Vectorization of deep learning kernels

Enrique GALVEZ

Under supervision of:
Marc CASAS
Alexandre DE LIMAS SANTANA

Barcelona Supercomputing Center

May - July 2023

Overview

1. Motivations and context
2. Methodology
   - Target architecture
   - Programming model
   - Execution platform
3. Vectorized kernels
   - The ReLU operation
   - The pooling primitive
   - The batch normalization primitive
4. Conclusion

Overview

1. Motivations and context
2. Methodology
   - Target architecture
   - Programming model
   - Execution platform
3. Vectorized kernels
   - The ReLU operation
   - The pooling primitive
   - The batch normalization primitive
4. Conclusion

Motivations and context

Deep learning algorithms:

- Became essential in most AI tasks
- Costly in terms of computation
- Example case for HPC research

Vector architectures history:

Intel MMX	Fixed vector length of 64bits	
AVX/AVX2	Fixed vector length of 256 bits	
AVX512	Fixed vector length of 512 bits	
SVE	Scalable vector length between 128 and 2048 bits	
EPI RV64V	Scalable vector length up to 16384 bits	(**target of this work**)

Goal:
How to program efficient deep learning algorithms on architectures with long vector length ?

Overview

1. Motivations and context

2. Methodology
   - Target architecture
   - Programming model
   - Execution platform

3. Vectorized kernels
   - The ReLU operation
   - The pooling primitive
   - The batch normalization primitive

4. Conclusion

Target architecture

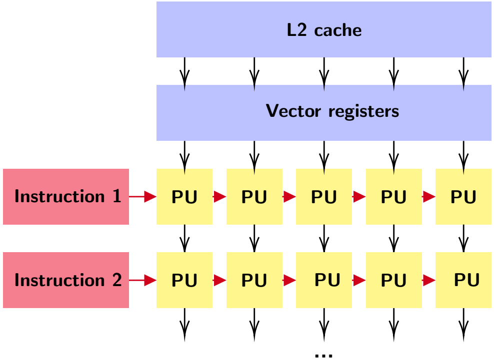


Figure: EPI RiscV Vector processor

EPI RiscV Vector processor:

- Implements SIMD parallelism across "vectors" of data
- Scalable vector length up to 16384 bits (512 $\times$ float 32)
- L2 Cache feeds the vector registers

Programming model

EPI's RISC-V Vector Extension Intrinsic:

- set vector length
- SIMD arithmetic operations
- SIMD relational operations
- memory accesses (loads/stores)
- masked operations
- ...

Key points that appeared to influence the performances of vector algorithms:

- **Data locality** Decrease memory latency (contLoads faster than idxLoads)
- **Vector length utilization** Use as much processing units as possible
- **Register-level data reuse** Improve memory accesses to decrease required memory bandwidth
- **Code generation** Helps reducing the number of instructions executed in PUs

Execution platform

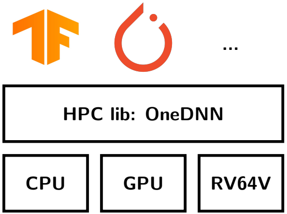


Figure: OneDNN library

How a deep learning application works ?

- Creates a model using a high level library (Tensorflow, Pytorch...)
- The high level library requests computations to a HPC library such as OneDNN
- OneDNN selects an implementation depending on architecture and operation parameters

Execution environment:

- Real RV64V machine: FPGAs (in development)
- MUSA simulator for features still unimplemented in the FPGAs

Overview

The ReLU operation

The formula for forward ReLU operation is:

$$d = \begin{cases} s & \text{if } s > 0 \\ \alpha \times s & \text{if } s \leq 0 \end{cases}$$

Where:

- s is the source pixel
- d is the destination pixel
- α is a fixed hyperparameter

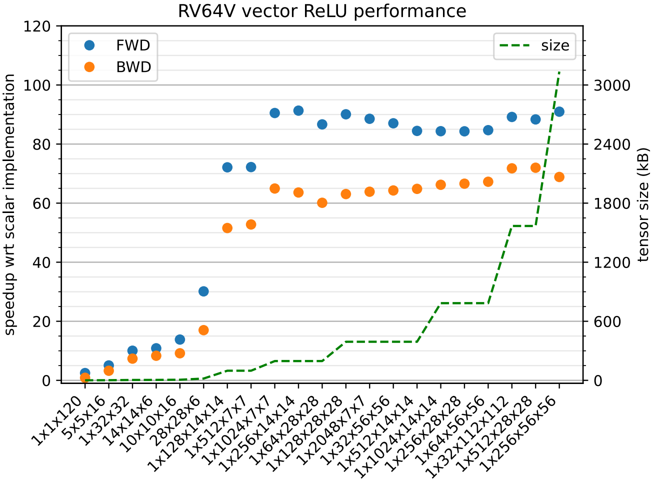
Scalar code:

```
for (i = 0 to MB * H * W * C)
    out[i] = (in[i] > 0) ? in[i] : (in[i] * α);
```

Vectorized code:

```
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);
}
```



Experiment:

- Test cases taken from real networks: ResNet and Yolo

Observations:

- Code vectorization brings up to $90\times$ speedup
- The size of the tensor has a high impact over the performance
- Peak performance is obtained for tensors of size around 192kB

The pooling primitive

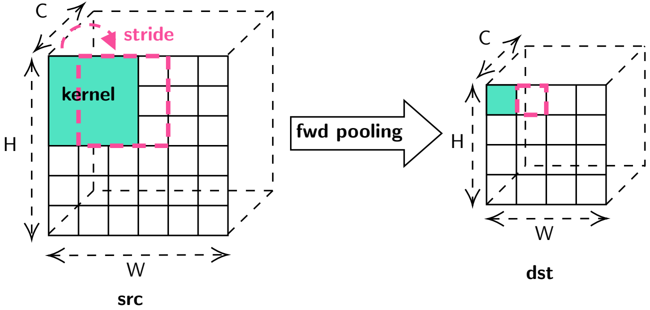


Figure: The pooling operation

Algorithm:

- Each output pixel is computed as the average or max of the input pixels among the kernel

Remarks:

- Involves a lot of memory accesses
- The speed of memory accesses will be decisive in the speed of the vectorized algorithm

Note: In order to access memory in the most optimal way, it is important to understand **memory formats**.

The pooling primitive

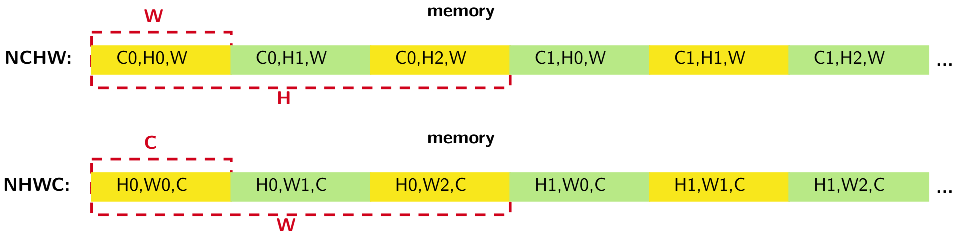


Figure: The 2 most common memory formats

What is the memory format of a tensor ?

- It is the way the tensor (multidimensional object) is stored in memory (linear)
- It specifies which dimension is stored contiguously and which aren't

The pooling primitive

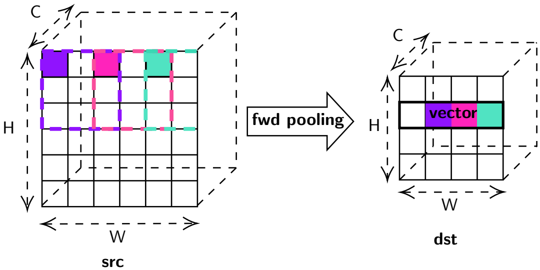


Figure: FWD pooling: IdxLoad algorithm

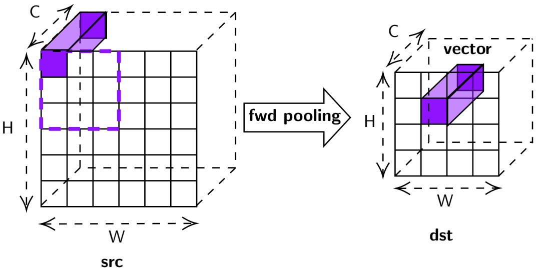
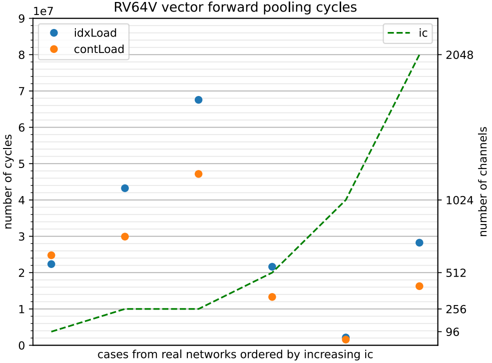


Figure: FWD pooling: ContLoad algorithm

- Works for any memory format
- Loads aren't contiguous
- Need to compute kernel indices for each output pixel

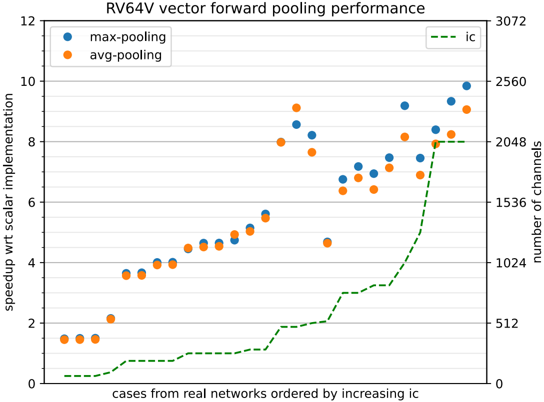
- Works only for NHWC
- Since C is inner-most, loads can be contiguous
- Kernel indices can be computed once and used among C dimension

The pooling primitive



- Cycles obtained with MUSA simulator
- idxLoad requires more cycles → slower
- Difference due to the locality of contiguous memory accesses

- Speedups depends on the size of vectorized dimension



The batch normalization primitive

Batch normalization formula:

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

- Almost an eltwise primitive
- Difference with eltwise: arrays over C

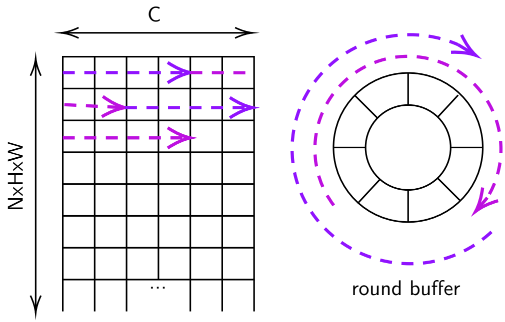
naive algorithm:

- Works for NHWC format
- Vectorize among C
- use contLoads for the arrays over C

FWD bnorm **naive** for NHWC (using contLoads):

```
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);
 }
}
```

The batch normalization primitive



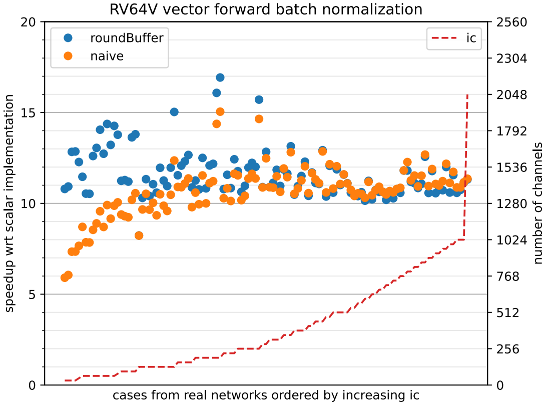
- Allow contLoads of channels values
- Maximize the use of vector length
- Optimization for naive targetting the cases where IC is small

FWD bnorm **roundBuffer** for NHWC:

```
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]);
}
```

The batch normalization primitive



Experiment:

- Realized on a RV64V machine (FPGA)
- Test set from benchDNN including real network sizes (ResNet, GoogLeNet, ...)

Conclusions:

- Roundbuffer is way better than naive for small IC
- For high IC, this optimization is useless
- Are there other optimizations for high IC ?

The batch normalization primitive

swapLoops algorithm for FWD batch normalization:

Algorithm:

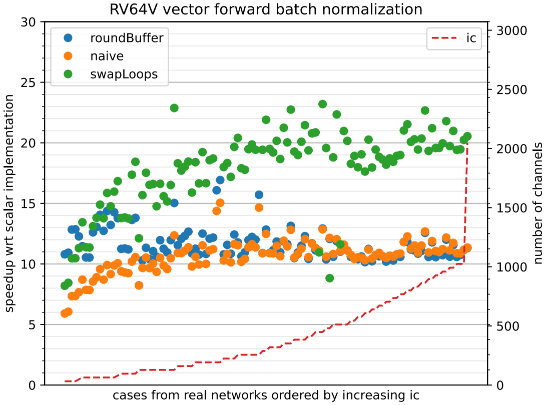
- Works for NHWC format
- Is an optimization for *naive*
- Loops are reordered

It improves data locality:

- Statistics loads are moved outside the inner-most loop
- Improved register-level data reuse

```
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);
 }
}
```

The batch normalization primitive



Experiment:

- Realized on a RV64V machine (FPGA)
- Test set from benchDNN including real network sizes (ResNet, GoogLeNet, ...)

Conclusions:

- swapLoops is significantly better than roundBuffer for $IC > 128$
- For these cases, improving data locality is more important than using full vector length
- swapLoops can still be improved

The batch normalization primitive

Original **swapLoop** algorithm:

```
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);
}
```

- 4 instructions in the nhw loop

swapLoops+aritOpt algorithm:

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

- Uses expanded arithmetic expression
- Only 1 operation remains inside nhw loop.

The batch normalization primitive

Loop unrolling:

- Optimization at the compilation level
- Allow overlapping between arithmetic operations and memory accesses

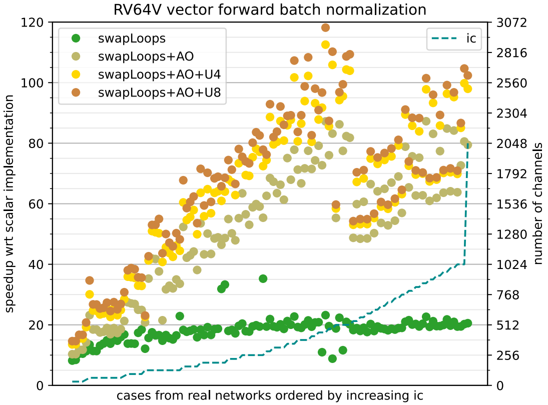
Non unrolled algorithm:

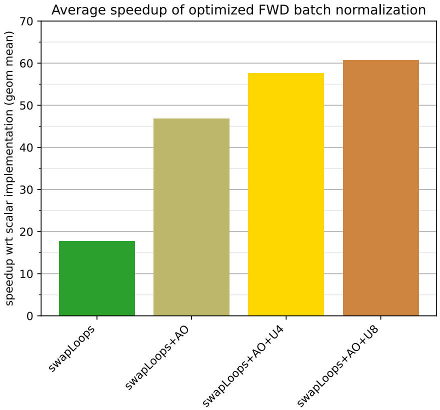
```
for(i=0; i < N*H*W; ++i){
  compute(v_src);
  // v_src = contLoad(src)
  // v_src = v_gam_sqrt_var * v_src - v_mu_beta
  store(v_src);
  // contStore(v_src)
}
```

Unroll4 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);
}
```

The batch normalization primitive





Experiments:

- Realized on RV64V machines (FPGA)
- Test set from benchDNN

AO arithmetic optimization

UX UnrollX algorithm

Overview

Conclusion

To program efficient algorithms on vector machines, it is important to consider:

The **memory format** used

- Defines the algorithm memory access pattern
- Can allow contiguous loads, faster than gathers

Maximizing the use of **vector length**

- Maximize hardware use during the computations
- Not useful for any problem shape

Keeping a good **data locality**

- Decrease the number of memory accesses
- Very important for memory-bounded algorithms

Loop unrolling and other optimization at compilation level:

- Can bring some additional speedups when the high-level code is already in its best version
- Can be difficult to implement

References

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.

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.

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.

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.

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