Received: 22 December 2021 | Revised: 25 May 2022 | Accepted: 24 August 2022

DOI: 10.1002/cpe.7424

SPECIAL ISSUE PAPER

WILEY

Acceleration with long vector architectures: Implementation and evaluation of the FFT kernel on NEC SX-Aurora and RISC-V vector extension

Pablo Vizcaino[1] | **Filippo Mantovani**[1] | **Roger Ferrer**[1] | **Jesus Labarta**[1,2]

[1]Barcelona Supercomputing Center, Barcelona, Spain

[2]Universitat Politècnica de Catalunya, Barcelona, Spain

Correspondence
Pablo Vizcaino, Barcelona Supercomputing Center, Plaça Eusebi Güell, 1-3, 08034 Barcelona, Spain.
Email: pablo.vizcaino@bsc.es

Summary

Novel architectures leveraging long and variable vector lengths like the NEC SX-Aurora or the vector extension of RISCV are appearing as promising solutions on the supercomputing market. These architectures often require re-coding of scientific kernels. For example, traditional implementations of algorithms for computing the fast Fourier transform (FFT) cannot take full advantage of vector architectures. In this article, we present the implementation of FFT algorithms able to leverage these novel architectures. We evaluate these codes on NEC SX-Aurora , comparing them with the optimized NEC libraries; and in a prototype of a RISC-V core with a vector processing unit. We present the benefits and limitations of two approaches of RADIX-2 FFT vector implementations. We show that our approach makes better use of the vector unit of the NEC SX-Aurora , reaching higher or equal performance than the optimized NEC library. More generally, we prove the importance of maximizing the vector length usage of the algorithm, taking advantage of the FFT properties to reduce long-latency vector operations, and reordering the instructions according to the specific hardware features to boost the performance of FFT-like computational kernels.

KEYWORDS

fast Fourier transform, FFT, NEC SX-Aurora, RISC-V, SIMD, vector acceleration

1 | INTRODUCTION

Accelerated computing is becoming more and more relevant in high-performance computing (HPC). The limitation to the performance improvements imposed by the slow-down of Moore's law applied to general purpose CPUs has made HPC architects looking for solutions that can complement the computational power delivered by standard CPUs (i.e., accelerators). The most visible example of this are GP-GPU based systems, that populate 3 places within the first 5 most powerful supercomputers in the world (Top500).

GP-GPUs, however are not the only approach to acceleration: the use of vector or SIMD extensions is becoming more and more relevant in HPC systems. Beside the AVX-512 SIMD extension by Intel, we detect appearing on the market the first CPU implementing the Arm SVE extension (Fujitsu A64FX, ranked first in the Top500) and the NEC SX-Aurora vector engine, a discrete accelerator leveraging vector CPUs able to operate with registers of up to 256 double precision elements. On top of this market movements, we cannot ignore the RISC-V architecture which recently ratified v1.0 of the V-extension, boosting vector computation from the academic world and the open-source community.

The efficient use of vector accelerators often require to adapt or rewrite classical algorithms to exploit their full computing power. In most cases, vendor specific libraries coupled with optimized compilers allow to port large HPC codes to vector accelerators in a relatively smooth way.

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.
© 2022 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

For portability reasons however, scientists often look for open-source libraries including kernels already optimized for specific architectures. The computation of the Fourier transformation using the FFT algorithms is an example of a relevant HPC kernel extremely used by the HPC community. For this reason we focused this article on the design and the evaluation of non-parallel vectorized FFT implementations.

The main contributions of this article are: (i) we developed four implementations of the FFT algorithms targeting large vector architectures; (ii) we evaluate our FFT codes on the NEC SX-Aurora accelerator and a RISC-V core leveraging a vector extension, analyzing benefits and limitations of each architecture with an in depth study of the single-core performance; (iii) we compare our performance results with the vendor library distributed by NEC; (vi) we extend the scheduling capabilities of the RISC-V compiler so to optimally leverage the underlying hardware.

This article is an extension of Reference 1: we include new algorithmic improvements of our FFT implementations, a more comprehensive evaluation on vector architectures (including a RISC-V based vector system), and the development of scheduling policies oriented to improve the performance of the code generated by the LLVM compiler employed on both vector architectures.

The remaining part of the article is structured as follows: Section 2 compiles the related work in the field of FFT implementations for HPC systems; Section 3 briefly presents the NEC SX-Aurora accelerator and the RISC-V system employed in our evaluation; Section 4 analyzes the optimizations targeting large vector architectures; Section 5 includes the measurements gathered on NEC SX-Aurora and the RISC-V architecture; Section 6 closes the article with general remarks and conclusions.

2 | RELATED WORK

FFT is a kernel of paramount importance in several algorithms of scientific computing. Therefore, a large body of research about FFT optimization on many architectures has been published in the last decades. The key reference publications used as background for our implementations are the book of Chu et al.[2] the article of Pease,[3] and the article of Swarztrauber.[4]

More recently, the research community is focusing on developing efficient FFT implementations targeting emerging architectures with different degrees of parallelism, for example, high number of cores and long SIMD or vector units. Chow et al.[5] report their effort in taking advantage of the IBM Cell BE for the computation of large FFTs; Anderson et al.[6] make use of FPGAs for accelerating 3D FFTs; Wang et al.[7] present an FFT optimization for Armv8 architectures; Malkovsky et al.[8] evaluate FFTs on heterogeneous HPC compute nodes including GP-GPUs. Most of those studies are limited to up to 8-elements SIMD units in CPUs or high thread-level parallelism in GPUs while the implementations proposed in our article are targeting wider vector units.

Bailey[9] and Swarztrauber[4] studied various FFT algorithms, including Pease's and Stockham's, for the firsts vector computers which were limited by their inefficiency accessing non continuous data. The algorithms they propose have a minimum vector length of $\sqrt{N}$ at best, which is lower than our algorithm's $\frac{N}{8}$. Moreover, our implementations propose an exploitation of the data locality in the many vector registers that the SX-Aurora has, reducing the accesses to the main memory. Furthermore, our method extends the approach of Franchetti et al.[10] since we explore larger FFT sizes as well as double precision data types.

Promising results for acceleration with the NEC SX-Aurora accelerator have been shown for SpMV in Reference 11 and for spectral element method for fluid dynamics in Reference 12. We extend those evaluation efforts of NEC' accelerator with FFT. This article continues the work done in the thesis from Pablo Vizcaino Serrano.[13]

3 | HARDWARE PLATFORMS

We evaluate our implementations on two systems capable of using long vectors: the NEC SX-Aurora and a RISC-V core that uses the RISC-V Vector extension (RISC-V V).

3.1 | The NEC SX-Aurora

We implemented and evaluated our FFT codes targeting the NEC SX-Aurora VE (VE), the latest NEC's long vector architecture which combines SIMD and pipelining. Vector units and vector registers use a 32 × 64-bit wide SIMD front in an 8-cycles deep pipeline resulting in a maximum vector length of 256 × 64-bit elements or 512 × 32-bit elements. The VE10B processor used for this publication was presented at the IEEE HotChips 2018,[14] and the first performance evaluation was described in the same year.[15]

Each of the 8 VE cores consists of a scalar processing unit (SPU) and a vector processing unit (VPU) and is connected to a shared last level cache (LLC) of 16 MB. Three fused multiply-add vector units deliver a peak performance of 269 GLFOPS (double precision) per core at 1.4 GHz. The peak performance of the used VE variant is 2.15 TFLOPS delivering a byte/FLOP ratio of 0.56.

Vector engines are integrated as PCIe cards into their host machines. Programmers can use languages like C, C++, Fortran, and parallelize with MPI as well as OpenMP, while accelerator code can still use almost any Linux system call transparently. The proprietary compilers from NEC support

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

automatic vectorization aided by directives. They are capable of using most features of the extensive vector engine ISA[*] from high-level languages loop constructs. For the work presented in this article, we employed the open-source LLVM-VE project[†], which supports intrinsics allowing tight control over VE features to operate with complex numbers, control vector registers, and LLC cache affinity.

3.2 | RISC-V core with vector processing unit (VPU)

We use an experimental setup for the evaluation of the RISC-V Vector extension (RISC-V "V") which is a platform composed by an FPGA and a host-x86 server. The server is a commodity server used to program the FPGA and to communicate with it, without any effect in the evaluation of our implementation. The FPGA board is the Virtex UltraScale+ HBM VCU128 FPGA Evaluation Kit [‡], powered by a VU37P FPGA [§].

Within the FPGA logic are mapped a RISC-V core tightly coupled with a VPU. The VPU has eight lanes, each having a floating point unit (FPU) connected to the RISC-V scalar core. Each vector register has a width of up to 16,384 bits per vector (256 double precision elements). The computational throughput of the system is 16 double precision FLOPs/cycle and it runs at a frequency of 50 MHz.

The scalar core is called Avispado and has been developed by Semidynamics[¶]. The VPU is called Vitruvius and has been developed by the Barcelona Supercomputing Center[#]. The interconnection of the different RTL components as well as the L2 cache has been developed by the Foundation for Research and Technology-Hellas[‖].

The system supports a lightweight Linux kernel based on busybox providing basic functional operations. We leverage an LLVM-based compiler developed at the Barcelona Supercomputing Center which supports builtins for vector instructions[**]. As with the compiler for the NEC SX-Aurora , autovectorization is also supported, but our implementations look to exploit the vector instructions directly via builtins.

4 | IMPLEMENTATION

Our implementations are centered around 1-Dimensional FFTs, as our work serves as a starting point in vectorizing the FFT kernel for these new architectures. All implementation ideas from this article can be later applied to an FFT with more dimensions.

There exist multiple algorithms for the computation of the FFT, each with its benefits and disadvantages from the computational point of view. In this article, we focus on a subset of algorithms, those that are denominated RADIX-2. Considering an FFT with N being the number of transformed elements, a RADIX-2 FFT requires N to be a power of two and divides the required computation into $\log_2(N)$ phases. With a higher RADIX, the number of complex multiplications needed to do the transform is reduced, but the memory access patterns' complexity increase. Since one of the biggest challenges when vectorizing for large vector lengths are memory access, we decided to stick with the relative simplicity of RADIX-2. FFT algorithms are also split into in-place and out-of-place, with the latest requiring an additional buffer alongside the input and output arrays. All implementations proposed in this article are out-of-place since our objective is an efficient vectorization and not a reduced memory footprint. Moreover, some FFT algorithms require a permutation of the resulting elements and others are self-sorting. In this article, we study both approaches.

All implementations in this article are designed for complex double-precision data. The visual representations of the algorithms shown in this article are simplified, presenting only the real component because the computation of the imaginary component is conceptually equivalent to its real counterpart.

For the FFT calculation, we often refer to *twiddle factors*. W is the set of the twiddle factors, which are complex exponents computed as $\mathrm{tf}(k, N) = e^{\frac{-2\pi ik}{N}}$, with $k \in \{0, N-1\}$.

4.1 | Pease FFT

4.1.1 | Description

The first vectorized implementation is the FFT algorithm developed by Pease.[3] In terms of arithmetic operations, each phase of a naive Pease's FFT implementation requires $N/2$ additions, $N/2$ subtractions, and $N/2$ multiplications. One critical downside of Pease's algorithm is the permutation requirement at the end of the last phase. Modern vector ISAs offer instructions to load and store scattered data, but they are typically less efficient than those that operate on contiguous or constant-strided data.

Pease's algorithms is characterized by a constant geometry, that means that the same elements are operated in each of the $\log_2(N)$ phases. More specifically, the first half of the N elements operate with the second half of each phase. This leads to a potential $N/2$ elements that can be operated simultaneously (i.e., vector length of $N/2$). Once a phase has been calculated, the vector registers no longer hold the first and second half of the N elements, so they must be shuffled. Not all vector architectures have an in-register shuffle instruction (RISC-V has vrgather.vv), in this case it must be emulated using memory scatter and gather instructions.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

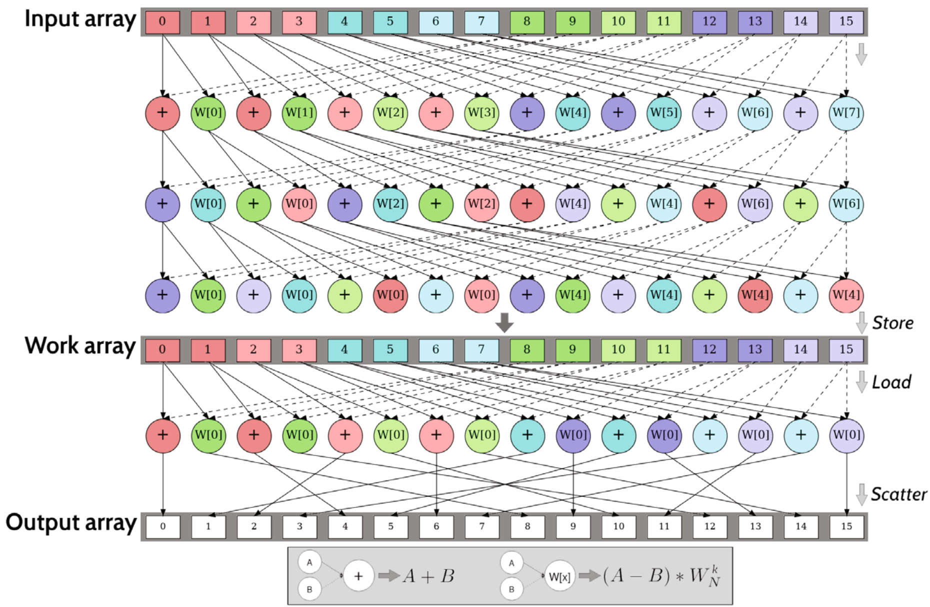


FIGURE 1 *8-Pease* vectorization for $N = 16$.

4.1.2 | Vectorization

To mitigate the slowdown introduced by the need of accessing the memory in each phase, we propose an implementation of the Pease algorithm that distributes the N elements in eight registers instead of two. Sacrificing some potential vector length and using a precise distribution, this allows us to compute three phases before having to reorder the elements in memory.

A visualization of this technique is shown on the right of Figure 1. This implementation is named *8-Pease* in the rest of the article. It uses a potential vector length of $N/8$ and only accesses memory every three phases, while still needing the data permutation at the final stage of the algorithm. The downside of having an upper limit on the vector length of $N/8$ instead of $N/2$ is suppressed for large FFT sizes where $N/8$ is larger than the maximum vector length (256).

In Figure 1, we also show that the twiddle factors are different in each phase; therefore, our first approach is to pre-compute them for each phase and to load them as the algorithm advances. The reason for not computing the twiddle factors during the execution is that they require the cosine operation, which is not present in the vector instruction set. Therefore, one needs to scalar compute them, store them in memory, and load them in vector registers.

The pseudocode of the *8-Pease* implementation is given in Algorithm 1.

Note that unlike Figure 1, the pseudocode shows the operation of the real and imaginary parts. The vector loads, the PairOperations, and the stores have been grouped for simplicity. The dark red elements from Figure 1 are loaded in *reg1*, the light reds in *reg2* and so forth. The function *3×4_PairOperation* is equivalent to executing the function in Algorithm 2 for 3 phases with 4 *PairOperations* each. All vector instructions operate on N/8 vector elements. In reality, both NEC and our RISC-V system limit the vector length to 256 double precision elements, requiring our code to compute the phases in various iterations.

4.1.3 | Optimizations

The function *PairOperation()* in Algorithm 2 takes advantage of fused operations to calculate the complex multiplication. Remember that the multiplication of two complex numbers, $(a + b \cdot i) \cdot (c + d \cdot i) = (e + f \cdot i)$ normally requires 7 operations: $e = a \cdot c - b \cdot d$, and $f = a \cdot d + b \cdot c$. We can group operations to have 2 multiplications and 2 fused operations (operations calculated with a single instruction are encapsulated using parenthesis): $t_1 = (a \cdot c)$, and $t_2 = (a \cdot d)$, so that $e = (t_1 - b \cdot d)$ and $f = (t_2 + b \cdot c)$.

We can apply some optimizations to the loading of twiddle factors. Considering a large enough vector length, each phase of the FFT requires four vector registers holding twiddle factors, one for each PairOperation described in Algorithm 2. In the proposed implementation of *8-Pease* we grouped three phases together to spare memory accesses, so we can say that each group of three phases needs to load 12 vectors of twiddle factors.

Looking at Figure 1, it can be noted that there are only $N/2$ different twiddle factors ($W_0, W_1, \ldots W_{N/2-1}$). More precisely, the number of different twiddle factors to be used in each phase is exactly half of the previous one.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 1. 8-Pease pseudocode

```
1:  procedure FFT_8Pease(Arr)
2:      for p ∈ [1 : 3 : log2(N)] do
3:          reg_{0, 1, … 7}_r ← v_ld(&real(Arr[{0, 1N/8, … 7N/8}]))
4:          reg_{0, 1, … 7}_i ← v_ld(&imag(Arr[{0, 1N/8, … 7N/8}]))
5:          3 × 4_PairOperation()
6:          if p < log2(N) then
7:              v_st_strideds(res_{0, 1, … 7}_r, &real(Arr[{0, 1, … 7}], stride = 8)])
8:              v_st_strideds(res_{0, 1, … 7}_i, &imag(Arr[{0, 1, … 7}], stride = 8)])
9:          else
10:             vindex ← v_load(&indexes[0])
11:             v_st_scatters(res_{0, 1, … 7}_r, base = real(Arr), index = vindex + {0, 1N/8, … 7N/8})
12:             v_st_scatters(res_{0, 1, … 7}_i, base = imag(Arr), index = vindex + {0, 1N/8, … 7N/8})
13:         end if
14:     end for
15: end procedure
```

Algorithm 2. PairOperation pseudocode

```
1: procedure PairOperation(reg1_r, reg1_i, reg2_r, reg2_i)
2:     res1_{r, i} ← reg1_{r, i} + reg2_{r, i}
3:     res2_{r, i} ← (reg1_{r, i} − reg2_{r, i}) ∗ W_{r, i}
4:     return res1_{r, i}, res2_{r, i}
5: end procedure
```

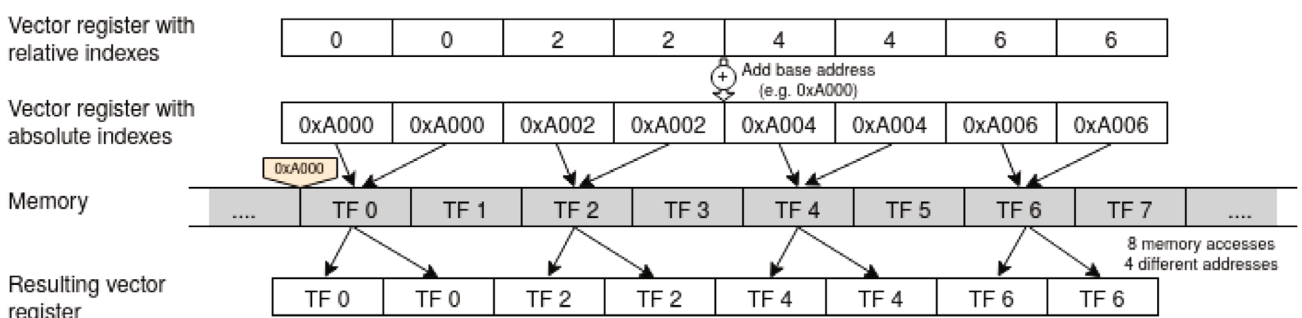


FIGURE 2 Example of the proposed accesses to twiddle factors using gather operations

The repetition of the twiddle factors across the FFT brings four important observations: (i) we are wasting memory since we were storing all of them for each phase; (ii) we are missing potential cache locality; (iii) for advanced phases, we could access a single twiddle factor per register and then replicate it; (iv) we could reuse the identical twiddle factors instead of loading them again.

Considering the first two observations, another implementation of the Pease algorithm is proposed. As the phases advance, some twiddle factors inside each vector have the same value, so we do not need to load them from replicated positions in memory. To implement this optimization, we use gather vector instructions to load the twiddle factors. In reality, gather instructions offer a more general functionality than what we require since they are meant to load sparse data, while we need strided chunks of data. However, since no ad hoc instructions exist for either RISC-V "V" nor the NEC architecture, we implemented this version using gather instructions. In NEC architecture, gather operations require a vector with absolute addresses to index the memory. We use two registers, one holding the constant relative indexes that are reused during the execution and another temporarily holding the absolute indexes after adding the offset. A graphical representation of the use of gather operation is provided in Figure 2, with its code equivalent in Algorithm 3. In the case of RISC-V "V", the gather operations can work with a base address and a vector with relative offsets to it, so the adding instruction in line 4 of Algorithm 3 can be omitted. This implementation is named *8-Pease-gt* in the rest of the article.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 3. Gather access to twiddle factors pictured on Figure 2

1: *_vr indexes* = *_vel_vseq_vl*(*VL*); ▹ 0,1,2,3,...
2: *indexes* = *_vel_vand_vsvl*(∼ (0x1), *indexes*, *VL*); ▹ 0,0,2,2,...
3: *indexes* = *_vel_vsll_vvsl*(*indexes*, 3, *VL*); ▹ 0,0,8,8,...
4: *indexes* = *_vel_vaddul_vsvl*(*addr*, *indexes*, *VL*); ▹ 0xA000,0xA000,0xA008,...
5: *_vr W* = *_vel_vgt_vvssvl*(*index*, *N*/8, *VL*);

We apply another optimization to the loading of twiddle factors, present in both *8-Pease* and *8-Pease-gt*. This one follows the observation that for each group of three phases, half of the vector registers have the same twiddle factors (e.g., in Figure 1, the dark blue register and the light red register of the second phase both hold W[0] and W[2] as their first and second element). In the third phase, the elements are identical for all registers. This means that for each group of three phases, we load 4 vector registers in the first one, 2 in the second one and 1 in the last one, only needing to load 7 twiddle factor registers instead of 12.

The loading of the twiddle factors can be optimized even further, taking advantage of trigonometric symmetries. Remember that they are computed as a complex exponent, so the real part is computed as $\mathrm{tf}(k,N)=\cos(\frac{-2\pi ik}{N})$ and the imaginary part as $\mathrm{tf}(k,N)=\sin(\frac{-2\pi ik}{N})$, with $k \in \{0, N-1\}$. Given k and k' such that $k' = k + \frac{N}{4}$, the real component of the twiddle factor $\mathrm{tf}(k', N)$ is equal to the imaginary component of $\mathrm{tf}(k, N)$, and the imaginary component is equal to the real component of the other twiddle factor but with opposite sign.

Coincidentally, the 4 vector registers with twiddle factors of each first phase are separated from one another by $\frac{N}{8}$ degrees, so the previously described symmetry property can be applied to two of them, reducing the 4 accesses to just 2. In each second phase, the two vectors are separated by $\frac{N}{4}$ degrees, so the 2 accesses can be reduced to just 1.

In summary, we started with 12 twiddle factors accesses per group of three phases, and thanks to trigonometric properties and a precise distribution of twiddle factor in the vector registers, we reduced the number of accesses to 4.

4.2 | Stockham FFT

4.2.1 | Description

The other algorithm that has been studied for vectorization is Stockham's algorithm.[4] While the algorithm is still RADIX-2 and out-of-place, it has two main differences from Pease's algorithm. The first one is that it is a self-sorting algorithm, so it does not require a permutation at the last phase. The second difference is that Stockham's algorithm does not have constant geometry like Pease's. This complicates the algorithm and its vectorization, limiting the maximum vector length depending on the phase.

4.2.2 | Vectorization

Using the same approach as with *8-Pease*, we can divide the N elements of each phase into eight vector registers to compute three phases before rearranging the elements in memory.

Due to the self-sorting nature of the algorithm, the process of storing and loading the elements changes for every three phases. With p being the phase where the loads occurs, the stores on $p+3$ consist of $\frac{vector_length}{2^p}$ groups of 2^p consecutive elements. Since an instruction that writes several consecutive elements before jumping a fixed stride does not exist in NEC's architecture, we have two options in the implementation. We can limit the vector length of the problematic phases to be equal to the size of the groups, 2^p. Since all the twiddle factors have the same value inside a group, we could use a broadcast operation to load them. The downside of this option is that for phases 3–5 and 6–8 this means limiting the vector length to 8 and 64. With SX-Aurora 's maximum vector length of 256, this limit implies not taking full advantage of the vectorization potential.

If we do not want to limit the vector length, we can store values with a scatter operation and load twiddle factors with a gather. The initial interest in using the Stockham algorithm was removing this type of memory operations, so adding them again may seem counterproductive, even though the pattern of Stockham's scatter operations contains consecutive elements while Pease's is sparser. This difference is represented in a simplified example diagram in Figure 3.

Regardless, using these long-latency instructions at the end of these particular phases outperforms having up to 32 times more instructions during three phases when limiting the vector length to 8, so the final implementation uses scatters.

A simplified pseudocode of this alternative is shown in Algorithm 4. We call this algorithm *8-Stockham*. Concerning the optimizations of this implementation, we applied the same improvements to complex operations and twiddle factor loading as with the Pease implementation.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 3 Simplified example of the scatter operations used in Pease and Stockham's algorithms, with a vector length of 8 elements.

Algorithm 4. 8-Stockham pseudocode

```
1:  procedure FFT_8Stockham(Arr)
2:      reg_{0, 1, … 7}_r ← v_ld(&real(Arr[{0, 1N/8, … 7N/8}]))
3:      reg_{0, 1, … 7}_i ← v_ld(&imag(Arr[{0, 1N/8, … 7N/8}]))
4:      3 × 4_PairOperation()
5:      v_st_strideds(res_{0, 1, … 7}_r, &real(Arr[{0, 1, … 7}]), stride = 8)
6:      v_st_strideds(res_{0, 1, … 7}_i, &imag(Arr[{0, 1, … 7}]), stride = 8)
7:      gsize = 8
8:      for p ∈ [4 : 3 : log2(N)] do
9:          reg_{0, 1, … 7}_r ← v_ld(&real(Arr[{0, 1N/8, … 7N/8}]))
10:         reg_{0, 1, … 7}_i ← v_ld(&imag(Arr[{0, 1N/8, … 7N/8}]))
11:         if gsize < VL then
12:             gatherW()
13:             3 × 4_PairOperation()
14:             vindex ← v_load(&indexes[0])
15:             v_st_scatter(res_{0, 1, … 7}_r, base = real(Arr), index = vindex + {0, 1gsize, … 7gsize})
16:             v_st_scatter(res_{0, 1, … 7}_i, base = imag(Arr), index = vindex + {0, 1gsize, … 7gsize})
17:         else
18:             broadcastW()
19:             3 × 4_PairOperation()
20:             v_st(res_{0, 1, … 7}_r, &real(Arr[{0, 1N/8, … 7N/8}]))
21:             v_st(res_{0, 1, … 7}_i, &imag(Arr[{0, 1N/8, … 7N/8}]))
22:         end if
23:         gsize = gsize * 8
24:     end for
25: end procedure
```

5 | EVALUATION

In this section, we study the performance of our implementations in the vector accelerator from NEC, the SX-Aurora , and in an experimental setup that uses a RISC-V core named Avispado alongside a VPU implementing the RISCV. We measure the real-time used to compute the FFT, including the communication to the accelerator or the vector unit and other system interferences. The pre-computation is disregarded because it can be used for multiple FFT of the same size.

5.1 | Evaluation of NEC SX-Aurora

NEC has optimized math libraries called NEC library collection (NLC)[††]. Our usage of NLC is limited to *aslfftw*, a vectorized FFT whose interface is compatible with fftw. First, we have compared the performance of the proposed implementations in Section 4 with *aslfftw*, computing it as a speedup to a scalar (i.e., without vector instructions) fftw, compiled with NEC's compiler *ncc*.

We see in Figure 4 how our implementations outperform *aslfftw* until an FFT size of 65,536 elements. From that point on, *aslfftw* doubles the performance of our Pease's implementations, while *8-Stockham* only underperforms *aslffw* by a small margin. It is also notable that while *8-Pease-gt* was designed to improve *8-Pease*, it obtained a lower performance.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

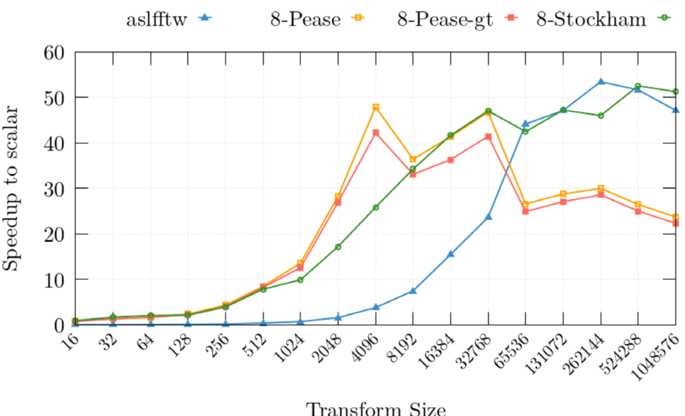


FIGURE 4 Speedup in NEC of the proposed vectorized FFT implementations and *aslfftw*.

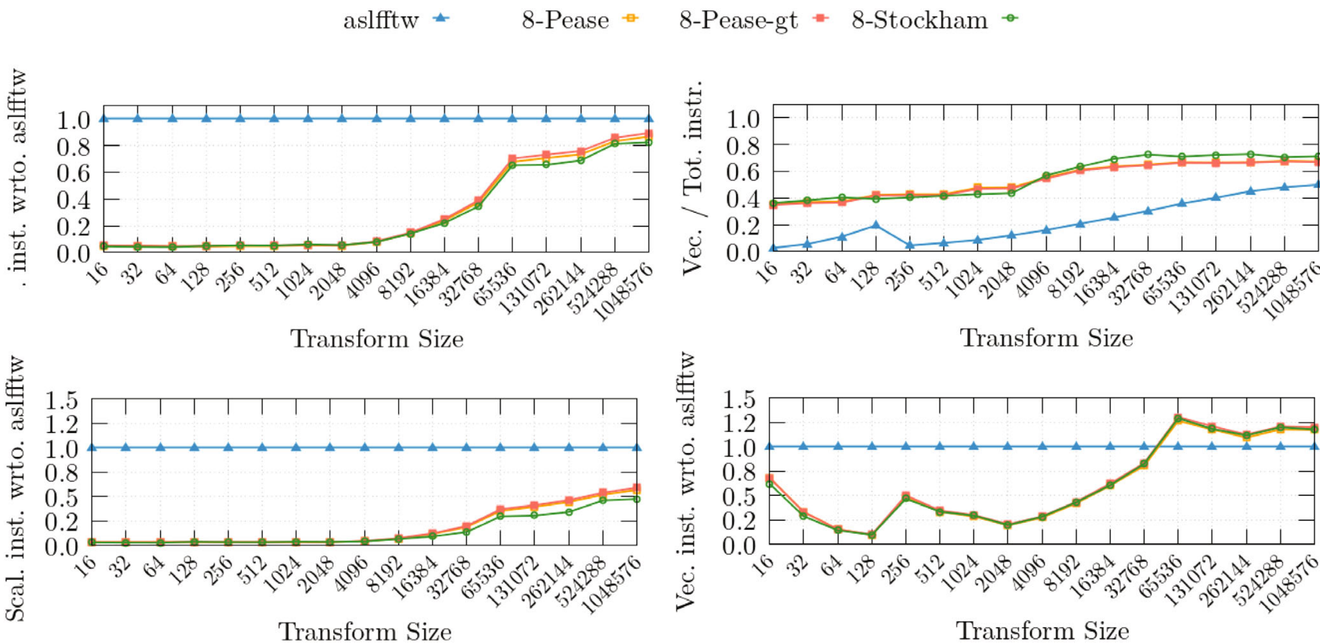


FIGURE 5 Total instructions (top left), vectorization percentage (top right), scalar instructions (bottom left) and vector instructions (bottom right) of the evaluated implementations.

In Figure 5, we show information about vector and scalar instructions of the implementations. The first clear observation is that NEC's implementation executes many more instructions than our implementations for all sizes, especially for small to medium sizes. This difference is even greater when looking just at scalar instructions.

In terms of vector instructions, our implementations execute approximately 20% more instructions than *aslfftw* for sizes bigger than 65,536. This disparity suggests a core difference in the FFT algorithm between our implementation and NEC's. The number of floating-point operations is related to the used RADIX in the FFT computation. These results suggest that *aslfftw* is using a different (bigger) RADIX for larger transforms since that would reduce the number of needed operations.

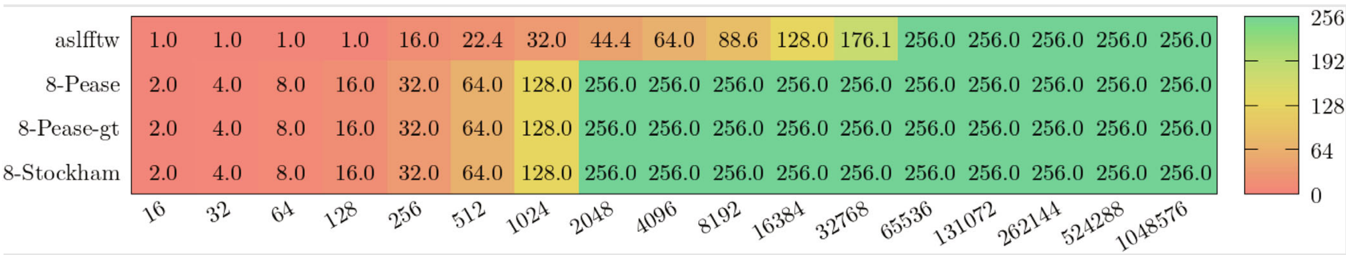
Second, our implementations accomplish a higher vectorization ratio than *aslfftw*, lingering at 40% for small to medium sizes and rising to 70% in large sizes. *Aslfftw* vector instructions represent less than 20% of the total instructions for small and medium sizes and do not reach 50% in the best case.

To better understand the difference in instructions, we have to consider the number of elements being operated with each vector instruction. In Table 1 we display the average vector length used by the implementations, with greener colors indicating a higher vector length.

We show that our proposed implementations are able to use the maximum vector length, 256 64-bit elements, with smaller problem sizes than *aslfftw*. This implies a better usage of the vector unit and a reduction in the number of instructions since each one is doing more operations.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 Average vector length in elements for different FFT sizes and implementations.



	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
aslfftw	1.0	1.0	1.0	1.0	16.0	22.4	32.0	44.4	64.0	88.6	128.0	176.1	256.0	256.0	256.0	256.0	256.0
8-Pease	2.0	4.0	8.0	16.0	32.0	64.0	128.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0
8-Pease-gt	2.0	4.0	8.0	16.0	32.0	64.0	128.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0
8-Stockham	2.0	4.0	8.0	16.0	32.0	64.0	128.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0	256.0

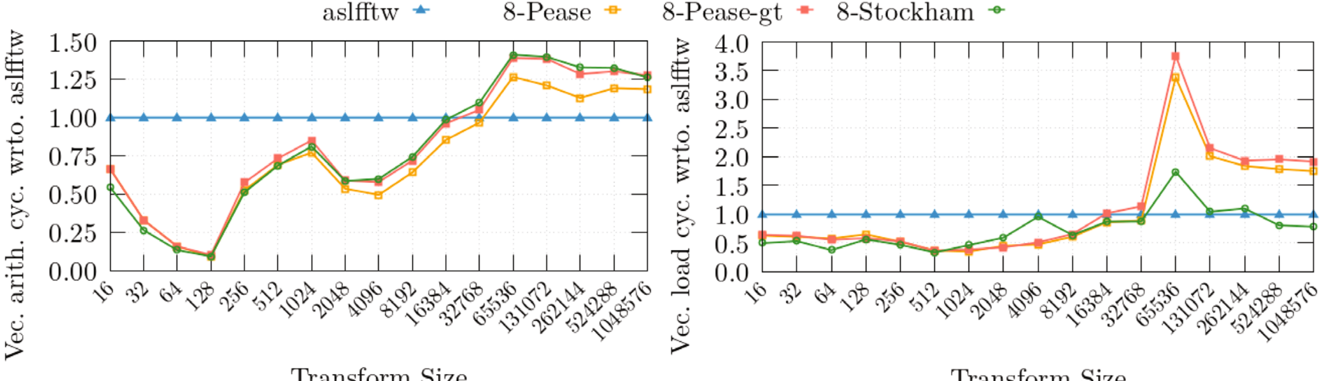


FIGURE 6 Vec arith (left) and load (right) cycles of our implementations wrto. *aslfftw*.

A pair of relevant counters to understand the performance of the implementations is *vec_arith_cyc* and *vec_load_cyc*, which count the cycles spent in arithmetic vector instructions and load vector instructions respectively.

In Figure 6, we see the number of cycles used by arithmetic and load vector instructions with respect to *aslfftw*. Our implementations execute approximately 25% more cycles doing vector arithmetic operations than *aslfftw* for larger FFT sizes. The finer grain of using a small vector length of *aslfftw* can allow it to be more precise with arithmetic optimizations, but the significant disparity in large sizes strengthens the theory of *aslfftw* using a higher FFT RADIX for large transforms.

A much larger difference is present in load vector cycles. We find a notable spike in the cycles spent by our Pease's implementation in size 65,536, needing four times more cycles to load vector elements than *aslfftw*. On the following sizes, the Pease's implementations are doubling the cycles spent in this task. A smaller but still significant spike is also found at that size for the *8-Stockham* implementation, but in the following sizes it gets to *aslfftw* level again.

We would also like to study the vector cycles spent in-store operations, but these cycles are not mapped in any hardware counter present in the architecture.

To study if the increment in vector load cycles of our Pease's implementations is due to loading more elements or due to slower loads, in Figure 7 we show how many vector elements are being loaded per each cycle spent in vector load instructions, as a metric of "efficiency" of the vector loads. We also show the implementations' vector load cache hit ratio, since it can be related to slower loads.

The left plot of Figure 7 shows that the vector load cycles difference between *8-Stockham* and *aslfftw* on size 65,536 is explained by our implementation having a lower vector load efficiency than *aslfftw* at that specific point.

Moreover, there is a big difference between our Stockham and Pease's implementations when looking at the load efficiency on the left of Figure 7. Our Pease implementations start with a load efficiency greater than Stockham but it drops at FFT sizes of 8 k-elements and 64 k-elements.

8-Pease and *8-Stockham* present a nearly identical cache hit ratio as seen on the left of Figure 7, and they load the same number of elements from memory. Considering that the vector load efficiency is not strongly lowered for *8-Stockham* at the size 65,536, we can suggest that the difference in the efficiencies between implementations is caused by an unfavorable memory access pattern inherent to Pease's algorithm, presumably related to the scatter operations executed at the last phase of the implementation, since the address distance between each element in these scatters grows as the FFT size increases.

We also note in the left plot of Figure 7 that the usage of the gather instruction in *8-Pease-gt* does not accomplish its intended results since it lowers the efficiency of vector load instructions with respect to *8-Pease* instead of improving it. The theoretically better memory access of *8-Pease-gt* is reflected in the cache hit-ratio, when comparing it to *8-Pease*.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

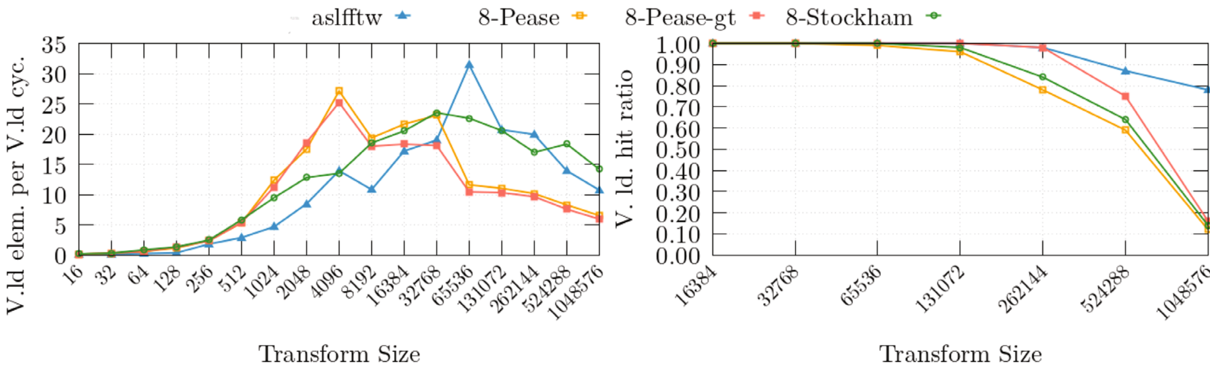


FIGURE 7 Vec. load elements per vec. load cycle (left) and hit ratio (right)

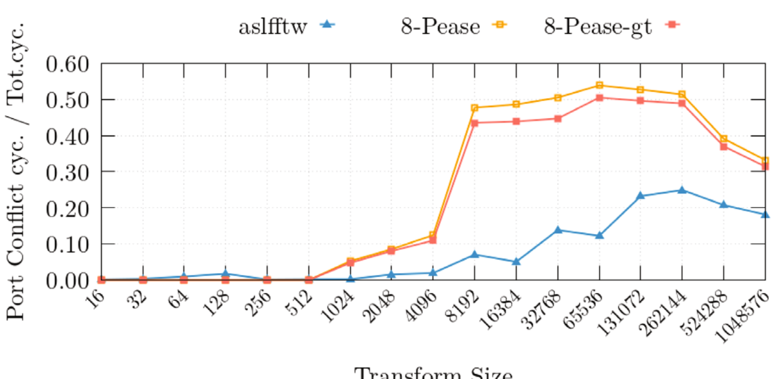


FIGURE 8 Port conflict cycles divided by total cycles of the Pease and aslfftw implementations

We also have access to another counter named Port Conflict cycles. While its documentation is scarce, it is related to conflicting accesses to memory devices and gives more insight into the Pease algorithm's vector load efficiency drop. In Figure 8, we show the proportion of cycles from the total execution that are spent in these port conflicts. There are no lines for the Stockham implementations, since not a single cycle is counted as a Port Conflict in their execution. At the first point where the performance of the Pease algorithm dropped, 8K elements, the Port Conflict cycle proportion rises to 47% of the total cycles. Then, at 64K it rises again to 55%.

5.2 | Evaluation of RISC-V "V"

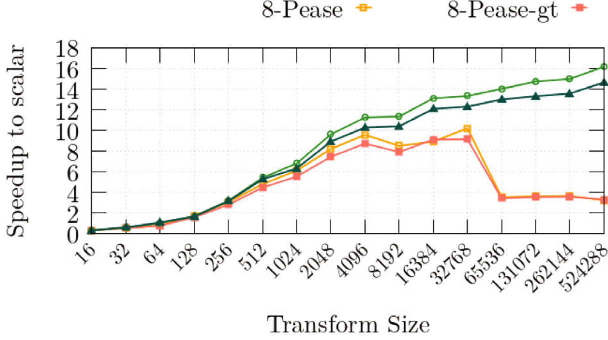
In the case of RISC-V, we do not have an optimized FFT library to be taken as a reference as we had with NEC and the *aslfftw*. The comparison point that we use is a scalar FFTW compiled with GCC for RISC-V. In the left plot of Figure 9 we show that the *8-Stockham* implementation reaches a maximum speedup of 16× compared to the scalar implementation. There are two behaviors that have not changed with respect to the evaluation in the NEC architecture: (i) The gather instructions, even when using relative offsets in RISC-V "V", do not improve the performance of our implementations (ii) The Pease implementation suffers a pronounced drop in performance with FFT size of 64 k-elements, while the Stockham implementation does not.

Due to the experimental nature of our setup, we do not have access to as many hardware counters as we did in the NEC SX-Aurora . One very relevant counter that we can read and that gives a key metric for vector computation is the cycles where the VPU is performing useful work, including both arithmetic and memory operations. We can divide this value by the total cycles of the execution and obtain what proportion of the execution cycles is vectorized.

In the right plot of Figure 9 we see how even for the smallest FFT sizes, the VPU is working during roughly 80% of the cycles. As the FFT size increases, this percentage goes up to virtually 100%. This does not mean that there is no scalar work being done, but that all of it can be overlapped with the vector work.

On larger sizes, the percentage of cycles where the VPU is active drops to 90%. While this is still a high value, the drop could be explained by an ongoing issue with context switches affecting our experimental RISC-V platform (out of the scope of this article). The operating system context switch happens on a fixed time interval. Since the FPGA is running at a significantly lower frequency than a real system, this context switch happens

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



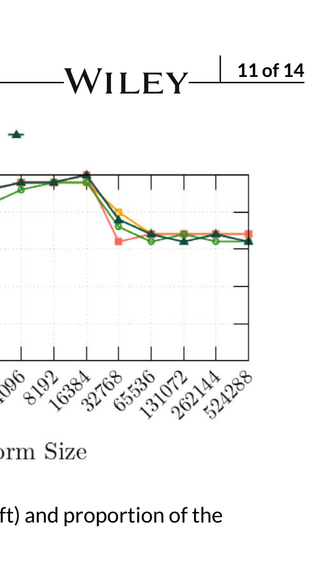


FIGURE 9 Speedup in the FPGA of the proposed vectorized FFT implementations against a scalar FFTW (left) and proportion of the execution cycles where the VPU is active (right).

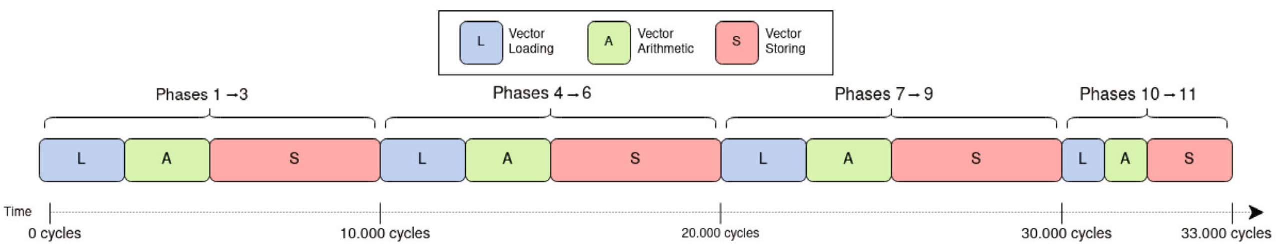


FIGURE 10 Execution diagram of *8-Stockham* with 2048 elements.

every few cycles. As the FFT size grows, so does the execution time and the number of context switches included in the measurements. These context switches are not using the VPU, so the proportion of vector cycles lowers.

Although we do not have access to many hardware counters, we can use an integrated logic analyzer (ILA) to snoop the signals inside the FPGA. This allows us to get an execution trace where we see what the VPU is doing at any moment. We analyzed the execution of the *8-Stockham* algorithm for sizes where the vector length is capped at 256 elements (so we are utilizing the VPU at its full potential).

In Figure 10, we can see a diagram of the types of instruction completed during the execution. We can see how the execution is heavily differentiated between groups of load instructions, groups of arithmetic, and groups of stores, which agrees with the algorithm presented in Algorithm 4.

While from a software perspective this distribution of the instructions is correct (i.e., load all data, operate it, store it), from a hardware point of view, this can be improved since the VPU allows to execute some instructions simultaneously. More precisely:

(i) arithmetic operations can be fully overlapped with memory operations (given that they do not have data dependencies); (ii) two loads can be executed at the same time, while only one simultaneous store is supported; (iii) loads and stores cannot be executed together.

Our goal is to leverage this property to hide the latency of some operations. The hardware is capable of some reordering, since it is an out-of-order machine, but the VPU has a small window to do so. If we want to take full advantage of the hardware, we need to reorder the instructions before the execution.

When studying the instructions of a program, there is a concept called the basic block, which is a sequence of instructions between two branch operations. They have the property that if one instruction is executed, the rest is executed too, so within a basic block, the instructions can be reordered without issues (given that the data dependencies are respected).

In our implementations, these basic blocks are constituted by approximately 130 consecutive vector instructions, which give us a large degree of freedom to reorder them. Following the three above-stated characteristics of the VPU, we manually reordered our code and analyzed it again with the ILA. This reordered implementation is called *8-Stockham-R* from now on. In Figure 11, we portrait how virtually all the arithmetic operations have been overlapped with the memory instructions; now only the loading and storing of data is affecting the execution time.

Figure 12 confirms that the performance of the reordered code, with and without gather instructions, improved by 20%–30% with respect to our initial versions (with the variant without gathers still on top).

While this improvement is great, it is the fruit of a great programming effort requiring low-level knowledge of the platform's architecture details. In order to ease this procedure for future works, we adapted the LLVM machine scheduler, which is in charge of reordering the instructions in compile-time to optimize the codes.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 11 Execution diagram of *8-Stockham-R* with 2048 elements.

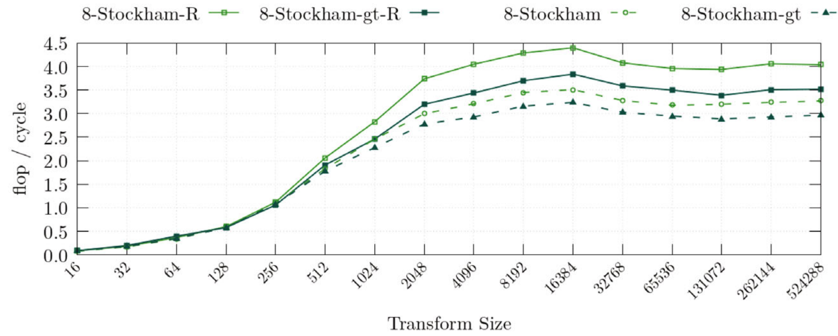


FIGURE 12 Flops per cycle after reordering the instructions of the Stockham implementation.

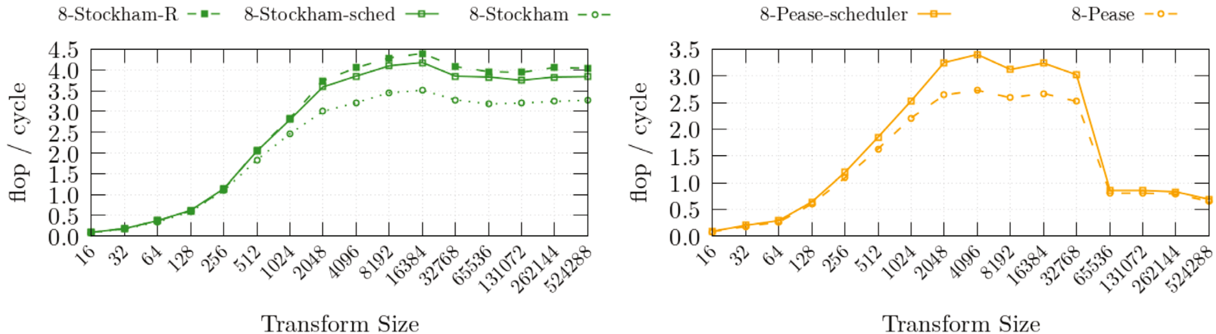


FIGURE 13 Flops per cycle after changing the compiler scheduler.

We differentiate the different types of vector instruction (e.g., Memory/ALU, strided/no-strided, etc.) and assign a resource and a latency to each of them. This allows the LLVM machine scheduler to create a rough simulation of the code it compiles and reorders the instructions to minimize the execution time.

In Figure 13, we show how the automatic reordering performs closely to the previous manual reordering version. We also applied the automatic reordering to the *8-Pease* algorithm to showcase its improvement without the need of any work from the programmer.

One important remark is that in our case the scheduler is not as good as the manual reorder, since the compiler has more constraints when it reorders instructions. For example, the compiler does not raise a store before a previous load, since it cannot currently know at compile time if their memory region overlaps and then the result of the load changes depending on their order.

5.3 | Comparison of RISC-V "V" and the NEC SX-Aurora

Finally, we compared the results of both architectures. For this final study, we ported to NEC the manually reordered Stockham code described in Section 5.2, which leads to a 10% performance improvement; thus, this type of scheduling benefits both long-vector architectures. In terms of

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

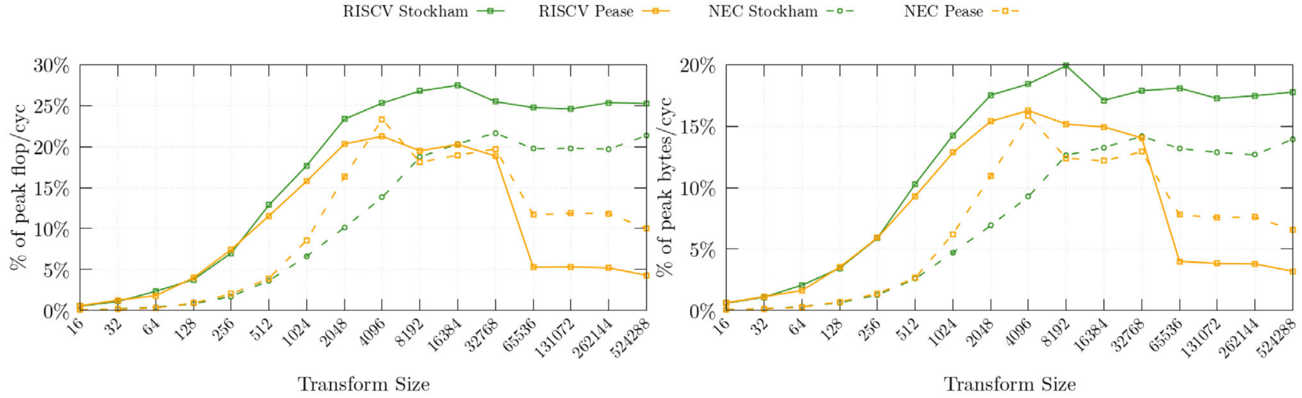


FIGURE 14 Percentage of the peak flops/c (left) and bytes/c (right) of both architectures for the different implementations

speedup to scalar or raw flops per cycle, the NEC accelerator outperforms the RISC-V system by approximately a factor of 12×. This is no surprise, since the vector engine of NEC has 12 times more functional units than the RISC-V VPU.

A good point of comparison for our implementations between two architectures with such a different scale is how efficiently they use their resources. We consider the peak performance as the maximum flops/cycle that the architectures can provide (i.e., executing FMAs without dependencies) to be 16 for the RISC-V architecture and 192 for the NEC SX-Aurora , in Figure 14. For the memory subsystem, using only unit-strided vector memory accesses reaches a bandwidth of 870 Bytes/cycle on NEC and 64 Bytes/cycle on RISC-V. Considering these limits, we can compute which percentage (i.e. efficiency) we are reaching.

We selected the best version of our Stockham and Pease implementations on each architecture. Considering flops/c, the Stockham code achieves up to 25% of the peak performance on RISC-V platform and up to 20% on NEC's. In terms of memory bandwidth, we reach 20% of the peak in RISC-V and 15% on NEC. The Pease algorithm also performs better on RISC-V until bigger sizes, where it is more than twice as fast on NEC.

6 | CONCLUSIONS

Our implementations of the FFT for the NEC SX-Aurora show an efficient usage of the vector engine, overtaking the highly optimized proprietary vendor implementation found in NEC library collection for FFT sizes up to 65,536 elements. We achieve 20× speedup for sizes under 1024 compared to NEC's FFT, down to a 2× speedup with 32,768 elements. In terms of speedup to scalar, our implementation reaches a 50× for large transforms. We also evaluated our implementations on an experimental RISC-V "V" system, reaching 16× speedup with respect to the scalar execution. We discussed the performance of vector memory gather operations in our implementations, finding that the memory access optimization that they give us does not pay off because of their long latency.

We compared two algorithms for the FFT computation, Pease's and Stockham's. We found that for vectorized codes, the complex permutations needed by Pease's impact negatively the performance, notably with large FFT sizes. We argue in favor of more specific register shuffling and memory accessing vector instructions. We evinced the importance of avoiding memory instructions that FFT computation often requires, even if this implies more vector registers or reducing the vector length.

We also highlight two weaknesses of our proposed implementations for larger FFT sizes when comparing them with NEC's implementation: (i) both Pease and Stockham implementations spend ∼25% more cycles executing floating-point operations, suggesting the need to explore different RADIX FFT algorithms. (ii) Pease's implementation has a lower vector load efficiency.

In the RISC-V study we emphasize the importance of a hardware-aware order of the instructions. While we show the potential of doing this work manually, we present an automatic mechanism to do it with the compiler; which proved to be nearly as efficient as the manual instruction reordering.

We leave for future work the parallelization of our implementations and the exploration of different RADIX and higher dimension FFT algorithms.

CONFLICT OF INTEREST

All authors declare that they have no conflicts of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ENDNOTES

*https://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf
†https://sx-aurora-dev.github.io/velintrin.html
‡https://www.xilinx.com/products/boards-and-kits/vcu128.html
§Complete device name: XCVU37P-2FSVH2892E
¶https://semidynamics.com/
#https://riscvsummit2021.sched.com/event/nfGE
‖https://www.ics.forth.gr/
**https://repo.hca.bsc.es/gitlab/rferrer/llvm-epi
††https://www.hpc.nec/documents/sdk/SDK_NLC/UsersGuide/main/en/

ORCID

Pablo Vizcaino https://orcid.org/0000-0002-9253-8275
Filippo Mantovani https://orcid.org/0000-0003-3559-4825
Roger Ferrer https://orcid.org/0000-0003-3306-8610
Jesus Labarta https://orcid.org/0000-0002-7489-4727

REFERENCES

1. Vizcaino P, Mantovani F, Labarta J. Accelerating FFT using NEC SX-aurora vector engine. Proceedings of the Euro-Par 2021: Parallel Processing Workshops; 2021; Nature Publishing Group, in press.
2. Chu E, George A. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC press; 1999.
3. Pease MC. An adaptation of the fast fourier transform for parallel processing. *J ACM (JACM)*. 1968;15(2):252-264.
4. Swarztrauber PN. FFT algorithms for vector computers. *Parallel Comput*. 1984;1(1):45-63.
5. Chow AC, Fossum GC, Brokenshire DA. A programming example: large FFT on the cell broadband engine. *Global Signal Process Expo (GSPx)*. 2005.
6. Anderson M. Accelerating the 3-D FFT using a heterogeneous FPGA architecture. Proceedings of the European Conference on Parallel Processing; 2017:653-663; Springer.
7. Wang Q. Optimizing FFT-based convolution on ARMv8 multi-core CPUs. Proceedings of the European Conference on Parallel Processing; 2020:248-262; Springer.
8. Malkovsky S. Evaluating the performance of FFT library implementations on modern hybrid computing systems. *J Supercomput*. 2021;1-29.
9. Bailey D. A high-performance FFT algorithm for vector supercomputers. *Int J J Supercomput Appl*. 1987;2(1):82-87.
10. Franchetti F, Puschel M. SIMD vectorization of non-two-power sized FFTs. Proceedings of the 2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07; Vol. 2, 2007:II–17–II–20.
11. Gómez Crespo C. Optimizing sparse matrix-vector multiplication in NEC SX-Aurora vector engine. Proceedings of the 26th Symposium on Principles and Practice of Parallel Programming; 2021.
12. Jansson N. Spectral element simulations on the NEC SX-aurora TSUBASA. Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region; 2021:32-39.
13. Vizcaino SP. Evaluación y optimización de algoritmos fast Fourier transform en SX-Aurora NEC; 2020.
14. Yamada Y, Momose S. Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA. Proceedings of A Symposium on High Performance Chips, Hot Chips; Vol. 30, 2018:19-21.
15. Komatsu K. Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC18; 2018:685-696.

How to cite this article: Vizcaino P, Mantovani F, Ferrer R, Labarta J. Acceleration with long vector architectures: Implementation and evaluation of the FFT kernel on NEC SX-Aurora and RISC-V vector extension. *Concurrency Computat Pract Exper*. 2023;35(20):e7424. doi: 10.1002/cpe.7424

15320634, 2023, 20, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7424 by Readcube (Labtiva Inc.), Wiley Online Library on [15/09/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License