'Ultra Large-Scale FFT Processing on Graphics Processor Arrays'

Author: J.B. Glenn-Anderson, PhD, CTO enParallel, Inc.

Abstract

Graphics Processor Unit (GPU) technology has been shown well-suited to efficient Fast Fourier Transform (FFT) processing. An illustrative example is the *NVIDIA CUFFT* library providing capability for GPU-based 1D/2D/3D FFT computations at sample sizes ranging up to 8×10^6. In this paper, a processing model and software architecture are described by which GPU-based FFT computations may be further scaled in both size and speed. In essence, the Ultra Large Scale FFT (ULSFFT) butterfly graph is restructured based upon adoption of preexisting FFT kernels as $RADIX_N$ Butterfly components ('N-Flys') to which ancillary *phase-factor* and *address-shuffle* operators are appended. A supercomputer-styled *scatter-gather* processing model is employed based upon a parallel schedule by which N-fly instances are mapped and queued at multi-GPU Array (GPA) and cluster interfaces. In this manner, FFT sample size is rendered independent of GPU resource constraints. CPU/GPU process pipelining enables optimization of effective parallelization based upon asynchronous work-unit transactions at the GPA/API interface. GPU instruction pipeline reuse is also employed to further amortize I/O transaction overhead. A characteristic 1×10^9 full-complex, $RADIX_{1024}$ ULSFFT design benchmark is then presented and discussed.

Introduction

ULSFFT employs a recursive composition rule whereby an equivalent butterfly network representation may be created in any radix for which **'$N_{SAMPLE} = RADIX^N, N \in Z^+$'** holds. Thus, N-fly network components are expressed as RADIX-sized FFT's augmented by phase factor and address shuffle operators. In this manner, a ULSFFT construct is generally expressed as a butterfly network composed of smaller FFT's.

ULSFFT employs the ePX *scatter-gather* processing model {2} as basis for all butterfly network calculations. This model facilitates joint map/schedule process optimization across all available Cluster/CPU/GPU resources based upon hierarchical integration of associated Distributed, SMP, and SIMD processing models. The feed-forward structure of butterfly networks admits precompiled (static) map/schedule optimization. A generic FFT implementation template is displayed in figure-1, where 'N' distinct I/O ports and internal calculations equivalent to an N^{th} order FFT are assumed for each fly instance (phase factors not displayed). From this network, all associated address shuffles, phase factor vectors, and scatter-gather process synchronization points may be extracted and precalculated. A corresponding parallel-process dataflow is then composed based upon map/schedule processing and work-unit assembly at each N-fly instance.

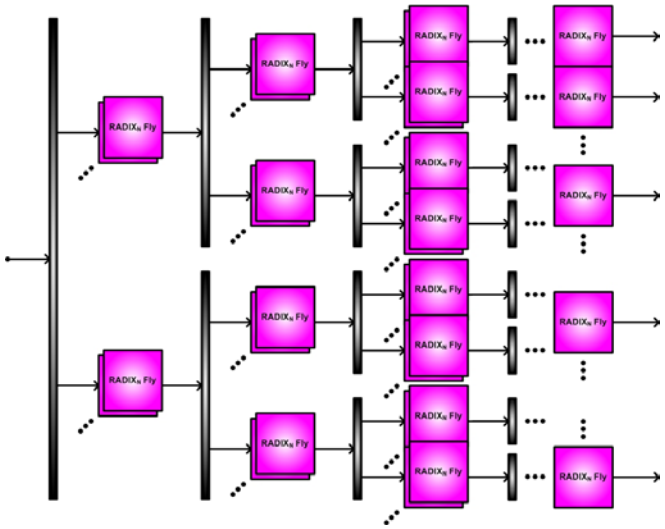

Figure-1: Generic ULSFFT Butterfly Network

Parallel FFT implementations imply datapath concurrency at stage boundaries[2]. This constraint is applied in form of scatter-gather synchronization points in the ULSFFT process network. A generic ULSFFT implementation is displayed in figure-2 where N-fly kernel sequences are mapped and scheduled onto a 4xGPU array; 'scatter-gather' points occur at the beginning and end of stage processing, respectively. In the ePX processing model, 'gather' synchronization points imply SMP update to global memory. In this case, process branches are balanced by virtue of equal numbers of equivalent work-units[3] on each process branch.

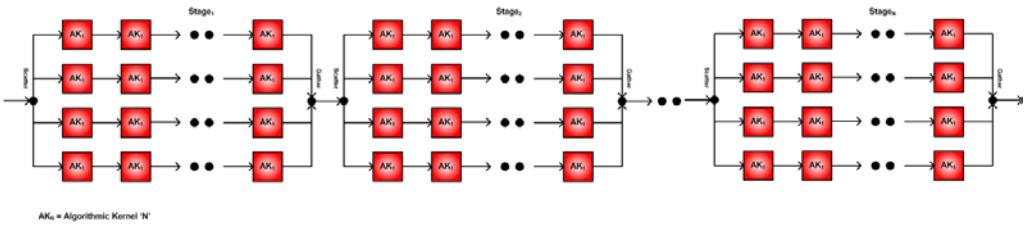

Figure-2: Generic ULSFFT Process Graph (4xGPU)

SIMD threads resident to a GPU resource are associated with a unique 'parent' CPU thread. The asynchronous nature of GPU I/O and process calls admits CPU/GPU pipelining (process overlap) at the parent thread. This feature enables amortization of overhead associated with CPU/GPU I/O, SMP updates to global memory, and application of phase factors. In this manner, effective parallelization is maximized based upon tightest possible packing of process components. Further, under circumstances where GPU calculations remain cyclostatic, I/O overhead may be further reduced based upon N-fly batch processing, (i.e. instruction pipeline 'reuse').

Scaling Properties

With assumption of parallel/pipelined ULSFFT processing across all available resources, ULSFFT speed is expected to scale as a product of architectural parameters: '$A_{ULSFFT} \propto N_{GPU} \times N_{TP} \times N_{CPU} \times N_{NODE}$', ('$N_{GPU}$' = *GPU Array Order*, '$N_{TP}$' = *Thread Processor Array Order*, 'N_{CPU}' = *multicore-CPU/SMP Array Order*, 'N_{NODE}' = *Cluster Node Order*). This relation has been experimentally confirmed up to 16xGPU array size (4xDSC cluster) and analytically extrapolated to 64xGPU array size (16xDSC cluster) based upon optimized process schedule simulations.

The fact a common cache infrastructure services all CPU threads during global memory updates at stage boundaries implies optimally efficient SMP performance is achieved under conditions of statistically perfect load balancing between parent (CPU) threads. Thus, CPU thread priorities must be identical and ULSFFT GPU processes must not be interleaved with other processes, (e.g. 'display'). In figure-3 an example process schedule for a single core implementation is displayed. Note while multithreading is assumed, thread slices are sequentially interleaved on the same processor core. However, as displayed in figure-4, significantly higher performance is generally available to multi-core (hyperthreaded) implementations whereby all pipelined CPU/GPU process components remain essentially parallel.

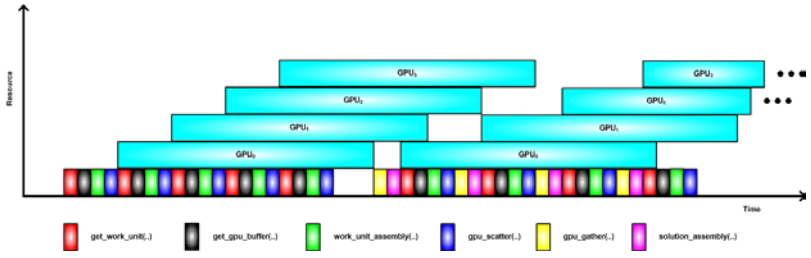

Figure-3: ULSFFT Process Schedule (Single-Core)

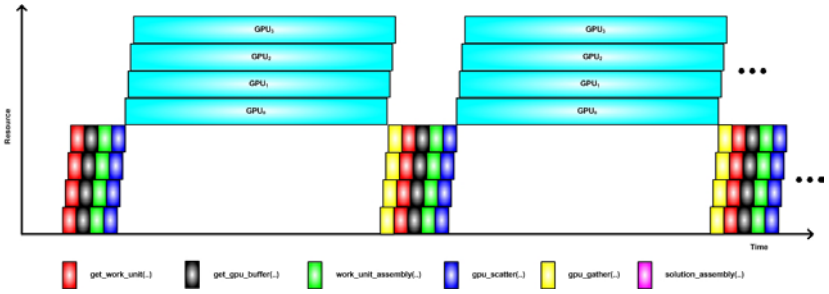

Figure-4: ULSFFT Process Schedule (Multi-Core)

Even higher effective parallelization may be realized based upon GPU-accelerated cluster architecture {3}. In this instance, map/schedule process optimization is extended to cluster node resources with result SMP/SIMD processing at each node is hierarchically integrated under an overarching Distributed scatter-gather processing model. An example

process schedule is displayed in figure-5. Nominal cluster scatter-gather pathways terminating on Node/CPU/GPU resources are displayed in figure-6.

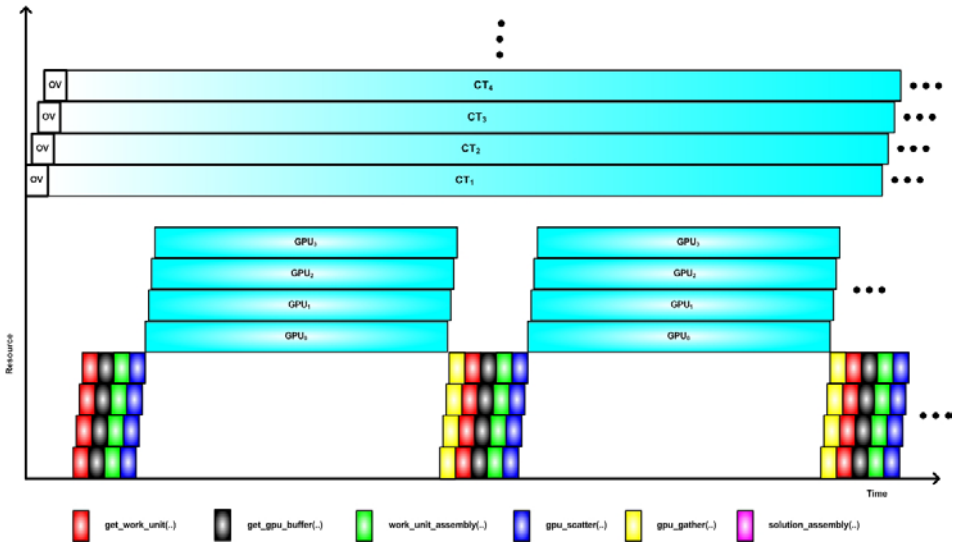

Figure-5: ULSFFT Process Schedule (Multi-Core + Cluster)

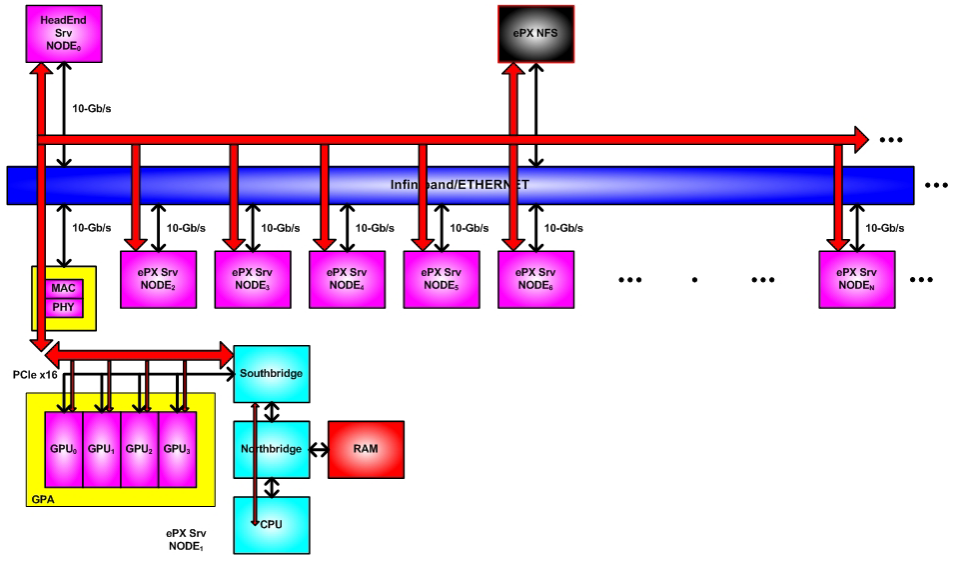

Figure-6: Nominal GPU-accelerated Cluster Scatter-Gather Pathways

The ULSFFT parallel/pipelined process graph displayed in figure-2 is characteristic modulo the selected radix. Thus, as long as a single N-fly can be loaded, an arbitrarily large sample size may in principle be processed. ePX-based ULSFFT applications employ generalized process queue constructs for each GPU instance and work-units are enqueued based upon map/schedule processing. Both process queue and global SMP data-references are memory-mapped based upon pointers to defined address space partitions. These pointers may optionally index host RAM (‘heap’; highest performance)

or HDD-based virtual RAM. In this manner, effectively infinite process queue and global memory resources may be applied to a given ULSFFT design.

Software Architecture

The ePX hierarchical scatter-gather processing model {1} is applied to ULSFFT applications based upon the generic architectural template displayed in figure-7 {2}. In effect, this architecture serves to add full-featured supercomputing infrastructure without requirement for specialized compilation technology, (e.g. 'parallelizing' compilers), or modification to the run-time environment, (e.g. OS-based parallel resource scheduler, scatter-gather manager, MMU). Here the ePX framework features *scheduler*, *dispatcher*, and *scatter-gather engine* components communicating with *generalized process queues* associated with each level of the Distributed/SMP/SIMD hierarchy. In this manner, cluster, multicore CPU, and GPA resources may be efficiently accessed at any processing node. Attached to each process queue are methods communicating with Application Programming Interface (API) components that effectively abstract-away all hardware detail at higher levels of software hierarchy. Thus, ePX software architecture remains more or less uniform across all applications. In nominal configuration, MPI {15}{16}, OpenMP {14}, CUDA {11}, and OpenCL {18} API's are employed. However, alternative API combinations may be supported with little architectural impact, (e.g. AMD CAL/CTM {21}{22}, and PVM {19}). In all cases, only standard OS runtime environments, (e.g. Linux, UNIX, Mac OS-X, and Windows XP/Vista), and development tools, (e.g. GCC, and MSVS), are required.

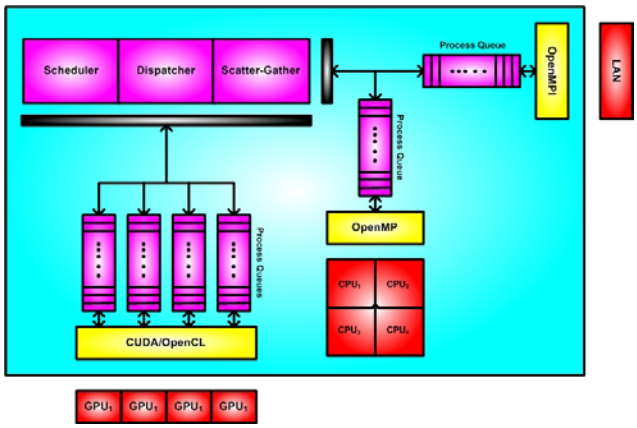

Figure-7: Standard ePX Software Architecture Template

As previously mentioned, the ULSFFT process network is static feed-forward. Thus, the scheduler block may be eliminated based upon precompiled work-unit assembly at all process queues. In figure-7, *dispatcher* and *scatter-gather* blocks are effectively reduced to software-based finite state machines operationally sequenced according to butterfly stage; *dispatcher* dynamically enqueues mapped work-units at a given stage while *scatter-gather* manages thread launch/collection, global memory updates, SMP MUTEX conditions, and thread-control semaphores.

Spectral Precision

The current generation of HPC-grade GPU architectures engender some degree of performance penalty for double precision implementations. Inasmuch as all processing is performed based upon mathematically equivalent butterfly network representations, the ePX architectural template has no direct bearing upon ULSFFT precision. However, accuracy will generally depend upon an assumed butterfly network radix, based upon the '$N_{STAGE} = \log_{RADIX}(N_{SAMPLE})$' structural relation and a defined N-fly ('+', '×') operational mix. From a strictly algorithmic point of view, the ULSFFT approach is recursive, (re: 'divide and conquer'); large FFT's are explicitly decomposed in form of smaller FFT's. However, these 'smaller' FFT's may actually be rendered large given SIMD and memory resource-pools typical of modern GPU architectures. Consequently, algorithm designers may be able to take advantage of high-radix optimizations, resulting in lower accumulated error and thus improved accuracy[1].

This has bearing on ULSFFT designs for which N-Flys are instanced as CUFFT library components {8}. In figure-7, a parametric sweep of CUFFT processing intensity (sample/s) is displayed at $RADIX_{2,4,8,16}$ values[4]. Optimized butterfly structures are expected to exhibit an '$PI \propto N \log_{RADIX}(N)$' performance dependency. However, no such trend is discernable in this data. This suggests possibility of significant ULSFFT optimization over what might be achieved with CUFFT, (e.g. in form of a custom library incorporating RADIX-based optimizations).

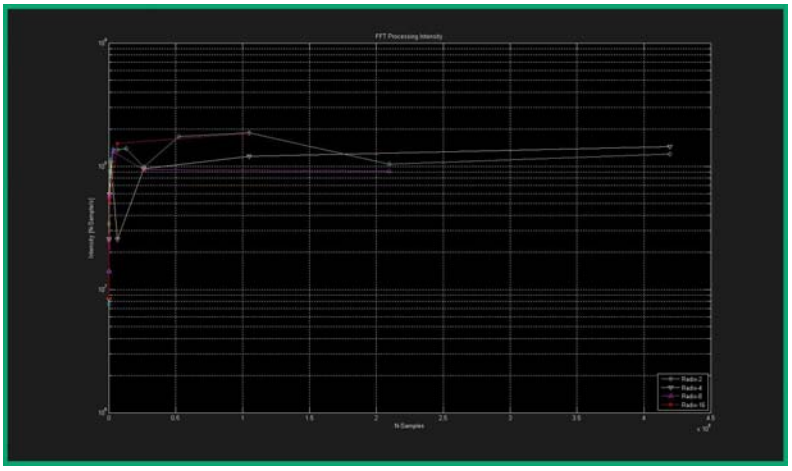

Figure-7: CUFFT(RADIX) Processing Intensity

Multidimensional ULSFFT

The ULSFFT composition rule and hierarchical scatter-gather processing model is directly extensible to the multidimensional case, based upon recursive decomposition of large transforms in form of smaller transforms. However, multidimensional array addressing can render instruction pipeline reuse ('batch' processing) inefficient. {8}. However, an alternative formulation based upon the *generalized row-column algorithm* enables expression of multidimensional FFT's in form of 1D FFT sequences {13}. In this

manner, the CPU/GPU process pipelining and GPU instruction reuse optimizations described above may be applied.

Example ULSFFT Design: N_{1024^3}

An example three stage $RADIX_{1024}$ network with $N_{SAMPLE} = (2^{10})^3 = 1{,}073{,}741{,}824$, (i.e. a 'billion-point' ULSFFT design), is displayed in figure-8. In this case, 4xGPUs are assumed with result '$N_{SAMPLE} / (RADIX \cdot N_{GPU}) = 262144$' N-fly instances per stage are processed at each GPU.

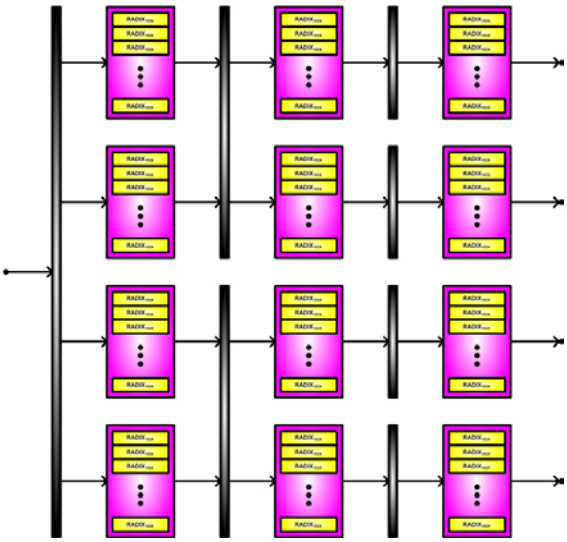

Figure-8: Example 3-Stage $RADIX_{1024}$ ULSFFT Network (4xGPU mapping)

Work units are composed based upon *N-fly*, *Phase Factor*, and *Address Shuffle* operator sequences and distributed to GPU array process queues per the ePX scatter-gather model. In this case, N-fly instances accrue in form of generic '$N_{SAMPLE} = 1024$' CUFFT library components. In figure-9, a complete GPU process queue image corresponding to the 3-stage $RADIX_{1024}$ ULSFFT design is displayed.

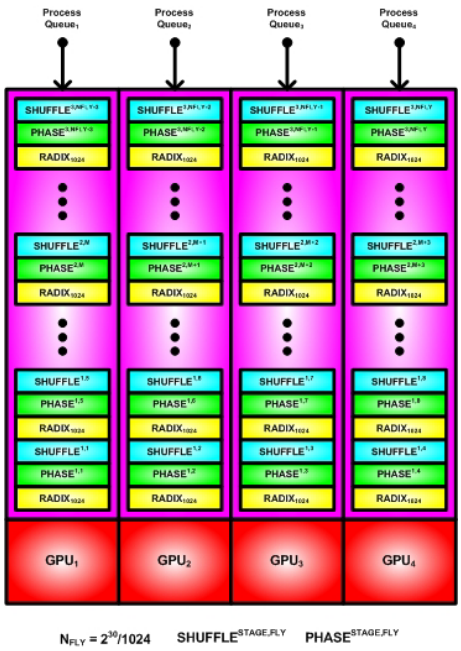

Figure-9: Example ULSFFT GPU Array Process Queue Image

Performance results on a GTX295 NVIDIA 4xGPU array are displayed in figure-10. The upper curve benchmarks single-GPU FFT kernel performance[5] and the single 'starred' result indicates the full 'C2C' complex transform was processed in 1.221 seconds, with a realized effective parallelism of 3.074. Averaging accumulated (single-precision) error over a subsequent inverse transform give a value 2.54×10^{-7}.

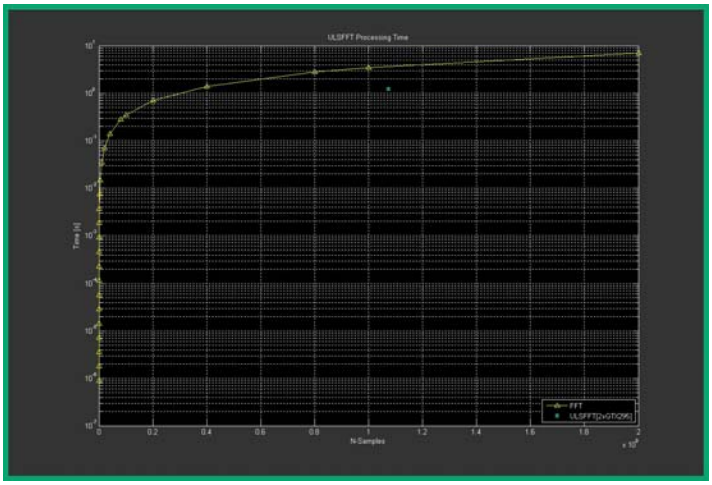

Figure-10: Example 3-Stage $RADIX_{1024}$ ULSFFT Measured Performance

In this case, the ULSFFT process implementation is single-core, (i.e. not hyperthreaded). Thus, reduction from the theoretical maximum '4.0' value is likely due to effective serialization of cache I/O processes at SMP updates.

Summary

A useful technique for GPU-accelerated processing on Ultra Large-Scale FFT (ULSFFT) networks is described. In this case, high-radix butterfly decompositions are leveraged for creation of an efficient scatter-gather processing model on GPU arrays at arbitrary scale. Size-scaling exploits the well-known recursive structure of FFTs, but at radix values optimized for efficient processing on GPU resources. Speed scaling exploits the inherent parallel structure of FFTs, but also incorporates pipelining optimizations by which high effective parallelism may be achieved. In particular, Distributed (Cluster), SMP (CPU), and SIMD (GPU) processing models are hierarchically integrated based upon asynchronous scatter-gather transactions at associated APIs. In this manner, thread processes may be significantly overlapped and the process-schedule maximally packed based upon joint map/schedule optimization applied to all algorithmic components within the process hierarchy.

An illustrative design example is considered in form of a 'billion point' ('$N_{SAMPLE} = 1024^3$') FFT. This design is implemented based upon the enParallel, Inc. 'ePX' architectural framework and mapped to a 4xGPU array consisting of two NVIDIA GTX295 cards. Each GPU features 240 thread processors, 896 MB GDDR3, 448 bit-width memory interface, and 223.8 GB/s memory bandwidth. In this case, the full complex transform was completed in 1.22s, with a realized effective parallelism of 3.07(4.0). Accumulated single-precision error was calculated 2.54×10^{-7}, (i.e. averaged over FFT/IFFT transform).

Note[1] - Optimizations targeted to CPU may be different from that for GPU. In each case, operational overhead is minimized. However, where GPU SIMD processing is considered this minimization must be consistent with a condition of full instruction pipeline coherency, (i.e. no serialization due to logical branching).
Note[2] - Includes datapath pipelining for concurrent sequential processes.
Note[3] - Equivalency implies identical thread architecture and algorithmic kernel implementation.
Note[4] - FFTs are mapped to a non-display GPU in order to assure no confounding due to an interleaved display process.
Note[5] – Process intensity values beyond 8×10^6 CUFFT limit are extrapolated.

References

{1} ***"GPU-based Desktop Supercomputing"***; J. Glenn-Anderson, *enParallel, Inc.* 10/2008
{2} ***"ePX Supercomputing Technology"***; J. Glenn-Anderson, *enParallel, Inc.* 11/2008
{3} ***"ePX Cluster Supercomputing"***; J. Glenn-Anderson, *enParallel, Inc.* 1/2009
{4} ***"GPU-accelerated Multiphysics Simulation"***; J. Glenn-Anderson, *enParallel, Inc.* 9/2009
{5} ***"NVIDIA CUDA Compute Unified Device Architecture – Reference Manual"***; Version 2.0, June 2008
{6} ***"NVIDIA CUDA Compute Unified Device Architecture – Programming Guide"***; Version 2.0, 6/7/2008
{7} ***"NVIDIA CUDA CUBLAS Library"***; PG-00000-002_V2.0, March 2008
{8} ***"NVIDIA CUDA CUFFT Library"***; **
{9} ***"NVIDIA Compute PTX: Parallel Thread Execution"***; ISA Version 1.2, 2008-04-16, SP-03483-001_v1.2

{10} http://www.nvidia.com/object/tesla_gpu_server.html
{11} ***"CUDA Version 2.1"*** download: http://www.nvidia.com/object/cuda_get.html
{12} ***"Principles of Parallel Programming"***; C. Lin, L. Snyder 1st Ed. Addison-Wesley 2008
{13} ***"Inside the FFT Black Box – Serial and Parallel Fast Fourier Transform Algorithms"***, E. Chu, A. George CRC Press 2000
{14} ***"OpenMP Application Program Interface"***; Version 3.0 May 2008 *OpenMP Architecture Review Board* http://openmp.org
{15} ***"MPI: A Message Passing Interface Standard Version 1.3"***; *Message Passing Interface Forum*, May 30, 2008
{16} ***"MPI: A Message Passing Interface Standard Version 2.1"***; *Message Passing Interface Forum*, June 23, 2008
{17} ***"Installation and User's Guide to MPICH, a Portable Implementation of MPI 1.2.7; The ch.nt Device for Workstations and Clusters of Microsoft Windows machines"***; D. Aston, et al. *Mathematics and Computer Science Division, Argonne National Laboratory*
{18} ***"The OpenCL Specification"***; *Khronos OpenCL Working Group*, A. Munshi Ed. Version 1.0, Document Revision 29
{19} ***"PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing"***; A. Geist, et al. MIT Press 1994
{20} ***"Principles of Parallel Programming"***; C. Lin, L. Snyder 1st Ed. Addison-Wesley 2008
{21} ***"ATI Stream Computing – User Guide"***; rev1.4.0a April 2009AMD
{22} ***"ATI Stream Computing – Technical Overview"***; V1.01 2009AMD