‘GPU-accelerated Multiphysics Simulation’

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

Abstract

In recent technology developments General Purpose computation on Graphics Processor Units (GPGPU) has been recognized a viable HPC technique. In this context, GPU-acceleration is rooted in high-order Single Instruction Multiple Data (SIMD)/Single Instruction Multiple Thread (SIMT) vector-processing capability, combined with high-speed asynchronous I/O and sophisticated parallel cache memory architecture. In this presentation we examine the enParallel, Inc. (ePX) approach in leveraging this technology for accelerated multiphysics computation {1}{2}{3}.

As is well understood, both complexity and size impact realizable multiphysics simulation performance. Multiphysics applications by definition incorporate diverse model components, each of which employs characteristic algorithmic kernels, (e.g. sparse/dense linear solvers, gradient optimizers, multidimensional FFT/IFFT, wavelet, random variate generators). This complexity is further increased by any requirement for structured communications across module boundaries, (e.g. dynamic boundary conditions, multi-grid (re)discretization, and management of disparate time-scales). Further, multiphysics applications tend toward large scale and long runtimes due to; (a) presence of multiple physical processes and (b) high-order discretization as result of persistent nonlinearity, chaotic dynamics, etc. It then follows acceleration is highly motivated, and any associated performance optimization schema must be sufficiently sophisticated so as to address all salient aspects of process resource mapping and scheduling, and datapath movement. For the GPU-accelerated cluster, this remains a particularly important consideration due to the fact GPU lends an additional degree of freedom to any choice of processing resource; multiphysics performance optimization then reduces to a goal of achieving highest possible effective parallelism across all available HPC resources, each of which is associated with a characteristic process model.

In ePX applications, processing models are organized hierarchically so as to structurally minimize high-overhead interprocess communications; process optimization is then performed based upon an assumed scatter-gather principle recursively applied at distributed (cluster) and Symmetric Multi-Processor (SMP; multicore CPU) hierarchy levels. This approach supports flexible optimization across all physics modules. In particular, explicit pipelining of cluster, CPU, and GPU processes is implemented based upon asynchronous transaction calls at an associated Application Programming Interface (API). This generally improves effective parallelization beyond what might otherwise be possible. Further, a complete multiphysics application must be accelerated consistent with dictates of Amdahl’s Law. In this context, ePX is shown to exhibit a full-featured supercomputer processing model, highly optimized for GPU-accelerated clusters or workstations, and particularly well suited to multiphysics applications. The ePX framework is then presented as a generic and reusable multiphysics development solution

featuring scatter-gather infrastructure as a software architectural component and obviating any need for specialized compilation technology or OS runtime support.

Introduction

HPC is the dominant enabling technology for advanced multiphysics simulation. Thus, dramatic improvements recently observed in HPC price/performance have rendered multiphysics applications a practical reality at unprecedented scale and complexity. This trend is based in emergence of enterprise cluster computing in the HPC market sector, concurrent with appearance of new multicore CPU and GPGPU technologies.

By definition, multiphysics simulation involves expression of multiple physical processes. Further, distinct mathematical formalisms may be employed for each process, with coupling amongst those processes imposed at defined boundaries and all within context of a single computational process. Thus, where multiphysics simulation is considered, complexity is increased over that of a scale-equivalent unitary physics simulation on at least two axes; (1) algorithmic kernel diversity, and (2) dynamic boundaries between processes. The cited dynamic boundaries may also engender rediscretization on coupled processes, further impacting problem representation scale and complexity. A characteristic example is the ORNL MFIX application combining (TDFD) *computational fluid dynamics* and *discrete event* physics modules {4}. The upshot is simulations involving unitary physics remain generally far easier to optimize than those involving multiphysics. In what follows, a new software architecture is presented as basis for optimized multiphysics simulation on GPU-accelerated clusters and workstations.

Hardware Architecture

GPU architecture

We assume the NVIDIA GPU as a more or less generic architectural template {4}. As displayed in *figure-1*, GPU processing resources are organized as an assembly of ‘N’ distinct multiprocessors, each of which consists of ‘M’ distinct thread processors. In this context, multiprocessor operation is defined modulo an ensemble of threads scheduled and managed as a single entity, (i.e. ‘warp’). In this manner, shared-memory access, SIMT instruction fetch and execution, and cache operations are maximally synchronized. Here, *SIMT* is distinguished from *SIMD* by virtue of the fact hardware vector organization is not exposed at software level and programmers are enabled to flexibly compose parallel code for both independent and coordinated (data-parallel) threads. While certainly a useful innovation, a subtle complexity is also introduced in that programmers must assume responsibility for minimizing warp divergence, (i.e. along logical branches), so as to achieve peak performance. In other respects the terms remain essentially equivalent and with noted exception will be used synonymously.

At high level, CPU/GPU memory is organized hierarchically: *Global* $\rightarrow$ *Device* $\rightarrow$ *Shared*. In this context, Global/Device memory transactions are understood as mediated by high-speed bus transactions, (e.g. PCIe, HyperTransport, QPI), and shared memory accrues in form of a parallel data cache available to all multiprocessor scalar cores. At a more fine-grained level, ancillary memory resources include; per-processor register banks, read-only constant cache shared by all scalar processor cores, and read-only texture cache again shared by all scalar processor cores and by which datapath spatio-temporal coherences may be exploited.

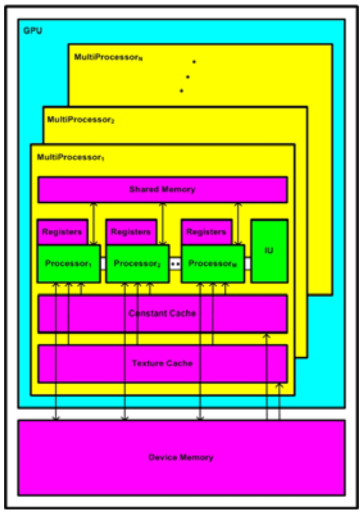

Figure-1: GPU Architecture

The GPU-accelerated PC/Server

Multiple GPU's are appended as ancillary processing resource to a PC/Server hardware platform via high-speed internal bus to form a vector processing array. In Figure-2, a PC-based architectural variant is displayed whereby 4x GPU's are attached to a multicore CPU via a *Northbridge↔Southbridge↔PCIe x16* pathway.

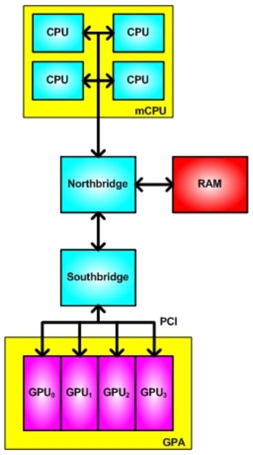

Figure-2: PC-based CPU/GPU System Architecture

Tailored Application Programming Interface (API) components {6}{13}{20} provide concurrent access to multicore CPU and GPU Array (GPA) processing resources. A key subtlety associated with the CPU/GPU processing is GPU resources may be accessed based upon a *non-blocking* transaction model. Thus, CPU processing may continue as soon as a work-unit has been written to the GPU process queue. As result, host (CPU) processing and GPU processing may be overlapped as displayed in figure-3. In principle, GPU work unit assembly/disassembly and I/O at the GPU transaction buffer may to large extent be hidden. In such case, one can expect GPU performance will effectively dominate system performance. Thus, optimal GPU processing gain is realized within an I/O constraint boundary whereby thread processors never stall due to lack of data.

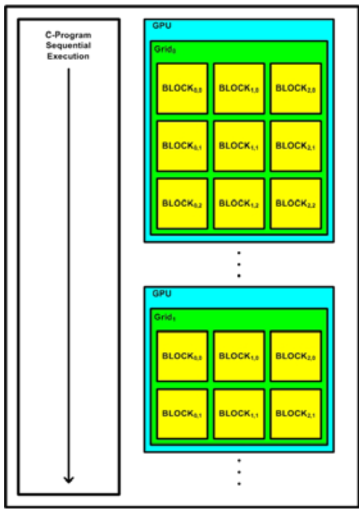

Figure-3: CPU/GPU Process Overlap

In the *ePX* approach, the traditional GPU coprocessor model is abandoned in favor of a *scatter-gather* approach by which full advantage is taken of Node/CPU/GPU process pipelining. In this manner, two fundamental objectives may be achieved; (1) *acceleration of complete applications* and (2) *performance scaling across broad classes of algorithmic kernels*. In effect, a full-featured supercomputing processing model is employed, for which *dynamic resource mapping and scheduling*, *instruction pipeline reuse*, and *CPU/GPU process pipelining* remain key features. While applied here to various forms of the (multicore) CPU/GPA architectural template, these features generally duplicate those found in traditional supercomputing systems.

The GPU-accelerated cluster

The ePX supercomputer processing model finds full expression when applied to a GPU-accelerated cluster. In this instance, a third API component {18} is added as basis for distributed interprocess communication. Depending upon application requirements, Operating System (OS), and specific architectural template, a variety of API's are supported; (1) MPI (cluster) {18}, (2) OpenMP (multicore CPU) {16}, (3) OpenCL (multicore CPU + GPA) {20}, CAL/CTM (GPA) {13}, and CUDA (GPA) {6}. The aforementioned *scatter-gather* service routines then implement API-specific calls for concurrent and transparent access to any given processing resource. The resulting processing model is distinguished by globally optimal map and schedule of algorithmic kernels across all Cluster/CPU/GPA processing resources. In particular, vectorized algorithmic kernels are dynamically assigned to GPU instances based upon; (1) GPU-element availability and (2) opportunistic SIMT instruction pipeline reuse. In this manner SIMT Cyclostatic Thread Residency (CTR) is maximized at any GPU instance[1], maximizing effective process parallelism cluster-wide.

Associated *scatter-gather work-unit* distribution[2] is performed according to scheduler state. A given thread-set may be applied to a GPU instance at initialization or may already exist in situ as result of a previous processing cycle. In the latter case, the scheduler will opportunistically forego pipeline reinitialization, (re: *instruction pipeline reuse*), and apply only datapath during a given *scatter* cycle. In this manner, algorithmic kernels are parallelized at the GPA transaction buffer and thread-sets optimally processed in parallel within GPU/SIMT 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/GPA thread processing may be effectively overlapped[3]. The *ePX Framework* further implements all required *scheduler*, *scatter-gather*, and *CPU/GPA pipelining* management functionality based upon an abstraction by which work-unit structure and interprocess communications implementation details are effectively hidden. In effect, all such details are pushed to process-queue service routines. Thus, *ePX management* operations remain generic across all *multicore-CPU/GPA* and derivative *cluster* architectural templates regardless of the specific nature and location of process components.

A characteristic GPU-accelerated cluster architecture displayed schematically in figure-4. In this context, $NODE_0$ acts as a more or less standard ‘head-end’ resource at which cluster management, ePX Framework, and software development components have been placed. $NODE_1$ through $NODE_N$ represent processing nodes to which GPU arrays have been appended, (i.e. with internals expanded at $NODE_1$). In addition, the traditional Network File Server (NFS) resource is replaced with an ePX/NFS variant intended to provide high-performance multiphysics scatter-gather functionality. Internode communications is based upon a dual-pathway network communications backbone, with node-management and interprocess communications transactions mapped to respective ETHERNET and Infiniband ports.

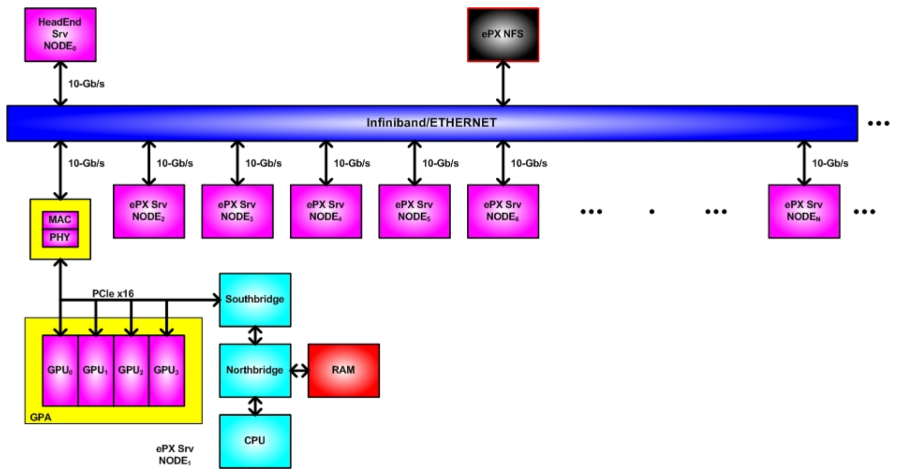

Figure-4: The GPU-accelerated Cluster

Multiphysics simulation may be characterized in terms of a collection of physics processes communicating with one another. For the large-scale problems typical of multiphysics, one would expect presence of a top-level scatter-gather process involving iteration on massive datasets featuring diverse and arbitrarily complex storage patterns. In current GPU-accelerated cluster state-of-the-art, this processing would be performed at the head-end, effectively as a non-parallelizable process component. However, ePX/NFS incorporates two key technology innovations by which critical performance bottlenecks are avoided; (1) use of an ancillary datapath server, and (2) GPU-accelerated (multiphysics) scatter-gather. In this context, ePX/NFS implements a demand-driven protocol by which data is moved only to that node actually processing a work-unit; any associated work-unit assembly/disassembly, (i.e. within context of hierarchical scatter-gather), is then performed based upon abstract datapath references. Further, localized GPU resources are employed to accelerate required transformations on component physics processes prior to (RAID) write-back. In this manner, any required head-end intervention is effectively eliminated, scatter-gather work-unit composition is greatly simplified, and internode communication overhead is significantly reduced.

At any *ePX* node, distinct *scatter-gather* process queues are maintained for each mapped GPA, CPU, and NODE processing resource {3}. Service methods attached to these queues implement *work-unit* transactions at associated buffers based upon nominal scatter-gather pathways displayed in *figure-5*. In this specific case, component-tasks originate at $NODE_0$ (head-end) and are propagated to all other nodes[4]. As shown in expanded view at cluster $NODE_1$, tasks are resolved into component algorithmic kernels and distributed to local CPU/GPA resources. Global multiphysics solution assembly is performed at ePX/NFS.

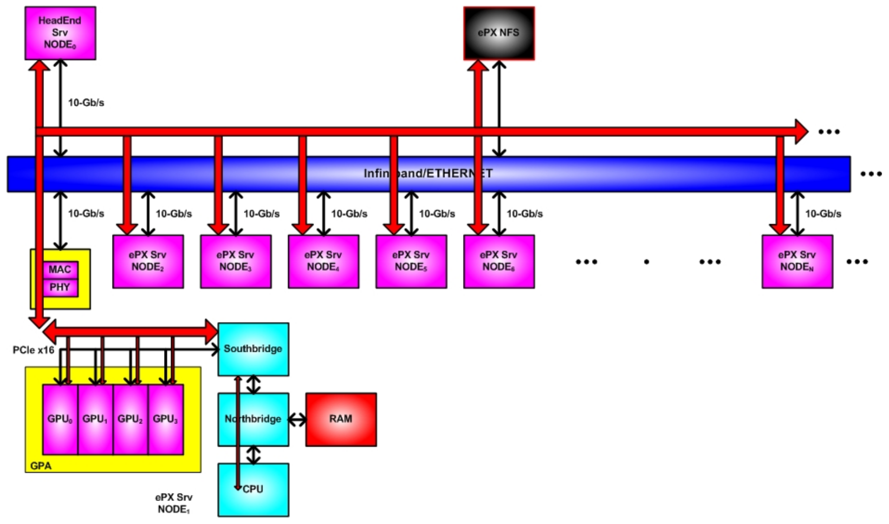

Figure-5: GPU-accelerated Cluster Scatter-Gather Patterns

GPGPU Multiphysics Processing Model

The GPU-accelerated cluster when combined with ePX scatter-gather reveals unique advantages pertinent to GPGPU multiphysics supercomputing. In particular, a nominal two-order-of-magnitude increase in processing threads may be efficiently accessed at each processing node. Further, integration of distributed (cluster), SMP (multicore CPU), and SIMT (GPU/GPA) processing models is well matched to the characteristic granularity of multiphysics processes when resolved along physics module, subprocess, and algorithmic kernel boundaries. To see this, we examine a simple dataflow example typical of multiphysics simulation.

In figure-6, four distinct physics processes are displayed, color-coded in *red*, *teal-yellow-green*, and *purple-blue-black*. Dotted lines indicate simple graph decomposition in form of recursive partitions. This decomposition is calculated based upon map/schedule optimization and serves to impose a *process-subprocess-algorithmic kernel* scatter-gather hierarchy matched to characteristic coarse-grained/medium-grained/fine-grained parallelism typified by cluster/multi-core CPU/GPU processing resources. In this manner, the ePX processing hierarchy is defined. Here, the *red* process is characterized by four subprocesses consisting of a single algorithmic kernel and for which there is no communication among subprocesses. The *teal-yellow-green* process consists of three sub-processes consisting of three distinct algorithmic kernels, with a single kernel-type reserved to each branch. In this case however, there exists communication among subprocesses, as indicated by internal datapath scatter-gather junctures. Finally, each of two *purple-blue-black* processes is characterized by two distinct subprocesses, each of which is composed of three distinct algorithmic kernels. There is also no communication among subprocesses.

The simplicity of this dataflow admits naïve map and schedule process optimization. Given fully resolved scatter-gather points at the boundaries of each physics module and availability of four GPU instances at each cluster node, modules are first distributed to distinct cluster nodes and all component algorithmic kernels processed on local CPU/GPU resources where additional optimizations become available; (1) as algorithmic kernels recur along subprocesses, instruction pipelines may be reused, and (2) as algorithmic kernels persist along subprocesses, intermediate results need not be updated in (CPU) global memory, (i.e. intermediate results are left in GPU device memory until subprocess terminates). Under circumstances where subprocesses communicate, intermediate datapath must be updated in global memory. However, explicit CPU/GPU process pipelining based upon asynchronous datapath transfers may be employed to hide much of the associated overhead.

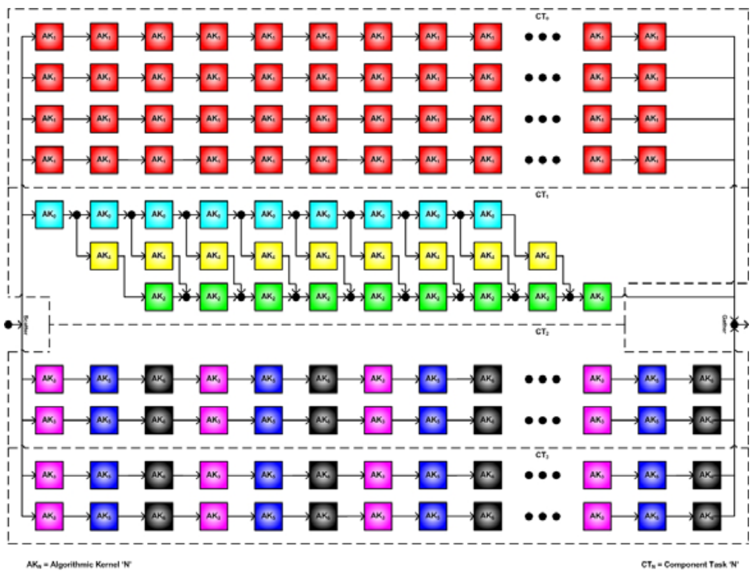

Figure-6: Example Multiphysics Dataflow Component

A simplified version of the optimized process schedule is displayed schematically in figure-7; processing resources are organized vertically and time horizontally. The process schedule is organized into three distinct (vertical) regions, each of which also corresponds to a distinct component of hierarchical scatter-gather. The upper-half indicates launch of four tasks at distinct cluster nodes (‘CT_K’) based upon a distributed processing model. Each of these processes is padded on the left with an interval encapsulating all transaction and process launch overhead components (‘OV’). We note an asynchronous-sequential scatter-gather cycle is implied at the cluster head-end CPU ($NODE_0$). The lower-half expands local process scheduling at cluster $NODE_1/CT_1$. The lower-most resource schedule refers to local CPU/SMP processing, with four GPU/SIMT resources scheduled immediately above (‘GPU_{0-3}’). For simplicity, we assume a single CPU core with multithreading. We note sequential launch of four CPU threads, each of which is associated with a single GPU resource. In this case, overlap of CPU and GPU processing implies asynchronous I/O transactions at each GPU process queue and derived parallelism across the entire Distributed/SMP/SIMT hierarchy. Specific CPU processes include; *get_work_unit(..)*, *get_gpu_buffer(..)*, *work_unit_assembly(..)*, *gpu_scatter(..)*, *gpu_gather(..)*, and *solution_assembly(..)*.

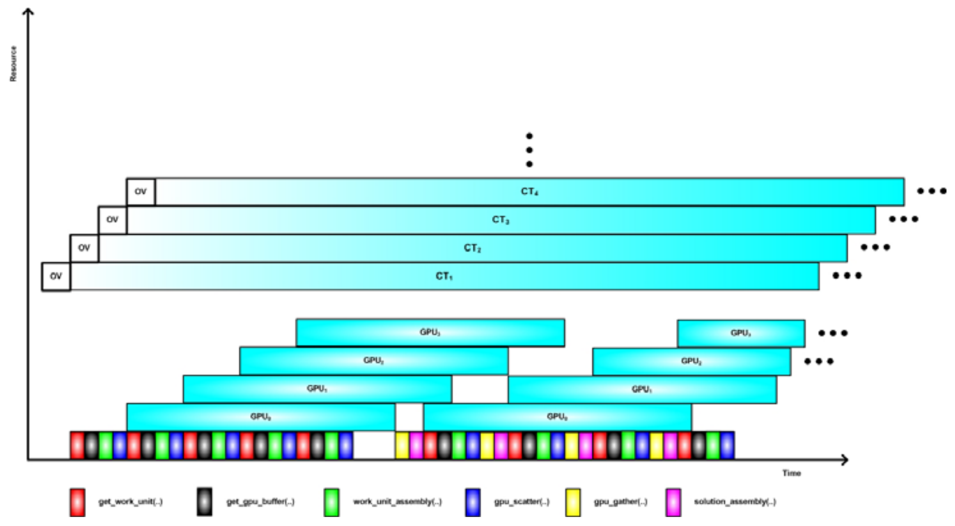

Figure-7: Optimized Process Schedule (local)

As previously mentioned, realistic multiphysics dataflow will typically incorporate internal communications among physics modules, (re: *dynamic boundary conditions* and *mesh rediscretization*). Thus, one would expect interprocess communication among cluster nodes at module boundaries. In figure-8, we consider a simple two-stage iterated relaxation among all four physics modules whereby intermediate solutions are exchanged based upon a top-level global iteration.

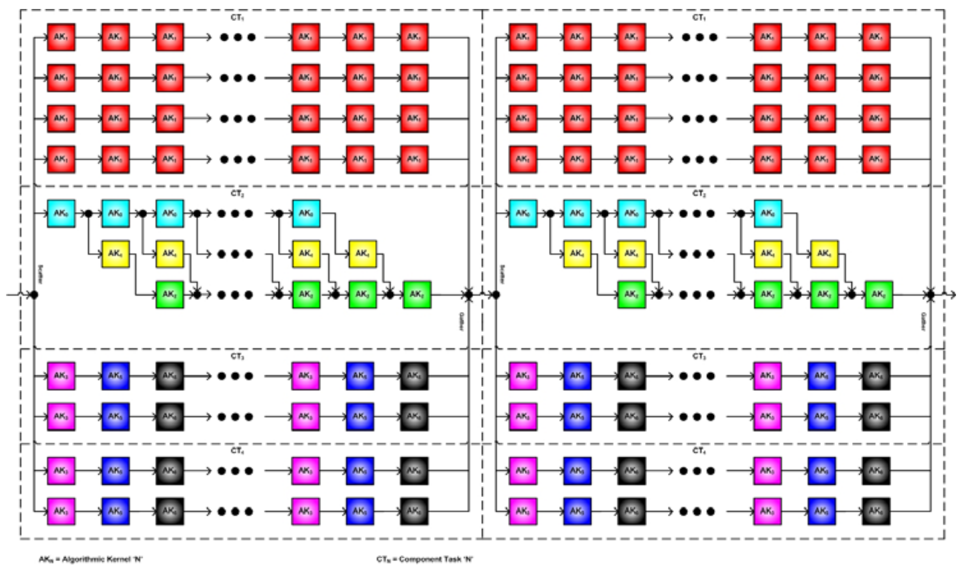

Figure-8: Example Multiphysics Relaxation Dataflow

An essential point is the corresponding process schedule is generated based upon the complete dataflow and for which map/schedule optimization is fully elaborated in terms of; (1) an arbitrarily diverse set of algorithmic kernels, (2) the complete set of available processing resources, and (3) the entire process timeline. Thus, the globally optimal nature of the ePX supercomputer processing model is expected to exhibit a generally superior effective parallelism when applied to multiphysics applications, as compared to that typical of a standard coprocessor model.

The GPU-accelerated MultiCluster

As displayed in figure-6, the GPU-accelerated cluster is easily integrated with existing cluster and enterprise network infrastructure. In particular, combination of Desktop SuperComputer (DSC) workstations with GPU-accelerated cluster and enterprise HPC cluster resources creates a third option to the traditional ‘client/server’ versus ‘distributed’ discussion in that local processing capability may be tailored according to needs of a (scientific) software ‘developer’ versus that of a ‘user’. More specifically, GPU-accelerated DSC workstations provide algorithm designers and library developers with a *low capital investment*, *low life cycle cost*, *high-availability* HPC resource well suited to *rapid prototyping*, *benchmarking* and *verification* of library components essentially independent of software updates at a cluster head-end. When incorporated within a suitable software engineering methodology, this capability can lead to significantly reduced development costs, based upon fast turnaround, extended test protocols, and complete portability between cluster and DSC build environments. Yet another advantage is reservation of incremental development work to DSC’s will significantly improve overall cluster availability and efficiency.

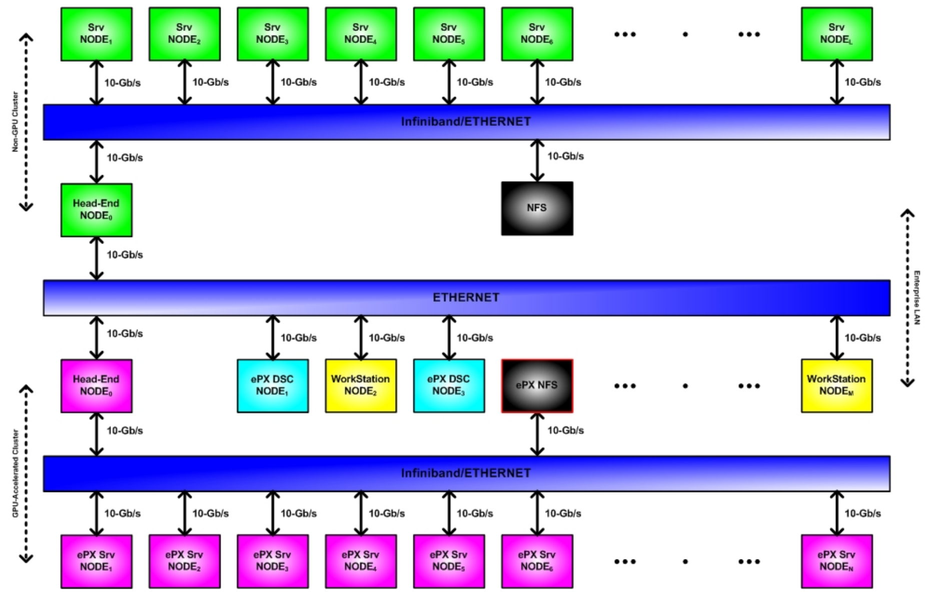

Figure-6: The GPU-accelerated Multi-Cluster

Multiphysics Software Architecture

The ePX software architecture is expressed in form of a generic and reusable framework for multiphysics applications. 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). As shown in figure-9, ePX framework features *scheduler*, *dispatcher*, and *scatter-gather engine* components communicating with *generalized process queues* associated with each level of the Distributed/SMP/SIMT hierarchy. In this manner, cluster, multicore CPU, and GPA resources may be efficiently accessed at any available 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 multiphysics software architecture remains more or less uniform across all applications. In nominal configuration, OpenMPI, OpenMP, and CUDA/OpenCL API’s are employed. However, alternative API combinations may be supported with little architectural impact, (e.g. AMD CAL/CTM {13}, PVM {21}, and alternate MPI flavors {14}{18}).

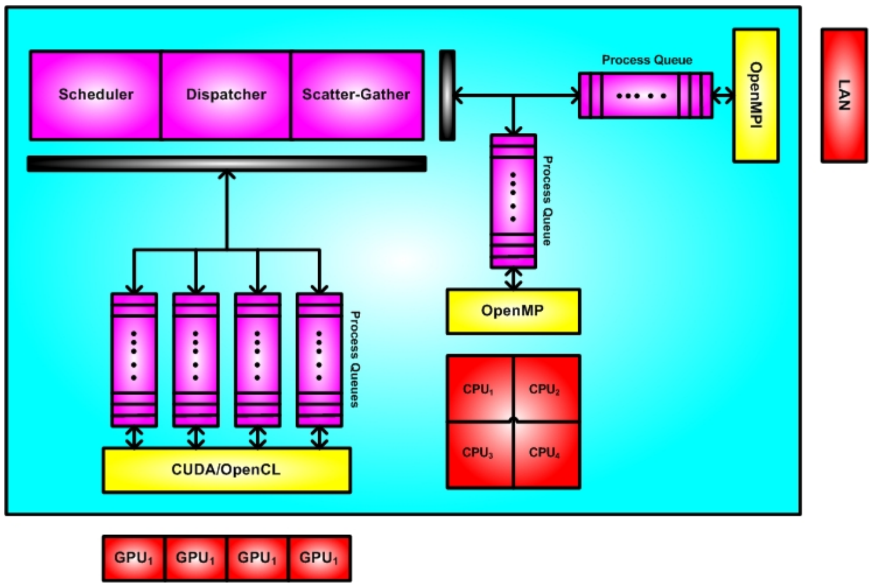

Figure-9: ePX Software Architecture

Multiphysics applications development is further simplified with use of ePX C/C^{++}/FORTRAN acceleration libraries. These libraries feature standard compilation bindings and may higher performance based upon functional aggregation and improved local optimizations. 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.

Amdahl’s Law

As previously reported, ePX technology is directed toward acceleration of complete applications {1}{2}. The basis of this claim is substantiated based upon a simple interpretation of the ePX processing model, based upon Amdahl’s Law {22} for which the basic mathematical model is:

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

Here, ‘A’ is the acceleration factor, ‘P’ the parallelizable code fraction, and ‘N’ the order of parallelization. We first consider hyperthreaded parallelization at a single cluster node. Thus ‘1’ becomes:

$$A = \frac{1}{(1 - P_{CPU} - P_{GPU}) + \frac{P_{CPU}}{N_{CPU}} + \frac{P_{GPU}}{N_{GPU} N_{TP/GPU}}} \tag{2}$$

Here, $N_{CPU} \equiv$ multicore order, $N_{GPU} \equiv$ GPU array order, and $N_{TP/GPU} \equiv$ Number thread processors per GPU. With use of asynchronous GPU transactions, we assume CPU-GPU pipelining at efficiency C_{CPU} , (constant $\in$ [0,1]). Thus, CPU and GPU processes are mutually parallelizable. Consequently, ‘2’ becomes:

$$A = \frac{1}{(1 - C_{CPU} P_{CPU} - P_{GPU}) + \frac{C_{CPU} P_{CPU} + P_{GPU}}{N_{CPU} N_{GPU} N_{TP/GPU}}} \tag{3}$$

For simplicity, we will assume identical processes at ‘N_{NODE}’ cluster nodes, mutually parallelizable with efficiency ‘C_{NODE}’. The resulting total acceleration is then:

$$A = \frac{1}{(1 - C_{NODE}(C_{CPU} P_{CPU} + P_{GPU})) + \frac{C_{NODE}(C_{CPU} P_{CPU} + P_{GPU})}{N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU}}} \tag{4a}$$

Simplifying, we obtain:

$$A = \frac{1}{(1-P') + \frac{P'}{N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU}}} \tag{4b}$$

Of particular interest is the limiting case:

$$\lim_{N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU} \to \infty} \left(\frac{1}{(1-P') + \frac{P'}{N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU}}} \right) = \frac{1}{1-P'} \tag{5}$$

Thus, with the term ‘ $C_{NODE}(C_{CPU} P_{CPU} - P_{GPU})$ ‘sufficiently large:

$$1-P' \ll 1 \rightarrow A = \frac{1}{1-P'} \gg 1 \tag{6}$$

In this limiting case, ‘A’ is ‘large’ only if the multiphysics application is parallelizable to a sufficiently high degree. Similarly, approximate linear scaling should be observed over some range of composite thread order:

$$\frac{P'}{N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU}} \gg (1-P') \rightarrow A \cong N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU} \tag{7}$$

Acceleration factors have been experimentally confirmed for a number of characteristic simulations[5]:

Algorithmic Kernel	**Acceleration**
Finite Difference: Heat Equation, SOR (Gauss-Seidel solver)	x310
FEM multi-grid: Mixed Precision Linear Solvers	x505
Image Processing: Optical Flow	x1030
CFD: 3D Euler solver	x540
CFD: Navier-Stokes (Lattice Boltzmann)	x1875
Signal Processing: Sparse Signal Recovery from Random Projections (NP-hard combinatorial optimization)	x580
Computational Finance: Quantitative Risk Analysis and Algorithmic Trading	x935
Computational Finance: Monte Carlo Pricing	x1500

Table-1: Acceleration Performance Test Kernels

Scaling is verified based upon measured acceleration as function of problem scale and composite thread order, post map/schedule optimization. The fundamental experimental methodology is characterized by the following steps:

(1) Select problem size (assume constant dataflow),
(2) Perform map/schedule optimization (assume unconstrained resource pool),
(3) Build/Run test-case,
(4) Measure acceleration factor,
(5) Plot against composite thread order,

It should be noted interpretation of any results thus generated is greatly complicated by an essential nonlinearity between assumed problem order and composite thread order as result of map/schedule optimization. Nevertheless, an approximate '$N_{NODE} N_{CPU} N_{GPU} N_{TP/GPU}$' scaling has been verified for each application cited in table-1 with test-points at N_{NODE} values {2,4,8}, (4xC1060/NODE). According to equations '4a,b' this scaling is expected to first become sublinear as problem size is further increased and then converge to a maximum.

Summary

Arguably, GPU-accelerated clusters represent a very promising 'enabling technology' for large-scale multiphysics simulation; massively parallel GPU arrays appended to multi-CPU, multicore servers at cluster nodes provide incredible aggregate peak-performance capability, and at unprecedented price-performance ratios. However, effectively harnessing this capability has proven difficult. Part of the reason for this is the technology is very new and also rapidly evolving. Yet another aspect is multiphysics is characterized by complex mathematical formulation involving distinct physics processes and must therefore be considered fundamentally difficult where; (1) diverse algorithmic content, (2) dynamically varying boundary conditions, (3) adaptive discretization, and (4) management of large-scale and arbitrarily complex datasets are implied. Further, where the GPU-accelerated cluster is considered, maximal parallelization demands efficient integration of three distinct processing models (Distributed/SMP/SIMT) for which CPU/GPU code components are inherently multithreaded. The upshot is any realization of performance potential comes at cost of significantly increased complexity in terms of processing model, software architecture, system infrastructure, and programming.

In this paper, the ePX accelerated multiphysics simulation solution is presented in form of two categories of technical innovation; (1) software architecture, and (2) system infrastructure. A fundamental technical goal of efficient utilization of all available processing resources is adopted. Ancillary goals of reusable software architecture, compatibility with standard OS and software development platforms, and global process optimization are also assumed. Traditional supercomputing is based upon use of parallelizing compilers and specialized runtime support for concurrent thread scheduling, SMP memory management, etc. From the perspective of software architecture, *ePX* reverses this principle with explicit addition of *map*, *scheduler*, and *generalized process queue* infrastructure directly to the application in form of a generic framework. This framework is then leveraged as basis for implementation of a scatter-gather (supercomputer) processing model; multiphysics dataflow is hierarchically parsed and corresponding process components applied to cluster/CPU/GPU processing resources based upon a custom map/schedule optimization across an entire execution timeline. The scatter-gather engine then employs methods attached to each process queue to asynchronously launch processes/threads at associated API's, and subsequently synchronize according to reductions on the dataflow graph. In this manner, ePX maximizes effective parallelization for complete applications of virtually arbitrary scale and complexity. Traditional cluster technology practice maps top-level multiphysics scatter-gather to non-parallelizable processes residing at head-end and NFS nodes. From a system infrastructure perspective, this approach creates fundamental performance-bottleneck issues associated with cluster-wide datapath movement. ePX addresses this problem with new NFS technology that; (1) abstracts datapath references within context of work-unit scatter-gather, and (2) implements multiphysics scatter-gather based upon GPU-accelerated post-processing prior to write-back.

Where use of GPU-accelerated cluster technology is considered, *enParallel, Inc. ePX technology* is seen to address three fundamental aspects of the multiphysics simulation performance equation; (1) integration of Distributed, multicore SMP, and GPU/SIMT processing models, (2) optimization of effective parallelization, and (3) minimization of scatter-gather datapath movement and ancillary head-end processing overhead.

Note[1] - In present context, CTR is defined as a measure on the expected proportion of time during which the instruction pipeline is performing actual datapath calculations, (e.g. as opposed to device I/O, instruction pipeline initialization, and thread synchronization).
Note[2] – A 'work-unit' consists of instructions + datapath sufficient to a map/schedule instance on any Node/CPU/GPU processing resource.
Note[3] – 'gather' remains blocking according to the associated dataflow representation and implied scheduler synchronization semantics
Note[4] - Any of MPI, PVM, or MOSIX distributed processing models may be employed for this purpose.
Note[5] – Test platform is single multithreaded DSC using 4xC1060 GPU's (4-TFLOPS aggregate).

Bibliography

{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} ***"Hybrid (OpenMP and MPI) Parallelization of MFIX: A multiphase CFD Code for Modeling Fluidized Beds"*** S. Pannala, E. D'Azevedo, M. Syamlal SAC 2003
{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 Compute PTX: Parallel Thread Execution"***; ISA Version 1.2, 2008-04-16, SP-03483-001_v1.2
{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} ***"ATI CTM Guide – Technical Reference Manual"***; V1.01 2006AMD
{13} ***"ATI Stream Computing – Technical Overview"***; V1.01 2009AMD

{14} ***"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
{15} ***"MPI Parallelization Problems and Solutions"*** UCRL-WEB-200945 https://computing.llnl.gov

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

{17} ***"MPI: A Message Passing Interface Standard Version 1.3"***; *Message Passing Interface Forum*, May 30, 2008

{18} ***"MPI: A Message Passing Interface Standard Version 2.1"***; *Message Passing Interface Forum*, June 23, 2008

{19} ***"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*

{20} ***"The OpenCL Specification"***; *Khronos OpenCL Working Group*, A. Munshi Ed. Version 1.0, Document Revision 29

{21} ***"PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing"***; A. Geist, et al. MIT Press 1994

{22} ***"Principles of Parallel Programming"***; C. Lin, L. Snyder 1st Ed. Addison-Wesley 2008