electronics

MDPI

Article

A Collaborative CPU Vector Offloader: Putting Idle Vector Resources to Work on Commodity Processors

Youngbin Son [1], Seokwon Kang [2], Hongjun Um [2], Seokho Lee [1], Jonghyun Ham [2], Donghyeon Kim [2] and Yongjun Park [1,2,*]

[1] Department of Artificial Intelligence, Hanyang University, Seoul 04763, Korea; youngbinson@hanyang.ac.kr (Y.S.); seokholee@hanyang.ac.kr (S.L.)
[2] Department of Computer Science, Hanyang University, Seoul 04763, Korea; kswon0202@hanyang.ac.kr (S.K.); hongjunum@hanyang.ac.kr (H.U.); hamjh9510@hanyang.ac.kr (J.H.); dhkim9309@hanyang.ac.kr (D.K.)
* Correspondence: yongjunpark@hanyang.ac.kr

Abstract: Most modern processors contain a vector accelerator or internal vector units for the fast computation of large target workloads. However, accelerating applications using vector units is difficult because the underlying data parallelism should be uncovered explicitly using vector-specific instructions. Therefore, vector units are often underutilized or remain idle because of the challenges faced in vector code generation. To solve this underutilization problem of existing vector units, we propose the Vector Offloader for executing scalar programs, which considers the vector unit as a scalar operation unit. By using vector masking, an appropriate partition of the vector unit can be utilized to support scalar instructions. To efficiently utilize all execution units, including the vector unit, the Vector Offloader suggests running the target applications concurrently in both the central processing unit (CPU) and the decoupled vector units, by offloading some parts of the program to the vector unit. Furthermore, a profile-guided optimization technique is employed to determine the optimal offloading ratio for balancing the load between the CPU and the vector unit. We implemented the Vector Offloader on a RISC-V infrastructure with a Hwacha vector unit, and evaluated its performance using a Polybench benchmark set. Experimental results showed that the proposed technique achieved performance improvements up to 1.31× better than the simple, CPU-only execution on a field programmable gate array (FPGA)-level evaluation.

Keywords: vector processors; job offloading; resource utilization; data parallelism; heterogeneous system architectures

Citation: Son, Y.; Kang, S.; Um, H.; Lee, S.; Ham, J.; Kim, D.; Park, Y. A Collaborative CPU Vector Offloader: Putting Idle Vector Resources to Work on Commodity Processors. *Electronics* **2021**, *10* , 2960. https://doi.org/10.3390/electronics10232960

Academic Editor: Ping-Feng Pai

Received: 4 November 2021
Accepted: 26 November 2021
Published: 28 November 2021

1. Introduction

As many emerging workloads—such as physics simulations and vision applications—have become more complex, vast, and diverse, they can no longer be handled efficiently by a simple, single central processing unit (CPU). Although a multitasking technique using multiple CPU cores has been proposed to achieve a high performance gain, it suffers from high parallelization overhead when creating multiple threads. To support single-thread performance efficiently, several accelerators such as vector processors, graphics processing units (GPUs), and neural processing units (NPUs) are often utilized. In particular, vector processors have become an essential part of modern computing because they can handle large workloads effectively through tight integration with existing CPUs.

Vector processing is a parallel computing technique that exploits data parallelism using a single-instruction multiple-data (SIMD) scheme, and it requires SIMD-purpose hardware and extensive software support to utilize the hardware. Most modern CPU cores for both general-purpose use and supercomputing generally contain vector processing units such as streaming SIMD extensions (SSE) [1], advanced vector extensions (AVX) (for Intel CPUs) [2], and Neon (for ARM CPUs) [3]. To accelerate applications using

vector processing units, programmers or compilers must generate special codes using these available vector instruction set architecture (ISA) extensions.

Although vector processing appears to be promising, the main challenge is programming. To accelerate applications efficiently using vector units, a compiler or programmer should find a substantial amount of underlying data parallelism and translate the parallelization potential into a real code to make sufficient use of the vector unit. Although many techniques for improving the quality of the vector code have been proposed [4–7], the resulting vector resource utilization is still low. Manual vector code optimization is a basic approach; however, it requires a deep understanding of the target vector architectures, and the optimized codes have limited reusability. Automatic compiler-level vectorization is a promising alternative to manual vector code generation, but it cannot provide sufficient coverage because it can vectorize only 45–71% of loops, even in synthetic benchmarks [8]. Moreover, many vectorized applications do not show sufficient performance gains, as expected, owing to the high data alignment overhead [4,7]. Although many vectorization libraries also utilize vector units by providing more general interfaces, they are still limited in use.

To solve this problem, we propose a method of utilizing vector units as simple scalar processing units, similar to general arithmetic logic units (ALUs) inside CPU cores. With this scheme, normal instruction-level parallelism can be exploited more efficiently using both internal ALUs and vector units, whereas not all of the computational capabilities of the vector units are utilized. By utilizing only a subset of the computational resources to run scalar instructions on the vector units, two potential problems are identified. The first problem is resource waste, owing to the unnecessary computations of the remaining parallel resources. This can be prevented using several vector masking capabilities, such as the configurable vector length [9,10] or masked vector instruction features [2,11], which are widely supported by modern vector ISAs to prevent such side effects.

The second problem is the low performance of vector units when parallel resources are not fully utilized. To compensate for the low performance gain, the collaborative execution of the target application on both internal ALUs and vector units is a possibility. To maximize the collaboration performance, the optimal offloading ratio of the target workloads between the internal ALUs and the vector units should be determined based on the difference in the raw performance as well as the job offloading overhead from the CPUs to the vector units. Because a precise prediction of the ratio is almost impossible to achieve statically, a profile-guided optimization algorithm is a promising solution.

Based on these insights, a collaborative execution technique, referred to as Vector Offloader, was designed in this study to make the collaborative execution of scalar programs possible on internal ALUs and vector units without complex vectorization techniques. The technique splits the target loops into two scalar partitions to execute on internal ALUs and vector units by finding the optimal vector offloading ratios based on the profile information. The proposed technique was implemented on a RISC-V infrastructure (Rocket core) with a Hwacha vector processor extension [12], and it was verified on a VCU118 FPGA board [13]. Although the proposed technique has been implemented specifically on the vector accelerator, it is also applicable to other architectures that contain vector ALUs if dynamic out-of-order instruction scheduling is supported to make possible the parallel execution of main integer ALUs and vector units. The Vector Offloader achieved a 1.31× performance improvement on average for the Polybench benchmark suite [14], with double, float, and integer data types for mini and small workload sizes, and it showed similar performance gains in all the test cases.

This paper provides the following contributions:

- A method that utilizes vector units without considering data parallelism is introduced;
- A profile-guided algorithm is introduced to determine the optimal offloading ratio based on an analysis of the importance of the work offloading size to the vector units;
- The Vector Offloader is evaluated in depth in a field programmable gate array (FPGA)-based environment.

2. Background

2.1. Modern Vector Architecture

SIMD exploits data-level parallelism (DLP) to efficiently run data-parallel code regions in terms of energy and area efficiency [15]. However, because of their low programmability and inflexible memory access patterns, SIMD units are often underutilized or not used at all. Many techniques have been proposed to solve this problem. To avoid memory access restrictions and control divergence problems, Intel SSE, AVX, and ARM NEON support lane-masking instructions such that multiple-way data paths can be turned on and off selectively. As an extended version of the traditional SIMD structure, NVIDIA proposed the SIMT [16] structure to support branch divergence more efficiently. Additionally, Maven [17], which is an earlier version of Hwacha [12], was proposed as a hybrid architecture between traditional vector structures and GPUs.

As briefly discussed, the operation units in modern vector units can be controlled individually. For example, Intel AVX, a subword-level SIMD structure, can select units to execute through masking, and Maven, a vector-thread (VT) architecture, can control each unit individually by not only masking, but also determining the hardware vector length at runtime. VT-based design variations, such as Maven, have a control thread that manages the micro-threads on the vector execution unit [18]. The control thread uses a vector load and store instructions to efficiently move data, and then uses vector fetch instructions to control all micro-threads concurrently.

2.2. Hwacha: Baseline Vector Architecture

Hwacha is a vector accelerator attached to a RISC-V Rocket [19] CPU with a Rocket custom coprocessor (RoCC) interface [12]. The main difference between Hwacha and the traditional vector assembly programming model is that vector operations on Hwacha are not fetched from a host CPU; instead, they are fetched by the Hwacha accelerator in the form of a separate worker thread. Therefore, the host CPU needs to control the Hwacha using control threads without executing the actual instructions [12]. RoCC is an interface designed to accelerate a Rocket core using a coprocessor, and the interface allows communication between the core and the attached accelerator. By exploiting the CPU–coprocessor decoupled structure with the RoCC interface, and then separating the vector instructions from the control thread, as discussed, Hwacha can execute the separate worker thread in parallel to the main thread of the CPU.

As shown in Figure 1a, the Rocket five-stage in-order CPUs and Hwacha are connected through the RoCC interface, and their memory data can be shared using the L2 cache. As shown in Figure 1b, the CPU executes simple Hwacha control threads, and Hwacha executes the corresponding worker threads. The Hwacha-specific instructions on the CPU side (control thread) consist of only the vector memory address (vmca), vector length configuration (vsetvl), and vector fetch (vf) instructions. Other Hwacha instructions are executed to manipulate data on Hwacha natively.

Regarding the control thread, the Hwacha control instructions on the CPU are executed in a manner similar to normal instructions. When the Hwacha instructions are fetched from the CPU, the decoded information is passed to Hwacha through a five-stage pipeline. In the instruction decoding stage, the CPU generates the required information by understanding the instructions, and then sends the information to Hwacha using the RoCC interface. For example, vmca and vf instructions pass on the specific memory addresses to Hwacha. To utilize the Hwacha execution unit, a vf instruction specifically passes the program counter (PC) of a target micro-thread. Based on the PC value, the vector memory unit (VMU) in Hwacha fetches instructions from the micro-thread. Therefore, the CPU has the simple role of passing the PCs of multiple micro-threads to be executed on Hwacha.

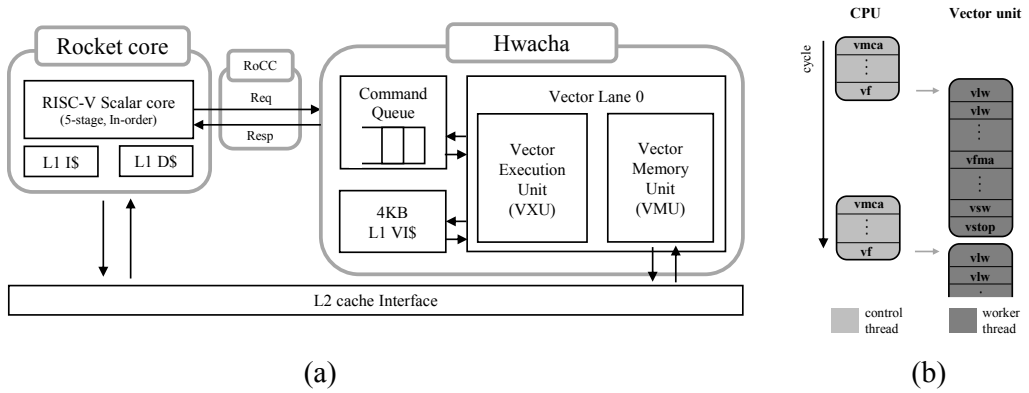

Figure 1. A baseline system architecture and an example execution scenario: (**a**) a RISC-V Rocket core and a Hwacha vector unit, and (**b**) an execution example of a control thread in the CPU and a separate work thread in the parallel vector unit.

3. Motivation

3.1. Vectorization Challenges

Although most modern CPU designs include vector units, they are often underutilized or not efficiently used in many applications. The low utilization is primarily caused by the difficulty in generating efficient vector instruction streams; this is because a compiler or programmer must understand the target application characteristics. The first solution is to have experienced programmers understand the algorithm fully and then translate the full underlying data parallelism into a vectorized code. Clearly, this is difficult for average programmers and requires high development efforts from programmers whenever new applications are introduced. The second solution is to use an automatic compiler-level vectorization technique, which is a typical approach employed by various widely used compilers. However, this technique results in low coverage because only some predefined patterns are accepted, and the performance gain is insufficient even when a vectorized code can be generated.

Figure 2 shows the auto-vectorization coverage of widely used compilers: GCC [20] and Clang [21]. We vectorized 17 loops on 9 benchmarks from the Polybench [14] suite by compiling them using the -O3 optimization level and -mavx2 microarchitecture options. As shown in Figure 2, Clang and GCC can successfully vectorize only 29% and 59% of the loops, respectively. More specifically, Clang and GCC can apply innermost-loop-level vectorization on 29% of the target loops (same loops on both compilers), whereas GCC alone can apply vectorization onto 30% more loops, using advanced techniques to consider a scope wider than the innermost loop. Based on reports from the compilers, loop vectorization was not applied owing to the complicated access patterns, safety concerns regarding instruction reordering, and low expected performance gains.

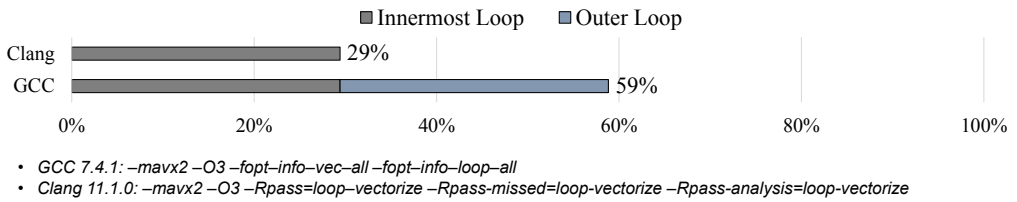

Figure 2. Auto-vectorization coverage of existing compilers. We tested GCC 7.4.1 and Clang 11.1.0 with -O3 optimization level and -mavx2 microarchitecture options. A total of 17 loops on 9 benchmarks from Polybench [14] were tested. The remaining options were used to report the vectorization optimization statistics.

3.2. Alternative: Parallel Execution of Scalar Workloads on CPUs and Vector Units

As discussed, it is difficult to fully utilize vector units using conventional approaches. Therefore, an advanced approach to efficiently utilize frequently idle or underutilized vector units without vectorization and to achieve significant performance improvement is required. To efficiently utilize the vector units, we decided to use them as additional scalar units through simple instruction translations. To improve the total performance, the task of the vector units can be offloaded partially. In this regard, the CPU and vector units should be able to execute in parallel, which is possible because most vector units are decoupled from the main CPUs. As a remaining issue, because CPUs must assign the tasks to the vector units, the task offloading cost on the CPU-side must be less than the corresponding execution cost such that offloading can be utilized.

To verify the execution latencies of each work on the CPU side, the offloading overhead, and the corresponding vector unit-side work, we measured the difference in execution time when executing a general matrix multiplication (GEMM) application from the Polybench suite [14]. A cycle-accurate simulation was performed to achieve a more accurate measurement of the execution time. Figure 3 shows the hot code region execution time (number of cycles) for the CPU and vector unit-side work with the offloading overhead on the CPU. Figure 3a shows the execution time of the CPU-side work when a CPU executes the hot code region in the CPU alone. Figure 3b shows the execution time when the vector unit is utilized. The offloading overhead is the time when a CPU configures the vector unit's registers and offloads the hot code region to the vector unit. The vector unit-side work is the time when the vector unit executes the hot code region allocated from the CPU side.

As shown in Figure 3, the offloading overhead time is much shorter than the time required by the CPU-side work; therefore, it is proven that offloading can reduce the CPU overhead. However, as a key observation, the time required by the CPU-side work (as shown in Figure 3a) differs from that for the vector unit-side work (as shown in Figure 3b). The time required by the CPU-side work is less than the time required by the corresponding vector unit-side work, and the same trend was observed for all benchmarks in the Polybench suite [14]. This result appears reasonable because each execution unit of the vector unit has a longer latency than the CPU, and the vector unit performs better than the CPU only when multiple execution units of the vector unit are executed in a lockstep. Therefore, because full offloading does not result in a sufficient performance gain, and the job offloading overhead on a CPU is much smaller than the native execution on the CPU, the parallel execution of the workload on both the CPUs and the vector units by segregating them into two partitions is a promising approach.

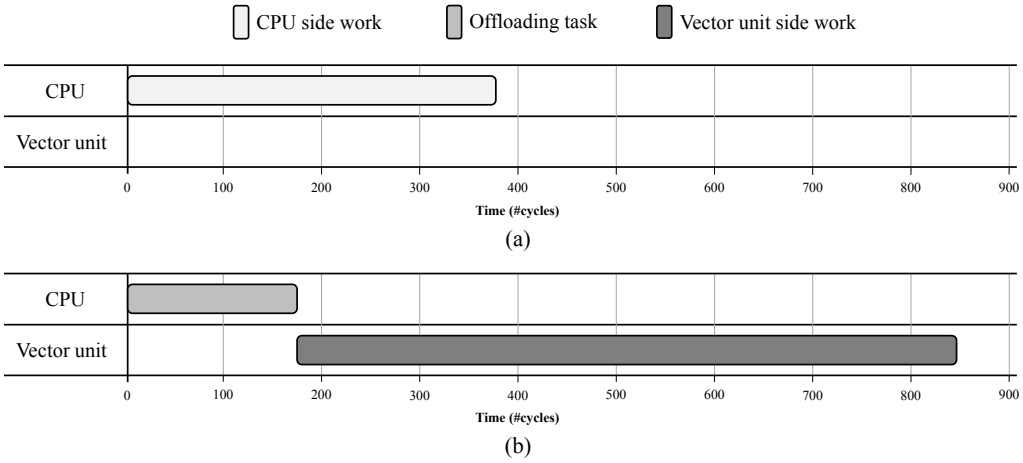

Figure 3. Execution latencies of the GEMM application on the baseline system. The x-axis represents the number of executing cycles: (**a**) the CPU-only execution latency, and (**b**) the vector unit-only execution latency through job offloading.

3.3. Maximizing Total Resource Utilization

A command queue in the vector unit is a container in which a vector scalar unit fetches instructions offloaded from the CPU. If the command queue size is unlimited, the performance will be the best under a parallel execution. Figure 4a shows that, based on this assumption, the CPU can initially offload all partitioned work on the vector unit in order to utilize the vector unit, and then execute the remaining tasks on the CPU-side. Because resource idleness does not occur at any time, the best performance is achieved. However, owing to the limited queue size, CPU stalls often occur when all tasks are assigned to the vector unit. Hence, the CPU should repeat the offloading of some of the tasks (bundle) and, as the vector unit is operating, execute the other tasks on the CPU side until the entire task is completed.

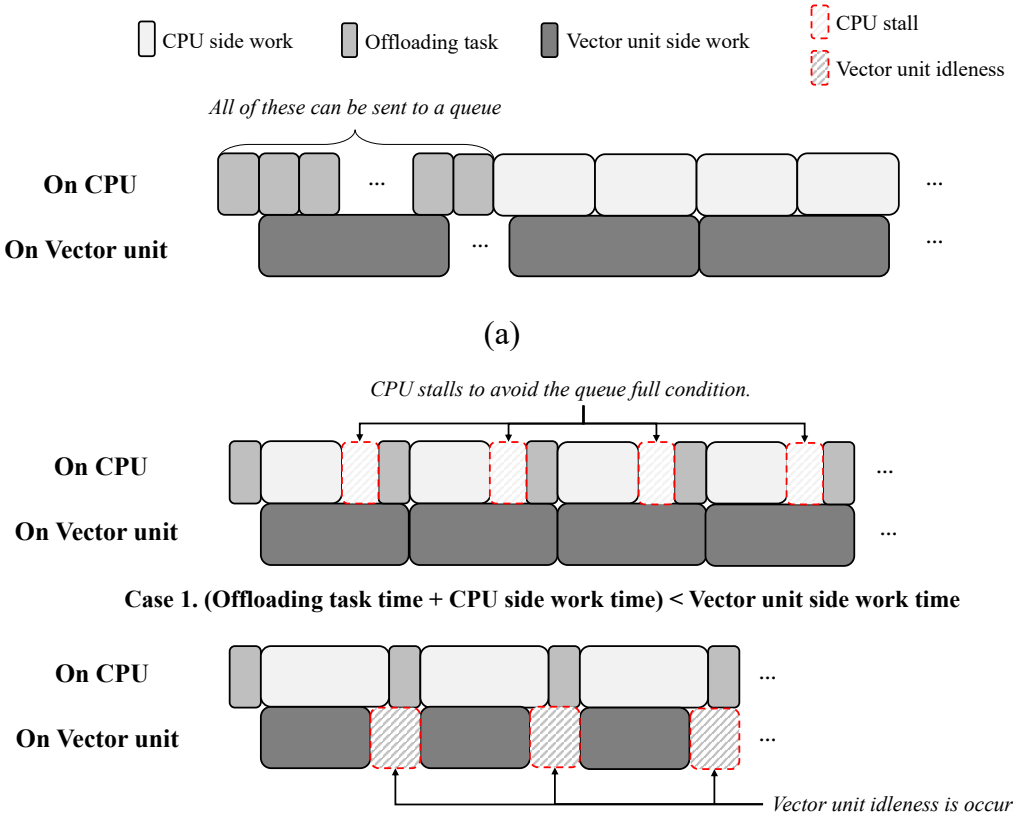

Figure 4. Repeated execution of workload bundles due to the limited command queue capacity: (**a**) an ideal scenario with an unlimited queue capacity, and (**b**) potential performance degradation scenarios on load imbalance between the CPU and the vector unit.

Figure 4b shows the inefficient offloading scenarios, including those on the bundled execution, to avoid the full command queue problem. As shown in Figure 4b, case 1 indicates that the vector unit-side work required a longer time compared with the time required for both the CPU-side work and the offloading overhead combined. In this case, although both the CPU-side and offloading tasks for a bundle are completed, the corresponding offloaded task on the vector unit is not accomplished. Therefore, the total CPU resources cannot be fully utilized. By contrast, Figure 4b shows an example for case 2, in which the time required for the vector unit-side work is less than that for the CPU-side task and offloading overhead combined. In this case, the vector unit will be in an idle state until the CPU completes the next offloading task. Therefore, the assigned workload should

be segregated at an appropriate ratio to fully utilize the CPU and vector unit in parallel, thereby minimizing CPU or vector unit stalls.

3.4. Summary and Insight

To utilize the vector unit efficiently, we considered using the vector unit as an additional scalar computation unit to achieve performance improvements without vectorization, as well as distributing a target workload across the CPU and vector unit to simultaneously execute the partitioned workloads. However, because of problems such as pending communication or resource idleness, it is crucial to determine the amount of work that needs to be transferred to the vector unit. Hence, we proposed a profile-guided optimal offloading ratio search algorithm, which will be discussed in detail in the next section.

4. Vector Offloader

As discussed in Section 3, programmers or compilers face difficulties in extracting the underlying data parallelism for the vectorization of various target applications. To solve this problem, we propose a code translation technique that offloads some portion of the workload to underutilized vector units without explicitly exploring data parallelism, but by retaining the original scalar code. Thus, we first generate instructions to be executed on the vector hardware from the original scalar code. We also insert additional codes to set up the vector hardware and prepare the data on the host-side code. Section 4.1 describes the main concepts for offloading operations to vector units. Moreover, in Section 4.2, we describe how to distribute the workloads on the CPU and vector units simultaneously. A profile-based algorithm for determining the optimal offloading ratio is proposed in Section 4.3.

Figure 5 provides an overview of the workload distribution process for the CPU and vector units. As shown in the figure, we apply optimization techniques to expand the workload distribution area and rewrite the application based on the optimal offloading ratio of the vector unit. The application is initially executed only on the CPU. In sequence, the optimal offloading ratio is calibrated using a profile-guided search algorithm. Finally, the resulting execution binary generated based on the optimal offloading ratio is executed on both the Rocket core-based CPU and the Hwacha vector unit.

4.1. Workload Offloading to Vector Units without Considering Data Parallelism

As previously explained, we propose the Vector Offloader in order to make use of an underutilized vector unit as an additional unit for scalar computation. To conduct scalar operations on the vector unit without hardware modification, we rewrite the application code using the vector unit ISA. Figure 6 shows the code rewriting process for running a target application on a vector unit without exploiting data parallelism. As shown in the figure, the code rewriting process consists of three steps: (1) adding the setup and cleanup codes (host side), (2) translating the scalar instructions into vector instructions (vector unit side), and (3) adding a data preparation code for use in the vector hardware (host side).

As a key to the rewriting process, the vector code should behave like a scalar unit without causing other critical side effects, such as the storage of garbage data in unintended memory locations. Vector operations on a vector unit with multiple vector lanes, especially stores, can produce unintended results and update the data in the main memory. However, in modern vector architectures, masking instructions or a reconfigurable vector length support capability can solve this problem by allowing the instructions to be performed only in a single lane. As a result, by changing the scalar registers to the corresponding vector registers and converting scalar instructions to the corresponding vector instructions, an identical vector code can be created from the original scalar code while producing the same result. Unlike arithmetic operations, in memory access instructions such as load and store, a simple translation into vector instructions does not guarantee the same behavior because of the complexity of memory addressing. Therefore, the CPU thread may need to preload the data into vector registers or pass memory addresses to the vector units, depending on the structures of the specific target vector unit.

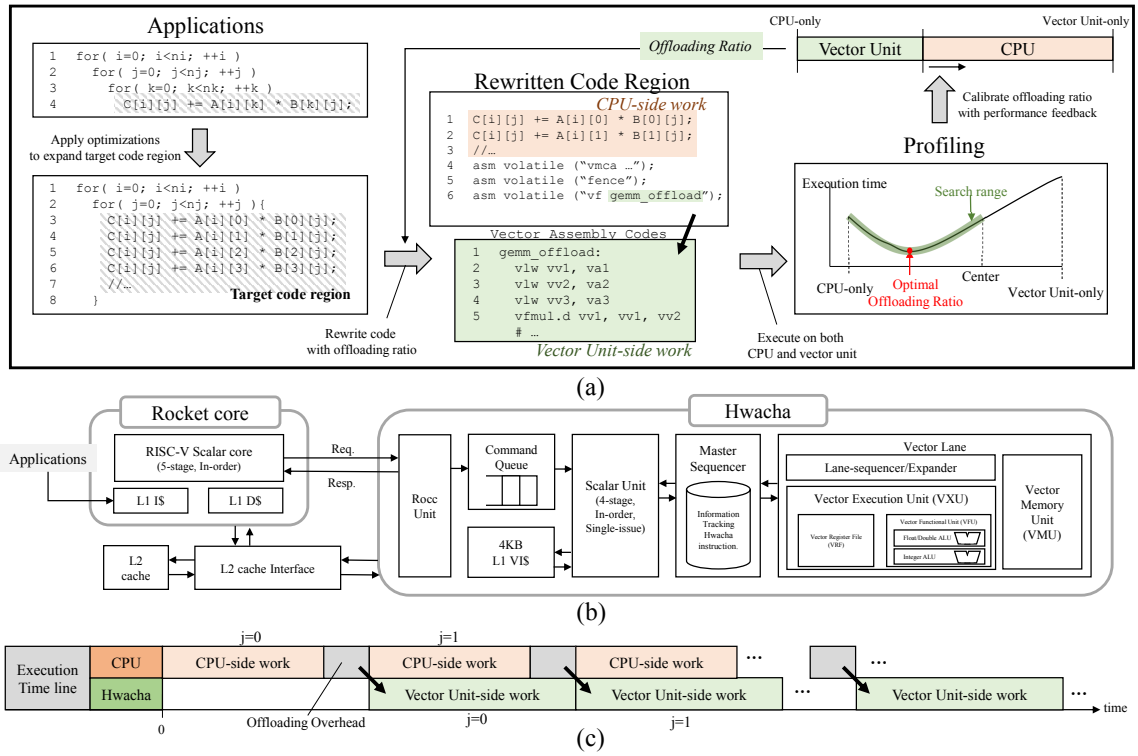

Figure 5. A Vector Offloader overview. (**a**) Based on the offloading ratio to vector units, the application is rewritten to utilize both the CPU and the vector unit. The optimal offloading ratio is obtained through a profile-based algorithm. (**b**) Rewritten applications execute in parallel on both a Rocket core and a Hwacha vector unit. (**c**) An ideal execution scenario with perfect workload balance between the CPU and the vector unit.

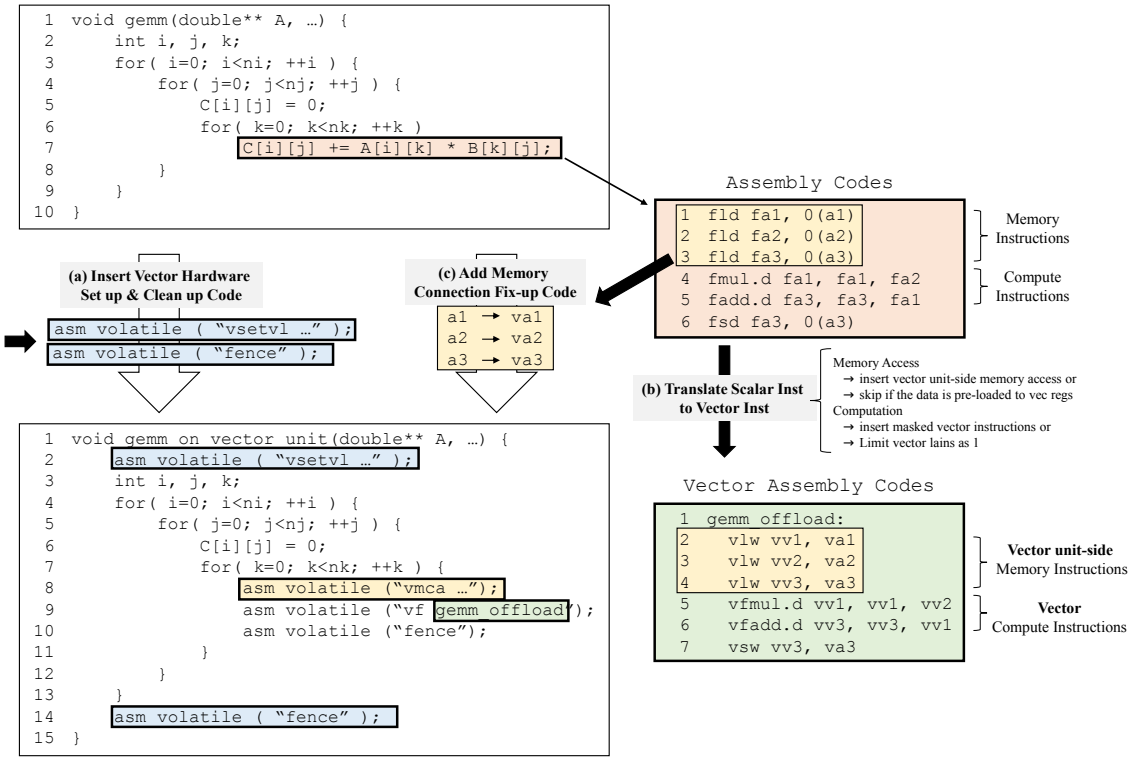

Figure 6. Detailed illustration of a code rewriting technique to offload workloads to vector units: (**a**) inserting vector hardware setup and cleanup codes, (**b**) converting scalar registers and instructions to corresponding vector registers and instructions, and (**c**) adding memory connection fix-up codes.

Based on these key considerations, we first insert the vector hardware setup and cleanup codes (Figure 6a) at the start and end points of the host-side workload (an original program code executed on the CPUs). At this point, the vector hardware is set to use only a single lane. Vector instructions are then generated based on the original CPU-based assembly code by converting the scalar instructions into the equivalent vector instructions (Figure 6b). If the vector hardware cannot be configured to use only a single lane in the setup code, masked vector instructions are used depending on the target vector hardware. The memory connection fix-up code is additionally generated in order for the vector unit-side memory operations to execute correctly; this is performed by preloading the memory data or by passing memory addresses from the CPU registers to the vector unit registers (Figure 6c). Finally, as shown in Figure 6, to perform the memory preparation and partial offloading workloads in the vector unit instead of the original computations on the CPU, we create modified codes by including the generated codes in the original application code.

4.2. Workload Distribution between CPUs and Vector Units

As discussed in Section 3, offloading entire workloads often results in worse performance on the vector units than on the CPUs in case a highly optimized vector code is not generated by fully exploiting data parallelism. To solve this problem, we support a collaborative execution using the CPU and the vector unit simultaneously. Figure 7 shows the potential opportunities in a general matrix multiplication application (GEMM) for distributing the workload to the CPU and vector units and concurrently performing the workload. We first apply several traditional compiler optimizations, such as the loop unrolling technique, to generate long and straight codes without branch divergence. Figure 7a shows the result of applying loop unrolling to the GEMM. As shown in Figure 7a, although the unrolled code is difficult to vectorize owing to the data dependencies within the loop body (intra-iteration data dependency), a parallel execution over different iterations is possible because there is no inter-iteration dependency. For more fine-grained workload balancing, we propose to split each iteration into CPU- and vector unit-side portions, and to concurrently execute the vector unit-side partition for the current (j) iteration and the CPU-side partition for the next (j + 1) iteration, similar to a two-stage pipeline execution. The concurrent execution of the CPU and vector units can be easily implemented by enforcing a partial order between the CPU and vector unit partitions within an iteration using fence instructions.

In addition, as more instructions are offloaded to the vector unit through loop unrolling, the vector code can be further optimized. For example, redundant instructions to access the same memory addresses within the vector code can be eliminated, as shown in Figure 7b. This vector code optimization can increase the vector performance; therefore, more workloads can be offloaded to improve the total performance. However, because the exact degree of additional performance improvement cannot be statically predicted, profile-guided optimization is required to determine the optimal offloading ratio of the vector unit partition over the entire workload. When all vector registers are utilized, the offloading ratio cannot increase further; hence, another important factor for the offloading ratio is the target vector unit specification, such as the number of vector registers.

Figure 8 shows the modified code for concurrently executing the GEMM workload on the CPU and the vector unit. As shown in the figure, the CPU-side code is updated by adding vector hardware setup/cleanup codes, data preparation codes, and offloading control codes. A vector assembly code is also generated, and the code is fetched from the target vector unit directly when the host code executes the offloading control codes (vf gemm_offload_balanced). The fence instruction is added to guarantee the correctness of concurrent execution, and the ratio of the vector unit-side portion to the CPU-side portion can change depending on the offloading ratio.

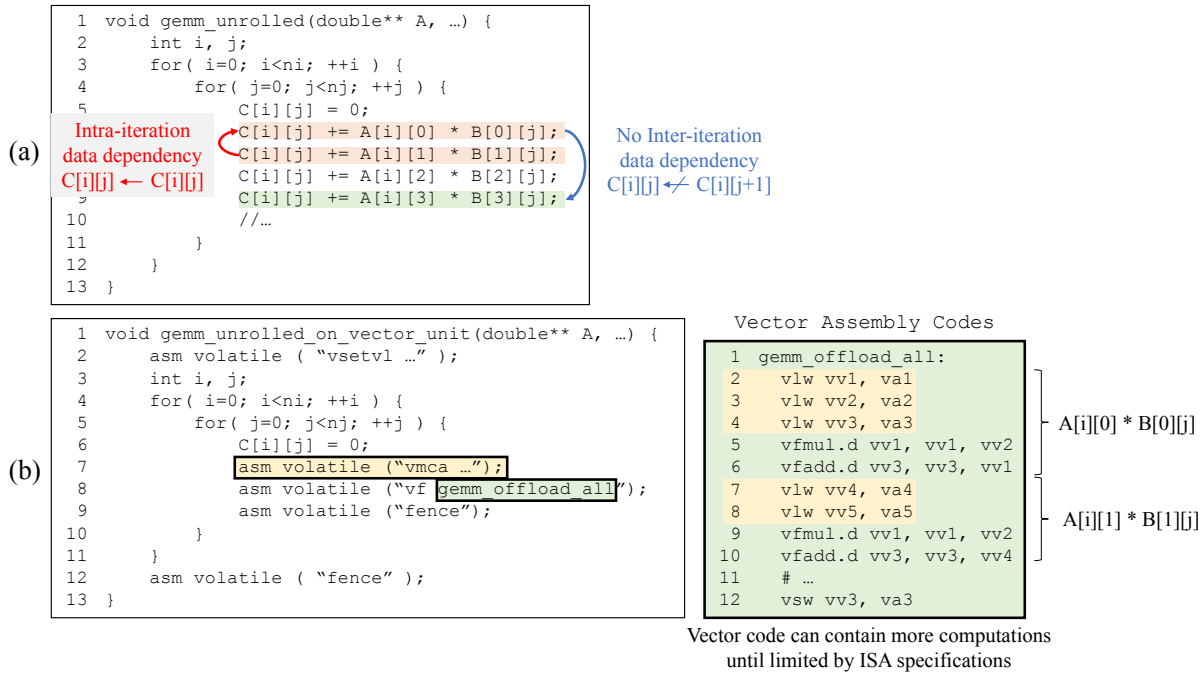

Figure 7. A code rewriting example of workload offloading for a GEMM application. (**a**) Increased concurrent execution opportunities from loop unrolling. (**b**) Additional vector code optimization opportunities with an increased number of offloaded instructions.

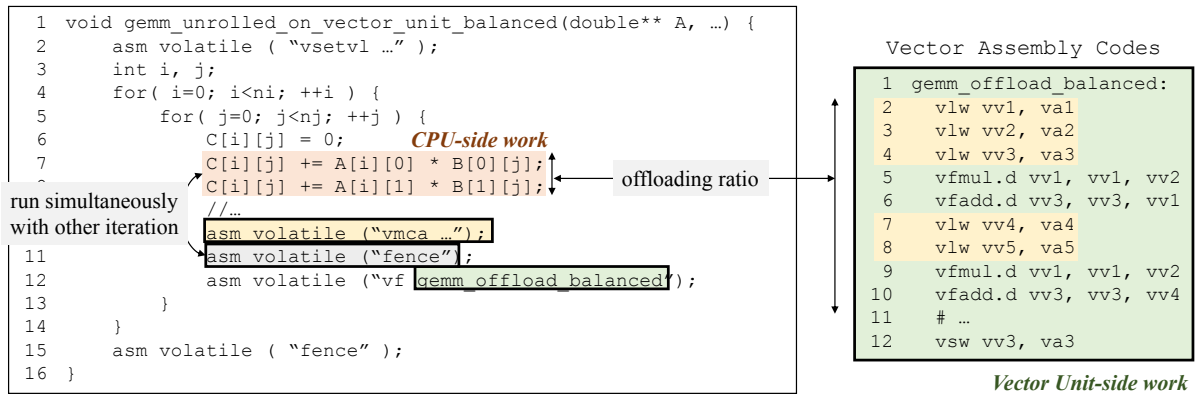

Figure 8. The generated application code for concurrent execution on both the CPU and the vector unit.

4.3. Profile-Guided Optimal Offloading Ratio Search Algorithm

In this study, we propose a profile-guided search algorithm to determine the optimal offloading ratio. Algorithm 1 represents a simple search algorithm that gradually increases the ratio of offloading to the vector hardware until the total performance is saturated. It first initializes the variables for the search (lines 1–2) and iteratively searches for the optimal offloading ratio by executing the rewritten program based on the calibrated offloading ratio (lines 3–10). The offloading ratio continues to increase as the resulting performance improves (line 9). The algorithm increases the offloading ratio by two in order to reduce the search cost, while the unit ratio indicates the smallest instruction set that can be offloaded. The offloading ratio can increase to a given maximum limit value (line 3). To avoid problems caused by the illegal execution of vector fetch instructions such as branch misprediction, the maximum value is set to guarantee that the CPU performs the least amount of work in order to avoid exceptional cases. After determining the offloading ratio at the point of performance saturation (lines 3–10), the algorithm reduces the offloading ratio by 1 once, and then chooses the final optimal offloading ratio by comparing their performances

(lines 11–17). The offloading ratio can increase up to the maximum offloading ratio (line 3) without a break (line 6), which means that vector-only execution is the best option.

Algorithm 1 Find Optimal Offloading Ratio

Require: *Application, Maximum Offloading Ratio*
Ensure: *Minimum Cycles and their Offloading Ratio*
1: initialize OffloadingRatio := 1
2: initialize MinResultCycles, ResultCycles
3: **while** OffloadingRatio < MAX_OFFLOADING_RATIO **do**
4: ResultCycles := RunApp (OffloadingRatio)
5: **if** ResultCycles > MinResultCycles **then**
6: **break**
7: **end if**
8: MinResultCycles := ResultCycles
9: OffloadingRatio += 2
10: **end while**
11: **if** OffloadingRatio < MAX_OFFLOADING_RATIO **then**
12: ResultCycles := RunApp (OffloadingRatio - 1)
13: **if** ResultCycles < MinResultCycles **then**
14: MinResultCycles := ResultCycles
15: OffloadingRatio -= 1
16: **end if**
17: **end if**
18: **return** OffloadingRatio, MinResultCycles

5. Experiments

5.1. Experimental Setup

Hardware Setup : Vector Offloader was evaluated on a VCU118 [13] embedded board equipped with a Xilinx Virtex UltraScale+ FPGA. A RISC-V Rocket core [19], which is a five-stage in-order scalar processor, and a Hwacha [12] vector coprocessor provided by the Chipyard framework [22] were used as the CPU and the vector unit, respectively.

Benchmark and Software Setup : Target applications were compiled with the optimization level 3 option (-O3) using the RISC-V GNU Compiler Toolchain [23] by default. To measure the performance gain of the proposed solution, various applications from well-known Polybench benchmark suites [14] with three different data types (double, float, and integer) were used. Mini and small data sizes from Polybench were used to run the benchmarks. Detailed information on the target benchmarks is presented in Table 1. To evaluate performance, RISC-V "RDCYCLE" instruction was used to count the number of cycles of target applications. Note that the experimental result shows the number of cycles for a core computation of each benchmark because other processes, such as data initialization, do not affect the resulting performance.

Table 1. Benchmark and Evaluation Environments.

Name	Input Size	Data Type	Compile Option	Description
Gemm	• Mini • Small	• Double • Float • Integer	• O0 • O1 • O2 • O3	Matrix-multiply C=alpha.A.B+beta.C
2 mm				Two Matrix Multiplications (D = A.B; E = C.D)
Stax				Matrix Transpose and Vector Multiplication
Bicg				BiCG Sub Kernel of BiCGStab Linear Solver
Gesummv				Scalar, Vector, and Matrix Multiplication
Mvt				Matrix Vector Product and Transpose
Syrk				Symmetric rank-k operations
Syr2k				Symmetric rank-2k operations
Gemver				Vector Multiplication and Matrix Addition

5.2. Experimental Results

5.2.1. Performance Evaluation

To evaluate the effectiveness of the technique, the performance of the solution was compared with the baseline CPU-only version by executing the Polybench benchmark programs with three different data types for the two datasets. Furthermore, the performance of the vector unit-only execution was compared with that of the baseline. This was intended to show the inefficiency when simply using the vector unit as an additional unit for scalar computations and offloading all workloads to the vector unit. Each bar in Figure 9 indicates the relative performance improvements in the Vector Offloader and the vector unit-only execution, normalized to the baseline performance. The graphs on the left side of Figure 9 show that the Vector Offloader achieves a speedup of 1.26×, 1.23×, and 1.21× with respect to double, float, and integer types on the mini dataset, respectively. In addition, speedups of 1.35×, 1.36×, and 1.27× were achieved on double, float, and integer types on the small dataset, respectively, as shown in the graphs in the middle portion of Figure 9.

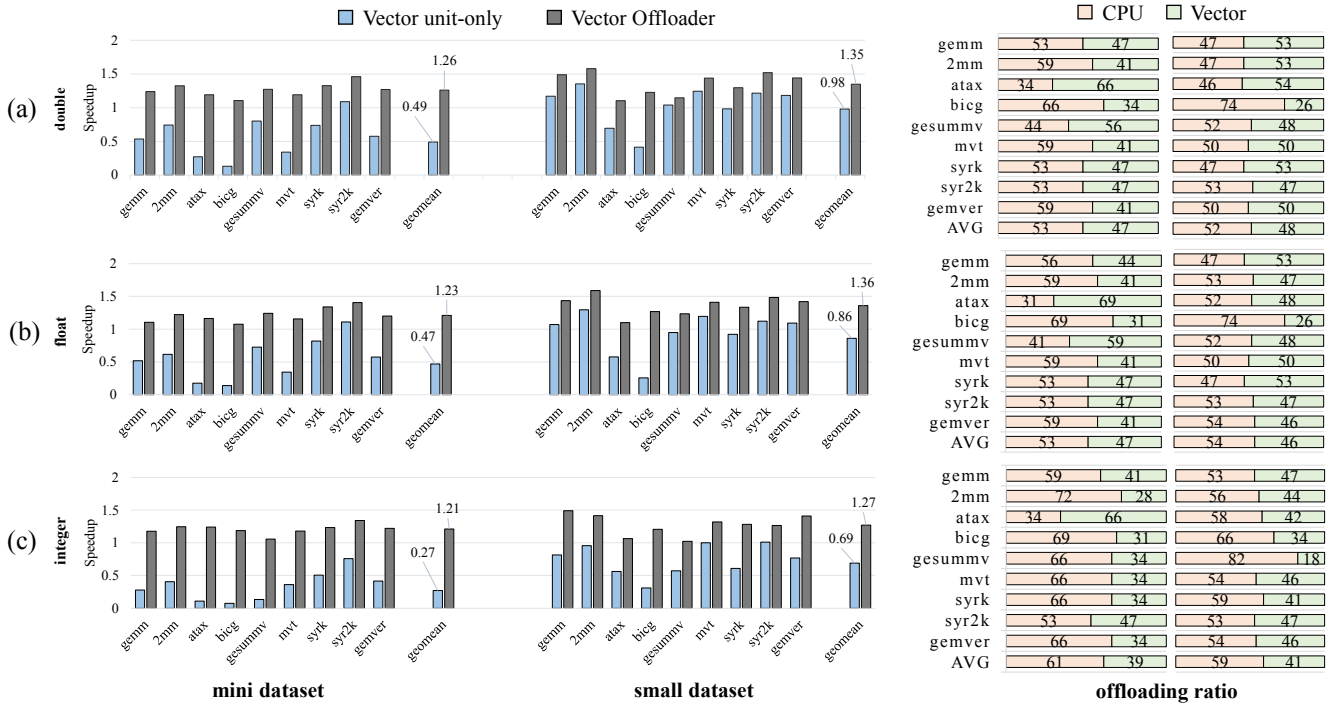

Figure 9. Performance evaluation and optimal offloading ratios on two datasets (mini, small) with three data types of (**a**) double, (**b**) float, and (**c**) integer.

An interesting point here is that a substantial performance degradation is shown in the vector unit-only execution. This is because the vector unit-only execution, as illustrated in Figure 3, requires an additional offloading overhead on the CPU side, and scalar operations without vectorization on the vector unit are much slower than a CPU-only execution. The average speedup of the Vector Offloader on the small dataset was higher than that of the mini dataset. This is because a loop tiling optimization technique is applied to support the small dataset efficiently by solving the vector register number limitation, and the optimization can improve the cache performance.

5.2.2. Optimal Offloading Ratio

The graphs on the right side of Figure 9 show the optimal offloading ratio of each evaluation, which was determined when the profile-guided search was completed. In these graphs, the optimal offloading ratio varies significantly depending on the benchmark characteristics, such as the computation/memory instruction ratio, data type, and dataset size. In general, allocating more workload to the CPU than the vector unit tends to be

optimal because the process of scalar operations without vectorization on the vector unit is relatively slower than the CPU-only execution, and vector unit offloading incurs additional overhead on the CPU side. When comparing the performance of the benchmarks with different variables of double and float types, the optimal offloading ratios are shown when the CPU portion is slightly higher than the vector unit portion. However, much more work should be allocated to the CPU side than the vector unit side when executing integer-type benchmarks because the CPU can execute integer operations much faster than the vector unit.

5.2.3. Performance Comparison with Different Compile Options

To demonstrate the effectiveness of the profile-guided optimization, the performance of the Vector Offloader was compared with the baseline with different compiler optimization levels from -O0 to -O3. For these experiments, although CPU-side codes were compiled with different optimization levels, vector unit-side codes that were generated from the CPU side compiled with an -O3 option were used to change the performance ratio of the CPU to the vector unit. In this environment, when a CPU-side code is compiled with an -O0 option, the performance of the CPU side can be considered slow, and when the CPU-side code is compiled with an -O3 option, the performance of the CPU side can be considered fast. Therefore, the ratio of the vector unit-side workload to the total workload and the performance gain over the CPU-only execution at lower optimization levels are expected to be higher than those at higher optimization levels.

Figure 10a shows the speedups of Vector Offloader relative to the CPU-only execution at different optimization levels. The x-axis shows the benchmarks, and the y-axis shows the speedup of the CPU-only execution at the same compiler optimization level. As expected, higher performance gains over the baseline were achieved at lower optimization levels. This is because a larger workload partition is offloaded to the vector unit at lower optimization levels, as shown in Figure 10b. As the optimization level increased from 0 to 3, the average portion of the offloaded workload decreased from 100 to 47. In this graph, the optimal ratio at the -O0 optimization level is 100, which means that vector unit-only execution is optimal because the CPU side execution is too slow. The results for the -O2 and -O3 optimization levels are identical because the generated binaries are the same at these optimization levels.

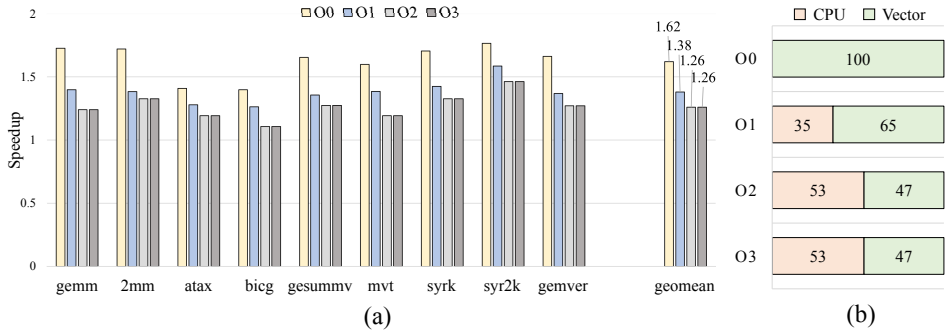

Figure 10. Performance evaluation and optimal offloading ratios when changing the CPU-side performance using different optimization levels (O0, O1, O2, and O3). (**a**) performance evaluation and (**b**) optimal offloading ratios.

6. Related Work

To efficiently utilize vector hardware, many studies have focused solely on the efficient scheduling of SIMD instructions, using improved auto-vectorization techniques or hardware modifications. However, these approaches could not fully utilize the vector unit. In this study, we proposed an alternate approach of utilizing the vector unit as an additional scalar unit rather than leaving it idle or underutilized.

Previous studies [24–26] have focused on utilizing all available resources in parallel. SKMD [24] presented a framework that manages the execution of a single kernel on avail-

able CPUs and GPUs by distributing the target workload. To maximize the performance in this framework, subsets of the data-parallel workload were assigned to multiple CPUs and GPUs by considering the performance differences and data transfer costs depending on the type of device. Paragon [25] proposed a compiler platform to run possible data-parallel applications on CPUs and GPUs, which checks the dependency of speculatively running loops on the GPU. When a dependency is detected, it transfers the execution to the CPU, and after the dependency is resolved on the CPU, the execution is restarted on the GPU. Qilin [26] is a heterogeneous programming system that generates code for both the CPU and GPU. It automatically partitions threads to the CPU and GPU by applying offline profiling and adaptive mapping of the computation to the target hardware. The key difference between our approach and these prior studies [24–26] is that they considered the GPU as the target, whereas our method considers the vector unit as the target.

On the hardware side, SIMD-Morp [27] introduced a flexible SIMD datapath to accelerate non-data-parallel applications by adding connectivity between the SIMD lanes. However, this method requires complex hardware configurations and good programming skills; moreover, efficient compiler support is necessary. Contrarily, our method does not change the baseline vector unit structure, and compiler support is much easier than vectorization.

7. Conclusions and Future Work

Most modern processors are equipped with vector units to exploit data-level parallelism. However, accelerating target applications using a vector unit is highly limited by the non-vectorizable (scalar) portions; therefore, the vector unit often remains underutilized. To overcome the underutilization of existing vector units, in this study, we proposed a technique for executing scalar programs by considering the vector unit as an additional scalar operation unit. We applied a code rewriting process and offloaded the split workloads based on the optimal offloading ratio into both the CPU and the vector unit. To identify the optimal offloading ratio, we presented a profile-based search algorithm that gradually increased the ratio of offloading to the vector unit until no additional improvement in performance was achieved. Our proposed method showed average speedups of 1.26×, 1.23×, and 1.21× with respect to double, float, and integer types on a mini dataset, and average speedups of 1.35×, 1.36×, and 1.27× for double, float, and integer types on a small dataset, respectively.

This study includes the following directions for future research: First, we believe that the Vector Offloader can be utilized in existing general parallelization infrastructures. Therefore, we will attempt to generalize our technique using existing parallel programming APIs such as OpenMP. Second, we will maximize the effectiveness of our technique using additional hardware modifications. Since our work can limit the vector width to support scalar codes on target vector units, fine-grained power control can substantially improve the power efficiency of our technique. To realize this, we have already applied the fine-grained clock gating logic to the baseline Hwacha architecture; to handle it, we will utilize the lane-wise clock gating for offloading with the use of customized instructions.

Author Contributions: The first author, Y.S., and the corresponding author, Y.P., are responsible for the overall work in this paper. The second author, S.K., helped to design the framework and write the paper. Moreover, he provided useful comments for the manuscript. The third author, H.U., and the fourth author, S.L., helped to set up the FPGA-based evaluation environments and write the paper. The fifth author, J.H., and the sixth author, D.K., performed several experiments and visualized the experimental results. All authors have read the manuscript and agreed to publish the current version of the manuscript.

Funding: This research was supported by the National Research Foundation of Korea (NRF) grants (2020M3H6A1085498, 2020M3H6A1085535), by the MOTIE (Ministry of Trade, Industry, and Energy (10080613), and by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant (No.2020-0-01373), funded by the Korean government (MSIT).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Raman, S.; Pentkovski, V.; Keshava, J. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro* **2000**, *20*, 47–57. [CrossRef]
2. Lomont, C. *Introduction to Intel Advanced Vector Extensions*; Intel White Paper; 2011; Volume 23. Available online: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf (accessed on 27 November 2021).
3. Goodacre, J.; Sloss, A. Parallelism and the ARM instruction set architecture. *Computer* **2005**, *38*, 42–50. [CrossRef]
4. Gou, C.; Kuzmanov, G.; Gaydadjiev, G. SAMS multi-layout memory: Providing multiple views of data to boost SIMD performance. In Proceedings of the 24th ACM International Conference On Supercomputing, Tsukuba, Ibaraki, Japan, 2–4 June 2010; pp. 179–188.
5. Govindaraju, V.; Nowatzki, T.; Sankaralingam, K. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK, 7–11 September 2013; pp. 341–351.
6. Chen, Y.; Mendis, C.; Carbin, M.; Amarasinghe, S. VeGen: A Vectorizer Generator for SIMD and Beyond. In Proceedings of the 26th ACM International Conference on Architectural Support For Programming Languages and Operating Systems, Virtual, USA, 19–23 April 2021; pp. 902–914. [CrossRef]
7. Park, Y.; Seo, S.; Park, H.; Cho, H.; Mahlke, S. SIMD Defragmenter: Efficient ILP Realization on Data-Parallel Architectures. In Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, 3–7 March 2012; pp. 363–374. [CrossRef]
8. Maleki, S.; Gao, Y.; Garzar, M.; Wong, T.; Padua, D. An evaluation of vectorizing compilers. In Proceedings of the 2011 International Conference On Parallel Architectures And Compilation Techniques, Galveston, TX, USA, 10–14 October 2011; pp. 372–382.
9. Stephens, N.; Biles, S.; Boettcher, M.; Eapen, J.; Eyole, M.; Gabrielli, G.; Horsnell, M.; Magklis, G.; Martinez, A.; Premillieu, N.; et al. The ARM scalable vector extension. *IEEE Micro* **2017**, *37*, 26–39. [CrossRef]
10. RISCV-V Release Vector Extension 1.0, Frozen for Public Review. RISCV/RISCV-V-spec. GitHub. 2021. Available online: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0 (accessed on 20 September 2021).
11. Smith, J.; Faanes, G.; Sugumar, R. Vector Instruction Set Support for Conditional Operations. In Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, BC, Canada, 10–14 June 2000; pp. 260–269. [CrossRef]
12. Lee, Y. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*; Technical Report No. UCB/EECS-2016-117; University of California: Berkeley, CA, USA, 2016.
13. VCU118 Evaluation Board. Available online: https://www.xilinx.com/products/boards-and-kits/vcu118.html (accessed on 17 October 2018).
14. PolyBench/C the Polyhedral Benchmark Suite 3.1. Available online: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/#documentation (accessed on 13 November 2011).
15. Flynn, M. Some computer organizations and their effectiveness. *IEEE Trans. Comput.* **1972**, *100*, 948–960. [CrossRef]
16. Nvidia, C. Nvidia's next generation cuda compute architecture: Fermi. *Comput. Syst.* **2009**, *26*, 63–72.
17. Lee, Y.; AviZienis, R.; Bishara, A.; Xia, R.; Lockhart, D.; Batten, C.; Asanovic, K. The Maven vector-thread architecture. In Proceedings of the 2011 IEEE Hot Chips 23 Symposium (HCS), Stanford, CA, USA 2011; p. 1.
18. Batten, C. *Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators*; Massachusetts Institute of Technology: Cambridge, MA, USA, 2010.
19. Asanović, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A.; et al. The Rocket Chip Generator. (EECS Department, University of California, Berkeley, 2016, 4). Available online: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html (accessed on 15 April 2016).
20. GNU Compiler Collection (GCC). Available online: http:/gcc.gnu,org (accessed on 6 December 2018).
21. Clang: A C Language Family Frontend for LLVM. Available online: https://clang.llvm.org (accessed on 12 October 2020).
22. Amid, A.; Biancolin, D.; Gonzalez, A.; Grubb, D.; Karandikar, S.; Liew, H.; Magyar, A.; Mao, H.; Ou, A.; Pemberton, N.; et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* **2020**, *40*, 10–21. [CrossRef]
23. RISC-V GNU Compiler Toolchain. 2017. Available online: https://github.com/riscv-collab/riscv-gnu-toolchain (accessed on 31 December 2017).
24. Lee, J.; Samadi, M.; Park, Y.; Mahlke, S. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In Proceedings of the 22nd International Conference On Parallel Architectures And Compilation Techniques, Edinburgh, UK, 7 October 2013; pp. 245–255.
25. Samadi, M.; Hormati, A.; Lee, J.; Mahlke, S. Paragon: Collaborative Speculative Loop Execution on GPU and CPU. In Proceedings of the 5th Annual Workshop On General Purpose Processing With Graphics Processing Units, London, UK, 3 March 2012; pp. 64–73. [CrossRef]
26. Luk, C.; Hong, S.; Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, USA, 12–16 December 2009; pp. 45–55.
27. Dasika, G.; Woh, M.; Seo, S.; Clark, N.; Mudge, T.; Mahlke, S. Mighty-Morphing Power-SIMD. In Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Scottsdale, AZ, USA, 24–29 October 2010; pp. 67–76. [CrossRef]