PyPy's Number Crunching Optimization

Richard Plangger and 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 implemented in the RPython subset of Python which includes a tracing Just-In-Time (JIT) compiler. In this article the new auto vectorizer built into the RPython optimizing backend is presented. It uses the linear sequence of instructions of a trace to find parallelism. Dependency information is gathered and used to reschedule some of the instructions as vector statements.
This optimization is neither tailored for the NumPy library nor for a specific hardware architecture. Every interpreter written in RPython can benefit from the new optimization. To empirically evaluate the optimization, the x86 assembler backend has been extended to emit SSE4 vector instructions for the optimized traces. Preliminary results show that it is indeed possible to leverage the speed gain SIMD instruction sets offer. The implementation is not very complex and the optimizer is reasonably fast.

1 Introduction

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. The implementation language is a statically typed subset of Python called Restricted Python (RPython). RPython is an abstraction for byte code interpreters and is able to automatically generate a garbage collector and a tracing JIT compiler. Thus it is not only used for PyPy but also for many other interpreters for dynamic functional languages or instruction set simulators.

In the last decade new Single Instruction Multiple Data (SIMD) instruction sets where built into processors to speed up multimedia applications. They 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.

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 routine in a low level language. They are compiled to the host computers architecture ahead of time. At runtime the language interpreter invokes the foreign function compiled earlier. 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. To simplify optimizations arrays in NumPy are homogeneous, primitive typed and continuous in memory.

In this article the new auto vectorizer built into the RPython optimizing backend is presented. First details of PyPys tracing JIT compiler are presented, then the vectorizer is described, finally preliminary results on the performance are given.

2 Related Work

The building block for Tracing JIT compilation has been introduced in the Dynamo project [BDB00]. Dynamo is an transparent optimization system that operates on a binary executable. Interpretation starts to execute the program and observes backward branches in a target address cache. By the time a backward branch threshold of an address has been reached, the interpreter switches to a mode where instructions are recorded at the same time as they are executed. This creates a single-entry, multi-exit linear sequence of instructions called a "trace". Results show that this approach is able to optimize opportunities that manifest themselves only at runtime.

Both [AK87] and [ZC90] have laid the foundation for loop transformation into vectorized or parallel form. To transform loops into a semantically equivalent vector form is the data dependence. Strongly Connected Components (SCCs) are built from the data dependency graph and aid to distribute the loop partly or totally into vector form.

Vectorization as an optimization technique in JIT compilers is seldom. One research project extended the Jikes RVM [ESEMEN09] to automatically vectorize loops. The technique that used an extended tree pattern matcher was not improved by follow up projects. Others [RDN$^+$11] try to find data parallelism on byte code level by annotating information that can later used by the JIT VM. [LCF$^+$07] is another example to annotate the byte code generated to enable the VM to vectorize loops.

3 PyPy's JIT

When we speak of PyPy's JIT compiler, we really mean the effort put into the RPython tool chain. The fundamental idea of RPython is to provide both a JIT compiler and garbage collector to any dynamic language written in RPython. Thus it is not only a subset of the Python programming language but also a tool chain to ease the construction of byte code interpreters. The translation itself is a vast topic. It is not the main topic of this document, thus information can be found in e.g. [BCFR09] or [BR07].

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 backwards jump. An automatic process creates an abstract representation that can be traced and JIT compiled. Figure 1 shows a sample trace tree.

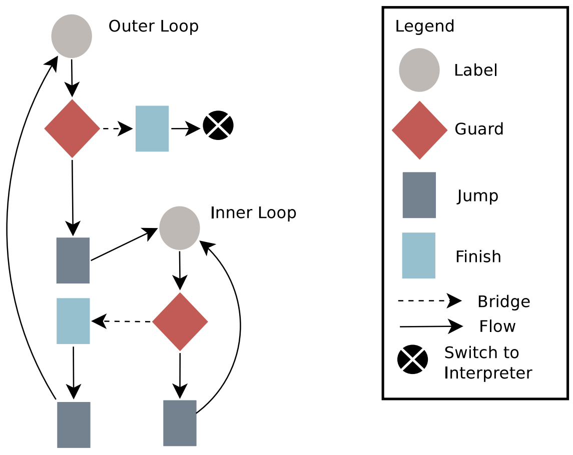

Fig. 1. A trace tree constructed by e.g. PyPy's tracing interpreter. It shows a doubly nested loop.

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.

4 Contributions to PyPy's TJIT

The following contributions have been made to the tracing just-in-time compiler backend and it is now able to:

- Create a dependency graph for trace instructions.
- Unroll a trace loop for a factor greater than two.
- Find, extend and combine groups of parallel instructions.

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

- 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).
- Emit SSE4.1 machine code for vector instructions.

Henceforward the term "VecOpt" will refer to both the branch that includes the changes[2] and the implemented algorithm. Section 8 refers to VecOpt as a compiled PyPy interpreter using the contributions of this document.

5 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 challenge 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 limitation 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[3] 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.

6 Design

Program transformations for vector machines try to maximize the size of vectors to be processed in parallel. The resulting parallel execution improves the bigger the input vectors. Statements and the loop nest provide the basic information

[2] Located at https://bitbucket.org/pypy/pypy. Aug. 2015.

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

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 base 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 algorithm proposed by Larsen [LA00] is able to vectorize basic blocks.

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 basic block. This can be done in a linear pass over the trace loop using a associative data structure to remember definitions. Opposed to this, approaches like the Power test [WT92], the Omega test [Pug91] or the well known GCD [Ban97] 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 emits vector instructions.

7 Superword parallelism on trace sequences

The optimization routine is outlined in Algorithm 1. Although the the implementation in the RPython optimization backend is quite similar to [LA00] and [PKH07] 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 an 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.

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 $T_{vec}$
```

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 e.i. 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.

InitPairs, Extend and Schedule are shown in Algorithm 2,3,4 respectively.

7.1 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.

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

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($m_1$,$m_2$)
```

instructions combinations possible for two pairs. These candidates are subject of extending the list of pairs. The clou of this algorithm is to find the pairs that directly use or 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.

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($i_3$,$i_4$)
```

7.2 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 the minimum 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.

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: $s = -cost + count(pack) * \text{benefit}$. 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[4].

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 (e.i. 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.

7.3 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 operations such as addition or multiplication as well as logical reductions as and ($\wedge$), or ($\vee$) or xor ($\oplus$).

[4] 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
```

8 Evaluation

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

Although the implementation is already nearly finished, these benchmarks are a preview for the final implementation.

8.1 Trace loop benchmarks

The following programs have been evaluated using the following configuration: Intel i7-4550U CPU @ 1.50GHz with 4 cores, Linux Kernel 4.0.6.

The source code can be found in the branch "vecopt" and is based on the PyPy release version 2.6.0. 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. Below the modified threshold, the garbage collector does not start a collection run.

For the following measurements, the tracer and JIT compiler has been instrumented[5] 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.

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

Table 8.1 shows the micro seconds that have been spent in the optimization pass. It excludes all other optimizations.

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

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

Figure 2 shows several different vector calculations. The horizontal line shows the baseline of the normal trace. Every program run iterates the operation for 1000 execution. The vector operands are sized four times the tracing threshold. The following listing shows a sample program that is used in Figure 2.

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

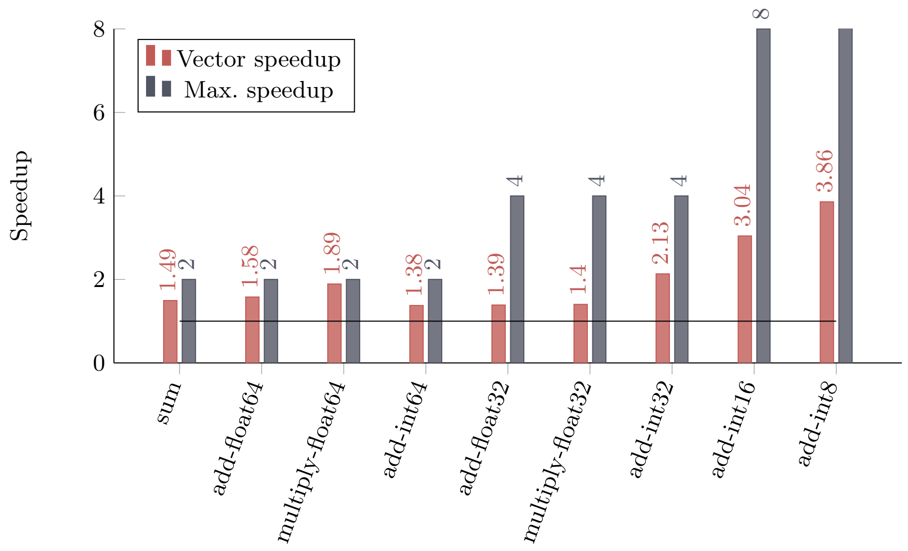

Fig. 2. Speedup of the vectorized trace loops. Horizontal line is the baseline for the calculated speedup values ($speedup = \frac{scalar}{vector}$).

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 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 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 revision **3742fae37** and the forked NumPy branch for PyPy (**504ee4757**). PyPy uses the 2.6.0 release binary (**295ee98b69**).

Table 3 shows a NumPy benchmark suite[6]. 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.

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 3. All kernels that are not listed loop 50 times and warm up 20 iterations.

Table 3 shows that for some benchmarks only minor improvements can be achieved. The current weakness both PyPy and VecOpt suffers from is related

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

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_4 and C_5 show the speedup. C_5 additionally marks the improvements introduced by VecOpt.

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.

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[7]:

- **som** - Self Organizing Maps[8].
- **dot** - Matrix vector dot product.
- **any** - Micro benchmark stressing the any NumPy operation.
- **fir*** - Finite impulse response.

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

[8] 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.

– **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.

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 3. All programs are run ten times and the mean value is used to calculate the speedup value.

9 Future work

PyPy is constantly changed and improved. Only recently work has been started to cut down the optimization time and improve the warm up speed of the virtual machine. Interestingly these changes already track the dependencies of the IR operations and with little effort, the dependency construction step can be merged with the new model. There are plans to integrate these changes and enable them by default with the new optimization setup.

One deficiency has already been mentioned in the evaluation section. Allocating memory frequently is not handled very well by PyPy's GC. Moreover it does not know that the allocated NumPyPy array is a large chunk of memory, but only sees the object encapsulating the pointer to the actual storage. This problem could be mitigated by changing parts of the NumPyPy library.

The unrolling heuristic performs very well for most numerical loops. But it is ignoring the fact that there could be already several memory load/store instructions that could be grouped to one vector operation.

Furthermore there are cases where the input memory locations are interleaved. If follow up instruction wanted to use the vector register, they would need to shuffle the slots to continue. This is not done in the current implementation.

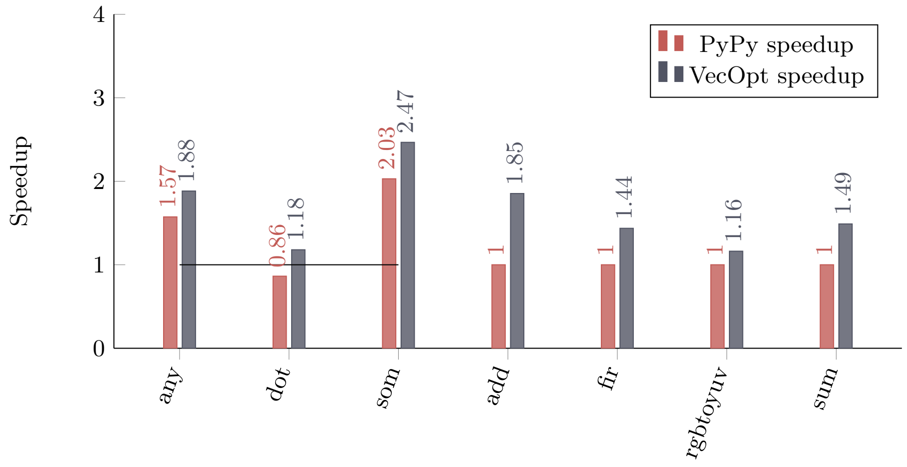

Fig. 3. Yet another benchmark plotting the speedup. The first four runs use CPython as base line to measure the speedup (Indicated by the horizontal line). For all others CPython had to be excluded from the benchmark run. All of them are written in pure Python. CPython is not able to execute any computation 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 bytes.

10 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 executes numerical computations fast using all the comfort a dynamic language provides.

References

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9:491–542, 1987.

[Ban97] Utpal Banerjee. *Dependence analysis*, volume 3. Springer Science & Business Media, 1997.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin 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.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[BR07] Carl Friedrich Bolz and Armin Rigo. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.

[ESEMEN09] Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed 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.

[LA00] Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets*, volume 35. ACM, 2000.

[LCF$^+$07] Piotr Lesnicki, Albert Cohen, Grigori Fursin, Marco Cornero, Andreas Ornstein, and Erven 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.

[PKH07] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Compiler optimizations for processors with SIMD instructions. *Software: Practice and Experience*, 37(1):93–113, 2007.

[Pug91] William 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.

[RDN$^+$11] Erven Rohou, Sergei Dyshel, Dorit Nuzman, Ira Rosen, Kevin Williams, Albert Cohen, and Ayal Zaks. Speculatively vectorized bytecode. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 35–44. ACM, 2011.

[WT92] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. *Parallel and Distributed Systems, IEEE Transactions on*, 3(5):591–601, 1992.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM, 1990.