AMD WHITE PAPER

AMD
Smarter Choice

The Need for Acceleration Technologies to Achieve Cost-effective Supercomputing Performance for Advanced Applications

Written by: Joe Landman

OVERVIEW

Supercomputing has become an essential tool for many users, including those working in the scientific, engineering, biomedical, and financial disciplines. Although processing capabilities continue to advance, in many cases demand is increasing at an even faster rate. Simulation and analysis problems — once intractable due to computational limitations — have become possible.

Supercomputing technology has evolved from highly specialized and custom circuitry to commodity circuits based largely upon x86 instruction-compatible processing hardware. Leveraging economies of scale lowers costs of production, amortizes research and development, and simplifies software development, as more programmers are familiar with the technologies. Indeed, many supercomputers listed on the TOP500 supercomputers site (http://www.top500.org) are aggregations of commodity processing units and parts. Programs may be easily ported from desktop Linux® machines — which are popular platforms in development and scientific and engineering — to Linux-based supercomputing systems.

Aggregating large numbers of processors into huge clusters and cluster-like machines, while useful, has a variety of limitations. Overcoming these limitations in order to extract the maximum performance from these systems requires extensive application of acceleration technologies. In order to understand the limitations and how acceleration technology addresses them, this paper will examine why clusters are built, and a brief history of supercomputing technology will provide context for the need for acceleration technologies.

HISTORICAL CONTEXT: BUMPING INTO THE PERFORMANCE CEILING

For some time, particular types of calculations have dominated scientific studies. They include the performance of repetitive tasks, usually with a strong emphasis on double-precision floating-point data types, while operating on increasingly larger problem sets.

Savvy users worked with computer vendors to extend the classical von Neumann architecture to include support for faster processing of these specific calculations. Vector and parallel processing originated from these designs, and later on, more generalized distributed processing. Vector processing enables multiple operations to occur simultaneously per clock cycle. It effectively provides a clock-cycle multiplicative effect, performing much more work per clock cycle than non-vector operations.

Vector processors worked on architectures that enabled high sustained-memory bandwidth and a reasonable ratio of floating point operations to memory fetch-and-store operations. Writing the most time-consuming algorithms in a vector manner achieved exceptional performance on these early machines. Typically, one spoke in terms of average vector length and the speedup they achieved over “ordinary” processing. For example, a vector length of 64 would usually result in a multiplicative factor of about 64x over non-vector code in various cases. Programming was not that difficult, as existing tools were quite good for this. The downside to such architectures was the cost associated with their acquisition and ongoing operation and maintenance; they were not easy to build, and the resulting machines cost millions of dollars for small configurations.

Parallel processing on "supermicros" in the 1990s was based on a number of individual processors that shared access to memory. These processors were not as robust as vector processors on vectorized algorithms, but they provided much faster processing than the non-vector portion of the application codes. While the code may not have been vectorized, the programmers used shared memory parallel (SMP) techniques to accelerate as much of the program as possible. For a 32-CPU system, speedup of nearly 32x was possible for well-constructed programs. Superlinear speedup, where a program achieves speeds more than N-times faster on N processors, was also observed. This typically occurs when the program working set (the memory used) per processor fits within the processor cache. Vendors noted this, and increased their processor caches to 8 MB and more in the late 1990s.

As machines grew larger, the inefficiencies inherent in this approach thwarted efforts to run huge SMP programs. Distributed memory programming (DMP) — as seen in MPI, PVM, and other implementations — gained favor, laying the foundation for the cluster.

THE IMPACT OF SPECIALIZED PROCESSING: VECTOR PROCESSING

All of these techniques attempted to provide something akin to a simple execution model: if a program required $N_{program}$ processor clock cycles to complete, and the processor had a clock cycle as measured in nanoseconds of τ_{cycle}, then the total execution time is equal to $\tau_{completion}$.

$$\tau_{completion} = N_{program} \cdot \tau_{cycle}$$

This assumes that each instruction takes the same length of time to execute. Though this is not generally the case, it could serve as a rough approximation for these machines.

A vector program would have a vector portion and a non-vector portion, where N_{vector} is the number of cycles executed on the vector processor, the $\tau_{non\text{-}vector\text{-}cycle}$ is the clock cycle of the scalar processor, the $\tau_{vector\text{-}cycle}$ is the clock cycle of the vector processor, and the A is average vector length (i.e., how much additional work can be done on average over the whole of the program, in the vector processor).

$$\tau_{total} = (N_{program} - N_{vector}) \cdot \tau_{non\text{-}vector\text{-}cycle} + \frac{N_{vector} \cdot \tau_{vector\text{-}cycle}}{A}$$

So if there were an average vector length of 64 and a vector clock cycle close to the non-vector clock cycle, it would be possible to get excellent performance (lower τ_{Total} value) by moving as much code as possible into the vector section of the program.

Put another way, the number of clock cycles required for program execution remained "constant," but how they are divided among the functional units, and how efficiently they could use all the functional units' resources, governed the performance achieved. The more functional resources leveraged per unit time, the higher the efficiency, and therefore the higher the apparent speedup over serial code.

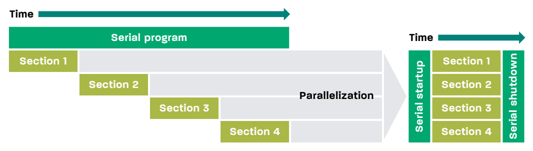
THE IMPACT OF SPECIALIZED PROCESSING: PARALLEL PROCESSING

Since computer design has largely followed the von Neumann architecture, parallel processing yielded a remarkably similar set of equations. Gene Amdahl noted this when he made the case for the law that now bears his name. The basic concept is that total program time is the sum of the time to execute the serial and parallel components:

$$\tau_{program} = \tau_{serial} + \tau_{parallel}$$

For a fixed CPU clock cycle of length τ_{cpu}, and again, assuming the instructions all take the same amount of time[1], we can write the execution time in terms of numbers of instructions executed.

$$N_{program} = N_{serial} + N_{parallel}$$



The number of instructions executed will be constant, no matter how they are subdivided among processors. In this representation, the number of instructions executed in the parallel section $N_{parallel}$ may be represented by $A(N_{cpu})\ N_{parallel\text{-}per\text{-}cpu}$, where N_{cpu} is the number of CPUs running on the parallel program, and $A(N_{cpu})$ is the parallel speedup—basically the ratio of the performance on a single CPU to the N_{cpu} CPUs. In a typical analysis, this is divided by $\tau_{program}$ and called the ratio of τ_{serial} to $\tau_{program}$. "s" is the serial fraction of the program and the ratio of $\tau_{parallel}$ to $\tau_{program}$, or "p," is the parallel fraction of the program. The equation reads:

$$1 = s + p$$

or

$$1 - p = s$$

The green represents the parallel portion and red represents the serial portion in the figure.

The parallel fraction "p" means that each N_{cpu} executes $1/N_{cpu}$ of the parallel cycles of the program. This is analogous to the vector processor's "A" or average vector length.

This immediately gives rise to Amdahl's law that predicts the best-case speedup $S(N_{cpu})$ (i.e., the ratio of the execution time of the serial and parallel portions of the program) as:

$$S\,(N_{cpu}) = \frac{1}{(1-p) + \frac{p}{N_{cpu}}}$$

As increasingly large clusters are used, the N_{cpu} grows and the ratio p/N_{cpu} becomes infinitesimal. This results in the maximum *acceleration* $S(N_{cpu})$ that can be expected, governed entirely by the serial or non-vector fraction of the program.

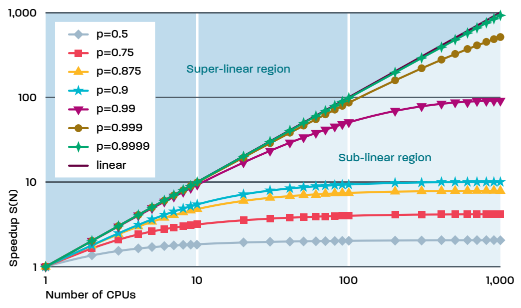
$$S(\infty) \approx 1/s$$

That is, regardless of whether a system uses a shared memory parallel, distributed memory parallel, or vector program, there are fundamental performance limits as the number of CPUs or vector elements increases.

For most programs, S(Ncpu) is less than Ncpu. This is called the sub-linear region. So for a parallel program using 1,000 CPUs, operating in a 99% parallel manner (p=0.99), the program will achieve a speedup of less than 100. This means that less than 100/1,000 or 1/10th of the resources are effectively being used. Increasing the number of processors will not increase the speedup for this program. In other words, simply throwing more processors at the problem will not solve it any faster.

AMDAHL'S LAW

Speedup as a function of parallel fraction (p) and number of CPUs.



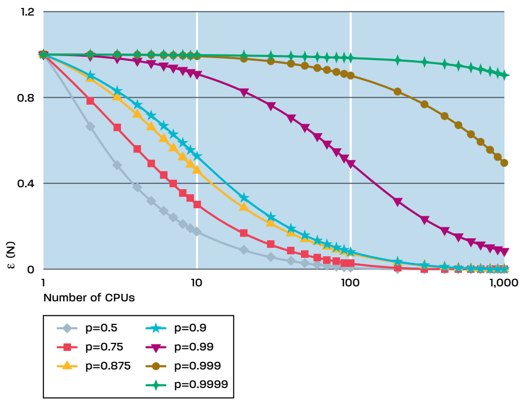
The efficiency $\varepsilon(Ncpu)$ of this process may be defined as:

$$\varepsilon(N_{cpu}) = S(N_{cpu}) \,/\, N_{cpu}$$

This efficiency is always less than one for normal (i.e., non-superlinear) programs. It is also dominated by the serial fraction at high CPU counts. Not much can be done to improve upon this, apart from minimizing serial work in the program or hiding it in a parallel thread so that it is non-blocking in relation to the parallel program.

EFFICIENCY

Efficiency as a function of number of CPUs at a fixed parallel fraction.

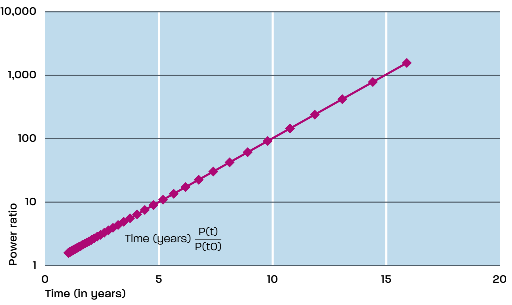


Clusters offer a low-cost mechanism that provides this increase in available processing cycles. As a consequence, the aggregate amount of processing cycles is quite large — though, as Amdahl's law shows, there are fundamental performance limits in this approach. Clusters, like parallel and vector machines, provide more processor cycles per unit of time than a single processor. In this sense, these systems are effectively cycle multipliers, providing N_{cpu} cycles or $A_{vector\text{-}length}$ cycles per unit processor clock cycle.

THE IMPACT OF UNDERLYING PROCESSOR TECHNOLOGY ON PERFORMANCE

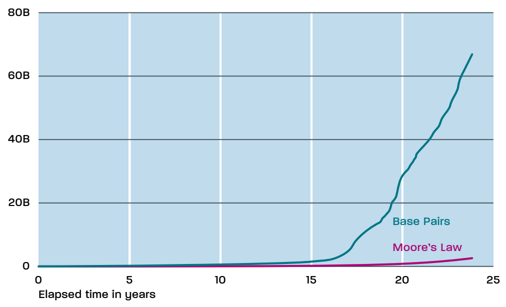
In addition, programs are fundamentally bound by the performance of the underlying processor technology. This performance and power curve has closely followed Moore's Law (which states that general processor performance will double every eighteen months) over the past three decades.

Scaled processor power as a function of thime assuming Moore's lawholds..



If data growth rates are not increasing faster than the underlying processor technology, this is not a problem. However, for large groups of researchers, engineers and scientists whose data sets are growing at exponential rates, this is a major challenge.

GROWTH OF (BASE PAIRS, MOORE'S LAW)



For example, consider data growth at Genbank, the National Institutes of Health's genetic sequence database. The data, found at ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt, has the classic hockey stick shape indicative of exponential growth. It is instructive to normalize Moore's law against the base pair data and compare how processing power has grown in this time.

Some end-users have approached this challenge by waiting for the underlying technology to generate an order of magnitude or more improvement in performance for general-purpose processors. Hypothetically, the user can then buy more performance for less money, assuming power, space, cooling or other considerations are not limiting factors. Unfortunately, this strategy has significant shortcomings including:

→ The lost value caused by delays in time-to-insight that could be captured by performing analysis sooner rather than later.

→ The immediate opportunity cost associated with performing more complex calculations faster. In commercial applications, like financial analysis, underperforming platforms carry economic penalties.

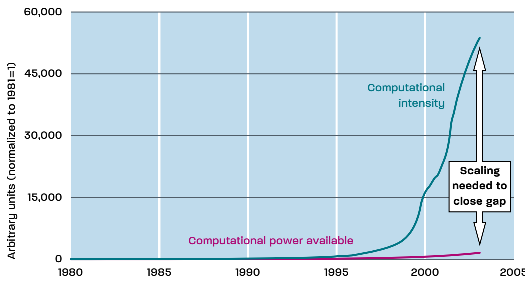
→ The underlying assumption that the growth of raw data requiring analysis will return to a more manageable level, allowing processing power to catch up.

Finally, this approach is based on an implied assumption that performance improvements in the underlying processor technology will continue indefinitely. This may unfairly discount future challenges associated with the process technologies, device physics, thermodynamics, and fundamental physical limitations of new processors.

A CURRENT EXAMPLE: GENOMIC RESEARCH

Using genomic data as an example, and postulating that computational analytical needs (or computational intensity) scales at least as fast as the rate of data growth, the best-case scenario would be that computational intensity is proportional to volume of data. In that case, underlying general-purpose processor technologies would provide the computing power scaling needed for current and future analysis.

GROWTH OF BIO-IT COMPUTATIONAL INSTENSITY AND COMPUTATIONAL POWER



However, this is not the case. Data is growing exponentially faster than computing power, overwhelming the ability of researchers to model, simulate, and analyze. The above graph plots data volume and the computational power available to process it as a function of time. The gap represents a large and rapidly growing problem for life-science researchers. Similar data from other fields shows that this trend is not limited to life sciences.

The best acceleration or speedup with general-purpose processors will be in solutions that emphasize parallel execution. That would require a program that is either exceptionally well tuned for parallel operation, or parallel by design, such as a Monte Carlo code.

Order-of-magnitude changes in run times are reasonable goals. If a computation would ordinarily require six months of processing time, an order of magnitude speedup would reduce the run time to about 18.3 days. Unfortunately, this is still not fast enough to satisfy most end-users. Providing 100x (two orders of magnitude) speedup would reduce the run time to slightly less than two days, making it possible to perform roughly 15 such runs per month. This could enable significantly more simulation per unit time. If the execution time could be reduced by one more factor of ten (three orders of magnitude, or 1,000x speedup), each run would require only about 4.4 hours.

This additional processing power comes with a high operating cost—a drastic increase in energy consumption per CPU. Common servers require about .25 kilowatt/hour to operate, and more than that to cool. Hundreds of thousands of processors, as some groups are considering, may require hundreds of megawatts of electrical and cooling power, hence the interest by some large companies in locating close to inexpensive hydroelectric power sources.

So the computing community faces both the limitations of existing processing technologies as demonstrated by Amdahl's law, and the challenges associated with the dependence on advances solely based on current processor technologies. One way forward is to consider alternative processing technologies.

ACCELERATED COMPUTING: USING SPECIALIZED SUBSYSTEMS TO ACHIEVE MAXIMUM PERFORMANCE

For the most part, processor designs have been die-shrunk and clock-speed bumped over time. This does not necessarily involve altering the processor's execution models or the way they are targeted with development tools. This means that on a given range of very similar hardware, programs will generally require a fixed number of clock cycles to execute, regardless of whether or not these are single- (serial) or multi-threaded (parallel) programs. Since the processing is symmetric with respect to processor interchange, the processing time does not vary when one element is used over another. This symmetry implies a conservation of processing cycles for these architectures.

Conversely, accelerated computing may be thought of as a grouping of a number of different computational paradigms. As previously noted, accelerated computing has taken the form of vector computing, symmetric multiprocessor computing, and parallel cluster computing. The latter currently dominates the high-performance computing (HPC) market.

These are not the only forms of accelerated computing in use. Specifically, dedicated circuits for offloading the processing of graphics calculations, I/O, and networking are common. These forms of accelerated computing are asymmetric, in that different programs run on different processors, and they talk via an application programming interface (API) to send messages and data to each other. These processors often do not have identical micro-architectures, and can not easily execute other processors' program code or microcode without an emulator layer. They generally are not as tightly tied to processor architecture as vector processors.

The fundamental advantage of this approach is that one can design a device or processor for a specific task. These specialized processors may be able to perform complex calculations that would require many hundreds or thousands of clock cycles on a general-purpose processor. Examples of these would include the nVidia GeForce 8800 processor, which is designed specifically to perform graphics calculations, or the ClearSpeed CSX600 processor, which is specifically designed to perform many double-precision floating-point and integer calculations in parallel.

Accelerated computing has extended to include field programmable gate arrays (FPGAs). An FPGA is effectively a blank semiconductor upon which the engineer may build the computing circuit needed for a task. This dedicated computing circuit could be, but usually is not, a processor. It is not a written program per se, but is more closely aligned with the notion of a designed circuit that needs to interface with the software program.

In most cases, processing is asymmetric. This means that if an engineer can take advantage of the design, or craft a unique design using an FPGA, it is possible to drastically alter the number of required processing cycles.

THE COST OF ACCELERATION: EVALUATING THE TRADEOFFS

Performance enhancements require altering software programs. Whether through more effective utilization of today's x86-based processors or through the use of specialized accelerators, it cannot be avoided. The important questions to ask are, how much alteration will the programs require, and will they be portable after alteration? Oddly enough, these questions are very similar to those asked when parallelization and SMP systems gained in popularity, and the answers are similar today.

End-users often hear that fast programs are not portable, and portable programs are not fast. The closer a program can get to the underlying architecture, the faster it can go. Conversely, the higher the level of abstraction, the heavier the performance penalty.

Programmers typically employ compilers to translate between higher-level languages and the physical system resources. These tools translate between individual lines of program code and machine instructions. When the programmer optimizes the code using the built-in optimization tools, the compiler tries to rearrange the computation or the instructions to better fit the computational model of the underlying architecture. Apart from various highly specialized cases, this optimization is not as efficient as one could realize at a lower level of code, with fewer abstractions in the way.

Parallel programs represent a compromise, and if they are written to a particular specification such as MPI, they may be quite portable. To make a program fast on a particular platform, it is critical that the programmer efficiently exploit the underlying physical processor resources.

A program can now be effectively tied to a particular processor or machine micro-architecture, but the programmer will not produce as many useful modules per unit time due to the extra effort to optimize. The advantage is that individual modules that are produced can very efficiently utilize the underlying machine model.

The AMD Core Math Library (ACML) is an excellent example. It represents hundreds of thousands of hand-coded lines of low-level program code. ACML provides a standard API to the end-user for the ACML code. This way, users can take advantage of accelerated SIMD (miniature vector) code for specific calculations, without needing to know anything about the details of the calculation. Any user can leverage this acceleration in their program simply by linking to the ACML code, possibly achieving 85% or more resource utilization efficiency on the processors. It may not be "portable" outside of its micro-architectural model, but it is fast. This is an example of software-based acceleration that illustrates the tradeoff the end-user must consider.

Hardware acceleration provides another mechanism, and can be achieved using symmetric (SMP) or asymmetric (aSMP) multiprocessing. SMP includes clusters and traditional shared-memory machines, while aSMP includes specialized processor units and FPGAs.

Generally, SMP programs are easier for developers to produce than aSMP programs, and are typically more portable. The drawback is that they offer lower acceleration—typically less than 4x faster per core in the best-case scenarios.

aSMP programs are generally harder for programmers because they often require algorithmic shifts, program flow modification, and more involved test cases. The benefits include 4x to 10x faster acceleration, with some exceptional codes achieving 100x and better per node.

When graphics processing units (GPUs) are used as the accelerator, the programmer will have to choose one of several competing execution models (GPGPU, Cuda, Cg, RapidMind, PeakStream) for coding, and these systems are not interchangeable.

EFFICIENT RESOURCE UTILIZATION: CRITICAL FOR ACCELERATION

As noted earlier, parallel execution rapidly loses efficiency at high processor counts, so this mechanism has limited applicability for overall acceleration. The more efficiently underlying resources can be utilized, the better the performance. If a computing resource can provide a 10x speedup over current technology, but can only be used at 10% efficiency, it is not likely that any speedup will be realized.

Each processor may have a particular mix of instructions that it can execute per clock cycle. Software designed to take advantage of this mix, such as ACML, can achieve excellent overall efficiency. What aSMP offers over SMP is that the underlying physical model can be designed for efficient resource utilization of particular calculations. If the underlying processor can take long vectors and rapidly perform dot products, it is helpful to expose it to the programmer via an API.

This is the critical aspect of any accelerator technology. By increasing efficiency of resource utilization, more efficient processing can be enabled per unit time, not simply more cycles per unit time.

ACCELERATOR TECHNOLOGIES: DEFINITIONS AND TYPES OF ACCELERATORS

Accelerators are processors or circuits specifically designed to increase performance of particular calculations. They do this by exploiting efficient use of resources and employing implicitly parallel operations. Accelerators may be closely coupled to processors, as in the SIMD unit, the SSE, or Altivec system onboard many of today's popular processors; more strongly coupled via HTX, using a bus such as PCIe or PCIx; or weakly coupled via Gigabit, USB2, or Firewire connections.

Accelerators are designed for particular calculations. For example, FPGAs are not processors or processing elements in a strict sense, but are rather circuits dedicated to execute a particular calculation.

Accelerators fall into several classes based on the focus of their processing and how much power they consume.

Technology	Performance (GFLOP S/D)	Focus: Single, Double, Integer, strings	Power consumption per unit (watts)
CPU (dual core): General processor	10/10	S/D/I/s	90
Graphics processor	500/-	S	200
Specialty processor	25/25	S/D/I	10
Gaming processor	220/12	S	50
FPGA	150/5[2]	S/D/I/s	10

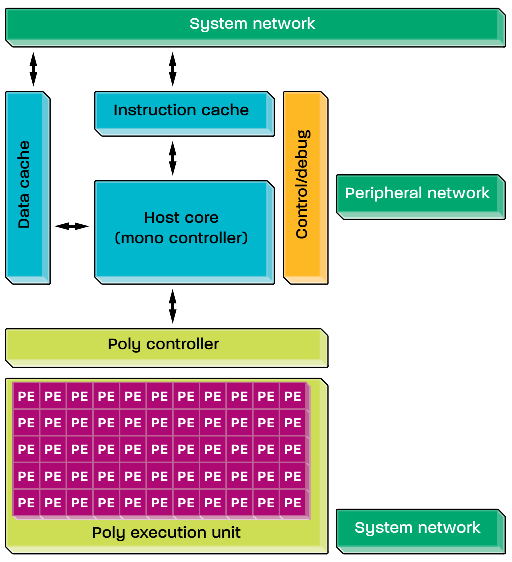
CPUS may have acceleration technology co-located on silicon. For example, integrating the SSE or Altivec SIMD units onto popular processors can increase the amount of data operated on per clock cycle. Aggregations of CPUs into an SMP or cluster can also be considered an accelerator of sorts, increasing the number of available clock cycles per unit of time. With the SIMD example, performance gains will be modest at best, typically improving by only 1.3x to 4x.

GRAPHICS PROCESSORS (GPU) are predominately used for specific computations on graphical objects. Given screen refresh rates, scene size, and other constraints, these calculations need to be repeated with great speed. Using OpenGL and similar technologies, graphical objects are described in terms of mathematical constructs: vertices, surfaces, normals, and operations are performed upon these objects. The GPU performs the calculations by streaming the scene or traversing a scene graph, operating upon the data, and performing the necessary calculations. Recent GPUs use single-precision floating-point calculations, which are often highly parallel. With a sufficient number of pipelines for processing, current GPUs are theoretically able to achieve 500 GFLOPS in single-precision calculations. Currently, they cannot easily perform double-precision calculations.

If a program has elements that can be described in a manner conducive to the shader language, Cg, CUDA, or any of the other emerging technologies, it may be able to tap into significant processing power. However, with this technology now in the formative stages, it is important to note that there may be a fairly wide gap between maximum theoretical performance and end-user observed performance.

SPECIALTY PROCESSORS are designed for specific tasks. For example, ClearSpeed makes a specialty processor named the CSX600™, which has 96 processing elements arrayed on a single silicon chip.

CLEARSPEED CSX600™ PROCESSOR BLOCK DIAGRAM



Each processing element in the Poly Execution Unit contains a 32- and 64-bit floating-point multiplier, a 32- and 64-bit floating-point adder, an integer arithmetic logic unit (ALU), a 16-bit MAC (multiply-and-accumulate), a 128-byte register file (16 double-precision or 32 single-precision numbers), 6 kB of SRAM, and a high-speed I/O channel. The chip itself is capable of sustaining 25 GFLOPS in DGEMM operations, which are generalized matrix-matrix products often used as the basis for higher-level linear algebra routines in LAPACK and its accelerated variants including ACML, Goto, and MKL. Since it has internal memory on each of its processing elements, it can move on the order of 96 GB/s from this memory to its associated processor. It can access DDR2 ECC RAM at 3.2 GB/s and communicate with another chip via a duplex channel at 3.2 GB/s in each direction.

GAMING PROCESSORS such as the Cell-BE offer interesting possibilities for scientific computing. The system is designed with a central processing core based upon a low power/speed PowerPC processor and connects with eight "synergistic" processor elements (SPE). The design of this chip returns to the core principles of RISC architecture, in which the mantra was "simpler is better." The PowerPC core handles many functions while the SPEs have vector registers as well as high-performance memory access to their local storage. The unit was designed to enable the SPEs to perform many single-precision calculations simultaneously. Each SPE is capable of 25.6 GFLOPS in single precision and 1.83 GFLOPS in double precision. With eight such units, the Cell-BE as a whole is capable[3] of about 205 GFLOPS in single precision and about 15 GFLOPS in double precision. The memory bandwidth per SPE is about 26 GB/s. However, because there is only one DRAM controller, memory access is serialized for the chip, and is as fast as that of a single SPE.

These processors are being manufactured with significant economies of scale, but not for accelerating high-performance computing. For this market, chips that cost more than several hundred dollars would not be acceptable. This suggests that cost-effective acceleration may be possible with an appropriate board design.

FIELD PROGRAMMABLE GATE ARRAYS (FPGAS) are literally collections of logic gates waiting for signal routing, connection, and other information. They enable the creation of a dedicated computational circuit that does not contain any extraneous or unused logic, and therefore tend to be self-contained. They may contain processor cores, and allow additional core logic or libraries of specific circuits to be used on them. There are several methods of building computational circuits, the highest-performing of which would involve low-level logic design and modeling with VHSIC hardware description language (VHDL). Other lower-performing options involve using compiler systems, which attempt to convert various portions of programs into "virtual processor" logic circuitry. The problem with these virtual processors is that they are significantly less efficient than VHDL.

This Progeniq Bioboost accelerator card is used in some of the data collection for the quantitative portion of this report. Note that on the left of this PCI-compatible card, there is a power connector and a USB 2.0 port. Despite the low-performance interface (480 Mb/s), this card was able to provide about seven to 10x better performance than the pure software version of the HMMER application (a popular tool for nucleic acid and protein sequence analysis).

PERFORMANCE BENEFITS AND PORTING ISSUES

With the accelerator technologies discussed, there is often a significant benefit to applying these tools to specific problems. Performance may vary significantly from roughly 5x better to well over 100x per acclerator.

It is important to note that the accelerator may only impact one aspect of a particular code. To estimate the potential impact, a typical scenario needs to be considered, its execution profiled, and a particular pattern sought. An execution profile indicating a significant amount of time spent in computationally expensive routines would offer an important clue as to whether or not acceleration is possible.

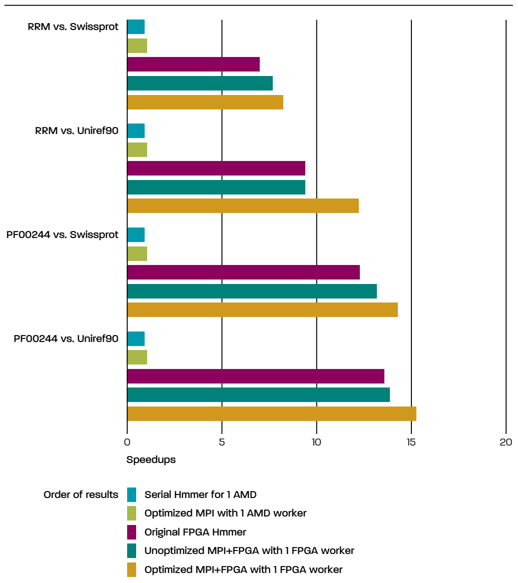
With the HMMER program, for example, about 98 percent of the execution time was spent in one function call during typical runs of HMM search. Even if all that processing time could be diminished to zero with an infinitely fast accelerator, the maximum speedup would be 50x. That is, $S(\infty) \approx 1/s = 1/0.02$, exactly analogous to Amdahl's law.

This suggests that single-layer acceleration techniques may not be able to achieve 1000x and better performance. As with the parallel and vector systems, there are fundamental limits to the performance of a particular technology.

Users can combine a variety of techniques and exploit multiple layers of acceleration: accelerating time-expensive portions on each computing node, and using parallel programming and serial optimization methods to distribute larger work to these accelerated nodes.

Using software techniques on single processors, researchers were able to provide an about 1.6x to 2.5x speedup. Combining these with a rewriting[4, 5, 6] of the serial and parallel portions of HMMER, they achieved significantly better performance. Using a single FPGA card (the Progeniq unit pictured on previous page) in conjunction with the MPI-HMMER yielded even better speedup.

HMM MODEL VS. DATABASE



Additional work will be reported with multiple FPGAs combined with serial and parallel optimization techniques. The initial data is quite encouraging, in that the parallel performance on sixteen CPUs is comparable to the performance of a single FPGA. With multiple FPGAs and larger work units, significantly better overall performance is expected.

One serious performance point not normally discussed by vendors is the ability to fit an algorithm onto a particular accelerator. For the FPGA HMMER code noted above, it is restricted to 256 states in the HMM. To enable larger HMMs, a reduction in the number of HMM processing elements would be required to fit the larger elements onto the FPGA. This in turn would reduce the overall performance, as the FPGA is processing significant amounts of data in parallel. Programmers often have to choose between more PEs and larger models.

When FPGAs are used for floating-point work, careful consideration should be given to the trade-off of performance versus compatibility with IEEE754 specifications, the widely used standard for floating-point computation. Implementing full IEEE754 specifications is expensive in terms of gates, and those functional units are rarely used in their entirety. FPGAs can be of enormous benefit when they can be fitted with large numbers of logic PEs per unit and kept busy. This offers the benefits of explicitly parallel operation with dedicated computational logic and an excellent performance combination, especially if users can relax the precision and IEEE754 requirements. MD-Grape systems are able to achieve in excess of 150 GFLOPS per FPGA and may have four to eight FPGAs per card.

While the performance of these units can be excellent, it requires changing programming paradigms. Users need to design a circuit for computing, not write software to compute. This is a significant difference, but when additional performance is required, the benefits are worth the effort.

In the case of the ClearSpeed unit, programming can be done at a low level via assembly, or at a higher level using their extended C compiler. This compiler provides an interface to the parallelism in a relatively simple manner, reminiscent of OpenMP. It introduces a new keyword — "poly" — that serves to define global variables, as well as PE indexing and other related functions. As with other programming, better performance will be obtained when carefully matching the expression of the problem to the underlying processor. This is usually most effectively done at a lower level to minimize the time-cost overhead of abstraction in higher-level languages.

GPU programming is also affected using "custom" languages such as Cg, RapidMind, PeakStream, and others. Some data types are well supported, such as integers. Some are partially supported, such as single-precision, in that IEEE754 may not be fully implemented. And some data types are not supported in any meaningful manner, such as double-precision. This currently limits the effectiveness of these platforms for calculations on problems requiring double precision.

Cell-BE programming is inherently an exercise in parallel assembly code development, with the parallelism implemented in terms of message boxes/message passing constructs. Compilers are being developed by IBM and others to provide higher-level programming support.

In every case, the best performance may be delivered by working at the lowest levels of programming. While making code easier to maintain, the higher levels of abstraction and language will not provide the maximum performance benefit. As with balancing performance against IEEE754 compliance, the programmer needs to balance development time and effort against expected payback.

None of these systems provides “drop-in,” or zero-effort, acceleration of software. All require some work to get applications onto them. There are currently few standards to work against, with some of the technology entirely proprietary and others quite open.

Some ship with real applications or applications available. All have development tools which range from nominally priced to very expensive.

INTERFACE ISSUES

In all cases, these accelerators exhibit massive internal bandwidths between their PEs. Unfortunately, they are not well matched with the rest of the system, as the ratio between the system communication bandwidth (PCIe, PCIx, or USB2) and the internal bandwidth is often 0.1 or less. This huge discrepancy drives the need to perform buffering operations as part of the solution porting/development process. Programmers may already be familiar with these techniques since single- and double-buffering are used to overlap communications and calculations in other scenarios, such as non-blocking MPI programming.

In double-buffering, DMA transfers are enabled to fill or empty one buffer for a PE while the other buffer is being processed by the PE. This leverages asynchronous programming technique and requires to the ability to inspect locks and semaphores.

While double-buffering may provide some relief and data access impedance matching, accelerators will likely remain bandwidth-constrained for some time to come. Existing PCIx buses cannot sustain more than one GB/s. And PCIe buses are typically limited to x16 slots, which provide eight GB/s in aggregate or four GB/s in each direction. Such buses limit data transfer rates for accelerators, which can individually consume more than three GB/s and six GB/s in pairs. Moreover, memory bandwidth on the host machine is generally insufficient for an accelerator to process data at its maximum potential capacity. While a gigabyte or two of high-performance onboard memory would be helpful, data motion within the system requires a high-speed, tightly coupled connection between the memory and I/O systems, the processors, and the accelerator.

Lower-speed connections are possible and offer acceleration that may meet the needs of some end-users, in which case these systems may be designed without the more expensive high-speed circuitry.

SUMMARY

Accelerators could significantly benefit high-performance computing. When combined with systems that provide data motion rates sufficient to keep an accelerator busy, they may provide acceleration of an order of magnitude or more per node, at costs comparable to or less than several nodes of a standard Linux cluster. Even accelerators coupled via lower-performance buses can provide significant acceleration and performance.

More important is the possibility of coupling multiple layers of acceleration and getting benefits from each. SIMD at the host CPU, accelerators coupled to the host CPU, and parallelization may enable multiple orders of magnitude of performance gain for a cost far below that of a traditional parallel system and code development effort.

The need for heterogeneous computing has arrived. It is expected that lowering barriers and making technologies widely available at low cost is a good way to increase the market, especially when rapid data insights are imperative. And over the next several years, the market will select the winning technologies, which will likely be easy to program, easy to use, and relatively low cost.

Joseph Landman Ph.D., is Founder and CEO of Scalable Informatics LLC, a Canton, Michigan based, privately owned high performance computing and storage solutions company. He has been a user of high performance computing systems and tools since 1986, and working on the vendor side since 1995. Previously Dr. Landman has held high performance computing technical posts at MSC.Software, and SGI. Dr. Landman has a Ph.D in Physics from Wayne State University, a Masters in Physics from Michigan State and a Bachelors in Physics from SUNY Stony Brook.

AMD
Smarter Choice

Advanced Micro Devices
One AMD Place
Sunnyvale, CA 94088-3453
www.amd.com

1. Note: this is not the case in x86 and x86-64 architectures. Instructions may take a wide range of cycle counts to complete. In these cases, this analysis is still usable, incorporating the average instruction execution time for τ cpu rather than simply the clock cycle time.
2. If willing to forgo IEEE754 mathematical operations, FPGAs can be quite fast. These operations are important to reproducibility of calculations, they specify actions and handling of underflow, overflow, exceptions, and related. Unfortunately, they also consume a significant number of gates on the FPGA, which limits the number of them per FPGA.
3. c.f. http://www.cs.berkeley.edu/~samw/projects/cell/CF06.pdf
4. J.P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER sequence analysis suite using conventional processors. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA '06)*, pages 289–294, Washington, DC, USA, 2006. IEEE Computer Society.
5. J.P. Walters, J. Landman, and V. Chaudhary. Optimized cluster-enabled HMMER searches. In E.G. Talbi and A. Zomaya, editors, *To appear in Grids for Bioinformatics and Computational Biology*. Wiley & Sons, 2007.
6. J. Landman, J. Ray, and J. P. Walters. Accelerating HMMER searches on AMD Opteron processors with minimally invasive recoding. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 2 (AINA '06)*, pages 628–636, Washington, DC, USA, 2006. IEEE Computer Society.