Accelerator Compiler for the VENICE Vector Processor

Zhiduo Liu
zhiduol@ece.ubc.ca
Dept. of ECE, UBC
Vancouver, Canada

Aaron Severance
aaronsev@ece.ubc.ca
Dept. of ECE, UBC
Vancouver, Canada

Satnam Singh
s.singh@acm.org
Google & Univ. of Birmingham
Mountain View, USA

Guy G.F. Lemieux
lemieux@ece.ubc.ca
Dept. of ECE, UBC
Vancouver, Canada

ABSTRACT

This paper describes the compiler design for VENICE, a new soft vector processor (SVP). The compiler is a new back-end target for Microsoft Accelerator, a high-level data parallel library for C++ and C#. This allows us to automatically compile high-level programs into VENICE assembly code, thus avoiding the process of writing assembly code used by previous SVPs. Experimental results show the compiler can generate scalable parallel code with execution times that are comparable to hand-written VENICE assembly code. On data-parallel applications, VENICE at 100MHz on an Altera DE3 platform runs at speeds comparable to one core of a 3.5GHz Intel Xeon W3690 processor, beating it in performance on four of six benchmarks by up to 3.2×.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Array and vector processors; C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded systems

General Terms

Design, Experimentation, Measurement, Performance

Keywords

vector, SIMD, soft processors, scratchpad memory, FPGA

1. INTRODUCTION

FPGAs offer low power operation and great performance potential through massive amounts parallelism. Harnessing the parallelism of FPGAs often requires custom datapath accelerators. C-to-hardware tools assist this process, but still require a lengthy place-and-route and timing closure process every time the software is changed. A soft vector processor such as VENICE provides an alternative that can accelerate a wide range of tasks that fit the SIMD programming model. To address the programmability issue of previous SVPs, this work presents a vectorizing compiler/back-end code generator based on Microsoft's Accelerator framework.

The VENICE architecture is chosen as a target SVP for this work. It is smaller and faster than all previously published SVPs. For applications that fit the SIMD programming model, VENICE is often fast enough that application-specific accelerators are not needed. For example, running at 100MHz, VENICE can beat the latest 3.5GHz Intel Xeon W3690 processor on data-parallel benchmarks. As well, VENICE achieves speedups up to 370× faster than a Nios II/f running at the same clock speed. The Accelerator compiler described in this paper achieves similar performance levels to manual coding efforts. Compared to past SVPs, the compiler greatly improves the usability of the system.

FPGA'12, February 22–24, 2012, Monterey, California, USA.

2. BACKGROUND AND RELATED WORK

Vector processing has been applied to scientific and engineering workloads for decades. It exploits the data-level parallelism readily available in applications by performing the same operation over all elements in a vector or matrix. It is also well-suited for image and multimedia processing.

Vectorizing Compilers. The VIRAM project [7] implemented a vectorizing compiler and achieved good results, auto-detecting over 90% of vector operations [8]. It is based on the PDGCS compiler for Cray supercomputers.

A common concern for soft vector processors is compiler support. Although based on VIRAM, early soft vector processors, VESPA [11, 12] and VIPERS [13, 14], required hand-written inline assembly code and GNU assembler (gasm) support. VESPA researchers investigated the autovectorizing capability of gcc, but have not yet used it successfully [12]. VEGAS [5] uses readable C macros to emit Nios custom instructions, but programmers must still track the eight vector address registers used as operands. This responsibility includes the traditional compiler roles of register allocation and register spilling.

The multi-core Intel SSE3 target of Accelerator [1] is a vector based target with shorter vector length. However, due to a load/store programming model, a fixed number of registers, and load balancing issues, the SSE3 target is entirely different in design than the VENICE target.

Intel's Array Building Blocks (ArBB) [2] is a system for exposing data parallelism. It combines Intel's Ct threaded

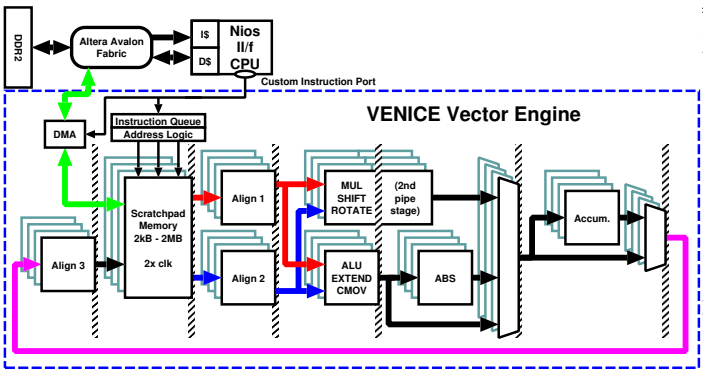

Figure 1: VENICE Architecture

programming model with RapidMind's object system, which provides, similar to Accelerator, C++ libraries for array types and array operations to express data-parallel computation. RapidMind could target CPUs, Cell, and x86 processors; ArBB appears to target only the latter.

VENICE Architecture. A block diagram of VENICE (Vector Extensions to NIOS Implemented Compactly and Elegantly) is shown in Figure 1. Based on VEGAS, VENICE makes the following improvements:

- Instruction-level support for 2D and 3D arrays. These avoid the need for VEGAS auto-increment modes.
- The vector address register file is removed. Hence, there is no need to track and spill the vector address registers. Instead, C pointers are directly used as operands to vector instructions.
- VENICE uses 3 alignment networks in the pipeline. This avoids the performance penalty with VEGAS when operands are misaligned.
- The shared multiplier/shift/rotate structure requires two cycles operational latency, allowing a general absolute value stage to be added after the integer ALU. This is followed by a general accumulator.

The native VENICE application programming interface (API) is similar to inline assembly in C. However, novel C macros simplify programming and make VENICE instructions look like C functions without any run time overhead, e.g. Figure 2 adds the scalar value 42 to a vector.

Each macro dispatches one or more vector assembly instructions to the vector engine. Depending upon the operation, these may be placed in the vector instruction queue, or the DMA transfer queue, or executed immediately.

The VENICE programming model uses a few basic steps: 1) allocate memory in scratchpad, 2) flush data from cache, 3) DMA transfer from main memory to scratchpad, 4) vector setup (e.g., set the vector length), 5) perform vector operations, 6) DMA transfer results from scratchpad to main memory, 7) deallocate memory in scratchpad.

The basic instruction format is `vector(VVWU, FUNC, VD, VA, VB)`. The `VVWU` specifier refers to 'vector-vector' operation (`VV`) on integer type data (`W`) that is unsigned (`U`). The vector-vector part can instead be scalar-vector (`SV`), where the first source operand is a scalar value provided by Nios. These may be combined with data sizes of bytes (`B`), halfwords (`H`) and words (`W`). A signed operation is designated by omitting the unsigned specifier (`U`).

```
#include "vector.h"
int main()
{
    int A[] = {1,2,3,4,5,6,7,8};
    const int data_len = sizeof(A);
    int *va = (int *) vector_malloc( data_len );
    vector_dma_to_vector( va, A, data_len );
    vector_wait_for_dma();
    vector_set_vl( data_len / sizeof(int) );
    vector( SVW, VADD, va, 42, va );  // vector add
    vector_instr_sync();
    vector_dma_to_host( A, va, data_len );
    vector_wait_for_dma();
    vector_free(); // deallocate scratchpad
}
```

Figure 2: VENICE API Adds Scalar to Vector

```
#include "Accelerator.h"
#include "VectorTarget.h"
using namespace ParallelArrays;
using namespace MicrosoftTargets;
int main()
{
    Target *tgtVector = CreateVectorTarget();
    int A[] = {1,2,3,4,5,6,7,8};
    IPA a = IPA( A, sizeof(A)/sizeof(int) );
    IPA d = a + 42;
    tgtVector->ToArray( d, A, sizeof(A)/sizeof(int) );
    tgtVector->Delete();
}
```

Figure 3: Accelerator Code Adds Scalar to Vector

The `vector_malloc(num_bytes)` call allocates a chunk of scratchpad memory. The `vector_free()` call frees the entire scratchpad; this reflects the common usage of the scratchpad as a temporary buffer. DMA transfers and instruction synchronization are handled by macros as well. In our experience, DMA transfers can be double-buffered to hide most of the memory latency.

Accelerator. The Accelerator system developed by Microsoft [1, 10] is a domain-specific language aimed at manipulating arrays with multiple back-end targets, including GPUs, multicore Intel CPUs, and VHDL [4]. Accelerator allows easy manipulation of arrays using a rich variety of element-wise operations. The restricted structure of Accelerator programs makes it easy to identify parallelism.

Accelerator data are declared and stored as Parallel Array (PA) objects. Accelerator does a lazy functional evaluation of operations with PA objects. That is, expressions and assignments involving PA objects are not evaluated instantly, but instead they are used to build up an expression tree. At the end of a series of operations, the PA `ToArray()` method must be called. This results in the expression tree being optimized, translated into native code using a JIT compilation process, and evaluated.

Figure 3 shows code to add the scalar value 42 to a vector in Accelerator. The `CreateVectorTarget()` function indicates that a subsequent `ToArray()` call will be evaluated on the vector processor. The `IPA` type represents an integer parallel array object. The `ToArray()` call triggers the compiler to generate VENICE-compliant code. Except for creating the proper target, the program is unaware of all hardware-related details, including whether a VENICE processor is being used or its size. To target a different device, one simply renames the `CreateXXXTarget()` function.

3. VENICE TARGET IMPLEMENTATION

The ability to manipulate arrays is intrinsic to both Accelerator and VENICE. In many cases, a direct translation

```
Front-end:
        input expression_graph;
        convert_to_IR();
        mark_and_add_intermediates();
        move_bounds_to_leaves();
Back-end:
        contant_folding_and_propagation();
        combine_operators();
        eval_ordering_and_ref_counting();
        buffer_counting();
        convert_to_LIR();
        calc_buffer_size();
        assign_buffers_to_input();
        allocate_and_init_memory();
loop:   transfer_data_to_scratchpad();
        set_vector_length();
        write_vector_instructions();
        transfer_result_to_host();
        if( !double_buffering_done ) goto loop;
        output VENICE_C_code;
```

Figure 4: Accelerator Compiler Flow

from Accelerator operators to VENICE instructions is possible. The compiler automatically breaks up large matrices into a series of smaller data transfers that fit in the scratchpad. Also, it uses double-buffering to hide memory latency.

For this work, we do not support JIT. Instead, we use Accelerator as a source-to-source compiler: it writes out another C program annotated with the VENICE APIs, which must be recompiled using gcc.

Figure 4 indicates the sequence of code optimizations and code transformations performed by the compiler. The front-end performs constant folding and common subexpression elimination to produce an optimized intermediate representation (IR). Then, the front-end analyzes all of the memory transforms and array accesses to produce index bounds for each leaf node (input array) in the computation.

Back-end Preliminaries. Next, target-specific optimizations are done before code generation. We found it beneficial to perform our own constant folding in the back-end in addition to existing front-end optimizations. Next, certain short sequences of operators are combined into a single compound VENICE operation, such as a multiply-add sequence or any add/subtract followed by absolute value.

Scratchpad Allocation. To load input data from main memory into the scratchpad, we need to allocate space in scratchpad memory first. The back-end treats the scratchpad as a pseudo-registerfile [6, 9]. This divides the scratchpad into as many equal-size registers (vector data buffers) as needed. However, several techniques are required to limit the number of registers to maximize their size.

To determine the size of these registers, the compiler first counts the total number of registers needed by the program. This is done by first determining an evaluation order for the subexpressions in each tree using a modified Sethi-Ullman algorithm [3]. To re-use registers, the back-end keeps a list of registers acting as input buffers for subsequent calculations, plus the number of remaining references to each of them. Whenever the reference count becomes zero, the register is no longer needed to hold an input array or an intermediate result, allowing the register to be re-used immediately. The total number of registers needed is the sum of registers used to hold leaf (input) data plus temporary intermediate data. After this step, a linear IR (LIR) is generated with references to precise register numbers.

One convenience feature in Accelerator is efficient handling of out-of-bounds array indices coming from memory

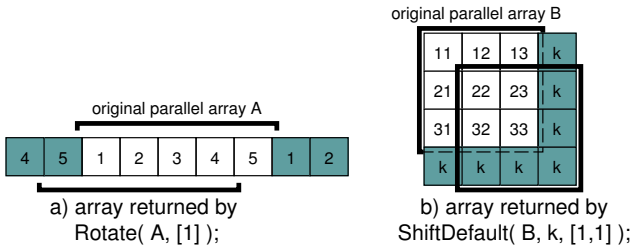

Figure 5: Memory Transform Examples

transform operations such as `Shift()` and `Rotate()`. In the front-end, Accelerator propagates the array bounds back to each leaf node, so the maximum extents are known. The back-end takes this additional information into account and allocates extra memory in the scratchpad for these cases.

All scratchpad memory is freed after each `ToArray()` call.

Data Initialization and Transfer. The initializing stage copies any user data to the output C file and prepares for memory transforms by padding input arrays with proper values for any out-of-bounds accesses.

Figure 5 demonstrates how input data padding is done. Part a) shows a rotation performed on a 1D array. The original array is white, with the out-of-bounds elements shaded. In this case, the last element 5 appears padded before first element, while the first element 1 appears padded after the last element. The new array formed by `Rotate()` is indicated by the bold black bar. Part b) shows a shift on a 2D array, up and to the left at the same time. Values past the bounds are initialized with the specified default value of k. The new array formed by `ShiftDefault()` is highlighted by a bold black box.

DMA transfer instructions are generated after memory allocation and data initialization. In the case where the full array is large, or doesn't entirely fit into scratchpad, the compiler generates code to move data in a double-buffered fashion by pre-fetching. This allows DMA transactions and vector computation to overlap. Overlapping the two can almost completely hide the overhead of the data transfer.

Generation of Vector Instructions. There is nearly a direct mapping of Accelerator operators to VENICE instructions for basic element-wise operations. In a few cases, we have prewritten library code to support Accelerator operators that are not directly supported by VENICE, such as divide, modulo and power.

For memory transforms on PA objects, we discussed in the previous subsection that we handle such operations by initializing the input data with a padded region outside of the normal array bounds. We refer to the examples in Figure 5 again here to demonstrate how memory transforms are executed. With all data properly initialized, extracting partial data from a 1D array is simply done by adding an offset to the starting address in the scratchpad memory. For 2D arrays, the VENICE row stride amounts can be adjusted to step over any padding elements added at both ends.

Implementation Limitations. VENICE does not support floating-point operations, so we are unable to support float, double and quad-float types in Accelerator. The Boolean type uses 32b integers. Most of the Accelerator APIs have been implemented in the VENICE back-end; a few were omitted due to time constraints.

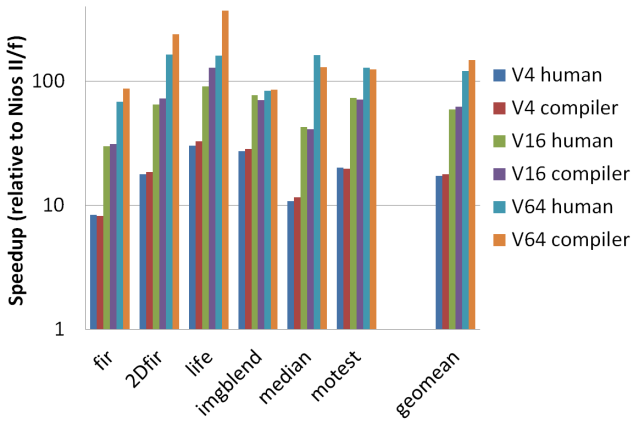

Figure 6: Compiler Speedups

CPU	fir	2Dfir	life	imgblend	median	motest
Xeon W3690	0.07	0.44	0.53	0.12	9.97	0.24
VENICE	0.07	0.29	0.23	0.33	3.11	0.22
Speedup	1.0	1.5	2.3	0.4	3.2	1.1

Table 1: Runtime (seconds) and Speedup

4. RESULTS

Soft processor results were run on an Altera DE3-150 with one DDR2-800 SODIMM. Different VENICE instances use 4, 16, and 64 parallel lanes of 32b ALUs, called V4, V16, and V64, respectively. All processors run at 100MHz, with the DDR2 memory at half rate; this allows easy estimation of runtime using scaled clock rates up to 200MHz.

A set of six benchmarks are used to measure the effectiveness of the compiler at scaling to large size SVPs. All benchmarks use integers because Accelerator does not support byte or short data types. However, smaller data types allow greater performance with VENICE because each 32b ALU can be fractured into four 8b or two 16b ALUs.

Speedups over serial Nios II/f C code for both human and compiler-generated parallel code are shown in Figure 6. The compiler outperforms the human in 11 of the 18 cases; when the human wins, it is only by a small margin, but the compiler often wins by a much larger margin. This is because the compiler puts more effort into the process than a human: 1) it fully unrolls inner loops to reduce overhead; 2) it carefully calculates the maximum buffer size that fits into the scratchpad, rather than conservatively rounding down or guessing; 3) it double-buffers all data transfers; 4) it inlines all function calls. The fastest, life, achieves 370× speedup compared to a Nios II/f. However, humans can sometimes do far better than the compiler; the graph does not show our human-written motion estimation result which is another 1.5× faster because it uses the VENICE accumulator in a way that cannot be expressed in the Accelerator language. Finally, we note that imgblend is memory bandwidth limited at V16, so it does not benefit from more ALUs at V64.

In Table 1, we compare the VENICE compiler (not human) results to a single-core 3.5GHz Intel Xeon W3690 processor compiled with Visual Studio 2010 with -O2. We ran each benchmark 1000 times and measured total runtime. Across the 6 benchmarks, Intel beats VENICE only on imgblend, which is memory bandwidth limited.

5. CONCLUSIONS

This work has shown that compiler-generated results with a soft vector processor can achieve significant speedups on data parallel workloads. Speedups up to 370× versus a Nios II/f, and speedups up to 3.2× versus a 3.5GHz Intel Xeon W3690 are demonstrated. Furthermore, compiler-generated results are comparable to human-coded results.

Currently, Accelerator and VENICE have limited data type support. Accelerator should add support for bitwise operations, plus byte and halfword data types. As well, VENICE should add floating-point data types. In our backend, some Accelerator APIs are not yet implemented.

6. ACKNOWLEDGMENTS

We thank NSERC for funding, Altera for hardware donations, and the Microsoft Accelerator group for their assistance during this project.

7. REFERENCES

[1] Accelerator. http://research.microsoft.com/en-us/projects/accelerator.

[2] Sophisticated library for vector parallelism. http://software.intel.com/en-us/articles/intel-array-building-blocks/.

[3] A. Appel and K. J. Supowit. Generalizations of the Sethi-Ullman algorithm for register allocation. *Software – Practice and Experience*, 17:417–421, 1987.

[4] B. Bond, K. Hammil, L. Litchev, and S. Singh. FPGA circuit synthesis of accelerator data-parallel programs. In *FCCM*, pages 167–170, Charlotte, North Carolina, USA, 2010.

[5] C. Chou, A. Severance, A. Brant, Z. Liu, S. Sant, and G. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *FPGA*, pages 15–24, Monterey, California, USA, 2011.

[6] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *PACT*, pages 321–330, Seoul, Korea, 2006.

[7] C. Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, May 2002. Technical Report UCB-CSD-02-1183.

[8] C. E. Kozyrakis and D. A. Patterson. Scalable vector processors for embedded systems. *IEEE Micro*, 23(6):36–45, 2003.

[9] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *PACT*, pages 329–338, Sydney, Australia, 2005.

[10] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *ASPLOS*, pages 325–355, San Jose, California, USA, 2006.

[11] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *CASES*, pages 61–70. ACM, 2008.

[12] P. Yiannacouras, J. G. Steffan, and J. Rose. Data parallel FPGA workloads: Software versus hardware. In *FPL*, pages 51–58, Progue, Czech Republic, 2009.

[13] J. Yu, C. Eagleston, C. Chou, M. Perreault, and G. Lemieux. Vector processing as a soft processor accelerator. *ACM TRETS*, 2(2):1–34, 2009.

[14] J. Yu, G. Lemieux, and C. Eagleston. Vector processing as a soft-core CPU accelerator. In *FPGA*, pages 222–232, Monterey, California, USA, 2008.