Vectorization in PyPy's Tracing Just-In-Time Compiler

Richard Plangger Andreas Krall
Institut für Computersprachen
Technische Universität Wien
planrichi@gmail.com, andi@complang.tuwien.ac.at

Abstract

PyPy is a widely known virtual machine for the Python programming language. PyPy itself is implemented in the statically typed subset of Python called RPython. RPython includes a tracing Just-In-Time (JIT) compiler and is capable of generating the compiler for a language from the specification of the interpreter for that language. In PyPy 4.0.0 we extended the tracing JIT compiler to support vectorization of loops and emit code for the SSE4 vector operations of the x86 instruction set. This article presents the details of the new vectorizer of PyPy. The vectorizer uses a loop unrolling approach to vectorization. It has been designed for efficient compilation as the compilation is done during the execution of the application. The scientific library NumPy introduced arrays which are homogeneous, primitive typed and contiguous in memory. These kind of arrays are used to avoid the problems with dynamic typing. Our contribution to PyPy's new vectorizer supports scalar and constant expansion, accumulator splitting for reductions, guard strengthening and array bounds check removal. The empirical evaluation shows that the vectorizer can gain speedups close to the theoretical optimum of the SSE4 instruction set.

1. Introduction

The programming language Python is becoming popular in the embedded system domain. PyPy is a widely known virtual machine for the Python programming language. Opposed to the standard implementation (CPython), it includes a tracing just-in-time (JIT) compiler supporting different architectures like x86 and PowerPC and making it suitable for embedded systems. The implementation language is a statically typed subset of Python called Restricted Python (RPython). RPython is not only a programming language but also an abstraction for byte code interpreters. It is able to automatically generate a meta-level tracing JIT compiler and a garbage collector. Thus it is not only used for PyPy but also for many other interpreters for dynamic and functional languages or instruction set simulators.

In the last two decades new Single Instruction Multiple Data (SIMD) instruction sets where built into processors to speed up multimedia applications. These instruction sets are not only useful for multimedia applications but also for scientific applications. In theory, given a single precision floating point operation in a loop, if the loop is vectorized to SSE4 instructions (a x86 instruction set architecture (ISA) extension) it executes 4 times faster. If a language implementation can use such SIMD instructions this can lead to very high performance gains of the application programs.

Recent developments in scientific computing have drawn attention to libraries for numerical computations (e.g NumPy). NumPy and others currently remove the interpreter overhead of numerical computations by writing the critical routines in a low level language like C. They are compiled to the host computers architecture ahead of time. At runtime the language interpreter invokes the foreign function compiled earlier. For different language interpreters the internals often are very different. CPython for example uses a different garbage collector than RPython. NumPy depends on some internals of CPythons garbage collector and therefore the library cannot be used directly by RPython. Since NumPy is a commonly used library, PyPy rewrote parts of NumPy and included it in the standard library. This setup renders most of the critical loops as normal program loops instead of foreign functions and makes it desirable to optimize such loops. Python in general is a dynamically typed language. This can lead to run time overheads when the type of values can not be determined at run time or if the type of some values changes often at run time. To simplify optimizations arrays in NumPy are homogeneous, primitive typed and contiguous in memory. These homogeneous and primitive typed arrays lay the ground for PyPy's vectorizer.

In this article the new auto vectorizer built into the RPython optimizing backend is presented. Since the vectorizer is built into the tracing JIT compiler, vectorization happens at run time. This leads to the requirement that vectorization should have low overhead. Traditional high overhead vectorization approaches are therefore not applicable, different approaches have to be used. The contributions of this article are

- an efficient vectorizer based on loop unrolling,
- fast data dependence analysis on traces,
- a cost model for the packing of instructions,
- simple and efficient heuristics for grouping and scheduling instructions and
- minimization of guard instructions by guard strengthening.

First details of PyPys tracing JIT compiler are presented, then the vectorizer with all optimizations is described in detail, finally results on the performance are given.

SCOPES '16, May 23 - 25, 2016, Sankt Goar, Germany.
Copyright © 2016 ACM ISBN 978-1-4503-4320-6/16/05…$15.00.
http://dx.doi.org/10.1145/2906363.2906384

2. Related Work

2.1 Traditional Vectorization

Traditional vectorization [1] [35] analyzes and transforms nested loops. To successfully transform loops transformations like loop normalization, loop collapsing, loop distribution, loop interchange and loop peeling are applied. To determine data dependences between array elements often equation systems considering array subscripts have to be solved. Tests like Banerjee's GCD test [3], the Omega test [28] or the Power test [32] are needed.

Naishlos [21] presented a first version of vectorization for the GCC tool chain in 2004. It uses traditional vectorization techniques [1]. This version did neither support different access patterns (such as even/odd) nor an elaborate alias analysis for pointers. Two years later Nuzman et al. [22] presented the new capabilities. The new version is able to vectorize more loops offering improved support for pointer alignment, pointer accesses, conditional operations, reductions and non uniform strides.

Intels vectorizer [6] uses the same approach as outlined in [1]. A lot of effort has been put into avoiding pitfalls of their architecture and support of accumulation as well as realignment by multiple loop peelings. The results show stunning performance boosts of the 16-bit integer dot product. However only two out of fourteen benchmarks from the SPEC2000 benchmark suite show significant speedups.

At the same time as the first SIMD instruction sets found their way into production processors (e.g. VIS in 1994 or the MMX in 1997), Cheong and Lam[9] describe an auto vectorizer split into two phases. A source to source compiler first translates loops into a parallel form. The second phase tackles the following problems: alignment of memory load and stores, mask generation, partial stores, expansion, packing and comparison vector operations.

2.2 Vectorization by Loop Unrolling

Vectorization by loop unrolling does not consider cyclic loops on the whole, but only the acyclic loop bodies. Krall et al. [16, 27] propose an algorithm that operates on a machine independent IR and generates short SIMD instructions for loop bodies with multiple basic blocks. It combines several structurally equivalent statements in unrolled loop bodies into single short SIMD instructions using an acyclic data dependency graph. Both static alignment analysis and dynamic alignment checks with loop versioning are employed to respect alignment restrictions.

Similarly Larsen [17] analyzes basic blocks from unrolled loops going through the following steps: finding adjacent memory references, extending packed instructions, combination and scheduling.

Leupers [19] modeled SIMD code selection as an integer programming problem. For a data flow graph of 95 nodes code selection takes 26 seconds.

In the SPIRAL project Franchetti et al. [13, 14] developed a source to source compiler that specifically targets Digital Signal Processing (DSP) transformations. C compiler macros are used to leverage SSE compiler intrinsics. SPIRAL transforms the mathematical representation of the problem to C source code which simplifies vectorization. The later work describes in detail how to overcome the issue of non adjacent memory accesses. The evaluation shows that the optimizer is comparable with hand optimized third party DSP transformation libraries.

Wu et al. [34] present a novel approach of "virtual vectors". A virtual vector consists of a length and type of its elements. Transformations like parallel reduction work on top of virtual vectors. The proposed solution is implemented in IBM's XL production compiler and shows that both micro kernels and real benchmarks gain significant speedup.

Up to this point, handling control flow constructs have not been addressed elaborately. Shin et al. [31] try to find efficient ways to map control flow to SIMD instruction sets. In the unrolled basic blocks control flow is converted into data dependencies using *if-conversion*. An "unpredicate" step turns the predicated instructions again back to normal control flow.

Hohenauer et al. [15] present an unroll and pack based SIMD optimization framework for retargetable compilers. Experimental results for different processors demonstrate that the proposed technique applies to real-life target machines and that it produces code quality improvements close to the theoretical limit.

Porpodas et al. [25] inject redundant instructions into the code to transform non-isomorphic code sequences into isomorphic ones. The padded instructions can then by vectorized.

2.3 Aligned memory operations

Some SIMD instruction sets only allow the loading and storing to memory addresses that are aligned. Pryanishnikov et al. [26] determine pointer alignment using inter-procedural abstract interpretation. Roughly 50% of all alignment decisions can be determined statically. Moreover this information can reduce the size of the compiled binary on the evaluated programs up to factor of 4.5.

As mentioned earlier, aligned load/store operations can be followed by several reorder operations to move vector elements to the right positions. Eichenberger et al. [11] and Wu et al. [33] show that it is possible to emulate alignment at runtime. The concept of a stream (i.e. a sequence of consecutive array elements) form the foundation of their contributions. Using a stream offset, they define the constraints of a valid SIMD transformation. Misalignment forces the compiler to insert data reorganization instructions to adhere the previous constraints. Because data reorganization can happen at several positions, three different policies (Zero,Eager,Lazy) control the placing of reorganizations.

Alvarez et al. [2] describe the performance impact of unaligned memory references. Their experiments show that it is critical to prove alignment or to provide unaligned access to memory in the instruction set to get performance gains for SIMD instructions. They conclude that for multimedia codes unaligned memory accesses are very useful.

2.4 Interleaved Data Formats

Nuzman et al. [23] present a transformation scheme that efficiently supports vectorization in the presence of interleaved data. The only constraint they impose is a stride that is a power of two. Their modifications to the GCC compiler tool chain track each load/store operation with an index to a group. Utilizing this information it can be determined how to reorganize the loaded data into the correct format and reorder it just before storing it to memory. Benchmark programs show significant speedup and highlight the benefit this reorganization can have for interleaved formats.

Ren et al. [29] present an algorithm to vectorize loops without restricting the reorganization to a power of two. Program source code is transformed to a generic representation followed by an optimization step minimizing the amount of data permutations. The code generator emits the necessary data reorganizations while assembling the final object code. Approximately 77% of data permutations can be eliminated gaining roughly 68% percent speedup for the benchmark programs on the SSE2 platform.

Liu et al. [20] improve grouping using a variable pack conflicting graph and a grouping graph. Data layout optimizations reduce mandatory packing/unpacking operations by reorganizing data in the memory.

2.5 Just-In-Time Auto Vectorizers

Vectorization as an optimization technique in JIT compilers is seldom. El Shobaki et al. [12] extended the Jikes RVM to automatically vectorize loops. The technique that used an extended tree pattern matcher was not improved by follow up projects. Barik et al. [5] also added a vectorizer to the Jikes RVM. Their framework included an efficient dynamic programming-based vector instruction selection algorithm which supports scalar packing of multiple scalar variables, the use of shuffle and horizontal vector operations and algebraic reassociation.

Rohou et al. [30] try to find data parallelism on byte code level by annotating information that can later be used by the JIT compiler in the virtual machine (VM). Lesnicki et al. [18] also annotated the byte code to enable the VM to vectorize loops with low overhead.

Another approach that couples an offline preparation stage and a lightweight online stage was implemented by Nuzman et al. [24]. The output of the offline stage is a portable vectorized byte code format. They provide a list of idioms the online stage can easily recognize and efficiently map to the target architecture. Configuration in the offline stage is provided to the online stage consisting of alignment properties, cost model metrics and other convenient parameters. The evaluation showed that this approach can be used to successfully speedup various benchmark programs.

3. PyPy's tracing JIT compiler

The meta-level tracing JIT (TJIT) compiler is the main component of PyPys language development environment for dynamic languages. Here only a short introduction is given, more details can be found in the articles describing PyPys TJIT compiler [7, 8]. To implement a dynamic language with PyPy an interpreter has to be written in RPython. The language interpreter is augmented with hints which specify which data is used for language interpretation, e.g. the program counter for a byte code interpreter or a frame pointer. The RPython interpreter starts execution of the language interpreter which executes the application program. When a loop is executed often the TJIT compiler is activated to translate a trace of the loop body's intermediate code instructions to machine code. The specialty of RPythons TJIT compiler is that it optimizes a trace which represents an unrolled interpreter loop which executes an application program loop. With the hints of the language implementer the TJIT compiler is able to eliminate the complete overhead of the interpreter loop. After the optimization only the instructions of the application programs loop body remain.

During the execution PyPy switches between various states. This separation is inspired by [10].

- **Interpret**. Byte code instructions are dispatched one by one modifying the internal state of the VM. This is the default mode when PyPy starts its execution. Backward jumps are instrumented to count the loop iterations. A loop is considered **hot** after exceeding the iteration threshold. The interpreter enters the **Trace** state.
- **Trace**. The next execution of the program loop records the instructions as a history. At points where the trace could be exited (e.g. conditional jumps) guard instructions are inserted. If the number of traced instructions does not overrun, the sequence is provided as input to the optimizer (State **Optimize**).
- **Optimize**. This is a critical step to simplify the trace, remove and exchange instructions. It applies commonly known optimizations such as constant folding, constant propagation, strength reduction, invariant code motion, partial redundance elimination and many others. Both guard implication and guard strengthening are two optimizations also implemented in PyPy. This state is immediately followed by **Compile**.
- **Compile**. This step transforms the tracing history to native machine code at runtime. It covers both register allocation and instruction selection. PyPy offers several different backend implementations such as x86 (both 32 and 64 bit), ARM and a PPC backend. After the compilation succeeded, the interpreter enters the **Run** state.

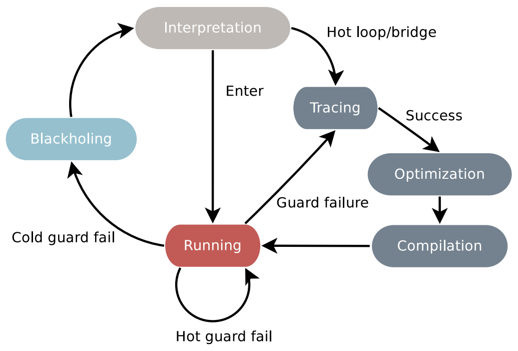


Figure 1. All interpreter states the TJIT switches between

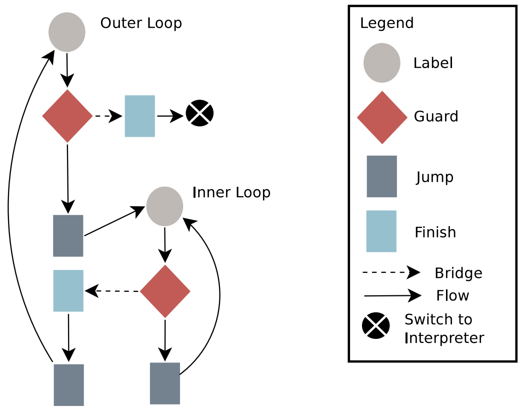


Figure 2. A trace tree constructed by e.g. PyPy's tracing interpreter. It shows a doubly nested loop.

- **Run**. The VM executes any compiled trace. Before the interpreter can resume normal execution, the **Blackhole** interpreter (reached from "Cold guard fail") reconstructs the interpretation state. Cold guard fails count guard exits, eventually reaching a threshold and tracing the interpreter instructions (Edge "Guard failure"). The trace is stitched to the guard instruction creating another branch to the trace tree. "Hot guard fail" continues to execute compiled machine code.
- **Blackhole**. The state to reconstruct the environment to continue interpretation. This is necessary when exiting a trace at any guarding instruction that has no stitched trace to it. Afterwards the next byte code can be sanely executed and the normal interpreter resumes.

Figure 1 shows the states and various transitions between them. An edge description containing "cold" means that the threshold has not been reached yet. On the contrary "hot" means that the execution can follow a stitched trace or that a candidate for tracing has been found.

3.1 Building Trace Trees

The tracing JIT compiler is generated for the main interpreter loop dispatching the byte codes. The only addition required to the interpreter annotates the dispatch header and the backward jump. An automatic process creates an abstract representation that can be traced and JIT compiled. Figure 2 shows a sample trace tree.

It represents a nested loop, switching to the inner loop in the middle of the outer loop. Guarding instructions ensure the correctness of the execution. Whenever a guard fails frequently, a "bridge" is created and attached to the trace. To exit a trace loop the bridge ends in a “Finish” operation and continues to execute an outer loop or switches back to the interpreter.

The instructions that form the outer loop body are split by the inner loop. Operations prior the inner loop are executed from the outer loop header until entering the inner loop by a “Jump”[1] operation. The guard exit leading into a “Finish” operation executes all operations that succeed the inner loop until the outer loop is closed again.

The existing TJIT compiler had to be extended in many areas to support vectorization. The following contributions have been added:

- Create a dependency graph for trace instructions.
- Unroll a trace loop by a factor greater than two.
- Find, extend and combine groups of parallel instructions.
- Schedule a dependency graph and emit vector statements.
- Strengthen guards that protect comparison.
- Create several different version of the trace loop and stitch it to guard instructions.
- Support accumulation patterns (e.g. sum).
- Remove redundant array bound checks.
- Emit SSE4.1 machine code for vector instructions.

4. Motivation

PyPy is eager to provide parts of the NumPy library within the standard library of their virtual machine. At the time of writing one of the biggest challenges is to compete with the speed of native code produced by an ahead of time compiler for NumPy kernels. It was decided to reimplement part of the library due to major limitations.

- Many array operations invoke foreign functions. The penalty can be significant for PyPy.
- They are written and must be maintained in a low level language (e.g. Fortran,C).
- By reason of the moving garbage collector, there is no API to let foreign code access PyPy’s internal objects. This is one of the biggest limitations that separates CPython and PyPy.

The native NumPy routines used by CPython are written in C and use the CPython API to manipulate Python objects. It uses a preprocessing utility[2] to generate all numerical kernels and use plain memory/pointer arithmetic to access elements. The loop kernels are unrolled manually to ensure that SIMD operations are emitted by the ahead of time compiler.

The numerical kernels of PyPy are written in RPython using an iterator API to access memory elements. The numerical kernels are parameterized with the kernel function, operator types and result type. They take full advantage of the tracing JIT compiler.

5. Design

Program transformations for vector machines try to maximize the size of vectors to be processed in parallel. The bigger the input vectors are the more instructions can be executed in parallel. Statements and the loop nest provide the basic information to build a cyclic dependency graph. Strongly connected components (SCC) are identified and the graph’s topological order is used to emit vector statements that are not contained in SCCs. SIMD instructions have a bounded vector size thus the usual abstractions force the code generation to split up the vectors into short vectors again.

In a tracing context the nesting of a loop is opaque and the inner most loop is always traced first. This limitation is a design decision that helps to cope with one problem object oriented languages impose on the runtime: abstraction through layering. A function call often flows through several object layers to accomplish small tasks. The well known optimization to improve performance in these cases is called “Inlining”. A tracing compiler can efficiently inline and optimize the execution. At the same time the assembled machine code size of a trace is only a fraction compared to a method based compiler.

Practically speaking, the abstractions for nested loops and acyclic dependency construction are well suited for vector machines. Whenever time is of essence and the vector size is bounded a different approach might yield similar results. The algorithms proposed by Krall and Lelait [16] and Larsen [17] are able to vectorize traces.

Parallel instructions are gathered by unrolling the loop. Dependency construction is simplified because cyclic dependencies are ignored. Only loop independent dependencies are tracked using the definition-use chains of the trace. This can be done in a linear pass over the trace loop using an associative data structure to remember definitions. By contrast, approaches like the Power test [32], the Omega test [28] or the well known GCD [4] test need linear/affine equations and solvers to determine the dependency.

The rest of the algorithm boils down to a scheduling problem. The dependency graph is used to group independent and isomorphic instructions. This information is then considered while rescheduling the trace and emitting vector instructions.

6. Vectorization on traces

The optimization routine is outlined in Algorithm 1. Although the implementation in the RPython optimization backend is quite similar to the work of Larsen [17] and Pryanishnikov [27] there are some key differences.

Algorithm 1 shows the preparation routine for a trace loop and the algorithm to vectorize trace loops. The function BASICINFO returns the smallest type in bytes (for load/store operations), a list of operations that reference memory (read/write) and all modifications on index variables. The three different information types can be acquired in a single forward pass. The unrolling factor is heuristically determined by the smallest type and the size of the vector register. The smallest type has been chosen, to offer more opportunity to pack instructions. By choosing the biggest type, occasionally packed instructions do not span over the whole vector register.

Tracing checks the loop index at the end of the trace, before it jumps back to the header. This check at the end adds a dependency to the next load instruction and the previous store instruction of the unrolled trace loop. It is impossible to execute the instructions in parallel. RELAX in Algorithm 1 finds the index guards and moves them to the beginning of the loop. This operation is then marked as an “early exit” which enables the dependency builder to reduce the dependencies.

The output of PREPARE is the input for VECTORIZE. I_v is used to determine if memory loads/stores alias or if they are adjacent in memory i.e. ADJACENT. Without inferring this information, the resulting dependency graph cannot assume that two memory stores don’t depend on each other. This introduces edges which are not necessary in most cases, but prohibit vectorization.

[1] In RPython, this jump is named “call assembler” and is a different operation than the jump to a loop header.

[2] It does not use the preprocessor to duplicated routines for different element types. The preprocessor is annotated in comments.

Algorithm 1 Vectorization optimization routine

```
T ... Trace loop
vs ... Size of the hardware vector register
$M_r$ ... Set of instructions that read/write memory references
$I_v$ ... Set of affine combinations for indices
function PREPARE(T,vs)
    T ← RELAX(T)
    b, $M_r$, $I_v$ ← BASICINFO(T)
    factor ← $\frac{vs}{b}$
    $T_u$ ← UNROLL(T,factor)
    return ($T_u$,$M_r$,$I_v$)
function VECTORIZE(T, $M_r$, $I_v$)
    G ← BUILDDEPENDECYGRAPH(T, $I_v$)
    $P$ ← INITPAIRS(G, $M_r$, $I_v$)
    $P$ ← EXTEND($P$, G)
    $P$ ← COMBINE($P$)
    $T_{vec}, savings$ ← SCHEDULE(G, P)
    if savings $\leq -1$ then
        return T
    return $T_{vec}$
```

INITPAIRS, EXTEND and SCHEDULE are shown in Algorithm 2,3,4 respectively.

6.1 Initialize and Extend

INITPAIRS create pairs of adjacent memory operations that are both isomorphic and independent. ISOMORPHIC is defined as "semantically equivalent intermediate instruction". Relying on these properties, a parallel execution is semantically valid.

Algorithm 2

```
function INITPAIRS(G, $M_r$, $I_v$)
    $P \leftarrow \emptyset$
    for $m_1, m_2 \in M_r \times M_r$ do
        if ADJACENT($m_1, m_2$) $\wedge$ ISOMORPHIC($m_1, m_2$) $\wedge$
         INDEPENDENT(G,$m_1$,$m_2$) then
            $P \leftarrow P \cup$ PAIR($m_1$,$m_2$)
```

EXTEND enumerates all known pairs and tries to follow the definition and use chains. The Cartesian product of the two calls to DEF/USE represent the instructions combinations possible for two pairs. These candidates are subject of extending the list of pairs. The idea of this algorithm is to find the pairs that directly use input pairs to the same argument slots. If the operation has a vectorized equivalent, a hardware SIMD instruction might be able to execute the operation faster. The routine continues as long as new candidate pairs are found.

6.2 Combine and Schedule

Up to this point only pairs of operations have been recorded. By design pairs can overlap with other pairs. Given the two pairs (l_1,l_2) and (l_2,l_3) they can be merged into a pack of three elements (l_1,l_2,l_3). This task is accomplished by COMBINE. It has been omitted from the listing, since it's implementation is straight forward. It simply compares pack by pack and merges them if the right most operation matches the left most. It already takes into account the vector size provided by the target ISA and stops to pack further operations if the limit of the vector size is reached.

To accomplish tight packing and to minimize the number of resulting packs the input pairs are sorted. Each pair's first operation is sorted ascending. The current pack is expanded as long as there are more matching packs and the capacity has been reached.

Algorithm 3

```
function EXTEND(P, G)
    $C \leftarrow \emptyset$
    while $C \neq |P|$ do
        $C \leftarrow |P|$
        for PAIR($i_1, i_2$)$\in P$ do
            for $i_3, i_4 \in$ USE(G,$i_1$) $\times$ USE(G,$i_2$) do
                if ISOMORPHIC($i_3, i_4$) $\wedge$ INDEPENDENT(G,$i_3$,$i_4$) then
                    $P \leftarrow P \cup$ PAIR($i_3$,$i_4$)
            for $i_3, i_4 \in$ DEF(G,$i_1$) $\times$ DEF(G,$i_2$) do
                if ISOMORPHIC($i_3, i_4$) $\wedge$ INDEPENDENT(G,$i_3$,$i_4$) then
                    $P \leftarrow P \cup$ PAIR($i_3$,$i_4$)
```

In the last step the trace is rescheduled using the information gathered earlier. The scheduling algorithm is interwoven with logic to estimate the savings of the loop. The estimated savings for packing an instruction is modeled using the CPU architecture in mind. The basic saving can be calculated using the following formula: $s = -cost + count(pack) * benefit$. E.g. The SSE4.1 instruction ADDPD is modeled as $s = -1 + 2 * 1 = 1$.

UNPACKCOST models the costs needed to unpack variables that are contained in any vector registers. Depending on the position the function estimates costs modeled after the CPU architecture. E.g. unpacking the higher element of a double precision floating point has a higher cost than unpacking the lower element[3].

Scheduling picks a candidate operation that is scheduleable. An operation in the dependency graph is schedulable if there are no edges that point to the operation. This is trivially true for the label operation, which starts the scheduling.

If the candidate operation to be scheduled has an associated pack, all operations are transformed to a single vector operation by VECTOROPERATIONS. For this to succeed all operations of the pack must be schedulable, otherwise the current candidate is postponed. Then all edges to descending operations (i.e. the ones that depend on the current operation) are removed in SCHEDULED. A call to NEXT gathers all operations that are now schedulable after edges have been removed.

6.3 Enhancements

Scalar constants and variables are expanded. If the scalar value is produced in the loop, the expansion creates the vector register before it is used. In any other case a dedicated vector register is reserved before the trace loop is entered. The constant or variable content is scattered to each slot of the vector register. The operation is later able to use the expanded register instead of executing the loop iterations one by one.

Accumulation of values (e.g. sum,product) can also be transformed into vector instructions. The summation of a vector contains dependent addition instructions for a value that is carried across the trace loop. This pattern is recognized and a special pair is added in EXPAND. Similar to variable expansion the accumulator is expanded before the loop is entered. The summation is done using a normal vector addition. Parts of the sum are accumulated at the slots of the vector register. After exiting the loop through any guard the vector register is added horizontally to a single value. This transformation is only valid for commutative and associative operations such as addition, multiplication, and ($\wedge$), or ($\vee$) or xor ($\oplus$).

[3] The assembler backend needs at least 2 assembler instructions for the high element, instead of a maximum of one for the lower element.

Algorithm 4

```
function SCHEDULE(P, G)
    $S \leftarrow 0$
    $T \leftarrow \emptyset$
    $N \leftarrow$ NEXT(G,$\emptyset$)
    while $N \neq \emptyset$ do
        $O \leftarrow$ HEAD(N)
        pack $\leftarrow$ PACK(P,O)
        if $\neg$ pack then
            $T \leftarrow T \cup \{O\}$
            $S \leftarrow S -$ UNPACKCOST(O)
            SCHEDULED(G,O)
        else
            if PACKSCHEDULEABLE(pack) then
                $S \leftarrow S -$ PACKCOST(pack)
                $T \leftarrow T \cup$ VECTOROPERATIONS(pack)
                $S \leftarrow S +$ ESTIMATESAVINGS(pack)
                SCHEDULED(G,pack)
            else
                $N \leftarrow N \cup \{O\}$
        $N \leftarrow$ NEXT(G,N)
    return T,S
```

6.4 Example

Figure 3 shows an example applying the algorithm described earlier. It spans over three steps. The first step takes a normal trace loop and unrolls it several times. In this example it is assumed that the algorithm determined to unroll the loop once (determined by the element size of the arrays `a,b,c`).

The third step in Figure 3 "relaxes" the guards by moving them to the beginning of the loop. Thereafter properties such as `j = i+1, k=i+2` are known and load/store operations can be found adhering the adjacency property. Pairs of operations are created. They are marked with boxes containing capital letters. Their color code indicates that they use the same operation code, their letter dictates the pack they are in.

After the initial load/store pairs have been found, the definition use chains of packs are followed to obtain more pairs. Pair E accumulates a value reducing `c` by adding each value.

If the loop body was unrolled further, the combination stage would merge pairs to packs. In this example all pairs are automatically transformed to packs, but no actual packing takes place.

The last step schedules the trace considering all vector packs. This emits special instructions that can be directly mapped to SIMD hardware instructions. Pack E forces the guarding operation to replace the accumulation variable with its vector counter part. This is necessary to finally sum up the contents of `v_d` when the loop is exited. This is written as `{..., v_d , ...}`. All other guard operations in this example also carry fail arguments, but for simplicity this is not included in the Figure.

7. Vectorization Heuristics

7.1 Trigger the Search

Basic block vectorization triggers the search and compares load-/store operations. Given `store(p,i,v), store(p,i+1,w)` this forms a pair of operations.

Even for `store(p,i,v), store(p,j,w)` it is possible to form a pair if and only if j is observed to be at a linear offset of i. This case can only succeed if j is either a constant modification of i or they have the same base variable b. Given the sequence `i = b * 2 + 1; j = b * 4/2 + 2` it will create a pair.

The linear combination is only comparable if the multiplicative factor is the same integral value. In addition, it prohibits integral modifications that include more than one variable. E.g. `i = b * 2 + a + 3, j = b * 2 + a + 4` would be adjacent numbers, but not recognized as such. The analysis is capable of expressing the latter case, but does not give indication that it would suffice the adjacent property.

The rationale behind this behavior is that this combination has never been observed either in the NumPyPy traces, nor in user traces.

This step yields pairs that track adjacent memory loads and stores, that could directly be mapped to SIMD instructions.

7.2 Maximize the Pair Candidates

Definition and use of a variable unveils new pairs. The candidates to group are only considered if there is a natural mapping. It requires that no rearranging of the slots is necessary. However, this case can still occur. Loops that swap elements in the array have already pairs for loading and storing. The later steps ensure that appropriate unpacking is done.

The definition of the pair `x = load(p,i), y = load(p,i+1)` leads to two uses: `z = x << 4` and `w = y << 3`. This forms a new pair. The algorithm proceeds not only in forward direction, but also in backward direction following uses to definitions.

The previous steps are executed until there is no new candidate found that could be added to the list of pairs. A fix point has been found, leading to another maximization step. Pairs are extended to packs (as described earlier).

7.3 Splitting extended packs

Given the list of packs, a machine vector register will not always be able to hold a pack. Assuming that the vector register v could be able to hold three elements. Then there would be three possibilities for v to fill it with a sub part of five elements a, b, c, d, e. Either of $v = \{a, b, c\}$, $v = \{b, c, d\}$, $v = \{c, d, e\}$ would be valid.

Even more combinations would be possible if it was allowed to only include lesser elements than the vector can actually hold.

PyPy's heuristic always splits the leftmost elements of the pack. It would choose $v = \{a, b, c\}$ leaving d, e in the pack. $\{a, b, c\}$ form a new vector instruction that is later assigned a hardware register. As for d, e, the pack is discarded because it cannot fill all slots of the vector register. If it could fill the vector register, the pack is reduced until it is empty or too few elements are left.

The decision how many elements are necessary to fill v is solely based on the type information gathered in an earlier step. Packs and their elements are typed (int, float), give hint about their size in bytes and if they are signed (integer case). In this step the hardware architecture leaks information and provides it to the optimizer.

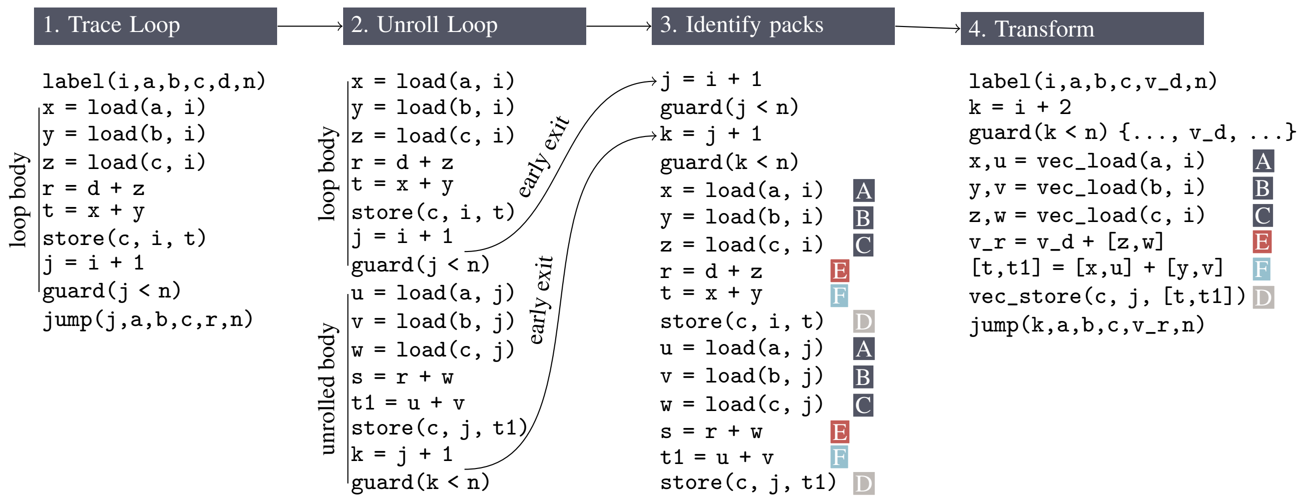
Transforming a half full pack into a vector operation would not be harmful for load operations. It is more problematic for some arithmetic operations (e.g. division by zero) and store operations. The former would terminate the process, the latter could corrupt memory.

These splitting steps are done in a greedy fashion for each pack. It does neither consider that the previous decision needs additional unpacking steps nor that the following packs cannot be used anymore.

In all observed traces in the NumPyPy library it is seldom a restriction. Most of the time unpacking is not done.

7.4 Scheduling

This processes the intermediate instruction. Hereafter they are simply called nodes. Scheduling completes to emit the scalar and vector instructions by walking all nodes that do not have any predecessor. Nodes are linked to their packs they reside in (or not linked



```
label(i,a,b,c,d,n)
x = load(a, i)
y = load(b, i)
z = load(c, i)
r = d + z
t = x + y
store(c, i, t)
j = i + 1
guard(j < n)
jump(j,a,b,c,r,n)
```

```
x = load(a, i)
y = load(b, i)
z = load(c, i)
r = d + z
t = x + y
store(c, i, t)
j = i + 1
guard(j < n)
u = load(a, j)
v = load(b, j)
w = load(c, j)
s = r + w
t1 = u + v
store(c, j, t1)
k = j + 1
guard(k < n)
```

```
j = i + 1
guard(j < n)
k = j + 1
guard(k < n)
x = load(a, i)    A
y = load(b, i)    B
z = load(c, i)    C
r = d + z        E
t = x + y        F
store(c, i, t)    D
u = load(a, j)    A
v = load(b, j)    B
w = load(c, j)    C
s = r + w        E
t1 = u + v       F
store(c, j, t1)   D
```

```
label(i,a,b,c,v_d,n)
k = i + 2
guard(k < n) {..., v_d, ...}
x,u = vec_load(a, i)       A
y,v = vec_load(b, i)       B
z,w = vec_load(c, i)       C
v_r = v_d + [z,w]          E
[t,t1] = [x,u] + [y,v]     F
vec_store(c, j, [t,t1])    D
jump(k,a,b,c,v_r,n)
```

Figure 3. Vectorization example. Calculates `sum(c)` and `c[:] = a[:] + b[:]`. `[a,b]` denotes a vector with two elements. Variables that are prefixed with `v_` denote vector variables. Vector addition is simply written as `v_a + [c,d]`. Step three covers PREPARE,BUILDDEPENDENCYGRAPH, INITPAIRS and EXTEND. Step 4. handles COMBINE and SCHEDULE.

if they are not packed). This can delay the scheduling of a packed node until all predecessors of all nodes in the pack are emitted.

While transforming a pack, it is merged into a single instruction. Several cases can trigger the scheduling of unpacking instructions and even constant/scalar expanded values.

It handles the following cases for each vector argument:

1. The argument can be immediately reused
2. Vector cropping. The size of the input vector is too big/small. This happens frequently for integer sign extensions. Some operations require a specific input type (e.g. 32-bit integer, but not 64-bit integer).
3. Gather values. The conversion from 64-bit to 32-bit float must merge two 32-bit value to four 32-bit values to fill up v.
4. Vector slot movement. This happens for conversion of e.g. 32-bit floating point, to 64-bit floating point. Four values can reside in v, but the conversion needs to move the upper two to the lower position to execute the conversion.
5. Invariant scalar/constant expansion. A dedicated vector register is allocated before the loop is entered.
6. Inline scalar/constant expansion. Values are assembled before they are used.

8. Evaluation

The evaluation is split into two different parts. The first part measures the time spent in the trace loops that have been vectorized and is compared to the scalar trace loops. The second part shows programs that do not stress the vectorization algorithm, but try to evaluate the gain the optimization is able to achieve.

Although PyPy supports many different CPU architectures, only SSE 4.1 for x86 is implemented and used as vectorization target. The next targeted ISA is AVX, the successor of SSE. Still, SSE was chosen as a first goal because of its omnipresence in x86 CPUs.

The vectorizer does not use platform specifics and can be used on different architectures as well. Thus other possibilities include NEON on ARM, AltiVec on PPC and the vector extension on s390x.

Count	Instruction count	Unroll factor	Microseconds	Variance
6	12-16	2	101.47	9.90
5	17-19	4	158.46	4.57
2	17	8	224.03	2.20
2	17	16	396.60	1.24

Table 1. Optimization time measured. Instruction count is the number before the transformation and unrolling has been applied.

8.1 Trace loop benchmarks

The following programs have been evaluated using the following configuration: Intel i7-4550U CPU @ 1.50GHz with 2 cores, Linux Kernel 4.0.6.

The garbage collector "incminimark" was prevented to be run in the trace loop benchmarks by setting the minimum memory threshold to 4GB of allocated memory to avoid collection during benchmarking.

For the following measurements, the tracer and JIT compiler has been instrumented[4] to measure the time elapsed in traces. The function to time the execution was *clock_gettime*. It records the CPU time spent in the process.

Table 1 shows the micro seconds that have been spent in the optimization pass. It excludes all other optimizations.

Figure 4 shows several different vector calculations. The horizontal line shows the baseline of the normal trace. Every program run iterates the operation for 1000 executions. The vector operands are sized four times the tracing threshold. The following listing shows a sample program that is used in Figure 4.

```
def bench(vector_a, vector_b):
  for i in range(1000):
    numpy.multiply(vector_a, vector_b,
                   out=vector_a)
```

Single floating point operations don't show a significant speedup to their scalar trace loops. The reason for this behavior is that floating point operations are always done on the biggest floating point

[4] The revision *a026d96015e4* was used for this benchmark run. It imposes a significant performance penalty when exiting or entering traces.

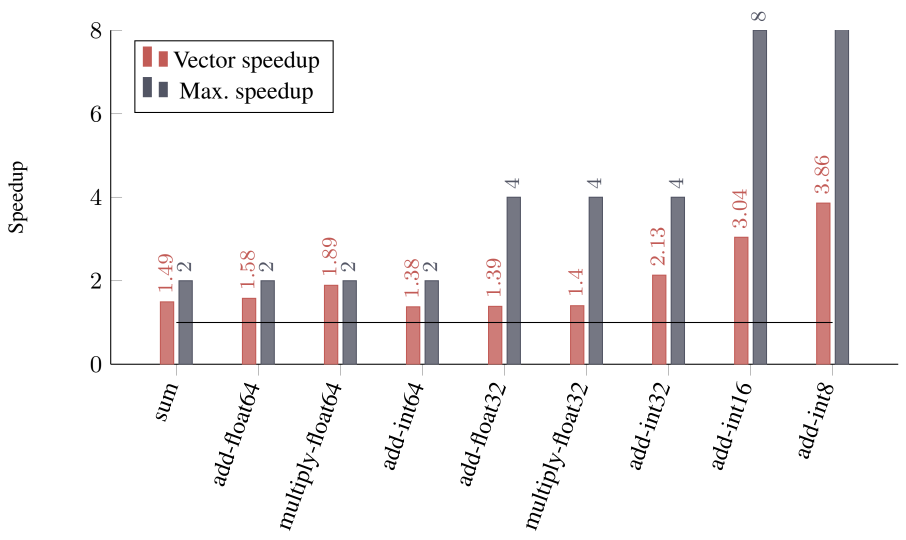


Figure 4. Speedup of the vectorized trace loops. Horizontal line is the baseline for the calculated speedup values ($speedup = \frac{scalar}{vector}$).

type available. The language semantics of Python make the size of floating point numbers platform specific, thus the tracer does not emit floating point operations for single floats, but casts them to double floats.

The theoretical maximum speedup can be observed for loops with double float multiply operations. Other loops show about half of the expected speedup. Considering that it is currently not possible to use aligned vector statements the results are quite satisfying.

Integer addition for 16/8 bits don't show very good results due to the small vector size. It has been observed that on bigger vector sizes these data types perform better. In any case these instructions are not expected to be used very frequently in NumPy programs.

8.2 NumPy benchmark suite

The following evaluates vectorization (henceforward called VecOpt) on small to medium sized numerical kernels. The latter configuration is a mobile CPU chip. For these benchmarks a hardware configuration was used that offers more performance. Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, 4 cores, Linux 4.1.5. Python 2.7.10 and NumPy version v1.9.2rc1 has been used as a base line implementation. VecOpt uses PyPy with the optimization enabled (revision 3742fae37).

Table 3 shows a NumPy benchmark suite[5]. The source code was forked and modified. The modifications executes the kernel several times to warm up the JIT compiler. Each benchmark is repeated five times and the mean value is displayed in the table. For PyPy the benchmark kernel is executed twenty times in the warm up phase. Table 2 shows the loop count of the kernels.

Table 3 shows that for some benchmarks only minor improvements can be achieved. The current weakness both PyPy and VecOpt suffers from is related to the allocation of memory in the benchmark kernel. CPython's GC uses reference counting which immediately frees NumPy arrays. PyPy's GC might keep memory for many more cycles. In Section 8.3 we will see custom written kernels, that do not allocate memory within the kernel loop.

Table 3 indicates that CPython most of the time is a better choice than PyPy. The only reason why CPython has such good results is because a significant fraction of time is spent in native code, removing all interpretative overhead. Furthermore note that the NumPyPy library has not completely implemented all features offered by NumPy.

[5] `https://github.com/planrich/numpy-benchmarks`. Aug. 2015

Name	Loop	Warm up
diffusion	20	5
allpairs-distances	30	20
vibr-energy	100	20
l2norm	100	20
rosen	30	10

Table 2. The loop count and warm up iteration count for the benchmark programs in Table 3. All kernels that are not listed loop 50 times and warm up 20 iterations.

8.3 Pure Python loops and other kernels

To show that there are really more significant improvements than presented in the previous section, a list of benchmarks has been compiled[6]:

- **som** - Self Organizing Maps[7].
- **dot** - Matrix vector dot product.
- **any** - Micro benchmark stressing the any NumPy operation.
- **fir*** - Finite impulse response.
- **add*** - Addition of a Python array.
- **sum*** - Summation of a Python array.
- **rgbtoyuv*** - RGB to Y'UV converions using Python arrays.

All benchmarks that end with an asterisk symbol (*) are pure Python implementations. Indeed the optimizer makes no distinction between NumPy and Python traces, but is currently by default deactivated for the latter.

[6] `https://github.com/planrich/pypy-simd-benchmark` Aug. 2015

[7] This implementation is not complete. It only simulates the "find nearest neighbor" and "update weight vector" step of the algorithm. Is a numeric application that makes heavy use of vector subtractions, multiplications, distance and summation. Similar to principal component analysis this procedure can be employed as a pre step for machine learning.

Name	CPython (C_1)	PyPy (C_2)	VecOpt (C_3)	$Speedup\frac{C_1}{C_3}$	$Speedup\frac{C_2}{C_3}$
allpairs-distances	0.9868	2.57	2.534	0.39	1.0
allpairs-distances-loops	1.826	4.287	4.177	0.44	1.0
arc-distance	0.07898	0.1813	0.1608	0.49	**1.1**
diffusion	0.5603	5.665	3.889	0.14	**1.5**
evolve	0.1967	1.815	1.728	0.11	**1.1**
fft	0.9507	0.2981	0.2955	3.2	1.0
harris	0.3485	3.119	1.504	0.23	**2.1**
l2norm	0.564	1.73	1.634	0.35	**1.1**
lstsqr	0.3844	1.506	1.39	0.28	**1.1**
multiple-sum	0.1432	0.6341	0.5768	0.25	**1.1**
rosen	0.5795	3.498	3.438	0.17	1.0
specialconvolve	0.4713	3.876	2.649	0.18	**1.5**
vibr-energy	0.2784	0.7552	0.699	0.4	**1.1**
wave	2.191	1.114	1.166	1.9	0.96
wdist	2.927	1.202	1.179	2.5	1.0

Table 3. Benchmark suite. C_1, C_2 and C_3 show the CPU clock time spent. C_1/C_3 and C_2/C_3 show the speedup. C_2/C_3 additionally marks the improvements introduced by VecOpt.

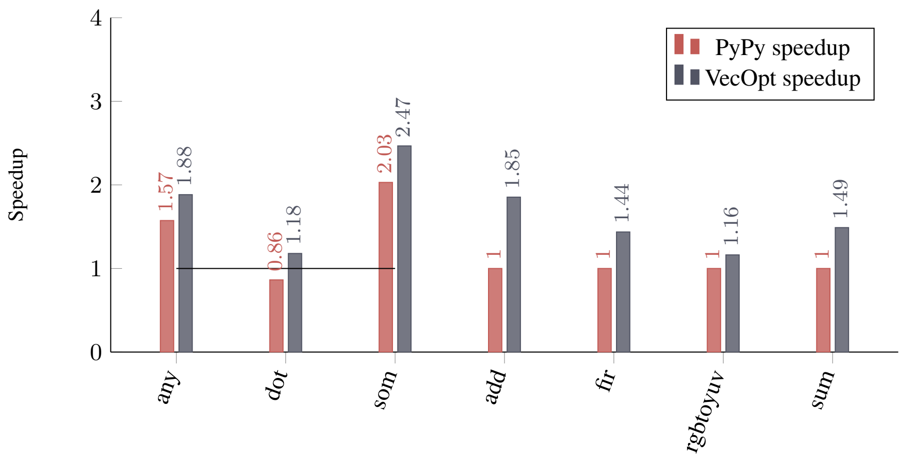


Figure 5. Benchmark plotting the speedup. The first three runs use CPython as base line to measure the speedup (Indicated by the horizontal line). For all others CPython had to be excluded from the benchmark run. All of them are written in pure Python. CPython is not able to execute any computation in native code and thus takes far to long to complete the benchmark run. The speedup of VecOpt in these cases uses PyPy as baseline implementation. Due to some limitations of the current prototype, RGB to YUV operations on floating points rather than 8/16 bit integers.

Name	Vector size	Repeat count
som	256	4000
dot	1000	1000
any	1024	1000
add*	2500	10000
sum*	2500	10000
fir*	200	3000
rgbtoyuv*	$1024 * 768$	500

Table 4. The vector size and the repetition count of the kernel benchmark programs in Figure 5. All programs are run ten times and the mean value is used to calculate the speedup value.

9. Conclusion

It has been shown that a tracing JIT compiler can indeed use SIMD instructions to speed up numerical loops. This is not only true for the NumPyPy standard library, but also for any other traces that adheres the pattern the transformer understands. It additionally shows that the optimization time is reasonably fast and the implementation complexity is rather low. The contributions do not only enhance PyPy, but for any other virtual machine written in RPython. This opens up new possibilities to write a virtual machine that efficiently executes numerical computations using all the comfort a dynamic language provides.

References

[1] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9:491–542, 1987. .

[2] M. Alvarez, E. Salamí, A. Ramírez, and M. Valero. Performance impact of unaligned memory operations in SIMD extensions for video codec applications. In *ISPASS'07: IEEE International Symmposium on Performance Analysis of Systems and Software*, pages 62–71. IEEE Computer Society, 2007. ISBN 1-4244-1081-9.

[3] U. Banerjee. Dependence tests. In *Dependence Analysis for Supercomputing*, pages 101–148. Springer, 1988.

[4] U. Banerjee. *Dependence analysis*. Kluwer Academic Publishers, 1997.

[5] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 201–212, Dec 2010. .

[6] A. J. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the Intel® architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[7] C. F. Bolz and A. Rigo. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.

[8] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[9] G. Cheong and M. Lam. An optimizer for multimedia instruction sets. *Contract*, 30602(95-C):0098, 1997.

[10] A. Cuni. *High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages*. PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, University of Genova, 2010. Technical Report DISI-TH-2010-05, 2010.

[11] A. E. Eichenberger, P. Wu, and K. O'brien. Vectorization for SIMD architectures with alignment constraints. *ACM SIGPLAN Notices*, 39 (6):82–93, 2004.

[12] S. El-Shobaky, A. El-Mahdy, and A. El-Nahas. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 63–69. ACM, 2009.

[13] F. Franchetti and M. Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In B. Werner, editor, *16th International Parallel and Distributed Processing Symposium (IPDPS '02 (IPPS & SPDP))*, pages 20–21, Washington - Brussels - Tokyo, Apr. 2002. IEEE. ISBN 0-7695-1573-8.

[14] F. Franchetti, S. Kral, and J. L. C. W. Überhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, Feb. 2005. ISSN 0018-9219. .

[15] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A SIMD optimization framework for retargetable compilers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1): 2:1–2:27, Mar. 2009. ISSN 1544-3566. .

[16] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[17] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. .

[18] P. Lesnicki, A. Cohen, G. Fursin, M. Cornero, A. Ornstein, and E. Rohou. Split compilation: an application to just-in-time vectorization. In *Workshop on GCC for Research in Embedded and Parallel Systems (GREPS'07)*, Brasov, Romania, 2007.

[19] R. Leupers. Code selection for media processors with SIMD instructions. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'00)*, pages 4–8. IEEE, 2000.

[20] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 347–358, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. .

[21] D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[22] D. Nuzman and A. Zaks. Autovectorization in GCC–two years later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–158. Citeseer, 2006.

[23] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices*, 41(6):132–143, 2006.

[24] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 151–160. IEEE Computer Society, 2011.

[25] V. Porpodas, A. Magni, and T. M. Jones. PSLP: Padded SLP automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 190–201, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8.

[26] I. Pryanishnikov, A. Krall, and N. Horspool. Pointer alignment analysis for processors with SIMD instructions. In V. Chaudhary, A. Dean, and J. Fritts, editors, *5th Workshop on Media and Streaming Processors at Micro'03*, pages 50–57, San Diego, December 2003.

[27] I. Pryanishnikov, A. Krall, and N. Horspool. Compiler optimizations for processors with SIMD instructions. *Software: Practice and Experience*, 37(1):93–113, 2007.

[28] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[29] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. *ACM SIGPLAN Notices*, 41(6):118–131, 2006.

[30] E. Rohou, S. Dyshel, D. Nuzman, I. Rosen, K. Williams, A. Cohen, and A. Zaks. Speculatively vectorized bytecode. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 35–44. ACM, 2011.

[31] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the international symposium on Code generation and optimization*, pages 165–175. IEEE Computer Society, 2005.

[32] M. Wolfe and C.-W. Tseng. The power test for data dependence. *Parallel and Distributed Systems, IEEE Transactions on*, 3(5):591–601, 1992.

[33] P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 153–164. IEEE, 2005.

[34] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 169–178, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. .

[35] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. ACM, 1990.