# Compilation and Architecture Support for Customized Vector Instruction Extension

Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Hui Huang, Bin Liu, Raghu Prabhakar, Glenn Reinman and Marco Vitanza
Computer Science Department
University of California, Los Angeles
{cong, ghodrat, mgill04, huihuang, bliu, raghu, reinman, mvitanza}@cs.ucla.edu

**ABSTRACT - Vectorization has been commonly employed in modern processors. In this work we identify the opportunities to explore customized vector instructions and build an automatic compilation flow to efficiently identify those instructions. A composable vector unit (CVU) is proposed to support a large number of customized vector instructions with small area overhead. The results show that our approach achieves an average 1.41X speedup over the state-of-art vector ISA. We also observe a large area gain (around 11.6X) over the dedicated ASIC-based design.**

## I. Introduction

SIMD vector processors are very effective in executing programs with extensive data-level parallelism, such as multimedia processing, graphics and scientific computing. In recent years, vector extension has become one of the most common additions to both general-purpose microprocessors and super computers due to the growing demands on high-performance computing. There are several state-of-art vector ISAs in the market, such as Intel's AVX [20] and IBM's AltiVec[10].

It has been recognized that customization can achieve significant performance improvement [8], and this is also the case with the vector or SIMD applications. Recently increasing attention has been paid to customized vector ISA support. The authors of [5] propose SystemC-based support for customized vector ISA. The work in[4] introduces a customized vector instruction set for multimedia applications. The work in [16] designs customized SIMD units with high-level synthesis techniques. The newly developed Convey system [19] provides support for application-specific vector instruction sets, where users are allowed to reconfigure the vector ISAs to match different application domains. This trend presents new challenges to both compiler and architecture design to provide an efficient customized vector ISA support with small hardware cost.

At the compiler side, the main challenge is how to efficiently identify application/domain-specific vector instructions and perform automatic customized vectorization. A crucial step to achieving high performance in a customized vector design is the identification of frequently executed vector instructions. Extensive work on customized *scalar* instruction exploration (e.g., [2][6]) already exists. However, a naïve employment of the existing techniques on the input program without considering the vector features will result in inefficient customized vector instruction generation. For example, one important feature of vector processing is the need for memory alignment in the vector architecture [11]. In AltiVec memory accesses are required to be aligned at a 16-byte boundary and it cannot handle unaligned vector loads and stores; In AVX misaligned memory accesses are supported with a large performance penalty. In the customized vector ISA exploration phase, if the memory alignment issue has not been resolved properly, it may result in undesired overhead on performance. Let's consider the vectorizable loop shown below (without loss of generality, in this paper we assume $A[0]$, $B[0]$, etc. are aligned to the memory boundary).

*for i = 0 to n*
$A[i] \mathrel{+}= B[i+1]*C[i];$

The scalar customized instruction candidates inside this loop only contain one *multiply-add* (MAC) operation. While for vector exploration, since additional alignment instructions are required to resolve the unaligned array reference $B[i+1]$, both MAC and aligned MAC should be considered as customized vector instruction candidates, as shown in Fig. 1(b). If we simply replace the sequential loop with unaligned vector MAC operations, it may result in either incorrect execution or pay a considerable performance penalty.

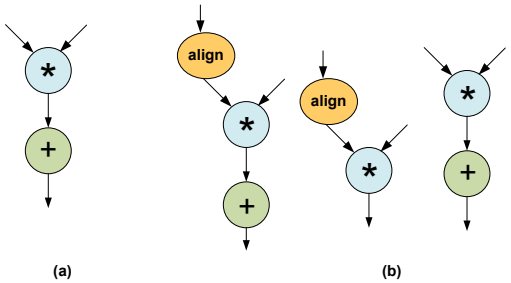


**Fig. 1. (a) Customized scalar instruction candidates (b) Customized vector instruction candidates.**

Earlier implementations of vector processor [1][12] are all based on non-customized vector instructions. The vector instructions in VIRAM [13] are designed to vectorize embedded system applications by adding support for narrower data-type and different styles of permutation. VESPA [17] is a flexible FPGA-based vector engine; however it only supports integer operations.

The key contributions of this work are summarized as follows:

(1) We develop an automatic compilation flow to extract customized vector instructions from one or a set of applications. Pattern recognition approaches have been used here to identify alignment-inclusive customized vector instruction candidates.

(2) We propose composable vector processing units (CVUs) that can be chained together to create customized vector instructions. This design allows programmable customized vector extensions and can achieve an average 1.41X speedup over standard vector ISA and 11.6X area gain over the dedicated ASIC-based design.

## II. Motivational Example

In this section we illustrate the existence of customizable vector patterns with one real-life application.

Let's consider one computation kernel of *Jacobi Rician-denoise* [19] in the medical imaging domain. The kernel loop in this application performs iterative local denoising on a 2D image, as shown in Fig. 2. Twelve array references are involved in the computation kernel as inputs and the loop body can be vectorized without violating data dependencies (here we only consider vectorization through the innermost loop).

```
for m = 1 to M - 1
    for n = 1 to N - 1
        u[m][n] = (ulast[m][n] + DT * (ug[m][n+1] +
                                       ug[m][n-1]  +
                                       ug[m+1][n] +
                                       ug[m-1][n]  +
                                       GM*f[m][n]))
                  / (c[m][n] + DT * (g[m][n+1] +
                                     g[m][n-1]  +
                                     g[m+1][n] +
                                     g[m-1][n]  +
                                     r[m][n]))
```

**Fig. 2. One kernel loop in *Jacobi Rician-denoise*.**

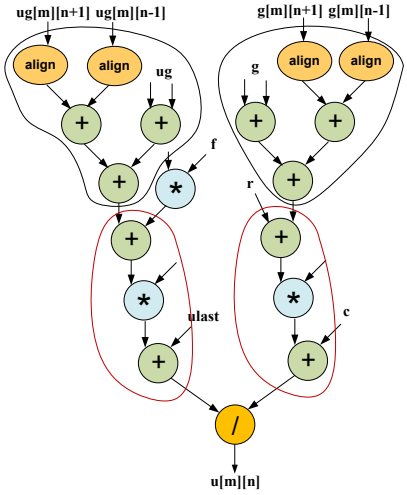


**Fig. 3. Data flow graph of *Jacobi Rician-denoise*.**

Fig. 3 shows the corresponding data flow graph for the vectorizable loop, in which each node represents a vector instruction, such as vector add or vector multiply. Alignment instructions have been appropriately inserted to ensure the functionality correctness.

From Fig. 3, we can see that the two branches of *div* operation are similar to each other in terms of operation counts, data type and data path. The left branch contains nine operation nodes and the right one contains eight nodes – differing by one *mul* operation only. This provides an opportunity to extract repeatedly executed customized vector instructions, sized from one operation to eight operations. Two vector pattern candidates with occurrence equaling two have been highlighted in Fig. 3.

## III. Customized Vectorization Flow

In this work, we build an automatic compilation flow to perform customized vectorization. The overall framework is implemented in the LLVM compiler infrastructure [21] together with the Omega Library [22] for dependency analysis. This flow is invoked as a back-end pass on the optimized LLVM intermediate representation (IR) code.

Fig. 4 shows the kernel loop in *Rician-denoise* application (to better illustrate each step in the framework, we use *Gauss-Seidel* implementation [19] with loop-carried true dependency). The corresponding data flow graph of LLVM IR is shown in Fig. 5. Each node in the data flow graph is labeled with the operation it performs and each edge represents the data flow dependency between two nodes. For example, node *phi* generates the value of loop induction variable *n* and the *gep* node is used to generate the pointer/ address for array references.

```
for m = 1 to M - 1
    for n = 1 to N - 1
        u[m][n] = (u[m][n] + DT * (u[m][n+1] +
                                   u[m][n-1]  +
                                   u[m+1][n] +
                                   u[m-1][n]  +
                                   GM*f[m][n]))
                  / (c[m][n] + DT * (g[m][n+1] +
                                     g[m][n-1]  +
                                     g[m+1][n] +
                                     g[m-1][n]  +
                                     r[m][n]))
```

**Fig. 4. Kernel code of *Gauss-Seidel Rician-denoise*.**

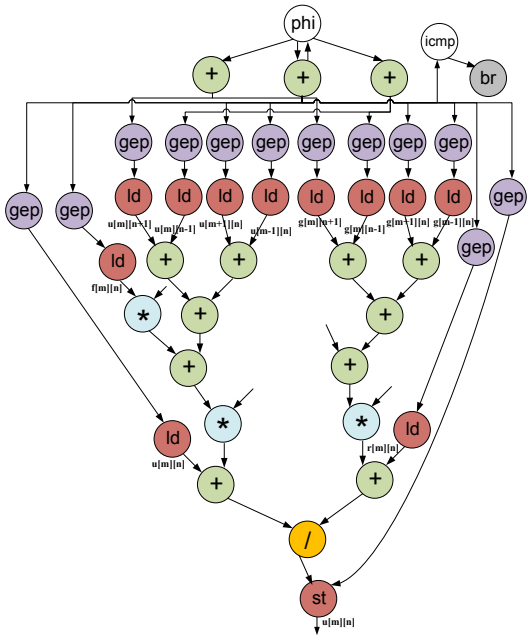


**Fig. 5. Data flow graph of *Gauss-Seidel Rician-denoise*.**

Our compilation flow is described in the following steps:

### *A. Vectorizable Code Region Extraction*

To extract the customized vector patterns which perform real computations, we need to remove the complementary nodes existing in the original LLVM IR, such as the loop invariant or branch instructions. For example, in Fig. 5 the three *add* operations below the *phi* node perform address calculation instead of real computation, and thus should not be explored as customized vector instruction candidates. In our flow, we propose a *boundary-node-directed* vectorizable code extraction approach. Here the *boundary nodes* of the vectorizable code region in a loop *L* are defined to be the legal inputs and outputs to a vector instruction, including memory load/store operations to continuous/strided memory, constants, etc. Any node that is located outside the subgraph enclosed by the boundary nodes will be removed, as shown in Fig. 6(a).

### *B. Vectorizability Checking*

To fully investigate the customized vector pattern candidates, we allow partial vectorization in our flow. As shown in Fig. 6(b), the violated dependency is between $u[m][n\text{-}1]$ and $u[m][n]$.

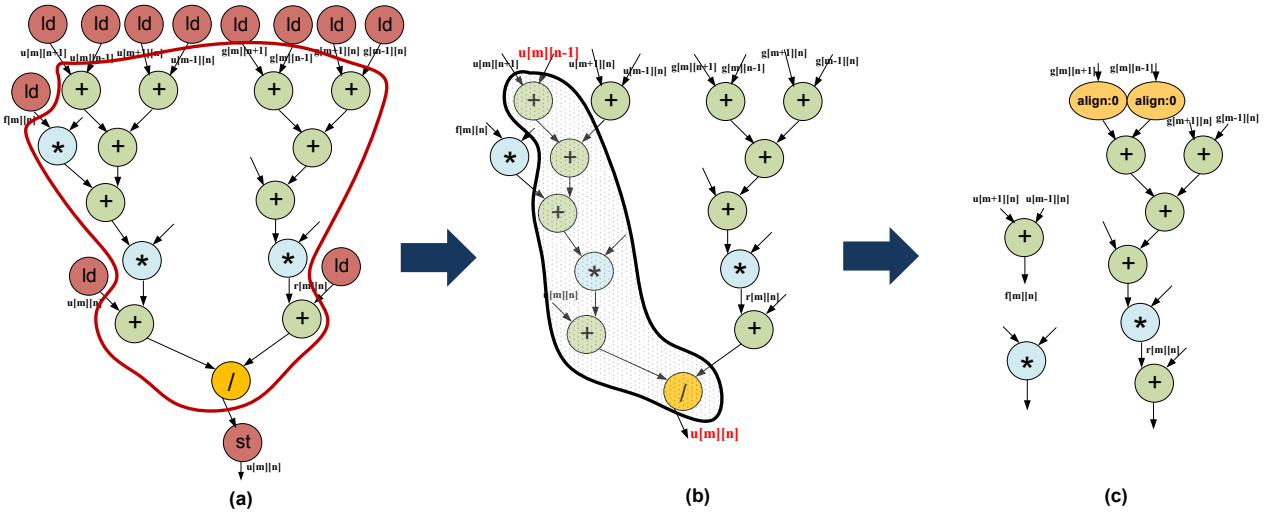


**Fig. 6. Customized vector instruction identification flow on the kernel code of *Gauss-Seidel Rician-denoise*.**

Therefore the *add* operation associated with $u[m][n\text{-}1]$ is excluded from the vectorizable code region, as well as the downstream nodes reachable from $u[m][n\text{-}1]$. This loop can be partially vectorized by executing the unshaded nodes in parallel. In this case, the exploration space for the customized vector pattern is enlarged, providing more opportunity to extract beneficial instruction candidates.

### *C. Vectorizable Data Flow Graph Expansion*

After vectorizability checking, alignment instructions are explicitly inserted into the original data flow graph to ensure functionality correctness in the vectorized code.

An alignment instruction is one that combines the results of two neighboring vector load instructions and logically performs a shift on the vector registers. Note that there are different ways to insert alignment instructions. In [11] four representative shifting schemes are described, as zero-shift, eager-shift, lazy-shift and dominant-shift. In our flow, we compare the four possible schemes for the input program and choose the one with minimal overhead (details of this algorithm are beyond the scope of this paper). Following the selected scheme, the original data flow graph will be expanded to include the corresponding alignment nodes, as shown in Fig. 6(c). In the expanded data flow graph, both $g[m][n\text{+}1]$ and $g[m][n\text{-}1]$ are aligned to $u[m][n]$ by the next two alignment nodes.

### *D. Customized Vector Instruction Extraction*

This section presents our pattern-based approach to efficiently identify vector patterns in the expanded vectorizable code region.

The pattern recognition approach we use is based on [7] which is very scalable in benefit of subgraph enumeration and similarity checking technique. A breadth-first search strategy is adopted in our flow to discover frequent pattern candidates. At step $k + 1$, all the convex patterns with $k$ nodes are extended by one neighbor. After a new subgraph is generated, it is compared to the existing patterns to perform graph isomorphism checking. A *characteristic-vector (CV)* based filtering scheme is adopted to reduce the number of expensive graph isomorphism checking. The characteristic vector is used to capture structural properties of the original subgraph. Therefore if the CV of a subgraph is different than the CV of a given pattern, the corresponding graph isomorphism checking can be avoided (in this work we only consider exactly matched patterns).

After applying the pattern-based approaches to the expanded dataflow graph, we can extract all the frequently executed vector pattern candidates. To measure the gain of a given customized vector pattern, we use $\#occur^{\alpha}\cdot(\#inst - |P_{critical}|/L)$ to estimate the performance improvement . Here *#occur* refers to the number of pattern occurrences considering the loop counts ($\alpha$ is an architecture dependent scaling factor; for example, platforms with tighter constraint on resource should use larger $\alpha$ in the pattern selection process to enforce more sharing); *#inst* represents the estimated scalar execution time; $L$ is the length of vector register, which corresponds to the level of data parallelism supported by the vector architecture. The length of the critical path ($|P_{critical}|$) divided by $L$ is used to estimate the vector execution time for each occurrence of the pattern.

## IV. Architecture Support

To efficiently support customized vector instruction, we propose a design based on *composable vector units (CVUs)*, which is a coarse-grained programmable approach in which the granularity of programmability is single vector unit. In this section, we discuss the architectural support for composable vector units, which can be chained together a support a large number of customized vector instructions.

Fig. 7 shows our architectural support for composable vector units. It consists of a series of CVUs, a programmable crossbar and a sequencer. They are all tightly-coupled connected to the core. The composable vector units are connected together through the programmable crossbar. The inputs to the programmable crossbar are from the outputs of CVUs and the core's register file. The outputs of the crossbar are connected to CVUs. The sequencer, which is programmed by the core, is responsible for reprograming the crossbar in every scheduling step. In this way different connection patterns between CVUs can be supported. Internally the crossbar is a series of multiplexers.

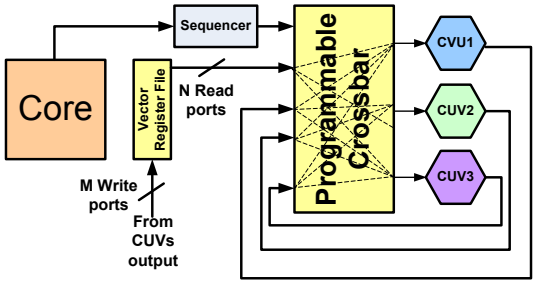


**Fig. 7. Architectural support for CVUs.**

Fig. 8 shows how the sequencer works for implementing a simple pattern like ($Va*Vb + Vc*Vd + Ve$). As we can see, in step (A), $Va*Vb$ is calculated; in step (B), $Vc*Vd$ is calculated and the result of $Va*Vb$ is added to $Ve$; finally in step (C) the two temporary results are added together in VADD1.

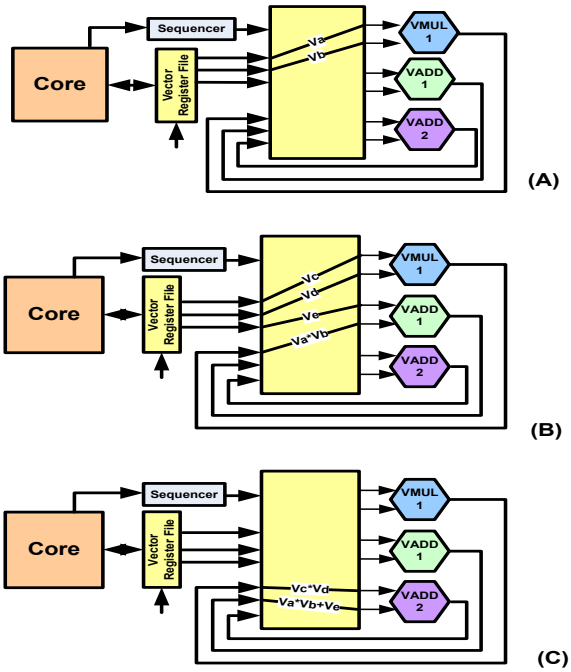


**Fig. 8. Implementation of *Va*Vb+Vc*Vd+Ve.***

Static analysis is performed on the given applications to obtain number and types of the CVUs that we need to include in our design. The internal structure of the programmable crossbar depends on how many CVUs are supported in the system, since it accepts inputs from those CVUs' outputs. The sequencer is responsible for routing correct inputs for CVUs. This is handled by scheduling a vector pattern on available CVUs under vector register-file port constraints. At each scheduling step, the needed configuration bits for the crossbar are generated by compiler and are written into the sequencer. Sequencer consists of a RAM and a small control unit which outputs the RAM content for each composed instruction at each scheduling step. The size of the RAM depends on the maximum height of a vector pattern and also the number of CVUs that can be scheduled in each step.

## V. Evaluation

### *A. Experimental Setup*

We have considered nine applications from widely known standard benchmarks suite like Parsec [3] (*streamcluster* and *swaptions*), Parboil [23] (*cutcp*, *mri-q* and *mri-gridding*) and four applications from the medical imaging domain [8] (*denoise*, *deblur*, *registration* and *segmentation*).

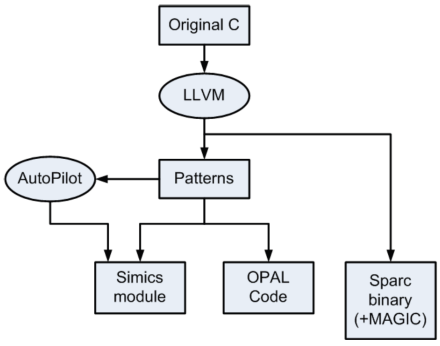


**Fig. 9. The simulation flow.**

We evaluate the proposed customized vector instruction extraction flow by running full system simulations on each benchmark. The overall simulator framework is implemented upon Simics [14] and the GEMS toolset [15] in the single core configuration. Normal vector engine support has been added to this framework. Fig. 9 shows the fully automatic flow that we developed in the extended Simics-GEMS infrastructure for customized vector instruction support. The input to this flow is the original C file. After applying the customization compilation flow introduced in Section III, the selected patterns are passed to AutoPilot [9] to be synthesized. The output of AutoPilot contains the timing information of the custom vector instruction when implemented in hardware. Once we have the patterns with their timing behavior, the Simics module plus the extension to OPAL can be automatically generated.

To evaluate the effectiveness of our proposed CVU-based architecture we also consider an extreme case in which all the custom vector patterns are implemented as ASIC. This is actually an upper bound for performance and not necessary a feasible solution.

### *B. Customized Vector Pattern Recognition*

Table 1 shows the customized vector pattern recognition results for the nine benchmarks kernels. At each row, columns 2-6 represent lines of kernel code, the number of pattern found, the number of pattern instances and runtime, respectively. For example, test bench *streamcluster*, the code in its kernel contains 96 lines of C code, and the 92 vector patterns are found with 240 pattern instances. The overall runtime is less than one second. From Table 1 we can see, the average number of instances for each pattern inside the kernel is around four. The repeated occurrence of the same vector pattern in program kernels offers the opportunity for program execution speedup by providing the customized vector support for the corresponding pattern.

The last column in Table 1 shows the area synthesis result for patterns in each benchmark, in total the area equals 5574062 $um^2$.

**Table 1. Pattern recognition results on nine computation-intensive benchmarks and their synthesized area on ASIC.**

| | #line | #pattern | #inst | time(sec) | Area |
|---|---|---|---|---|---|
| *streamcluster* | 96 | 92 | 240 | 0.13 | 1171 |
| *swaptions* | 152 | 69 | 292 | 0.21 | 1533 |
| *cutcp* | 67 | 78 | 285 | 0.19 | 4718 |
| *mri-q* | 79 | 71 | 100 | 0.33 | 6242 |
| *mri-gridding* | 119 | 119 | 385 | 0.49 | 9043 |
| *denoise* | 274 | 187 | 650 | 0.52 | 1357 |
| *deblur* | 202 | 29 | 151 | 0.24 | 2274 |
| *registration* | 222 | 1499 | 3122 | 1.42 | 5061 |
| *segmentation* | 179 | 2211 | 4172 | 1.72 | 1774 |

### *C. Architecture Parameter Selection*

In this section we discuss the approach we use the extracted pattern information to decide the architecture parameters in the CVU-based design.

#### *C.1. CVU selection*

Table 2 shows the average CVU usage statistics in all the benchmarks. From the result we can see that seven types of CVU are needed (*Other* include *vfsqrt*, *vfcmp*, *vf2i* and *vi2f*). However, there is one problem to have all those CVUs, which can be seen from Fig. 7 – having 7 CVUs will make crossbar huge. On the other hand, by looking at the selected vector patterns we find that those CVUs are not equally frequently appearing in all benchmarks. For example, the *Other* CVU shows a relatively small number of occurrences, thus we can give less priority to those units. By less priority, we mean we can either share them on one crossbar port or use a hierarchy of crossbars and connect these less critical CVUs to lower levels of hierarchy.

The next important architecture parameter is how many CVUs we should include in the system. Having only one for each CVU type may result in a structural hazard, even if the corresponding operations can be executed in parallel. This is the very common for "VFADD", "VFMUL" and "VFDIV", which as Table 2 shows have high usage frequency. In our approach, we first collect the delay of each selected pattern with different numbers of available CVUs. As shown in Fig. 10, the number of "VFADD", "VFMUL" and "VFDIV" are varied from 1 to 3 (shown as different configurations in the x-axis). The corresponding delay multiplied by area is shown in the y-axis. Based on the synthesized area results in Table 3 and Table 4, in the current design we decide to select the 2-2-1 configuration point, namely two vector adders, two multipliers and one divider. This configuration point has the minimum (#steps * area) value among all the points. In Fig. 10 we only show the configurations that have a speedup of more than 10% over the base configuration (1-1-1 case).

**Table 2. Normalized usage (per benchmark)**

| | VFADD/SUB | VFMUL | VFDIV | Other |
|---|---|---|---|---|
| *Aver. usage* | 4.6 | 2.6 | 1.3 | 1 |

**Table 3. CVU synthesis results.**

| CVU | Area (um$^2$) | Power (mW) |
|---|---|---|
| *VFADD/VFSUB* | 35799 | 0.524761 |
| *VFMUL* | 10208 | 0.22694 |
| *VFDIV* | 16237 | 0.246 |
| *VFSQRT* | 264624 | 8.397 |
| *VFCMP* | 1601 | 0.0185 |
| *VF2I* | 4743 | 0.0462 |
| *VI2F* | 7060 | 0.0749 |

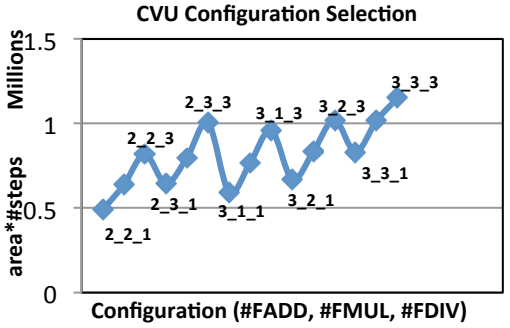


**Fig. 10. Profiling results for CVU selection.**

*C.2. Crossbar design*

Table 4 shows the synthesis results of the crossbar for one lane in the vector engine. These results are generated using AutoPilot and Synopsys Design Compiler [24] for 90nm ASIC. Based on the discussion above, we chose to use 18-input, 12-output crossbar together with a secondary crossbar, as shown in Fig. 11. From the figure we can see two vector adders, two vector multipliers, one divider and one *Other* type of CVU are included the system.

**Table 4. Crossbar synthesis results.**

| # CVU | #in | #out | area (um$^2$) | latency (ns) |
|---|---|---|---|---|
| 3 | 9 | 6 | 35581 | 0.60 |
| 4 | 12 | 8 | 54481 | 0.60 |
| 5 | 15 | 10 | 99567 | 0.61 |
| 6 | 18 | 12 | 129992 | 0.61 |
| 7 | 21 | 14 | 163614 | 0.62 |
| 8 | 24 | 16 | 208660 | 0.63 |

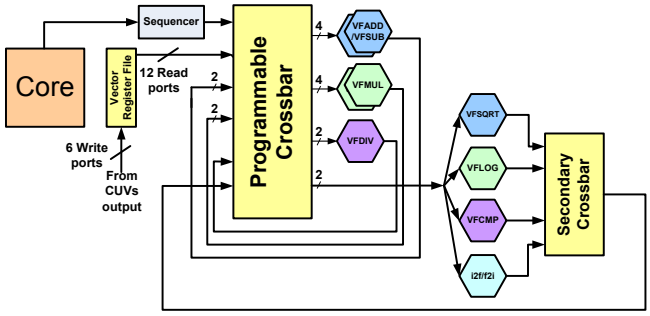


**Fig. 11. CVU-based design.**

*C.3. Sequencer design*

As discussed in Section III, sequencer has a RAM. The size of the RAM depends on the maximum height of a vector pattern and also the size of the crossbar. In our design an 18-input, 12-output crossbar has been used. This makes the number of bits needed for each scheduling step equal to (5*12) plus 2 bits for the secondary crossbar, in total of 62 bits. The largest pattern in our benchmarks needs 28 steps under the 2-2-1 CVU configuration. In this case, we picked a 32 x 64 bits RAM for the sequencer design. Table 5 shows the synthesis results of the sequencer using CACTI [18].

**Table 5. Sequencer synthesis results.**

| Sequencer RAM |
|---|
| W = 64 bit |
| Size = 32 words |
| Access time (ns): 0.598579 |
| Total dynamic read energy/access (nJ): 0.00953638 |
| Total leakage read/write power per bank (mW): 0.0519508 |
| Area (um$^2$): 7733.4 |

Each row in memory contains the configuration of the programmable switch at each cycle that custom vector instruction is being executed. Those information are generated statically by compiler. To generate these steps the compiler should consider the following parameters: 1) Pipeline depth and initiation interval (II) of the CVUs; 2) the ratio between vector length (VL) and the number of vector lanes (VL); 3) the number of available CVUs. If there is no CVU constraint or VL/NL < LPL (LPL refers to the longest length along the dependent operators that map to the same CVU), then on each cycle, a new input can be fed into CVU; otherwise a new input can only be applied after LPL cycles. If there are two CVUs with different latencies assigned to one scheduling step, then that CVU with lower latency will be adjusted with the longer latency CVU.

*D. Performance and Area Comparison*

We consider three reference points in the experiments:

(1) *Normal vector* (NV): Execution of the program using standard state-of-art vector instructions (Intel AVX).
(2) *Dedicated custom vector* (DCV): Execution of the program using dedicated ASIC-based customized vector instructions.
(3) *Composable custom vector* (CCV): Execution of the program using CVU-based customized vector instructions.

Fig. 12 shows the normalized speedup on each individual benchmark. Speedups have been normalized to the normal vector version. We make the following observations:

(i) Benchmarks such as *mri-gridding*, *mri-q* and *deblur* achieve a very large speedup. This is because the kernels in these benchmarks i.e., the critical functions have a structured pattern which is suitable for our architecture. Our compilation flow successfully captured such vectorizable patterns.

(ii) Benchmarks *denoise, registration* and *segmentation* achieve moderately good speedups. We find that most of the patterns in those benchmarks contain two or three parallel *add* or *mul* operations. Due to the CVU resource constraint we have, such parallelism cannot be fully supported in the CVU-based design.

(iii) The execution time difference between CCV and DCV is very small. Though the latter design does not need to consider the resource constraint. On average CCV is 5% slower than the DCV design on all the benchmarks, which further illustrate the efficiency of our CVU configuration selection strategy.

**Table 6. Area comparison between DCV and CCV.**

| CCV (2-2-1 configuration) ($um^2$) | | DCV ($um^2$) |
|---|---|---|
| *CVUs* | 340272 | **5574062** |
| *Crossbar* | 129992 | |
| *Sequencer* | 7733.4 | |
| *Total* | **477997.4** | |

Table 6 shows the area comparison between composable customized vector design and dedicated customized design. Table 6 includes the synthesized area results for each component in the CVU-based design and the corresponding dedicated ASICs which implements all the selected patterns. From the table we can see the area of dedicated ASIC is 11.6X larger than the proposed CVU-based approach.

In summary, we observe a significant speedup with customized vector support – DCV shows an average 1.43X speedup over the normal vectorized code and the proposed CCV design shows an average 1.4X speedup. Comparing to the dedicated ASIC design, the proposed CVU-based customization has a slight slow-down factor (< 5%) while shows much better area efficiency (~12X).

## VI. Conclusion

The customized vector domain is attracting increasing attention from both academia and industry. To provide efficient customization support, in this work we build an LLVM-based compilation flow to perform automatic customized vector ISA extension. A composable vector unit (CVU) is proposed to support a large number of customized vector instruction by allowing chaining among vector units. Our future direction is to extend the composable vector unit design to a multi-core environment such that the CVUs can be shared among multiple requesting cores.

## Acknowledgements

This work is partially supported by MARCO Gigascale Systems Research Center (GSRC), the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127, and the NSF grant CCF-0903541.

## REFERENCES

[1] K. Asanovic, et al., The T0 vector microprocessor. In *HOT Chips VII*, 1995.

[2] K. Atasu, L. Pozzi and P. Ienne, Automatic application-specific instruction-set extensions under micro-architectural constraints. In *Proc. DAC*, pp.256-261, 2003.

[3] C. Bienia, S. Kumar, J.P. Singh and K. Li, The PARSEC benchmark suite: Characterization and architectural implications, In *Proc. PACT*, pp. 72-81, 2008.

[4] M.O. Cheema and O. Hammami, Customized SIMD unit synthesis for system on programmable chip – A foundation for HW/SW partitioning with vectorization. In *Proc. ASP-DAC*, pp. 54-60, 2006.

[5] V.A. Chouliaras, et al., SystemC-defined SIMD instructions for high performance SoC architectures. In *Proc. ICECS*, pp. 1-4, 2006.

[6] J. Cong, Y. Fan, G. Han and Z. Zhang, Application-specific instruction generation for configurable processor architectures. In *Proc. FPGA*, pp. 183-189, 2004.

[7] J. Cong and W. Jiang, Pattern-based behavior synthesis for FPGA resource reduction. In *Proc. FPGA*, pp. 107-116, 2008.

[8] J. Cong, G. Reinman, A. Bui and V. Sarkar, Customizable domain-specific computing. In *IEEE Design & Test*, vol. 28, pp. 6-15, 2011.

[9] J. Cong, et al., High-level synthesis for FPGAs: From prototyping to deployment. In *IEEE TCAD*, vol. 30, pp. 473-491, 2011.

[10] K. Diefendorff, P.K. Dubey, R. Hochsprung and H. Scales, Altivec extension to powerpc accelerates media processing. In *IEEE Micro*, pp. 85–95, 2000.

[11] A.E. Eichenberger, P. Wu and K. O'Brien, Vectorization for SIMD architectures with alignment constraints. In *Proc. PLDI*, pp. 82–93, 2004.

[12] R. Espasa, et al., Tarantula: A vector extension to the alpha architecture. In *Proc.ISCA*, 2002.

[13] C. Kozyrakis and D. Patterson., Scalable vector processors for Embedded Systems. In *IEEE MICRO*, vol. 23, no. 6, pp. 36-45, 2003.

[14] P. Magnusson, et al., Simics: A full system simulation platform. In *IEEE Computer*, vol. 35, pp. 50-58, 2002.

[15] M. Martin, et al., Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. In *Computer Architecture News*, pp. 92-99, 2005.

[16] V. Raghunatha, A. Raghunathan, M.B. Srivastava and M.D. Ercegovac, High-level synthesis with SIMD units. In *Proc. VLSI Design*, pp. 407-413, 2002.

[17] P. Yiannacouras, J.G. Steffan and J. Rose, VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *Proc. CASES*, 2008.

[18] CACTI, http://www.hpl.hp.com/research/cacti/

[19] Convey system, http://www.conveycomputer.com/

[20] Intel AVX, http://software.intel.com/en-us/avx/

[21] LLVM Compiler Infrastructure, http://llvm.org/

[22] Omega library, http://www.cs.umd.edu/projects/omega/

[23] Parboil benchmark suite, http://impact.chrc.illinoise.edu/

[24] Synopsys design compiler, http://www.synopsys.com/

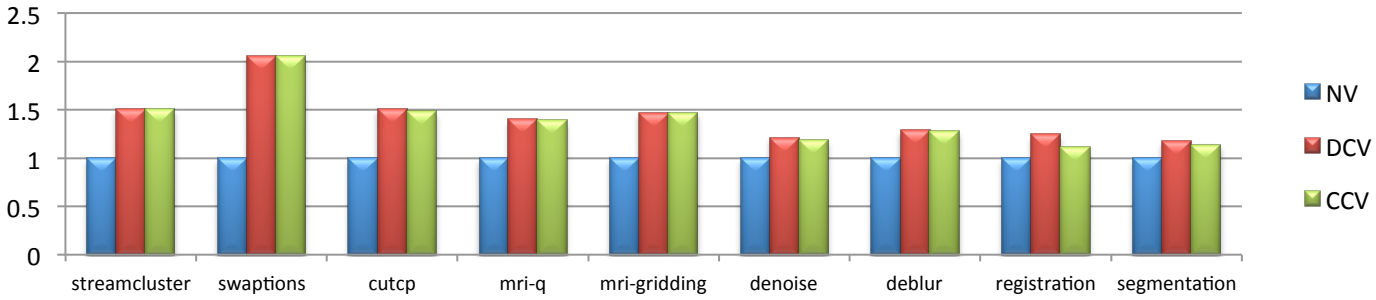


**Fig. 12. Normalized speedup.**