École Normale Supérieure de Lyon

Master 1 Computer Science

Internship report

Vectorization of deep learning kernels

Enrique GALVEZ

Under supervision of
Marc CASAS
Alexandre DE LIMAS SANTANA

Laboratory : Barcelona Supercomputing Center

EPI team

1^{st} of May 2023 – 21^{st} of July 2022

Contents

Context and motivations

In the dynamic realm of artificial intelligence, deep learning algorithms have emerged as a game-changer, bringing with them new technologies and new challenges. However, the growing complexity of models and datasets has placed a considerable computational strain on these algorithms. To meet this challenge, the field of High-Performance Computing (HPC) has been actively investigating new ways to accelerate computations, by creating new computer architectures or by creating high performance applications on already existing ones.

Because of the highly parallel nature of deep learning operations, the option of Single Instruction, Multiple Data (SIMD) parallelism can be very interesting for such algorithms. Moreover, because vector architectures are equipped with CPUs supporting efficient SIMD instructions, they may offer very interesting speedups on deep learning computations.

Over the course of the history of vector architectures, there has been a trend towards increasing vector length, which is the size of SIMD parallelism. It began with Intel MMX [1], offering a fixed vector length of 64 bits, providing moderate levels of parallelism. Subsequently, AVX (Advanced Vector Extensions) and AVX2 [2] architectures raised the bar with a vector length of 256 bits, enhancing computational capabilities.

As the demand for even more significant computational power intensified, AVX512 emerged with a vector length of 512 bits, further amplifying the parallel processing potential. The first vector architecture with scalable vector length were the SVE [3] (Scalable Vector Extension) architectures, offering vector lengths between 128 bits and 2048 bits. Currently, under development, the EPI (European Processor Initiative) RV64V [4] architecture takes this trend to new heights by implementing a device where vector length is scalable and can be set up to 16384 bits.

In this context of evolving vector architectures, it can be interesting to propose efficient algorithms for long vector architectures, in our case the EPI RV64V architecture. My work takes place in the continuity of Alexandre De Limas Santana's work [5], which focus on proposing efficient convolution algorithms for such architectures. Indeed, the significant speedups obtained by vectorizing the convolution layers have motivated this work, which propose efficient vector algorithms for 3 other deep learning operations: ReLU, pooling and batch normalization.

Besides proposing efficient vector algorithms for these layers, this work also aims to establish principles for programming efficient vector algorithms. These principles will serve as guiding factors for conceiving algorithms that exploit the full potential of long vector architectures.

1 Methodology

1.1 Target architecture

The target architecture is the EPI RV64V architecture, a vector architecture being developped for the EPI project [6]. As a vector architecture, it implements SIMD parallelism in an efficient way.

As a reminder, SIMD parallelism (Single Instruction, Multiple Data) is a type of parallel processing technique used in computer architectures to perform the same operation on

multiple independent data elements simultaneously. Instead of processing data sequentially, SIMD parallelism allows a single instruction to operate on a vector of data elements in parallel, greatly accelerating computation for tasks that can be parallelized.

Our work especially targets long vector architectures, especially EPI RV64V processor, which can be reprensented in the Figure 1 below. As we can see on the sketch, the L2 cache feeds the vector registers, which contains the data processed in parallel by the processing units (PU).

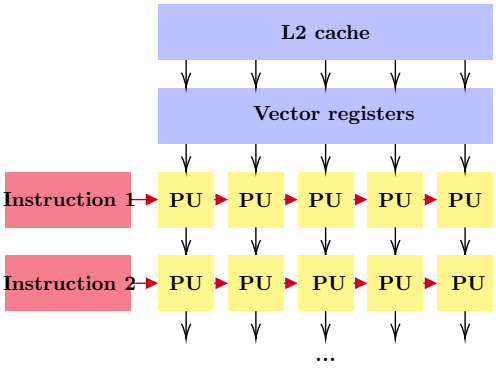

Figure 1: EPI RiscV Vector processor

It is important to keep in mind that the vector length is scalable, up to 16384 bits which represents 512 × the size of an element of type float32.

1.2 Programming model

In order to program efficient algorithms on vector architectures, we have to use specific instructions in order to tell the architecture how to perform the SIMD instructions. In our case, it is made, using the **EPI Intrinsincs** [7] instruction set. This set contains instructions that can be used to tell the compiler which computations should be done in parallel and how.

Figure 2 shows a list of instructions proposed in EPI Intrinsincs:

Set vector length

- gvl = vsetvl(rvl, sew, lmul)

SIMD arithmetic operations:

- vr = vfadd(v1, v2, gvl)
- vr = vfmul(v1, v2, gvl)
- vr = vfmadd(v1, v2, v3, gvl)
- ...

Memory accesses:

- vr = vLoad(src, gvl)
- vStore(v_data, dst, gvl)
- vr = vIdxLoad(v_idx, src, gvl)
- vIdxStore(v_idx, v_data, dst, gvl)
- ...

SIMD relational operations: (return a mask)

- meq = vmfeq(v1, v2, gvl)
- mge = vmfge(v1, v2, gvl)
- ...

Masked operations:

- v3 = vfadd_mask(v_def, m1, v1, v2, gvl)
- v1 = vLoad_mask(v_def, m1, src, gvl)
- ...

Figure 2: EPI Intrinsincs instruction set

We can see in Figure 2 that we have at our disposal different kinds of instructions. A very important one is `vsetvl` which allows us to set the vector length. We can also perform SIMD arithmetic operations, using for example the above examples, working with floating-point operands. Regarding the memory accesses, we will divide them in two kinds: the "contiguous" loads/stores (`vLoad/vStore`[1]) which allows to load contiguous data from memory and secondly the "gather/scatter" (`idxLoad/idxStore`) which allows to load/store memory from sparse locations specified in an index vector.

1.3 Exploiting the full potential of vector architectures

After understanding how SIMD parallelism and vector architectures work, it is possible to establish some factors that seem to play a decisive role in the performance of vector algorithms.

The first one is **preferring loading contiguous data to gather operations**. Indeed, the architecture is made in such a way that loading data in sparse locations is way slower than just taking some contiguous array and loading it into vector registers. [8]

A second one is **improving cache-level data reuse**. Indeed, as L2 cache feeds the vector registers, the device is able to keep vectors in cache to reuse it for further computations instead of having to re-load it.

A third one is **maximizing the vector length of used vectors**. Indeed, the architecture is able to make computations on up to 512 operands of type float32 so using a smaller vector length will mean using less computational power as we could have. In the following, we call that "waste of vector length".

Overall, we can also take a look to smart **code generation** in order to minimize the final number of operations performed by the hardware.

In the following, we will try to apply as much as possible these four principles while trying to vectorize deep learning kernels.

1.4 Execution platform

This work aims at optimizing deep learning algorithms. But, in order to work directly on real use cases, all my algorithms will be integrated into OneDNN [9], an open-source HPC library developped by Intel. OneDNN is designed to accelerate deep learning workloads on a variety of hardware architectures, including CPUs, GPUs, and FPGAs. Written

[1] sometimes noted as `contLoad/contStore` in the following

in C++, OneDNN allows me to write algorithms directly in C++ with the EPI Intrinsincs instructions. Then, the application BenchDNN can be used to test the correctness of the algorithms and run experiments on real networks.

Moreover, to run the experiments presented below, I had access to real EPI RV64V machines. However, this hardware is still in development and some operations such as integer operations are still not implemented. For the algorithms that required integer computations (especially for indices), I had to use a simulator.

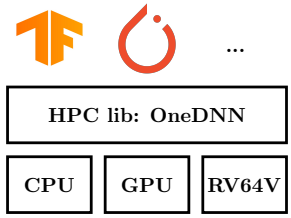

Figure 3: OneDNN library

Figure 3 shows how an HPC library such as OneDNN interacts with widely used machine learning libraries (Tensorflow, Pytorch...) by providing a fast implementation of the computations requested by these high level libraries, optimized for the detected hardware.

2 The ReLU primitive

2.1 How to vectorize the ReLU primitive ?

A simple example of a vectorized program is the forward ReLU operation.

The formula for the forward ReLU operation is:

$$dst[n,c,h,w] = \begin{cases} src[n,c,h,w] & \text{if } src[n,c,h,w] > 0 \\ \alpha \times src[n,c,h,w] & \text{if } src[n,c,h,w] \leq 0 \end{cases}$$

The pseudo code of the ReLU operation is simple:

```
for (n = 0 to MB)
    for(c = 0 to C)
        for(h = 0 to H)
            for(w = 0 to W)
                out[n,c,h,w] = (in[n,c,h,w] > 0) ? in[n,c,h,w]
                    : (in[n,c,h,w] * α);
```

Listing 1: forward non-vectorized ReLU on a tensor of sizes (MB,H,W,C)

In this code, we observe that the computation of each value in the destination tensor is independent from the others. This allow us to vectorize the loops by assigning to each element of the tensor an index in a vector. For this kind of computation, the vectorized code is simple because the vector instructions follows the sequential operations.

```
loop_size = MB * H * W * C;
int gvl = 0;

for (i = 0; i < loop_size; i += gvl) {
    // compute size of vectors
    gvl = vsetvl(loop_size-i);
    // load vectors
    vf32    vin = vload(&(in[i]), gvl);
    vf32 vzeros = vbroadcast(0.0f,  gvl);
    vf32 valpha = vbroadcast(alpha, gvl);
    // test positivity -> pos is a mask
    vi1 pos = vmfge(vin, vzeros, gvl);
    // mul masked by pos
    vin = vfmul_mask(vin, vin, valpha, pos, gvl);
    // store in memory
    vstore(&(out[i]), vin, gvl);
}
```

Listing 2: forward vectorized ReLU on a tensor of sizes (MB,H,W,C)

In the piece of code above (Listing 2), we can see some important concepts in algorithm vectorization:

- gvl is given by the hardware thanks to `vsetvl` operation
- We can use broadcasts for the values involved in all the computations
- Condition evaluation returns a mask, i.e. a vector of 0 and 1 values
- The final result is computed using an instruction mul_mask with vin as default value

Moreover, this algorithm is our first chance to apply one of our 4 principles stated in the last part. Indeed, because tensor size depends on the problem, we can not guarantee that the last iteration of a vectorized loop will use the full vector length. So, in order to maximize the vector length among the computations, it is important to decrease as much as possible the number of times when the algorithm is at the last iteration of a vectorized loop. Here, the solution is to "collapse" the loops and vectorize among the collapsed loop. By doing this, we execute only once the vectorized loop, so there remains only one situation involving a waste of vector length.

2.2 Performance improvements

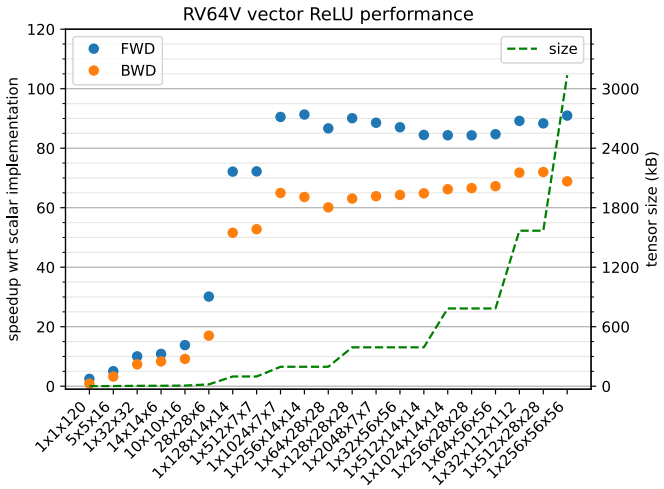

Figure 4: Performance of the ReLU operation

Figure 4 shows the performance of ReLU operation on a real RV64V hardware. The test cases are taken from real netwworks: ResNet [10] and Yolo [11]. We can see that code vectorization brings an execution time up to 90 times faster than the scalar implementation. We also see that the tensor size has a high impact over the speedup. The peak speedup is obtained for test cases where the tensor size is greater than 192kB.

3 The pooling primitive

3.1 Introducing the pooling operation

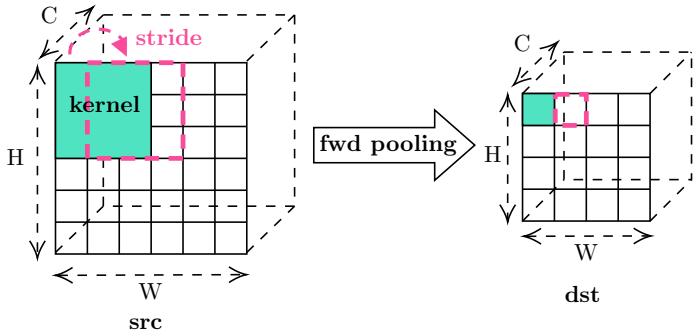

Figure 5: The forward pooling operation

Figure 5 shows how the pooling operation works. Each destination pixel is computed as the maximum or the average among a certain kernel in the source image.

This operation involves a lot of memory accesses and the vectorization of pooling might be slightly more difficult than the vectorization of ReLU because of the absence of the direct correspondence between a source pixel and its equivalent in the destination.

In order to elaborate vectorized algorithms for the pooling operations, we need to understand how the data is organized in memory.

3.2 About memory formats

Memory formats are used to describe the way data is organized in memory. We refer to a memory layout using letters describing the different dimension, ordered in the same way as the dimensions are organized in memory.

As an example, the Figure 6 below shows the 2 common memory formats:

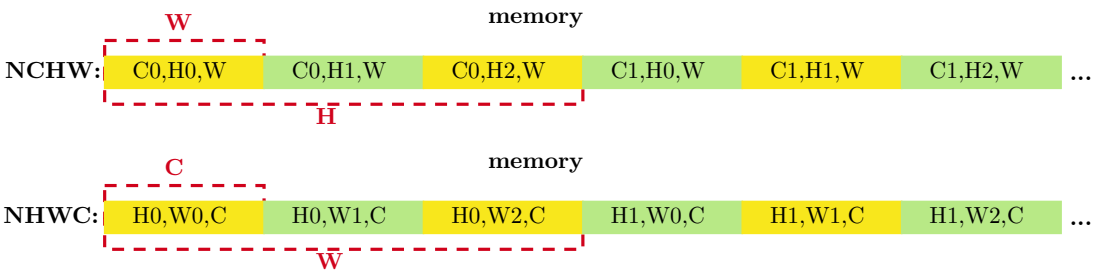

Figure 6: The 2 most common memory formats

An important thing to note is that with NHWC format, for fixed N,H and W, the elements among the C dimension are contiguous in memory, while for NCHW format, the elements among the W dimension are the ones being contiguous in memory for N,C and H fixed.

Using an appropriate memory format is crucial for vector algorithms because it will decide if we can use contiguous loads or if we will be forced to use indexed loads which are slower. This being said, we observe that the pooling operation is an occasion to put in application the first of the four principles stated in the introduction.

3.3 Vectorizing pooling operations for an appropriate memory format

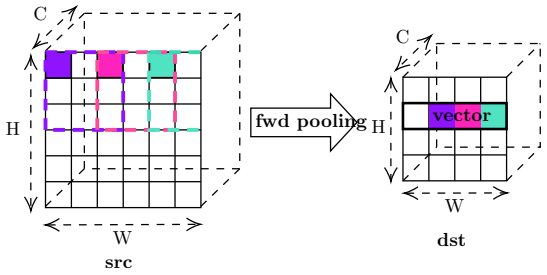

Figure 7: FWD pooling: IdxLoad algorithm

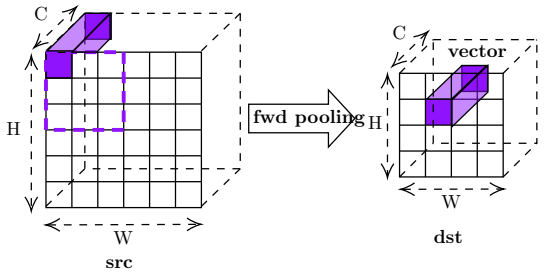

Figure 8: FWD pooling: ContLoad algorithm

A first natural algorithm which we can think about to vectorize the pooling operation is the **idxLoad** algorithm, represented above in figure 7. In this version, we vectorize among the pixels in the destination tensor and we use vector instructions to compute the max or the average among the kernel. In order to follow the vector-length principle, the loops among the destination tensor are collapsed. As we can see it in the figure, the data to gather from the input tensor may not be contiguous, so this version requires indexed loads to work. We can also note that this algorithm works for any memory format.

Secondly, in order to follow the first principle enounced previousy, we should prefer contLoads to idxLoads. In order to do so, I proposed the **contLoad** algorithm, represented in Figure 8. This algorithm works only for NHWC format and takes advantage of the fact that C is inner-most to do contiguous loads. Something to note here is that contLoads algorithm involves vectorizing only among the C dimension.

3.4 Performance improvements

Because the hardware which ran the experiments was still in developpment, it had no support for vector integer computations. This is the reason why I had to use the MUSA simulator in order to evaluate the performance of the **idxLoads** algorithm. The output of the simulator gives the number of CPU cycles needed for the execution of the application in an appropriate vector machine. The bigger the number of cycles is, the slower the algorithm is.

This being said, the first graph (Figure 9) shows that the **idxLoad** is way slower than **contLoad** algorithm in most cases. This confirms that indexed loads are slower than contiguous vector loads.

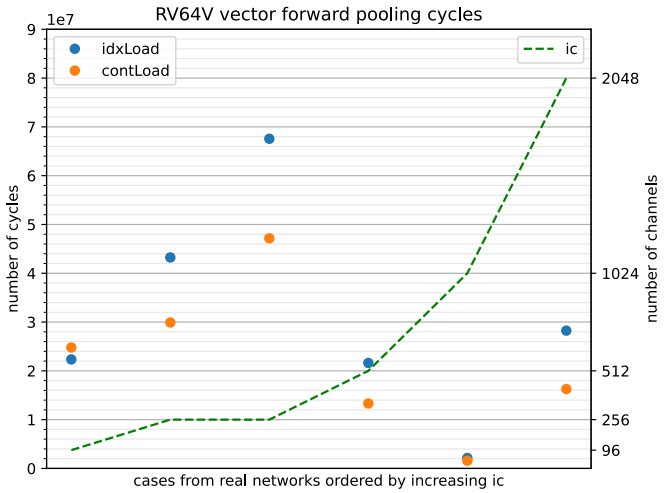

Figure 9: FWD RV64V pooling cycles

However, to be able to do contiguous vector loads, we have to reduce the size of vectorized dimension. Indeed, when **idxLoad** algorithm has vectors among all the loops gathered, **contLoad** version only vectorize the C dimension. This is the reason why, for cases with very small C dimension, the benefice of contLoads is not enough to balance the benefice of collapsing all the loops.

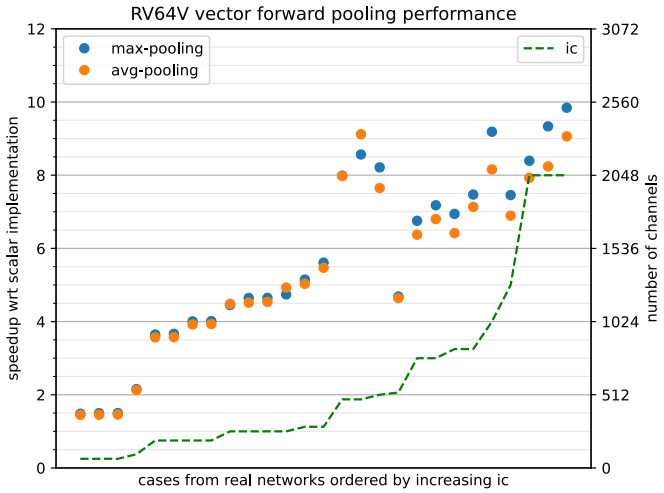

Figure 10: FWD RV64V pooling performance

Overall, the algorithm to keep here is **contLoad**, because contiguous vector loads allow to really take advantage of the vector architecture. Compared to sequential algorithm on Figure 10 on real hardware, it brings up to 10 times speedups.

4 The batch normalization primitive

4.1 Introducing sequential batch normalization

Batch normalization can be described by the formula below:

$$dst(n,c,h,w) = \gamma(c) \cdot \frac{src(n,c,h,w) - \mu(c)}{\sqrt{\sigma^2(c) + \varepsilon}} + \beta(c),$$

Where

- $\gamma(c), \beta(c)$ are optional scale and shift for a channel
- $\mu(c), \sigma^2(c)$ are mean and variance for a channel
- ε is a constant

The key factor for vectorizing this primitive is how we access the values in γ, β, μ and σ^2. Indeed, without these arrays, batch normalization would have been an elementwise operation, such as ReLU.

We can imagine a first vectorized algorithm for batch normalization:

```
for(n,h,w){
 for(c=0; c<C; c+=gvl){ // vectorized
   gvl = vsetvl(C - c);
   v_gamma  = contLoad(gamma[c]);
   v_mu     = contLoad(mu[c]);
   v_sigma2 = contLoad(sigma2[c]);
   v_beta   = contLoad(beta[c]);
   v_eps = vbroadcast(epsilon);
   v_src = contLoad(src);
   v_dst = v_gamma * (v_src - v_mu) / sqrt(v_sigma2 + v_eps) + beta;
   contStore(v_dst);
 }
}
```

Listing 3: Forward batch normalization, **naive** algorithm

This algorithm is made for NHWC in order to do contiguous loads and stores on source and destination tensor. Moreover, algorithm is vectorized among the C loop to allow contiguous loads on the arrays of the C value.

4.2 Maximizing the use of vector length

According to our principles, a first issue with **naive** algorithm is the fact that the vectorization among C dimension may involve a poor use of vector length.

In order to improve the use of vector length for cases with small IC, we want to collapse

However, the issue in batch normalization which prevent us from doing loop collapsing are the arrays of the C values which we should load. Basically, there are several ways to

access these arrays but the point of this first algorithm is to make these access in a smart way in order to allow loop collapsing and, by doing so, maximizing the use of vector length.

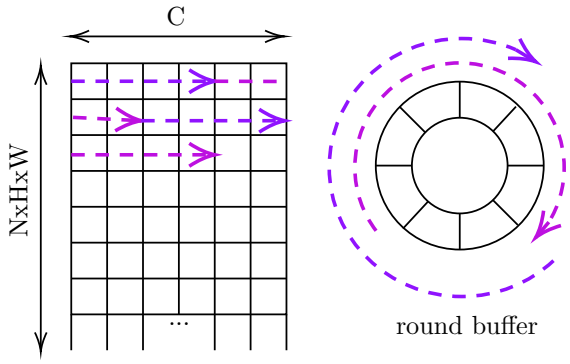

Figure 11: the roundBuffer

The idea of the algorithm is to use a roundBuffer (see Figure 11) to store the statistics and allowing us to access them using contiguous loads, even if the starting point of the contLoad is not the index 0. In the following, we call **roundBuffer** this algorithm.

```
buff_size = C + maxvl - gcd(C, maxvl);
extendBuffersTo(buff_size);

for(i=0, i < N*C*H*W, i+=gvl){ // vectorized
  gvl = vsetvl(N*C*H*W - i);
  v_eps = vbroadcast(epsilon);
  // load contiguously the arrays
  v_gamma  = contLoad(buff_gamma[i%C]);
  v_mu     = contLoad(buff_mu[i%C]);
  v_sigma2 = contLoad(buff_sigma2[i%C]);
  v_beta   = contLoad(buff_beta[i%C]);
  // compute dst
  v_src = contLoad(src[i]);
  v_dst = v_gamma * (v_src - v_mu) / sqrt(v_sigma2 + v_eps) + beta;
  contStore(v_dst[i]);
}
```

Listing 4: FWD bnorm roundBuffer

The Listing 4 shows how the roundBuffer algorithm works. It starts by an initialization of the roundBuffers by extending the arrays to the size $buff\_size = C + max\_vl - gcd(C, max\_vl)$. Then, at any point of the next loop, the algorithm is still able to perform contLoads to load the data in these arrays.

The Figure 12 shows the performance of roundBuffer algorithms. We can see that it provides good speedups over the naive version, especially for low values of IC. This is coherent with the fact that this algorithm is made to prevent the "waste of vector length" which can occur when vectors are not fully used, which is the case for small IC.

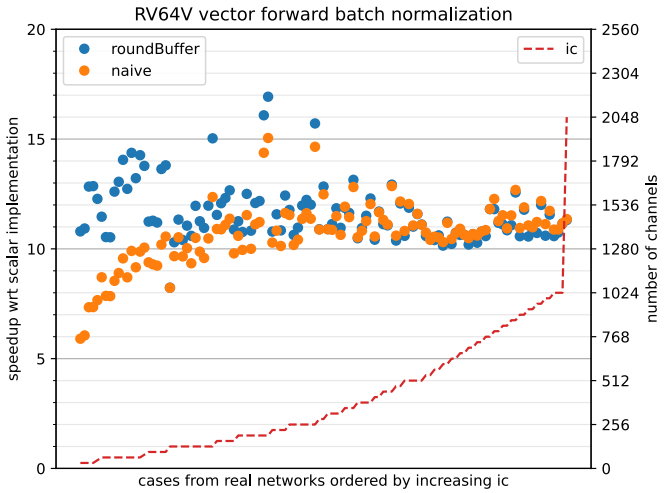

Figure 12: Performance of roundBuffer algorithm

However, its performance is equivalent to naive's around IC=vector_length and it became slower than naive for big IC cases.

Despite the good results we had, they encourage us to look for even better algorithms.

4.3 Improving cache-level data reuse

As I said previously, naive algorithm was non optimal regarding the vector length used. But it was also non-optimal regarding the locality of the data used in each computation.

We can indeed notice that we do not need to fill the destination tensor in order, and we can, by reordering the loops, fill this tensor while keeping in memory the values among C variable. This results in the **swapLoops** algorithm, in Listing 5.

```
for(c=0; c<C; c+=gvl){ // vectorized
  gvl = vsetvl(C - c);
  v_gamma  = contLoad(gamma[c]);
  v_mu     = contLoad(mu[c]);
  v_sigma2 = contLoad(sigma2[c]);
  v_beta   = contLoad(beta[c]);
  v_eps = vbroadcast(epsilon);
  for(n,h,w){
   v_src = contLoad(src);
   v_dst = v_gamma * (v_src - v_mu) / sqrt(v_sigma2 + v_eps) + beta;
   contStore(v_dst);
 }
}
```

Listing 5: FWD bnorm roundBuffer

In Figure 13, we can observe the expected very good speedup obtained by swapLoop

over roundBuffer and naive. However for cases with very small IC, roundBuffer is still better because swapLoops does not maximize the vector length. These results show the importance of a good data reuse, which was our third principle in the introduction. Moreover, we see in these results that a good data reuse can provide a faster algorithm than vector length in some cases.

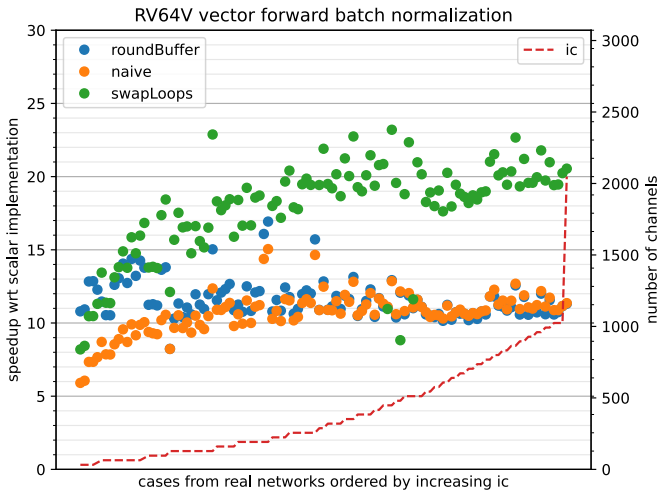

Figure 13: RV64V batchNorm performance

4.4 Decreasing the number of operations

Now we have a fast algorithm with **swapLoops**, we can try to apply the last principle: reducing the number of operations. A first way to do this is to think about the way we do the arithmetic operations to compute the batch normalization formula.

Listings 6 and 7 shows how we can decrease this number of operations, by changing the way the result is computed.

```
v_sqrt_var = vsqrt(v_sigma2 + v_eps);
for(n,h,w){
    v_src = contLoad(src);
    v_dst = v_gamma * (v_src - v_mu) / sqrt(v_sigma2 + v_eps) + beta;
    // 4 instructions
    // v_dst = vfsub(v_src, v_mu)
    // v_dst = vfmul(v_dst, v_gamma)
    // v_dst = vfdiv(v_dst, v_sqrt_var)
    // v_dst = vfadd(v_dst, v_beta)
    contStore(v_dst);
}
```

Listing 6: Original swapLoops

```
v_sqrt_var = v_gamma / vsqrt(v_sigma2 + v_eps);
v_beta = v_mu * v_sqrt_var - v_beta
for(n,h,w){
    v_src = contLoad(src);
    v_dst = v_sqrt_var * v_src - v_beta;
    // 1 instruction
    // v_dst = vfmsub(v_src, v_sqrt_var, v_beta)
    contStore(v_dst);
}
```

Listing 7: swapLoops + aritOpt

Another way to decrease the amount of operations is loop unrolling. This technique takes place during the compilation and it allows the compiler to reduce the number of `jump` operations and it also allows overlapping between arithmetic operations and memory accesses. Listing 9 shows how an unrolled by 4 loop would look like in C.

```
for(i=0; i < N*H*W; ++i){
  compute(v_src);
  // v_src = contLoad(src)
  // v_src = v_gam_sqrt_var * v_src
  //     - v_mu_beta
  store(v_src);
  // contStore(v_src)
}
```

Listing 8: non unrolled algorithm

```
for (i=0; i < N*H*W/4; ++i) {
    compute(v_src0);
    compute(v_src1);
    compute(v_src2);
    compute(v_src3);
    store(v_src0);
    store(v_src1);
    store(v_src2);
    store(v_src3);
}
i *= 4;
for (; i < N*H*W; ++i) {
    compute(v_src0);
    store(v_src0);
}
```

Listing 9: Unroll4 algorithm

4.5 Performance improvements

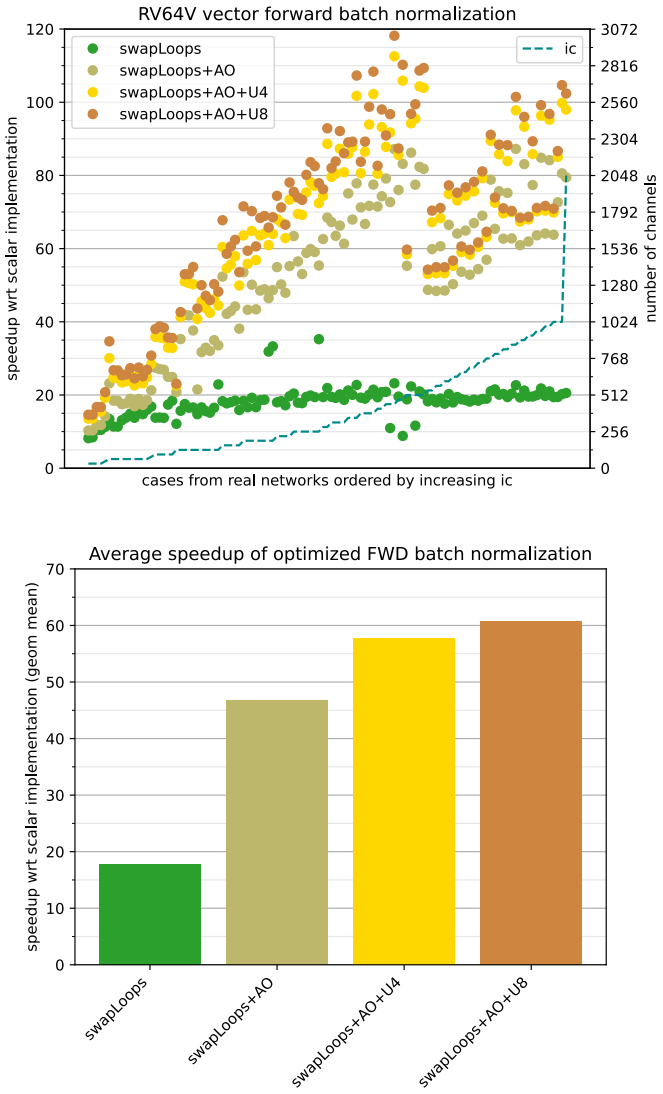

Figure 14: RV64V batch normalization performance

In the graphs above, AO stands for "aritOpt" and UX stands for "Unrolled by X".

We can see in Figure 14 the outstanding results that we got by applying all of the previously mentioned optimizations. It shows that for an algorithm with a high level of data reuse, reducing the number of operations can provide very high speedups (here in average 60 times faster than the scalar implementation, and in average 4 times faster than swapLoops without optimizations).

Conclusion

In this report, we have explored the efficient programming of deep learning algorithms on vector architectures, with a specific focus on a long vector architecture: EPI RV64V. Our investigation centered on three critical operations in deep learning networks: ReLU, Pooling, and batch normalization.

Through the conception of vector algorithms for these operations, we have identified a set of principles playing a vital role in maximizing computational performance, thereby leveraging the full potential of vector architectures.

The first and most crucial principle emphasizes the preference for loading contiguous data over "gather" operations. By strategically loading data into vector registers from contiguous arrays, we exploit the architecture's design, enabling faster data access and minimizing overhead associated with sparse data loading.

The second principle emphasizes improving cache-level data reuse. Leveraging the L2 cache to feed vector registers enables the device to maintain vectors in cache for subsequent computations, reducing the need for redundant data reloading.

Next, maximizing the vector length of used vectors emerges as the third principle. As the architecture allows computations on up to 512 operands of type float32, using a smaller vector length would result in a waste of computational power. Hence, optimizing vector length is essential to fully harness the computing capabilities.

Overall, we acknowledge the significance of smart code generation. By reducing our code to a minimal amount of CPU instructions, we can greatly improve the performance of our algorithms.

In conclusion, this work sheds light on the paramount importance of adhering to these principles in designing efficient algorithms for vector architectures. By combining these insights and optimizing our approach to ReLU, Pooling, and batch normalization operations, we have paved the way for improving the support of deep learning computations on the specific EPI RV64V architecture.

References

[1] A. Peleg, S. Wilkie, and U. Weiser, "Intel mmx for multimedia pcs," *Commun. ACM*, vol. 40, p. 24–38, jan 1997.

[2] C. Lomont, "Introduction to intel advanced vector extensions," *Intel white paper*, vol. 23, pp. 1–21, 2011.

[3] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[4] community, "Risc-v vector extension." https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc.

[5] A. d. L. Santana, A. Armejach, and M. Casas, "Efficient direct convolution using long simd instructions," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, (New York, NY, USA), p. 342–353, Association for Computing Machinery, 2023.

[6] F. Minervini, O. Palomar, O. Unsal, E. Reggiani, J. Quiroga, J. Marimon, C. Rojas, R. Figueras, A. Ruiz, A. González, J. Mendoza, I. Vargas Valdivieso, C. Hernández Calderón, J. Cabre, L. Khoirunisya, M. Bouhali, J. Pavon, F. Moll, M. Olivieri, and A. Cristal, "Vitruvius+: An area-efficient risc-v decoupled vector coprocessor for high performance computing applications," *ACM Transactions on Architecture and Code Optimization*, vol. 20, 12 2022.

[7] E. processor initiative, "Epi intrinsincs reference." https://ssh.hca.bsc.es/epi/ftp/doc/intrinsics/EPI/epi-intrinsics.html.

[8] C. Rodrigues, A. Phaosawasdi, and P. Wu, "Simdization of small tensor multiplication kernels for wide simd vector processors," in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'18, (New York, NY, USA), Association for Computing Machinery, 2018.

[9] Intel, "Oneapi deep neural network library." https://oneapi-src.github.io/oneDNN/.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[11] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.