Vectorizing Pytorch for RISC-V RVV

Author
Johannes Laute

Supervisor
[Marc Casas, BSC]

Supervisor
[Prof. Adrien Cassagne, Sorbonne University]

[Sorbonne University]
[Universitat Politècnica de Catalunya Barcelona Tech - UPC]

Abstract

In this internship we explore avenues for the vectorized execution of Pytorch models on RISC-V CPUs with Vector support. We identify 3 areas where Pytorch would benefit from vectorization: 1. the ATen computation backend, 2. the BLAS library, 3. the oneDNN compute library. Our contributions are as follows: we implement the vectorized class of ATen using RVV intrinsics, and we integrate vectorized version of BLAS and oneDNN into the Pytorch build process. This required us to setup an advanced, custom cross-compilation toolchain, including automated assembly modifications. Finally we evaluation the performance gained in elementary functions, fundamental building blocks of Deep Learning models (Linear Layers, Attention Layer and Convolutional Layers) and full AI models on our target hardware system, which is the EPAC (European Processor Accelerators) design, which is part of the European Processor Initiative.

September 13, 2024

Contents

1 Acknowledgements

I want to express special thanks to my supervisors Marc Casas from Barcelona Supercomputing Center and Prof. Adrien Cassagne from Sorbonne University. Their guidance and advice throughout the whole project was invaluable.

Furthermore I want to thank Prof. Jesus Labarta, Prof. Francisco Igual, Alex Santana and Roger Ferrer.

This paper is co-financed by the Barcelona Zettascale Laboratory with reference REGAGE22e00058408992 which is financed by the Ministry for Digital Transformation and Public Services within the framework of the Resilience and Recovery Facility - and the European Union - NextGenerationEU.

2 Introduction

With the release of ChatGPT in November 2022 OpenAI [2024], there has been a massive surge in interest for AI and Deep Learning. The previous trend of AI adoption has been accelerated, as new models like LLMs (Large Language Models) and VLM (Vision Language Models) are significantly more general. Companies and Institutions are evaluating and deploying these models and new applications and use cases are presented at a rapid pace.

At the same time, the compute requirements for these models have grown even more rapidly, with the compute requirements for training state of the art transformer models having increased at a rate of 750x per two years from 2018 to 2022 Gholami et al. [2024]. This progress has mainly been driven by advances in two areas: performance improvements in GPUs, that now contain specific hardware and instructions for AI models, and by improvements in algorithms and parallelization strategies that enable massively parallel training Rasley et al. [2020], with the current LLama 3 model having been trained using 16,000 Nvidia H100 GPUs MetaAI [2024].

On the other hand, smaller, lighter and more specialised models have also experienced big gains in performance and deployment. Unlike the truly massive SOTA models, here often inference and sometimes even training is done on the CPU. Major CPU providers have reacted and implemented optimizations for common operations, for example Intel introduced the AMX (Advanced Matrix Extensions) into its processors in 2023.

This rise in AI concurs with another trend in the computing landscape, the emergence and accelerating adoption of RISC-V Waterman and Asanović [2019]. RISC-V is an open standard, extensible Instruction Set Architecture for CPUs which was first introduced in 2014. As RISC-V continues to gain traction, this work focuses on preparing the software ecosystem for the arrival of these new processors. In the field of Deep Learning and AI, PyTorch has become the most widely used framework for research and development of new models and methods, so we focus our efforts on optimizing PyTorch for RISC-V.

The RISC-V RVV Vector extension offers a more dynamic and flexible way of vectorization than traditional SIMD instruction sets like AVX or ARM Neon, both for the chip manufacturers and for the software implementations, including the development of vector length agnostic codes, that is, codes that run on any RISC-V machine independently of the vector length. As a result of this flexibility, RISC-V also allows the implementation of Long Vector Architectures, a design that first appeared in the 1970s and was later overtaken by small fixed width SIMD architectures. In recent years however, there has been renewed interest in Long Vector Architectures with the relese of the Arm Scalable Vector Extension (SVE) and NEC Aurora SX architecture. Recent research has shown that Long Vector Architectures can have benefits compared to the more common SIMD approach Alexandre de Limas Santana [2023] Vizcaino et al. [2023].

In this work, we focus on vectorizing the ATen Tensor backend of Pytorch for RVV. In addition we integrate an already optimized BLAS library, and an optimized deep learning compute library into the Pytorch build process. We then perform benchmarks of specific operations, as well as runs of single layers and whole models which can be seamlessly downloaded and run in Python via Huggingface.

The hardware environment of this project was the experimental Arriesgado cluster at the Barcelona Supercomputing Center, which comprises of different nodes, some running commercially available RISC-V CPUs, others emulating BSCs Avispado CPU via FPGAs.

The primary challenge of this work was adapting PyTorch's compilation and build process for Avispado CPUs, which required overcoming both the complexity of the build process and the lack of mainline compiler support for the Avispado CPU implementation of RISC-V RVV specification version 0.7.

All in all, we succeeded in vectorizing the ATen backend and manage to run Pytorch models with no code changes on our target platform. Next steps are to compare this approach against other, compiler based approaches to deploying Pytorch models, and to contribute the source code to the PyTorch project on Github.

3 Background

3.1 RISC-V

RISC-V is an open ISA (Instruction Set Architecture) first released by the University of California Berkeley in 2014. Waterman and Asanović [2019]

It is a load-store architecture, and adheres to the Reduced Instruction Set Architecture (RISC) approach of designing Instruction Sets. In recent years, many companies have announced and released first CPUs implementing it. RISC-V has a modular design, with a base instruction set and many optional extensions, providing instruction for various use cases.

The base instruction set (I) is very minimal with about 50 instructions, which include common control flow instructions, loads, stores and simple integer arithmetic. It defines 32 registers of XLEN bits, the two common variants are RV32I and RV64I, with XLEN=32 and XLEN=64 respectively. Like in MIPS, there is a hardwired zero register, which is called x0 and is read-only.

This base instruction set is not enough for the vast majority of implementations, this is where the modular concept of extensions comes into play. Some common extensions are:

- **M (Multiplication)** The multiplication extension adds integer multiplication, division and remainder operations. Due to their much higher latency compared to the simpler arithmetic operations included in the base instruction set, it was decided to make them optional so small micro-controllers don't need to add extra logic to implement these instructions if they don't require them.
- **A (Atomic)** The atomic extension adds support for atomic instructions which are necessary for multi-threaded and and multi-core processors, specifically Load-Reserved, Store-Conditional, and a range of other instructions like atomic add, and, max etc.
- **F (Floating Point)** The F extension adds 32 registers for IEEE 754 Binary32 floating point numbers, and a variety of instructions to perform floating point computations.
- **D (Double Precision)** The D extension is very similar to the F extension, but for IEEE 754 Binary64 double precision floating point numbers.
- **C (Compressed)** While most instructions are encoded in 32 bits (which is the standard), RISC-V also supports different encoding widths. To reduce the code size, the C extension adds 16 bit encoded versions of common instructions of the I/F/D extensions, which can in practise reduce code size by 25-30%.
- **V (Vector)** The V extension adds Vector instructions and will be explained in detail in the next section.

3.2 RISC-V Vector Instructions

The RISC-V Vector Extension adds vector computation to the RISC-V ISA. It is flexible and the CPU architect can chose the size of the vector register (VLEN) and the maximum size of an element (ELEN) of that vector.

VLEN and ELEN need to be powers of 2, with $8 \leq ELEN \leq VLEN$.

RVV adds 32 registers of size VLEN, and many instructions to operate on these vector registers. Because there are a lot of instructions, there is not enough space left in the 32 bit instruction encoding space to accommodate all of them. Thus CPU state is used to decide which instruction will be executed. There are 2 status registers that influence the execution of the RVV instructions: VTYPE and VL.

The VTYPE status register contains the currently selected element width (SEW, $8 \leq SEW \leq ELEN$) and the Length Multiplier (LMUL) which describes the current register grouping configuration.

The Length Multplier (LMUL) allows us to group vector registers together. With an LMUL of 1, we operate on a single register, with LMUL > 1, we operate on multiple registers with a single instruction, the maximum LMUL is 8. RVV 1.0 introduced fractional LMUL values, (1/8, 1/4, 1/2), which correspondingly operate on a fraction of a single register.

The VL status register contains the current number of elements that we operate on, with $0 \leq vl \leq (VLEN/SEQ) * LMUL$. This means that if we use smaller data types, we can operate on more elements at the same time.

These two register together with the instruction completely determine the operation that will be performed, ie they can be seen as implicit operands to every instruction in the RVV extension. These status registers can be read and written via the standard control and status register instructions of the RISC-V ISA, and are also exposed to the programmer via Compiler Intrinsics.

3.3 Pytorch

PyTorch is a popular deep learning framework released in 2019 Paszke et al. [2019] by Facebook AI research (FAIR). It is a Python library, that calls into a C++/CUDA backend to execute the computationally intensive operations. Unlike its main competitor at the time, TensorFlow Abadi et al. [2016], PyTorch implements the define-by-run design. This means that the computational graph of the neural network is recorded during execution, and then backpropagation and training can take place. In contrast Tensorflow required the user to define the whole graph ahead of time, and then execute it after. This difference in design is often credited for PyTorch's ease of usability and success.

Using Pytorch is very similar to using Numpy, but we have a plethora of deep learning operators already implemented. After defining the computation, we ask Pytorch perform backpropagation to calculate the gradients, and then we can optimize our parameters using one of Pytorch's implemented optimizers. Here is a very minimal Linear Regression example:

```
import torch
import torch.nn as nn
x = torch.randn(10, 5) # x is our input
target = torch.randn(10, 1)
linear = nn.Linear(5, 1) # define the model
optimizer = torch.optim.SGD(linear.parameters(), lr=0.01) # initialize an optimizer
output = linear(x) # calculate our prediction
loss = nn.MSELoss()(output, target) # calculate the Mean Squared error
optimizer.zero_grad() # clear out the gradients from the last iteration
loss.backward() # perform backpropagation
optimizer.step() # update the weights
```

The adoption of Pytorch grew rapidly, especially in the research community. Recently, there has been a move to more compiler oriented approaches (XLA, JAX), to take full advantage of new accelerators (TPUs) and GPUs. The main motivation here is to eliminate the back and forth between the accelerator and the CPU. Pytorch has reacted to this by introducing the torch.compile framework, which captures the computation graph during the initial execution of the Python code, and then JIT compiles it for subsequent executions.

3.4 ATen

ATen (A Tensor Library) is the multi-dimensional array library and computation backend of Pytorch. It is written in C++ and implements a Tensor class and associated operations. This library is then used to implement backpropagation and training algorithms which are then exposed to Python via Pybind11.

ATen has multiple execution backends, most importantly a CPU backend and a CUDA backend. The CPU backend makes heavy use of a Vectorized class that abstracts SIMD operations for different Instructions Set Architectures (currently AVX2, AVX512 and ARM NEON). In this work we add a

RISC-V implementation of this class, which enables all ATen functions that make use of it, to now use RISC-V RVV instructions.

As explained earlier, RISC-V RVV, similar to Arm SVE, is not a fixed with SIMD instruction set, but a flexible vector instruction set. Due to ATen being designed specifically for fixed width SIMD instruction sets, we cannot take advantage of the flexibility of RVV in full. Instead we have settled on specifying the vector width at compile time.

4 Execution Environment

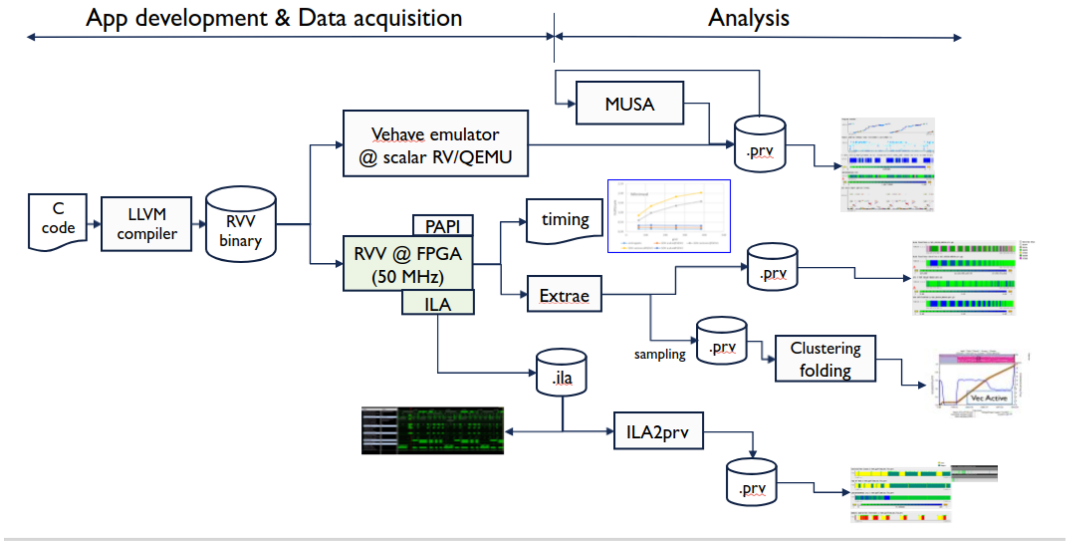

Figure 1: Overview of the Arriesgado hardware and software environment, taken from Jesus Labarta's presentation at 384HPC Summit

4.1 Hardware

BSC has developed an execution environment in the HCA Arriesgado cluster as part of the SONAR (SOftware research and development vehicles for New ARchitectures) research group. It consists of commercially available RISC-V machines, for example the Sifive U74 core. As this CPU doesn't implement the vector instruction set, BSC has developed Vehave, which intercepts SIGILL exceptions, simulates the execution of vector instructions and logs them for later analysis with the BSC Paraver tool.

In addition BSC is using the Avispado core, designed by Semidynamics, that contains the Vitruvius vector unit, designed by BSC. The core implements the RV64GCV variant of the RISC-V ISA, where G stands for the "IMAFD" extensions as they were mentioned before. This means that the processor has support for Integer Arithmetic, including multiplication and division, single and double precision floating point arithmetic (FD), atomic instructions (A), compressed instructions (C), and vector instructions (V).

The registers of the Vitruvius vector unit are 16384 bits wide, which means they can hold a maximum of 256 64-bit double precision floating point values, or 512 32-bit single precision values. The current configuration has 8 vector lanes, meaning that 8 64-bit arithmetic operations can be performed in parallel, 16 32-bit operations respectively. In addition a lightweight out-of-order engine is implemented that allows for overlapping execution of memory and arithmetic operations.

This core is running in simulation on an FPGA at 50 MHz frequency, which means that using all 8 lanes we have a maximum performance of $0.05 * 8 = 0.4$ GFLOPS/s of double precision compute (0.8 GFLOPS/s in single precision). The fabricated chip will run at a much higher frequency. The FPGA implements a simulation of 1 of these core/vector unit pairs, the current baseline chip has 4 of these tiles, and the design allows a scaling of up to 512 core/VPU tiles.

4.2 Software

Our compilation setup is quite complicated. The reason behind this is that the Avispado CPU currently implements RVV 0.7, but mainline compilers (Clang/GCC) don't support this version of the standard, as version 1.0 has been ratified. In addition, the Clang fork for RVV 0.7 maintained at BSC is based on Clang 12 and does not support some features that are needed to build Pytorch and use RVV intrinsics in the way that we would like to. Specifically, only Clang 17 allowed to specify the actual vector register size at compile time, and this is necessary in our approach as explained later.

For this reason, we use Clang to compile, and then do a replacement step on the assembly level to convert the RVV 1.0 assembly into RVV 0.7 compliant assembly, before finally linking with the GCC linker, which has introduced support for the almost compatible XTheadVector instruction set. In addition, it is necessary to execute compiled code during the Pytorch build process, as we are in a cross compilation setting this presents a problem as our x86 host is unable to execute the generated RISC-V binaries. To overcome this problem we use "binfmt", which is a tool that allows us to inspect binary headers before execution, and if we detect that it is a RISC-V binary, we run the binary using the QEMU virtualization software Bellard [2005].

4.2.1 Vehave

Vehave is a RVV simulation tool developed at BSC. It enabled to run an analyse RISC-V code containing RVV instructions on hardware that doesn't contain a vector unit. It works by adding a library to the "LD_PRELOAD" environment variable, which registers an exception handler for the "SIGILL" (Illegal Instruction") exception. During execution, when a RVV instruction is encountered the processor produces a SIGILL exception, and then Vehave decodes and simulates the execution of this instruction. In addition it can log the details into a trace file, that can later be analysed with Paraver. Vehave is a functional simulator and doesn't report the cycles or performance of the simulated instructions.

4.2.2 Clang

Clang is part of the LLVM project which was started in 2003 by Chris Lattner. It introcued the LLVM IR (intermediate representation), and offers many modular pieces like the Clang compiler for C and C++, a new C++ standard library (libc++). We chose Clang because at the time GCC did not support RVV intrinsics, and the support for fixed length RVV vectors was also not present. We disable all automatic vectorization, as Clangs aggressive vectorization has caused problems with our assembly replacement step. We use the "-B" option to direct Clang to our custom assembly script that invokes first our modified rvv-rollback, and then uses the GCC assembler after we change the ISA string at the top of the assembly file to the XTheadVector extension for the GCC Assembler and Linker.

4.2.3 RVV Rollback

RVV Rollback Lee et al. [2023] is a software released in 2023 by Joseph K. L. Lee et all from Edinburgh University. Due to the early release of the 0.7 draft specification of the RVV extension various companies and research labs have implemented processors based this unratified version of the specification. Because it was not ratified at the time, compilers like GCC and Clang have decided not to include it into the main project, and various forks of different versions of these compilers have been made to support RVV 0.7, with most of them no longer being maintained. Now, after the release and ratification of RVV 1.0 in 2021, GCC and Clang have implemented this version, and the work on optimizations and code generation is being done with RVV 1.0 as the target. This creates a problem for all the organizations that have acquired or designed a CPU implementing RVV 0.7, as they cannot harness new compiler advances. The authors aim to solve this problem by introducing an automated translation at the assembly step, as the differences between RVV 0.7 and 1.0 are not very big. This is implemented as a Python program that goes through the assembly output of a compiler that produces RVV 1.0 assembly line by line and using string matching to replace any instruction that are not part of RVV 0.7. This is a quite limited approach, as there is no context known of previous instructions. As a consequence, the replacement must maintain the processor state exactly, ie. any temporary registers used need to saved and restored via the stack, and any changes in CSRs (Control and Status Registers) need to be reverted after the completion of the replaced instruction.

To give an example, the instruction to load elements from memory as bytes, `vlb.v` (vector load byte) has been renamed to `vle8.v` (vector load element of 8 bits) in RVV 1.0. In this case we simply rename the instruction.

A more complicated case, which came up many times, is the following:

RVV 1.0 has introduced ”whole-register” memory operations. This means that regardless of the currently configured vector length, these instructions will load or store the whole register into memory. As RVV 0.7 does not have this instruction, we need to query the system of the maximum vector length, configure this as our new vector length, perform the memory operation, and then restore the exact state the processor was in before. These ”whole-register” instructions are used often by the compiler in case of register spills, and also in case of function calls where the operand is passed via the stack.

In these cases the RVV 1.0 instruction

```
vl1re8.v v2, (a0) # load the whole register with bytes stored at address a0
```

becomes

```
sd      t0, 0(sp)      # store the current content of t0 on the stack
sd      t1, 8(sp)      # store the current content of t1 on the stack
cssr    t0, vl         # read the current vl into t0
cssr    t1, vtype      # read the current vtype into t1
vsetvli x0, x0, e8, m1 # set the vl to 0 (which sets it to the maximum possible value)
vlb.v   v2, (a0)       # perform the "whole-register" load
vsetvl  x0, t0, t1     # restore the previous vtype and vl
ld      t0, 0(sp)      # restore register t0
ld      t1, 8(sp)      # restore register t1
```

We have modified the rvv-rollback script extensively to work correctly in this project: We added instruction that were not included, though they can be translated, we added a workaround for the absence of fractional LMUL values (as explained in the RISC-V section) in the RVV 0.7 specification, which causes an error message in the original rvv-rollback. Also, we have added support for the `vlenb` status register, which holds the maximum number of bytes that can be held in a vector register which is not part of RVV 0.7. All in all we were able to compile all of Pytorch using this assembly replacement scheme, with the modifications and bugfixes that we added.

5 Vectorizing Pytorch

5.1 ATen Vectorized

ATen is the Tensor backend of Pytorch as described in the Background Section. The Vectorized ⟨*Type*⟩ class is the backbone of the CPU vectorization used in ATen. It is a small header only library, that is then included in all implementations that are vectorized in the ATen.Native.CPU namespace.

The structure of the class is as follows:

```
typedef vfloat32m1_t vfloat32m1xn_t __attribute__((riscv_rvv_vector_bits(VLEN)));
template <> class Vectorized<float> {
private:
    vfloat32m1xn_t values;
public:
    using value_type = float;
    using size_type = int;
    static constexpr size_type size();
    // Constructor and memory operations
    Vectorized(float val);
    static Vectorized<float> loadu(const void* ptr, int64_t count = size());
    void store(void* ptr, int64_t count = size()) const;
    // ...
    // Math functions implemented via sleef
    Vectorized<float> sin() const;
    Vectorized<float> cos() const;
    // Comparison operators
    Vectorized<float> operator==(const Vectorized<float>& other) const;
    // ...
};
// Arithmetic operators
template <>
Vectorized<float> inline operator+(const Vectorized<float>& a,
                                   const Vectorized<float>& b);
// ...
// Other functions like blend, clamp, fmadd etc
// ...
```

As we can see, we are holding the vector of values as a member of the class. This is a common design pattern when working with SIMD ISAs like AVX or NEON, but presents a problem with the dynamic RVV Vector ISA. The reason for this is that in C/C++ the size of a class must be known at compile time, and in general in RVV the vector register size (VLEN), is not known at compile time.

This is a well known problem in adding support for RVV or ARM SVE, which is also a Vector ISA, in libraries that were originally written for SIMD ISAs. In our situation we must match the interface exactly, because otherwise all the already implemented functions don't work correctly. We have considered 3 different ways of dealing with this problem:

1. Store a pointer to an array containing the values in the class, query the vector length in the constructor and allocate sufficient memory. Then load-modify-store for every operation that we need to perform.
2. Make use of flexible array members (which are a C feature but Clang accepts them in C++), to put a variable length array at the end of our class. Then during the construction of the class, query the vector size and allocate sufficient memory. This removes the added indirection of the pointer in approach 1.
3. Make the vector register size known at compile time, this way we can include it as a normal member of the class.

The main problem with 1. is that we add a lot of memory overhead, if we load and store the data into memory for every operation. This a problem because we are implementing very elementary functions like add, and, etc. If we implemented full algorithms or neural network layers, two more memory operations would not be so costly, but performing 2 memory operations for each add is prohibitively costly. In experiments with the *Compiler Explorer* we have discovered that very often the compiler is not able to eliminate these loads and stores, thus this option is not viable.

Almost the same applies to 2., though we do remove the indirection of going via a pointer. Another issue is that when using flexible array members like this, we cannot implement a constructor for this class, which is required by the API. The reason is that in the C++ object model, the memory for the object is allocated before the constructor is run, but in our case we would only calculate the memory required during the execution of the constructor.

We chose option 3, which became possible with Clang 17 (similar functionality is also available in GCC now). In the typedef at the top of the file, we inform the compiler of the size of our vector register, and during compilation we pass `-rvv-vector-bits=16384`. Now the compiler is able to treat the vectors as normal types (they are no longer "sizeless" types), and we can proceed as normal. This means however that for a CPU with a different vector register width, Pytorch needs to be recompiled.

5.2 Sleef

SLEEF (SIMD Library for Evaluating Elementary Functions) Shibata and Petrogalli [2020] is a library that provides vectorized implementations for all C99 real floating point math functions. All functions are implemented with a maximum of 1 ULP error, handle infinity and NaN inputs according the the C standard, and are well tested. For some functions faster versions with a higher error are also implemented. Recently, SLEEF has started to support RISC-V RVV in addition to SSE, AVX, ARM

NEON, ARM SVE and more. We have decided to use SLEEF as our vector libm for multiple reasons: implementing a well optimized and tested libm is a huge undertaking, SLEEF is open source and has a permissive license, and lastly, the ARM NEON ATen backend already uses SLEEF, so we did not have to add any dependency to Pytorch. Instead we only had to modify the build process to build SLEEF in the correct configurations.

We have observed mixed performance when using Sleef, more on this later in the Benchmarks Section.

5.3 BLAS/BLIS

BLAS (Basic Linear Algebra Subprograms) is a software specification that describes an API for Linear Algebra subroutines. There are multiple implementations, like openBLAS and Intel MKL. These implementation are highly optimized for specific processors to achieve maximum performance.

BLIS is a newer library written in C and assembly Van Zee and van de Geijn [2015]. It has its own interface, but also provides a BLAS compatibility layer. This allows us to use it in Pytorch, as Pytorch makes heave use of BLAS operations (GEMM in particular), for Linear Layers and the Attention mechanism. BSC has an optimized version of BLIS, that was optimized by Francisco Igual specifically for RVV and the Avispado core with the Vitruvius vector engine. We use this version in our build of Pytorch, because otherwise Pytorch falls back on a unvectorized, naive implementation of GEMM, that impacts performance very negatively.

5.4 OneDNN

OneDNN is an open source library developed by Intel that implements high performance kernels for a range of deep learning operations. It is specifically optimized for Intel CPUs, but also other hardware is supported. We mention it here because it is the default CPU backend for convolutions in Pytorch. Pytorch has implemented convolutions, but these not very optimized. BSC Alexandre de Limas Santana [2023] has optimized OneDNN for RISC-V RVV, using a optimized algorithm for long vector architectures, implemented via a JIT compiler that generates kernels inside of oneDNN. For the convolutional neural networks we compare with and without oneDNN.

6 Results

We benchmark our implementation on the Avispado Core with the Vitruvius vector engine as described above.

6.1 Micro Benchmarks

The micro benchmarks were run in C++, where we build and link against libtorch.so. This was purely done for convenience, all the functions are available and working via the Python interface. The reason we chose C++ was performance, as running the full Python interpreter is quite slow on our 50 MHZ core. For the micro benchmarks we measure the time to process 1 million elements to reduce variance. We are specifically interested in the impact of different vector widths on the performance.

6.1.1 Base Functions

We first started with simple elementary operations which are we directly implemented via compiler intrinsics in the Vectorized<float> class. We benchmark 10 different functions: 'abs', 'add', 'clamp', 'min', 'mul', 'div', 'ceil', 'sqrt', 'reciprocal', 'isnan'. We compare against the current implementation in Pytorch ("torch_scalar"), against a implementation without Pytorch ("no_torch_scalar") and then with various vector widths. We start with the average speedup obtained against the current version in Pytorch.

The first thing we notice is that there is some Pytorch overhead compared to directly calling into the C++ Standard Library, thus "no_torch_scalar" version is around 30% faster on average. Next, we notice that small vector sizes are significantly slower than the scalar baseline, with 256 bit registers (ie. 8 element vectors) only achieving 23 % of the scalar baseline. For larger vector sizes, we see a good performance growth, with the performance almost doubling each time we double the vector width. We

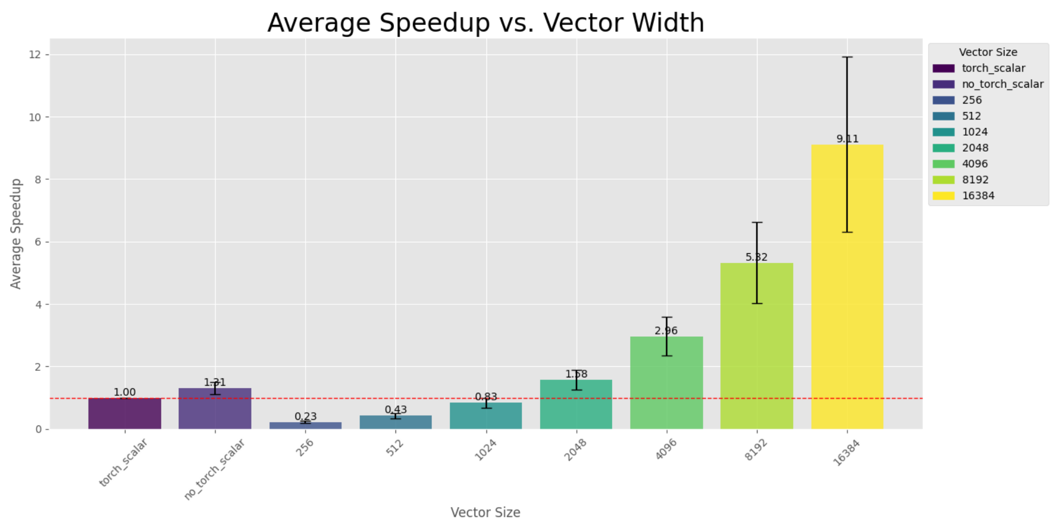

Figure 2: Average speedup of different implementations processing (load, operate, store) on an array of 1 million single precision float values

surpass the scalar implementation with 64 element wide vectors, and when the the full vector width of 16384 bits, we achieve an 9.11x speedup averaged over all operations. One thing that immediately catches the eye are the large error bars, especially at the largest vector sizes.

To investigate this further, we have looked at the best and worst functions regarding speedup. Here are the two slowest functions:

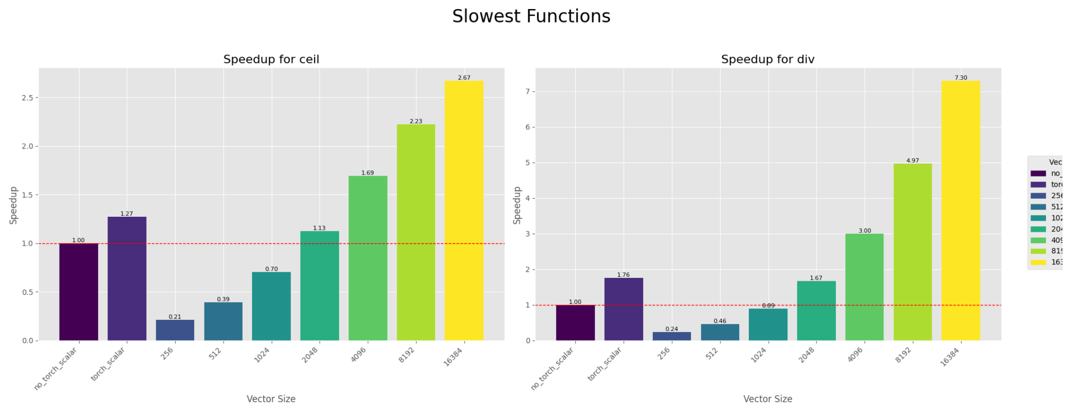

Figure 3: Base Functions with the lowest speedup obtained

Interesting its ”ceil” and ”div”, with ceil being particularly poor. This is due to the fact that we implement ceil by calling the ”at::native::ceil_impl” function, following the the ARM Neon implementation, which is not vectorized. This is a obvious point for improvement in the future.

Next we look at the two fastest functions:

These are ”clamp” and ”min”. Clamp is directly implemented in terms of ”min” and ”max”, thus it is also very fast, as these two functions are very fast compared to the scalar implementation.

Next we are looking at functions implemented in via the SLEEF libm.

6.1.2 Sleef Functions

First we again look at the average speedup, we tested these 7 functions: ’cos’, ’exp’, ’log’, ’log10’, ’round’, ’sin’, ’trunc’. Here we only compare the current Pytorch implementation on RISC-V against a minimal C++ implementation and against using the full vector width on our hardware. In earlier

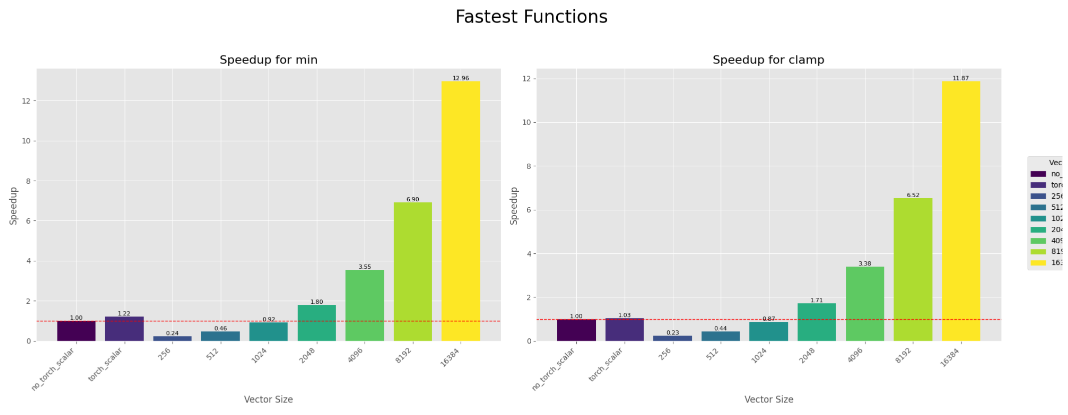

Figure 4: Base Functions with the fastest speedup obtained

experiments we have tried smaller vector widths as well, and the trend is similar to the base functions, larger vector widths are faster than smaller vector widths on our hardware.

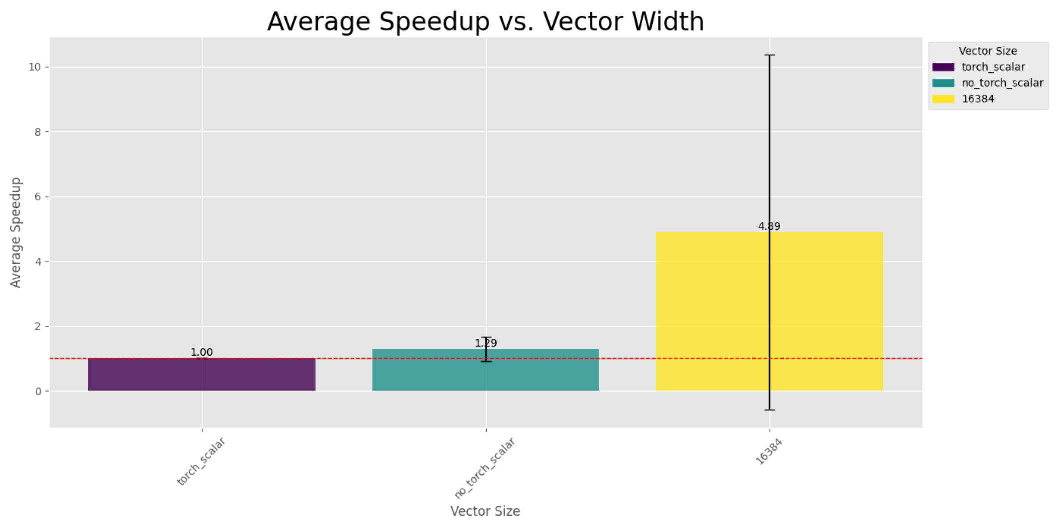

Figure 5: Average speedup of different implementations processing (load, operate, store) on an array of 1 million single precision float values

We observe that for Sleef we get a significantly worse speedup, only about 5x on average using our full vector register. We identify 3 possible causes for this: Sleef is configured to be build with the Zba, Zbb and Zbs bit manipulation extensions. These extensions add support for more advanced bit manipulations like count leading zeros, register rotations, combined shift-add operations and more. Our target hardware does not implement these extensions, thus we need to disable them, leading to Clang needing to implement these operations using more instructions. The second, and more important factor is that we are performing a function call into a shared library. Function calls can be expensive in general, and in addition the calling convention for RISC-V RVV vectors is still not completely specified. We have observed that in some cases, the vector register are passed via the stack, thus incurring a store and load to the memory. These are implemented as "whole register" memory operations, thus we are storing and loading 16384 bits, when passing a vector of 256 bits. Thirdly, the RVV implementations in Sleef could just be less optimal than the highly tuned and optimized standard library implementations, especially because RVV support for Sleef was recently added.

We have considered multiple strategies for mitigation:

1. Use the new `__attribute__((riscv_vector_cc))` attribute. In the RISC-V ELF psABI Document a new vector function calling ABI is described, and this has recently been implemented

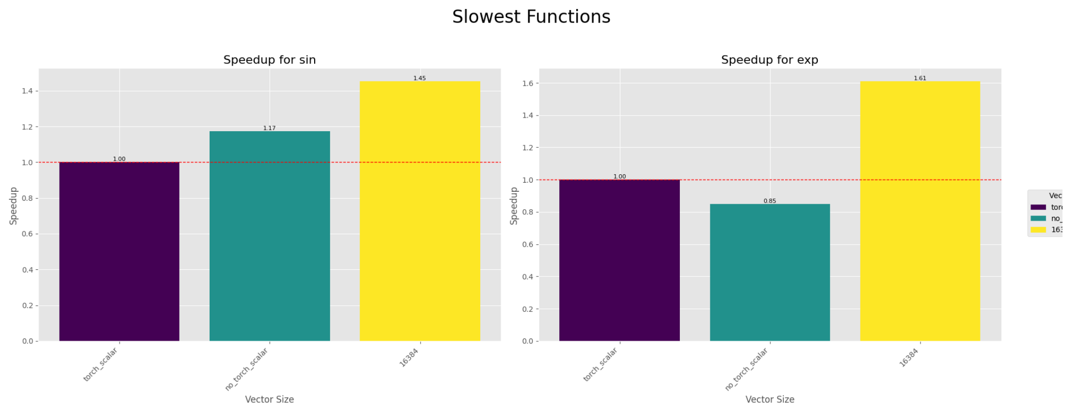

Figure 6: Sleef Functions with the slowest speedup obtained

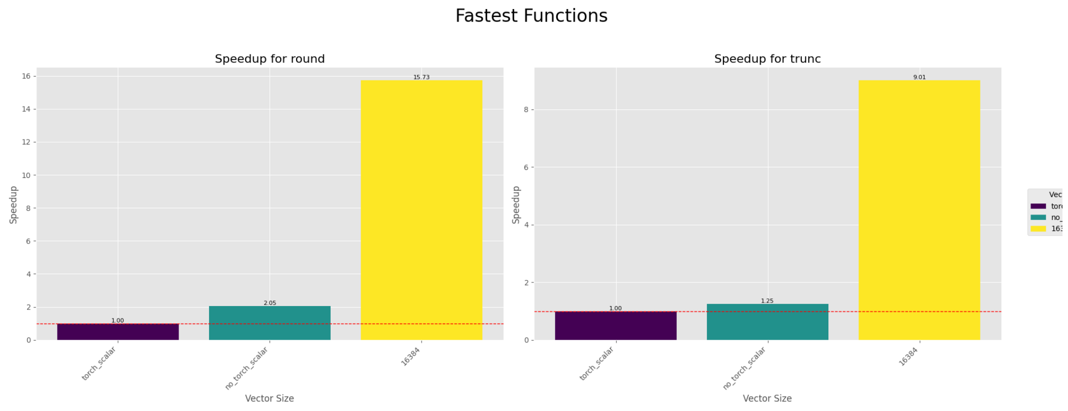

Figure 7: Sleef Functions with the fastests speedup obtained

in Clang. This could help Clang to avoid ever passing vectors via the stack, unless it absolutely has to.

2. Modify the rvv-rollback assembly translation to translate whole register stores to a smaller sized store. We would have to decide at compile what the "maximum" allowed vector length for our program is, which we have to do anyway when compiling Pytorch. Then we would translate whole register stores to only store that amount of bits. Remember that currently whole register stores are being translated into a sequence of instructions that at runtime queries the actual maximum size of the register, regardless of what was specified at compile time.

3. Ignore the problem and chose the maximum possible vector size for our final build. Then at least, while we are still doing unnecessary memory operations, at least all the data we load and store is actually used.

We didn't implement 1 because it would require a code change to Sleef, and we wanted to limit the amount of code changes in all our efforts, to allow our contribution to eventually be accepted into Pytorch. Implementing 2 led to a stack corruption and we abandoned this approach for the time being. Essentially what happens is that our "whole-register" stores, are no longer aligned with how Clang handles the stack pointer leading us to overwrite important parts of the stack in certain situations. As we only have the context of the current assembly instruction when doing our assembly translation step, we decided not to pursue this further. What we would actually need to do, is to also modify the handling of the stack pointer, so an optimization like this would likely have to be implemented in Clang itself. Thus we went with the 3rd option, and hope that with further compiler improvements, in the future there will be no unnecessary memory operations.

6.1.3 Matrix Multiplication

In most modern deep learning models, there are 3 fundamental operations that are the most compute intensive. Linear layers, which are a implemented via a single matrix multiplication, Attention layers which require generally 5-6 GEMM operations, and Convolutions, which while they can be implemented via GEMM as well for example with the Im2Col algorithm, are normally done using special direct kernels.

This means that an optimized matrix multiplication implementation is the backbone of any optimized deep learning framework.

Pytorch does not implement optimized routines for Matrix Multiplications itself, instead it makes calls into an optimized BLAS library, as described in the previous section. For the benchmarks, we are comparing a scalar implementation from OpenBLAS as the default against the RVV optimized version from BSC that is implemented in BLIS. We don't compare against a naive fallback implementation because these are very slow, and there is really never a situation where Pytorch is used without a BLAS library available.

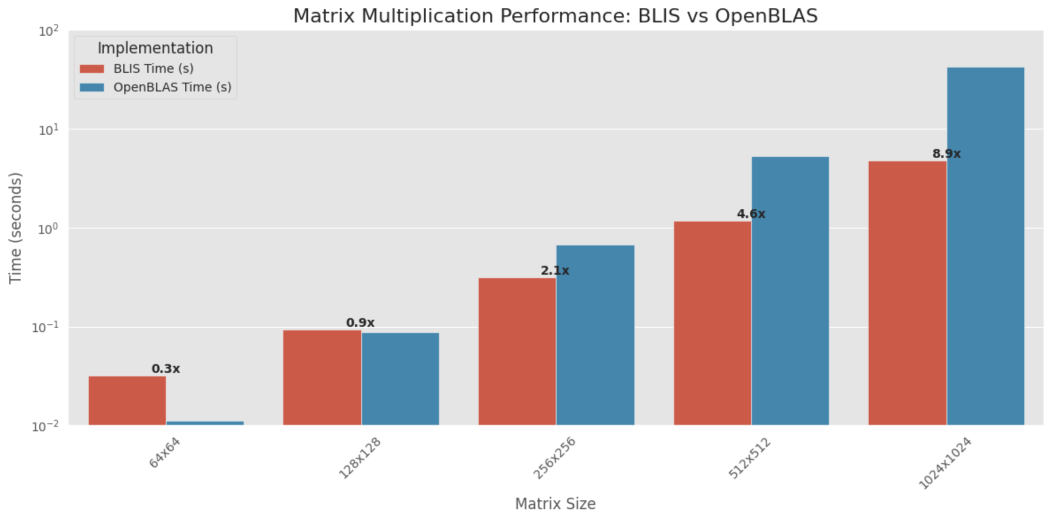

Figure 8: OpenBLAS vs BLIS matmul performance

We see that the scalar implementation of OpenBLAS is well optimized, as it employs various optimization techniques like micro-kernels and cache friendly memory layout. Slotin [2024]

Finally we see that the vectorized version is significantly faster, especially at larger matrix sizes.

Next we calculate the efficiency of the BLIS GEMM implementation, by comparing it to the theoretical maximum performance of the Vector Unit. In single precision we have a maximum performance of 0.8 GFlops/second, for double precision this halves to 0.4 GFlops/second.

We can see that especially the double precision implementation is well optimized, reaching over 70% of the theoretical maximum performance on the 2048x2048 matrix multiplication. For small matrix sizes the performance is very low, this is due to the function call overhead. Normally, when a large number of very small matrix multiplications is necessary, they are done in a batched fashion to eliminate this problem.

There is an even more optimized version of the BLIS library, that reaches over 80% efficiency on double precision and over 70% in single precision for the 2048x2048 case, but this version unfortunately suffers from a heap corruption for small matrices, that leads to a program crash. This is currently being fixed by the maintainer and we will integrate the fixed version into our Pytorch build as soon as it is available.

6.2 Layer Benchmarks

In this section we take a look at running full neural network layers. We have selected Attention, which is the backbone of all modern Language Models Vaswani et al. [2017], Convolutions, which are still the most common layer for Image Processing, and Linear Layers which are used in almost all Neural

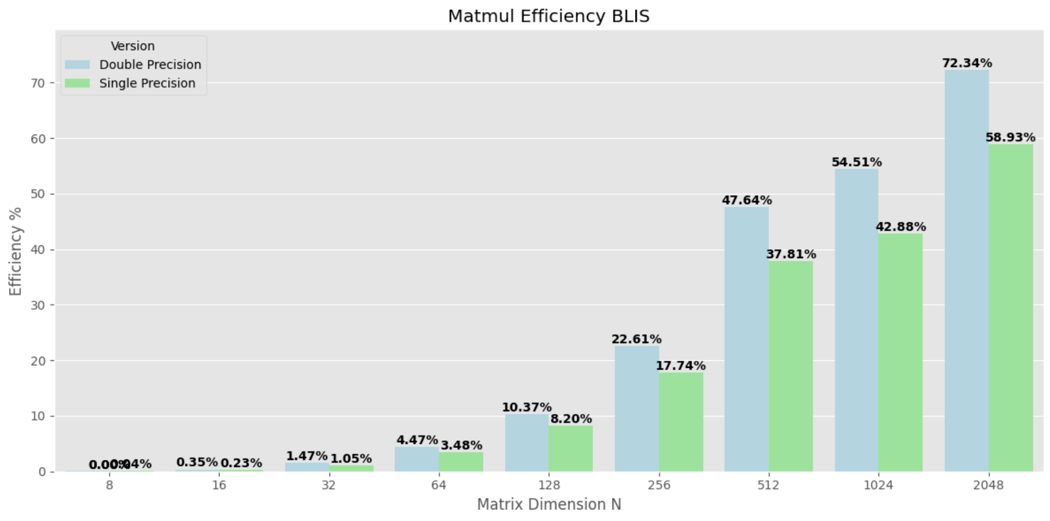

Figure 9: Single vs Double precision Efficiency in BLIS

Networks. The benchmarks are run in C++ and Python, but we only measure the execution time of the C++ function that does the actual computation, ie we don't include the Python overhead. For Linear and Attention, we are comparing between versions using the vectorized BLIS vs using the scalar OpenBLAS implementation, as the GEMM operation is the most expensive part of the operation, for the Convolution Layer we compare the naive ATen implementation provided by Pytorch against the vectorized OneDNN implementation developed at BSC.

6.2.1 Linear

For the Linear Layer we observe something similar to the matrix multiplication benchmark above: For small sizes, the OpenBLAS implementation is much faster, but eventually the BLIS implementation will overtake, and deliver significantly better performance with a 3.26x speedup at matrix dimension 2048x2048. This due to the aforementioned fact that the BLIS version with optimized small matrix multiplication suffers from a memory corruption bug at the moment, and we have to use the version that only optimizes larger Matrix Multiplication. Another thing that we have noticed is that using larger batch sizes hurts performance of the BLIS implementation a lot, this is something that we have to analyse further in the future.

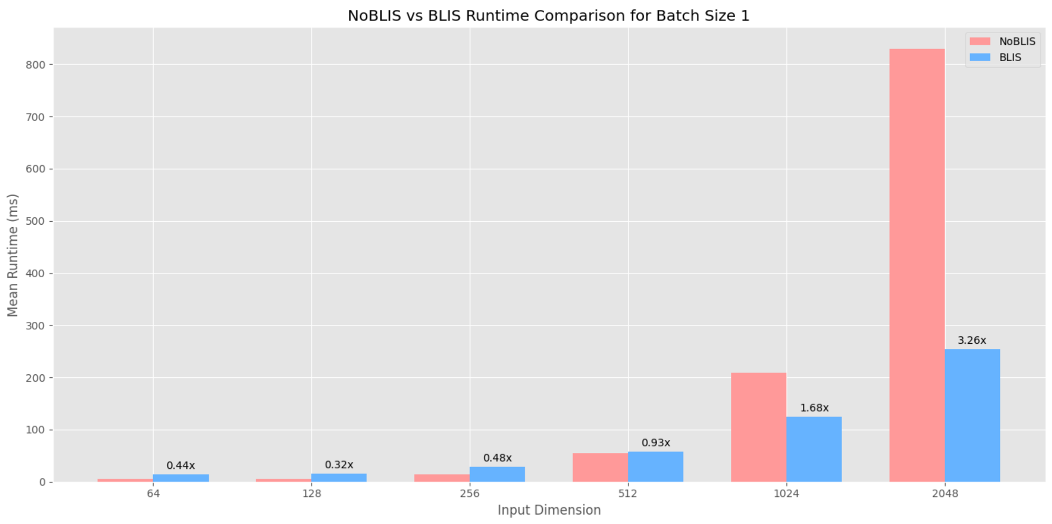

Figure 10: Linear Layer Performance with varying dimension

6.2.2 Attention

The attention layer was introduced in 2017 by Vaswani et al. [2017]. It computes 3 matrices from the input, the Keys (K), Queries (Q) and Values (V). Then it computes a dynamic weight matrix from the Query and Key matrices and multiplies this by the values. Compared to a Linear Layer, it can change the weights it uses depending the input, this is one way to explain its recent success.

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$
$$X \in R^{n\times d}, \quad W_Q, W_K, W_V \in R^{d\times d_k}, \quad Q, K, V \in R^{n\times d_k}$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

For the Attention layer we analyse the performance for varying batch size and sequence length. The results are similar to the previous results. Small problems that cannot take full advantage of the optimized BLIS GEMM operation fully are slower, but as the problem size grows the performance of the BLIS version overtakes the scalar OpenBLAS implementation.

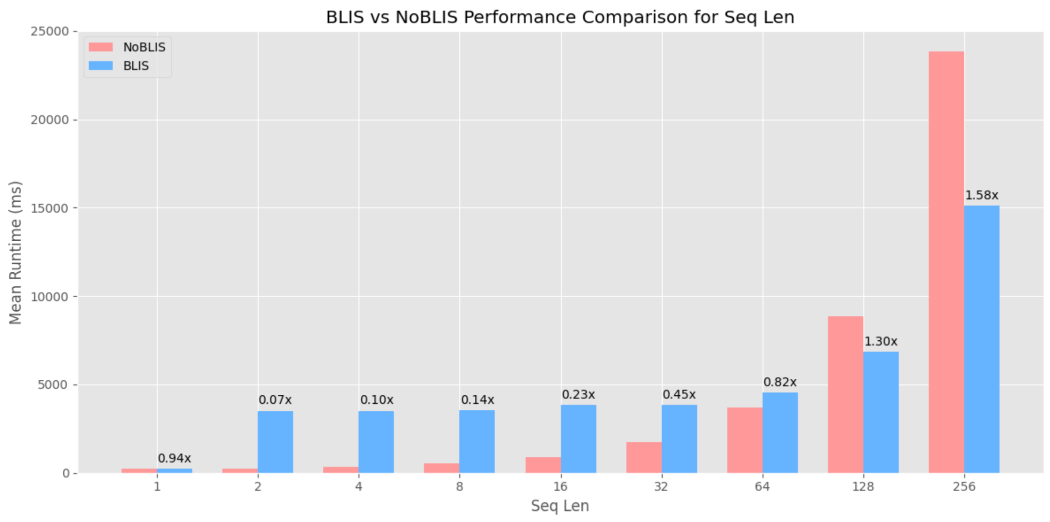

Figure 11: Attention Layer performance with varying Sequence Length

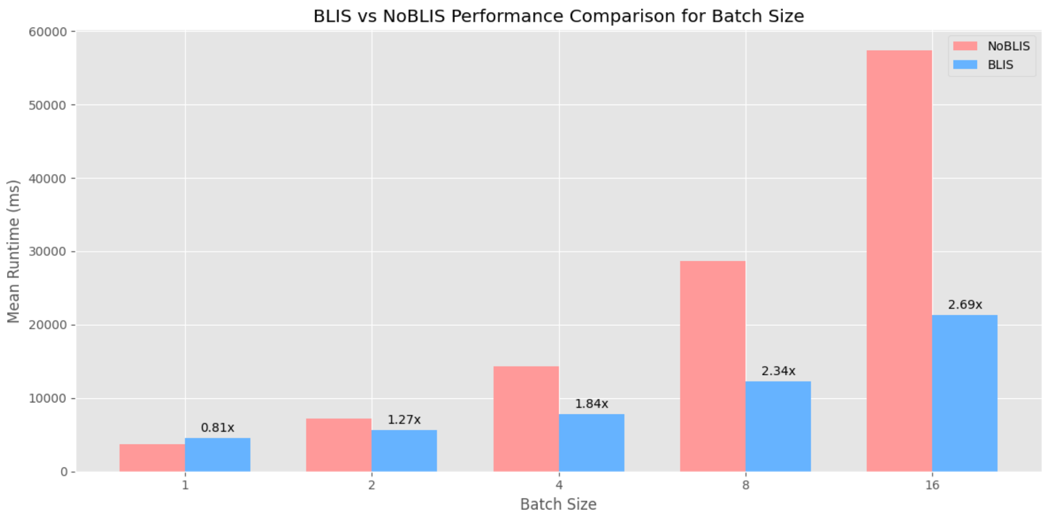

Figure 12: Attention Layer performance with batch size

6.2.3 Convolution

For the convolutional layers, we compare the performance of using the aforementioned optimized oneDNN implementation against the standard implementation of ATen, that is used if no specialized implementation is available. The optimized oneDNN implementation was significantly faster in all benchmarks, generally providing a speedup between 2x and 3x.

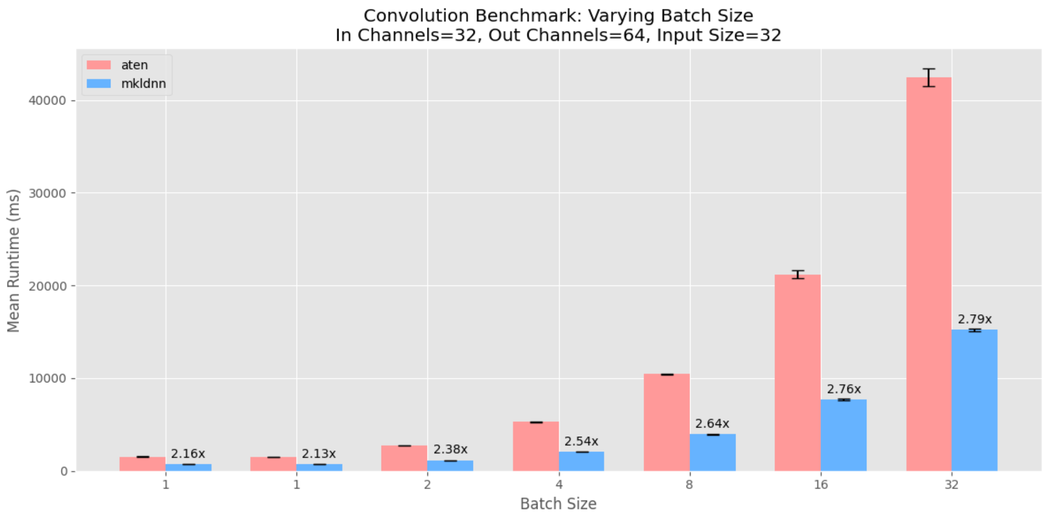

Figure 13: Convolution Layer performance with varying Batch Size

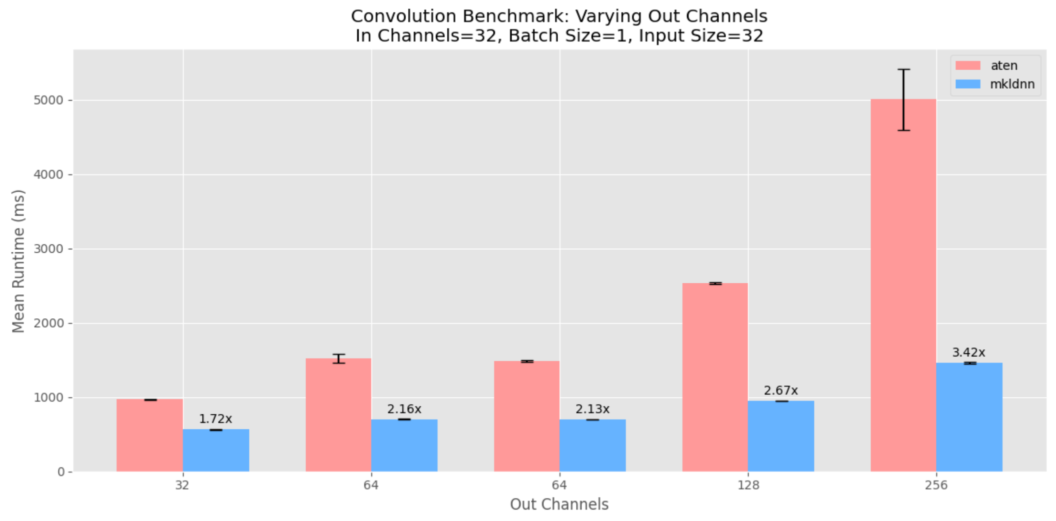

Figure 14: Convolution Layer performance with varying Output Channels

6.3 Model Benchmarks

Our initial goal was to run full, trained models on our hardware via Python. In this section we are reporting the performance results we obtained.

6.3.1 Language Models

We have chosen the tinyllama models for our benchmarking purposes. The main reason is that on our target platform, we only have 4 GB of RAM, thus all the bigger and more capable models will not fit into memory. For example LLama 7b requires 28 GB of memory to be loaded in single precision, and we don't have support for smaller data types. RISC-V has an extension which adds FP16 vector instructions, but this is not implemented on our CPU. Another reason why we chose smaller models is that we experience frequent crashes when we run compute intensive programs over longer periods of time. We were not able to determine the reason for these crashes, as the whole node it crashing and not our program. We suspect that it has something to do with the FPGA simulation the core.

We can run the models as follows:

```
from transformers import pipeline
pipe = pipeline("text-generation", model="nickypro/tinyllama-42M-fp32")
output = pipe("Input Text...")
```

In these Language model tests our results where disappointing. Our version of Pytorch with the optimized BLIS library only achieves 74.9 % of the performance of the version using scalar openBLAS (5.621 seconds vs 4.21 seconds) for the 15 million parameter model, and only 54.5 % of the performance on the 42 million parameter model. We attribute this poor performance to the fact that our BLIS version is not optimized for small matrix sizes, and these models are performing a lot of small matrix multiplications. We verify using the Pytorch Profiler that all of the performance difference is indeed due to slower calls to `aten::mm`, which then in turn either calls openBLAS or BLIS. Specifically, the 15m model uses an internal dimension (d) of 288, and the 42m model one of 512. The projection matrix multiplications of the Attention Layers have the following shape: $Q = XW_Q$, where $X \in \mathbb{R}^{seq_len \times d}$ and $W_Q \in \mathbb{R}^{d \times d}$. These matrix multiplications are precisely what is currently missing from our GEMM library.

We will evaluate these models again when BLIS has been updated, and we are especially interested in running larger models when a system is available with more memory and multiple cores, so we can run the larger Llama models.

6.3.2 Convolutional Neural Networks

We benchmark a convolutional neural network of the ResNet family He et al. [2015]. We chose the most common, the Resnet50. In our benchmarks, we compare the version using ATen with no vectorization against a version using the BSC optimized oneDNN.

In these full model benchmarks, we are using an older version of Pytorch and oneDNN, as in our main setup we experience node crashes that we not present during the individual layer runs. This will be investigated and resolved in the near future.

We test the execution time for a single 224x224 image, as this is the size that Resnet was originally trained for.

We observe a 6.11x speedup, which is very good. We explain the higher speedup compared to the individual layer test by the fact that the convolutions in Resnet are significantly larger than the individual ones that we tested before, thus oneDNN is even faster compared to the ATen implementation. We expect that when we resolve the node crashes the speedup will increase even more, as then more operations in the model will be vectorized.

7 Conclusion

In Conclusion, we have succeeded in overcoming the hardware and software limitations, and produced a version of Pytorch that is taking advantage of RISC-V Vector instructions. We have set up a complicated cross-compilation environment, mitigated the issue of the outdated RVV 0.7 specification on the target hardware, integrated already optimized numerical libraries into Pytorch, and evaluated the results. We have clearly seen that the vectorized operations are faster than the current scalar fallback implementations, but have also experienced that in runs of full models, the hardware limitations prevented us from seeing a similar speedup and there remain problems with the integration of the optimized libraries. Our benchmarks indicate that with the actual CPU chip, the full models will also experience a similar speedup as we have seen in the GEMM and ATen benchmarks.

The tooling for RISC-V, especially with RVV 0.7 is still limited, for example, most disassemblers are unable to disassemble the 0.7 instructions, which resulted in us having to manually decode offending instructions to understand what could be the reason for certain crashes. Luckily, all of this is quickly improving and RVV 1.0 will not suffer these issues as it is being more and more integrated into GCC and Clang and the GCC binutils software collection.

The future of RISC-V and RVV looks bright, with more and more companies and institutions releasing software, tools, simulators and CPUs, and the EU is putting special emphasis on the development of this technology. Developing, implementing and creating a good software environment for a new ISA is a hard and slow process, but we believe that the hardest parts have already been done and that we will see more RISC-V based system on the whole spectrum of computation, from energy efficient embedded devices to high performance supercomputers.

This project was much more challenging than initially anticipated. The actual code was implemented and passed the Pytorch test suite early on in QEMU simulation (RVV 1.0). By far the largest part of the internship was spend on the cross compilation setup, compiling all dependencies in the correct version, implementing the RVV 1.0 -¿ 0.7 translation and resolving many problems that arose in the process of actually executing the code on the target hardware, like crashes, and unimplemented instructions in the CPU.

All in all we think that this was a successful project, but the work is not yet finished. In the next section we present some ideas for future work we want to focus on next.

8 Future Work

The obvious first step will be to improve the integration of BLIS and oneDNN, so that we can harness their full potential also in full model runs, as currently, while we are getting promising results especially in the layer benchmarks, we are not yet getting good speedups when executing full, trained models.

Next, we want to run the evaluations on a more performant, multi-core version of the CPU. This will also allow us to perform a strong and weak scaling analysis, and determine how well the vectorized RVV implementations interact the with OpenMP parallelization that is employed by Pytorch. Furthermore, our current implementation is able to take advantage of the new torch.compile compiler, which captures a graph during the first execution of a Deep Learning model via a Python interpreter hook, and then generates a full C++ version of the model using oneDNN, BLIS and the ATen library. We were able to execute this in a QEMU environment, but we currently cannot run it on the target hardware, as it requires to compile using Clang and our compilation setup is on a different machine. Another direction would be to compare the performance of our version of Pytorch with different deep learning inference frameworks that can take Pytorch models and execute them. We are particularly interested in comparing inference performance, as CPUs are mostly used for inference, and in almost all cases of deployed deep learning systems, the inference cost will outweigh the initial training cost. As one interesting candidate, we want to compare against the TVM deep learning compiler, as that allows us to compare our version that was optimized with intrinsics, against a compiler based approach that uses autovectorization.

Of course, we want to contribute our vectorized ATen version back to the Pytorch main repository, for this it will be necessary to integrate our version into the Automated Testing environment of the project. Our ATen Vectorized implementation passes all tests on QEMU, but on the target hardware some functions are not able to run due to SIGILL, as certain instructions of the ISA are not implemented in the current hardware design. This problem will be resolved with the upcoming hardware update to RVV 1.0.

9 References

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016. URL https://arxiv.org/abs/1605.08695.

Marc Casas Alexandre de Limas Santana, Adrià Armejach. Efficient direct convolution using long simd instructions. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP 2023, pages 342–353, 2023. URL https://dl.acm.org/doi/10.1145/3572848.3577435.

Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024. doi: 10.1109/MM.2024.3373763.

Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015. URL https://api.semanticscholar.org/CorpusID:206594692.

Joseph K. L. Lee, Maurice Jamieson, and Nick Brown. Backporting risc-v vector assembly, 2023. URL https://arxiv.org/abs/2304.10324.

MetaAI. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

OpenAI. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703.

Naoki Shibata and Francesco Petrogalli. Sleef: A portable vectorized library of c standard mathematical functions. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1316–1327, 2020. doi: 10.1109/TPDS.2019.2960333.

Sergey Slotin. Matrix multiplication, 2024. URL https://en.algorithmica.org/hpc/algorithms/matmul/. Online; accessed 11-September-2024.

Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015. URL https://doi.acm.org/10.1145/2764454.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Pablo Vizcaino, Georgios Ieronymakis, Nikolaos Dimou, Vassilis Papaefstathiou, Jesus Labarta, and Filippo Mantovani. Short reasons for long vectors in hpc cpus: A study based on risc-v. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1543–1549, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. doi: 10.1145/3624062.3624231. URL https://doi.org/10.1145/3624062.3624231.

Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, December 2019. URL https://riscv.org/technical/specifications/. Document Version 20191214-draft.