FFTW and Complex Ambiguity Function Performance on the Maestro Processor

Karandeep Singh, John Paul Walters, Joel Hestness, Jinwoo Suh, Craig M. Rogers, Stephen P. Crago
University of Southern California / Information Sciences Institute
3811 N. Fairfax, Suite 200
Arlington, VA 22203
703-812-3704
{karan, jwalters, jsuh, rogers, crago}@isi.edu

Abstract—Maestro is a 49-core general-purpose, radiation-hardened processor for space. Fourier Transform is an important computation in many space applications. FFTW is a high performance package that computes one and multidimensional Fourier Transforms. We ported FFTW library to the Maestro processor and used it extensively in implementing the Complex Ambiguity Function (CAF), which is a reference space application. We were able to obtain 145 MFLOPS (0.36 FLOPs/Cycle) on a single Maestro core with double-precision computations and 202 MFLOPS (0.50 FLOPs/Cycle) on a single Maestro core with single-precision computations running at 260 MHz.12

TABLE OF CONTENTS
1. INTRODUCTION ...............................................1
2. FFTW ..................................................................1
3. MAESTRO ARCHITECTURE ...............................2
4. MAESTRO FFTW IMPLEMENTATION ..............3
5. EXAMPLE: THE COMPLEX AMBIGUITY FUNCTION .............................................................5
6. CONCLUSIONS ...................................................7
REFERENCES ...........................................................8
BIOGRAPHY ..........................................................8

1. INTRODUCTION

A Discrete Fourier Transform (DFT) transforms the time domain representation of a function to its frequency domain representation, in effect transforming a sequence of values into components of different frequencies. DFTs are widely used in a large number of applications, including for spectral analysis, signal processing, data compression and several mathematical operations like solving partial differential equations, performing convolutions and multiplying very large integers. DFTs are very commonly used computation in many space applications, so a good implementation of this very important function is necessary for low power, high performance requirements in this domain. FFTW [1,2] is a popular package that computes single and multi-dimensional DFTs for both real and complex data. FFTW and its working principles are discussed in detail in Section 2.

Maestro is a many-core, general-purpose, radiation-hardened processor designed for space applications. It is based on the commercial TILE64 architecture from Tilera. The Maestro and Tile64 architectures are discussed in Section 3.

We ported single-precision and double-precision FFTW to the Maestro architecture. Both the single-precision and double-precision versions of the FFTW library are made available. To leverage the parallelism available in Maestro, a pthreads version of the package was also implemented. FFTW has very good performance on Maestro and the performance also scales well with FFT size and with more than one FFT computed in parallel. Our FFTW port to Maestro and its performance characteristics are presented in Section 4.

To demonstrate its use and benefits in real space applications, we used the ported FFTW library extensively in the Complex Ambiguity Function (CAF) [7,8], which is a reference space DSP function used in radar, sonar, and image processing for object tracking. Our CAF implementation and its performance analysis are presented in Section 5.

2. FFTW

A Fast Fourier Transform (FFT) is an efficient algorithm to compute the DFT and its inverse. Computing the DFT naively on N points takes O(N^2) operations, but FFT performs this computation in O(N lg N) operations. The difference in performance is proportional to N/lg N and this can be significant in real world applications. This huge performance difference has made many DFT based applications practical. FFTW (“Fastest Fourier Transform in the West”) is a C subroutine library for computing FFTs. It was developed at the Massachusetts Institute of Technology and is available for free download [2] with a GNU General Public License, but can also be licensed commercially.

FFTW can compute Fourier Transforms of any size for both real and complex data. It supports a wide range of FFT algorithms, and adapts the DFT algorithm according to the hardware in order to maximize performance. It consists of highly optimized and composable blocks of C code called “codelets.” Each codelet either performs a small transform or a part of it, such that codelets can be combined together to obtain bigger transforms. Transform computation is split into

---

1 978-1-4244-7351-9/11/$26.00 ©2011 IEEE.
2 IEEEAC paper #1691, Version 2, Updated January 10, 2011

---
two phases. First, FFTW’s “planner” determines the fastest way to compute the desired transform by running small test algorithms and measuring their performance on the hardware. The planner produces a data structure called a “plan” that contains this information. A plan describes an efficient way to compose codelets in a specific order to obtain the transform. Then, the FFTW “executor” implements the transform on an array of input data as described by the plan.

FFTW supports the concept of “wisdom” that allows a plan to be stored in a file or as a string and reused as many times as needed. A plan is a string that shows the order in which specific codelets need to be executed to compute FFTW of a particular size, and a typical plan is very small in size. FFTW wisdom may contain several plans for desired sizes. In typical space applications, many transforms of the same size are computed, and consequently, a relatively expensive planning stage is acceptable. Moreover, it is not necessary to plan in real time. If FFT sizes are known in advance, then plans can be computed on the ground and stored as wisdom. During deployment, these plans can be used directly for computing the transforms. On the other hand, if FFT sizes are not known in advance, or if plans cannot be stored and reused, the one-time cost of the planner may become significant. For such cases, FFTW provides fast planners based on heuristics.

FFTW is generally portable to platforms with C compilers and provides both C and Fortran interfaces. It also provides parallelized code that can be ported for utilizing parallelism in platforms that support Cilk and for SMP machines that support threads like POSIX threads. A MPI version is also available for distributed systems. It has been ported to several platforms and has been shown to have very good performance characteristics.

3. Maestro Architecture

Tiles

The Maestro processor is a rad-hard-by-design many-core processor for space applications. Maestro has 49 identical cores, called tiles as shown in Figure 1. The 49 Maestro tiles are arranged in a two-dimensional 7x7 array. Each core is

![Maestro Architecture Diagram](image)
Maestro architecture is based on the commercial TILE64 processor [3]. The TILE64 from Tilera is the first commercial, general-purpose many-core microprocessor. TILE64 has 64 tiles arranged in an 8x8 array. The on-chip networks are the same for the TILE64 and Maestro processors.

The main processor in a Maestro core utilizes a three-way Very Long Instruction Word (VLIW) instruction format [4]. Each instruction is 64 bits long. Instructions are packed into bundles of up to three instructions. Bundles are atomically executed. Thus either all of the instructions in a bundle are executed or none of the instructions in a bundle are executed. Maestro performs floating-point operations with a co-processor, which interfaces with the tile processor and memory system through special purpose registers. The FPU consists of three compute pipelines that can run in parallel, and each tile has a FPU co-processor.

Memory Subsystem

Each tile contains a level 1 data cache (L1D), a level 1 instruction cache (L1I), and a shared level 2 cache (L2). L2 caches can be accessed from remote tiles and thus in the aggregate also serve as a level 3 (L3) cache. The memory system also supports cache-coherent shared memory. Each core has a memory management unit for virtual memory support. A DMA engine is also available on each tile.

On-chip Networks

Tiles are inter-connected via multiple two-dimensional mesh networks collectively called iMesh™ [5]. Due to this mesh network topology, every network has a port at every tile. Each tile contains a “Switch Engine” to handle communication over the networks. The switch engines also implement buffering and flow-control within all the networks, so that each tile can process data asynchronously.

There are five inter-tile communication networks on the Maestro chip. The user accessible Static Network (STN) switches scalar data between tiles with very low latency. Data transfers over the static network are pre-programmed by the user using library functions.

The other four on-chip networks are dynamic networks, which facilitate streaming and packet data transfer among tiles and I/O devices. Data to be transferred over dynamic networks is assembled into packets, which also contain the routing information. Packet lengths are between 1 and 128 32-bit data words plus a header word. Of the four dynamic networks (the UDN, TDN, MDN and IDN), only the User Dynamic Network (UDN) is visible to the user. Users can transfer data over the UDN using available library calls. The Memory Dynamic Network (MDN) routes data between cores and between cores and main memory and is accessible only through the caches. The Tile Dynamic Network (TDN) routes memory data between tiles and is also accessible only through the caches. The I/O Dynamic Network (IDN) routes data from cores to I/O interfaces and is accessible from the cores from operating system code.

I/O Architecture

The Maestro/TILE64 processors have four on-chip memory controllers that support 64-bit DDR2 DRAM for a total peak bandwidth of 25.6 GB/s. Each memory controller can operate independently, enabling better memory access concurrency. Any tile can communicate with any memory controller through the on-chip mesh network. Full-duplex XAUI-based 10Gb ports with integrated MACs are also available. Two GbE MAC interfaces, as well as a serial port, HPI, SPI-SROM port, JTAG boundary scan port for testing, and I2C interfaces are also available.

Performance

At the time of submitting this paper, the Maestro hardware is being tested. The performance numbers presented here have been obtained from the Maestro simulator assuming a clock speed of 260 MHz, although operation up to 300 MHz may be possible on the hardware.

4. MAESTRO FFTW IMPLEMENTATION

The FFTW package was cross-compiled for the Maestro platform using the GNU configure and build system on a Linux host. Both single-precision and double-precision libraries were generated. A pthreads version of the library was also ported and is provided to utilize the parallelism offered by the Maestro architecture.
FFTW has a code base of over 300K lines of code. The Maestro compiler does extensive unrolling and the VLIW instruction bundles are nicely filled, which enables good performance without the need for hand optimization.

**Performance numbers**

Figure 2 shows the performance numbers in MFLOPS and FLOPs/cycle obtained using the Maestro simulator for the single-precision and double-precision versions of the FFTW library ported to Maestro. We were able to obtain 145 MFLOPS (0.36 FLOPs/Cycle) on a single tile with double precision computations and 202 MFLOPS (0.50 FLOPs/Cycle) on a single tile with single precision computations running at 260 MHz. Currently the Maestro hardware is being tested at 300 MHz. In that case, the MFLOPS performance numbers shown in Figure 2 would be correspondingly higher. When the Maestro hardware is available, data for bigger sized transforms will be obtained and evaluated.

**Performance Scaling**

Figure 3 shows FFTW performance on Maestro when independent FFTs are computed in parallel on multiple tiles. Performance numbers for three FFT sizes are presented. In these experiments, for the 1, 2 and 4 tile cases, the tiles compute their own plans. Since FFTW generates the best plan based on run-time performance, we get optimal plans and corresponding good performance. But since these experiments are run on a simulator, for the 16 and 40 tiles cases, where simulation times are very high, planning is done on only one tile. This plan is stored as wisdom, which is then used as the plans for the rest of the tiles. We believe the performance would be better if this planning was done in parallel and each
tile executed its own plan. We will test this once the hardware is available.

We also exploited the pthreads back-end to exploit parallelism within FFTs on Maestro. Figure 4 shows the performance scaling for FFT code automatically parallelized using pthreads on one, two and four threads with automatic thread scheduling on the Maestro platform. Performance gains due to pthreads become more significant with the size of the FFT being computed. Further testing on larger FFTs and for more threads will be carried out when the hardware is available.

Performance comparison

Several other efficient FFT algorithms/libraries were also ported to Maestro and their performance was compared to FFTW. FFT code from Numerical Recipes and IEEE DSP Committee, Programs for Digital Signal Processing books were implemented for Maestro. We also ported the mixed radix Kiss FFT [6] implementation.

Figure 5 shows the performance comparison of FFTW with the best performing FFT algorithm for a given size of the other three algorithms mentioned above. FFTW performs significantly better than all three algorithms, across all the sizes tested in this experiment.

Finally, Figure 6 shows the performance difference between the single-tile FFTW and two other well-optimized FFT implementations. These numbers are presented to provide an overall sense of performance across different kinds of FFT implementations. The lowest performance is that of an optimized Radix-4 FFT implementation. Tilera’s Tile64 fixed-point implementation uses SIMD data constructs with 16-bit vector operations, which are different from the floating-point FFTW computations. In general, FFTW performs very well on the Maestro platform.

5. EXAMPLE: THE COMPLEX AMBIGUITY FUNCTION

As an example of FFTW’s use in a real scientific application we present our parallel implementation of the complex ambiguity function (CAF)[7]. CAF computes the time and frequency delay of arrival between two input signals. It is widely used in radar, sonar and image processing for tracking the position and movement of objects for military and space applications. It is computationally challenging, relying
heavily on FFT calculations and is highly parallel, offering a natural fit for the Maestro architecture. In this section we describe our CAF implementation and parallelization strategy as well as the performance measured through simulation of the Maestro architecture.

**CAF Algorithm**

The CAF source code used in our Maestro implementation is derived from work performed by Enright, et al. from The Aerospace Corporation [8]. This work originally targeted multi-core x86 processors using OpenMP.

The implementation is composed of 3 primary computations: Hilbert transforms, channelizing through FIR filtering, and CAF surface processing/detection.

Initially, two input signals (one from each collector) are transformed into a complex stream using the Hilbert transform. Following the Hilbert transform, both complex streams are broken into multiple frequency channels using an FIR filter. Two of these channels (one from each input signal) are used to form CAF surfaces on which the CAF processing and detection computations are performed. FFTW figures prominently in each stage of the computation.

In Figure 7 we present a sample execution profile of the entire CAF processing pipeline. We used this execution profile to guide our optimization and parallelization effort. More than 75% of the total runtime is consumed by FFTW, making a highly optimized FFT library a critical component of the CAF processing pipeline.

**CAF Parallelization**

Our parallel CAF implementation was written using the iLib communications API from Tilera [9]. iLib supports both message passing and shared memory computation. Our implementation used shared memory exclusively. The Hilbert transform and channelization stages divide chunks of the input streams evenly over the total number of tiles participating in the computation. This is an efficient and straightforward loop-level parallelization.

CAF surface processing is embarrassingly parallel. That is, there are no inter-surface dependencies that require synchronization or other communication. This makes a block decomposition strategy, such as that done in the Hilbert transform and channelizing stages the most obvious approach. However, CAF surface processing accounts for more than 60% of the overall runtime of the CAF processing pipeline, and through experimentation was found to be highly susceptible to load imbalance. A naive block decomposition strategy was found to be incapable of keeping tiles busy throughout the computation, particularly in the nominal surfaces case.

The solution was to implement a job queue. Each tile maintains a queue of CAF surfaces. Between the channelizing and CAF processing stages, CAF surfaces are evenly distributed over all tiles. During the CAF surface processing stage, each tile begins by computing on surfaces assigned to its queue. Once the tile exhausts all of the surfaces within its queue, it begins stealing surfaces from other tiles’ queues.

The job queue implementation requires an array of shared memory queues, one per tile. Each queue is protected by a mutex to ensure consistency, and the array of mutexes is shared by all tiles. Thus, for N tiles, we have N queues and N mutexes. Every time a tile removes a surface from its work queue, it acquires its mutex. Once a tile’s work queue has
been exhausted, the tile will attempt to steal surfaces from other non-empty queues. It does this by simply acquiring the mutex associated with the non-empty queue and removing a job. This approach was found to greatly improve the load balance over the lifetime of the computation.

**CAF Performance on Maestro**

In Figure 8 and Figure 9 we present the results of our parallelized CAF implementation for the Maestro processor. The results reported come from a data set with SNR of -7dB with a block size of 1024. 819200 samples were used in these experiments.

We preserved the surface optimization described by Enright et al. [8] where, optionally, surfaces with signal strength less than a predetermined threshold would be discarded without further processing. This reduces the computation substantially. Further, the results shown are for a single simulated CAF iteration. The times shown are simulated times, not wall time.

Results show a baseline implementation, using FFTW’s approximation mode, as well as a wisdom and wisdom + DMA result. The wisdom mode uses pre-computed and stored FFTW wisdom to select the optimal FFT plan at run time. The wisdom + DMA results include the wisdom optimization, and also use the Maestro’s DMA engine during the channelizing phase to improve I/O performance. By using the DMA engine, we are able to perform communication and computation concurrently.

In Figure 8 we present the results for all CAF surfaces. The results show clearly that FFTW wisdom provides a significant advantage to CAF processing. The use of stored wisdom provides a boost of 45-74% for increasing numbers of tiles. Moreover, the use of wisdom proves especially critical for larger tile numbers where the greatest increase in performance due to wisdom is observed. Owing to the large overall processing time, Figure 8 demonstrates minimal improvement when using DMA during the channelization process. This is due to the relatively small contribution of the channelizing process to the overall CAF processing pipeline. However, when discarding those surfaces with power less than the nominal threshold, the ratio of channelizing computation to CAF surface processing tips to favor the channelization process.

In Figure 9 we present our nominal surface results. Again, we notice a substantial improvement due to the use of stored FFTW wisdom. However, we clearly see an improvement due to the use of DMA during the channelization process. This is especially true as the number of tiles increases.

**6. CONCLUSIONS**

Single-precision, double-precision and pthread parallel versions of the FFTW package were ported to the Maestro platform. Performance numbers for FFTW on single and multiple cores were collected using the Maestro simulator. Performance analysis shows that the FFTW library works very well on the Maestro platform. We were able to obtain 145 MFLOPS (0.36 FLOPs/Cycle) on a single Maestro tile with double precision computations and 202 MFLOPS (0.50 FLOPs/Cycle) on a single Maestro tile with single precision computations running at 260 MHz. The FFTW library was used extensively in implementing the Complex Ambiguity Function (CAF), which is a reference space application. About 83% of the total computation time in CAF is used for FFT computations. We showed that by using a job queuing model along with the tile’s DMA engine, we are able to achieve high performance CAF processing through 16 tiles.
REFERENCES


BIOGRAPHY

Karandeep Singh is a Research Programmer at the University of Southern California’s Information Sciences Institute in Arlington, VA. His research interests include high performance computing, embedded systems and cloud computing. He has a MS in Electrical and Computer Engineering from the University of Maryland, College Park.

John Paul Walters is a computer scientist at the University of Southern California’s Information Sciences Institute, located in Arlington, VA. He received his PhD in computer science from Wayne State University in 2007, and his BA in computer science from Albion College in 2002. His research interests include fault tolerance, high performance computing, cloud computing, parallel processing, and many-core architectures.

Jinwoo Suh is a Computer Scientist at the University of Southern California’s Information Sciences Institute since 1999. He received a Ph. D. degree in electrical engineering from the University of Southern California in 1999, an MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea in 1990, and a BS degree in electronic engineering from Dongguk University, Seoul, Korea in 1988. He was an engineer at the research center of the DAEWOO Electronics Co., Ltd, Seoul, Korea from 1990 to 1994. His research interests include cloud computing, fault tolerance, parallel processing, algorithm design, performance optimization, real-time processing, and computer architecture. He is a senior member of the IEEE and IEEE Computer Society.

Stephen Crago is an Assistant Director at the University of Southern California’s Information Sciences Institute, where he currently leads a group focused on multi-core software. He has led development of software and hardware for data-intensive, power-aware, polymorphous, cognitive information, and space-based processing systems. He has been at USC/ISI’s Arlington facility since 1997. He has a BSCEE and MSEE from Purdue University and a Ph.D. in EE from the University of Southern California.