Mixed Data Layout Kernels for Vectorized Complex Arithmetic

Doru T. Popovici, Franz Franchetti, Tze Meng Low
Department of Electrical and
Computer Engineering
Carnegie Mellon University
Email: {dpopovic, franzf, lowt}@cmu.edu

***Abstract*—Implementing complex arithmetic routines with Single Instruction Multiple Data (SIMD) instructions requires the use of instructions that are usually not found in their real arithmetic counter-parts. These instructions, such as shuffles and `addsub`, are often bottlenecks for many complex arithmetic kernels as modern architectures usually can perform more real arithmetic operations than execute instructions for complex arithmetic. In this work, we focus on using a variety of data layouts (mixed format) for storing complex numbers at different stages of the computation so as to limit the use of these instructions. Using complex matrix multiplication and Fast Fourier Transforms (FFTs) as our examples, we demonstrate that performance improvements of up to 2x can be attained with mixed format within the computational routines. We also described how existing algorithms can be easily modified to implement the mixed format complex layout.**

I. Introduction

Complex arithmetic is often required in many different computational domains such as signal processing, material sciences [16], and machine learning [15]. In all these domains, the complex data is typically stored in the traditional interleaved data format where real and imaginary components of a complex number are stored in sequential locations in memory. This default layout is convenient as it allows us to treat complex numbers in the same manner as real numbers when it comes to laying out the data in memory, or performing computation on them.

The reality of implementing complex arithmetic is that while complex addition is an extension of real addition, the same does not hold for complex multiplication. Consider how complex multiplication is implemented[1]:

$$\big(a + b\mathbf{i}\big) \times \big(c + d\mathbf{i}\big) = \big(ac - bd\big) + \big(ad + bc\big)\mathbf{i}. \qquad (1)$$

Notice that the result is dependent on both the real and imaginary components of the source operands. Additional operations, such as addition and subtraction, are also performed. These differences between real and complex multiplication becomes even more apparent when high performance math kernels involving the multiplication of complex numbers are implemented using Single Instruction Multiple Data (SIMD) instructions. The traditional format necessitates permuting elements to facilitate the multiplications of real with imaginary components of different complex numbers. Furthermore, instructions such as `addsub` are required to perform the additional computation required by the complex multiplication. As the throughput (i.e. number of parallel instructions executed per cycle) of such instructions are usually limited, this creates a bottleneck in the computation of complex kernels.

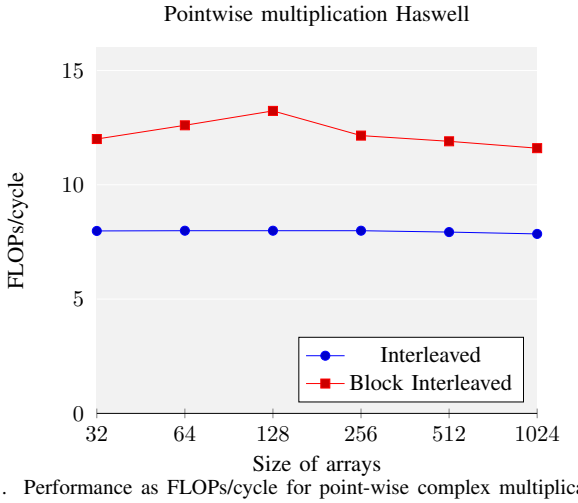

Fig. 1. Performance as FLOPs/cycle for point-wise complex multiplication using interleaved and split data layout. The interleaved data layout has shuffle instructions on the critical path.

This bottleneck is illustrated by the performance plot in Figure 1. We report performance numbers for computing a point-wise complex multiplication between two vectors of various lengths on an Intel Haswell architecture. The Haswell processor can compute two fused-multiply-add instructions per cycle, whereas only one permutation can be performed in each cycle. As the operations required for complex multiplication are dependent on the completion of the permutations, the limited throughput of the permutation becomes a bottleneck that lowers the achievable performance when using the interleaved format (blue line with circle markers). Changing the data layout eliminates permutations, and therefore performance increases (red line with square markers).

We make the observation that while the input and output values of complex mathematical routines have to be conformal with the traditional interleaved data format, the intermediate values can be stored in a computationally more convenient data

[1]There is an alternative method for computing a complex multiplication, known as the 3M method, described by Higham [8]. This computation requires 3 real multiplication and 5 real additions. This method of computation is not discussed in this paper.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

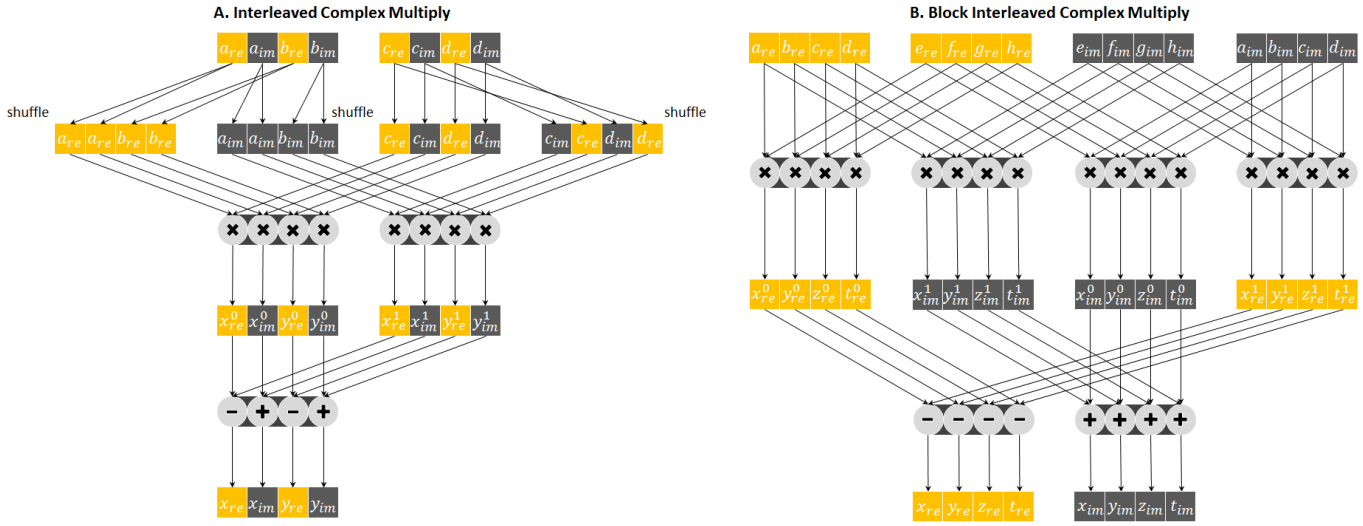

Fig. 2. Two implementations of the SIMD complex multiplication when data is stored in complex interleaved (left) and block interleaved (right) data formats. The SIMD vector length is $\nu = 4$. When data is stored in interleaved format, each vector register can hold two complex numbers. When data is stored in block interleaved, each vector registers stores four real components or four imaginary components. The real components of the complex numbers are represented by the yellow color, while the imaginary components are represented by the dark gray color.

format that reduces the number of special instructions used and increases overall performance. In this paper, we leverage this observation that allows us to design complex arithmetic kernels that change the data layout during the computation in order to reduce the use of the special instructions.

Contributions Specifically our contributions are as follows:

- *Mixed data layout complex arithmetic kernel.* We introduce the use of different data layout formats at different stages of the complex arithmetic kernels to reduce and even eliminate permutations and other instructions that have limited throughput on most systems. These instructions are often bottlenecks of complex arithmetic kernels, and by limiting their use, even highly optimized kernels can benefit in yielding higher performance.
- *Opportunities for introducing mixed format.* Using examples from two different domains where complex arithmetic are used, we illustrate different opportunities where the switching of mixed format can be introduced. By leveraging existing data movement patterns in existing kernels, we show that the cost of switching between data formats within the computation can be hidden effectively.

II. Data Layout for Complex Numbers

In this section, we discuss the pros and cons of two different data layout for storing complex numbers when complex multiplication is required.

A. Traditional / Interleaved Data Layout

Traditionally, sequences of complex numbers have been stored in interleaved format, where the real and imaginary components are located in consecutive memory locations. While this format is natural, implementing complex multiplication with SIMD instructions will require the permutation of the individual components of the complex numbers.

Complex multiplication on interleaved data requires three separate permutations to reorder the data into the appropriate location in order to perform SIMD computation. Data needs to be unpacked, then multiplied to obtain the intermediate results and finally added together with an `addsub` instruction to obtain the final result. It is important to notice that the shuffle operations must be executed before the floating point computation could start. Having the permutations on the computation's critical path will degrade performance. Figure 2 (left) describes one possible way of computing the complex multiplication on interleaved data using SIMD instructions.

B. Split Data Layout

The split data format stores the real and imaginary components as two disjoint sequences. using this format, a sequence of N complex numbers is split into a sequence of N contiguous real components and a sequence of N contiguous imaginary components. Storing complex number as two disjoint sequences of real and imaginary components allows one to perform the complex multiplication as if one would perform the scalar complex multiplication, i.e.

$$\left(\vec{a} + \vec{b}\mathbf{i}\right) \times \left(\vec{c} + \vec{d}\mathbf{i}\right) = \left(\vec{ac} - \vec{bd}\right) + \left(\vec{ad} + \vec{bc}\right)\mathbf{i}.$$

The difference is that instead of computing one complex multiplication at a time, the SIMD implementation computes ν complex multiplications at a time, where v is the SIMD vector length. Figure 2 (right) shows the steps required to do a complex multiplication if data is stored in split layout. It can be seen that this implementation does not require shuffle/permute operations before each multiplication. A variant of the split data layout is the *block interleaved layout* [2], where every block of b contiguous complex numbers is stored in the split data format. Typically, the value of b is equal to the length of the SIMD register ν.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

While we suspect that these benefits of the split data format may be known[2], it is not commonly seen in implementations of complex arithmetic kernels. One possibility is that many existing software only support the interleaved data format, and switching from the interleaved data format to split format is time-consuming as data has to be packed and unpacked between library routines.

III. Mixed Format Complex Arithmetic Kernels

In this section, we discuss how both interleaved and blocked interleaved data format can be introduced into state-of-the-art implementations of matrix multiplication, and fast Fourier transforms. In each of the domains, we highlight opportunities within the computational routine to perform the data layout transformation and how the computation needs to be updated to incorporate these mixed format kernels.

A. Complex Matrix Multiplication

High performance matrix multiplications are built with loops around a small but highly optimized matrix multiplication as the kernel [12], [7]. Within the BLAS-like Instantiation Software (BLIS) framework [17], a relatively recent open-source framework for implementing high performance BLAS-like operations using the Goto approach [6], the kernel is known as the *micro-kernel*. This micro-kernel computes the operation $C = \alpha AB + \beta C$ where where C is a $m_r \times n_r$ matrix, A is $m_r \times k_c$ and B is $k_c \times n_r$, and $k_c \gg m_r, n_r$. α and β are complex scalar values. In addition, matrices A and B are assumed to be packed into two separate blocks of consecutive memory, and are stored in column-major order and row-major order respectively. Pictorially, the BLIS framework is shown in Figure 3.

1) Implementation with interleaved data format: Inspecting the BLIS micro-kernel that computes a complex matrix multiplication [18], we observe that the micro-kernel is computed as a sequence of three distinct operations,

$$\begin{aligned} T &:= AB \\ T &:= \alpha T \\ C &:= \beta C + T, \end{aligned}$$

where A, B C and the temporary matrix T are stored in interleaved format. As most of the complex multiplications required to compute the micro-kernel occur in the first operation, $T := AB$, the expert avoids performing multiple permutations for each complex multiplication by storing the four intermediate values (i.e. ab, ac, bc and bd as described in Equation 1) separately. These four intermediate values are accumulated (either added or subtracted) to form the real and imaginary components of the values to be stored in T at the end of the first operation. The latter two operations are then performed via a point-wise multiplication when input and output operands are stored in interleaved data format.

[2]Fastest Fourier Transform in the West (FFTW) library [4], a commonly used FFT library, offers an interface for computing FFTs using the split data layout.

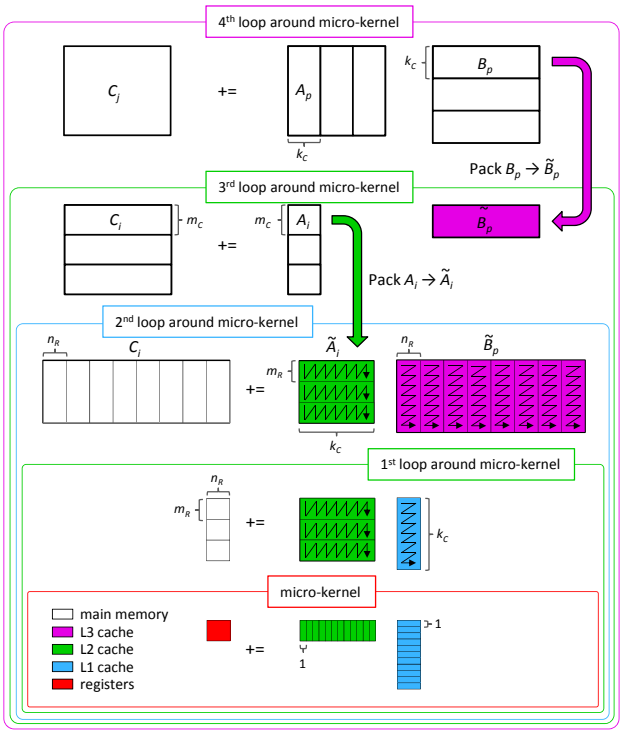

Fig. 3. Structure of the high performance matrix multiplication within the BLIS framework [17]. Image taken from [18]. By providing an optimized micro-kernel, a high performance implementation for matrix multiplication is obtained.

The cost of this approach is that twice as much memory is required to store the intermediate results. Four intermediate values are required to store each complex number as opposed to storing just the real and imaginary values. For the case, where C and T are kept in registers (as is the case with the micro-kernel), this means that m_r and n_r, the dimensions of C and T, are restricted to smaller values, thus requiring more round-trips through the data to perform the computation.

2) Mixed format implementations: In order to implement the mixed format, recall that A and B are packed into contiguous storage. As data packing is inherently a memory-bound operations, this is a natural opportunity to switch the data format of matrix B from interleaved to the blocked interleaved format.

Given that matrix B has been packed accordingly, the computation within the kernel can be simplified. Firstly, both real and imaginary components of elements from matrix B is scaled by a real component of an element from matrix A to yield temporary values for matrix T. Subsequently, the imaginary component of the element from A is used to scale the same real and imaginary components of B and accumulated (or subtracted) into the same matrix T to finish computing the complex multiplication. This reduces the need for storing twice as many intermediate values in T.

This approach eliminates any permutation required for the complex multiplication. As data in matrix T is already grouped

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

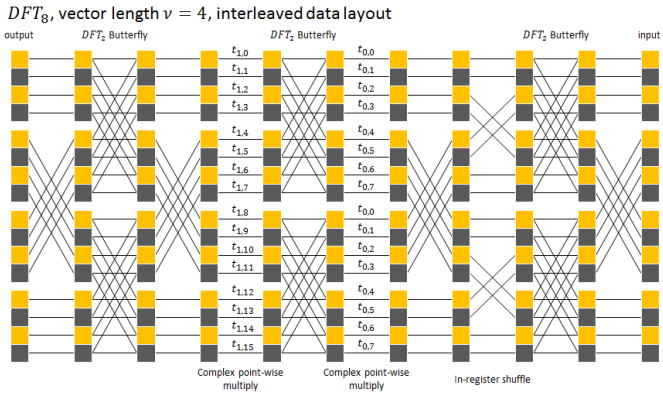

Fig. 4. The dataflow for a DFT of size 8 if implemented for a vector length $\nu = 4$. The DFT is decomposed in several stages. It is important to notice that the DFT requires one stage of in-register permutations and two stages of complex multiplications with twiddle factors.

into the real and imaginary components, the scaling with α can be performed without permutation as well. Finally, the permutation back to interleaved format can be performed either before or after the last operation, $C = \beta C + T$, preserving the input and output behavior of the complex kernel.

The net effect of this simplification are multi-fold. Firstly, the output matrix C of the micro-kernel has to be made larger to hide the latency of the computational instruction [13]. Secondly, the reduction in the number of intermediate values also allows us to create a larger micro-kernel. Finally, the blocking parameters of the loops around the micro-kernel has to be changed to optimize for the new micro-kernel. To determine the new sizes of the micro-kernel and the parameters of the surrounding loops, we derived these new values using the analytical models introduced in [13].

B. Discrete Fourier Transform (DFT)

Fast Fourier Transform (FFT) algorithms for computing the discrete Fourier Transform (DFT) such as the Cooley-Tukey algorithm [1] decompose the discrete Fourier Transform (DFT) of a composite size N into smaller DFTs, in order to reduce computational complexity. Thus, a DFT of size $N = mn$ can be expressed as: the computation m DFTs of size n, followed by a point-wise complex multiplication with the roots of unity (commonly known as twiddle factors [5]), and the computation of n DFTs of size m. If m and n are composite numbers, the smaller DFTs are recursively decomposed following the same steps. The decomposition stops when the size of the DFT is two. The DFT_2 is represented by the butterfly matrix.

1) Implementation with interleaved data format: Notice that each decomposition of a DFT requires a point-wise complex multiplication with the roots of unity. Therefore, the SIMD implementation of the complex multiplication will suffer from the permute instructions if data is preserved in interleaved format. In addition, the FFT inherently performs data transpositions, since all the input data points are required to compute each output data point. The transpositions themselves require permute instructions. Figure 4 illustrates the dataflow of a DFT of size 8 using a vector length of size $\nu = 4$. It can be seen that the DFT requires one stage of data permutations and two stages of complex multiplications with twiddle factors.

One solution to reduce the number of permute instructions from the critical path is to duplicate the twiddle factors and shuffle the data at initialization. This solution comes though at the cost of an increased memory footprint. Details on SIMD implementations for the FFT can be found in [2].

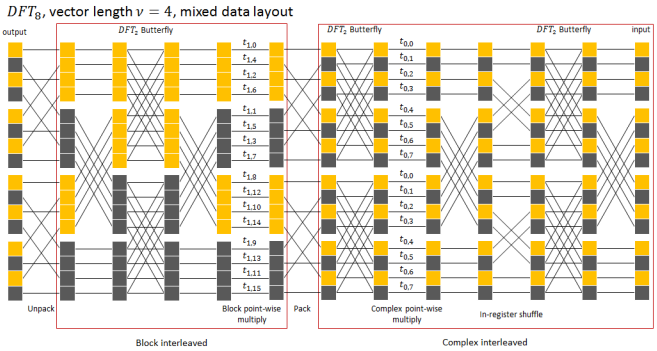

Fig. 5. The dataflow for a DFT of size 8 if implemented for a vector length $\nu = 4$. The first part of the DFT is compute in the interleaved format. This part will include the data permutations and some of the twiddle multiplication. Once data is in the correct shape, it can be changed to the block interleaved format. This will remove the permute instructions from the second twiddle multiplication.

2) Implementation with mixed data format: An alternative solution is to change the layout of the data set. Since the twiddle factors are precomputed arrays, they can be stored in block interleaved at initialization. However, changing the input layout cannot be done upfront as this would incur additional permutation instructions to ensure that every input element is used to compute every output element. Instead, the format change must be done as computation is performed.

Figure 5 shows the data flow for a mixed data layout DFT of size 8 when the SIMD vector length is $\nu = 4$. Essentially, the FFT algorithm is split into two separate stages of computation. The first stage of the mixed data layout FFT algorithm is identical to the traditional FFT algorithm, where permutations and twiddle multiplications are performed using the interleaved data format.

At the end of first stage of computation, the data is unpacked as part of the necessary data transposition incurred by the FFT algorithm into the blocked interleaved format. This allows us to compute the second part of the DFT in blocked interleaved format, thus avoiding the permutations required by the twiddle factor multiplications. In addition, by selecting where the second stage of the DFT starts, the permutations incurred from data transposition can also be avoided.

The outstanding question is the duration of the first stage of the computation. With a SIMD vector length of ν, $\nu/2$ complex numbers can be held in the vector. These $\nu/2$ complex numbers need to be accumulated together with $\log_2(\nu/2)$ DFT_2 butterflies, which means that there must be $\log_2(\nu/2)$ sets of butterflies in the first stage before we can change the

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

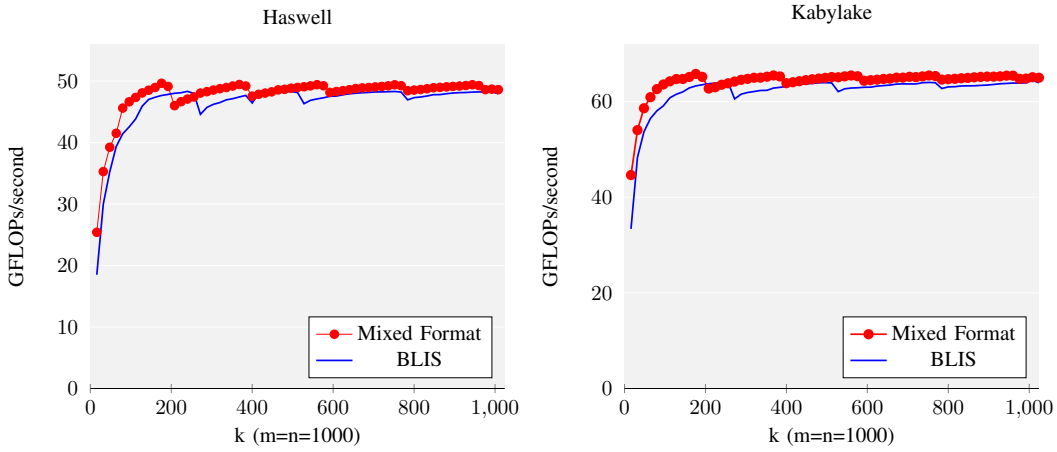

Fig. 6. Performance of double-precision complex matrix multiplication attained on Intel Haswell and Intel Kabylake. The mixed format implementation achieves peak performance faster than the interleaved version as indicated by the steeper slope when the k dimension is small. Overall, as k gets larger, the performance gap between the mixed format and interleaved format decreases from 37% and 33% on the Haswell and Kabylake respectively to 2% .

format. For $\nu = 4$, we require two stages in interleaved format before swapping to the blocked interleaved layout.

IV. Results

In this section, we compare performance results attained by the mixed format implementations against implementations found in commonly-used software libraries. We show performance results obtained on complex mathematical kernels implemented using Advanced Vector Extension (AVX) [10] instruction set on two architecture, the Intel Haswell $4770K$ and Intel Kabylake $7700K$. In all plots, the top of the chart represents the theoretical peak of the machine.

A. Complex Matrix Multiply

In Figure 6, we report performance attained by the different implementations of double-precision complex matrix multiplication. Comparison is performed against implementations in BLIS, as the BLIS framework allows us to selectively replace computation and packing kernels. This also permits us to isolate the performance changes due to the data layout swap. For each architecture, we fixed the output matrix size to 1000×1000, and increases the inner dimension k from 16 to 1024. Performance is reported in GLOPs/second. The theoretical peaks for Haswell and Kabylake are 56 GFLOPs/s and 72 GFLOPs/s.and Kabylake respectively.

Of particular interest is the rate at which the kernel achieves peak performance. This is of interest as a faster rate means that peak is sustained over a larger range of sizes of matrices. As shown in the plots, the mixed format achieves peak performance with a block size of 192, while the traditional mixed format data requires a block size of 256 to reach peak performance. The performance improvement over small values of k ($k < 100$) with the mixed format implementation is between 10-33 and 10-37% over the interleaved implementation on the Kabylake and Haswell respectively.

These results are as expected because the mixed format allows us to compute with almost twice as large a micro-kernel when compared with the interleaved micro-kernel. Since the L1 cache is fixed, a larger micro-kernel means that less data in the k dimension is required to utilize the L1 cache efficiently. The mixed format implementation has also an slight and unexpected benefit of attaining a 1% to 2% improvement over the traditional layout implementation even as the inner dimension of the matrices becomes larger.

B. Discrete Fourier Transform

In Figure 7, we present the performance attained by different implementations of double-precision 1D and 2D FFTs. We compare our approach using interleaved and mixed format against state-of-the-art implementations offered by MKL [9] and FFTW [11]. In addition, performance results for a number of kernels of the form $DFT_n \otimes I_\nu$, where ν represents the vector length. These kernels are chosen as they are similar to the codelets used by FFTW to generate implementations for larger FFT sizes.

We implement our data layout-aware FFTs within the Spiral framework [14]. We use Spiral to generate small 1D FFT codelets of size $n = 64, 128, 256, 512$. The 2D FFT implementations are obtained writing loops around the 1D kernels, similar to FFTW's implementation [3]. We report performance for both the kernels and the 2D FFTs as FLOPs/cycle. Given that FFTs have predominantly more additions than multiplications and/or fused multiply-add instructions, we can make the assumption that FFTs are bounded by additions. Since the Intel Haswell architecture can compute only one vector addition per cycle, we can determine that for a vector length $\nu = 4$ the theoretical peak is 4 FLOPs/cycle. Intel Kabylake has twice the throughput (two vector additions per cycle), therefore the FFT peak performance for the same $\nu = 4$ is 8 FLOPs/cycle.

Recall that the complex multiplication requires two multiplies and one `addsub` to compute the final result. Due to the permutations, both multiply instructions have to be computed before the `addsub` can be executed. Any deviation from the peak performance can be attributed to this dependency.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

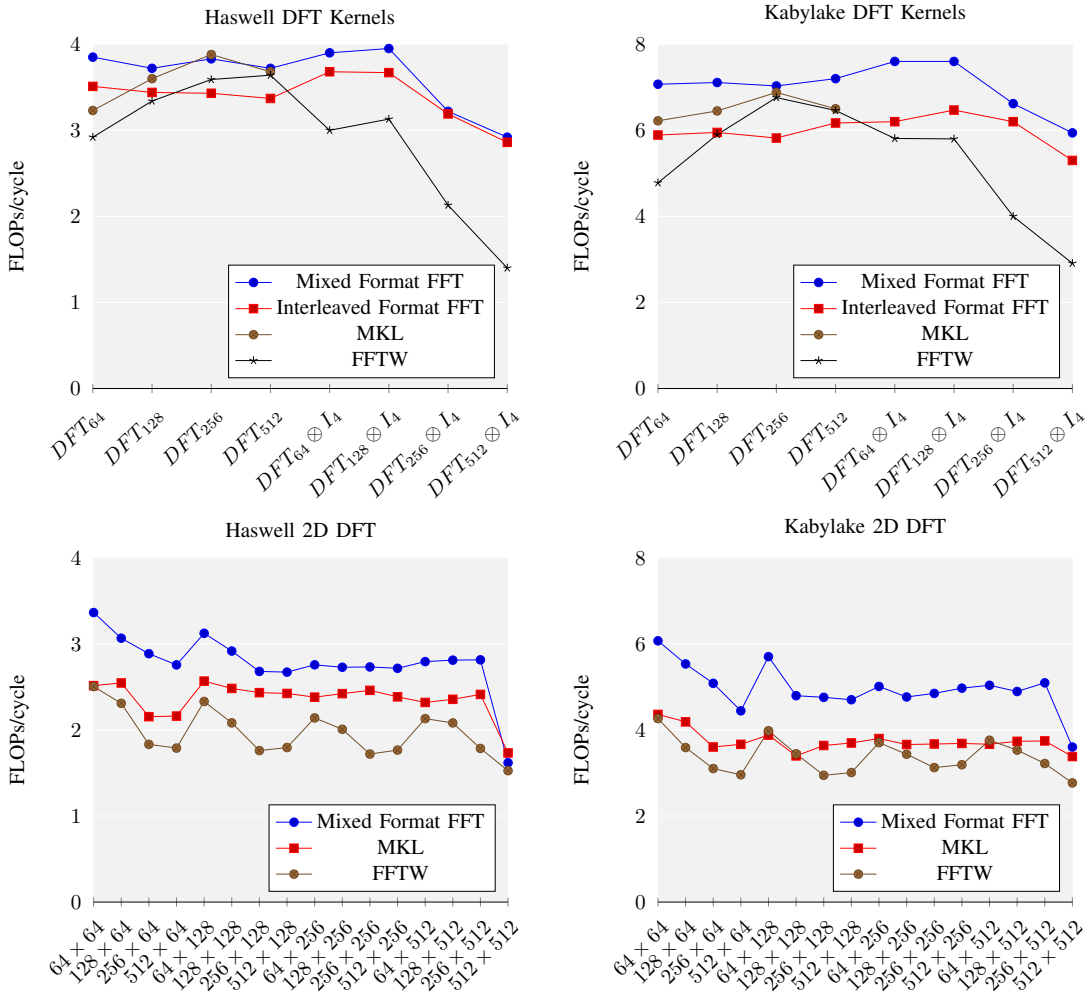

Fig. 7. Performance of double precision 1D kernels and 2D FFTs on Intel Haswell and Intel Kabylake. The mixed format implementations of the 1D FFT kernels (top plots) achieves close to peak performance and match or overpass the library implementations. The mixed format 2D FFTs (bottom plots) achieve between 10% and 30% performance improvement against MKL and FFTW implementations.

1D FFT Performance. Notice that the implementation of the 1D FFTs using the mixed format is slightly better (5% to 15%) then the FFTW and MKL implementations. In addition, the mixed format implementation achieves closer-to-peak performance on both architectures. Similar behavior is observed with the 1D kernels $DFT_n \otimes I_\nu$, though overall performance decreases for when the data is outside of the L1 cache.

As MKL does not offer methods to implement the kernels of the form $DFT_n \otimes I_\nu$, we only compare our implementations against FFTW. Overall, our implementations are 1.3 to 2x faster than FFTWs implementations.

2D FFT Performence. We compare the 2D FFT implemented using the mixed format against MKL and FFTW. The bottom two plots in Figure 7 show the performance of the three implementations on the two architectures. All sizes shown exceed the L1 cache. In addition, the 512×512 reside in main memory. As shown in the plots, the performance of our approach is between 10% and 30% over the implementations offered by two libraries. Notice that as the size of the 2D FFT increases the performance drops since computation requires data to be brought frm the lower levels of the cache hierarchy.

V. Conclusion

In this paper, we describe how switching data layouts for complex numbers within the computational routine can further improve performance of highly-tuned complex arithmetic kernels. By storing the intermediate results in computationally convenient data formats, we reduce (and in some case eliminate) the need for special instructions that are often bottlenecks of many systems. We show that applying our approach to important kernels such as matrix-matrix multiplication and FFT we achieve up to 2x performance improvements over state of the art library implementations. Using important kernels in the linear algebra and spectral domain, we highlight opportunities for improving overall performance of applications from signal processing, machine learning and material sciences where such kernels are used together. One such opportunity for improving performance is preserving the block interleaved data format across computation kernels, thus reducing the overall number of packing and unpacking routines.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

ACKNOWLEDGMENT

This work was sponsored partly by the DARPA BRASS program under agreement FA8750-16-2-003, and NSF through award ACI 1550486. The content, views and conclusions presented in this document are those of the authors(s) and do not necessarily reflect the position or the policy of the sponsoring agencies.

REFERENCES

[1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. of Computation*, 19:297–301, 1965.

[2] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. A rewriting system for the vectorization of signal transforms. In *High Performance Computing for Computational Science (VECPAR)*, volume 4395 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2006.

[3] Matteo Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[4] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):216–231, 2005.

[5] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578. ACM, 1966.

[6] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12:1–12:25, May 2008.

[7] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft.*, 35(1):1–14, July 2008.

[8] Nicholas J Higham. Stability of a method for multiplying complex matrices with three real matrix multiplications. *SIAM journal on matrix analysis and applications*, 13(3):681–687, 1992.

[9] Intel. Math Kernel Library. http://developer.intel.com/software/products/mkl/, 2012.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Sept. 2015.

[11] Steven G Johnson and Matteo Frigo. Implementing ffts in practice. *Fast Fourier Transforms*, 2008.

[12] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[13] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. Analytical modeling is enough for high-performance blis. *ACM Trans. Math. Softw.*, 43(2):12:1–12:18, August 2016.

[14] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[15] Oren Rippel, Jasper Snoek, and Ryan P Adams. Spectral representations for convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 2449–2457, 2015.

[16] Chris-Kriton Skylaris, Peter D Haynes, Arash A Mostofi, and Mike C Payne. Introducing onetep: Linear-scaling density functional simulations on parallel computers. *The Journal of chemical physics*, 122(8):084119, 2005.

[17] Field G. Van Zee and Robert A. van de Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Trans. Math. Softw.*, 41(3):14:1–14:33, June 2015.

[18] Field G. Van Zee and Tyler M. Smith. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Trans. Math. Softw.*, 44(1):7:1–7:36, July 2017.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE