Empirical Auto-tuning Code Generator for FFT and Trigonometric Transforms

Ayaz Ali and Lennart Johnsson
Texas Learning and Computation Center
University of Houston, Texas
{ayaz,johnsson}@cs.uh.edu
Dragan Mirkovic
MD Anderson Cancer Center
The University of Texas, Houston
dmirkovi@mdanderson.org

***Abstract*—We present an automatic, empirically tuned code genenrator for Real/Complex FFT and Trigonometric Transforms. The code generator is part of an adaptive and portable FFT computation framework - UHFFT. Performance portability over varying architectures is achieved by generating highly optimized set of straight line C *codelets* (micro-kernel) that adapt to the microprocessor architecture. The tuning is performed by generating several variants of same size *codelet* with different combinations of optimization parameters. The variants are iteratively compiled and evaluated to find the best implementation on a given platform. Apart from minimizing the operation count, the code generator optimizes for access pattern, register blocking, instruction schedule and structure of arrays. We present details of the optimizations conducted at several stages and the performance gain at each of those levels. We conclude the paper with discussion of the overall performance improvements due to this aggressive approach to generating optimized FFT kernels.**

I. Introduction

The large gap in the speed of processors and main memory that has developed over the last decade and the resulting increased complexity of memory systems introduced to ameliorate this gap has made it increasingly harder for compilers to optimize an arbitrary code within palatable amount of time. The challenge to achieve high efficiency is now becoming more complex through the emergence of multi-core architectures and architectures using "accelerators" or special purpose processors for certain tasks, such as FPGAs, GPUs or the Cell Broadband Engine. To address the challenge to achieve high efficiency in performance critical functions, domain specific tools and compilers have been developed. There is a need for the development of high performance frameworks that aid the compilers by generating architecture conscious code for performance critical applications.

The Fast Fourier Transform (FFT) is one of the most widely used algorithms for scientific and engineering computation especially in the field of signal processing. Since 1965, various algorithms have been proposed for solving FFTs efficiently. However, the FFT is only a good starting point if an efficient implementation exists for the architecture at hand. Scheduling operations and memory accesses for the FFT for modern platforms, given their complex architectures, is a serious challenge compared to BLAS-like functions. It continues to present serious challenges to compiler writers and high performance library developers for every new generation of computer architectures due to its relatively low ratio of computation per memory access and non-sequential memory access pattern. FFTW[7], [6], SPIRAL[13] and UHFFT[12], [11] are three current efforts addressing machine optimization of algorithm selection and code optimization for FFTs.

Current state-of-the art scientific codes use re-usable components and a layered scheme to adapt to the computer architecture by using run-time performance analysis and feedback[14], [15]. At the lower level, the components may use highly optimized sets of straight line parametrized codes (micro-kernels) that are generated to adapt to microprocessor architecture. At the higher level, the parametrization allows for optimal data access patterns to be searched to enhance effective memory bandwidth and lower latency without extensive code optimization. In UHFFT[12], [11], the adaptability is accomplished by using a library of composable blocks of code, each computing a part of the transform. The blocks of code, called *codelets*, are highly optimized and generated by a code generator system called FFTGEN. We use an automatic code generation approach because hand coding and tuning the DFT is a very tedious process for transforms larger than size five. Moreover, by implementing a number of optimizations, we were able to achieve operation counts that were smaller than traditionally assumed for the transform for many sizes.

As shown in Figure 1, eleven different formulas for FFT size 12 result in three different floating point operation counts. In the given graph and the results to follow, we use (Million Floating Point Operations per Second) "MFLOPS" metric to evaluate the performance. We use standard *radix-2* FFT algorithm complexity to estimate the number of floating point operations and then divide that by the running time in micro seconds.

Related Work

Using simple heuristics that minimize the number of operations is not sufficient to generate the best performance FFT kernels; especially on modern, complex architectures.

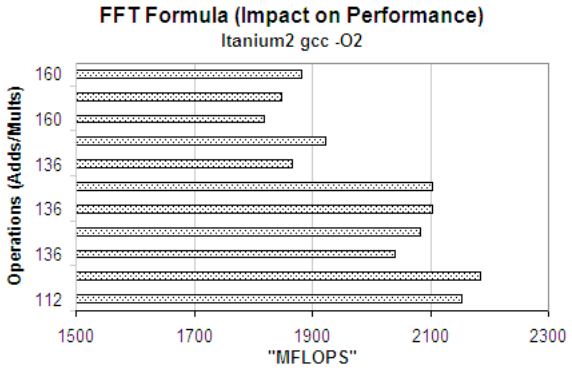

Fig. 1. Performance vs Operation Count for different formulas of size 12 Complex FFT.

Among other factors, instruction schedule and pipelining play an important role in the overall performance. For embedded systems with limited number of registers and instruction cache, it is even more important to employ aggressive optimization techniques to make best use of available resources.

In [9], authors describe an iterative compilation approach to exploring the best combination of tuning parameters in an existing compiler. The empirical technique is shown to yield significant performance improvement in linear algebra kernels. However, the optimizations are mainly focused on loops, which reduces the prospects of major performance gain in small FFT code blocks. More recently, the iterative empirical compilation techniques have been studied [5], [8], [4] on whole applications instead of compute intensive kernels.

An excellent example of automatic generation of tuned linear algebra micro-kernels is given in [14], [15]. The methodology, called Automatic Empirical Optimization of Software (AEOS), employs iterative empirical evaluation of many variants to choose the best performing BLAS routines.

Because of the unique structure of FFT algorithms with output dependence between successive ranks (columns), loop level optimizations are not as effective as in the case of linear algebra codes. The most important tuning parameter for FFT turns out to be memory (register and cache) blocking and the factorization (formula). SPIRAL[13] searches for the best FFT formula for a given problem size using empirical evaluation and feedback. FFTW[7], [6] employs simple heuristics, similar to the one given in section 3, to determine the best formula. Unlike SPIRAL, both FFTW and UHFFT[12], [11] generate micro-kernel of straight line FFT code blocks (*codelets*) and defer the final optimization of a given problem till the run-time. The set of *codelets* (micro kernel), generated at installation time is usually restricted to sizes depending on the size of instruction cache and the number of floating point registers. For some architectures, FFTW also generates ISA specific *codelets* to boost the performance. We believe that such translation should be performed by compiler and the code generator should aid the compiler to achieving that goal. This paper presents an aggressive approach to generating adaptive and portable code for FFT and Trigonometric Transforms by iteratively compiling and evaluating different variants. Apart from exploring the best FFT formula and register blocking, we try limited number of instruction schedules, translation schemes to probe and adapt to both architecture and compiler.

This paper is organized as follows. Section 2 gives the design details and functionality of the code generator. Section 3 describes the automatic optimization and tuning methodology using compiler feedback loop in the UHFFT. Finally, in section 4, we report performance gain due to our new approach and develop models to understand the cache performance of *codelets* for different strides.

II. Design Details

The UHFFT system comprises of two layers i.e., the code generator (FFTGEN) and the run-time framework. The code generator generates highly optimized straight line C code blocks called *codelets* at installation time. These *codelets* are combined together by the run-time framework to solve large FFT problems on Real and Complex data. The type of code to generate is specified by the type and size of the problem. The code generator adapts to the target platform i.e., compiler and hardware architecture by empirically tuning the *codelets* using iterative compilation and feedback. In order to limit the search space of various parameters, tuning is conducted in three phases (stages) and at the end of each phase, values of the parameters are selected. The design overview of FFTGEN is given in Figure 2. FFTGEN2, which implements the new empirical tuning strategy will be integrated in UHFFT version 2.0.1; the beta version is available online and can be downloaded at [1].

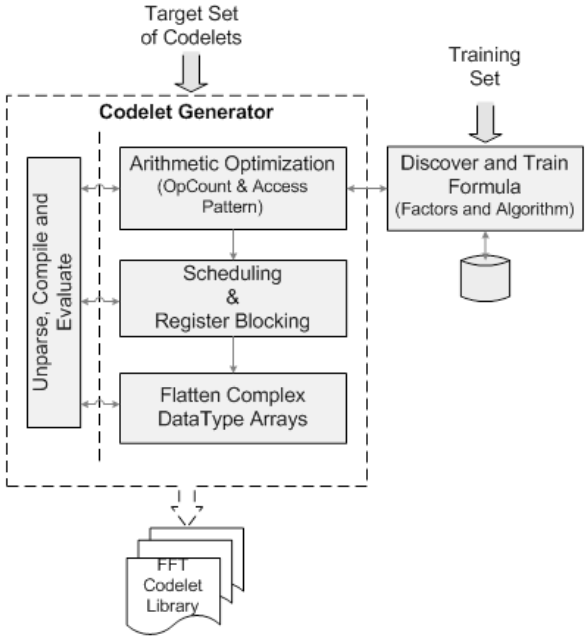

Fig. 2. Fftgen2 Block Diagram

A. *Specifying the Set of* Codelets

The set of desired *codelets* can be defined in a script file, which internally uses FFTGEN2 to generate the *codelets*.

The *codelet* sizes should typically be limited by the size of instruction cache and the number of registers on target architecture. After the set of desired *codelet* sizes is specified in the script file, code generator does not require any further user intervention in order to produce highly optimized micro-kernel of *codelets* for the platform. Nevertheless, an expert user can suggest alternative FFT formulas (factorizations) that FFTGEN2 should try as variants of desired *codelet*. We have implemented a concise language called *FFT Formula Description Language* (FFDL), which is used to describe the FFT factorizations. The code generator supports a subset of FFDL rules as given in Figure 3. Full FFDL specification is part of the UHFFT 2.0.1 run-time framework, which includes multiprocessor and multi-dimensional FFT expressions [3].

#	Productions
1-4	FFT → Module | FFTMR | FFTSR | FFTPF
5-6	FFTPF → FFTPF ***pfa*** Module | Module
7-9	Module → **[*rader*, FFT] *z*** | **[FFT] *z*** | Codelet
10	FFTMR → FFT ***mr*** Module
11	FFTSR → **[*sr* Module, Module] *z***
12	Codelet → **{2,3,4,…,16,32,64 …}**
13	z → $\mathbb{Z}$

Fig. 3. FFT Formula Description Language (FFDL) Rules.

B. Types

The code generator (FFTGEN2) is capable of generating various types of tuned code blocks. As shown in Figure 4, each *codelet* is identified by a string of characters and size. For a given size and rotation, many different types of *codelets* can be generated depending on the following parameters:

Precision: *Codelets* with either double or single precision can be generated depending on this flag.

Direction: *Codelets* with both forward and inverse direction for FFT (complex/real) and DCT/DST Type-I can be generated.

Twiddle: Cooley Tukey FFT algorithm involves multiplication with diagonal twiddle matrix between the two FFT factors. Similar to the approach in FFTW[7], we generate special *twiddle codelets*, which have this multiplication fused inside to avoid extra loads.

I/O Stride: Every *codelet* call has at least two parameters, i.e. input and output vectors. The vectors can be accessed with different strides or same strides as in case of inplace transform. If the strides are same, excess index computation for one of the strides can be avoided by generating a *codelet* that takes only one stride parameter.

Vector Recurse: In most cases multiple vectors of same size need to be transformed. A *codelet* with the vector flag enabled lets user specify the strides (distance) between successive vectors.

Transform: Transform type is specified at the sixth position in the string. Apart from generating FFT and *rotated* (PFA) *codelets*, trigonometric transforms can also be generated by specifying flags '*c*' and '*s*' for DCT and DST transforms respectively.

Datatype: Both real and complex data *codelets* can be generated depending on the transform type.

Rotation: This parameter is only applicable to *rotated* (PFA) *codelets* that are used as part of the Prime Factor Algorithm. Note that a PFA *codelet* with rotation 1 is same as a NON-PFA *codelet*.

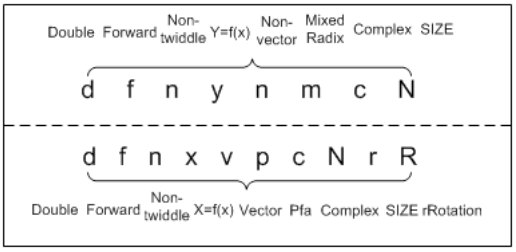

Fig. 4. Two example strings for specifying the type of *Codelet* to be generated by FFTGEN2.

An illustration of two codelet types is given in Figure 4. *Rotated codelets* are useful as part of the PFA execution. Only a few options are applicable to PFA and DCT/DST; for instance, there is no need to generate separate inverse and forward direction *codelets* for PFA execution. Also the *twiddle* flag is ignored except when the *codelet* transform type is FFT (without rotation).

C. Fast Trigonometric Transforms

Trigonometric Transforms have a wide range of applicability, both in signal processing and in the numerical solution of partial differential equations. As given in [10], we have implemented FFT based algorithms for these transforms that require $2.5m \log m$ flops, assuming that m is a power of two. The basic idea is to expand the input vector x to $\tilde{x}$, as given in Table I, using certain symmetries, apply real FFT $F_{2m}\tilde{x}$ of size $2m$ and then exploit the symmetries to generate the *codelet* of complexity $\sim 2.5m \log m$. In our current version we have implemented only Type-I DCT and DST. Extending it to other types of transforms is straightforward for algorithms that are based on FFT.

III. Compiler Feedback Loop

Performance of the *codelets* depends on many parameters that can be tuned to the target platform. Instead of using global heuristics for all platforms, the best optimization parameters are discovered empirically by iteratively compiling and evaluating the generated variants. The optimization is performed in three stages (levels) independently; for example, all factorization policies are not tried for all block sizes. The evaluation module benchmarks the variants and returns an array of performance numbers along with the index of best

TABLE I
DCT/DST - INPUT VECTOR EXPANDED TO USE FFT ($m = 4$)

Transform	x	$\tilde{x}$
DST	$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$	$\begin{bmatrix} 0 & x_1 & x_2 & x_3 & 0 & -x_3 & -x_2 & -x_1 \end{bmatrix}$
DCT	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 \end{bmatrix}$	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_3 & x_2 & x_1 \end{bmatrix}$
DST-II	$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}$	$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & -x_4 & -x_3 & -x_2 & -x_1 \end{bmatrix}$
DCT-II	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \end{bmatrix}$	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_3 & x_2 & x_1 & x_0 \end{bmatrix}$

performing variant. In order to generate statistically stable measurements, each *codelet* is executed repeatedly. *Codelet* are normally called with non-unit strides in the context of a larger FFT problem. Therefore, it's performance should take into account the impact of strided data access. To achieve that goal, the benchmarking data is collected for strides 1 to 64K with increments of 2. We use average over all the samples to represent the quality of a variant. However, the selection policy can be easily extended to incorporate more complicated models.

Level 1: Arithmetic Optimizations

Different FFT formulas are evaluated in this phase to optimize for the floating-point operations and the access pattern. This phase generates the *butterfly computation*, which is abstracted internally as a list of expressions. Simple optimizations such as constant folding, strength reduction and arithmetic simplifications are applied on the list of expressions to minimize the number of operations.

Due to exponential space of possible formulas for a given size, built-in heuristics along with limited user supplied formulas (training set) are tried. Following is the algorithm that is used to generate different formulas that will eventually be evaluated to select the best.

Algorithm 1 FFT Factorization and Algorithm Selection

If $N = 2$ then $algo \leftarrow DFT$ and $r \leftarrow 2$
Else if $IsPrime(N)$ then $algo \leftarrow RADER$ and $r \leftarrow N$
Else
  $FindClosestBestSize\ n$ in Trained Set S
  If $n = N$ then then $algo \leftarrow S[n].algo$ and $r \leftarrow S[n].r$
  Else /* Use Heuristics */
    If n is a factor of N then $k \leftarrow n$
    Else $k \leftarrow GetMaxFactor(N)$
    If $gcd(k, \frac{N}{k})$ then $algo \leftarrow PFA$, $r \leftarrow k$
    Else if $N > 8 \,\&\, 4 \mid N$ then $algo \leftarrow SR$, $r \leftarrow 2$
    Else if $N > k^3$and $k^2 \mid N$ then $algo \leftarrow SR$, $r \leftarrow k$
    Else $algo \leftarrow MR$ and $r \leftarrow k$
  If $FactorizationPolicy \neq LeftRecursive$ then $r \leftarrow \frac{N}{r}$

The algorithm given above is called each time the factorization or algorithm selection is to be made. In the simplest case when N is equal to 2 or when N is a prime number, the algorithm returns the size N as a factor, selecting appropriate algorithm. If the size of *codelet* can be factorized then the training database is queried to find the best and closest size (n) to N such that either n is a factor of N or vice versa. If there is no such record found in the database, heuristics are used that minimize the operation count by preferring algorithms with lowest complexity. Note that when the training knowledge is utilized, the factors are selected in a *greedy* fashion, i.e. the factor with the highest performance is selected.

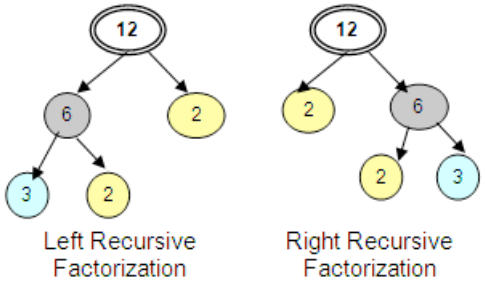

Fig. 5. Examples of Left and Right Recursive Factorization Policies.

TABLE II
OPTIMIZATION LEVEL 1 TUNING PARAMETERS

Variants	Left Recursion	Rader Mode
1-3	0	CIRC,SKEW,SKEWP
4-6	1	CIRC,SKEW,SKEWP

In this phase, six variants are generated depending on the factorization policy, i.e. left recursive or right recursive factorization tree as illustrated in Figure 5. For each type of factorization policy, different rader algorithm options (depending on the convolution solver) are enabled as listed in Table II.

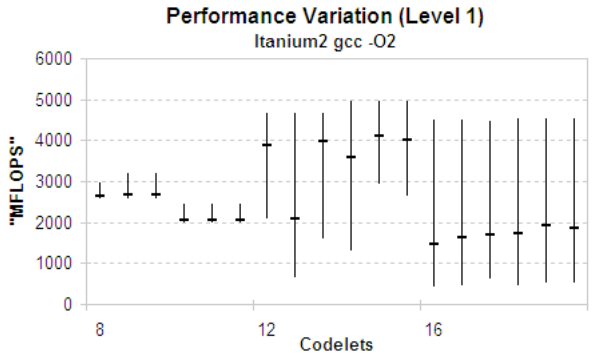

Fig. 6. Level 1 Variants for Real FFT *Codelets* of size 8, 12 and 16. Six variants are evaluated for each codelet for varying strides given by vertical lines. The mean of performance for different strides is used to represent the performance of that variant.

Figure 6 shows performance variation for six variants of each of the three Real FFT *codelets* (8, 12 and 16). Each

vertical bar represents the minimum and maximum performance for that codelet depending on the stride. Notice that rader mode does not bring the performance variation to sizes that are powers of two. Hence only two variants need to be evaluated for powers of two FFTs, ignoring the rader mode parameter.

Level 2: Scheduling and Blocking

Second phase performs scheduling and blocking of expressions by generating a Directed Acyclic Graph (DAG). The main purpose of this scheduling is to minimize register spills by localizing the registers within block(s). A study [2] conducted by the authors of this paper revealed that even for very small straight line *codelets* the performance variation due to varying instruction schedules could be as large as 7%. Finding the best schedule and blocking for all factorizations is a hard problem, hence, only a few possible permutations of instructions and block sizes are evaluated.

TABLE III
OPTIMIZATION LEVEL 2 TUNING PARAMETERS

Variants	Reverse	Blocking
1-3	0	2,4,8
4-6	1	2,4,8

Three different block sizes, i.e. 2,4 and 8 are tried in combination with the option of reversing independent instructions within that block, as given by Table III. Each block is topologically sorted internally in the end. In total, six different variants of schedules and block sizes are generated and the best is selected after empirical evaluation.

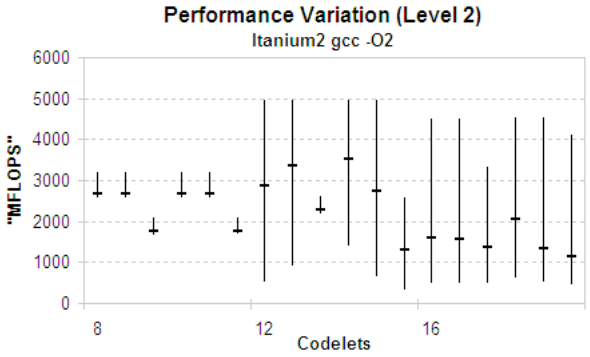

Fig. 7. Level 2 Variants for Real FFT *Codelets* of size 8, 12 and 16. Six variants are evaluated for each codelet for varying strides given by vertical lines. The mean is used to represent the performance of a variant.

As shown in Figure 7, the performance variation at level 2 indicates that smaller block sizes perform better in most cases. There is no clear winner between the two sorting strategies of instructions. The reversing is intended to be a trial and error mechanism that generates some random permutations of instructions.

Level 3: Unparse scheme

All *codelets* take two essential parameters, i.e. input and output vector. For computation of FFT over complex data type, the vectors are represented by arrays of structures. Different compilers behave differently to the structure of input and output arrays. To generate a code that results in the best performance, we tried three different representations for input and output vectors of complex type as given in Figure 8. In the first representation, input and output vectors are accessed as arrays of structure (Complex). In the second scheme, each of the complex vectors is accessed as a single array of Real data type with the imaginary values at odd indices. In the third scheme, the complex vector is represented by two separate real and imaginary arrays of Real data type.

Fig. 8. Input or Output Vector of complex data elements can alternatively be accessed as a single array of interleaved Real/Imaginary data or two arrays of Real and Imaginary data.

TABLE IV
OPTIMIZATION LEVEL 3 TUNING PARAMETERS

Variants	Scalar Rep.	I/O Vector Structure
First	0	Complex, 1 and 2 Real
Second	1	Complex, 1 and 2 Real

Apart from the three array translation schemes for Complex type *codelets*, two more variants are tried for all types of *codelets*, as given in Table IV. As shown in Figure 9, for small size codelets, we noticed that explicit step of replacing I/O vector elements in temporary scalar registers performed better in most cases. However it did increase the total size of code in terms of lines of C code.

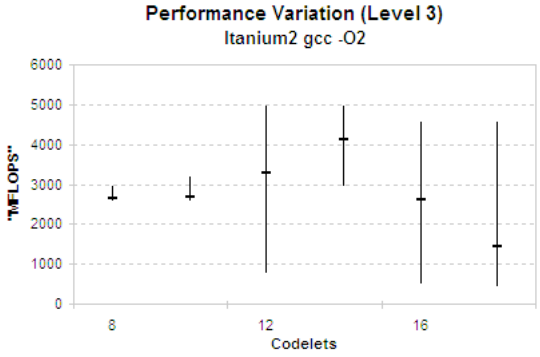

Fig. 9. Level 3 Variants for Real FFT *Codelets* of size 8, 12 and 16. For Real Codelets, only two variants (based on the scalar replacement flag) are evaluated for varying strides given by vertical lines. The mean is used to represent the performance of a variant.

Table V shows the values of parameters selected after evaluating ten variants of Real FFT *Codelet* of size 16.

IV. RESULTS

We performed benchmarking of the complex type FFT *codelets* of sizes that were powers of 2 up to 128 for a range

TABLE V
SELECTED VARIANT FOR *Real* CODELET OF SIZE 16

Option	Selection
Left Recursion	On
Reverse Sort	On
Block Size	2
Scalar Replacement	Off
I/O Structure	One Real Vector

of input and output strides (also powers of 2). Since the sizes of cache lines, cache and memory are mostly equal to some power of 2, we expect to catch some of the worst performance behavior this way. Each reported data item is the average of multiple runs. This was done to ensure that errors due to the resolution of the clock are not introduced in the results presented. The benchmarking and evaluation was carried on two hardware architectures, Itanium2 and Opteron. A summary of the platform specifications is given in Table VI.

TABLE VI
ARCHITECTURE SPECIFICATIONS

	Itanium2	Opteron
Processor	1.5GHz	2.0GHz
Data Cache	16K/256K/6M	64K/1M
Registers	128 FP	88 FP
Inst. Cache	16K	64K
Associativity	4/8/12 way	2/16 way
Compilers	gcc3.4.6/icc9.1	gcc3.4.6/pathcc2.5

In the first set of benchmarks, the results were collected using two compilers for each of the two architectures. Note that same compiler and hardware architecture was used to generate the variants. We compared the performance of empirically tuned *codelets* with that of the *codelets* generated by previous version (1.6.2) of UHFFT to evaluate the efficacy of our new methodology. In the previous version of UHFFT, the *codelets* were generated using simple heuristics that reduced operation count without any blocking or scheduling.

As shown in Figure 10 and 11, there is significant performance improvement for large size Complex FFT *Codelets* for both compilers on Itanium 2. In most cases the performance improvement was seen for small as well as large strides as shown in the graph by vertical lines. Having said that, there is slight degradation of performance for size 16 *codelet*. We understand that the reason behind that could be our simple model for evaluating and selecting the best variant. As the stride is increased the performance of a *codelet* is dominated by the memory transactions which introduces noise in the evaluation of variants. As an alternative to choosing the variant with the best average performance over many strides, a *codelet* with best maximum and minimum performance could be chosen or the evaluation of variants could be limited to low strides so that cache misses are minimized.

In Figure 12 and 13, we performed the same exercise on Opteron using gcc and pathscale compilers. Even though we got some performance improvement for most sizes, the gain was not as much as on the Itanium 2. That may be due to the fact that Itanium 2 has 45% more FP registers than Opteron; thereby affecting the performance of large size *codelets.*

In the second experiment, we compared the performance

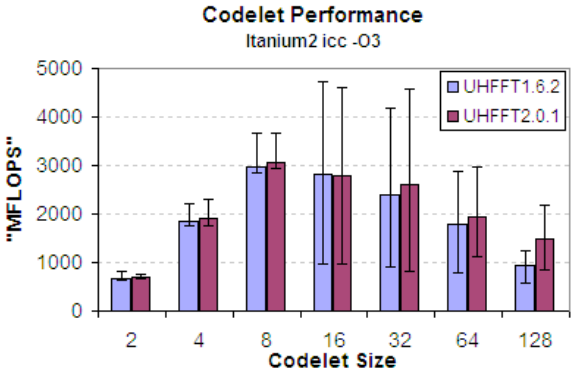

Fig. 10. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Itanium 2 using icc. There is small performance improvement for most codelets. Each vertical error bar represents the variation in performance due to strides.

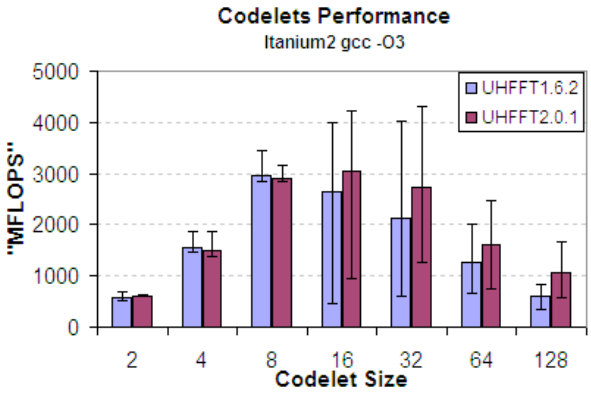

Fig. 11. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Itanium 2 using gcc. There is significant performance improvement for larger size codelets. Each vertical error bar represents the variation in performance due to strides.

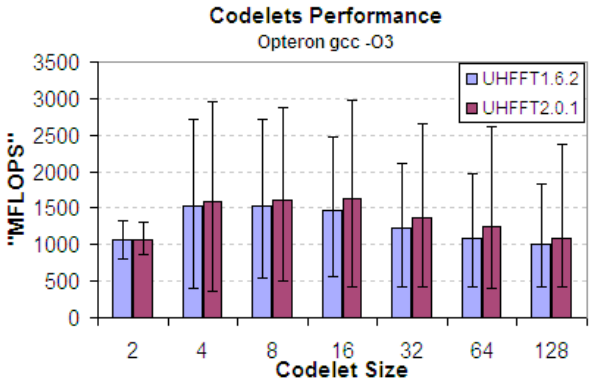

Fig. 12. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Opteron using gcc.

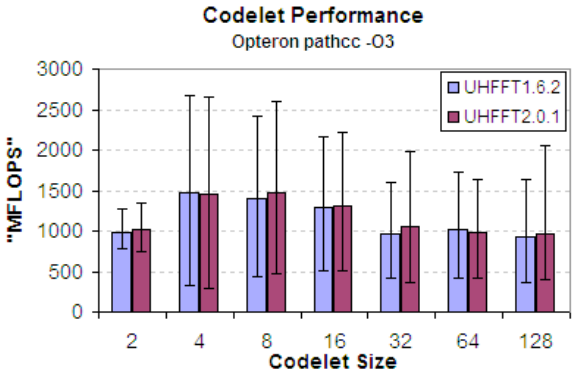

Fig. 13. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Opteron using pathscale compiler.

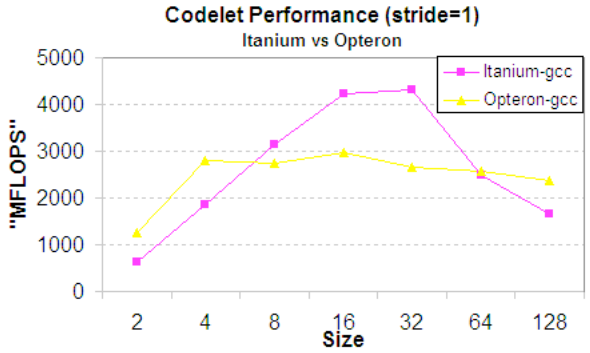

Fig. 14. Impact of size of *codelet* on the performance. Larger codelets suffer from performance degradation due to register pressure and instruction cache misses.

of same *codelet* sizes for unit stride data on both Itanium 2 and Opteron. The performance of *codelets* increases with the size of transform and then starts deteriorating once the code size becomes too big to fit in the instruction cache and registers as shown in Figure 14. Interestingly, the performance decline on Opteron was not as sharp as found on Itanium 2. We believe, that is owing to the bigger instruction cache on Opteron compared to Itanium 2.

For all platforms considered, the performance decreases considerably for large data strides. If two or more data elements required by a particular *codelet* are mapped to the same physical block in cache, then loading one element results in the expulsion of the other from the cache. This phenomenon known as cache trashing occurs most frequently for strides of data that are powers of two because data that are at such strides apart are usually mapped to the same physical blocks in cache depending on the type of cache that is used by the architecture. On Itanium 2, the cache model shown in Figure 15 was harder to predict due to deeper hierarchy and more complex pipelining. However, as shown in Figure 16, for a more conventional architecture like Opteron, the sharp decrease in performance due to cache trashing occurs when:

$$size_{datapoint} \times stride \times 2 \times size_{codelet} \geq \frac{size_{cache}}{Associativity}$$

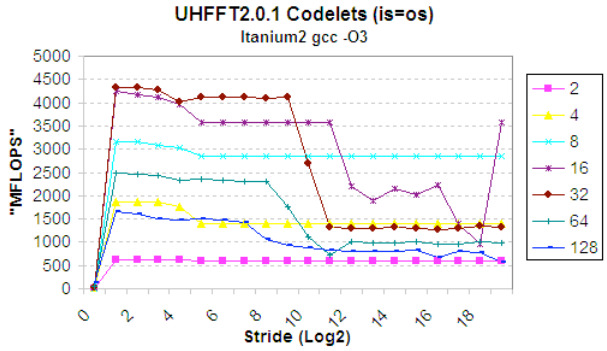

Fig. 15. Cache Performance Model generated using Complex FFT *Codelets* with varying strides on Itanium 2.

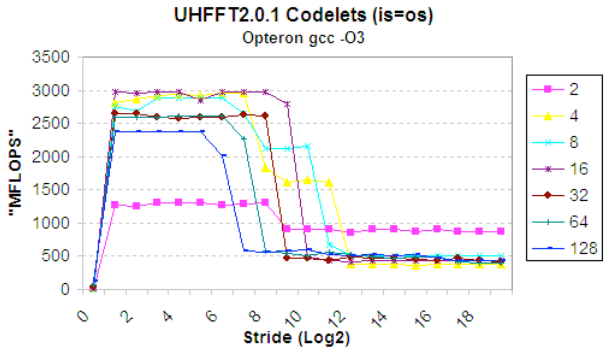

Fig. 16. Cache Performance Model generated using Complex FFT *codelets* with varying strides on Opteron.

where datapoint size is the size of one data element (for complex data with 8 Byte real and imaginary data, each data point is of 16 Bytes), *codelet* size is the number of data elements being transformed by the *codelet*, cache size is the total size of the cache in Bytes, stride is the data access stride, and associativity is the type of cache being used by the architecture.

Conclusion

We have implemented and evaluated the empirical auto-tuning methodology in UHFFT library to generate highly optimized *codelets* for FFT(Real/Complex) and Trigonometric Transforms. The adaptive approach that we have chosen for the library is shown to be an elegant way of achieving both portability and good performance. Internally the code generator implements flexible mathematical rules that can be utilized to extend the library to generate other kinds of transforms and convolution codes, especially when the algorithms are based on FFT. The ease with which the whole UHFFT library tunes itself without user intervention allows us to generate any supported type of transform on any architecture.

References

[1] Uhfft-2.0.1 www.cs.uh.edu/ ayaz/uhfft. 2006.
[2] Ali, A. Impact of instruction scheduling and compiler flags on codelets' performance. Tech. rep., University of Houston, 2005.
[3] Ali, A. An adaptive framework for cache conscious scheduling of fft on cmp and smp systems. Dissertation Proposal, 2006.

[4] ALMAGOR, L., COOPER, K. D., GROSUL, A., HARVEY, T. J., REEVES, S. W., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2004), ACM Press, pp. 231–239.

[5] CHAME, J., CHEN, C., DINIZ, P., HALL, M., LEE, Y.-J., AND LUCAS, R. An overview of the eco project. In *Parallel and Distributed Processing Symposium* (2006), pp. 25–29.

[6] FRIGO, M. A fast fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), ACM Press, pp. 169–180.

[7] FRIGO, M., AND JOHNSON, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] FURSIN, G., O'BOYLE, M., AND KNIJNENBURG, P. Evaluating iterative compilation. In *LCPC '02: Proc. Languages and Compilers for Parallel Computers.* (College Park, MD, USA, 2002), pp. 305–315.

[9] KISUKI, T., KNIJNENBURG, P., O'BOYLE, M., AND WIJSHO, H. Iterative compilation in program optimization. In *Proc. CPC2000* (2000), pp. 35–44.

[10] LOAN, C. V. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[11] MIRKOVIC, D., AND JOHNSSON, S. L. Automatic performance tuning in the uhfft library. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I* (London, UK, 2001), Springer-Verlag, pp. 71–80.

[12] MIRKOVIC, D., MAHASOOM, R., AND JOHNSSON, S. L. An adaptive software library for fast fourier transforms. In *International Conference on Supercomputing* (2000), pp. 215–224.

[13] PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B. W., XIONG, J., FRANCHETTI, F., GAČIĆ, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*, 2 (2005), 232–275.

[14] WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2 (2001), 3–35.

[15] WHALEY, R. C., AND WHALEY, D. B. Tuning high performance kernels through empirical compilation. In *ICPP '05: In Proceedings of the 2005 International Conference on Parallel Processing* (Oslo, Norway, 2005), EEE Computer Society, pp. 89–98.