SAC: An Efficient Retargetable Source-to-Source Compiler for Multimedia Instruction Sets

Abstract

Multimedia instruction sets allow developments of time and power efficient applications. As a consequence they are widely used in embedded systems, e.g. NEON in ARM processors, but also in general purpose processors e.g. SSEx in Intel and AMD processors and AVX for new processors). Unfortunately, it is difficult to write low-level code for such instruction sets, and no portability is possible. So developers rely on compilers to perform this optimization step. But the wide range of hardware targets and the diversity of multimedia instruction sets make it difficult for the compiler community to provide time-to-market compilers for all possible combination of targets. In this paper, we propose a source-to-source approach that combines an efficient vectorization algorithm with a generic instruction set. Back-ends in the form of C implementation of various intrinsics provide retargetable performances. The two first contributions are the parameterized vectorization algorithm and the use of source-to-source capabilities for easy debugging and easy retargeting. As a third contribution, the approach is validated by experiments from several scientific fields carried out on Intel and AMD machines, and comparisons with recent version of `gcc`, `llvm` and `icc`. The average speed-up is 10% over `gcc -march=native -O3`, 20% over `llvm` and 5% over `icc -O3` with some case showing 70% speedup over `gcc` and 20% speedup over `icc`

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords multimedia instruction set, source-to-source compiler, retargetable compiler, automatic vectorization

Introduction

Processors with multimedia instruction sets are now widespread, both for embedded devices (the ARM processors are embedded in billions of mobile phones and mobile Internet devices), desktop stations (Intel and AMD processors provide instructions from the old MMX and 3Dnow! through AltiVec and SSE to the newcomer AVX), game consoles and high-performance environments (as targeted by CUDA and OpenCL with vector types).

However each founder has its own instruction sets, which differ from each other in the nature of the SIMD instructions, the width of supported registers and the variety of data types. Moreover these instruction sets have a solid reputation of being difficult to use, and the average developer relies on his compiler to generate efficient code instead of fine-tuning it. Indeed, it puts an heavy burden on the compiler, because of the constant evolution of the instruction sets and the wide variety of targets.

In this paper, we propose an approach allowing efficient SIMD code generation coupled with a modular, easily retargetable infrastructure. All the algorithms detailed in this paper are implemented in a tool called SAC (SIMD Architecture Compiler). All the implementations and experiments are done within the PIPS [10, 19] source-to-source compiler infrastructure.

The article is organised as follows: In § 1 we expose current usage of multimedia instructions and emphasise the need of a retargetable compiler. In § 2 we present the instruction set abstraction used in the remaining of the paper. The overall compiler organization and the online iterative vectorization algorithm are both presented in § 3 whereas the retargetablity feature is detailed in § 4. Section § 5 presents experimental results on a wide range of scientific applications among linear algebra kernels and signal processing operators. Last section concludes and proposes future works.

1. Motivation & Related Works

There are several approaches[1] to take advantage of multimedia instruction sets as those found in Intel/AMD/ARM processors. Writing inline assembly remains the best option for those who lurks for performance, but its prohibitive development and maintenance costs and the absence of portability layer limit this approach to critical code segment.

Many portability issues can be side-stepped by using `gcc` support for vector types, or intrinsics – `C` functions that directly maps to a sequence of one or more assembly instruction. The later remains a low-level approach and does not change the development costs while the other binds us to a specific compiler. As a consequence, most developers of non time-critical code rely on automatic vectorization. In that field, `icc` traditionally outperforms `gcc`, but only

[1] Among them, only those involving the `C` language, the *de facto* standard for writing "close to the metal" code, are considered.

int8_t, int16_t, int32_t, int64_t, float, double,
complex<float>, complex<double>

Figure 1. Types supported in our meta-instruction set

+, -, *, /, <<, >>, ?:, >, <, min, max
and their saturated variant when relevant

Figure 2. Operators supported in our meta-instruction set

for platforms supported by Intel. On the other hand, gcc supports a wide variety of targets but often provides poorer performance. In either case, auto vectorization is the way to go, provided you can generate efficient code.

However, from a compiler developer point of view:

- instruction sets are in constant evolution;
- debugging generated code (or intermediate code) is difficult;
- integrating new transformations is a long-term task;
- dealing with code written in a legacy instruction set is difficult.

So to the constraints of **auto-vectorization** and **efficiency** of generated code, compiler ought to add an objective of **retargetabilty** if they want to keep up with hardware designer pace.

Several solutions have been proposed to overcome these difficulties: The detailed view of icc internal given in [4] shows that they achieve performance by intensive use of loop vectorization technique[3] and that they rely on re-rolling techniques to vectorize code without loops. For obvious economical reasons, they do not focus on retargetability issues, when llvm[8, 18] and gcc[13, 21] do. They both provide auto-vectorizer, in early stage for llvm [8], more advanced for gcc, with two approaches. One [21] is based on the combination of loop vectorization techniques and super word level parallelism [17, 22], the other relies on the strength of the polyhedral model [24].

Some papers [5, 15] focus on the discovery of instruction-set specific patterns, e.g. multiply-add or horizontal-add. Although finding such patterns provide significant improvement for several kernels, we will focus on pure SIMD instructions for the sake of generality.

A study of the llvm and gcc developer guides convinced us that implementing a full back-end to generate efficient assembly from internal representation is a complex task. On the other hand, using a source-to-source compiler that would generate intrinsics for those compilers seems a remarkably profitable approach, for it benefits from the complex back-end, from the low-level optimisations and make it possible to focus on high level transformations.

This is indeed the approach taken by many research compilers: SWARP [20] uses annotated C code to describe SIMD instruction patterns, a very flexible approach that proved, from its author, to be "too hard to maintain". The approach of [14] achieves retargetability through detailed description of each instruction at the source-level. Pre-processing phases are in charge of applying unrolling and scalar-expansion to their vectorization engine.

2. Meta Instruction Set

Because multimedia instruction sets are in constant evolution, it is critical not to bind the compiler to a specific instruction set. Instead we define an instruction set as the union of the functions found in the different instruction sets. It results in a meta-instruction set very similar to those found in [6, 12], with traditional memory, initialization, arithmetic, logical and comparison operations for integer and floating point data of different size. More accurately, it contains all possible combination of types given in 1 and operators given in 2.

```
void SIMD_ADD_V4SF(float out[4],
                   float in0[4],
                   float in1[4]) {
  for(size_t i=0;i<4;i++)
    out[i]=in0[i]+in1[i];
}
```

Figure 3. Sequential implementation of float vector add

```
#include <xmmintrin.h>
#define SIMD_ADD_V4SF(out,in0,in1) \
        (out) = _mm_add_ps((in0),(in1))
```

Figure 4. SSE implementation of float vector add

```
#include <altivec.h>
#define SIMD_ADD_V4SF(out,in0,in1) \
        (out) = vec_add((in0),(in1))
```

Figure 5. AltiVect implementation of float vector add

```
a4sf pdata0 = {alpha, alpha, alpha, alpha};
//SAC generated temporary array
a4si pdata2 = {1, 1, 1, 1};
//PIPS generated variable
v4sf vec00_0, vec10_0, vec20_0, vec30_0, vec50_0, vec60_0
     , vec80_0, vec90_0;
/* prelude skipped */
SIMD_LOAD_V4SF(vec10_0, &pdata0[0]);
SIMD_LOAD_V4SI_TO_V4SF(vec50_0, &pdata2[0]);
SIMD_SUBPS(vec30_0, vec10_0, vec50_0);
for(LU_IND0=LU_IB00; LU_IND0<=LU_NUB00-1; LU_IND0+=4) {
  //PIPS:SAC generated v4sf vector(s)
  SIMD_LOAD_V4SF(vec20_0, &src1[LU_IND0]);
  SIMD_MULPS(vec00_0, vec10_0, vec20_0);
  SIMD_LOAD_V4SF(vec80_0, &src2[LU_IND0]);
  SIMD_MULPS(vec60_0, vec30_0, vec80_0);
  SIMD_ADDPS(vec90_0, vec00_0, vec60_0);
  SIMD_STORE_V4SF(vec90_0, &result[LU_IND0]);
}
```

Figure 6. Alphablending kernel vectorized by SAC

By doing this, we trivially ensure all considered instruction sets are representable in our model, and we postpone to the back-end phase the choice of using SIMD intrinsic or sequential code for the implementation of each instruction of our meta instruction set.

As a running example, we can consider the arithmetic operation that sums two vectors of four floating-point elements. In our instruction set, it is denoted SIMD_ADD_V4SF and has the sequential implementation given in figure 3.

An SSE implementation would result in the macro given in figure 4. Its AltiVec variant would be as given in figure 5.

A key component of the meta instruction set is its sequential implementation. In this implementation, vector types are viewed as regular arrays of appropriate length and intrinsics are viewed as function operating of those arrays.

Providing the vectorization step only generates pure C code containing calls to these sequential implementations, generated code is still valid C source that can be dumped into a file (thanks to the source-to-source aspect) and compiled in a regular binary. Such a dump is shown in figure 6, which shows the output of SAC after vectorization of an alphablending kernel with the SSE driver.

The binary will prove to be very inefficient, but is semantically equivalent to the original file. This equivalence is critical to achieve easy debugging: an error in the vectorization algorithm leads to a

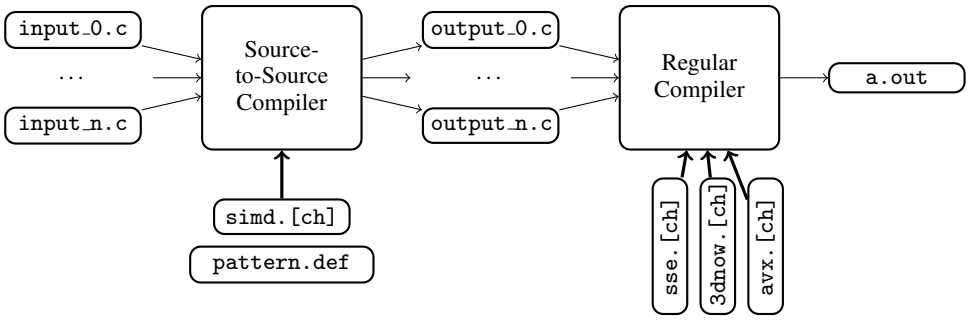

Figure 7. Overview of SAC retargetable compilation scheme

code producing different results (or segfaults...). Regular debugging tools can be used on this code to understand the reason of the failure, which would be almost impossible in a standard compilation infrastructure.[2]

There is another significant benefit of using this sequential implementation: traditional code transformations can be directly reused on the code and can optimize it: dead code elimination, invariant code motion among others can still be used to optimize the generated vector code, because it is actually a sequential code. It however assumes that the considered compiler is capable of producing accurate interprocedural array analysis, which is the case of the PIPS compiler infrastructure used by the authors to produce results exposed in this paper.

3. Compilation Steps

3.1 Overview

To enforce ease of reuse and retargetability, we proposed to adopt a source-to-source approach. Figure 7 should give a practical overview of the basic blocks involved in the building of our compiler.

The inputs are raw `C99` source codes, and an additional `simd.c` `C` source file that provides to the compilation infrastructure the sequential version of the meta instruction set needed for its interprocedural analysis. Instruction patterns are described in a separate definition file named `pattern.def`.

Depending on the target architecture, some patterns would be activated or not. To follow the example of § 2, `SIMD_ADD_V4SF` would be activated for both `SSE` and `AltiVec` but would not for `MMX` or `3Dnow!`, despite the later would have the equivalent operation over two floating points elements, `SIMD_ADD_V2SF`. The importance of the pattern file is discussed in following section.

The code generation process involves the inclusion of a target-specific header (e.g. `sse.h`) and the linkage with a target specific source (e.g. `sse.c`) that provides a specific implementation of the meta instruction set.

An advantage over other approaches is that both the meta-instruction set and its various implementations are provided as raw `C` code, without any annotation or specific formalism. Next section describes in detail the compilation step.

3.2 Compilation Steps

The aim of the algorithm presented in Algorithm 1 is to generate efficient code for the meta instruction set introduced in § 2. In order to do so, we combine an enhanced SLP algorithm described in § 3.3 with traditional compilation phases. Our approach distinguishes from other papers by the intensive reuse it makes of existing transformations: the choice of representing vector register as standard arrays of appropriate type and size makes generated code analyzable by the compiler infrastructure: it only calls regular `C` function operating on fixed-size array. As a consequence, optimization phases can be used further to perform various optimizations without requiring extension or rewriting of those transformations. For example, the SLP algorithm always generates a store statement after a SIMD instruction, and the interprocedural, region-based *dead code elimination* phase takes care of eliminating unneeded statements.

Data: *register_width* ← width of vector register
Data: *prog* ← whole program
Result: vectorized program
for *function* $f \in prog$ **do**
  if_conversion(f) ([2]);
  three_address_code_generation(f);
  reduction_parallelization(f) ([16]);
  for *innermost loop* $l \in f$ **do**
    unroll(f,l,*vec_size*);
  end
  for *statement block* $b \in f$ **do**
    scalar_renaming(f,b) ([1]);
    instruction_selection_for_slp(f,b,*register_width*) (§ 3.3);
  end
  dead_code_elimination(f);
  redundant_load_store_elimination(f) ([7]);
  specialization for target (§ 4);
  register allocation & more via compiler call ;
end

Algorithm 1: SAC compilation algorithm

Indeed the key component of the algorithm is the parameterized SLP algorithm described in next subsection.

3.3 Parameterized SLP Code Generation

In [17], Larsen et al. introduced the concept of superword level parallelism, an alternative to loop-based vectorization that combines dependency analysis and pattern matching to generate multimedia instructions. Since then, this approach has been extended to work beyond basic blocks [23] or too cooperate with traditional loop-based approach [21].

We use a parameterized version of this algorithm described in algorithm 2, that achieves the same level of performance and provides good retargetability properties. All the configuration is driven by two parameters:

1. the width of registers targeted by selected instruction set ;
2. a pattern file that describes the pattern characterizing selected instruction set.

[2] Indeed this has proved to be quite useful for debugging SAC itself.

It takes a block of statements as input and generates a block of statements, eventually packed in call to our meta-instruction set. There is nothing specific to an instruction set in this algorithm, except the register width and the pattern file.

Data: *register_width* ← width of vector register
Data: *pattern* ← set of pattern characterizing the instruction set
Data: b ← list of statements
Result: list of potentially vectorized statements
visited ← ∅;
new_b ← ∅;
live_registers ← ∅;
while $b \neq \varnothing$ **do**
  s ← head(b);
  if $s \notin$ visited **then**
    visited ← *visited* ∪{s};
    if match*(s,pattern)* **then**
      non_conflicting ← extract_no_conflict(tail(b),s);
      isomorphics_statements ←
      extract_isomorphics(*non_conflicting*,s);
      if *isomorphics_statements* ≠ ∅ **then**
        simd_s ←
        select_best(*isomorphics_statements*,
          s,*register_width*);
        load_s ← generate_load_statements(
          simd_s,*live_registers*);
        store_s ← generate_store_statements(
          simd_s,*live_registers*);
        update_live_registers(*simd_s*,*live_registers*);
        append *load_s* to *new_b*;
        append *simd_s* to *new_b*;
        append *store_s* to *new_b*;
        for $s' \in$ *simd_s* **do**
          visited ← *visited* ∪{s'};
        end
      else
        append s to *new_b*
        update_live_registers(s,*live_registers*);
      end
    else
      append s to *new_b*
      update_live_registers(s,*live_registers*);
    end
  end
  b ← tail(b);
end

Algorithm 2: Parameterized SLP algorithm

Algorithm 2 uses several subroutines that are detailed in following subsections.

3.4 Padding Aware Pattern Matching

`match(`*`statement`* `s,`*`pattern_file`* `p)` checks whether statement **s** matches any pattern in **p**. The originality of the pattern matching algorithms is that it can generate padding instructions to create vector instructions.

Let us consider code fragment from figure 8. It is not vectorizable in SSE because it lacks an instruction. Our pattern matching algorithm recognizes such situations and generates an extra instruction that does not alter the control flow but makes vectorization possible. In the case presented in figure 8, the generated instruction would be the one given in figure 9.

This make vectorization possible to the expense of a conditional store. However, it introduces a read and a write to potentially

```
int a[8],b[3];
a[0]=a[0]+b[0];
a[1]=a[1]+b[1];
a[2]=a[2]+b[2];
```

Figure 8. Not vectorizable sequence

```
int a[8],b[3];
a[0]=a[0]+b[0];
a[1]=a[1]+b[1];
a[2]=a[2]+b[2];
a[3]=true?a[3]:a[3]+b[3];
```

Figure 9. Padded instruction sequence

```
union {
  int array[3];
  int padding[4];
} b;
```

Figure 10. Padded array declaration

unallocated memory. In the presented example, `a[3]` is a valid location, but `b[3]` is not. As a consequence, each variable is padded to ensure no illegal memory access is performed, as shown in the declaration of `b` in code 10.

This optimization step enables significant improvements in the vectorization of `convolution` kernels, as show in the experiments § 5.
`extract_isomorphics(`*`statement set`* `l,`*`statement`* `s)` builds a set of statement from **l** that matches the same pattern as **s**).

3.5 Dependency Graph Analysis

`extract_no_conflict(`*`statement list`* `l,`*`statement`* `s)` builds a set of statement from **l** that have no R-W,W-R,W-W, dependence arc with **s**. It is based on the dependency graph based itself on the inter-procedural array region analysis [9] implemented in PIPS. Because we model vector register by regular `C` arrays, this array region analysis is critical.

3.6 Greedy Statement Packing

`select_best(`*`statement set`* `l,`*`statement`* `s,`*`int`* `register_width)` chooses **register_width** -1 statements among **l** and pack with **s**. This routine uses a greedy algorithm to choose the best candidate among elements of **l** and is described in algorithm 3.

A matrix of offsets between arguments of statement **s** and arguments of statements of **l** is computed and then greedily searched for consecutive accesses. If no consecutive access is found, the first statements (according to lexical order) are returned.

3.7 Load Store Generation

`generate_load_statements(`*`statement list`* `l,`*`set`* `live_registers)` generates data transfers from the scalar registers to the vector registers. For each data needed by statements in **l**, it checks if they are already present in **live_registers**. If so, it uses the according vector, otherwise it generates a load from the memory to a new vector and registers this vector in **live_registers**. As of now, the underlying algorithm does not generate `shuffle` operations, but it is an interesting option.
`generate_store_statements(`*`statement list`* `l,`*`set`* `live_registers)` generates data transfers from the vector registers back to the scalar registers. It builds a list of store statement,

Data: $l \leftarrow$ statement list
Data: $s \leftarrow$ initial statement
Data: *register_width* $\leftarrow$ width of vector registers
Result: packed statement, or unpacked statement
$[a_0, \ldots, a_k] \leftarrow$ arguments of s;
offsets $\leftarrow \varnothing$;
for *statement* $s' \in l$ **do**
  $[a'_0, \ldots, a'_k] \leftarrow$ arguments of s';
  append $(s', [\mathit{offset}(a'_0, a_0), \ldots, \mathit{offset}(a'_k, a_k)])$ to *offsets*;
end
sort *offsets* based on the offset key;
packed_statements $\leftarrow \varnothing$;
for arg_id $\in [|1, \ldots, \frac{register\_width}{k}|]$ **do**
  packing_index $\leftarrow 1$;
  for $(s'$,*offset*$) \in$ *offsets* **do**
    if offset_{arg_id} =*packing_index* **then**
      append s' to *packed_statements*;
      packing_index $\leftarrow 1+$ *packing_index*;
    end
  end
  if length*(packed_statements)* $\neq \frac{register\_width}{k}$ **then**
    empty *packed_statements*;
    break;
  end
end
if *packed_statements* $= \varnothing$ **then**
  return $[s] + l[0 : \frac{register\_width}{k} - 1]$
else
  return *packed_statements*
end

Algorithm 3: Offset-based statement packing

one for each array pack generated by statements in **l**. Those statements may be useless, in which case they will be removed by the *dead code elimination* phase. `update_live_registers` is called for each statement of **l**
`update_live_registers(`*`statement`* `s, set live_registers)` updates **live_registers** by removing each vector that used to reference a variable written by **s**.

4. Achieving Retargetability

We have already seen in § 2 how we provide a sequential backend to our compilation infrastructure. The following subsections detail how we specialize this backend for a particular instruction set. Two specialized backends are already available : an SSE implementation, written using Intel's SSE4 intrinsics ; and a 3DNow! implementation, using (obsolete) AMD's 3DNow! intrinsics.

4.1 Supporting a New Instruction Set

As we saw in § 2, adding a new backend is mainly a matter of translating the pseudo-instruction set generated by SAC into the intrinsics of the target processor, and making this translation available to the compiler.

While the core of SAC transformations are implemented in C, the preferred user interface is PyPS, a Python API to PIPS. The user creates a "workspace", a Python object giving accesses to the "modules" of code, that is, the compilation units and functions of the user's C programs. Each PIPS transformation is made available to the user as a method to be applied to a module.

Utilizing Python's dynamism, we developed a method for the optional, customizable, run-time composition of special kinds of workspaces. In particular, there is a `sac` Python module, defining a

```
import pyps
import sac

ws = pyps.workspace(["file1.c", "file2.c"], parents = [
     sac.workspace], driver = "sse")
ws["my_function"].sac()
ws.simd_compile(outfile = "program")
```

Figure 11. Sample PyPS usage

```
sub: = REFERENCE - REFERENCE REFERENCE ;
sub: = REFERENCE - REFERENCE CONSTANT ;
sub: = REFERENCE - CONSTANT REFERENCE ;
sub: = REFERENCE + UNARY_MINUS REFERENCE CONSTANT ;
sub: = REFERENCE + CONSTANT UNARY_MINUS REFERENCE ;
sub: = REFERENCE + REFERENCE UNARY_MINUS REFERENCE ;
```

Figure 12. Description of subtraction pattern in SAC

`sac.workspace` class. From the user point of view, it is used like in figure 11.

In the `__init__` method of `pyps.workspace`, a `sac.workspace` object is created. When creating this object, some code is injected into the `pyps.workspace` class, in particular it adds a `.sac()` method to each module, and some code to the `compile` command. This allows us to add to each compilation unit a set of definitions, which are chosen depending on the `driver` parameter used on the creation of the workspace.

Inside the `sac` module, each backend is defined by a `sac`*`TARGET`* (`sacsse` or `sac3dnow` or...) class, that inherits from `sacbase`. The body of the `.sac()` method added to each module is defined in the `sacbase.sac()` function, with `sac`*`TARGET`*`.sac()` being only a small wrapper around it, specifying the register width (128 bits for SSE and 64 bits for 3DNow!, for instance).

This technique is both convenient for the user, and easy to extend. If another backend is to be added, say for AVX, one would do the following:

- create a `sacavx` class inheriting from `sacbase` ;
- define a `sacavx.sac()` method that just calls `sacbase.sac()` with a register width of 256 ;
- define a `sacavx.CFLAGS` variable, which would probably be `-mavx -O3` or similar ;
- write a series of `#define` implementing the SIMD pseudo-language.

The last part is the longest, whereas the first parts are rather straightforwards. In addition, the `call_sac.py` script, whose first purpose was to show an example of the usage of the `sac.workspace` module, is also able to compare the output of programs when compiled with the sequential implementation against the output when using some other instruction set.

4.2 Supporting New Instructions

As shown in § 3.3, our vectorization algorithm is parameterized by a pattern file that feeds the pattern engine. The format of this file use polish notation to describe the pattern of a single instruction. As a consequence it only supports patterns where the same instruction appears $register\_width$ times, thus by-passing non-SIMD operators such as `horizontal add` found in SSE3.

Listing 12 shows the pattern describing the `SIMD_SUB_*` operator:

5. Experiments

5.1 Description of the Experiments

Two series of experiments were conducted. They both used the same set of source files, which are available in the PIPS repository, under the `Passes/pyps/drivers/sac/bench` directory. The first series use the SSE driver, while the second uses the 3DNow! driver.

`daxpy_u?r.c`, `ddot_u?r.c` and `dscal_u?r.c` are taken from the Linpack [11] benchmark. `matrix_*.c` are taken from the Coremark benchmark of the EEMBC suite. Other benchmarks are textbook version of well known computations kernels (Finite Impulse Response filter, average power, alpha-blending, jacobi, convolution with a 3x3 kernel). Raw benchmarks results are given in table 13.

The vertical scale is the speedup of each execution over `gcc` run with compilation flags `-O3 -fno-tree-vectorize -march=native`, that is without vectorization step. The time taken by an execution was measured by wrapping the `main` function of each benchmark between two calls to `gettimeofday(2)` and then computing the elapsed time. The time returned is the median over 50 runs on an otherwise idle machine.

The experiments with the SSE driver were conducted on an Intel Core i5 CPU, with a 2.67GHz frequency. The machine was running a 2.6.32 Linux kernel. In the graphics 14, 15, 16, and 17 we compare the performance of GCC 4.4.4 (series marked with `gcc`), ICC 11.1 (marked with `icc`) and LLVM (through llvm-gcc-4.2, Based on Apple Inc. build 5658, series marked with `llvm`) before and after transforming the code with SAC. The `+seq` variant uses a naïve sequential implementation of SIMD operations, while the `+sac` variant uses the intrinsics of the platform. In each case, the code was compiled with `-O3`. The SSE variant was compiled with `-march=native`.

The experiments with the 3DNow! driver were made on an AMD Opteron Processor 252 with a 2.5GHz clock. The machine was running a 2.6.26 Linux kernel. The code was compiled with GCC 4.3.6, using the flags `-m3dnow -march=opteron -O3`. The results are presented on figure 18.

5.2 Comments

The SIMD simulator is an invaluable tool during the development of SAC, for checking the correctness of the intrinsics implementation. However, it is not of practical use, with some programs running up to 5 times slower.

To the notable exception of the matrix-matrix multiplication, SAC generally improves the behavior of `gcc` by a factor of 5%. However, some applications (`convol3x3.c`, `jacobi.c`) greatly benefits from it, reaching more than 60% speedup, which shows the efficiency of our algorithm in complex situation.

ICC, which is well-known for generating very efficient code, does not benefit from SAC to the same extent. In various situation, a small speedup is achieved (e.g. `average_power.c`), while in other `icc` outperforms SAC by several order of magnitude (e.g. `ddot_ur.c`.

Vectorization engine of `llvm` proves to be less efficient than others, and is the one that benefits the most of SAC vectorization engine. `jacobi.c` and `convol3x3.c` achieve very good speedup.

The production of 3DNow! code came rather late in the development cycle of SAC and is more a proof of concept than a significant experiment (3DNow! is no longer in production), but it shows the retargability of SAC: 3DNow! backend is only several hundred of `C` code describing the implementation of a subset of the meta-instrucion set.

Either way, the source-to-source approach proved to be beneficial: existing vectorizer are leveraged by the SAC compiler, but speedup greatly depends upon the back-end compiler, which asserts the idea that SAC is to be used as a generic vectorized, leaving to the real compiler the low-level optimizations.

Conclusion & Future Works

As many developers know, making a code portable is not easy. Making performance portable is all the more difficult, because efficient code relies on the usage of specific hardware. It is up to the compiler to fill the gap, and to support ever-changing architectures they are forced to be as retargetable as possible. In this paper, we described the SAC compiler based on the PIPS source-to-source compiler framework that achieves this objective for multimedia instruction set while still generating efficient code. Thanks to a meta-instruction set and parameterized SLP algorithm, this compiler achieves performance within the range of `gcc`, `llvm` and `icc` while remaining fully retargetable. A source-to-source approach and a dedicated SLP algorithm made this possible, and we believe that more performance / specialization can be achieved through the use of an exploratory algorithm, capable of selecting the transformation better-suited to a specific architecture at runtime. Generating shuffle instructions is also a challenging issue, because such instructions vary a lot across instruction sets. An interesting addition to the SAC compiler would be the ability to parse existing code written in a particular instruction set, to decompile it and to recompile it in another instruction set, possibly of different register width. We have made a step forward in that direction by implementing `mmx` header, `pmmintrin.h` with sequential alternative, so that including our header instead of the system header results in generation of generic code. However, this also results in complex code filled with `union` and structure copies, which is not analyzable by PIPS as of now. We plan to use our portable vectorizer to generate vector type code for CUDA and OpenCL to get better performance on GPU. In this case, the instruction set is directly the C code, without almost no intrinsics, just relying on the CUDA and OpenCL vector types.

Acknowledgments

This work has been funded by through the project. Many people contributed to the development of the SAC compiler: tribute must be returned to and for their previous work. Indirect work of and on the scripting capabilities of PIPS. Work of to extend array region analysis to the `C` language proved to be priceless.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *POPL*, pages 177–189, 1983.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[4] A. J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004. ISBN 0974364924.

[5] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic detection of saturation and clipping idioms. In *LCPC*, pages 61–74, 2002.

[6] R. L. Bocchino, Jr. and V. S. Adve. Vector llva: a virtual vector instruction set for media processing. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-6.

Compilation	gcc-ref	gcc	icc	llvm	gcc+seq	gcc+sac	icc+sac	llvm+sac
daxpy_r.c	1933734	1935851	1941974	1932617	1941856	1932959	**1929604**	1938193
daxpy_ur.c	**1931779**	1932338	1932753	1945305	1948584	1934827	1933631	1937369
ddot_r.c	17793	19386	**12191**	21101	25490	18924	18560	18435
ddot_ur.c	19305	19257	**16635**	19316	26037	17623	17227	18074
dscal_r.c	1934692	1944013	**1923703**	1943200	1938208	1930294	1928215	1929183
dscal_ur.c	1935732	1932348	1925334	1933790	1940586	1928948	**1922907**	1923074
matrix_add_const.c	7484	7503	**6821**	7913	11037	7865	7535	7709
matrix_mul_const.c	7560	7598	7429	7938	10544	7304	**6970**	7162
matrix_mul_vect.c	5213	5247	4836	6911	9453	4761	4615	**4319**
matrix_mul_matrix.c	4173621	4202763	**3976309**	4400926	12115274	4744732	4537512	4542691
alphablending.c	110085	110017	90327	110503	257956	92103	**84904**	85918
average_power.c	153171	154508	140225	173350	276630	153949	**135584**	143839
convol3x3.c	18361	18097	15140	18132	39146	**5146**	5242	10815
fir.c	1191050	1190567	**585705**	3429364	6022014	747824	736276	827715
jacobi.c	757998	757962	**146916**	761945	1122687	282890	288375	283137

Figure 13. Raw benchmarks of `gcc`, `icc`, `llvm` and SAC behavior. Timings are given in μs.

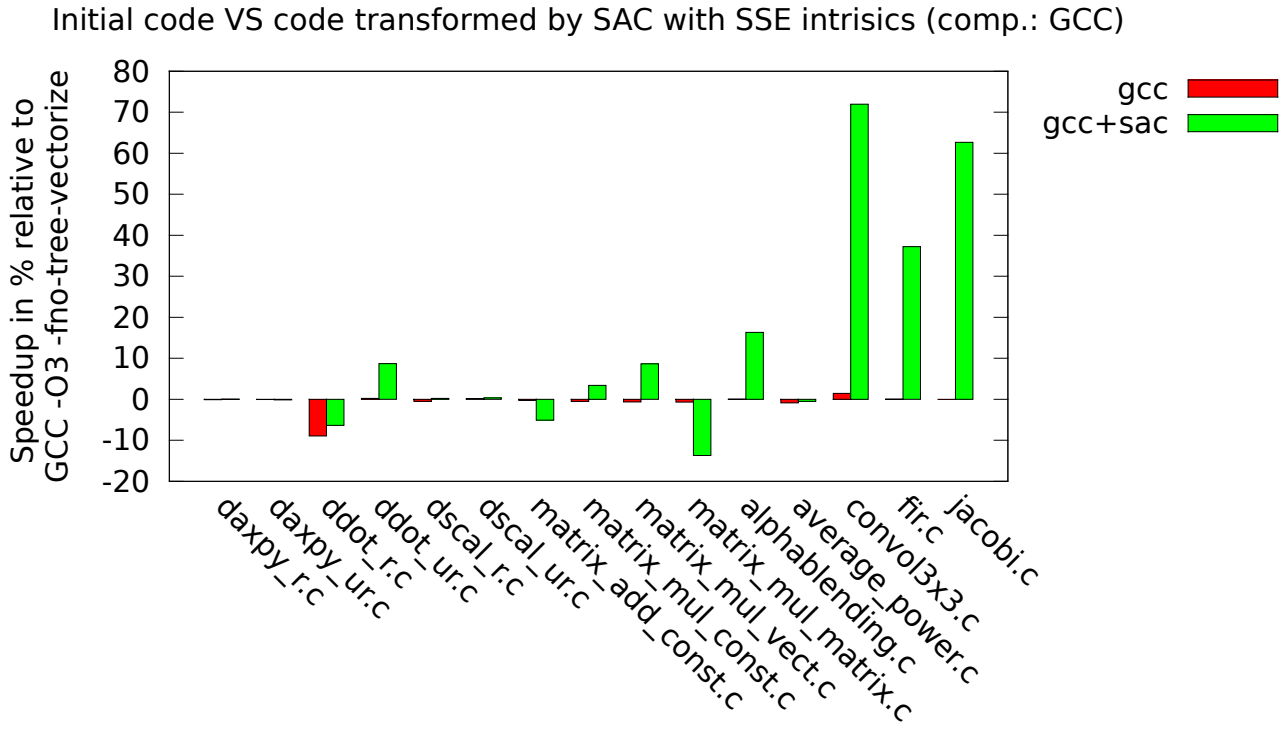

Figure 14. Speedup before and after SAC using GCC. Positive means SAC is faster.

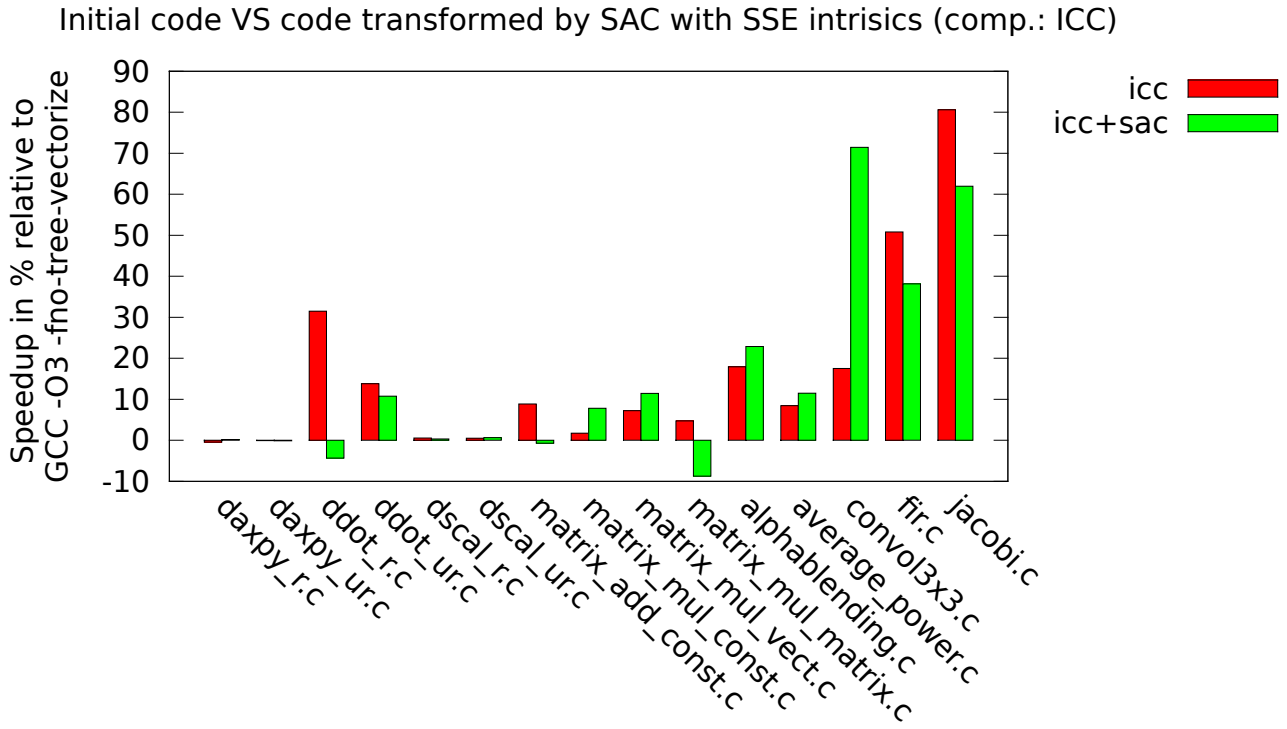

Figure 15. Speedup before and after SAC, using ICC. Positive means SAC is faster.

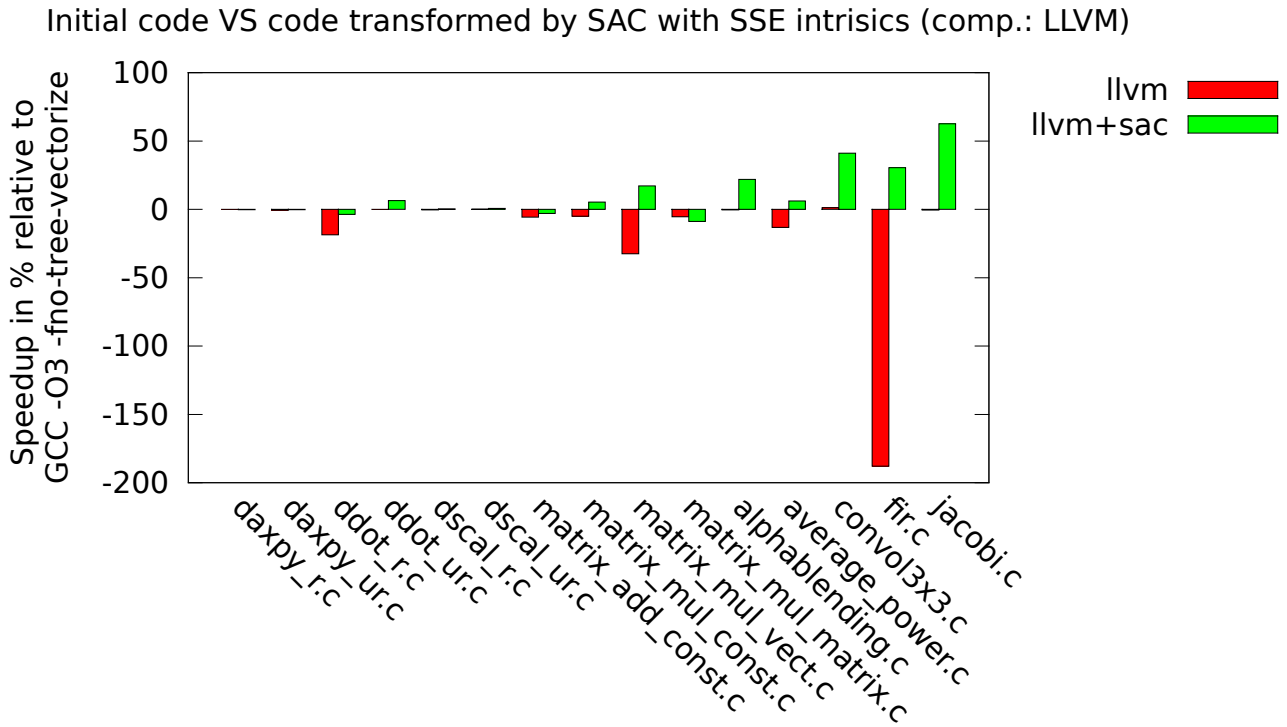

Figure 16. Speedup before and after SAC, using LLVM. Positive means SAC is faster.

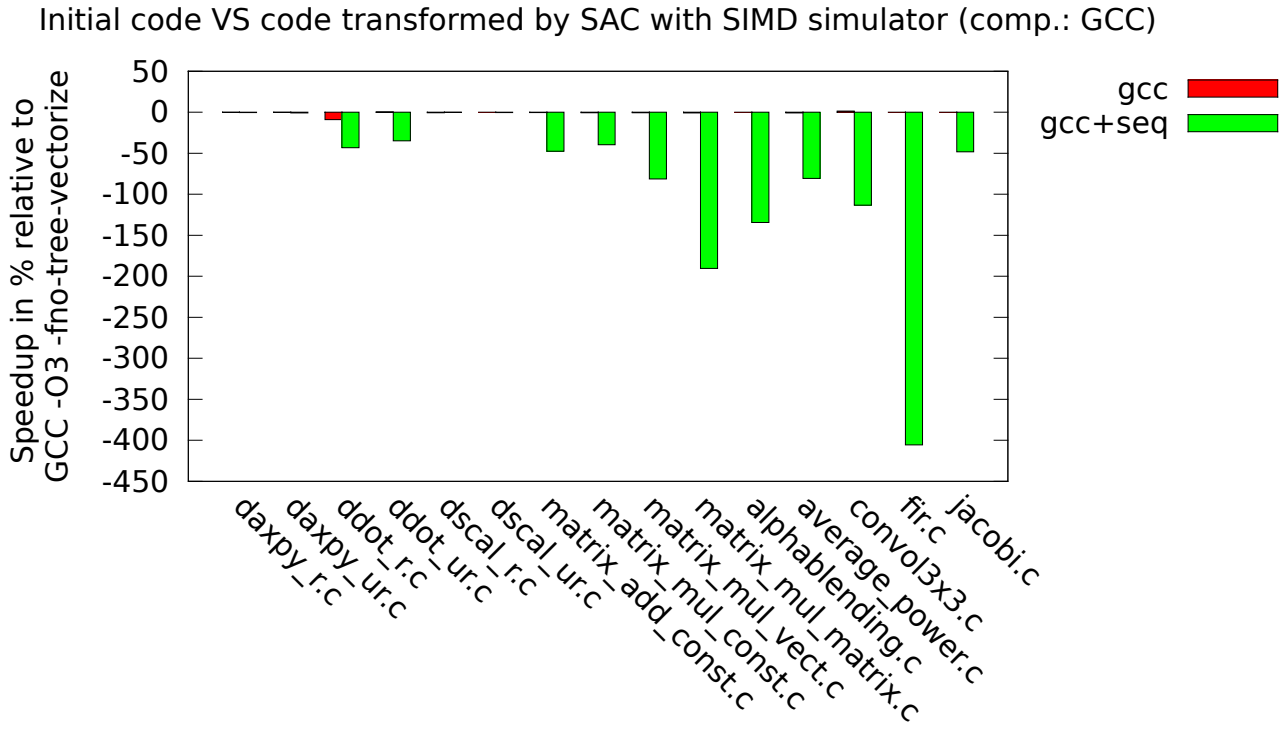

Figure 17. Speed-down before and after SAC, using the SIMD emulator

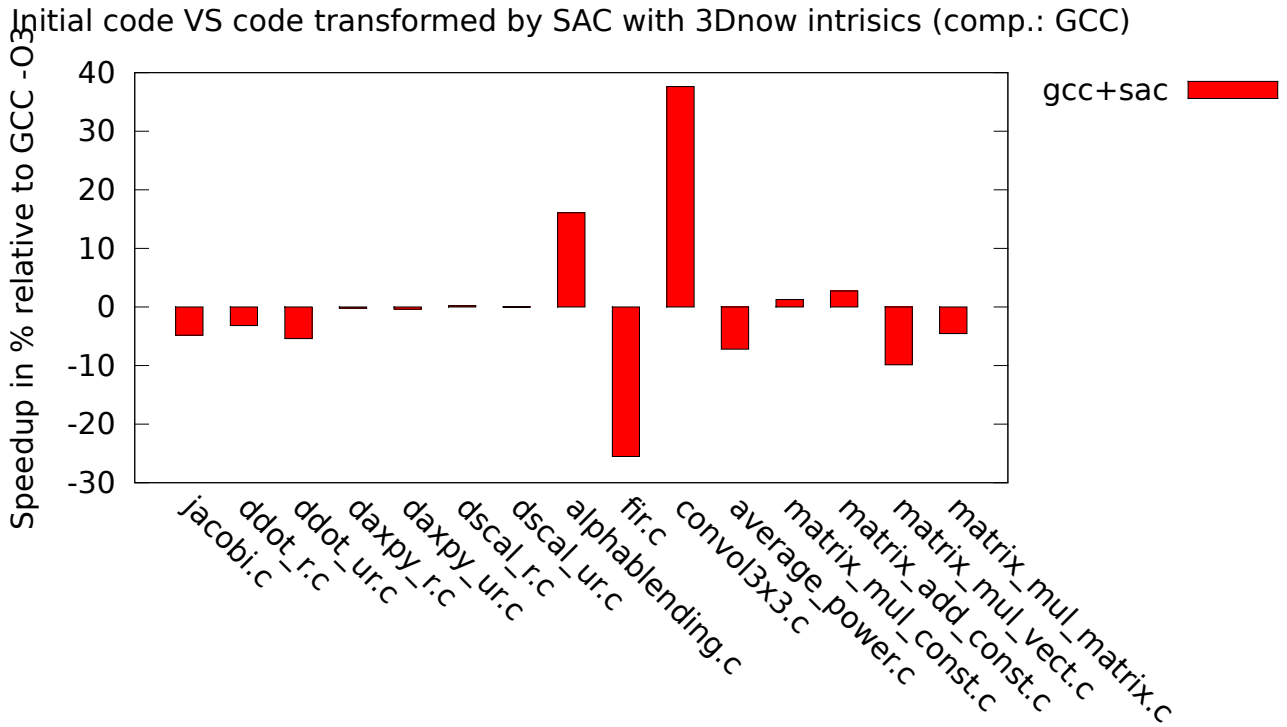

Figure 18. Comparison of SAC performance using the 3DNow! driver

[7] R. Bodík and R. Gupta. Array data flow analysis for load-store optimizations in superscalar architectures. In *LCPC*, pages 1–15, 1995.

[8] K.-H. Chen, B.-Y. Shen, and W. Yang. An automatic superword vectorization in LLVM. In *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing*, pages 19–27, Taipei, 2010.

[9] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. J. Parallel Program.*, 24(6):513–546, 1996. ISSN 0885-7458.

[10] . PIPS, 1989–2010. Open source, under GPLv3.

[11] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[12] R. J. Fisher and H. G. Dietz. Compiling for SIMD within a register. In *LCPC '98: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 290–304, London, UK, 1999. Springer-Verlag. ISBN 3-540-66426-2.

[13] B. J. Gough and R. M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004. ISBN 0954161793.

[14] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A SIMD optimization framework for retargetable compilers. *TACO*, 6 (1), 2009.

[15] W. Jiang, C. Mei, B. Huang, J. Li, J. Zhu, B. Zang, and C. Zhu. Boosting the performance of multimedia applications using SIMD instructions. In *CC*, pages 59–75, 2005.

[16] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *ICS*, pages 186–194, 1989.

[17] S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 145–156, 2000.

[18] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[19] . PIPS Tutorial at PPoPP 2010.

[20] G. Pokam, S. Bihan, J. Simonnet, and F. Bodin. SWARP: a retargetable preprocessor for multimedia instructions. *Concurrency and Computation: Practice and Experience*, 16(2-3):303–318, 2004.

[21] I. Rosen, D. Nuzman, and A. Zaks. Loop-aware SLP in GCC - two years later. In *GCC summit*, July 2007.

[22] J. Shin, J. Chame, and M. W. Hall. Exploiting superword-level locality in multimedia extension architectures. *J. Instruction-Level Parallelism*, 5, 2003.

[23] J. Shin, M. W. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO*, pages 165–175, 2005.

[24] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. *Parallel Architectures and Compilation Techniques, International Conference on*, 0: 327–337, 2009. ISSN 1089-795X.