SIMD Programming in GNU Radio: Maintainable and User-Friendly Algorithm Optimization with VOLK

Thomas W. Rondeau
GNU Radio
Email: tom@trondeau.com

Nicholas McCarthy
University of Maryland
Email: namccart@gmail.com

Timothy O'Shea
University of Maryland
Email: oshea@umd.edu

***Abstract*—We present VOLK as an easy to use single-instruction multiple-data (SIMD) math library and as a structure for open-source development of SIMD code. VOLK is the Vector-Optimized Library of Kernels and provides an abstraction layer for hardware-specific SIMD operations. The abstraction layer aids SIMD code construction, enforces a common interface for library development, and complements data-streaming computation models common in software radio development.**

We discuss how VOLK is used in GNU Radio to provide significant speed-up to signal processing blocks, and we survey current programming models for incorporating VOLK within those blocks. Results for different blocks are shown.

I. Introduction

I wanna go fast!

—Ricky Bobby

While general purpose processors (GPP) have included vector instruction capabilities for a number of years, their use is still esoteric and difficult to generalize. We present here the Vector-Optimized Library of Kernels (VOLK) project from GNU Radio that provides a simple to use, extensible, and architecture-independent programming tool to enable vectorized mathematical operations.

These GPP vector extensions are known as SIMD for single-instruction, multiple-data. SIMD extensions generally involve a specialezed set of wide registers and an extended set of instructions to manipulate data within these registers. Typical implementations have used 128-bit registers, although x86 processors first used 64-bit registers, and current versions use 256. A 128-bit SIMD register can hold four single-precision floats, eight shorts, or sixteen bytes. Families of processors from the same vendor and processor families from different vendors can feature different hardware SIMD implementations, and corresponding instruction set extensions can differ between generations or from one company to another.

Although compilers like GCC [5] and Intel's ICC try to vectorize when they can, it is not an easily solved problem, especially for more complicated math functions. Further, access to SIMD registers generally requires low-level programming tools, and assembly language is still often used. For many processors, there exist *i*ntrinsics, which essentially expose individual assembly instructions to C language compilers. But there exists a tension between the universality of C code and hardware optimization at the SIMD level. With libraries, instruction sets, and architectures differing on a per-processor basis, it becomes difficult to program and manage code written to access SIMD instructions, especially when portable code is the aim. Even acknowledging differences on a per-processor basis does not go far enough. SIMD code enables processes to run faster or more efficiently, but code optimization is a run-time problem. The same processor running the same code within different memory structures or at different loads can show different efficiency properties.

The VOLK library is an abstraction designed to fix these problems. It provides a platform-agnostic interface called a *kernel* for each conceptual execution unit subject to SIMD vectorization. Underneath, VOLK has a set of *proto-kernels* designed for particular platforms, SIMD architecture versions, or run-time conditions. The VOLK library compiles all possible proto-kernels supported by the compiler toolchain. At run-time, during the first call to an abstract kernel, VOLK resolves the kernel to a specific proto-kernel. VOLK tests the run-time platform for its capabilities to ensure the resolved proto-kernel will run correctly and employs a dynamic rank-ordering to select the best possible proto-kernel. The process of proto-kernel resolution is critical to the functionality of VOLK and will feature in following sections.

This paper first describes SIMD programming in more detail to expose both benefits and challenges. We discuss the design of VOLK, examining platform independence, extensibility, and optimized kernel selection. We study how best to integrate VOLK and GNU Radio, quantizing observed benefits. The paper will conclude with a discussion on future improvements to VOLK, both speculative and in-production.

II. Basics of SIMD

SIMD is a common computer instruction architecture feature provided to enable efficient vector operations. A *vector operation* is the simultaneous application of a single operation to each element in a vector of data. In principle, using vector operations results in significant speedups versus using equivalent scalar operations: performing a vector operation on a four-long vector should take one-quarter as much time and energy as performing four different scalar operations to achieve the same purpose. The x86 (Intel and AMD), PowerPC, and ARM GPP architectures all offer SIMD variants. When Intel released

MMX (multimedia extension), its initial GPP SIMD architecture feature, the company signaled an intention to target SIMD toward multimedia and graphics processing applications [4]. Such applications often translate easily to signal processing applications. Both application classes feature high-rate data streaming.

We provide here a quick example using SSE to perform a vector multiplication with two input vectors of floating point values. In VOLK, this is the proto-kernel *volk_32f_x2_multiply_32f_a_sse*.

```
1  static inline void
   volk_32f_x2_multiply_32f_a_sse(
     float* cVector, const float* aVector,
     const float* bVector, unsigned int num_points)
   {
6    unsigned int number = 0;
     const unsigned int quarterPoints = num_points/4;
     float* cPtr = cVector;
     const float* aPtr = aVector;
     const float* bPtr=  bVector;
11   __m128 aVal, bVal, cVal;
     for(;number < quarterPoints; number++){
       aVal = _mm_load_ps(aPtr);
       bVal = _mm_load_ps(bPtr);
       cVal = _mm_mul_ps(aVal, bVal);
16     _mm_store_ps(cPtr,cVal);
       aPtr += 4; bPtr += 4; cPtr += 4;
     }

     number = quarterPoints * 4;
21   for(;number < num_points; number++){
       *cPtr++ = (*aPtr++) * (*bPtr++);
     }
   }
```

In this code, the vectors **aVector** and **bVector** each contain **num_points** floats. They are multiplied together and produce output values in **cVector**. In each round of the for loop, four floats from each of the input vectors are loaded into SIMD registers and then multiplied together. The *_mm_mul_ps* performs the four floating point multiplications at the same time, and each of the vector pointers then increments by four float items, or sixteen bytes, so that the next round uses four new float pairs. In total, the loop runs four times fewer than an equivalent, non-vectorized loop.

A final for loop at the end cleans up any remaining data items if **num_points** is not a multiple of four, so this loop multiplies one, two, or three extra float pairs to ensure the entire vector is processed.

This example showcases the use of C intrinsics to access SIMD functionality. Most VOLK proto-kernels feature intrinsics as opposed to in-line assembly for two reasons. The first reason is simplicity and decreased development time. Intrinsics leave register mapping to the compiler and reduce the set of required technologies for SIMD programming. The second reason is portable efficiency. Not all code optimization challenges are equal to the compiler. C compilers often do not optimally leverage SIMD architectures, as our results for specific GNU Radio blocks show. They can, on the other hand, handle register mapping, software pipelining, and loop unrolling quite well. Performance benchmarks from the Spiral Viterbi project demonstrate this phenomenon. Across a range

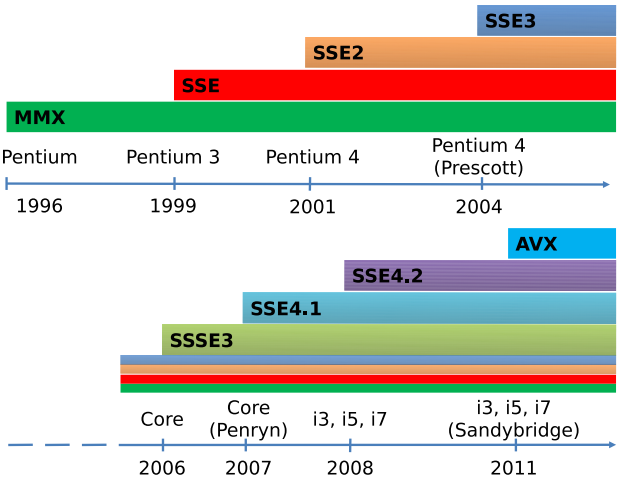

Fig. 1. Chart of when Intel introduced its various versions of SIMD architectures.

of Viterbi Algorithm implementations, code auto-generated using SSE intrinsics compares at least comparably and in most cases better than Phil Karn's hand-optimized mixture of intrinsics and assembly [1]. Compilers can solve these issues differently for different hardware. Hand-optimized assembly is static from machine to machine.

SIMD programming can provide dramatic performance benefits, but developers often choose not to support SIMD enhancements. Practically speaking, it can be difficult to code using SIMD extensions where the project in question assumes a team of developers and wants to produce cross-platform compatible source code. Intrinsics help abstract away some machine-specific differences, but intrinsics do not abstract to the point where two code contributors running computers from different decades with different compiler versions can easily cooperate.

Vendors use architectures with critical differences. Intel has a range of architecture families ranging from MMX to SSE and AVX. AMD produced chips with a distinct architecture called 3DNow!, but has since dropped those extensions in an effort to harmonize SIMD across x86. PowerPC has the Altivec instruction set, and ARM uses the NEON extensions.

Within one vendor's product line, hardware and software evolves. Figure 1 depicts various generations of Intel's SIMD architecture family. Generally, all newer processors still contain older architectures for backwards compatibility, but code written for newer architectures will not work on older processors. Barriers to compatibility can easily relegate SIMD programming to projects for which efficiency is paramount. Nor is compatibility the only barrier to using SIMD for software projects.

SIMD code can place demands for memory management techniques on the code surrounding it. Common to all the SIMD architectures previously mentioned are some restrictions on loading (putting samples into the registers) and storing (moving samples out of the registers). Many architectures,

Intel's x86 among them, have specific memory alignment requirements for vector loads and vector stores. For SSE architectures, true vector loads and stores require 16-byte alignment, and unaligned data management requires calls to substantially less efficient load and store instructions. Attempting to use an aligned load with unaligned data can result in errors and unexpected behavior. A data-streaming programming model must assume many processes will be bound by memory bandwidth and not computation-bound. In order really to expose the efficiencies of SIMD architectures to an SDR programming model, the model itself must take the responsibility to coordinate memory alignment.

Various attempts have been made to solve the problem of generalizing SIMD programming. Intel's compiler (ICC) and the Intel Performance Primitives (IPP) and Math Kernel Library (MKL) do a good job of hiding the SIMD optimizations from the users, but these approaches only target Intel processors. One of the more accessible and successful tools in this area is ORC (the OIL runtime compiler) [3]. VOLK is actually capable of using ORC to create proto-kernels, but that is beyond the scope of this paper.

III. VOLK Programming Model

A. Coding

In §I, we describe VOLK both as a library of kernels and as an abstraction layer designed to solve problems inherent to open source development with SIMD code. The description of SIMD architecture in §II identifies code compatibility as one such problem. Open source projects must support the hardware their communities use, and that hardware varies in age and ability. Beyond hardware compatibility, a successful abstraction layer must enable meaningful collaboration. Open source development relies heavily on integrating various solutions from users working on different aspects of multiple problems. Communities can fracture, and competition can yield benefits, but freedom in software means the ability to cull from the best available ideas. In this section, we tour the particulars of the VOLK abstraction layer with an eye toward these stated goals.

VOLK defines three different conceptual object types: kernels, archs, and machines. In §I, we introduced the kernel and the proto-kernel. A kernel corresponds to a file containing source code for its proto-kernels. The naming of that file follows a structure enforced by regular expressions in the VOLK build system:

volk_*input-fingerprint*_*function-name*
_*output-fingerprint*.h.

The input-fingerprint is an underscore-separated list of argument types, $[\mathrm{s}]BT$, and argument counts, $\mathrm{x}N$. For an argument type, the optional letter s identifies a scalar argument. Bitcount B is one of $64, 32, 16,$ or 8, and indicates item size for the corresponding argument. Type, T, is one of i for fixed-point integers, u for unsigned fixed-point integers, f for floats, ic for interleaved fixed-point complexes, or fc for interleaved float complexes. An argument count, $\mathrm{x}N$ for $N > 1$ an integer, always follows an argument type descriptor when the kernel takes N arguments of the same type. If the kernel has an input vector argument and operates in place, the coder must list that argument in the input-fingerprint, and scalar arguments follow vector arguments in order. The function-name is any string describing kernel functionality and must not contain a substring matching the argument type template. The output-fingerprint follows the same structure as the input-fingerprint but describes output arguments rather than input arguments.

A VOLK kernel can have any number of proto-kernels. A proto-kernel is a static inline function. Declaration and definition of the proto-kernels allow for a standard, portable interface to the individual proto-kernels for use on systems without C++ compilation and shared objects. Proto-kernels follow the naming structure

volk_*input-fingerprint_function-name*
_*output-fingerprint_alignment-flag_tag*,

where tag is the descriptor for the particular proto-kernel. The tag need not refer to any SIMD architecture, though often it does. Each of a kernel's proto-kernels must have a unique tag. The alignmentflag a is for a kernel expecting boundary-alignment of u for a kernel with no alignment expectations. An alignment flag is required for all but two special-case proto-kernels: one with tag *generic* and another with tag *dispather*. A proto-kernel argument list positions the output arguments described in the output-fingerprint first, followed by those arguments described in the input-fingerprint. A final argument, unsigned int num_points, closes the list and holds the value of the number of points in the longest input vector. Within the function definition, the coder is free to code. VOLK developers have not yet found it necessary to write proto-kernels incompatible with a ANSI C compilation. Since C++ is probably a more common environment for VOLK code, "volk_complex.h" provides an abstraction bridging the chasm between float complex types in C and C++.

Surrounding each proto-kernel is an #ifdef compiler directive with argument

$$\text{LV\_HAVE\_}ARCH1 \;\&\&\; \text{LV\_HAVE\_}ARCH2 \;\&\&\; \cdots \;\&\&\; \text{LV\_HAVE\_}ARCHN, \tag{1}$$

where $ARCHJ$ is a VOLK arch object for each J.

An *arch* is an abstraction for any hardware-specific property. To GCC, an arch corresponds roughly to one or several *-m* flags such as *-msse3*. To CPU hardware, an arch describes physical silicon or firmware attributes enabling, for instance, the CPU to execute machine code for a particular assembly instruction.

An arch is defined by its entry in an XML table, "archs.xml." Each $ARCH$ in the table corresponds to a VOLK macro LV_HAVE_$ARCH$. The XML table indexes information CMake [2] can use to determine whether the project compiler can generate code specific to $ARCH$. Each arch entry may contain flag subentries for the project compiler, and the coder can assume all compiler flags are set within the directive #ifdef LV_HAVE_$ARCH$. For example, within #ifdef LV_HAVE_SSE3 a coder may assume the flag *-msse3* is set and include "pmmintrin.h."

The macro LV_HAVE_GENERIC is always set to 1. The coder uses this macro unaccompanied by any other to denote that the enclosed code contains no dependence on a VOLK arch. A *generic proto-kernel* is one written inside the directive #ifdef LV_HAVE_GENERIC. Intuitively, a generic proto-kernel is a universally available fall-back proto-kernel. Each kernel must contain one (or more!) generic proto-kernels. In addition, for reasons made clear in §III-C, one unaligned proto-kernel must have the tag "generic." In defining the concept of the generic proto-kernel, it is important to distinguish source-code universality from compiled-code universality. Within VOLK, a generic proto-kernel can compile to assembly code that is very much not universal. In fact, the compiler can *promote* any proto-kernel using compiler flags for VOLK archs unrequired by the source code. To address how VOLK handles promotion requires a discussion of VOLK machines and library compilation.

B. Compiling

A VOLK *machine* is an abstraction for a processor. To CMake, a machine corresponds roughly to a list of compiler flags used to compile each VOLK kernel. To CPU hardware, a machine is a full description of the various architectural and software attributes required for a processor to run binaries within a shared object. The machine set is defined in "machines.xml," and from this perspective, a machine essentially boils down to a list of archs. How VOLK compiles comes down to resolving any tension between what the system and compiler can do on one hand and what each machine definition asks the system and compiler to do on the other hand.

At compile time, CMake tests each VOLK arch in "archs.xml" by compiling a dummy program using the flag selection in the XML entry for that arch. VOLK exposes the arch list to CMake via a Python utility which parses the XML and holds intermediate Python class representations of the various archs. If a follow-on test is required, VOLK runs the follow-on test and overrules the result of the first test. For example, current versions of GCC can handle both the $-m64$ and the $-m32$ flags separately. A Linux install might include development libraries for one or the other flag, not both. A follow-on test selects one or the other of the flags according to the operating system. (The user can override this behavior and cross-compile but must install both development libraries beforehand.) A list, $available\_archs$, of available archs results. CMake then tests each VOLK machine. VOLK exposes the machine list to CMake via a Python utility which parses the XML and holds intermediate Python class representations of the various machines. Each machine is a list, $machine\_archs$, of archs. If

$$available\_archs \cap machine\_archs = machine\_archs,$$

then the compiler has what it needs to compile a library for the machine. A list $available\_machines$, of available machines results.

A VOLK Python utility parses each kernel definition file and builds a dictionary of intermediate Python kernel class representations. In particular, a kernel class representation holds onto arch dependency information and identification tags for each of the proto-kernels. The arch dependency list $\_tagdeps$ for a given kernel is

$$(ARCH1, ARCH2, \cdots, ARCHN), \tag{2}$$

using definitions from (1) in §III-A. VOLK auto-generates C source code for each kernel using C code templates and a Python utility. Each of a kernel's proto-kernels shares an argument and return value fingerprint, and VOLK typedefs a kernel function pointer using this fingerprint. VOLK generates in "volk.h" three distinct kernel object interface declarations with this fingerprint type for every kernel. Specifically, for each kernel "volk_*kernelname*.h," VOLK furnishes volk_*kernelname*_a, volk_*kernelname*_u and volk_*kernelname* as interfaces to that kernel. Volk filters definitions for the interfaces through a call to *get_machine()*, and this filtration is the primary subject of §III-C. The return value of *get_machine()* is a volk_machine struct. The volk_machine struct definition itself is templated per-kernel such that the resulting struct contains, among other information, each kernel's $impl\_names$, $impl\_alignment$, n_impls, and $impls$. The string array $impl\_names$ holds the tag names for the kernel's proto-kernels. The function pointer array $impls$ holds pointers to the proto-kernels themselves, and n_impls holds the number of proto-kernels in the array $impls$. The boolean array $impl\_alignment$ indicates whether the proto-kernels in $impls$ make alignment assumptions. The index of tag t in the array $impl\_names$ corresponds to the index of the proto-kernel with tag t in the array $impls$ and the index of the alignment requirements for that proto-kernel, allowing for lookups. Of course, n_impls, $impl\_names$, $impl\_alignment$ and $impls$ all differ on a per-machine basis.

VOLK localizes most machine-specific code variation in templated code generated from the file "volk_machine_xxx.tmpl.c." By restricting this code generation appropriately, CMake ensures VOLK will compile code only for those machines the compiler can handle. (VOLK generates some other machine-specific code from "volk_machines.tmpl.h" and "volk_machines.tmpl.c." The resulting code relates to code generated from "volk_machine_xxx.tmpl.c.") CMake loops over only $available\_machines$ when generating code from this file and produces a file defining a $volk\_machine$ struct with each iteration. While generating code from this file, a Python utility function selectively sets #define LV_HAVE_$ARCH$ for each VOLK arch $ARCH$ in accordance with the current machine, and any machine in $available\_machines$ will set only archs available to the compiler. Code generated from this template forms a sole point of contact between user-generated code in kernel definition files and VOLK shared objects. CMake generates one shared object per machine, applying the compiler flags corresponding to that machine. The objects linked into each library are identical with the exception of the objects generated from the "volk_machine_xxx.tmpl.c" template. Each library links only one unique such object.

The compiler flags applied to generate any one shared object apply equally to all code generated for that shared object. If the machine allows for SSE3 instructions but a proto-kernel requires only SSE2 instructions, the compiler has free reign to generate assembly translating from the proto-kernel's original SSE2 instructions to SSE3 instructions. This promotion effect observed in §III-A is crucial to VOLK taking full advantage of optimizing compilers. Developers write VOLK proto-kernels in order to gain efficiency. We have described compilation such that for each kernel, there are multiple proto-kernel definitions competing for access to kernel interface function pointer objects. VOLK must allow this competition to remain fair. Within the restrictions imposed by a machine definition, the compiler should do everything it can to make generic versions as efficient as possible. VOLK establishes a situation such that hand-coding complements compiler optimization rather than substituting for it.

C. Runtime

VOLK runtime behavior defines the competition for the kernel function pointers among the proto-kernels, and this competition establishes a situation such that differently hand-coded and compiled proto-kernels complement one another rather than displace one another. In §III-B, we mention the kernel definitions filter through a call to $get\_machine()$. In fact, they filter through two calls: $get\_machine()$ and $volk\_rank\_archs()$. The first call implements runtime selection from among the libraries generated at compile time. The second implements runtime selection from among a kernel's proto-kernel set. We examine the two separately.

The ordering of the archs in "archs.xml" is not arbitrary. CMake defines a macro, LV_$ARCH$ for each VOLK $ARCH$ according to the order of $ARCH$'s appearance in the file.

Each arch contains a *check* entry referring to a runtime method determining whether a processor can run that arch. The runtime checks generated from "volk_cpu.tmpl.c"' use check entry definitions specifying calls to utilities like **cpuid** or tests for register bits set in a standard way across architecture families to indicate functionality (x86 chips from AMD and Intel implement such an identification protocol).

In principle, the set of machines maps one-to-one into ${\mathbb{Z}_2}^n$ where n is the number of archs: each machine either has or does not have each arch. We assign to each machine a unique integer via a simple inclusion ϕ from ${\mathbb{Z}_2}^n$ to $\mathbb{Z}$. The LV_$ARCH^{th}$ bit in $\phi(m)$ is 1 for a machine m if and only if m has the arch $ARCH$. The ordering on ints now reflects our intentional ordering of archs in "archs.xml." Machine m outranks machine l if the most valuable arch for which the two machines differ belongs to machine m and not l (this is known as *dictionary order*).

With the first call to $get\_machine()$, VOLK performs the appropriate runtime check for all archs and calls $volk\_get\_lvarch()$ to find $\phi(I)$ where I is the runtime machine. VOLK then defines $\hat{m} \equiv \arg\max_T(\phi(m))$ where

$$T = \{m \in available\_machines : \tilde{}\phi(\mathtt{I})\&\phi(m) = 0\}.$$

(The symbols ˜ and & take their usual meanings as bitwise operators.) To express this formula simply, VOLK will not select a machine if the hardware cannot run it, but the best ranking machine wins out otherwise. The initial call to $get\_machine()$ sets a pointer to $\hat{m}$, but subsequent calls simply return this pointer.

To resolve the kernel interfaces volk_*kernelname*_a and volk_*kernelname*_u, VOLK determines the best available proto-kernels among those in $\hat{m}$ via a call to $volk\_rank\_archs()$. This function expresses a default behavior and an ability to override that behavior via settings in a preferences file. The default behavior plays out much like the initial call to $get\_machine()$. The $volk\_machine$ struct at address $\hat{m}$ has an array of ints, $arch\_defs$, populated with $\phi(p_i)$ where $0 \leq i \leq n\_impls$ and p_i is the sub-machine corresponding to the proto-kernel identified by the tag $impl\_names[i]$. A *sub-machine*, like a machine, is a list of archs. Unlike a machine, a sub-machine does not characterize a processor. A sub-machine characterizes hardware requirements for a proto-kernel. The sub-machine corresponding to a proto-kernel is defined to be the list $\_tagdeps$: (2), §III-B. VOLK defines $\hat{p} \equiv \arg\max_S(\phi(p_i))$ where

$$S = \{i \in [0, n\_impls] : \tilde{}\phi(\hat{\mathtt{m}})\&\phi(p_i) = 0\}.$$

The initial call to the kernel sets volk_*kernelname*_a to $\hat{p}$, and subsequent calls return this pointer.

To define volk_*kernelname*_u, VOLK further filters the set S to include only aligned proto-kernel indices. If

$$S(u) = \{i \in S : impl\_alignment(i) = false, \}$$

then for $\hat{p}(u) = \arg\max_{S(u)}(\phi(p_i))$, the initial call to the kernel sets volk_*kernelname*_u to $\hat{p}(u)$.

Code specifying the override behavior for $volk\_rank\_archs()$ resides in "volk_prefs.c" and "volk_prefs.h" and relies on setting in a preferences file. What populates this file? In principal, anything can populate the file, making VOLK runtime behavior user-accessible. In practice, VOLK offers a utility, **volk_profile**, for assessing runtime-accessible proto-kernels. This utility shares a code base with a **test_all**, a quality assurance utility. Both utilities rely on code auto-generated following the kernel naming structure. The files "qa_utils.h" and "qa_utils.c" determing auto-generation behavior. The header file defines three macros: VOLK_RUN_TESTS, VOLK_PROFILE, and VOLK_PUPPET_PROFILE. Essentially, VOLK_RUN_TESTS loads arrays from the input-fingerprint with random data, runs the kernel, and tests data in the output arrays against results from the proto-kernel with tag "generic." VOLK_PROFILE times each runtime-accessible proto-kernel, writing results to the preferences file.

The macros support only a limited number of fingerprint variations: fingerprints with a combined number of up to 4 vector arguments and up to only one scalar argument (a float or a float complex). All vector arguments must have the same length. The user can control a tolerance with which to

compare results, the value of the scalar used in the function call, the length of the vector arguments, and the number of iterations to run. In order for VOLK to test a user-defined kernel, the user must add a call to the VOLK_RUN_TESTS macro within the file "testqa.cc." In order for VOLK to profile a user-defined kernel, the user must add a call to the VOLK_PROFILE or VOLK_PUPPET_PROFILE macro within the file "volk_profile.cc."

The macro fingerprint restrictions necessitate the VOLK_PUPPET_PROFILE macro and the concept of puppet kernels. VOLK_PUPPET_PROFILE works identically to VOLK_PROFILE, with one exception. VOLK_PROFILE takes as an argument the name of a kernel to be profiled, and profiling results apply to that kernel. VOLK_PUPPET_PROFILE takes two name arguments: a kernel to be profiled (the *puppet*) and a kernel to which the profiling results apply (the *puppet master*). When a coder writes a kernel that does not conform to the fingerprint restrictions, the coder can use puppets to access equivalent auto-generated functionality. For instance, if a coder writes a kernel with vector arguments but also wants to pass a scalar value by pointer in order to recover that value on completion of the kernel call, the coder has broken the fingerprint assumptions. The auto-generator will interpret any pointer argument as a vector argument, and all vector arguments must have the same length. The coder writes a puppet variant for the kernel. A *puppet variant* is a kernel with a compliant fingerprint such that puppet proto-kernels call corresponding puppet master proto-kernel's directly. The coder manages delinquent arguments manually such that running the puppet proto-kernels under VOLK_RUN_TESTS provides meaningful results. The coder can add a call to the puppet as a replacement for a call to the puppet master in "testqa.cc," but in order for the runtime selection mechanism to take note of profiling results for a puppet, the coder needs to specify the puppet/puppet master relationship explicitly by a call to VOLK_PUPPET_PROFILE.

The **volk_profile** utility harmonizes competition among competing proto-kernels. Code written meticulously to avoid cache misses for a particular, older-generation chip can co-exist with code written quickly to leverage new hardware instructions. If yesterday, code written to leverage SIMD instructions explicitly ran faster than code written generically, but today that same generic code runs faster due to compiler advancements, nothing is lost. The **volk_profile** utility is a fair judge, and unused proto-kernels can lie around like unexpressed genes, waiting for their day in the sun.

We have described resolution for only two of the three interfaces provided for each VOLK kernel. In addition to volk_*kernelname*_a and volk_*kernelname*_u, VOLK also provides volk_*kernelname*. This final interface uses the VOLK *dispatcher* to arbitrate between volk_*kernelname*_a and volk_*kernelname*_u at runtime. By default, VOLK auto-generates a proto-kernel with tag "dispatcher." The dispatcher inserts a check before each kernel execution and determines whether the input and output buffers handed to the kernel will allow for execution of the aligned interface. If not, the dispatcher executes the unaligned interface. The coder can override the default behavior of the dispatcher by defining a proto-kernel within the directive #ifdef LV_HAVE_DISPATCHER using the tag "dispatcher." Within this directive the coder can use volk_*kernelname*_a and volk_*kernelname*_u with impugnity. The coder can also use volk_*another_kernelname*_a, volk_*another_kernelname*_u, and volk_*another_kernelname* with impugnity, providing a convenient mechanism for developing *meta-kernels* (kernels calling code from other kernels). The dispatcher proto-kernel must match the output of the generic proto-kernel, but this restriction is essentially the only one.

IV. Integrating VOLK into GNU Radio

The VOLK dispatcher ensures correctness across all alignment situations at runtime, but correctness is not the exclusive goal of a vector-optimized library. VOLK goes a long way toward selecting optimal code on a per-situation basis, but it does nothing to create optimal situations. GNU Radio is an implementation for a particular computational model, and the specifics of that implementation create the runtime situations exposed to the VOLK library. We discuss some of the specifics of that implementation and trade-offs behind different approaches to using VOLK within the GNU Radio code base.

A. Alignment

GNU Radio uses circular buffers to pass data between signal processing blocks. The dynamic scheduler keeps track of the read and write pointers of each buffer to determine how much data is available for a block's input and how much space is available for a block to write its output. Each call to a block requires a set of calculations to determine how many items the block may handle and produce.

GNU Radio creates these buffers on page boundaries. Due to buffer page-alignment we know GNU Radio blocks start life with input and output buffers aligned to SIMD architecture requirements. Unfortunately, due to the dynamic nature of the scheduler and the dynamic nature of sample arrival into flowgraphs, we cannot guarantee *a priori* any call to a block's work function should continue to respect byte alignment beyond the first call. And so we have to be more clever with our scheduling algorithms.

It is relatively trivial to force the scheduler to allow blocks to produce an *output multiple* of items. This value is an integer multiple such that each time a block is passed data, the scheduler will always pass it data sufficient to produce output-multiple-many items. Forcing an output multiple can keep input and output buffers aligned to SIMD architecture requirements. If we use single-precision floats as an example with an SSE architecture, we need 4 floats per SSE register, and so using and output multiple of 4 will preserve SIMD byte alignment between calls to work.

In GNU Radio, there exists a call, $set\_output\_multiple$, which lets the user tell the scheduler always to expect an out-

put multiple from a given block. VOLK can internally query the machine architecture with a call to $volk\_get\_alignment$ that returns a value for SIMD byte-alignment. The two tools together allow GNU Radio to guarantee a SIMD byte alignment for the use of VOLK kernels in work functions.

Dynamic arrival times of samples through source blocks poses a problem to this approach. If a GNU Radio application receives packets of samples, and if those packets happen to contain a number of samples incommensurate with the output multiple requirement of the buffer, GNU Radio will simply hold the remaining samples until there are enough to meet alignment requirements. If there is a significant time difference in the arrival of packets, we then pass significant latency on to the system. GNU Radio waits patiently for another few bytes in order to finish crunching on a packet that has already arrived in its entirety. For example, using floats with SSE, we need a multiple of 4 floats, or 16 bytes, to call the processing block. If a packet arrives with 516 bytes (or 129 floats), we can easily process the first 512 bytes. The scheduler can break 512 bytes into chunks that preserve SIMD byte-alignment, but the packet has an inconvenient extra sample. GNU Radio will hold this last sample of 4 bytes stranded in a source block because the source block will release floats only in multiples of 4. If a packet arrives every two seconds, the application finishes our example packet only after the passage of two seconds brings another packet and after the final byte traverses the GNU Radio flowgraph. Faster though the individual processing blocks might be, the entire system (for this example at least) is slower.

GNU Radio has adopted an alternative strategy: rather than force an output multiple to maintain alignment, we merely *request* the alignment. In this model, a block requires an alignment value to call an aligned VOLK kernel, and we use the $volk\_get\_alignment$ mechanism combined with knowledge of the block's item size to derive a desired output multiple. The block then avails itself of a new GNU Radio utility function: $set\_alignment$. This function defines a block's VOLK alignment requirement in items (*i.e.*, floats, ints, complex, shorts, etc.).

The scheduler interprets this alignment value as a goal but not a requirement. When possible, the scheduler will produce a number of items to the block meeting the alignment goal, but the scheduler will pass a reduced number of items to the block if it cannot satisfy the alignment goal.

After passing an unaligned buffer of data to the block, the scheduler will then attempt to correct the problem as quickly as possible to allow the block to resume aligned processing.

Two points regarding requested alignment deserve note. First, we have introduced more logic into the scheduler to test if there are enough samples for alignment and try to re-establish alignment as quickly as possible. Meanwhile, the blocks themselves must now make a branching statement within the VOLK dispatcher to test for unalignment. All of these steps add cycles to the operation of GNU Radio applications, and cycles can have critical performance implications.

The discussion falls to the subject of amortizing overhead. Unlike the GNU Radio block's work function, and unlike the VOLK kernel, the scheduler is the unique entity within the GNU Radio system best able to load-balance work functions dynamically. The scheduler is, therefore, unique in its ability to force overhead amortization. And the scheduler does so: it tries to maximize the number of items passed to a block. When shifting from aligned to to unaligned states, and when shifting back again, the scheduler packs as many items as possible into the input buffers of its blocks. It does not optimize for running as much data as possible through aligned VOLK kernels when shifting to or from an unaligned state. Early attempts at integration showed the overhead of the extra cycles required to maximize flow through the aligned VOLK kernels and added buffer manipulation come at a far greater cost to the system than falling back on an unaligned VOLK kernel.

Pushing overhead up the stack into the scheduler appears optimal when an application leverages VOLK kernels. The scheduler, unlike the kernel, can adapt to the overhead across an entire application by scheduling larger granules of work. On the other hand, pushing branching cycles up the stack adds overhead even to applications without VOLK kernels and without alignment concerns. This trade-off appears sound: the applications likely to require efficiency are precisely those likely to need alignment; they are the applications likely to leverage VOLK.

There are limits to the value of these unquantized arguments. GNU Radio cannot afford to throw performance for general applications under the bus in support of applications taking maximum advantage of SIMD programming. Added overhead to the scheduler is a concern. We examine consequences to using VOLK in the next section, investigating this overhead in particular.

B. Correctness

Part of VOLK's principle contract with the users is the assertion that all proto-kernels will behave the same on a given machine. When developing a kernel, the generic proto-kernel should be externally verified for correctness. Then, each new proto-kernel is tested against the generic version to see if they produce the same results. Comparisons are made to some degree of accuracy since different processing engines may produce minor numerical differences. The quality assurance (QA) code can account for these differences, but the developer must remain aware of them and understand to what level of precision VOLK enforces its contract.

GNU Radio provides mechanisms for QA testing all of its blocks, and these mechanisms allow the block designer to account for variance in the VOLK QA mechanisms. The GNU Radio block designer generally crafts sets of known inputs and outputs to a block. The known outputs are compared against the results of the GNU Radio block given the inputs. If they match, again to a predetermined level of accuracy, the QA code passes. We use this technique to ensure that the VOLK and GNU Radio kernels produce the correct numbers and behave properly under various conditions as outlined in the QA tests.

C. Results

VOLK's integration with GNU Radio is only partly complete. Most of the simple signal processing blocks have been converted to use VOLK, but many blocks requiring sophisticated manipulation to expose parallelism or coordination between multiple VOLK kernels remain untouched. In this section, we present results achieved with some of these VOLK-converted, simpler blocks. As a hedge against generalization, we note most of the blocks discussed essentially call a VOLK kernel and do nothing else within their work functions. Though not necessarily typical of VOLK integration, this sort of block deserves heightened attention. These are some of the most heavily used blocks, and many of them are used toward the top of flowgraphs where data rates are highest and efficiency is most crucial.

The ambition set out for this paper is twofold: to introduce in detail one implementation for a cross-platform SIMD design environment and to demonstrate the subtleties behind properly using that design environment for optimization of the GNU Radio code base. Results in this section characterize this second goal only. Development in the GNU Radio community has (unsurprisingly) focused on Intel SIMD architectures, and the following discussion represents speed tests on a single processor, namely an Intel i7 870 (quad-core, 2.93 GHz, 8 MB cache). We report these results primarily to demonstrate how some of the most widely-used hardware and compiler technologies interact with the VOLK mechanisms and the GNU Radio integration pattern to produce efficiency gains (or not).

Our desire is to isolate two effects of the updates to GNU Radio to support VOLK. The first effect is an expected gain in performance owing to the replacement of ordinary scalar operations with SIMD-vectorized operations. The second effect is an expected loss in efficiency owing to added computation within the GNU Radio scheduler to support VOLK, specifically computation to manage alignment of the data buffers passed to work functions. Since keeping buffers aligned comes at a potentially significant cost to the scheduler, a central GNU Radio component, studying this second effect takes on a heightened importance. Without any hard data, we have no basis to evaluate concerns that increased overhead for the scheduler across all of GNU Radio might swamp gains shown for individual VOLK kernels in a simple profiling test.

We test using the simplest GNU Radio flowgraph we can create. The flowgraph uses a 'null_source' block as an input, connected to the block under test, followed by a 'null_sink' block. The null sources and sinks do nothing, so the data passed to the block under test is unstructured contents of GNU Radio buffers. Null sinks and sources are used because they provide source and sink functionality required to define a flowgraph but otherwise have minimal impact on the processing times. These results test for speed but not accuracy of the kernels. The QA tests previously discussed already provides us the assurance that the VOLK functions produce the correct values.

With this basic structure in place, we tested three different scenarios. First, we looked at a GNU Radio scheduler version without modifications to support VOLK kernel usage and at blocks without calls to any VOLK kernel. These experiments use GNU Radio version 3.5.1, and we refer to flowgraph test scenarios with this configuration as 'v3.5.1'. Next, we tested the new scheduler, but without VOLK kernel integration. These experiments use GNU Radio version 3.5.2, and we refer to flowgraph test scenarios with this configuration as 'v3.5.2'. Finally, we tested the same 3.5.2 GNU Radio scheduler with blocks modified to use VOLK functionality. The processor used in these tests supported Intel SSE architectures from MMX up to SSE 4.2. The *volk_profile* application was used to determine which proto-kernels would actually be used, so these tests represent the most efficient proto-kernels available for this processor. We refer to flowgraph test scenarios with this configuration as 'VOLK versions'.

To depict the results for each block, we take performance for v3.5.1 as a baseline. We then plot percentage in runtime improvement versus v3.5.1 for both v3.5.2 and for the corresponding VOLK version. Improvement of 100% indicates the plotted version ran in half the time of v3.5.1 (a $2\times$ improvement). Improvement of -100% indicates the plotted version ran in twice the time of v3.5.1.

We calculated the time required to run the GNU Radio test program by using the Python 'time.time()' function to get the time before the program was run and the time when the program finishes. The difference in this time is the related to the speed of the program.

The Python time function relies on the underlying C library's timing functions. The 'time()' call returns seconds since the epoch. Although its time accuracy is only guaranteed to the second, the Linux platform these were run on provides a time resolution of 10 ms. Further, this is what is known as 'wall time' from the start and finish of the test, which includes any interruptions of the operating system, which are unknown and can skew the timing results. As such, each test was run with one billion samples to ensure that the time span of each tet will not be affected by the coarse 10 ms resolution, and each test was run for 20 iterations with the minimum value selected as the result. In these cases, the minimum represents the time where variations in the system's operations were the least intrusive.

Figure 2 shows the performance difference between different versions for type conversion blocks. The results provide some interesting context to the discussion about GNU Radio's performance. First, in most cases, we get significant *improvements* in speed moving from version v3.5.1 to v3.5.2. Surprisingly, the changes to the scheduler added computational and branching overhead but produced faster code. Likely this improvement is due to aligned memory maintenance affecting cache management positively.

The use of VOLK, then, shows the potential to add another level of performance enhancement, even for these simplest blocks. For some conversions, speedup is only minimal, or even slightly reduced, compared to v3.5.2. The *com-*

plex_to_float conversion demonstrates a small reduction. In all cases where the VOLK version compares unfavorably to v3.5.2, performance is quite similar.

In certain cases, when the compiler can generate SIMD code for a work function, the compiler might easily generate slightly more efficient code from generic C than from C intrinsics, and block overhead for testing alignment is missing from the generic C v3.5.2 experiments. Also, there are times when the compilers can introduce better pipelining, and for simple functions, this might be more important than the ability to parallelize (as pipelining is, indeed, a form of processor parallelizing). These observations could explain slight reductions in performance moving from v3.5.2 to the corresponding VOLK version for some of these tests. Dramatic gains in the other test of the conversion blocks indicate the compiler does not always appropriately handle these cases, even for straightforward conversions.

In particular, we examine the *complex_to_float* conversion that shows a slightly worse performance when using VOLK. Compiling the source code for this block from v3.5.1 into assembly code, we can study it to see if it was vectorized. Using GCC 4.6.3 with the compiler flag '-O2', which is the standard mode it is compile with, the assembly code shows no use of SIMD instructions. Instead, since this is simply a case of moving data appropriately to isolate one of the pairs of floating point values, the compiler is apparently able to handle this in a highly efficient way. Why this block performs any different than the *complex_to_real* or *complex_to_imag* blocks, which do the same thing, is unknown. It is likely, however, that we are seeing a measurement bias.

The next set of tests look at the performance of VOLK for simple math kernels like multiplying and adding signals, and the tests feature in Figure 3. Again, we see that the new scheduler provides a performance improvement, not a reduction, versus the old scheduler. VOLK provides a significant performance jump over v3.5.2 in all but the conjugate block.

To address the equity of the results for the conjugation block, we compiled the source code into assembly. We note that GCC version 4.6.1, the compiler used for this experiment, does have some ability to identify vector parallelism and compile ordinary C code to SIMD architecture instructions. In this case, the compiler is indeed smart enough to handle the conversion using SSE instructions. Because conjugation is simply flipping the sign bit of the imaginary part, GCC appears to have no trouble efficiently unrolling the loop eight times and using the SIMD 'xorps' instruction (XOR single-precision floating point values) along with the appropriate SIMD load and store calls. Handling the SIMD in-compiler and performing the loop unrolling without any extra VOLK overhead has, in this case, proved slightly more efficient.

The results for the processor shown here, an Intel i7 870, do not include information about newer Intel AVX instructions. Performing the same tests using a newer 'Sandybridge' processor (i7 2620M) shows the same trends in both the v3.5.2 and VOLK-enabled GNU Radio tests. A few gains in performance from one generation to the next stood out. With the improved SIMD architecture of the newer processor, the *complex_to_mag* improved from a 600% increase in speed to 1650%. The other type converters scaled accordingly, but this was the most dramatic. Similarly, the *multiply_cc* increased from 550% to 1300%. Nothing inherent to VOLK, GCC of the GNU Radio integration pattern limit performance gains here shown to a single Intel processor.

V. Conclusions and Future Work

The results in this paper show that VOLK integration provides a great improvement to GNU Radio performance. The blocks shown here are only a sampling of the blocks converted to use VOLK kernels. The surprising result is that, even without VOLK, the modifications to the scheduler needed to enforce alignment boundaries also provide performance improvements. These improvements manifest despite added scheduler logic. The improvement is likely due to improved memory management and cache handling within the system.

We experimented only on x86 processors, the processors most commonly used to run GNU Radio. We did not investigate whether these performance patterns hold on other architectures of emerging importance to commercial markets, particularly ARM processors. Looking more into these architectures is a major part of future GNU Radio development.

Further VOLK integration work is forthcoming as we continue to evolve GNU Radio. Where performance limitations are identified, we can direct attention to how VOLK might improve efficiency. Specifically, more complicated GNU Radio blocks stand to benefit by using one or more VOLK kernels.

Both the development of new kernels and proto-kernels as well as their use in GNU Radio will be a continued evolution. One of the strengths of VOLK is the ability to abstract and seperate this evolution from end users performing block-level algorithm development and building GNU Radio application in higher level Python and GRC. When newer and better kernels and proto-kernels are introduced for differnet functions and on different platforms, users of each of these classes will immidiately benefit by the inclusion of these kernels from the underlying volk system with little to no effort.

VI. Acknowledgments

We would like to acknowledge the work of Josh Blum and Nick Foster from Ettus Research, LLC for their contributions to the build system and automatic test and profiling tool for VOLK. The use of VOLK in GNU Radio would not have been possible without their help.

References

[1] www.spiral.net/software/viterbi.html
[2] cmake.org
[3] code.entropywave.com/orc
[4] M. A. Greene, "Pentium(R) processor with MMX(TM) technology performance," IEEE Proc. Compcon, 1997, pp. 263 - 267.
[5] D. Naishlos, "Autovectorization in GCC," Proc. GCC Developers Summit, 2004.

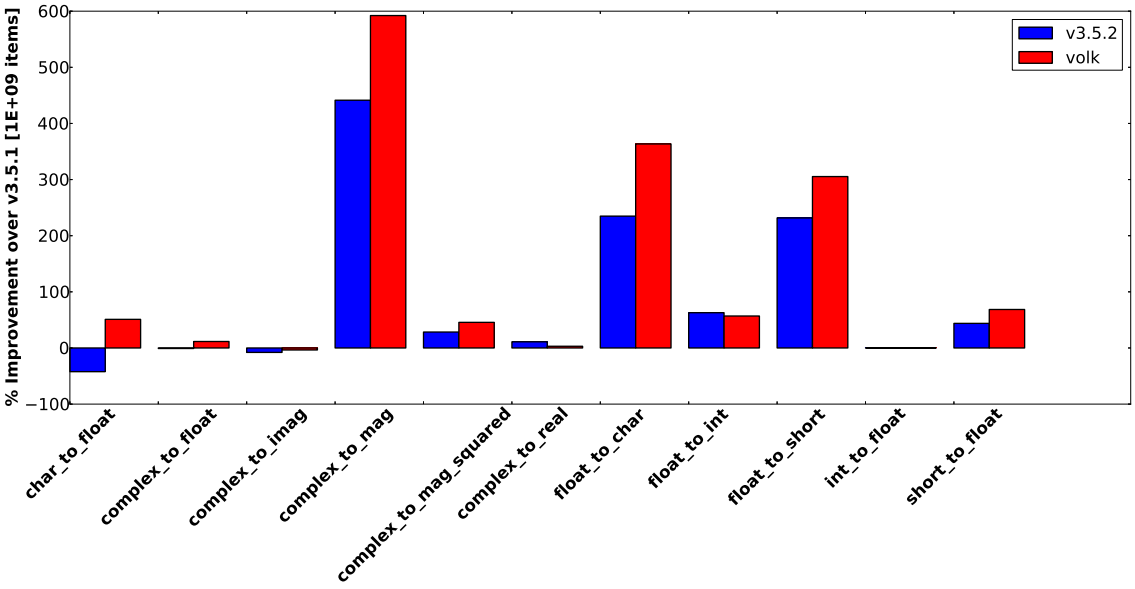

Fig. 2. Improvements for the new scheduler in v3.5.2 and VOLK in type conversion blocks versus GNU Radio v3.5.1.

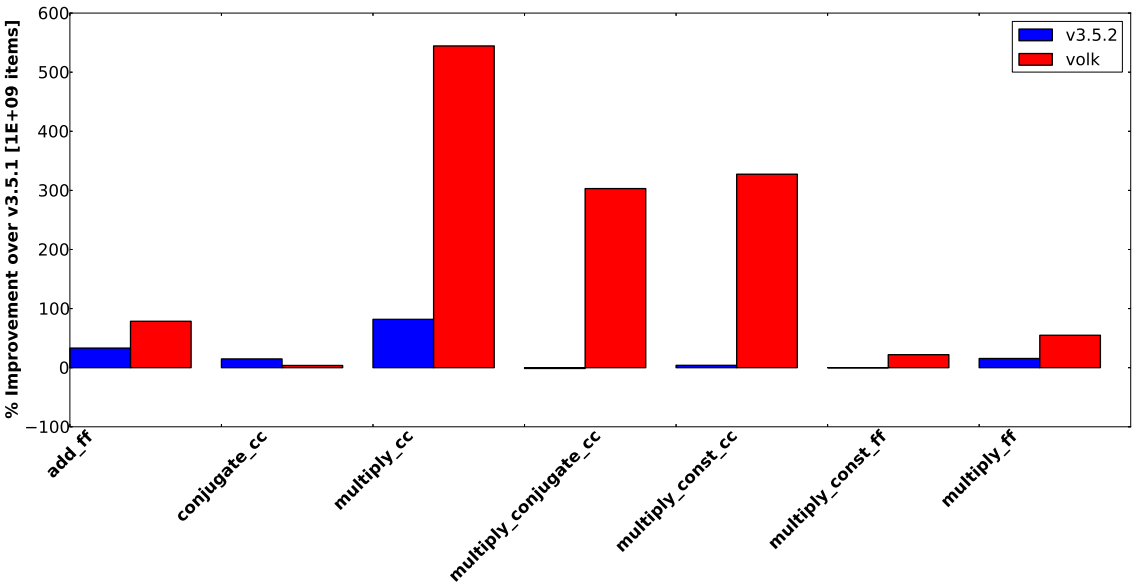

Fig. 3. Improvement of new scheduler and VOLK for simple math blocks from GNU Radio 3.5.1.