Closing the Performance Gap Between Lisp and C

Marco Heisig
Chair for System Simulation
FAU Erlangen-Nürnberg
Erlangen, Germany
marco.heisig@fau.de

Harald Köstler
Chair for System Simulation
FAU Erlangen-Nürnberg
Erlangen, Germany
harald.koestler@fau.de

ABSTRACT

Lisp is the second oldest programming language, and the first one to value productivity more than raw execution speed. This initial disregard for performance has indeed led to some mind-bogglingly slow implementations, especially in the early days, but modern Lisp compilers such as SBCL have almost fully closed the performance gap to the fastest imperative programming languages. Almost, but not quite: Until now, many loop optimizations and support for single instruction, multiple data (SIMD) programming are still missing in Lisp.

We correct this deficiency with two libraries: The first one is sb-simd, an SBCL-specific library that provides convenient bindings for various SIMD instructions sets. The second one is Loopus, a portable loop optimization framework that works via macros and source to source transformations. The most prominent features of Loopus are its optimization of array accesses and that, on SBCL, it automatically applies SIMD vectorization to certain loops.

We conclude with a performance evaluation for several example programs, and show that Lisp code using our libraries can achieve up to 94% of the performance of highly optimized C code.

ACM Reference Format:
Marco Heisig and Harald Köstler. 2022. Closing the Performance Gap Between Lisp and C. In *Proceedings of the 15th European Lisp Symposium (ELS'22).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/zenodo.6335627

1 INTRODUCTION

A common misconception about Moore's law is that it promises a doubling of the performance of our computers roughly every two years. In fact, Gordon Moore was predicting that the number of transistors in an integrated circuit would double roughly every two years. This fine distinction between performance and number of transistors was hardly relevant for a long time, where hardware manufacturers managed to translate more transistors directly into more performance. Unfortunately, the last two decades show that this particular free lunch is now over, and that additional performance can only be gained in combination with additional complexity. We see chips having multiple cores, multiple levels of caches, and larger, more specialized instruction sets.

In terms of programming models for multiple cores, pretty much all programming languages are sitting in the same boat. Programmers are compelled to use multiple threads, and some of the dozens of possible synchronization and communication primitives for distributing and coordinating work. The more interesting challenge is to increase the performance of a single core. In that case, and assuming memory bandwidth and latency is not an issue, the performance can only be improved by using more powerful instructions. The most important ones for doing so are SIMD instructions, where a single instruction can perform an operation simultaneously on all elements of a small, specialized vector. This paper is about using SIMD instructions in Common Lisp.

There are two challenges when incorporating SIMD instructions into a programming language. The first challenge is to seamlessly integrate the low-level functionality for manipulating the 128 bit, 256 bit, or even 512 bit wide SIMD packs offered by the hardware. The second challenge is to provide the programmer with tools that can automatically convert scalar code to efficient SIMD code, at least for the most frequently occurring cases. Without such a tool for automatic conversion, SIMD programming is unnecessarily error-prone and cumbersome, and will likely only be used by a small group of extremely dedicated programmers.

In this paper we present two libraries. The first one is sb-simd[1], a SBCL-specific library that provides convenient bindings for various SIMD instructions sets. The second one is Loopus[2], a portable loop optimization framework that works via macros and source to source transformations, and that automatically turns many kinds of scalar loops into more efficient versions using SIMD instructions. We conclude with a performance comparison of Common Lisp code using Loopus, and C code using the most recent version of GCC.

2 RELATED WORK

Programs written in Common Lisp are frequently among the fastest in scenarios that benefit from extensive metaprogramming, or from incremental compilation. Examples of this are Paul Graham's macro for generating Bézier curves[2], Breanndán Ó Nualláin's DSL for graph algorithms[5], or Børge Svingen's use of on-the-fly compilation for genetic programming[3]. These projects achieve high performance by exploiting cases where Lisp has an inherent advantage. Our work differs from this in that we want to achieve high performance in cases where Lisp has no inherent advantage, such as image processing or number crunching.

The most recent publication comparing the performance of Lisp and C for a case where Lisp has no inherent advantage is from Didier Verna and was published in 2006[4]. In that paper, the author shows how Lisp can reach and sometimes even exceed the performance of

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'22, March 21–22, 2022, Porto, Portugal
© 2022 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.6335627

[1]https://github.com/marcoheisig/sb-simd
[2]https://github.com/marcoheisig/Loopus

C for several simple image manipulation tasks. However, neither the C code nor the Lisp code in that paper use SIMD instructions. All recent C compilers will use SIMD instructions for such tasks, so the only way to catch up once more is to use SIMD instructions in Lisp, too.

Apart from the paper of Verna, there is also the seminal, although somewhat dated, book about the performance and evaluation of Lisp Systems by Richard P. Gabriel[1], that discusses the performance of various Lisp implementations in general. From today's perspective, the most important information therein is the discussion of the various tradeoffs being made when implementing Lisp and their implications for performance.

3 SB-SIMD

The library sb-simd allows Common Lisp code to utilize SIMD instructions. In contrast to most other SIMD interfaces, e.g., C intrinsics, sb-simd incorporates the usual conveniences of Common Lisp. Each instruction set has its own package, and instruction set inheritance is modeled by re-exporting the symbols of each parent instruction set. All SIMD data types have their own built-in classes that can be queried and specialized upon. All functions automatically coerce all supplied arguments to the correct type, broadcast them to the correct SIMD width, and, when the underlying operator allows it, accept any number of arguments.

Luckily, we didn't have to trade convenience against speed when designing the public interface of sb-simd. SBCL's type inference and compiler are powerful enough to eliminate the overhead of dynamic typing, implicit conversions, broadcasts, and variadic arguments where necessary. Most calls to functions in sb-simd are compiled to a single machine instruction.

3.1 Software Architecture

The biggest challenge in writing this library was the sheer number of SIMD instructions available on a modern machine. The AVX instruction set alone provides almost 1000 instructions, and more than the number of functions in the CL package!

The solution we came up with is, perhaps unsurprisingly, Lisp macros. We first create a table of metadata for each instruction set, which contains information about all data types, functions, corresponding mnemonics, mathematical properties, and so forth. Then we use the data in these tables to generate all the types, declarations, compiler transformations, machine code emitters, and functions, that are specified in each table. Not a single line of the code that is invoked when calling a function in sb-simd is written by hand. This is not just an eccentricity. By generating all code, we ensure that each bug breaks all functions simultaneously, which is much easier to detect and to fix.

All of the tables that are used to generate the functions and data structures in sb-simd can also be queried at run time. This makes it possible to look up the supported instruction sets, the functions exported by each instruction set, the arguments and return types of each function and much more. Our loop optimization library Loopus uses this metadata to perform automatic vectorization.

Something we didn't manage, unfortunately, is to make our SIMD interface portable across multiple Lisp implementations. The $k \times n$ interactions in code that runs on k implementations and n architectures turned out to be too big of a headache. At some point we decided that it is better to have a high-quality SIMD interface for one Lisp implementation, than a mediocre one that is portable.

3.2 Data Types

The central data type in sb-simd is the SIMD pack. A SIMD pack is very similar to a specialized vector, except that its length must be a particular power of two that depends on its element type and the underlying hardware. The set of element types that are supported for SIMD packs is similar to that of SBCL's specialized array element types, except that there is currently no support for SIMD packs of complex numbers or characters.

The list of supported scalar types is shown in Figure 1. For each scalar data type `X`, there exists one or more SIMD data type `X.Y` with `Y` elements. For example, in AVX there are two supported SIMD data types with element type `f64`, namely `f64.2` (128bit) and `f64.4` (256bit).

sb-simd	Common Lisp	
f32	single-float	
f64	double-float	
sN	(signed-byte N)	$N \in \{8, 16, 32, 64\}$
uN	(unsigned-byte N)	

Figure 1: Scalar data types in sb-simd and their corresponding Common Lisp type specifiers.

SIMD packs are regular Common Lisp objects that have a type, a class, and can be passed as function arguments. The price for this is that SIMD packs have both a boxed and an unboxed representation. The unboxed representation of a SIMD pack has zero overhead and fits into a CPU register, but can only be used within a function and when the compiler can statically determine the SIMD pack's type. Otherwise, the SIMD pack is boxed, i.e., spilled to the heap together with its type information. In practice, boxing of SIMD packs can usually be avoided via inlining, or by writing their values to specialized arrays (see section 3.11) instead of passing them around as function arguments.

3.3 Casts

For each scalar data type `X`, there is a function named `X` that is equivalent to `(lambda (v) (coerce v 'X))`. For each SIMD data type `X.Y`, there is a function named `X.Y` that ensures that its argument is of type `X.Y`, or, if the argument is a number, calls the cast function of `X` and broadcasts the result.

All functions provided by sb-simd (apart from the casts themselves) implicitly cast each argument to its expected type. So to add the number five to each single float in a SIMD pack `x` of type `f32.8`, it is sufficient to write `(f32.8+ x 5)`. We don't mention this implicit conversion explicitly in the following sections, so if any function description states that an argument must be of type `X.Y`, the argument can actually be of any type that is a suitable argument of the cast function named `X.Y`.

3.4 Constructors

For each SIMD data type `X.Y`, there is a constructor named `make-X.Y` that takes `Y` arguments of type `X` and returns a SIMD pack whose elements are the supplied values.

3.5 Unpackers

For each SIMD data type `X.Y`, there is a function named `X.Y-values` that returns, as `Y` multiple values, the elements of the supplied SIMD pack of type `X.Y`.

3.6 Reinterpret Casts

For each SIMD data type `X.Y`, there is a function named `X.Y!` that takes any SIMD pack or scalar datum and interprets its bits as a SIMD pack of type `X.Y`. If the supplied datum has more bits than the resulting value, the excess bits are discarded. If the supplied datum has less bits than the resulting value, the missing bits are assumed to be zero.

3.7 Associatives

For each associative binary function, e.g., `two-arg-X.Y-OP`, there is a function `X.Y-OP` that takes any number of arguments and combines them with this binary function in a tree-like fashion. If the binary function has an identity element, it is possible to call the function with zero arguments, in which case the identity element is returned. If there is no identity element, the function must receive at least one argument.

Examples of associative functions are `f32.8+`, for summing any number of 256 bit packs of single floats, and `u8.32-max`, for computing the element-wise maximum of one or more 256 bit packs of 8 bit integers.

3.8 Reducers

For binary functions `two-arg-X.Y-OP` that are not associative, but that have a neutral element, we provide functions `X.Y-OP` that take any positive number of arguments and return the reduction of all arguments with the binary function. In the special case of a single supplied argument, the binary function is invoked on the neutral element and that argument. Reducers have been introduced to generate Lisp-style subtraction and division functions.

Examples of reducers are `f32.8/`, for successively dividing a pack of 32 bit single floats by all further supplied packs of 32 bit single floats, or `u32.8-` for subtracting any number of supplied packs of 32 bit unsigned integers from the first supplied one, except in the case of a single argument, where `u32.8-` simply negates all values in the pack.

3.9 Comparisons

For each SIMD data type `X.Y`, there exist conversion functions `X.Y<`, `X.Y<=`, `X.Y>`, `X.Y>=`, and `X.Y=` that check whether the supplied arguments are strictly monotonically increasing, monotonically increasing, strictly monotonically decreasing, monotonically decreasing, equal, or nowhere equal, respectively. In contrast to the Common Lisp functions <, <=, >, >=, =, and `/=` the SIMD comparison functions don't return a generalized boolean, but a SIMD pack of unsigned integers with `Y` elements. The bits of each unsigned integer are either all one, if the values of the arguments at that position satisfy the test, or all zero, if they don't. We call a SIMD packs of such unsigned integers a *mask*.

3.10 Conditionals

The SIMD paradigm is inherently incompatible with fine-grained control flow. A piece of code containing an `if` special form cannot be vectorized in a straightforward way, because doing so would require as many instruction pointers and processor states as there are values in the desired SIMD data type. Instead, most SIMD instruction sets provide an operator for selecting values from one of two supplied SIMD packs based on a mask. The mask is a SIMD pack with as many elements as the other two arguments, but whose elements are unsigned integers whose bits must be either all zeros or all ones. This selection mechanism can be used to emulate the effect of an `if` special form, at the price that both operands have to be computed each time.

In sb-simd, all conditional operations and comparisons emit suitable mask fields, and there is a `X.Y-if` function for each SIMD data type with element type `X` and number of elements `Y` whose first arguments must be a suitable mask, whose second and third argument must be objects that can be converted to the SIMD data type `X.Y`, and that returns a value of type `X.Y` where each element is from the second operand if the corresponding mask bits are set, and from the third operand if the corresponding mask bits are not set. An example of masks and conditionals is given in Figure 3.

3.11 Loads and Stores

In practice, a SIMD pack `X.Y` is usually not constructed by calling its constructor, but by loading `Y` consecutive elements from a specialized array with element type `X`. The functions for doing so are called `X.Y-aref` and `X.Y-row-major-aref`, and have similar semantics as Common Lisp's `aref` and `row-major-aref`. In addition to that, some instruction sets provide the functions `X.Y-non-temporal-aref` and `X.Y-non-temporal-row-major-aref`, for accessing a memory location without loading the referenced values into the CPU's cache.

For each function `X.Y-foo` for loading SIMD packs from an array, there also exists a corresponding function `(setf X.Y-foo)` for storing a SIMD pack in the specified memory location. An exception to this rule is that some instruction sets (e.g., SSE) only provide functions for non-temporal stores, but not for the corresponding non-temporal loads.

One difficulty when treating the data of a Common Lisp array as a SIMD pack is that some hardware instructions require a particular alignment of the address being referenced. Luckily, most architectures provide instructions for unaligned loads and stores that are, at least on modern CPUs, not slower than their aligned equivalents. So by default we translate all array references as unaligned loads and stores. An exception are the instructions for non-temporal loads and stores, that always require a certain alignment. We do not handle this case specially, so without special handling by the user, non-temporal loads and stores will only work on certain array indices that depend on the actual placement of that array in memory. We'd be grateful if someone could point us to a mechanism for constraining the alignment of Common Lisp arrays in memory.

3.12 Specialized Scalar Operations

Finally, for each SIMD function `X.Y-OP` that applies a certain operation `OP` element-wise to the `Y` elements of type `X`, there exists also a functions `X-OP` for applying that operation only to a single element. For example, the SIMD function `f64.4+` has a corresponding function `f64+` that differs from `cl:+` in that it only accepts arguments of type double float, and that it adds its supplied arguments in a fixed order that is the same as the one used by `f64.4`.

There are good reasons for exporting scalar functions from a SIMD library, too. The most obvious one is that they obey the same naming convention and hence make it easier to locate the correct functions. Another benefit is that the semantics of each scalar operation is precisely the same as that of the corresponding SIMD function, so they can be used to write reference implementations for testing. A final reason is that scalar functions can be used to simplify the life of tools for automatic vectorization.

3.13 Instruction Set Dispatch

One challenge that is unique to image-based programming systems such as Lisp is that a program can run on one machine, be dumped as an image, and then resumed on another machine. While nobody expects this feature to work across machines with different architectures, it is quite likely that the machine where the image is dumped and the one where execution is resumed provide different instruction set extensions.

As a practical example, consider a game developer that develops software on an x86-64 machine with all SIMD extensions up to AVX2, but then dumps it as an image and ships it to a customer whose machine only supports SIMD extensions up to SSE2. Ideally, the image should contain multiple optimized versions of all crucial functions, and dynamically select the most appropriate version based on the instruction set extensions that are actually available.

This kind of run time instruction set dispatch is explicitly supported by means of the `instruction-set-case` macro. The code resulting from an invocation of this macro compiles to an efficient jump table whose index is recomputed on each startup of the Lisp image. An simple example of such an instruction set dispatch is given in Figure 2.

```
(defpackage #:sb-simd-user
  (:use #:common-lisp #:sb-simd)
  (:local-nicknames
   (#:sse2 #:sb-simd-sse2)
   (#:avx #:sb-simd-avx2)))

(in-package #:sb-simd-user)

(defun quadruple4 (array)
  (declare (type (simple-array f64 (4)) array))
  (declare (optimize (speed 3) (safety 0)))
  (prog1 array
    (instruction-set-case
      (:avx
       (setf (avx:f64.4-aref array 0)
             (avx:f64.4*
              (avx:f64.4-aref array 0)
              4)))
      (:sse2
       (setf (sse2:f64.2-aref array 0)
             (sse2:f64.2*
              (sse2:f64.2-aref array 0)
              4)
             (sse2:f64.2-aref array 2)
             (sse2:f64.2*
              (sse2:f64.2-aref array 2)
              4))))))

```

Figure 2: A simple example for selecting the best available instruction set at run time: The eight elements of a supplied vector of double floats are quadrupled, using either AVX instructions, or, if those aren't available, SSE2 instructions.

4 LOOPUS

Even though the interface provided by sb-simdis relatively convenient — at least when comparing it to similar libraries in other programming languages — there are certain repetitive patterns when writing vectorized code that almost beg for another layer of abstraction via macros. The most frequent repetitive pattern is that of using two loops to process a range of data: One with a step size that is the vectorization width, and one with a step size of one for handling the remainder. Figure 3 gives an example for this pattern. Further repetitive patterns are that of rewriting calls to `aref` as uses of `row-major-aref`, and hoisting all the loop invariant part of the index calculation outside of the loop.

After writing a variety of prototypes, we decided to create a portable loop optimization library for Common Lisp that can be used via macros. The library is invoked by using the `loopus:for` macro for looping over a range of integers. Once that macro is encountered, the whole form is turned into a tree of loops, where each loop contains zero or more data flow graphs whose nodes are function calls, and whose roots are array store instructions or reduction statements. Each data flow graph may also reference nodes from any of the graphs of the surrounding loops. The leaves of each data flow graph are either constants or array load instructions.

Only a small subset of Common Lisp is allowed in the body of a `loopus:for` macro: functions, macros, and the special operators `let`, `let*`, `locally`, and `progn`. For now, this subset strikes the right balance between expressiveness and ease of optimization, but we may add support for further special operators in the future. The good news is that once a programmer obeys these restrictions, the entire loop nest and all expressions therein are subject to the following optimizations:

- Rewriting of multi-dimensional array references to references using only a single row-major index.
- Symbolic optimization of polynomials, and especially of the expressions for calculating array indices.
- Hoisting of loop invariant code.
- Automatic SIMD vectorization.

One may wonder why we use macros and didn't just add our loop optimizations to SBCL directly. The reason is that implementing loop optimizations for the entire Common Lisp language is a daunting task. The many possible interactions of language features would force us to be conservative in terms of optimization, or spend much more time on the development that we can currently spare. One advantage of providing optimizations as a macro, is that they are automatically available to all Lisp implementations.

5 EXAMPLES

5.1 Sum of Positive Numbers

This first example illustrates the various features provided by sb-simd. We deliberately don't utilize Loopus for this example to give a realistic impression of how programming with raw SIMD instructions looks like. The example problem is that of summing numbers in a supplied vector, with the additional constraint that numbers less than zero shall be ignored. The AVX2 vectorized code to perform this task is given in Figure 3.

```
1   (in-package #:sb-simd-avx2)
2
3   (defun sum-positive-numbers (vec)
4     (declare (type (simple-array f64 (*)) vec))
5     (let ((n (array-total-size vec))
6           (i 0)
7           (acc (f64.4 0))
8           (result 0d0))
9       (declare (f64.4 acc) (f64 result))
10      (loop while (<= i (- n 4)) do
11        (let ((v (f64.4-aref vec i)))
12          (f64.4-incf acc
13              (f64.4-if (f64.4> v 0) v 0))
14          (incf i 4)))
15      (f64-incf result (f64.4-hsum acc))
16      (loop while (< i n) do
17        (let ((v (f64-aref vec i)))
18          (f64-incf result
19              (f64-if (f64> v 0) v 0)))
20        (incf i))
21      result))
22
```

Figure 3: Summing all positive numbers in a vector, using AVX2 intrinsics. Two loops are needed to process any number of elements correctly: One vectorized loop with a step size of four (lines 10–14), and another one for handling the remainder (lines 16–20).

5.2 Jacobi

In this second example, we compare the performance of Common Lisp and C for the problem of applying Jacobi's method on a two-dimensional domain. For the C code, we took the best implementation we could find (Figure 4) and compiled it with GCC 9.2 and with highest optimization settings (`-Ofast -march=native`). For the Lisp code in Figure 5 we used SBCL 2.2.0 and our loop optimization framework Loopus. The most critical part of assembler code of both versions is shown in Figures 6 and 7.

```
 void jacobi(double* dst, double* src,
             unsigned int rows,
             unsigned int columns) {
 double *C =  dst +   columns + 1;
 double *N =  src             + 1;
 double *W =  src +   columns    ;
 double *E =  src +   columns + 2;
 double *S =  src + 2*columns + 1;

 for(size_t iy = 0; iy < rows - 2; ++iy) {
  for(size_t ix = 0; ix < columns - 2; ++ix) {
      size_t idx = iy * columns + ix;
    C[idx] = 0.25 * (N[idx]+ S[idx]+W[idx]+E[idx]);
  }
 }
}
```

Figure 4: An efficient C implementation of Jacobi's method.

```
(defun jacobi (dst src)
 (declare (type (simple-array f64 2) dst src))
 (loopus:for (i 1 (1- (array-dimension dst 0)))
   (loopus:for (j 1 (1- (array-dimension dst 1)))
     (setf (f64-aref dst i j)
           (f64* 0.25d0
                 (f64+
                  (f64-aref src i (1+ j))
                  (f64-aref src i (1- j))
                  (f64-aref src (1+ i) j)
                  (f64-aref src (1- i) j)))))))
```

Figure 5: A Common Lisp implementation of Jacobi's method.

One can see that the assembler code produced by GCC (Figure 6) and SBCL (Figure 7) is extremely similar. Both versions use three 256 bit vector additions and one 256 bit vector multiplication. The only differences are that GCC's assembler code combines two loads directly with the subsequent addition, and manages to perform the loop test entirely in registers. In SBCL, the nature of how operations of its virtual machine are translated to assembler instructions makes it very hard to combine loads with subsequent instructions. We haven't yet investigated why SBCL decided to reference the stack for checking for termination of the innermost loop.

The reason that SIMD operations appear at all in the code by SBCL is that Loopus has automatically rewritten the scalar loop

```
L1: vmovupd ymm5, [rbx+rax*1]
    vmovupd ymm6, [r11+rax*1]
    vaddpd  ymm0, ymm5, [rcx+rax*1]
    vaddpd  ymm1, ymm6, [r10+rax*1]
    vaddpd  ymm0, ymm0, ymm1
    vmulpd  ymm0, ymm0, ymm3
    vmovupd [rdx+rax*1], ymm0
    add rax,0x20
    cmp [rsp+0x20], rax
    jne L1

```

Figure 6: The assembler code of the innermost loop produced by GCC 9.2 for our C code.

```
L1: vmovupd ymm0, [rsi+rbx*4+8]
    vmovupd ymm1, [rsi+rbx*4-8]
    vmovupd ymm2, [r9+rbx*4]
    vmovupd ymm3, [r8+rbx*4]
    vaddpd  ymm0, ymm0, ymm1
    vaddpd  ymm1, ymm2, ymm3
    vaddpd  ymm0, ymm0, ymm1
    vmulpd  ymm0, ymm4, ymm0
    vmovupd [rcx+rbx*4], ymm0
    add rbx, 8
    cmp rbx, rdx
    jl L1

```

Figure 7: The assembler code of the innermost loop produced by SBCL 2.2 for our Lisp code.

from Figure 5 line 4–11 as two loops, where the first loop has a step size of four and uses SIMD instructions and where the second loop has a step size of one and handles the remainder in case the loop length is not divisible by four. Furthermore, Loopus replaces each access to a Common Lisp array by direct pointer arithmetic, and hoists most of the loop index calculations outside of the innermost loop. This way, each array access can be encoded as a single load instruction whose address is the sum of two registers, where the value of the second register is scaled by a power of two, plus a small constant.

To compare the performance of our Lisp and C codes, we ran each Jacobi implementation for several minutes on a problem that fits well into the L1 cache of the target machine. In doing so, we ensure that the computation is not limited by memory throughput and thus accurately reflects how well the CPU can digest the generated machine code. Our results for a variety of x86-64 CPUs are shown in Figure 8.

CPU	Lisp	C	Ratio
AMD EPYC 7451 "Naples"	8.1	8.6	0.94
AMD EPYC 7452 "Rome"	8.2	10.0	0.82
Intel Xeon "Skylake" Gold 6148	10.8	13.9	0.78
Intel Xeon "Cascade Lake" Gold 6248	7.2	9.3	0.77
AMD EPYC 7543 "Milan"	12.8	18.1	0.71
Intel Xeon "Haswell" E5-2695 v3	6.9	9.8	0.70
Intel Xeon "Icelake" Platinum 8360Y	11.3	16.1	0.70
Intel Xeon "Icelake" Platinum 8358	7.9	11.3	0.70
Intel Xeon "Broadwell" E5-2697 v4	5.0	7.5	0.67

Figure 8: Performance in GFlop/s for Jacobi's method on various CPU architectures, as well as the ratio of the performance of the Lisp version and the C version.

6 CONCLUSIONS

We have narrowed the performance gap between Common Lisp and C for number crunching to something between 6% and 33%, depending on the target hardware. We achieved this by developing a low-level SIMD library for SBCL, named sb-simd, and a portable library for loop optimization and automatic vectorizaton, named Loopus.

The interface provided by sb-simd is the most convenient way of using SIMD intrinsics among all programming languages known to us. It treats SIMD packs as regular, typed objects, allows the development of SIMD codes at the REPL, and even provides an introspection mechanism for querying the avaiable instructions and data types at run time. We hope that the interface provided by sb-simd will eventually be supported by multiple Lisp implementations and turn into a de-facto standard similar to bordeaux-threads.

The loop optimization library Loopus is still in its infancy, but already powerful enough to vectorize inner loops written in a certain subset of Common Lisp. This makes it possible to harness SIMD instructions for a wide variety of loops using minimal effort. We hope that this paper will attract further contributors, and that Loopus will one day reach feature parity with the loop optimization machinery in GCC and Clang.

What makes us particularly excited about these new libraries is that they turn Common Lisp into a viable language for high performance computing. Programmers in that domain can now finally enjoy the convenience and flexibility of using Lisp, and, most importantly, harness the power of Lisp macros to develop lightweight, domain-specific optimizations. We are confident that in many cases, such domain-specific optimizations can outperform the general-purpose work that is normally done by a compiler. We are looking forward to working on such optimizing Lisp macros in the future.

7 ACKNOWLEGMENTS

This work wouldn't have been possible without the support of many other people and organizations. We express our heartfelt gratitude to:

- KONWIHR, the Competence Network for Scientific High Performance Computing in Bavaria, for funding the advancement of SIMD instructions in SBCL and our work on sb-simd.

- Paul Khuong, for writing the initial support for SIMD programming in SBCL.
- Stas Boukarev, for adding the low-level machinery required for AVX, AVX2, and FMA instructions to SBCL.
- All the other SBCL developers for kindly supporting this work and providing feedback, especially Douglas Katzmann, Charles Zhang, and Christophe Rhodes.
- Bela Pecsek for being an ardent user of sb-simd since the first day, for porting various benchmarks from C to SIMD-enhanced Lisp, and for his frequent feedback and bug reports.
- Nicolas Neuss, Hayley Patton, Michał Herda, Jan Münch, and Shubhamkar Ayare, for some valuable discussions.
- The group from the Erlangen National High Performance Computing Center (NHR@FAU) for giving us access to their benchmark systems, and for insistently questioning the performance of Lisp. The latter turned out to be a surprisingly good motivation.

REFERENCES

[1] Gabriel, R. P. *Performance and Evaluation of LISP Systems.* The MIT Press, 08 1985.

[2] Graham, P. *On LISP.* Pearson, Upper Saddle River, NJ, Sept. 1993.

[3] Svingen, B. When lisp is faster than C. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2006), GECCO '06, Association for Computing Machinery, p. 957–958.

[4] Verna, D. How to make lisp go faster than C. In *Proceedings of the International MultiConference of Engineers and Computer Scientists* (Hong Kong, June 2006), International Association of Engineers.

[5] Ó Nualláin, B. Executable pseudocode for graph algorithms. In *Proceedings of the 8th European Lisp Symposium* (Apr. 2015), ELS2015.