

A Parallel 64K Complex FFT Algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor

Jonathan Greene
jgreene@mc.com

Robert Cooper
rcooper@mc.com

Mercury Computer Systems
199 Riverneck Road, Chelmsford MA 01824
+1 978 256 1300

Introduction

This paper begins to explore how so-called parallel algorithms may be efficiently implemented on the multi-core Cell Broadband Engine processor (Cell). For purposes of this paper, a parallel or collective algorithm is loosely defined as the coordinated processing of a single data set across multiple (i.e., two or more) processing units. Typically, parallel algorithms are used to reduce the latency for processing a particular data set. Less commonly, such algorithms may also increase throughput when applied to successive data sets. This can happen if the individual processing units have either insufficient or poorer quality resources (e.g., memory size, bandwidth) when operating separately rather than collectively on each data set, even if the collective implementation suffers from more overhead, as is typically the case.

Bearing this in mind, we have carefully chosen a class of FFTs for which we believe the best parallel Cell implementations outperform, both in terms of latency and throughput, the best non-parallel implementations. The class is comprised of power-of-2 sized complex, 1D FFTs that happen to “fit” within the collective Local Storage (LS) regions of all eight of the Cell’s SIMD math cores but fail to fit in the LS of any one such core. Here, “fit” is loosely defined to include all resident SPU code, DMA descriptors, FFT “twiddles” (whether fully or partially pre-computed) as well as enough data buffers to permit both efficient processing and overlapped (concurrent) I/O to and from external memory, also known as Main Storage (MS).

In truth, it is a little tricky to precisely pin down “fit”, given that the best individual and parallel FFT implementations vary significantly based on

just how tight the LS fit actually is. There are interesting trade-offs between in-place versus out-of-place FFT primitives, pre-computed versus computed-on-the-fly twiddles, intra-FFT versus inter-FFT overlapped I/O along with other, even more subtle algorithmic choices. Mapping FFT algorithms onto complex computer architectures has seldom been straight forward. The Cell is no exception.

The purpose of this paper is therefore not to offer an exhaustive array of optimal Cell FFT algorithms, even within the class we have loosely defined. Rather, it is to reveal and demonstrate a handful of techniques one might want to consider when striving to develop a parallel algorithm on the Cell. Toward that end, we focus on just one member of the above class: a 64K complex FFT. Specifically, we describe and analyze the performance of a parallel Cell implementation that performs successive 64K complex FFTs on data that both begin and end in Main Storage.

Cell Architecture in a Nutshell

Very briefly, the Cell Broadband Engine Architecture [1] consists of a single 64-bit PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs), all connected together by a high-bandwidth Element Interconnect Bus (EIB). The EIB also connects the MS to all nine processors. (Note that the PPE plays no real role in our FFT implementation so we can pretty much ignore it from here on in.)

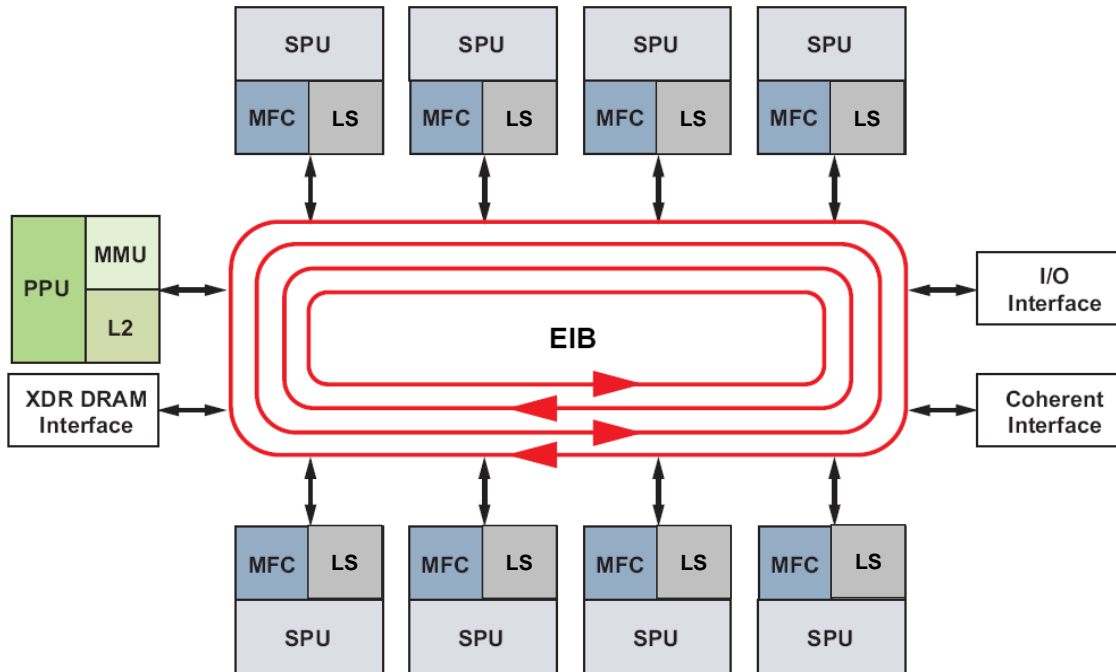


Figure 1: Cell Broadband Engine Architecture

Each SPE contains a Synergistic Processor Unit (SPU), a Memory Flow Controller (MFC) and 256K bytes of LS, used for both the SPU's code and data. The MFC consists of several DMA engines that can be used to transfer data (or code) across the EIB from/to the SPE's LS to/from either the MS or the LS of any of the other seven SPEs. Each SPU contains a 128-bit-wide SIMD engine. The 4-way 32-bit fixed and floating point instructions are capable of delivering 8 FOPs or FLOPs (4 multiply-accumulate instructions) per cycle per SPU.

Here are some of the more relevant architectural features and constraints that inform the design of the parallel FFT implementation described in this paper:

- We assume a **3.0 GHz** Cell clock rate for the purposes of this algorithm design, but note that the Cell processor can operate at higher frequencies
- **256K** bytes of LS per SPE
- $3.0 * 8 = 24$ theoretical GFLOPS per SPU or **192 GFLOPS** for all 8 SPUs
- $3.0 * 8 = \sim 24$ Gbytes/sec maximum sustained bandwidth between the MS and all LS

memories. Allowing for a modest amount of overhead, we chose to de-rate this to a *design* bandwidth of **22 Gbytes/sec**.

- $3.0 * 8 * 8 = \sim 192$ Gbytes/sec maximum aggregate bandwidth of 8 concurrent LS \leftrightarrow LS transfers. We are aware that certain patterns of all-to-all transfers will likely not attain this maximum rate due to the interconnect properties of the EIB but, as we shall later see, the implementation requires less than a quarter of this rate and so should comfortably fall within the limits of even the most pathological all-to-all cases.
- DMA engines:
 - The DMA command queues are generally non-blocking meaning that multiple queued commands can execute concurrently
 - Each DMA command can be assigned a tag identifier to monitor its status (e.g., completion) and to enforce in-order (non-overlapping) execution of two or more commands that share a common tag (i.e., tag group)
 - DMA transfers can overlap SPU processing with little or no performance degradation of either

- To attain optimal bandwidths, DMA transfers must be a multiple of 128 bytes and the source and destination addresses 128-byte aligned
- The DMA is capable of gathering from any remote memory to the local LS or scattering from the local LS to any remote memory at optimal bandwidths, provided that all of the contiguous component transfers satisfy the size and alignment requirements detailed in the prior bullet.

Generic “2D” FFT algorithm for 1D FFTs

The 64K Cell FFT implementation is based on a well-known FFT algorithm [2] that “views” the N -element time domain sequence as a 2D matrix of dimensions NR rows by NC columns (where $N = NR * NC$), logically arranged in row-major order (i.e., elements along each row are contiguous in memory).

The generic algorithm proceeds as follows:

1. Perform NC NR -point “column” DFTs (logically, in-place).
2. Perform an element-wise multiply of the results of step 1 with a $NR \times NC$ complex twiddle matrix, \mathbf{W} , of the form $\mathbf{W}[r,c] = e^{-j(2\pi/N)rc}$ where $j = \sqrt{-1}$ and where $c = 0$ to $NC-1$ and $r = 0$ to $NR-1$.
3. Perform NR NC -point “row” DFTs (logically, in-place).
4. Transpose the $NR \times NC$ results of step 3 to a $NC \times NR$ matrix to produce the correctly ordered results.

For our algorithm $NC = NR = 256$.

A Cell Implementation for Successive 64K FFTs

Data Type and Organization

The complex data is organized in a split fashion in external memory as separate real and imaginary 32-bit floating point vectors of 64K elements each. This split complex organization is maintained within the SPEs, as it generally allows for the most efficient SIMD processing of complex data.

SPU Code

Our 64K FFT implementation requires three optimized SPU “primitives”. Below is a very brief description of each along with its approximate text size:

- **zfft_cols()** performs $4n$ 2^k -point forward FFTs on the columns of a $2^k \times 4n$ split complex matrix. **zfft_cols()** is an out-of-place routine. (~ **5K** bytes of text)
- **zmat_emul_trans()** performs an element-wise multiply of two $4m \times 4n$ split complex matrices and transposes the result to form a $4n \times 4m$ split complex output matrix. **zmat_emul_trans()** is an out-of-place routine. (~ **2K** bytes of text)
- **zmat_out_prod()** performs an outer product multiply of two split complex vectors of length $4n$ and $4m$, respectively to form a $4n \times 4m$ split complex output matrix. (~ **2K** bytes of text)

The implementation also requires a main control program, including initialization, DMA and synchronization routines that together total no more than **16** Kbytes of text. Thus, the code in each SPU adds up to about $9 + 16 = 25$ Kbytes of text.

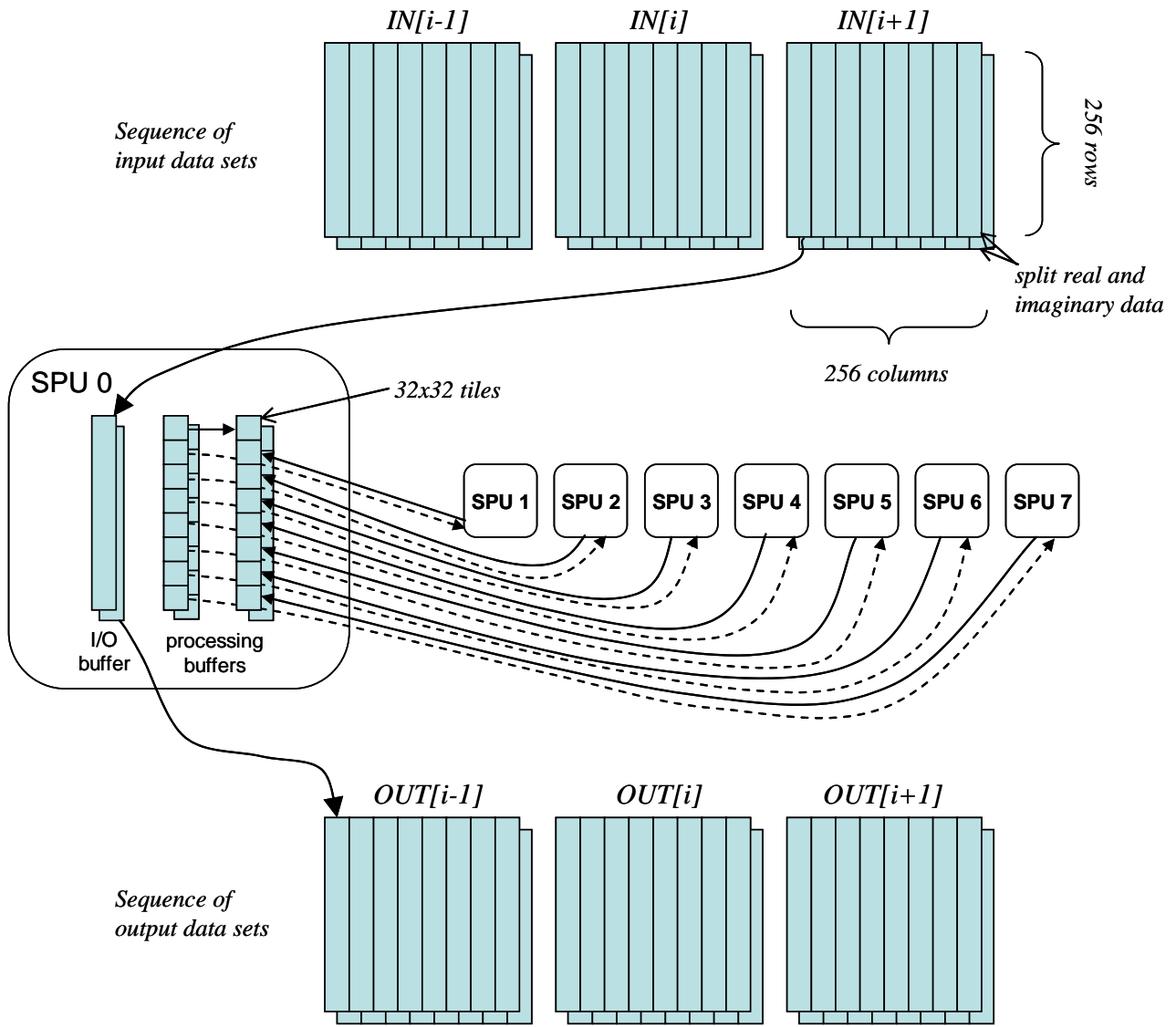


Figure 2: Data layout and movement

SPE Buffering

The following buffer sizes are for a single SPE. The same buffers are required by all eight SPEs although they will mostly contain different data.

Our implementation requires two split data buffers for “ping-pong” processing and inter-SPE transfers and another buffer for I/O to and from MS. The three buffers actually form a triple buffering scheme in which the processing and I/O buffers are effectively permuted with each successive FFT. The I/O buffer generally contains the output from the prior ($i-1^{\text{st}}$) FFT, followed immediately (in time) by the input to the next ($i+1^{\text{st}}$) FFT. It is emptied and filled while the current (i^{th}) FFT is being computed using the two current processing buffers. Thus, transfers to and from MS are overlapped with processing as well as inter-SPE transfers. Each of these buffers is one eighth the FFT size or **64K** bytes, making a total of **192K** bytes for all three.

The 64 sets of pre-computed twiddles for the component 256-point column FFTs require a **1K** byte buffer. Because there is not enough room in the LS to store the complete local set of complex 2D twiddles (256×32), they are instead generated (via complex multiplication) and consumed on the fly in 32×32 element chunks or tiles. This requires an **8K** byte buffer. Additionally, 256 pre-computed complex twiddles (**2K** bytes) are also required to generate the eight tiles of 2D twiddles. In total, then, all the twiddle buffers add up to **11K** bytes.

The implementation must also maintain 2 DMA lists (one for input and one for output), each containing 512 list elements, for a total of $2 * 512 * 8 = \mathbf{8K}$ bytes.

Finally, the software stack and other miscellaneous local storage, including synchronization objects, are conservatively estimated to total another **10K** bytes.

Thus, the total required data size is **192K + 11K + 8K + 10K = 221K** bytes.

Summary of LS Requirements

The combined code and data size per SPE is thus **25K + 221K = 246K** bytes, just barely fitting into the **256K** bytes of LS. One of the three data buffers can possibly be eliminated, thereby saving 64K bytes, if double buffering is used instead of triple buffering. But a double buffered implementation will result in either a substantial reduction in throughput (because I/O to and from MS would no longer be able to fully overlap processing) or a substantial increase in program complexity (because SPU primitives and inter-SPE I/O would have to all be done in-place).

Detailed description of the 64K FFT Cell Implementation

The 64K input buffers (**IN[0], IN[1], ...**) and output buffers (**OUT[0], OUT[1], ...**) should be viewed as 256×256 split complex matrices in the MS.

The three split complex data buffers in each SPE (numbered 0 – 7) are each arranged as two “stacked” vertical, 32-bit floating point matrices of 256 rows by 32 columns in which the first (lower addressed) matrix contains the real component and the second the imaginary component. Making them stacked allows a single DMA list with 512 list elements to transfer both the real and imaginary components either from or to the MS. Although these matrices are split, it is helpful to logically think of each of them as a single 256×32 complex matrix since the real and imaginary components are always operated on in tandem.

For purposes of the ensuing description, it is also useful to think of each of these vertical matrices as being comprised of 8 32×32 element “tiles”, numbered 0 to 7 where the lower numbered tiles correspond to lower LS addresses. So, each tile houses a unique partition ($1/64^{\text{th}}$) of the transforming data matrix based on SPE number and tile number.

Step-by-step description of the Cell implementation

SPU **m** (where “**m**“ is a particular SPU number: 0 – 7):

1. assigns **i** \leftarrow 0.
2. queues a DMA list command to transfer the 256 x 32 sub-matrix in **IN[i]** starting at column **32m** to a contiguous LS data buffer.
3. queues a DMA list command to transfer the 256 x 32 sub-matrix in **IN[i+1]** starting at column **32m** to another contiguous LS data buffer.
4. waits for the DMA transfer queued in step 2 (when **i** = 0) or step 3 of the prior iteration (when **i** > 0) to complete.
5. invokes **zfft_cols()** to perform 32 256-point column FFTs.
6. performs a barrier synchronization with the other 7 SPUs to ensure that all target buffers are available for the DMA transfers that will be queued in step 11.
7. assigns **j** \leftarrow 0.
8. assigns **t_{cur}** \leftarrow **target_lut[m, j]** (**t_{cur}** is a tile number in the range 0 through 7, excluding m).
9. invokes **zmat_out_prod()** to generate the 2D twiddles corresponding to tile **t_{cur}**.
10. invokes **zmat_emul_trans()** to twiddle multiply and transpose tile **t_{cur}**.
11. queues a DMA command to transfer the tile produced in step 10 to **SPE t_{cur}**.
12. assigns **j** \leftarrow **j** + 1 and goes to step 8 if **j** < 7.
13. invokes **zmat_out_prod()** to generate the 2D twiddles corresponding to tile **m**.
14. invokes **zmat_emul_trans()** to twiddle multiply and transpose tile **m**.
15. waits for all 7 of the remote LS \rightarrow local LS DMA transfers queued in step 11 to complete.
16. waits for all 7 of the local LS \rightarrow remote LS DMA transfers queued in step 11 to complete.
17. invokes **zfft_cols()** to perform 32 256-point column FFTs.
18. queues a DMA list command to transfer the 256 x 32 contiguous output from step 17 to the sub-matrix in **OUT[i]** starting at column **32m**.
19. assigns **i** \leftarrow **i** + 1 and permutes the 3 data buffers.
20. goes to step 3.

Additional Implementation Notes

The DMA commands queued in step 18 (LS \rightarrow MS) and step 3 (MS \rightarrow LS) of the next iteration specify the same LS buffer and so must be made to execute in-order (non-overlapping in time).

Step 4 in loop iteration **i** waits for the completion of the back-to-back, in-order DMA transfers queued in step 18 of loop iteration **i-2** and step 3 of loop iteration **i-1**, as these specify the same SPE buffer (first emptied then filled).

The order of the 56 (8 * 7) SPE to SPE (point-to-point) transfers initiated in step 11 is chosen by an 8 x 7 lookup table (**target_lut**), one row per SPE.

The lookup table was determined empirically to minimize the total time required for all 56 transfers. Notice how there is no attempt to explicitly schedule the individual transfers to force them to overlap in rigid time intervals; rather, it is assumed that the EIB will arbitrate these transfers efficiently without such explicit scheduling.

The “ping-pong” processing between two buffers involves both local computation and inter-SPU DMA operations. Therefore, the SPUs are synchronized at key points to avoid race conditions between the SPU processors and the DMA engines.

The 8-SPU barrier synchronization is required in step 6 to ensure that the targets of the DMA transfers that will be queued in step 11 are in fact available.

Steps 15 and 16 are required to ensure that input data tiles and output buffer space is available for the **zfft_cols()** operation in step 17. The input tiles are available once the DMAs from remote SPUs have completed (step 15); the output buffer space becomes available once the DMAs to remote SPUs have completed (step 16).

Performance Estimates

Computation (@ 3.0 GHz)

Function	usecs per call	number of calls per SPU per 64K FFT	Total usecs per SPU
zfft_cols()	17.7	2	35.4
zmat_out_prod()	0.7	8 (once per 32x32 tile)	5.6
zmat_emul_trans()	0.9	8	7.2
Total			48.2

I/O

MS ⇔ LS round trip of 64K elements (512Kbytes) @22 Gbytes/sec	47.7 usecs
LS ⇔ LS all-to-all transfers totaling $7/8^{\text{th}} * 64\text{K}$ elements (448 Kbytes) @ 192 Gbytes/sec	2.4 usecs

Summary

The tile transfers should come for “free” because they will be completely overlapped with the seven sets of **zmat_xxx()** calls needed to produce the seven tiles transferred to remote SPEs. The minimum required bandwidth for the all-to-all transfers to exactly match the computation time is **41** Gbytes/sec. This is well within even the most pathological cases for concurrent SPE ⇔ SPE transfers.

The MS ⇔ LS transfers overlap with the computation, yielding a total throughput of **48.2** usecs and a latency of $47.7 + 48.2 = \mathbf{95.9}$ usecs. Contrast this to an “embarrassingly” parallel Cell implementation in which each FFT is performed separately on each SPE with no SPE ⇔ SPE communication. Since such an implementation requires two full round trips of the data from MS to LS, the maximum throughput is one FFT every $2 * 47.7 = \mathbf{95.4}$ usecs while the minimum latency is $8 * 95.4 = \mathbf{763.2}$ usecs per given FFT, assuming all the computation can be overlapped with I/O.

References

- [1] IBM Corporation. Cell Documents. http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_USA, 2005.
- [2] L.R. Rabiner and B. Gold. Theory and Application of Digital Signal Processing. Prentice-Hall, Englewood Cliffs (NJ), USA, 1975.