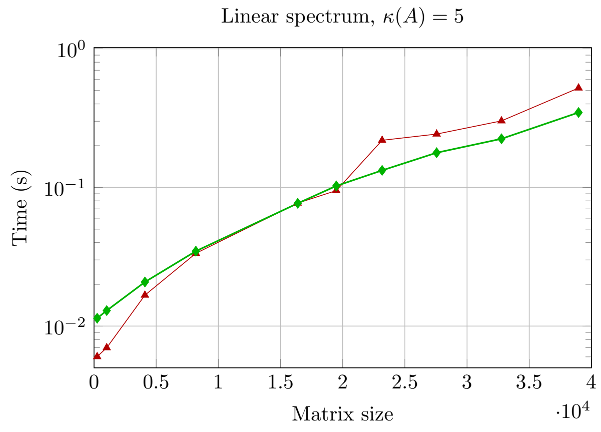


(d) Custom Clustered Spectrum

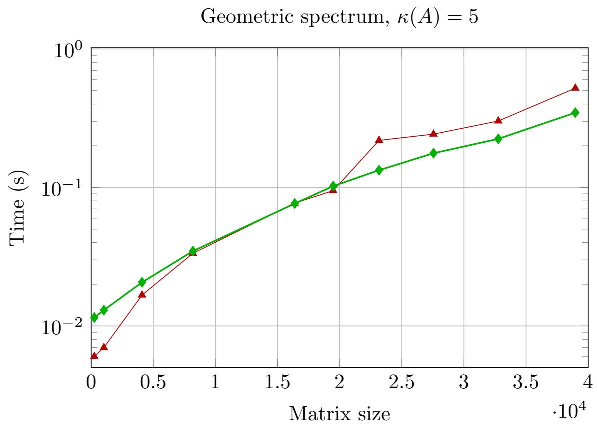
Figure 4: Elapsed time in different spectrums with condition number $\kappa(A) = 10^5$ for standard double precision and mixed precision (Cholesky based GMRES-IR) solvers on ALIS and Kestrel HPC machines. Black curve represents ALIS with the standard double precision solver, blue curve represents ALIS with the mixed precision solver, red curve represents Kestrel with the standard double precision solver, and green curve represents Kestrel with the mixed precision solver.

the covariance matrix over these training samples is computed to perform model fitting (or hyperparameter tuning), as well as obtaining predictions and their corresponding uncertainties. Computing the Cholesky factor R and using it to solve a linear system requires $\frac{1}{3}n^3$ and $\frac{5}{2}n^2 + \frac{1}{6}n$, respectively [16]. In particular, model fitting executes both of these operations repeatedly, and as also discussed in [16], is one if its major bottlenecks. By reducing the cost of floating point operations on the factorizations, we aimed at an improvement in the overall performance.

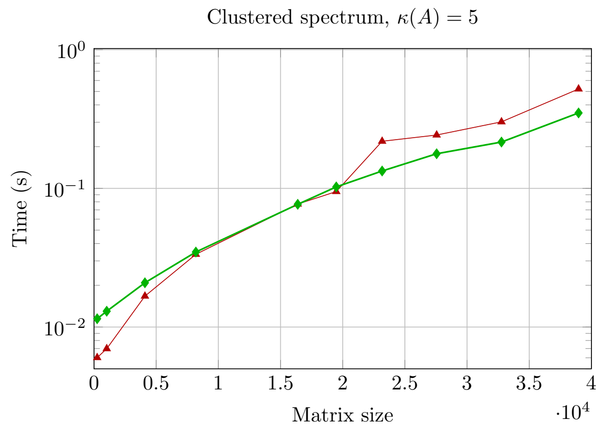
For the experiments presented here, we used the GPyTorch library [5], using the L-BFGS optimizer with learning rate $\alpha = 0.15$, 50 iterations and 3 restarts, as well as the Anisotropic Matern 3/2 kernel. The source code is available at `https://github.nrel.gov/jdeoliv/gpytorch_solver_comparisons`. We used the Pol dataset from the UCI Datasets [4] and first used PyTorch profiler tool to verify what were the functions that were more computational demanding during the model fitting. Table 10 indicates that the Cholesky factorization is indeed among these functions, which is in accordance with the literature on GPR. When the same is done using the mixed-precision factorization, a speedup of approximately 1.75 is noticed in the training time, as described on Table 11. We also compared if there was a difference in the training and testing error between the two approaches. Table 12 indicates there is a slightly lower error using only the single precision factorization and solver.



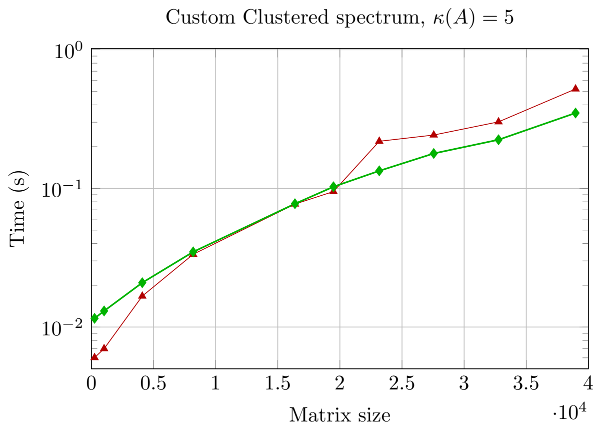
(a) Linear Spectrum



(b) Geometric Spectrum



(c) Clustered Spectrum

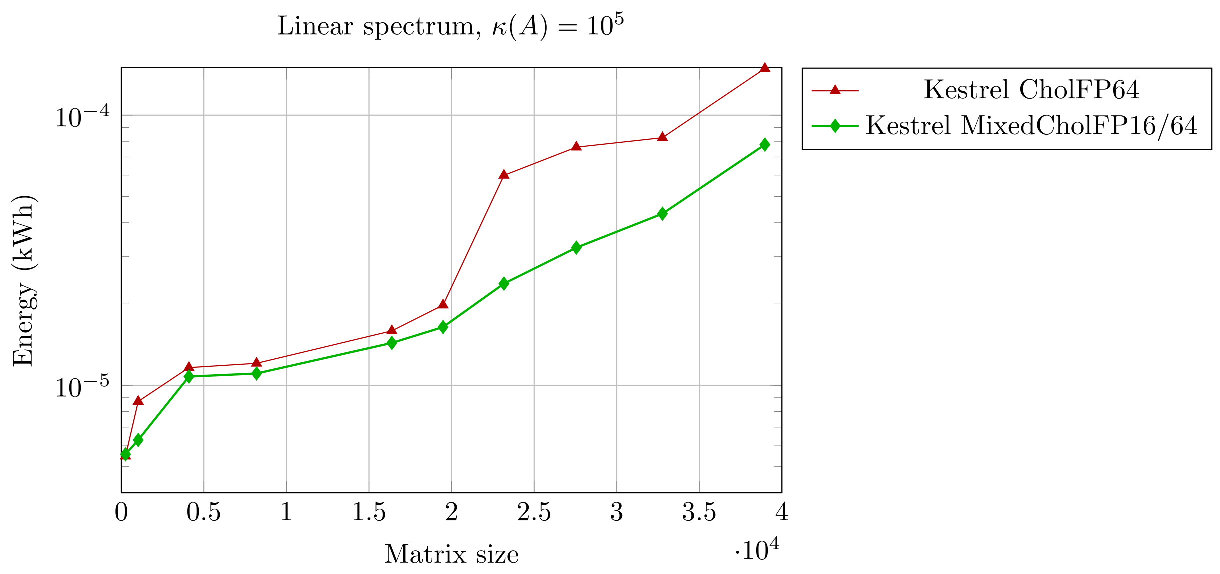


(d) Custom Clustered Spectrum

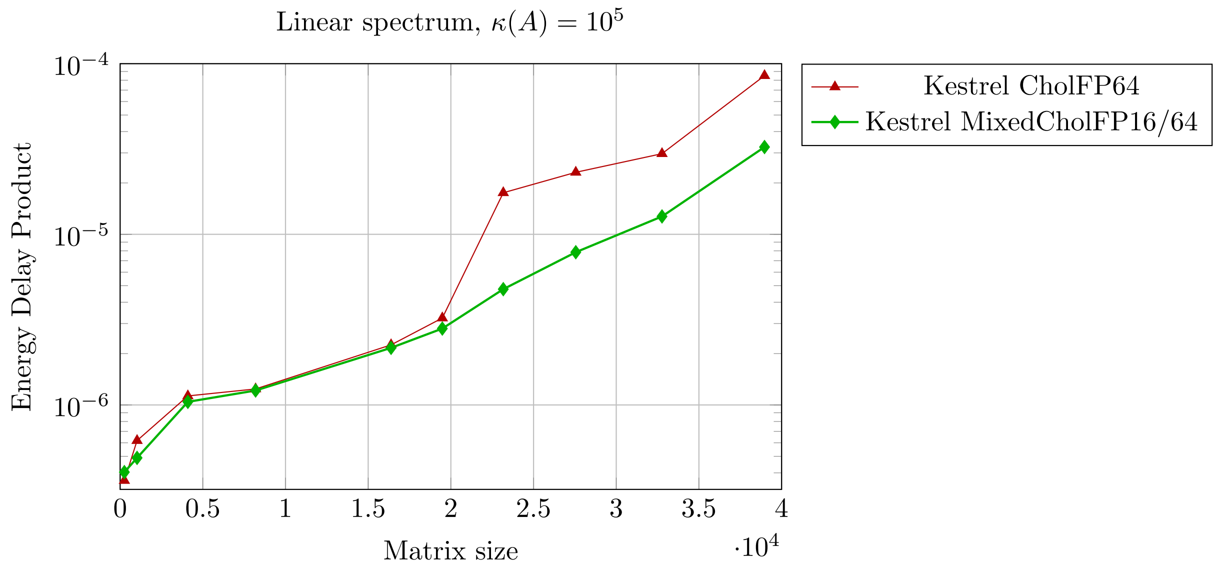
Figure 5: Elapsed time in different spectrums with condition number $\kappa(A) = 5$ for standard double precision and mixed precision (Cholesky based GMRES-IR) solvers on Kestrel HPC machines. Red curve represents Kestrel with the standard double precision solver and green curve represents Kestrel with the mixed precision solver.

It is worth noting that results presented in [12] suggest a lower MSE training error is obtained through GPR compared to the ones depicted in Table 12 for both single and mixed precision approaches. By fixing a value for the noise hyperparameter $\sigma = 0.06321$ instead of including it with the other kernel function hyperparameters, similar results were obtained (see Table 13). Since there is no longer an iterative refinement step, errors associated with the mixed-precision implementation were slightly larger than the ones obtained with single precision. As for energy efficiency, Table 14 describes that there is a reduction in over 35% in the energy consumption in both cases (including or not including the noise hyperparameter in the training process) and the obtained EDPs are lower on both cases.

One drawback of the mixed-precision approach is that the factorization of the kernel matrix is more susceptible to rounding errors, an effect that may be even increased depending on the support of the chosen kernel and not optimizing the noise hyperparameter σ [12]. Our experiments with the RBF kernel with the same fixed noise $\sigma = 0.06321$ even failed after adding jitter to the diagonal 4 times. We believe the pre-processing discussed in [8] could alleviate the effects of rounding in the factorization, although more experiments are needed to confirm it.



(a) Energy consumed (kWh)



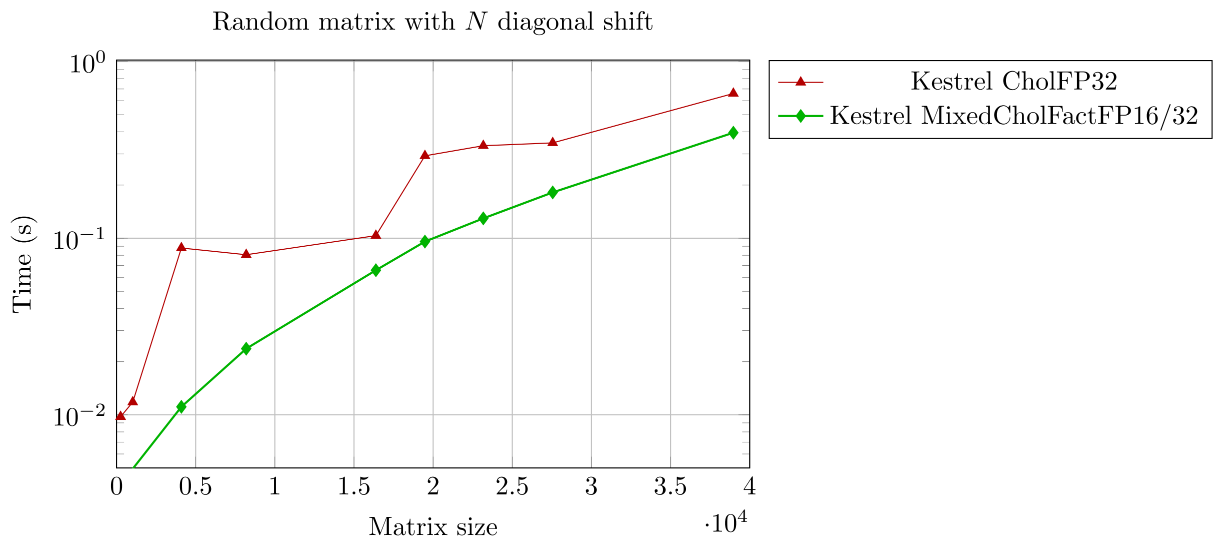
(b) EDP (kWh^2)

Figure 6: Energy consumed and EDP with respect to matrix size.

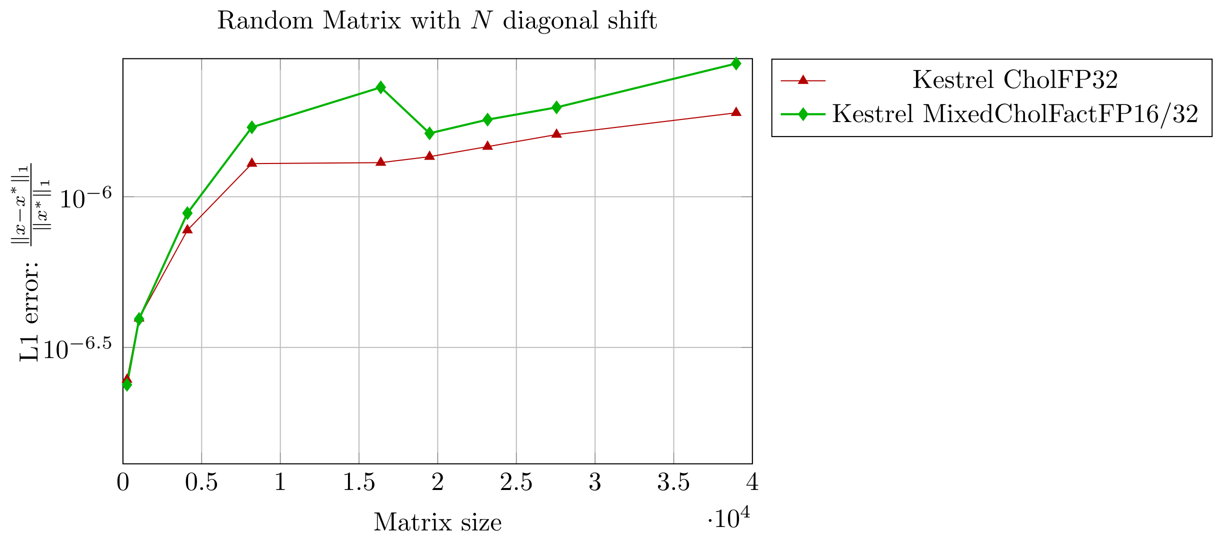
Proposal for adding mixed-precision algorithms on Pytorch. In `https://github.com/pytorch/pytorch/issues/132940`, we listed some advantages of having mixed-precision algorithms from MAGMA and cuSOLVER available via Pytorch. We provided examples on how these wrappers could be introduced via code samples. We were unable to provide a proper implementation because we lack deep understanding of Pytorch and the linkage with its backends.

6 Key takeaways and suggestions for future work

In this document, we have presented the basics about mixed-precision algorithms and low-precision types, and how to use those for better energy efficiency on two sets of applications. We also provided insights about how to measure energy consumed by a program, and how programming languages influence those



(a) Time



(b) Forward error

Figure 7: Computation time and forward error of solving linear system only using mixed precision during Cholesky factorization step.

measurements.

We verified that the use of Automated Mixed Precision, available on packages like Pytorch and TensorFlow, can be used to reduce energy and execution time of training and inference of neural networks. We showed examples on how to configure the neural networks so as to explore the benefits of both Intel AMX and Tensor Cores technologies. It was crucial to use wider networks to get good results with mixed precision, especially in the case of Intel AMX. When Pytorch and TensorFlow are put side-by-side, we notice that Pytorch was more energy efficient in most of the cases.

For the problem of solving SPD linear systems in mixed precision, we verified that the performance gain and energy reduction gets more relevant as the matrix size increases. The turnover point also depends on the hardware. By just replacing the single-precision factorization by the mixed-precision factorization in the GPR, we noticed improvements in energy consumption and time with less relevant reduction in accuracy.

All software considered in this work lack essential components for the examples studied. TensorFlow's

Function	**Time (s)**
Optimizer.step	75.345
cudaStreamSync	48.256
aten::item	42.121
aten::_local_scalar_dense	42.115
aten::is_nonzero	42.076
autograd::engine::eval_func: LinalgCholeskyExBackward0	41.465
LinalgCholeskyExBackward0	41.463
aten::linalg_cholesky_ex	28.929
aten::linalg_solve_triangular	23.521
aten::matmul	17.363
aten::mm	17.362
cudaGetDeviceProps	15.207

Table 10: Total computation time of 12 most demanding routines using standard single precision factorization and solvers. Each entry includes also the time elapsed in function this particular function may call, in particular, Optimizer.step function calls all the other functions and has the total time of training. Cholesky related routines are associated with more than 50% of the elapsed time.

Function	**Time (s)**
Optimizer.step	43.182
cudaStreamSync	25.706
autograd::engine::eval_func: LinalgCholeskyExBackward0	25.4755
LinalgCholeskyExBackward0	25.4747
aten::item	25.1471
aten::_local_scalar_dense	25.149
aten::is_nonzero	24.8482
aten::linalg_cholesky_ex	14.889
aten::linalg_solve_triangular	14.45742
aten::matmul	10.66078
aten::mm	10.660
cudaMemcpyAsync	5.2775

Table 11: Total computation time of 12 most demanding routines using mixed precision factorization and single precision solver. Each entry includes also the time elapsed in function this particular function may call, in particular, Optimizer.step function calls all the other functions and has the total time of training. Cholesky related routines are associated with more than 50% of the elapsed time.

Method	**MSE (Train)**	**MSE (Test)**
Single	1.0298	2.83799
Mixed	1.1166	2.90533

Table 12: Training and Testing MSE for GPR of the Pol dataset with single and mixed precision (including noise σ in the hyperparameter tuning process).

Method	**MSE (Train)**	**MSE (Test)**
Single	0.0757	4.33389
Mixed	0.1527	5.09504

Table 13: Training and Testing MSE for GPR of the Pol dataset with single and mixed precision (hyperparameter tuning process not including noise σ).

	Time (h)	**Consumed Energy (kWh)**	**EDP (kWh^2)**
Single	0.014343	0.011389	0.000163
Mixed precision	0.009629	0.0075018	7.2e-05
Single (tuning noise σ)	0.020516	0.0165463	0.000339
Mixed precision (tuning noise σ)	0.010606	0.0087406	9.3e-05

Table 14: Time, Energy and EDP in the overall Gaussian Process Regression (training and prediction). Results indicate a reduction of more than 50% and 35% including and not including the noise σ hyperparameter, respectively. Furthermore, the lower EDP values indicate that the new approach is more efficient all around.

mixed precision does not seem to be available for BNNs on GPUs (See discussion at 4.5). Moreover, TensorFlow has more ways to enable mixed precision than explained in the official documentation, which may complicate its usage. Pytorch does not provide access to all mixed-precision implementations available in its backends as mentioned in Section 5. MAGMA lacks support for multiple right-hand sides on its mixed-precision linear system solver, and this hinders its usage on applications such as GPR. Thus, there is a need for software that can fill current gaps in the application stack, enabling the use of state-of-the-art mixed-precision implementations.

Suggestions for future work:

- Reproduce the mixed-precision tests on Apple Metal [21] backend with macOS Sonoma and above.
- Test the diagonal scaling proposed at [8] with the GPR instead of adding the diagonal jitter.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, National Renewable Energy Laboratory under contract number DE-AC36-08GO28308.

References

[1] A. Abdelfattah, S. Tomov, and J. Dongarra. Investigating the benefit of fp16-enabled mixed-precision solvers for symmetric positive definite matrices using gpus. In *International Conference on Computational Science*, pages 237–250. Springer, 2020.

[2] L. Bouza, A. Bugeau, and L. Lannelongue. How to estimate carbon footprint when training deep learning models? A guide and review. *Environ. Res. Commun.*, 5(11):115014, 2023.

[3] B. Courty, V. Schmidt, S. Luccioni, Goyal-Kamal, MarionCoutarel, B. Feld, J. Lecourt, LiamConnell, A. Saboni, Inimaz, supatomic, M. Léval, L. Blanche, A. Cruveiller, ouminasara, F. Zhao, A. Joshi, A. Bogroff, H. de Lavoreille, N. Laskaris, E. Abati, D. Blank, Z. Wang, A. Catovic, M. Alencon, Michał Stechły, C. Bauer, L. O. N. de Araújo, JPW, and MinervaBooks. mlco2/codecarbon: v2.4.1, May 2024.

[4] D. Dua and C. Graff. UCI machine learning repository, 2017.

[5] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in Neural Information Processing Systems*, 2018.

[6] P. E. Gill and E. Wong. Methods for convex and general quadratic programming. *Mathematical programming computation*, 7(1):71–112, 2015.

[21] https://developer.apple.com/metal/

[7] N. J. Higham and T. Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[8] N. J. Higham and S. Pranesh. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM Journal on Scientific Computing*, 43(1):A258–A277, 2021.

[9] H. Hoteit and A. Firoozabadi. Modeling of multicomponent diffusions and natural convection in unfractured and fractured media by discontinuous galerkin and mixed methods. *International Journal for Numerical Methods in Engineering*, 114(5):535–556, 2018.

[10] IEEE. Working Group P3109 Interim Report. Technical report, IEEE, 2024. v0.7.0.

[11] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Van Dyke, C. Vaughan, J. H. Laros III, K. Pedretti, S. M. Kelly, et al. Energy delay product. *Energy-Efficient High Performance Computing: Measurement and Tuning*, pages 51–55, 2013.

[12] W. J. Maddox, A. Potapcynski, and A. G. Wilson. Low-precision arithmetic for fast gaussian processes. In *Uncertainty in Artificial Intelligence*, pages 1306–1316. PMLR, 2022.

[13] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. In *ICLR*, 2018.

[14] NVIDIA. Mixed precision training. `https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/`, 2024. Accessed: 2024-08-08.

[15] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[16] C. K. Williams and C. E. Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[17] J. Yang, X. Liao, X. Yuan, P. Llull, D. J. Brady, G. Sapiro, and L. Carin. Compressive sensing by learning a gaussian mixture model from measurements. *IEEE Transactions on Image Processing*, 24(1):106–119, 2015.