Large Multicore FFTs: Approaches to Optimization

Sharon M. Sacco
MIT Lincoln Laboratory, Lexington MA 02420
ssacco@ll.mit.edu

Abstract

With the end of increasing frequencies, new processor solutions are emerging for increased performance. Multicore is one of the leading choices. The Fast Fourier Transform (FFT) historically has been a challenge to optimize on most architectures. The complexity of multicore programming only increases that challenge. Here the STI Cell Broadband Engine (Cell) is used as an example multicore processor for implementing a 1M point FFT.

Approaching the FFT

Understanding the capabilities of the processor is the first step in designing a highly optimized FFT[1]. Small FFTs fitting with the local store memory (LS) of a single SPE require understanding of the SPE architecture. Mid-sized FFT that fit on multiple LS additionally require a good grasp of the Element Interconnect Bus (EIB). In the case considered here the 32-bit floating point input data (8 MB) exceeds the entire LS of all the SPEs, so XDR memory must be used. Knowing the advantages and limitations of the entire Cell is crucial to this FFT design.

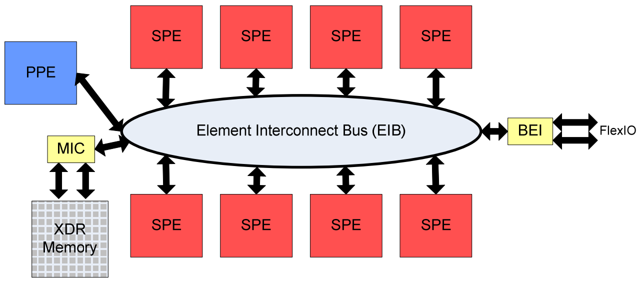

Fig. 1 Cell Block Diagram

Traditional Large FFT

When a 1D FFT's memory requirement exceeds of the smallest usable unit in a system such as data cache or the size is sufficiently large to make a multiple processor solution viable, the 1D FFT is usually cast as a 2D FFT. The FFT data is logically divided into equal segments that are regarded as rows.

The usual method performs a corner turn to compact the columns, performs FFTs over the columns, performs another corner turn to return the modified data to the original order, multiplies the data elementwise by a matrix of weight factors (central twiddles), performs FFTs over the rows, and performs a final corner turn to order the data. Fig. 2 illustrates this method graphically.

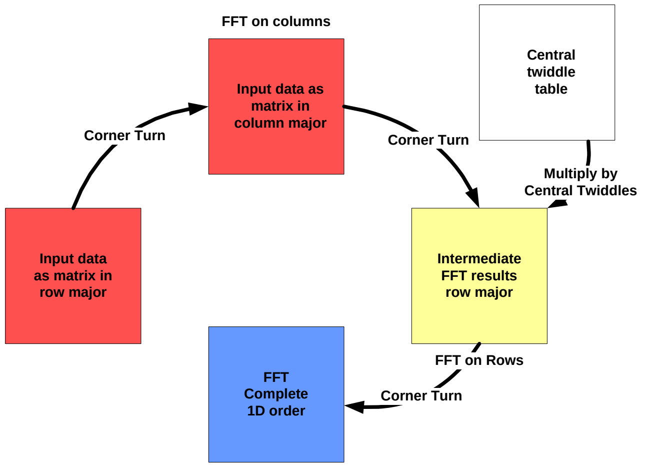
FFT on columns
Input data as matrix in column major
Central twiddle table
Corner Turn
Corner Turn
Multiply by Central Twiddles
Input data as matrix in row major
Intermediate FFT results row major
FFT on Rows
FFT Complete 1D order
Corner Turn

Fig. 2 Computing a 1D FFT as a 2D FFT

This method has very good data locality. Typically the FFT data can fit nicely into the local cache, or blocks of rows can be efficiently distributed among the processors.

Another advantage is it allows code to be reused. Only a single FFT is required. If the FFT is configured as a square matrix, the same weight table is used for all of the FFTs. All of this helps to minimize memory requirements as well as reduce I/O overhead.

There is a major disadvantage to this method. Processor speeds are much faster than communication speed. The timing of the 1D as 2D solution is dominated by I/O.

Minimizing Data Transfers

Recognizing the largest consumers of time for an algorithm is the first step in optimization. For large FFTs minimizing the communication is the priority. The corner turns shown in Fig. 2 have little opportunity to overlap with computation. The first two accommodate the use of a single FFT throughout the algorithm. Rethinking this section of code is key to optimization.

Small FFTs can display the ideas of a design that can then be scaled to larger sizes. Fig. 3 shows the signal flow diagram for a size 16 FFT. This figure uses two processors, red and black. The data is divided between the two processors such that no exchange of data is needed until the middle of the FFT processing. At that point the data is exchanged between the two processors to form new blocks and the computation resumes to completion. Note that for the reorganization the amount of data that needs to be exchanged is less than the total data since each processor gets to retain half its data. For larger FFTs with N processors each processor can retain 1/N of its original data.

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and not necessarily endorse by the United States Government.

[1] The reader's familiarity with Cell architecture is assumed.

This organization is the central idea behind 1D to 2D, but here we want to apply it to the column FFTs. Since these FFTs are relatively small and thus have small twiddle tables, there isn't a good reason to use the central twiddle step since resuming the FFT is as effective.

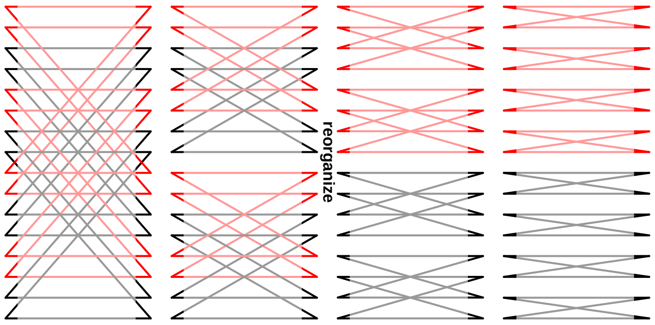

Fig. 3 Signal Flow Graph for Size 16 FFT

Computing a single column at a time would yield very poor performance since the DMA engines are not designed for transfers less than 128 bytes. A better choice is to transfer a block of column data to the SPEs.

Fig. 4 shows an initially distributed band. The band can be viewed as having 32 blocks of 32 rows. Each block is distributed on 4 SPEs. The rows on a single processor are assumed to be contiguous from block to block. Since SPE registers are SIMD, 4 column FFTs are computed together.

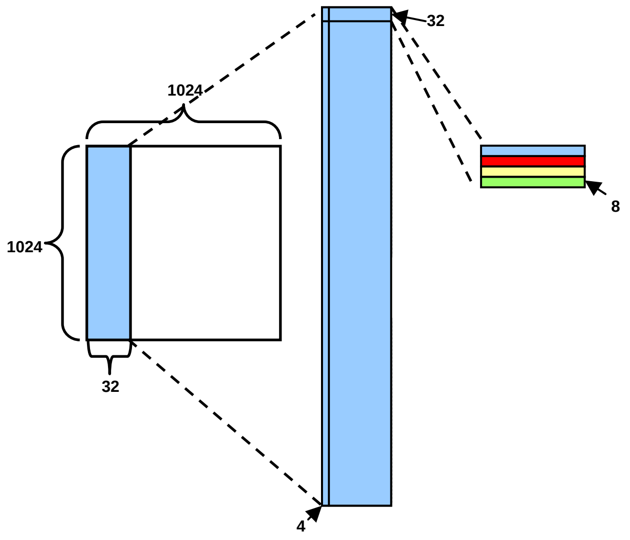

Fig. 4 Band of Column FFTs Using 4 SPEs

Once the initial distribution has been exhausted, the data can be redistributed via the EIB bus with its higher bandwidth. The final distribution consists of blocks of contiguous rows as in Fig. 3. Note that this does not support bit reversal. The access order of the row FFTs in the second half will provide efficient bit reversal.

Once the column bands have been completed, the central twiddles are applied and FFTs are applied to the rows to complete the computation. The row FFTs can be the conventional FFTs such as those provided by vendors. The final step is a corner turn that can be omitted for convolutions.

The advantages to this algorithm is that the lower bandwidth XDR memory accesses (25.3 GB/s read or write) have been reduced from 6 to 2 for the columns. The cost has been additional code and possibly a small twiddle table.

Accuracy and Choice of Algorithm

The Cell designers chose to limit the rounding mode for 32-bit floating point calculations to truncation. The consequence is that there is a bias in these computations. Maintaining accuracy is always challenge for a 1M point FFT, but on Cell it is particularly so.

Since manually correcting for truncation errors is prohibitive, the best solution is to minimize the number of operations in the algorithm. The two common methods of computing, Cooley-Tukey and Gentleman-Sande, will differ in accuracy. Gentleman-Sande will produce some results that are better than Cooley-Tukey, but some output data may have up to 50% more round off error. Cooley-Tukey should give more even error.

With I/O dominating the timings it is tempting to implement a radix 2 algorithm since it has a smaller twiddle table and is easier to implement. Higher radices should produce less round off error since they are more efficient. A radix 4 implementation should produce 15% less error than a radix 2 based on the number of computations.

Estimating Performance

Estimating performance for an FFT requires balancing the computation and the I/O requirements. The starting point is based on the number of floating point operations in a radix 2 FFT, $5\ N \log_2 N$. For a 1M point FFT, this gives roughly 105×10^6 operations. At an optimistic 90% of the peak 205 GFLOPS performance on 8 SPEs we would expect the timing near 0.6 ms compute time.

With 25.3 GB/sec maximum transfer rate, the I/O time dominates. A more realistic 20 GB/sec with a minimum of two complete read and write of the full data set gives 1.7 ms for the I/O. Given more than a factor of two between the compute and I/0 times, this FFT can be computed on 4 SPEs leaving the other 4 as data caches to handle XDR data transfers. The timing should be near 2 ms if handled correctly.

Conclusions

Achieving high performance for FFT is always difficult. The introduction of multicore processors such as the Cell has only increased this difficulty. Older algorithms, while producing slow results, can be the basis for new ideas. In this described version, a 1M FFT should take about 2 ms to execute. This FFT will be built and run to compare with the prediction. This should be completed by the end of summer 2008.

References

[1] E. Oran Brigham, *The Fast Fourier Transform and Its Application*, Prentice Hall, 1988.

[2] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992.

[3] A. Arevalo et al., *Programming the Cell Broadband Engine*, IBM, 2007. (ibm.com/redbooks)