'ePX' Supercomputing Technology

James Glenn-Anderson, Ph.D. CTO ***enParallel, Inc.***

Introduction

In contradistinction to standard **G**raphics **P**rocessor **U**nit (GPU) coprocessor use models, *ePX*[1] GPU-based supercomputing technology is grounded in two fundamental objectives; (1) *acceleration of complete applications* and (2) *generic applicability to broad classes of algorithmic kernels*. Satisfaction of these objectives necessarily imply adoption of a full-featured supercomputing processing model for which *scatter-gather*, *dynamic resource scheduling*, *instruction pipeline reuse*, and *CPU/GPU process pipelining* remain key features. While applied here to various forms of the (multi) CPU/**GPU A**rray (GPA) architectural template, (i.e. see *figure-1*), these features generally duplicate those found in traditional supercomputing systems. An ancillary requirement is GPA-based supercomputing must be performed within context of standard Windows/Unix/Linux **O**perating **S**ystem (OS) environments. In particular, no parallelizing compiler or explicit OS runtime support for parallel-process scheduling and management are employed. This is generally motivated by a goal of accommodating supercomputing applications/applications development part and parcel of a generic **P**ersonal **C**omputing (PC) environment, while preserving intact any existing applications code-base. In what follows, *ePX* satisfies all derived design requirements by virtue of incorporating supercomputing infrastructure within a tailored applications framework, based upon existing GPU **A**pplications **P**rogramming **I**nterface (API) libraries {4}{11} and implementing all components of the indicated supercomputer processing model.

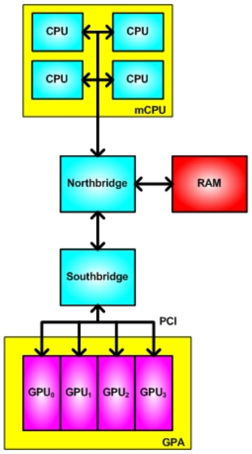

Figure-1: Standard Multi-CPU/GPA Architectural Template

Processing Model

The *ePX* supercomputer processing model is distinguished by optimal scheduling against GPA processing resources. In particular, algorithmic kernels are dynamically mapped to GPU instances (scheduled) based upon; (1) GPU-element availability and (2) opportunistic SIMD instruction pipeline reuse. In this manner SIMD **C**yclostatic **T**hread **R**esidency (CTR) is effectively maximized at any GPU instance, affording highest possible acceleration efficiency. In present context, *‘CTR’* is defined as a measure on the expected proportion of time during which the instruction pipeline is active, (*‘f_{IPL}’* ≡ *Instruction Pipeline Initialization Rate, ‘f_{INSTR}’* ≡ *Instruction Execution Rate*):

$$CTR = \frac{f_{INSTR}}{f_{IPL} + f_{INSTR}} = \frac{\tau_{IPL}}{\tau_{IPL} + \tau_{INSTR}} \qquad (1)$$

Of particular interest are limiting cases:

$$\lim_{\tau_{IPL} \to \infty} \frac{\tau_{IPL}}{\tau_{IPL} + \tau_{INSTR}} = 1 ; \ \lim_{\tau_{IPL} \to 0} \frac{\tau_{IPL}}{\tau_{IPL} + \tau_{INSTR}} = 0 \qquad (2)$$

In the first case, as the characteristic time between instruction-pipeline rewrites increases toward infinity, *CTR* increases to maximum. This implies maximal SIMD efficiency. In the second, as the characteristic time between instruction-pipeline rewrites decreases, *CTR* approaches zero. This implies a corresponding zero SIMD efficiency regardless of the rate at which instructions might be executed. A critical point is GPA resources are aggregated in the GPU coprocessing model and algorithmic kernels applied sequentially according to a serialized dataflow; while the aggregate GPU will indeed exhibit higher SIMD parallelism, *CTR* remains generally far lower. Thus, where complete applications consisting of diverse algorithmic kernels are considered, the supercomputer processing model must be considered superior to the GPU coprocessing model.

The *ePX* supercomputer processing model implements *scatter-gather* distribution of *work-units* to GPA resources according to scheduler state. In this context, the *work-unit* is understood to consist of; (1) processing thread-set and (2) any associated datapath. Note the thread-set may be applied to a GPU instance at initialization or may already exist in situ as result of a previous processing cycle, (re: *instruction pipeline reuse*). Given the aforementioned composition of *work-units* based upon algorithmic kernels, a bipartite parallel-processing scheme is employed whereby kernels are processed in parallel at the GPA transaction buffer and thread-sets are processed in parallel within GPU/SIMD instruction pipelines. This bipartite parallelism critically depends upon the fact *scatter* at the GPA transaction buffer is *non-blocking*. Thus, the CPU does not have to wait for completion of a GPU processing cycle. In this manner, CPU processing and (multi) GPU thread processing may be effectively overlapped, as shown in *figure-2*. Note *gather* remains blocking according to the associated dataflow representation and implied scheduler synchronization semantics.

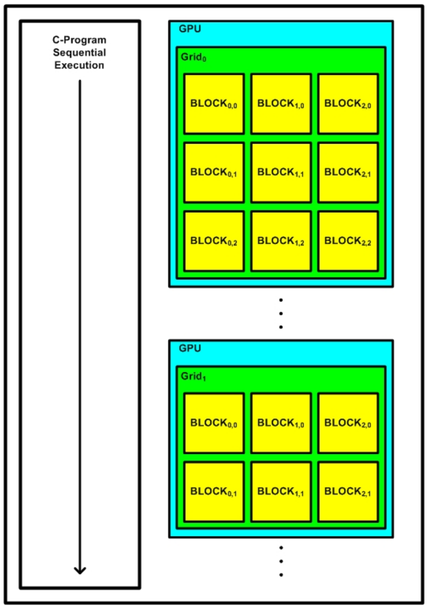

Figure-2: CPU/GPU Pipelining

Workstations based upon the CPU/GPA architectural template plus *ePX* may be clusterized for extended performance scaling {2}{9}. Given obvious bandwidth constraints to be associated with inter-node communications, the expected tripartite parallelization is nominally based upon a *task-kernel-thread* application decomposition. In this manner, I/O constraint boundaries may more easily be placed at associated processing performance constraints, (i.e. no processing element stalls for lack of datapath), and highest performance potential realized.

In contrast to standard supercomputing environments, *ePX* leverages the GPU/API to implement *scatter-gather* infrastructure within the targeted application itself, in form of a generic software framework. Thus, no parallelizing compiler or specialized OS runtime support is required. While the *ePX* framework provides all required *scheduler*, *scatter-gather*, and *CPU/GPA pipelining* functionality, each such component exists in parametric form that must be rendered specific to each application. In effect, the dynamic schedule is hand-tuned for each application based upon (potentially extended) off-line profiling and load-balancing analyses. Thus, while optimally high performance may be achieved for applications appropriate to the indicated supercomputer processing model, *ePX* in present form is not considered a 'turn-key' solution[2]. On the other hand, the *ePX* software framework is indeed usefully generic; structurally common to all derived applications, formulated based upon well established process queue concepts, and extensible to multicore-CPU/GPA architectural templates.

Scheduler

The scheduler establishes a basic organizational schema for all *scatter-gather* and CPU/GPA pipelining according to a process dataflow. A sample dataflow and corresponding process schedule are displayed in *figure-3*. In simplest form, the entire application may be statically scheduled, with implication a complete process schedule may be generated at initialization. However, for many applications of interest the process dataflow is *dynamic*, (i.e. incorporates process conditionals), with result the scheduler is being updated as processing proceeds. While dynamic scheduling within context of parallel-processing lends complexity and potential overhead, the *ePX* processing model flexibly accommodates associated conditionals based upon CPU/GPA datapath state. While any of a multitude of approaches at varying complexity may be employed for scheduler design, the *ePX* scheduler is currently based upon a particularly simple heuristic affording robust and near optimal performance; *"Everything that can be scheduled is scheduled"*. Thus, conditionals associated with kernel synchronization, (re: *scatter-gather*), CPU process pipelining, (re: *work-unit pre/post processing*), and instruction pipeline reuse, (re: *GPA instruction pipeline state*), are updated at each processing cycle according to dataflow elaboration.

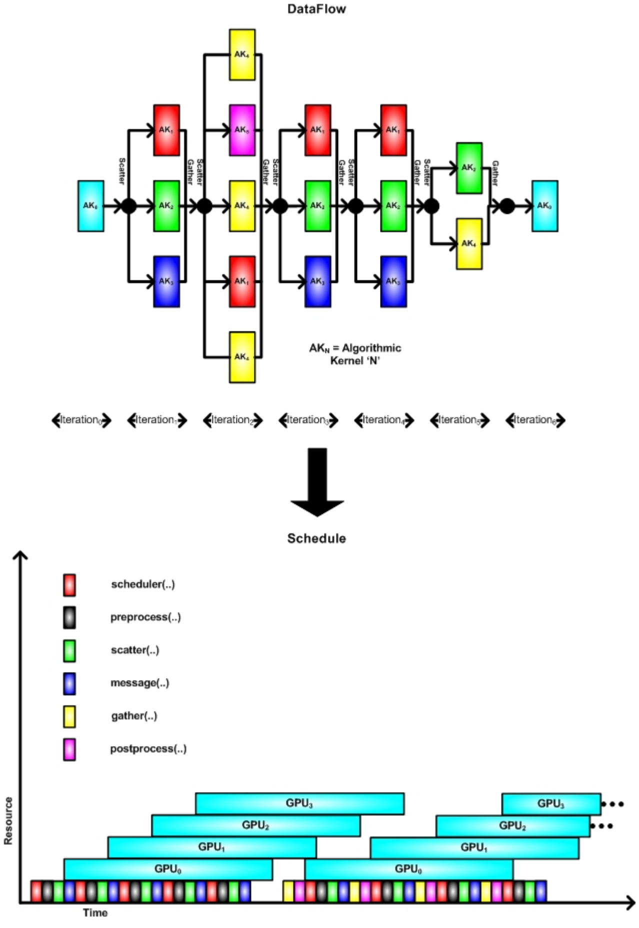

Figure-3: Example *ePX* DataFlow – Process Schedule Mapping

In *figure-3*, a single CPU, 4-GPU array is assumed. Note function calls generated based upon dataflow elaboration result in overlapped processing at all four GPU instances based upon *coarse-grained scatter-gather*. Further, single-CPU/GPA pipelining results in completely sequential CPU processing in parallel with GPA processing. However, where multi-core CPU is considered, some component function calls may in principle be overlapped. A three-fold benefit is thus derived at the process schedule; (1) effective management of finite GPU input buffer effects, (2) CPU processing overhead is effectively hidden, and (3) maximum parallel processing gain is achieved. Under conditions where the number of algorithmic kernels exceeds the number of GPU array elements and equivalent kernels are present, opportunistic scheduling may be employed to reuse SIMD threads without full kernel rewrite. This will serve to further increase SIMD *CTR*, (re: *equation-1*), and thus improve overall processing efficiency, (i.e. note *figure-3* - AK_4 instances at *iteration '2'*). The scatter-gather process may also be embedded with a sequential iteration over parallel code components as expressed in the data-flow diagram. Under such circumstances, ***'scatter(..)'*** will reuse SIMD pipelines across iterations wherever possible, (re: identical kernel index/color-code at successive iterations). In this manner, processing performance may be optimized via extended *CTR* maximization for any associated processing threads.

Scatter-Gather

'Scatter-Gather' is performed at the CPU part and parcel of a bipartite hierarchical parallel-processing schema whereby CPU/GPA concurrency is implemented at the top-level and GPU/SIMD concurrency implemented at the bottom-level. In this form, a critical advantage is realized whereby mapping of software components to processing resources is rendered highly flexible. This flexibility is expressed in form of specific parametric dependencies commonly associated with a full-featured supercomputer processing model. In this context, *scatter-gather* may be applied to generic applications amenable to SIMD processing and tuned for maximum performance.

The *ePX* supercomputer processing model critically depends upon non-blocking 'scatter' at the GPU transaction buffer, as implemented by the GPU/API. This feature enables overlap of GPU processing at all array elements and pipelining of CPU/GPA processes. The result is an effective concurrent process schedule by which true supercomputer performance may be realized. On the other hand, 'gather' operations throughout the dataflow remain blocking, as defined by scheduler synchronization semantics. Scheduler synchronization is defined according to dataflow topological (graph theoretic) structure by which specified inputs must be concurrently present in order for processing to proceed along some dataflow branch.

'Scatter-Gather' work-unit assembly is based upon an algorithmic kernel decomposition created part and parcel of *ePX* applications software development methodology. In particular, *ePX* software design is fundamentally expressed as dataflow with algorithmic kernels mapped to processing nodes. Concurrent kernels may involve task or data parallelism[3] constrained by available GPU device memory and processing resources and optimally sized based upon; (1) SIMD cyclostatic residency, (2) GPA load-balancing, and (3) SIMD instruction pipeline reuse.

CPU/GPA Pipeline

Where the CPU/GPA architectural template is being considered, control is returned to the CPU after any non-blocking *scatter* operation. Thus, function calls at the CPU are processed sequentially according to some variant on the Von Neumann model. In the *ePX* supercomputer model, CPU processing accrues part and parcel of a complete dataflow design. In particular, the complete dataflow is jointly optimized across all CPU/ GPA resources. Further, *ePX* manages CPU and GPA processing in an essentially identical manner; defined WRITE/READ operations are performed on specialized transaction buffers as determined by scheduler state. For example, *ePX* queues *work-unit* pre/post processing operations on a CPU task buffer; when no GPA *scatter-gather* operation is current, CPU tasks are performed. Further important examples include resolution of dataflow conditionals and management of structured interprocess communications between GPU elements.

In this context, *ePX* explicit use of a CPU task queue greatly facilitates scaling to multi-CPU/GPA architectural templates. In such case, the scheduler is extended to include *scatter-gather* to multi-CPU resources in manner virtually identical to that employed for GPA *scatter-gather*[4].

As shown in the example process schedule, (re: *figure-3*), transaction buffer overhead is overlapped with GPA processing. As previously mentioned, non-blocking scatter effectively hides this overhead. However, I/O bandwidth and latencies associated with data transfer and pipeline initialization add to the total interval over which any given GPU resource is unavailable for SIMD processing. Thus, effective processing bandwidth for the complete system is degraded. In particular, where acceleration of complete applications is being considered, reduction of effective parallelism (acceleration) is of concern where GPU instruction pipelines are being reinitialized throughout a dataflow and across diverse kernels. The *ePX* scheduler ameliorates this degradation via SIMD instruction pipeline reuse. For statically scheduled dataflow, reuse is fully specified at initialization. However, *ePX* also features a capability for dynamic reuse, based upon scheduler mapping state over successive processing cycles.

Kernel Dependencies

Optimal *ePX* processing generally results under conditions where; (1) the GPA is maximally load-balanced, (2) CPU processing is maximally pipelined, and (3) mapped kernels are fully encapsulated. In this context, an *encapsulated kernel* is self-contained; all required data and instructions are available at SIMD pipeline initialization. Thus, there is no interprocessor communication, (i.e. *message passing*), or thread resynchronization required for completion of a processing cycle. Generally speaking, use of non-encapsulated kernels in an application dataflow necessarily implies additional CPU processing and resynchronization across CPU/GPU instances; any increased CPU processing gives rise to increased CPU/GPA pipeline overhead and GPU resynchronization reduces equivalent SIMD processing bandwidth. Typical uses of non-encapsulated kernels are; (1) dynamic dataflow elaboration control, and (2) passing intermediate results between GPA elements. While appearance of any overhead is undesirable it is also important to note use of non-encapsulated kernels provides additional resource mapping options. In fact, some applications will require use of non-encapsulated kernels in order to satisfy hardware resource constraints, (e.g. GPA order, device memory, thread processors), or otherwise optimize the process schedule. An essential point is all overhead associated with kernel dependencies must be included in any process schedule optimization.

ePX Framework Pseudo-Code

The *ePX* software framework is abstracted in form of the simple 'C-esque' pCode component displayed below. In essence, all *scheduler*, *scatter-gather*, and *pre/post-process* functionality is combined under an iteration loop polling completion of all function components. Processing begins with dataflow elaboration and resource scheduling, and proceeds sequentially through all functional components. A key subtlety associated with the *ePX* framework is revealed in the fact although function-invocations are performed sequentially at the CPU, associated operations are performed in effectively arbitrary order, number, and 'in-parallel' as determined by the process schedule. This enables use of a wide variety of control paradigms, based upon diverse optimization schemes, applied resource constraints, and desired schedule properties; parallel-process scheduling remains a matter of design choice, as long as synchronization and completeness requirements are satisfied.

```
//
//          Application 'Front-End'…
//
.
scheduler_structure schedule;
.
.
.
do
{
          scheduler(..,&schedule,..);
          preprocess(..,&schedule,..);
          scatter(..,&schedule,..);
          message(..,schedule,..);
          gather(..,&schedule,..);
          postprocess(..,&flag_process,..,&schedule,..);
}
while(flag_process);
.
.
.
//
//          Application 'Back-End'…
//
```

***ePX* Component Function**	**Description**
scheduler(..)	Dynamically elaborates dataflow independently along all branches, consistent with *scatter-gather* semantics as expressed by graph-theoretic structure. Algorithmic kernels along with all associated CPU/GPA processing calls are indexed on a global *schedule* object. All supercomputer processing control is then dynamically generated based upon contents of reserved *schedule*-control fields. Selected *schedule*-state components are also placed in reserved data fields and leveraged within context of process control.
preprocess(..)	Assembles work-unit instances for scheduled kernels in form of pointers to; (1) available GPU-instance, (2) function to be executed, and (3) associated datapath. Where SIMD pipeline reuse is performed, *schedule*-state is examined to determine if the associated function is already present. Completed work-unit assembly is then updated on reserved *schedule*-state fields, subsequently used to conditionally initiate *kernel-scatter*.
scatter(..)	Applies work-units to mapped GPU-instance at the GPA transaction interface; (1) generates all associated WRITE operations at GPU transaction buffer, (re: *kernel-scatter* - *non-blocking* return), and (2) updates *schedule*-state.
message(..)	Manages GPU/GPU - CPU/GPU message-passing or exchange of intermediate datapath based upon interprocess communication permissions, as defined by associated *schedule*-control field; (1) formats message/datapath datagrams, (2) generates associated READ/WRITE operations at GPA transaction buffer, and (3) updates CPU-resident conditionals, (e.g. associated with datapath elaboration).
gather(..)	Applies (blocking) synchronization lock at dataflow 'gather', as defined by associated *schedule*-control field; (1) generates all associated READ operations at GPU transaction buffer, (2) assembles intermediate results for subsequent post-processing, and (3) updates *schedule*-state.
postprocess(..)	Completes processing at dataflow 'gather' as defined by associated *schedule*-control/state fields; (1) applies post-processing operations to complete set of intermediate results, (2) stages result for subsequent work-unit assembly, (3) updates *schedule*-state, and (4) flags completion of all processing on dataflow, (re: *'flag_process'*).

Table-1: *ePX* Framework Component Functions ('pCode')

Amdahl's Law

As previously indicated, *ePX* technology is directed toward acceleration of complete applications. As will be seen from the following simple analysis, this capability is fundamentally dependent upon non-blocking WRITE ('scatter') to GPU I/O streams. The critical feature being CPU processing may continue as soon as a work-unit has been written to the GPU transaction buffer. This in turn enables process pipelining such that GPU work unit assembly/disassembly, I/O, and other CPU-resident processes are effectively 'hidden' on the process schedule. At a sufficient degree of pipelining, GPU performance will effectively dominate system performance. As might be expected, optimal GPU processing gain is then achieved under circumstances whereby SIMD (datapath) processing bounds performance, (i.e. thread processors never stall due to lack of data). At an application level, the maximum achievable speedup is governed by Amdahl's Law; any acceleration (*'A'*) due to thread parallelization will critically depend upon; (1) the fraction of code than can be parallelized (*'P'*), (2) the degree of parallelization (*'N'*), and (3) any overhead associated with parallelization[2]. Thus, expected acceleration is modeled by:

$$A = \frac{1}{(1 - P) + \frac{P}{N}} \tag{3}$$

However, CPU code pipelining, (i.e. overlap with GPU processing), must also be factored into any calculation for 'A'; pipelined code segments processed at the CPU execute at least partially in parallel with GPU code segments. Thus, a modified form of Amdahl's Law must be used:

$$A = \frac{1}{(1 - C_{CPU}P_{CPU} - P_{GPU}) + \frac{P_{GPU}}{N}} \tag{4}$$

$C_{CPU} \equiv$ pipeline efficiency constant $\in [0,1]$, $P_{CPU} \equiv$ pipelined CPU code fraction, $P_{GPU} \equiv$ GPU accelerated code fraction. Of particular interest is the limiting case:

$$\lim_{N \to \infty} \left(\frac{1}{(1 - C_{CPU}P_{CPU} - P_{GPU}) + \frac{P_{GPU}}{N}} \right) = \frac{1}{1 - C_{CPU}P_{CPU} - P_{GPU}} \tag{5}$$

However, with $C_{CPU}P_{CPU}$ sufficiently large, $\frac{P_{GPU}}{N} >> (1 - C_{CPU}P_{CPU} - P_{GPU})$. Amdahl's Law then becomes:

$$A \cong \frac{1}{\frac{P_{GPU}}{N}} = \frac{N}{P_{GPU}} \cong N \tag{6}$$

Where the CPU/GPA supercomputing technique is considered, this relation becomes:

$$A_{CPU-GPA} \cong N_{GPU} \cdot N_{TP/GPU} = N_{GPA} \tag{7}$$

'N_{GPU}' = Number of GPU instances in array, *'$N_{TP/GPU}$'* = Number of thread processors per GPU). An important point is well motivated software architecture design can take advantage of this effect with result acceleration potential for the complete application is greatly improved.

Where **M**ulticore **CPU** (MCPU) is considered within context of CPU/GPA supercomputing, Amdahl's Law assumes a more complex form but is amenable to essentially the same analysis. Initially assuming CPU and GPA processing are not pipelined, Amdahl's Law is expressed as:

$$A_{MCPU-GPA} = \frac{1}{\left(1 - C_{MCPU}P_{MCPU} - P_{GPA}\right) + \frac{P_{CPU}}{N_{CPU}} + \frac{P_{GPA}}{N_{GPA}}} \tag{8}$$

However, when pipelining is employed, $\frac{P_{CPU}}{N_{CPU}} + \frac{P_{GPA}}{N_{GPA}}$ must be replaced with $\frac{P_{GPA}}{N_{GPA}}$. The resulting form for Amdahl's law is then:

$$A_{MCPU-GPA} = \frac{1}{\left(1 - C_{MCPU}P_{MCPU} - P_{GPA}\right) + \frac{P_{GPA}}{N_{GPA}}} \tag{9}$$

As before, with $C_{MCPU}P_{MCPU}$ sufficiently large, $\frac{P_{GPA}}{N_{GPA}} >> \left(1 - C_{MCPU}P_{MCPU} - P_{GPA}\right)$, Amdahl's Law reduces to:

$$A_{MCPU-GPA} \cong \frac{1}{\frac{P_{GPA}}{N_{GPA}}} = \frac{N_{GPA}}{P_{GPA}} \cong N_{GPA} \tag{10}$$

However, CPU multicore parallelization will generally lead to a condition $C_{MCPU} > C_{CPU}$. Thus, where $P_{CPU} = P_{MCPU}$ and assuming identical P_{GPA} and N_{GPA} values, $A_{MCPU\text{-}GPU} > A_{CPU\text{-}GPU}$ implies CPU multicore parallelization will generally result in superior acceleration performance, (i.e. closer to theoretical maximum).

ePX Application Development

A major practical advantage to be associated with *ePX* is only standard Operating Systems[5] are required for application development and deployment. Thus, no changes to an existing PC application base are required to accommodate *ePX* supercomputing applications. Further, tools used for *ePX* application development are based upon standard MSVS/GCC C^{++} compilation technology augmented with GPA/API, *ePX* Framework, and *ePX* mathematics libraries. The complete *ePX* tool suite includes *ePX application builder*[6], *GPA profiler*[7] and *ePX schedule optimizer*[6] application components.

References

{1} ***"Current Trend of Supercomputer Architecture"*** H. Zhang, Univ. Connecticut Dept. of Computer Science and Engineering

{2} ***"GPU Cluster for High-Performance Computing"***, Z. Fan, et al. Center for Visual Computing and Department of Computer Science, ACM/IEEE Supercomputing Conference 2004

{3} ***"SIMT Architecture Delivers Double Precision Teraflops"*** W. Wong, *Electronic Design*, #19280 10Jul08 http://electronicdesign.com/Articles/Index.cfm?AD=1&ArticleID=19280

{4} ***"NVIDIA CUDA Compute Unified Device Architecture – Reference Manual"***; Version 2.0, June 2008

{5} ***"NVIDIA CUDA Compute Unified Device Architecture – Programming Guide"***; Version 2.0, 6/7/2008

{6} ***"NVIDIA CUDA CUBLAS Library"***; PG-00000-002_V2.0, March 2008

{7} ***"NVIDIA Compute PTX: Parallel Thread Execution"***; ISA Version 1.2, 2008-04-16, SP-03483-001_v1.2

{8} http://www.nvidia.com/object/tesla_gpu_server.html

{9} ***"GPU Cluster for Scientific Computing and Large-Scale Simulation"*** Z. Fan, et al. Stony Brook University ACM Workshop on General Purpose Computing on Graphics Processors 2004

{10} http://www.gpgpu.com

{11} ***"A Performance-Oriented Data Parallel Virtual Machine for GPUs"***; M. Segal, M. Peercy, ATI Technologies, Inc.

{12} ***"Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation"***; E. Gabriel, et al. *Proceedings 11th European PVM/MPI Users' Group Meeting* http://www.open-mpi.org

{13} ***"MPI Parallelization Problems and Solutions"*** UCRL-WEB-200945 https://computing.llnl.gov

{14} ***"OpenMP Application Program Interface"***; Version 3.0 May 2008 *OpenMP Architecture Review Board* http://openmp.org

*Note[1] – **en**Parallel **X**cellerated software (ePX) - patent pending.*

Note[2] – Arguably, ePX is capable of superior performance optimization, as compared to any existing turn-key solution.

Note[3] – Not necessarily equivalent to functional description; a single 'large' function may be decomposed into kernels all performing the same operations (re: 'data' parallelism) or some set of 'small' functions may be combined to create a larger function, (re: 'task' parallelism).

Note[4] – Any extant global data dependencies are resolved within context of scheduler synchronization semantics.

Note[5] – WinXP, Linux, and Unix currently supported.

Note[6] – Currently enParallel, Inc. proprietary, (i.e. internal use only).

Note[7] – 'NVIDIA CUDA Visual Profiler', Version 1.0.