ARCHITECTURE INDEPENDENT SHORT VECTOR FFTS

Franz Franchetti, Herbert Karner, Stefan Kral, Christoph W. Ueberhuber

Department of Applied and Numerical Mathematics
Technical University of Vienna
Wiedner Hauptstrasse 8–10/115, A–1040 Vienna, Austria
franz@aurora.anum.tuwien.ac.at, christof@aurora.anum.tuwien.ac.at

ABSTRACT

This paper introduces a SIMD vectorization for FFTW—the “fastest Fourier transform in the west” by Matteo Frigo and Steven Johnson. The new method leads to an architecture independent short vector SIMD FFT vectorization that utilizes the architecture adaptivity of FFTW. It is based on special FFT kernels (up to size 64 and more) that are utilized by FFTW to compute the whole transform. This vectorization supports all features of complex transforms in FFTW (arbitrary size, dimension and stride of the data vector; in-place and out-of-place transforms) and is fully transparent to the user. It is suitable for arbitrary vector sizes of the underlying hardware.

1. INTRODUCTION

Major vendors of general purpose microprocessors have included SIMD extensions into their instruction set architecture (ISA) to improve the performance of multi media applications by exploiting the parallelism available in most multi media kernels.

All these ISA extensions are based on the packing of large registers with smaller data types (usually of 8-bits, 16-bits or 32-bits) and parallel operation on the subwords within one register. This is called *short vector* SIMD parallelism.

In this paper, a SIMD vectorized version of FFTW [2] for different platforms is presented. It extends the FFTW-framework with new vectorized codelets introduced in this paper. In this respect we focus on the balance between performance, portability and the integration into the existing FFTW system.

The paper is organized as follows. Section 2 describes the architecture of different multi media extensions available on recent processors. Section 3 shortly describes FFTW, while Section 4 describes how FFTW SIMD code can be made architecture independent. Section 5 discusses the vectorization of the kernels and Section 6 presents runtime results.

2. SHORT VECTOR EXTENSIONS

The following architectures feature a floating-point short vector SIMD instruction set extension and are therefore potential targets for FFTW SIMD vectorization.

The AltiVec SIMD architecture [8] [9] of Motorola’s G4 generation of PowerPC microprocessors, the MPC7400, expands the current PowerPC architecture through the addition of a 128-bit vector execution unit, which operates independently of the existing integer and floating-point units. This new execution unit provides integer SIMD and four-way single-precision floating-point SIMD instructions.

The Intel Pentium III streaming SIMD extensions (SSEs) [5] [6] [7] include 70 new instructions, for instance, general purpose floating-point instructions, which operate on a new set of eight 128-bit SSE registers, and integer instructions as well as cacheability control and data streaming capabilities.

Intel’s Willamette Processor (which is the first processor with the new IA kernel and successor of the P6 processor line) is capable of two-way double-precision floating-point SIMD operations in addition to the four-way single-precision floating-point SIMD capabilities.

The new Intel IA-64 architecture includes two-way single-precision floating-point SIMD instructions that operate on generic 64-bit registers.

The AMD 3DNow! technology provides an additional instruction set for SIMD processing. The 3DNow! instructions operate on 64 bit registers, divided into two single-precision floating-point words which are mapped onto the floating-point registers.

3. FFTW: A RECURSIVE, KERNEL-BASED APPROACH

FFTW (Frigo, Johnson [2]) is currently the most sophisticated and fastest FFT package. It is based on a recursive algorithm that handles arbitrary array sizes, dimensions and strides. Portability and architecture-adaptivity are further important features of FFTW.

In FFTW, the computation is accomplished by an *executor* which calls highly optimized blocks of code called *codelets*. The combination of codelets applied by the executor is specified by a special data structure called a *plan*. The plan is determined at runtime in an extra initialization phase by the *planner*. The planner measures the runtime of many plans and selects the fastest one (Frigo, Johnson [2], Haentjens [3]). Since the generated plans depend on the computer used, FFTW performs an *architecture adaptive* FFT computation.

Codelets are powerful machine-generated blocks of code for performing computational steps for FFT execution of various sizes. The standard distribution of FFTW includes various codelet sizes (powers of two up to 64 and combinations of smaller primes). Arbitrarily sized codelets can be generated automatically by the FFTW codelet generator `genfft` [4].

Basically there are two codelet types, *twiddle codelets* and *no twiddle codelets*, which have different fields of utilization. A twiddle codelet of size p computes an in-place FFTs of given size

This work was supported by the Special Research Program SFB F011 “AURORA” of the Austrian Science Fund FWF.

```
void fftw_twiddle_2 (fftw_complex * A, const fftw_complex * W,
    int iostride, int m, int dist)
{
  int i;
  fftw_complex *inout;
  inout = A;
  for (i = m; i > 0; i = (i - 1), inout = (inout + dist), W = (W + 1))
    {
      fftw_real tmp1, tmp8, tmp6, tmp7;
      tmp1 = c_re (inout[0]);
      tmp8 = c_im (inout[0]);
      {
        fftw_real tmp3, tmp5, tmp2, tmp4;
        tmp3 = c_re (inout[iostride]);
        tmp5 = c_im (inout[iostride]);
        tmp2 = c_re (W[0]);
        tmp4 = c_im (W[0]);
        tmp6 = (tmp2 * tmp3 - tmp4 * tmp5);
        tmp7 = (tmp4 * tmp3 + tmp2 * tmp5);
      }
      c_re (inout[iostride]) = (tmp1 - tmp6);
      c_re (inout[0]) = (tmp1 + tmp6);
      c_im (inout[0]) = (tmp7 + tmp8);
      c_im (inout[iostride]) = (tmp8 - tmp7);
    }
}
```

Figure 1: A Radix-2 Twiddle Codelet.

scaled by the twiddle factors which can be outlined by

$$y := (F_q \otimes I_r) T_r^{qr}\, y\,, \tag{1}$$

where T is diagonal scaling matrix. See Fig. 1 for a standard twiddle codelet of size 2.

A no twiddle codelet of size r performs an out-of-place FFT of given size with different strides for the input and output which can be outlined by

$$y := F_r\, x. \tag{2}$$

4. THE SIMD MACRO FRAMEWORK

SIMD vectorization is a highly machine dependent process with no common language extension or standard API. Short vector extensions differ in various aspects like the programming model, instruction set, data types, and syntax.

Currently, application developers have two ways to access SIMD hardware. They can either rewrite key portions of the application in assembly language using the SIMD instructions, or they use a high-level language and apply vendor-supplied macros that provide the functionality of the SIMD-processing primitives.

The most common language extension for specifying SIMD-processing primitives is to provide function-call like macros within the C programming language. Each macro directly translates to a single SIMD processing instruction, leaving register allocation and instruction scheduling to the compiler. This approach would be even more attractive to application developers if the industry agreed to a common set of macros, rather than having a different set from each vendor.

Different programming models and proprietary software support do not have an impact as severe as expected on the vectorization process, because there are only a few operations needed within FFTW. These are standard operations like (i) loading a complex number, (ii) storing a complex number, (iii) loading a twiddle factor, (iv) declaring a constant, (v) using a constant, (vi) extracting the real parts into a vector, (vii) extracting the imaginary parts into a vector, $(viii)$ building complex numbers from a vector of real parts and a vector of imaginary parts, (ix) adding two vectors, (x) subtracting two vectors, and (xi) multiplying two vectors.

These operations must be mapped onto the underlying SIMD-architecture. To achieve a high abstraction level, the basic FFTW operations are mapped directly onto a set of macros. All operations within the codelets are done using these macros (see Figs. 2 and 3 for details). So the vectorized version of FFTW is architecture independent and all architectural differences are covered by one single include file that defines the needed high-level macros.

```
typedef __m128 FFTW_SIMD_VECT;
typedef __m64 FFTW_SIMD_COMPLEX;

/*  define const operations         */
#define FFTW_SIMD_KONST(c,v)                                          \
    static const __declspec(align(16))  float (c)[4]={v,v,v,v}
#define FFTW_LOAD_KONST_SIMD(c) *(FFTW_SIMD_VECT *)(c)

/*  define arithmetic operations    */
#define SIMD_ADD(a,b)   _mm_add_ps((a),(b))
#define SIMD_SUB(a,b)   _mm_sub_ps((a),(b))
#define SIMD_MUL(a,b)   _mm_mul_ps((a),(b))

/*  define load operations          */
#define LOAD_RE_IM(re,im,input,stride)                                \
{                                                                     \
    FFTW_SIMD_VECT ldtmp1,ldtmp2;                                     \
    ldtmp1=_mm_loadl_pi(ldtmp1,(input));                              \
    ldtmp1=_mm_loadh_pi(ldtmp1,(input) + (stride));                   \
    ldtmp2=_mm_loadl_pi(ldtmp2,(input) + 2 * (stride));               \
    ldtmp2=_mm_loadh_pi(ldtmp2,(input) + 3 * (stride));               \
    (re)=_mm_shuffle_ps(ldtmp1,ldtmp2,_MM_SHUFFLE(2,0,2,0));    \
    (im)=_mm_shuffle_ps(ldtmp1,ldtmp2,_MM_SHUFFLE(3,1,3,1));    \
}
```

Figure 2: FFTW SIMD Macros for the Intel Pentium III.

FFTW accesses single real numbers like the real or imaginary part of a complex number. That results in an access to quantities of size 4 byte (single-precision) or 8 byte (double-precision) that are naturally aligned. But SIMD instructions normally offer fast access only to naturally aligned n-way vectors, e. g., 16 byte aligned 16 byte quantities (vectors of 4 single-precision floating-point numbers) for four-way SIMD architectures. All other accesses are called unaligned and are very expensive in terms of access time. Avoiding extra loads, stores and shuffling is therefore one of the key problem when producing fast FFTW SIMD kernels.

5. VECTORIZING THE KERNELS

To vectorize the FFTW codelets, two different approaches have been applied.

Internal Vectorization. The computation in the codelets can be vectorized, if the codelet contains a loop. The vectorization results in computing some passes of the loop simultaneously. Twiddle codelets can be handled that way.

External Vectorization. If more than one codelet has to be executed with the same parameters on strided data, subsequent calls to this codelet can be replaced by one call to a vectorized form of this codelet. This can be done using the no twiddle codelets as well as using the twiddle codelets. The different methods result in a different number of memory accesses due to the way, twiddle factors are accessed. The cache locality of FFTW can be perturbed as memory accesses are reordered.

Per invocation, an internally vectorized codelet does the same amount of work as a standard codelet, while an externally vectorized codelet does n times the work of a standard codelet on a n-way SIMD architecture.

The vectorization presented in this paper requires changes in the codelets, the executor and the planner as well as in FFTW's internal data structures. Every codelet (no twiddle and twiddle) gets an internally vectorized and an externally vectorized version associated. Whenever a codelet (i. e., a pointer to a codelet) is saved, three pointers to the associated codelets are saved after the vectorization. So the planner has to initialize the plans in a different way. These pointers are used by the executor, to use the appropriate vectorized codelet instead of a loop of standard codelets.

To vectorize a codelet, either a loop has to be vectorized or the computation of four codelets are put into one vector codelet. Like the FPU codelets the SIMD codelets are generated automatically by incorporating the following transformations into the FFTW codelet generator.

- In the FPU codelet the real parts and the imaginary parts can be accessed independently. In a SIMD codelet, the real part and the imaginary part have to be loaded with one macro.
- An equivalent transformation is applied to the data storing instructions.
- Any arithmetic operation is transformed into the corresponding macro.
- The function prototypes and data types have to be adjusted.
- An extra stride parameter is passed to the twiddle codelets.
- According to the vectorization type, the twiddle factor access has to be adjusted.

See Fig. 4 for the Radix-2 SIMD twiddle codelet.

The changes in the internal data structure must be handled by the executor loop and by the loops in the twiddle codelets within the executor and results in an different index computation within the executor. E. g., for a vectorized loop with 10 passes on a 4-way SIMD architecture, 2 vectorized passes and 2 standard passes are needed when operating with externally vectorized codelets. The function `fftw_executor_simple` is the FFTW recursion entry point. All further steps are done by calls to `executor_many_vector`.

In `fftw_executor_simple`, the no twiddle case can only occur with only one call to the codelet as it is the entry point of the recursion. So this case cannot be vectorized. The key issue is the twiddle case.

In the `executor_many_vector`, both the twiddle and the no twiddle case have to be changed. The no twiddle case occurs with a lot of repetitions and is the leaf of the recursion. So the vectorized version of the codelets can be used.

In the twiddle case, both the internally and the externally vectorized case is possible. The original code can be modified in two different ways: The first method leads to the externally vectorized version. Only the standard codelets and the externally vectorized codelets are used. This vectorization may lead to cache problems, as elements with big power of two strides are loaded within a few lines of code within the codelet. However, this technique is more obvious and faster to implement.

The internally vectorized version has the same data access pattern as the original FFTW version and shows a better data locality than the externally vectorized version. As the no twiddle case has no obvious internal vectorization, two different vectorization methods have to be used: the external version for the no twiddle case and the internal vectorization for the twiddle case. This leads to a more complex executor implementation.

6. EXPERIMENTAL RESULTS

The new SIMD version of FFTW was tested on a 650 MHz Pentium III system operating under Windows NT 4.0 using the native

```
#define FFTW_TWIDDLE_STRIDE_2 1

void fftw_twiddle_simd_int_2 (FFTW_SIMD_COMPLEX * A,
    FFTW_SIMD_COMPLEX * W, int iostride, int m, int dist)
{
  int i;
  FFTW_SIMD_COMPLEX *inout;
  inout = A;
  m >>= FFTW_LD_SIMD_LEN;
  for (i = m; i > 0; i = (i - 1), inout = (inout + FFTW_SIMD_LEN * dist),
    W = (W + FFTW_TWIDDLE_STRIDE_2 * FFTW_SIMD_LEN))
    {
      FFTW_SIMD_VECT tmp1, tmp8, tmp6, tmp7;
      LOAD_RE_IM (tmp1, tmp8, inout + (0), dist);
      {
        FFTW_SIMD_VECT tmp3, tmp5, tmp2, tmp4;
        LOAD_RE_IM (tmp3, tmp5, inout + (iostride), dist);
        LOAD_RE_IM (tmp2, tmp4, W + (0), FFTW_TWIDDLE_STRIDE_2);
        tmp6 = SIMD_SUB (SIMD_MUL (tmp2, tmp3), SIMD_MUL (tmp4, tmp5));
        tmp7 = SIMD_ADD (SIMD_MUL (tmp4, tmp3), SIMD_MUL (tmp2, tmp5));
      }
      STORE_RE_IM (SIMD_ADD (tmp1, tmp6),
        SIMD_ADD (tmp7, tmp8), inout + (0), dist);
      STORE_RE_IM (SIMD_SUB (tmp1, tmp6),
        SIMD_SUB (tmp8, tmp7), inout + (iostride), dist);
    }
}
```

Figure 3: The Radix-2 SIMD Twiddle Codelet.

Intel C/C++ Compiler 4.5 and on a 400 MHz G4 AltiVec system operating under Yellodog Linux 1.2 using `gcc-vec` 2.9.5. In both cases the highest available optimization was used (see [1] for details).

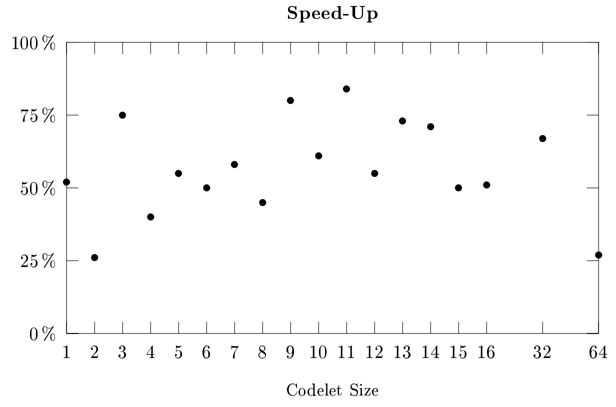

Figure 4: Speed-Up of No Twiddle Codelets on a 650 MHz Pentium III.

In the first test series, only the codelet runtime was measured. The internally vectorized codelets and the externally vectorized codelets were compared to the original FPU codelets. In these tests, the externally vectorized codelets were typically about 15 % faster than the internally vectorized codelets due to a smaller number of shuffling operations. Speed-ups of 25 % to 70 % are achieved over the standard FPU codelets. See Figs. 5 and 6.

In the second test series, the new codelets were tested within FFTW. The SIMD FFTW version was tested using vectorlengths ranging from 2^4 to 2^{20} and ranging from 10^1 to 10^6. Compared to the standard version of FFTW typically speed-ups of 25 % to 50 % have been achieved.

For the vectorlengths of type 2^k the performance of the externally vectorized version degrades (due to cache associativity problems) while the internally vectorized version shows a good speed-up for both the in-cache case and the out-of-cache case.

For vectorlengths of type $k10^d$ the externally vectorized version and the internally vectorized version show similar characteris-

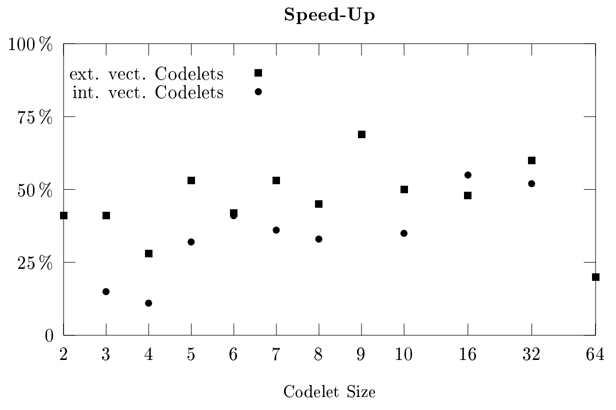

Figure 5: Speed-Up of Twiddle Codelets on a 650 MHz Pentium III.

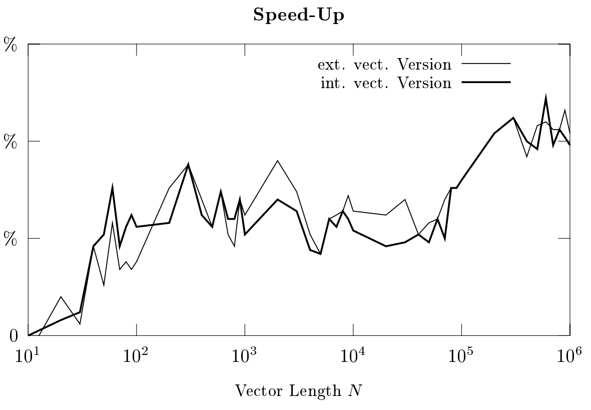

Figure 7: Speed-Up compared to FFTW FPU, $N = k10^d$ on a 650 MHz Pentium III.

tics with an average speed-up of about 25 % which even increases for the out of cache case. Typically more than 99 % of the computation is done in the SIMD part. See Figs. 7 and 8.

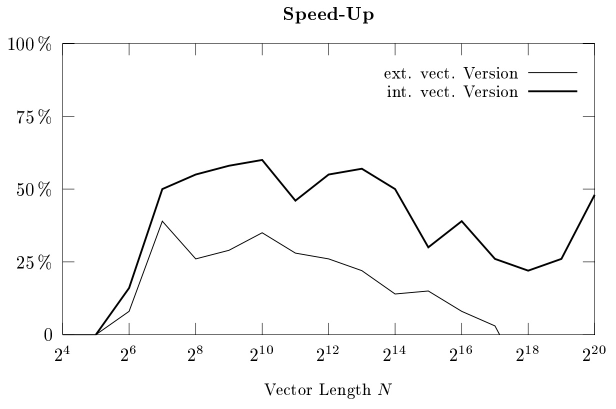

Figure 6: Speed-Up compared to FFTW FPU, $N = 2^k$ on a 650 MHz Pentium III.

7. CONCLUSION

In this paper a SIMD extension to FFTW is presented. It supports all features of complex transforms in FFTW (arbitrary size, dimension and stride of the data vector; in-place and out-of-place transforms) in an architecture independent way. Arbitrary n-way floating-point SIMD is supported within arbitrary sized FFTW kernels. It turns out that the access to complex numbers instead of SIMD vectors within the kernels is the key problem.

Significant speed-ups over the standard version of FFTW have been achieved.

For unit stride data, special codelets are more appropriate. They are incorporated into the next version of FFTW as specialized solvers.

8. ACKNOWLEDGEMENTS

Matteo Frigo, with whom we have worked closely for more than a year, has had a very strong influence on the greater part of this paper.

9. REFERENCES

[1] F. Franchetti, *Short Vector FFTs*, Master Thesis, Department of Applied and Numeric Mathematics, Technical University of Vienna, 2000.

[2] M. Frigo, S. Johnson, *FFTW: An Adaptive Software Architecture for the FFT*, Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation, 1999, Pages 169 - 180.

[3] G. Haentjens, *An Investigation of Recursive FFT Implementations*, Masters Thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[4] M. Frigo, *A Fast Fourier Transform Compiler*, Proceedings of the PLDI Conference, May 1999, Vol. 3, p. 1381.

[5] Intel Corporation, *Intel C/C++ Compiler User's Guide — With Spport for the Streaming SIMD Extensions*, 1999.

[6] Intel Corporation, *Intel Architecture Software Developer's Manual*, 1999.

[7] Intel Corporation, *AP-833 Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*, 1999.

[8] Motorola Corporation, *AltiVec Technology Programming Environments Manual*, 1998.

[9] Motorola Corporation, *AltiVec Technology Programming Interface Manual*, 1998.