'GPU-based Desktop Supercomputing'

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

Introduction

It is interesting to note a major force impacting **G**raphics **P**rocessor **U**nit (GPU) evolution has been a seemingly endless demand for increased **P**ersonal **C**omputer (PC) graphics subsystem performance. In this context, the GPU is understood to exist as an ancillary coprocessor subsystem, attached to some internal high-speed bus and memory-mapped into global system memory resources. In particular, gaming, computer vision, and advanced graphics design applications have driven sharp MIPS performance improvements and increased diversity and algorithmic sophistication on part of relevant graphics standards {1}{2}{15}{16}. Arguably, this is all part and parcel of a larger evolutionary trend whereby PCs supplant dedicated workstations for a host of compute intensive applications.

At a deeper level GPU evolution has hinged upon assumption of a processing model appropriate to achieving highest possible performance for broad classes of graphics algorithms. This in turn drives all relevant aspects of hardware architecture design, (e.g. memory/cache architecture, instruction pipeline, thread scheduler, math unit/floating point processor). The overall most efficient GPU processing model determined thus far is **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD). The SIMD model has been leveraged to great advantage in traditional vector processor/supercomputer designs, (e.g. Cray X-MP, CDC Star-100, Convex C1), by virtue of a capability to accelerate datapath computation based upon concurrent execution of processing threads, (i.e. see discussion of Amdahl's Law below). The SIMD concept has also been usefully employed in recent CPU architectural advancements arguably more relevant to PC, such as the IBM Cell processor, x86 with MMX extensions, SPARC VIS, Sun MAJC, ARM NEON, et al. At some point it was noticed the SIMD processing model already adopted for GPU had potential use for general classes of scientific computation not specifically associated with graphics applications. This was the start of the **G**eneral **P**urpose computing on **GPU** (GPGPU) movement {13} and basis for many examples of GPU accelerated scientific processing {3}{4}{5}{11}{12}.

GPGPU depends intimately upon **A**pplication **P**rogramming **I**nterface (API) access to GPU processing resources; the GPU API hides much of the complexity associated with manipulation of hardware resources and provides convenient access to I/O, memory management, and thread management functionality in form generic programming function calls, (e.g. C, C^{++}, Java, Python). In this manner, GPU hardware is effectively virtualized as a standard programming resource, greatly facilitating ubiquitous application development incorporating GPU acceleration. Existing APIs include NVIDIA's **C**ompute **U**nified **D**evice **A**rchitecture (CUDA) {6}{7}{9} and ATI's **D**ata **P**arallel **V**irtual **M**achine (DPVM) {14}.

GPU Architecture

As displayed in figure-1, the SIMD GPU is nominally 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, SIMD instruction fetch and execution, and cache operations are maximally synchronized. Memory is organized hierarchically *Global* $\rightarrow$ *Device* $\rightarrow$ *Shared* where Global/Device memory transactions are understood as mediated by high-speed bus transactions, (e.g. PCIe, HyperTransport).

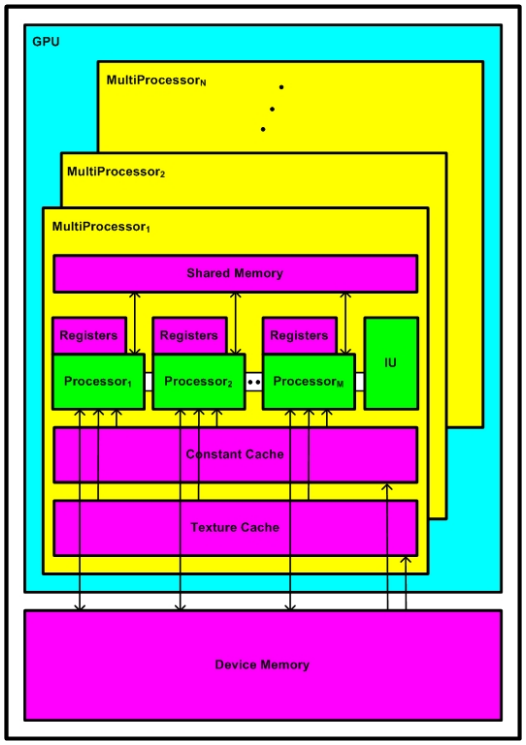

Figure-1

A key subtlety associated with the CPU/GPU processing architecture is GPU processing is effectively *non-blocking*. Thus, CPU processing may continue as soon as a work-unit has been written to the GPU transaction buffer[1]. Consequently, host (CPU) processing and GPU processing may be overlapped as displayed in figure-2. In principle, GPU work unit assembly/disassembly and I/O at the GPU transaction buffer may to large extent be hidden. In such case, GPU performance will effectively dominate system performance. As might be expected, optimal GPU processing gain is achieved at an I/O constraint boundary whereby thread processors never stall due to lack of data.

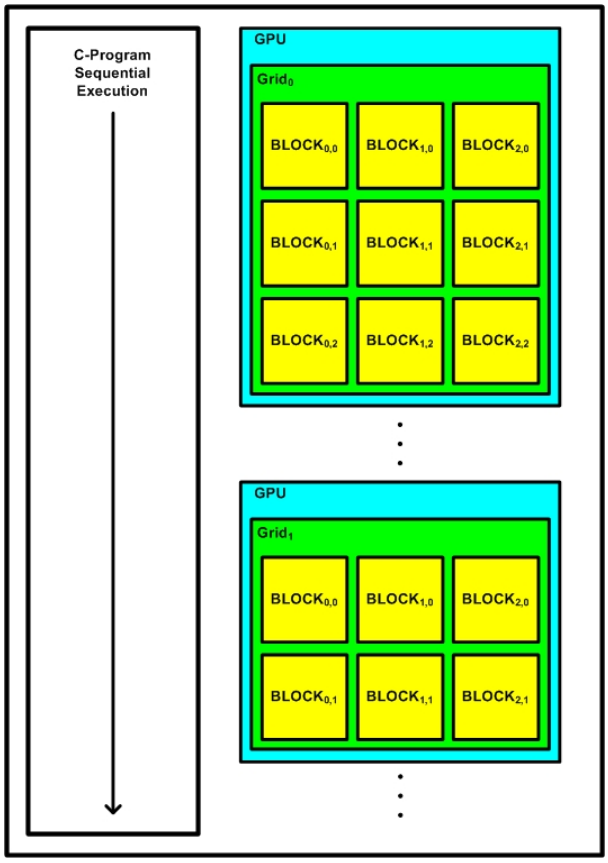

Figure-2

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:

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

A key consideration is the limiting case:

$$\lim_{N \to \infty} \left(\frac{1}{(1 - P) + \frac{P}{N}} \right) = \frac{1}{1 - P} \qquad (2)$$

This indicates a theoretical maximum acceleration for the complete application. However, CPU code pipelining, (i.e. overlap with GPU processing), must also be factored into any calculation for 'P'; pipelining effectively parallelizes CPU and GPU

code segments reducing the non-parallelized code fraction $'(1-P)'$. Thus, under circumstances where decrease is sufficient to claim $\frac{P}{N} >> (1-P)$, Amdahl's Law then becomes:

$$A \cong \frac{1}{\frac{P}{N}} = \frac{N}{P} \cong N \qquad (3)$$

The essential point is well motivated software architecture design can take advantage of this effect, greatly improving acceleration potential for the complete application.

Desktop Supercomputing

"CPU + GPU(s) does not a supercomputer make!" The point here is emphasis of a useful distinction between 'CPU + ancillary (GPU) coprocessor' and true supercomputer architectural models; supercomputers makes explicit use of the *'scatter-gather'* programming and execution models. Implicit to this is leveraging of at least two levels of parallelism; coarse-grained (kernel) and fine-grained (thread). In this context, 'scatter-gather' is understood as; (1) composition of multiple diverse algorithmic kernels, (2) active generation of concurrent processing schedules for algorithmic kernels mapped to specific processor (GPU) resources, and (3) management of all I/O work unit assembly/disassembly functions associated with any such processor resource. The key difference is the capability to map diverse algorithmic kernels at a coarse-grained level admits a degree of programming flexibility and enhanced performance not generally possible for the CPU + GPU coprocessor. The fundamental performance advantage of SIMD depends critically upon cyclostatic residency in the instruction pipeline. Thus, SIMD processing generally achieves highest performance where all synchronized thread processes are effectively *'doing-the-same-thing-for-as-long-as-possible'*, (e.g. dot-product calculation in a 'matrix-multiply' operation). The point here is any 'mixing' of algorithmic kernels may lead to sharp decrease in performance due to instruction unit (pipeline) thrashing. On the other hand, true scatter-gather on a multi-GPU allows mapping of a specific algorithmic kernel to a distinct GPU resource, in principle allowing associated instruction pipelines to remain resident for longer intervals. Further, opportunistic reuse of instruction pipelines already in-place may be employed. In this manner, highest performance is achieved for applications composed of diverse algorithmic kernels. Extending this notion, an architecture consisting of CPU + multiple *ganged* GPUs, (e.g. NVIDIA 2,3,4-way SLI), would not usefully be considered a supercomputer, but a CPU + (a very good) graphics coprocessor.

Implications of GPU-based supercomputing include; (1) supercomputer processing model implementation at the software application level, (2) *scatter-gather* on multiple independent GPU resources, (3) use of CPU/GPU process scheduling/pipelining, and (4) realization of a significant performance advantage relative to naïve SIMD processing on diverse algorithmic kernels. The essential point is, to extent the SIMD vector processing model is appropriate to algorithmic kernels commonly used in scientific software applications, GPU-based supercomputing must then be considered a viable **H**igh-

Performance **C**omputing (HPC) solution option and fully consistent with GPGPU objectives {13}.

At present time GPU components are commonly available as PCIe form-factor graphics cards available at virtually every commercial outlet servicing the PC components market. As ubiquitous commodity components, prices remain astoundingly low particularly when one considers the incredible processing power made available by these devices. One can easily construct a four GPU supercomputing platform[3] for approximately $2K-$3K; overall, the hardware configuration is essentially that of an advanced gaming system. An essential point is GPU technology has effectively created a new HPC niche typified by processing power hitherto restricted to distributed grid clusters, (e.g. Beowolf, Sun Ultra-SPARC, IBM Linux Cluster), and involving orders of magnitude smaller capital investment[4]. The resulting implications are both substantive and diverse. However, we will focus upon only one; *the PC has been rendered a viable low-to-mid-tier scientific supercomputing platform*. This in turn implies existing and generic PC-based scientific software may with relatively little effort be scaled to performance levels commonly associated with custom software on distributed grid clusters.

As a particularly significant example, we consider the **MAT**rix **LAB**oratory (MATLAB) software system {17}; with a user-base exceeding 1-million installed seats worldwide MATLAB represents a de facto standard algorithm development application for diverse engineering and scientific disciplines. As one might expect from the name, MATLAB employs a fundamental matrix representation for all datapath and associated operators. Thus, one would expect algorithms developed in MATLAB generally amenable to accelerated vector processing. However, in standard form, MATLAB accrues as a PC-based application for which no such optimizations are available. Thus, where applications development methodology is being considered for high-performance scientific computing, specific algorithmic kernels are first developed and tested in MATLAB and then recoded in some other language and implemented as a custom application on a separate HPC resource, (e.g. grid cluster or stand-alone mainframe supercomputer)[5]. However, where the desktop supercomputing model is applicable, this methodology can be greatly simplified. Algorithmic kernels are initially coded in C/C^{++} as SIMD processing blocks and subsequently integrated under a top-level function responsible for GPU array scatter-gather, I/O, global memory management, etc. This top-level function is then used as basis for creation of a **M**atlab **EX**ternal (MEX) DLL directly accessible from the MATLAB run-time environment. In this manner, accelerated kernel processing on the GPU array is completely virtualized via the associated API in form of standard MATLAB function calls; datasets may then be created and applied, results rendered, and top-level processing control applied, all from the standard MATLAB command line or an associated ‘M’-file, (MATLAB program). In essence, GPU-accelerated MATLAB is rendered dual-purpose as algorithm kernel development environment and software application delivery platform. Further, any such application may be reused component-wise in form of derivative function libraries and tool-boxes. For complex projects, potential savings in terms of schedule time, capital investment, and manpower could prove compelling. Similar reasoning may be applied where other PC-based algorithm

development/math-modeling applications are considered, (e.g. COMSOL, SOFA, SciLab, Octave).

Performance Benchmarks

The essential distinction of the aforementioned GPU-based supercomputing model is characteristic acceleration values approaching a limit $A \cong N_{THREAD}N_{GPU}$, ($N_{THREAD} \equiv$ (Effective) Number of Thread-Processors per GPU and $N_{GPU} \equiv$ Number of GPUs in the processing array). Thus, kernel acceleration on GPA arrays may be estimated from single-GPU test applications on specific algorithmic kernels, with multiplication by an assumed array order[6]. In this case, the experimental platform is nominally assumed to consist of a 2.3GHz dual-core (Intel) processor featuring 2GB RAM, WinXP OS, and a GPU array consisting of 4xNVIDIA GeForce 8800GTX graphics cards, (i.e. 128 thread processors per GPU). Single-GPU acceleration numbers include enParallel, Inc. test application benchmarks and survey of published results {18}.

Algorithmic Kernel	Acceleration
Computational Chemistry: 2nd order Moeller-Plesset	x17
Life Sciences: Smith-Waterman	x120
Finite Difference: Heat Equation, SOR (Gauss-Seidel solver)	x68
FEM multi-grid: Mixed Precision Linear Solvers	x108
Image Processing: Optical Flow	x220
Image Processing: Cubic B-spline interpolation on 3D textures	x1308
CFD: 3D Euler solver	x116
CFD: Navier-Stokes (Lattice Boltzmann)	x400
Signal Processing: Sparse Signal Recovery from Random Projections (NP-hard combinatorial optimization)	x124
Computational Finance: Quantitative Risk Analysis and Algorithmic Trading	x200
Computational Finance: Monte Carlo Pricing	x320
EDA: static timing analysis	x1040
EDA: SPICE simulator	x32

References

{1} ***"OpenGL Reference Manual"***, D. Shreiner 4th Edition Addison-Wesley

{2} ***"OpenGL Programming Guide"***, D. Shreiner, et al. 5th Edition Addison-Wesley

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

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

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

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

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

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

{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} ***"Accelerating large Graph Algorithms on the GPU Using CUDA"***; P. Harish, P.J. Narayanan, Center for Visual Information Technology, Hyderabad, India

{12} ***"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

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

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

{15} ***"Programming Guide for Direct3D 10"***; Microsoft Corporation http://msdn.microsoft.com/en-us/library/bb205067(VS.85).aspx

{16} ***" Reference for Direct3D 10"***; Microsoft Corporation http://msdn.microsoft.com/en-us/library/bb205147(VS.85).aspx

{17} http://www.mathworks.com

{18} http://www.nvidia.com/object/cuda_home.html#

Note[1] - Algorithmic components offloaded to GPU are organized as a 'grid' of concurrently executed 'blocks'; the grid temporally orders warp ensembles that may be executed concurrently and spatially orders concurrently ordered warps on multiprocessor elements, (i.e. 'thread' processors).

Note[2] 'A' = Acceleration, 'P' = Proportion of code accelerated by parallelization, 'N' = Number of processors applied to 'P'. Parallelization overhead, (e.g. scatter-gather, GPU I/O), is assumed a component of code not parallelizable, (i.e. the quantity '1-P').

Note[3] – PC motherboards supporting an array consisting of 8xGPUs are currently available.

Note[4] - GPU-based desktop supercomputing systems may also be 'clusterized' {3}.

Note[5] - For problems sufficiently parallel, applications can be resolved into distinct algorithmic kernels mapped to a cluster resource. MPI-based functions and global array semantics, (i.e. distributed arrays and parallel for-loops), are available for interdependent kernels.

Note[6] - Given the generally superior kernel scheduling anticipated of the GPU scatter-gather (supercomputer) processing model, estimates thus derived may prove conservative.