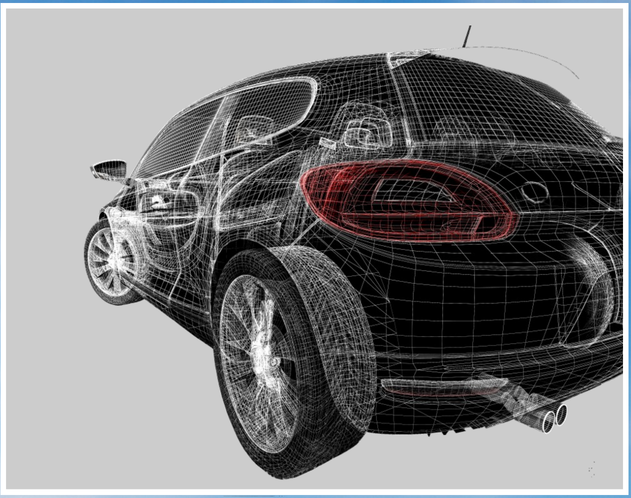
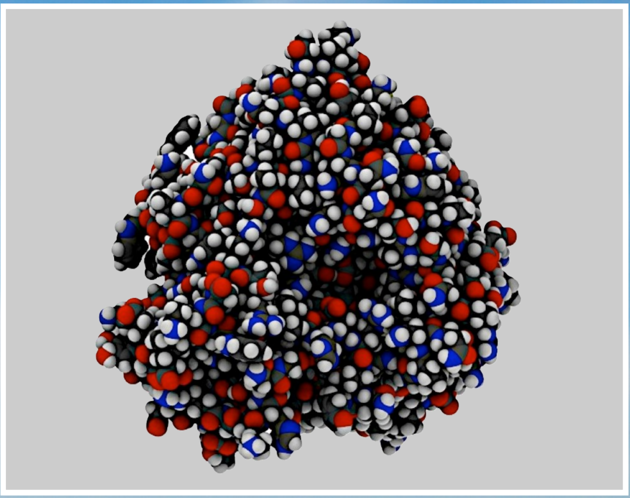
The GPU Computing Revolution

From Multi-Core CPUs to Many-Core Graphics Processors

A Knowledge Transfer Report from the London Mathematical Society and Knowledge Transfer Network for Industrial Mathematics

By Simon McIntosh-Smith





Front cover image credits:

Top left: Umberto Shtanzman / Shutterstock.com
Top right: godrick / Shutterstock.com
Bottom left: Double Negative Visual Effects
Bottom right: University of Bristol
Background: Serg64 / Shutterstock.com

THE GPU COMPUTING REVOLUTION

From Multi-Core CPUs To Many-Core Graphics Processors

By Simon McIntosh-Smith

Contents

September 2011

A Knowledge Transfer Report from the London Mathematical Society and
the Knowledge Transfer Network for Industrial Mathematics
Edited by Robert Leese and Tom Melham

London Mathematical Society, De Morgan House, 57–58 Russell Square, London WC1B 4HS
KTN for Industrial Mathematics, Surrey Technology Centre, Surrey Research Park, Guildford GU2 7YG

AUTHOR

Simon McIntosh-Smith is head of the Microelectronics Research Group at the University of Bristol and chair of the Many-Core and Reconfigurable Supercomputing Conference (MRSC), Europe's largest conference dedicated to the use of massively parallel computer architectures. Prior to joining the university he spent fifteen years in industry where he designed massively parallel hardware and software at companies such as Inmos, STMicroelectronics and Pixelfusion, before co-founding ClearSpeed as Vice-President of Architecture and Applications. In 2006 ClearSpeed's many-core processors enabled the creation of one of the fastest and most energy-efficient supercomputers: TSUBAME at Tokyo Tech. He has given many invited talks on how massively parallel, heterogeneous processors are revolutionising high-performance computing, and has published numerous papers on how to exploit the speed of graphics processors for scientific applications. He holds eight patents in parallel hardware and software and sits on the steering and programme committees of most of the well-known international high-performance computing conferences.

www.cs.bris.ac.uk/home/simonm/ simonm@cs.bris.ac.uk

Executive Summary

Computer architectures are undergoing their most radical change in a decade. In the past, processor performance has been improved largely by increasing *clock speed* — the faster the clock speed, the faster a processor can execute instructions, and thus the greater the performance that is delivered to the end user. This drive to greater and greater clock speeds has stopped, and indeed in some cases is actually reversing. Instead, computer architects are designing processors that include multiple *cores* that run in parallel. For the purposes of this report we shall think of a core as one independent 'brain', able to execute its own stream of instructions, independently fetching the data it requires and writing its own results back to a main memory. This shift from increasing clock speeds to increasing core counts — and thus increasing parallelism — is presenting significant new opportunities and challenges.

During this architectural shift a new class of *many-core* computer architectures is emerging from the graphics market into the mainstream. This new class of processor already exploits hundreds or even thousands of cores. Many-core processors potentially offer an order of magnitude greater performance than conventional processors, a significant increase that, if harnessed, can deliver major competitive advantages.

But to take advantage of these powerful new many-core processors, most users will have to radically redesign their software, potentially needing completely new algorithms that can exploit massive parallelism.

The many-core opportunity

For computationally challenging problems in business and scientific research, these new, highly parallel, many-core architectures represent a technological breakthrough that can deliver significant benefits in science and business benefits. Such a step-change in available computing performance could enable faster financial trades, higher-resolution simulations, the design of more competitive products such as increasingly aerodynamic cars and more effective drugs for the treatment of disease.

Thus new many-core processors represent one of the biggest advances in computing in recent history. They also indicate the direction of *all* future processor design — we cannot avoid this major paradigm shift into massive parallelism. All processors from all the major vendors will become many-core designs over the next few years. Whether this is many identical mainstream CPU cores or a heterogeneous mix of different kinds of core remains to be seen, but it is inevitable that there will be lots of cores in each processor, and importantly, to harness their performance, most software and algorithms will need redesigning.

There is significant competitive advantage to be gained for those who can embrace and exploit this new, many-core technology, and considerable risk for any who are unable to adapt to a many-core approach. This report explains the background to these developments, presents several success stories, and describes a number of ways to get started with many-core technologies.

From Multi-Core to Many-Core: Background and Development

Historically, advances in computer hardware have delivered performance improvements that have been transparent to the end user – software has run more quickly on newer hardware without having to make any changes to that software. We have enjoyed decades of ‘single thread’ performance improvement, which relied on converting exponential increases in available transistors (known as Moore’s Law [83]), into increasing processor clock speeds and advances in microprocessor pipelining, instruction-level parallelism and out-of-order execution. All these technology developments are essentially ‘under the hood’, with existing software generally performing faster on next generation hardware without programmer intervention. Most of these developments have now hit a wall.

In 2003 it started to become apparent that several fundamental physical limitations were going to change everything. Herb Sutter noticed this significant change in his widely cited 2004 essay ‘The Free Lunch is Over’ [110]. Figure 1 is taken from Sutter’s paper, updated in 2009, and shows how single thread performance increases, based on increasing clock speed and instruction level parallelism improvements, could only have continued at the cost of significant increases in processor power consumption. Something had to change — the free lunch was over.

The answer was, and is, parallelism. It transpires that, for various semiconductor physics-related reasons, a processor containing two 3GHz cores will use significantly less energy (and thus dissipate less waste heat) than an alternative processor containing a single 6GHz core. Yet in theory the dual-core 3GHz device can deliver the same performance as the single 6GHz core (much more on this later). Today even laptop processors contain at least two cores, with most mainstream server CPUs already containing four or six cores. Thus, due to the power consumption problem, increasing core counts have replaced increasing clock speeds as the main method of delivering greater hardware performance. (For more details see Box 1 later.)

Of course the important implication of ‘the free lunch is over’ is that most existing software will not just go faster when we increase the number of cores inside a processor, unless you are in the fortunate position that your performance-critical software is already ‘multi-core aware’.

So, hardware is suddenly and dramatically changing: it is delivering ever more performance, but is now doing so through rapidly

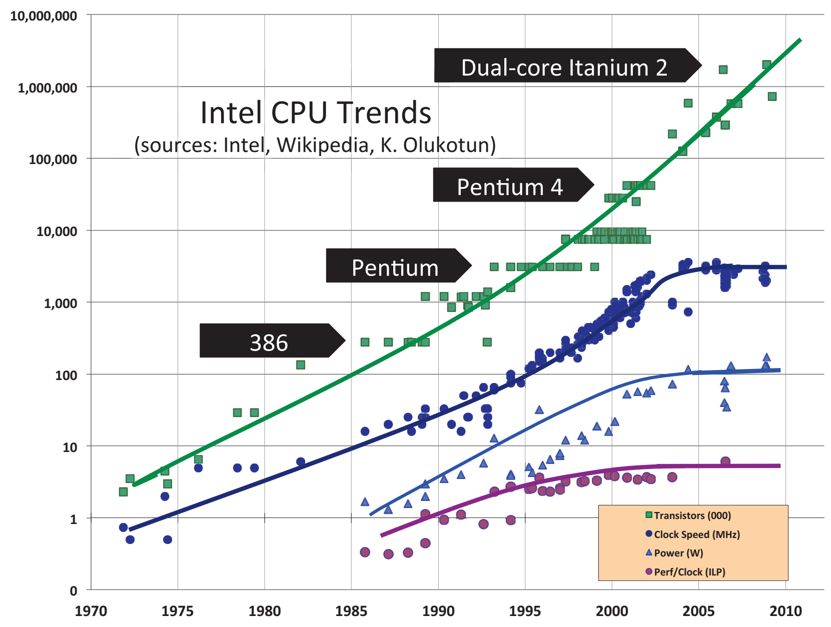


Figure 1: Graph of four key technology trends from ‘The Free Lunch is Over’: transistors per processor, clock speed, power consumption and instruction level parallelism [110].

increasing parallelism. Software will therefore need to change radically in order to exploit these new parallel architectures. This shift to parallelism represents one of the largest challenges that software developers have faced. Modifying existing software to make it parallel is often (but not always) difficult. Now might be a good time to re-engineer many older codes from scratch.

Since hardware designers gave up on trying to maintain the free lunch, incredible advances in computer hardware design have been made. While mainstream CPUs have been embracing parallelism relatively slowly as the majority of software takes time to change to this new model, other kinds of processors have exploited parallelism much more rapidly, becoming massively parallel and delivering large performance improvements as a result.

The most significant class of processor to have fully embraced massive parallelism are *graphics processors*, often called Graphics Processing Units or *GPUs*. Contemporary GPUs first appeared in the 1990's and were originally designed to run the 2D and 3D graphical operations used by the burgeoning computer games market. GPUs started as fixed-function devices, designed to excel at graphics-specific functions. GPUs reached an inflection point in 2002, when the main graphics standards, OpenGL [68] and DirectX [82] started to require general-purpose programmability in order to deliver advanced special effects for the latest computer games. The mass-market forces driving the development of GPUs (over 525 million graphics processors were sold in 2010 [63]) drove rapid increases in GPU performance and programmability. GPUs quickly evolved from their fixed-function origins into fully-programmable, massively parallel processors (see Figure 2). Soon many developers were trying to exploit these low-cost yet incredibly high-performance GPUs to solve non-graphics problems. The term 'General-Purpose computation on Graphics Processing Units' or *GPGPU,* was coined by Mark Harris in 2002 [51], and since then GPUs have continued to become more and more programmable.

Today, GPUs represent the pinnacle of the many-core hardware design philosophy. A modern GPU will consist of hundreds or even thousands of fairly simple processor cores. This degree of parallelism on a single processor is usually called 'many-core' in preference to 'multi-core' — the latter term is typically applied to processors with at most a few dozen cores.

It is not just GPUs that are pushing the development of parallel processing. Led by Moore's Law, mainstream processors are also on an inexorable advance towards many-core designs. AMD started shipping a 12-core mainstream x86

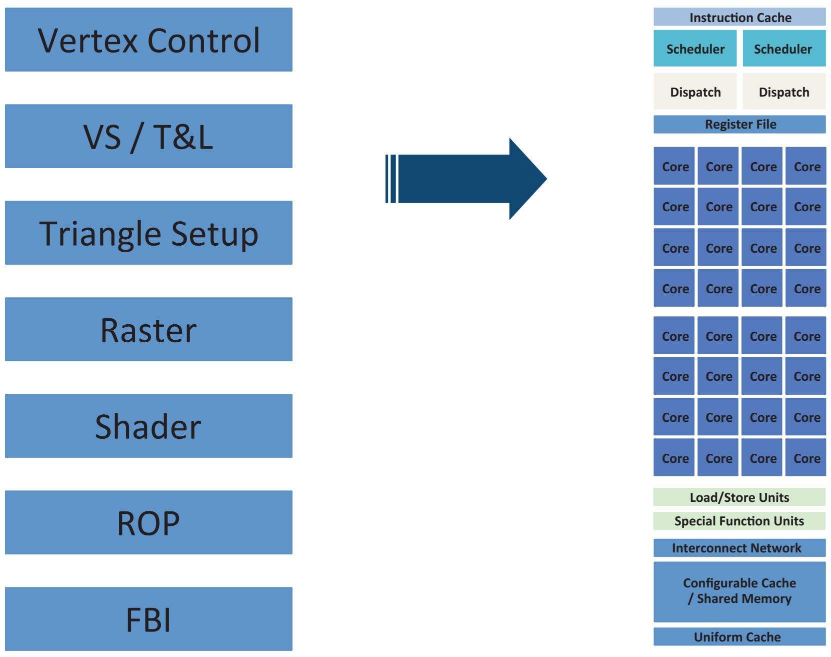


Figure 2: The evolution of GPUs from fixed function pipelines on the left, to fully programmable arrays of simple processor cores on the right [69].

CPU codenamed Magny-Cours in 2010 [10], while in Spring 2011 Intel launched their 10-core Westmere EX x86 CPU [102]. Some server vendors are using these latest multi-core x86 CPUs to deliver machines with 24 or even 48 cores in a single 1U 'pizza box' server. Early in 2010 Intel announced a 48-core x86 prototype processor which they described as a 'cloud on a chip' [61]. These many-core mainstream processors are probably the future of server CPUs, as they are well suited to large-scale, highly parallel workloads of the sort performed by the Googles and Amazons of the internet industry.

But perhaps the most revealing technology trend is that many of us are already carrying heterogeneous many-core processors in our pockets. This is because smartphones have quickly adopted this new technology to deliver the best performance possible while maintaining energy efficiency.

It is the penetration of many-core architectures into all three predominant processor markets — consumer electronics, personal computers and servers — that demonstrates the inevitability of this trend. We now have to pay for our lunch and embrace explicit parallelism in our software but, if embraced fully, exciting new hardware platforms can deliver breakthroughs in performance and energy efficiency.

Box 1: The Power Equation

Processor power consumption is the primary hardware issue driving the development of many-core architectures. Processor power consumption, P, depends on a number of factors, which can be described by the power equation,

$$P = ACV^2 f\,,$$

where A is the activity of the basic building blocks, or *gates* in the processor, C is the total capacitance of the gates' outputs, V is the processor's voltage and f is its clock frequency.

Since power consumption is proportional to the square of the processor's voltage, one can immediately see that reducing voltage is one of the main methods for keeping power under control. In the past voltage has continually been reducing, most recently to just under 1.0 volt. However, in the types of CMOS process from which most modern silicon chips are made, voltage has a lower limit of about 0.7 volts. If we try to turn voltage down lower than this, the transistors that make up the device simply do not turn on and off properly.

Revisiting the power equation, one can further see that, as we build ever more complex processors using exponentially increasing numbers of transistors, the total gate capacitance C will continue to increase. If we have lost the ability to compensate for this increase by reducing V then there are only two variables left we can play with: A and f, and it is likely that both will have to reduce.

The many-core architecture trend is an inevitable outcome from this power equation: Moore's Law gives us ever more transistors, pushing up C, but CMOS transistors have a lower limit to V which we have now reached, and thus the only effective way to harness all of these transistors is to use more and more cores at lower clock frequencies.

For more detail on why power consumption is driving hardware trends towards many-core designs we refer the reader to two IEEE Computer articles: Mudge's 2001 'Power: a first-class architectural design constraint' [85] and Woo and Lee's 'Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era' from 2008 [121].

Success Stories

Before we analyse a few many-core success stories, it is important to address some of the hype around GPU acceleration. As is often the case with a new technology, there have been lots of inflated claims of performance speedups, some even claiming speedups of hundreds of times compared to regular CPUs. These claims almost never stand up to serious scrutiny and usually contain at least one serious flaw. Common problems include: optimising the GPU code much more than the serial code; running on a single host core rather than all of the host cores at the same time; not using the best compiler optimisation flags; not using the vector instruction set on the CPU (SSE or AVX); using older, slower CPUs etc. But if we just compare the raw capabilities of the GPU with the CPU then typically there is an order of magnitude advantage for the GPU — not two or three orders of magnitude. If you see speedup claims of greater than about a factor of ten then be suspicious. A genuine speedup of a factor of ten is of course a significant achievement and is more than enough to make it worthwhile investing in GPU solutions.

There is one more common issue to consider, and it is to do with the size of the problem being computed. Very parallel systems such as many-core GPUs achieve their best performance when solving problems that contain a correspondingly high degree of parallelism, enough to give every core enough work so that they can operate at close to their peak efficiency for long periods. For example, when accelerating linear algebra, one may need to be processing matrices of the order of thousands of elements in each dimension to get the best performance from a GPU, whereas on a CPU one may only need matrix dimensions of the order of hundreds of elements to achieve close to their best performance (see Box 2 for more on linear algebra). This discrepancy can lead to apples-to-oranges comparisons, where a GPU system is benchmarked on one size of problem while a CPU system is benchmarked on a different size of problem. In some cases the GPU system may even need a larger problem size than is really warranted to achieve its best performance, again resulting in flawed performance claims.

With these warnings aside let us look at some genuine success stories in the areas of computational fluid dynamics for the aerospace industry, molecular docking for the pharmaceutical industry, options pricing for the financial services industry, special effects rendering for the creative industries, and data mining for large-scale electronic commerce.

Computational fluid dynamics on GPUs: OP2

Finding solutions of the Navier-Stokes equations in two and three dimensions is an important task for mathematically modelling and analysing flows in fluids, such as one might find when considering the aerodynamics of a new car or when analysing how airborne pollution is blown around the tall buildings of a modern city. Computational fluid dynamics (CFD) is the discipline of using computer-based models for solving the Navier-Stokes equations. CFD is a powerful and widely used tool which also happens to be computationally very expensive. GPU computing has thus been of great interest to the CFD community.

Prof. Mike Giles at the University of Oxford is one of the leading developers of CFD methods that use *unstructured grids* to solve challenging engineering problems. This work has led to two significant software projects: OPlus, a code designed in collaboration with Rolls-Royce to run on clusters of commodity processors utilising message passing [28], and more recently OP2, which is being designed to utilise the latest many-core architectures [42].

OP2 is an example of a very interesting class of application which aims to enable the user to work at a high level of abstraction while delivering high performance. It achieves this by providing a *framework* for implementing efficient unstructured grid applications. Developers write their code in a familiar programming language such as C, C++ or Fortran, specifying the important features of their unstructured grid CFD problem at a high level. These features include the nodes, edges and faces of the unstructured grid, flow variables, the mappings from edges to nodes, and parallel loops. From this information OP2 can automatically generate optimised code for a specific target architecture, such as a GPU using the new parallel programming languages CUDA or OpenCL, or multi-core CPUs using OpenMP and vectorisation for their SIMD instruction sets (AVX and SSE) [58]. We will cover all of these approaches to parallelism in more detail later in this report.

At the time of writing, OP2 is still a work in progress, in collaboration with Prof. Paul Kelly at Imperial College, but early results are already impressive. Using an airfoil test code with a single-precision 2D Euler equation, a problem with 0.75 million cells and 1.5 million edges has been solved in 7.4 seconds using a single NVIDIA C2070 GPU. Double-precision performance was roughly half this, taking 15.4 seconds on the same problem. When OP2 generates similarly optimised code for a pair of contemporary Intel 'Westmere' 6-core 2.67GHz X5650 CPUs, the same problem takes 34.2 seconds to solve in single precision and 44.6 seconds in double precision. One GPU has therefore delivered impressive relative speedups of 4.6X and 2.9X for single and double precision respectively, compared to 12 top-of-the-range CPU cores.

OP2 is being developed as an open source project and potential users can contact Prof. Giles via the project webpage [44].

Other CFD examples — Another successful project using GPUs for CFD has been Turbostream, developed by Tobias Brandvik and Dr Graham Pullan in the Whittle Laboratory at the University of Cambridge [18, 19]. In contrast to OP2's unstructured grid approach, Turbostream solves the Navier-Stokes equations by discretising the problem onto a structured grid and then performing explicit time-stepping to solve the resulting partial differential equations (PDEs) using a three-dimensional stencil operation.

Turbostream is another example of an approach that enables the programmer to work at a higher level of abstraction: rather than programmers having to write complex, low-level code themselves, the programmer specifies the required stencil operations in a high-level description from which Turbostream automatically generates optimised code targeting the latest GPUs and multi-core CPUs.

Turbostream has achieved impressive performance improvements, with speedups in the range of 10–20X on the latest NVIDIA GPUs compared with optimised code running on a fast quad-core Intel CPU. Additionally Turbostream has shown good scalability, using up to 64 GPUs at once. Turbostream is available to license and the reader is referred to the project website for more information [20].

Others have also been achieving great success in using GPUs to accelerate CFD problems. BAE Systems has announced it is using a GPU-accelerated cluster to deliver near-interactive speeds for their CFD simulations [37]. A team at Stanford University was recently successful in modelling hypersonic airflows using a GPU-based system [33].

These early successes indicate that CFD is an application area that should see increasing benefit from GPUs in the future; for additional information see the CFD Online website [24].

Molecular docking using GPUs: BUDE

The University of Bristol has been at the forefront of molecular docking for the past decade [41]. Molecular docking is the process of simulating the interaction between two molecules, where one molecule is typically a protein of medical interest in the human body, and the second represents a potential new drug.

The Bristol University Docking Engine (BUDE) was originally developed to run on a commodity cluster of x86-based CPUs, but as early as 2006 it was ported to one of the first generation of many-core processors, the CSX architecture from ClearSpeed [75]. In 2010

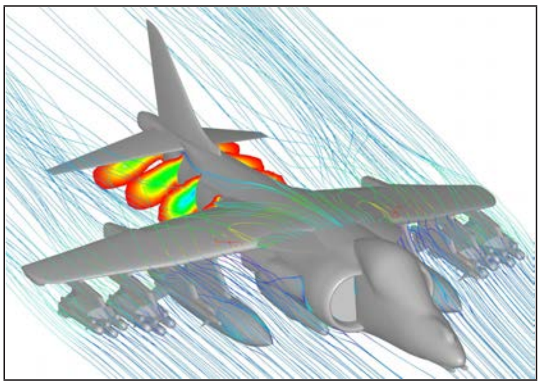


Figure 3: GPUs have enabled BAE Systems to investigate the aerodynamic performance of aircraft through advanced CFD simulations (source: BAE Systems).

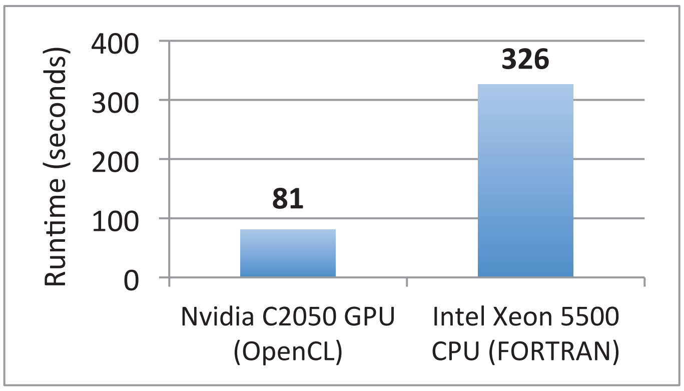


Figure 4: BUDE molecular docking performance in seconds (lower is better).

BUDE was ported to the OpenCL parallel programming language. Figure 4 compares the performance of this OpenCL code running on a GPU to the original, optimised Fortran code running on a fast, quad-core CPU. The results for both performance and energy efficiency were compelling for the GPU-based system: NVIDIA C2050 GPUs gave a real-world speedup of 4.0X versus a fast quad-core CPU while using about one half of the energy to complete the same work [76, 77].

An important benefit of using the new OpenCL parallel programming language was the ability to run exactly the same code on a range of different GPUs from different vendors, and even on multi-core x86 CPUs. Thus BUDE can now use whichever hardware is the fastest available at any given time. BUDE can also use GPUs and CPUs at the same time to deliver the maximum possible aggregate performance.

Options pricing using GPUs: NAG

One of the earliest application areas explored using GPUs was Monte-Carlo-based numerical integration for derivative pricing and risk management in financial markets. The primary need in this application is the ability to generate rapidly high-quality pseudo-random or quasi-random numbers. Fortunately parallel algorithms for generating these kinds of random numbers already exist, and several of these algorithms are well matched to the GPU's many-core architecture.

The Numerical Algorithms Group (NAG) is a provider of high-quality numerical software libraries. NAG is one of the first vendors to support GPUs – their GPU-accelerated pseudo-random number generators show speedups of between 5X and 34X when running on NVIDIA's C2050 GPU, compared to Intel's highly optimised MKL library when running on all eight cores of a contemporary Intel Core i7 860 operating at 2.8GHz [17]. The exact speedup of these routines depends on the kind of random number generator being used (MRG32k3a, Sobol or MT19937) and also the distribution required (uniform, exponential or normal).

However, in modern financial models, generating the random numbers is typically only a fraction of the overall task, with the models themselves often requiring considerable computation to evaluate. Since the Monte Carlo method is inherently parallel, the entire simulation can be performed on the GPU, where the abundance of computing power means that these complex models can be evaluated very rapidly. This capability has attracted several users from the financial services industry, including several of the more sophisticated insurance companies. This class of potential GPU user is faced with modern risk management requirements such as 'Solvency II', which require large amounts of simulation. When combined with the complex cashflow calculations inherent in insurance portfolios, this application becomes massively parallel and very compute-intensive, making it well suited to modern GPUs.

NAG has published the results it achieved while working with two of its tier-one financial services customers. One of these customers has used NAG's GPU-accelerated random number generator library to speed up its 'local vol' code and saw speedups of approximately ten times compared to a fast quad-core x86 CPU.

In summary, the generation of large sets of pseudo-random numbers for Monte Carlo methods has been one application area where GPUs have had a big impact. Speedups for the random number generation routines of 100X or more versus a single CPU core are regularly achieved, often translating into real-world performance improvements of 10X or more compared to a host machine with a total of eight cores, as in the NAG cases described above.

Have we just ignored the earlier warning about excessive speedups? Not quite — this is a rare instance where the GPU really does have an additional performance advantage over the CPU. GPUs uniquely include dedicated hardware for executing certain functions very quickly, in only a few clock cycles. These functions include exponential, sine, cosine, square root and inverse square root amongst others. Many of these functions are used extensively when generating pseudo-random numbers, and so in this instance GPUs really do have a greater performance advantage over CPUs than the peak speed comparison would indicate.

For an overview of GPUs in computational finance see [43].

Special effects rendering using GPUs: Double Negative

London-based Double Negative is the largest visual effects company in Europe [31]. Their Oscar-winning work can be seen in many of the big blockbuster movies over the last twelve years. One of the biggest industrial users of high-performance computing, visual effects companies require datacentres with thousands of servers to produce their computer-generated visuals.

Water, smoke and fire-based effects have become increasingly popular in movies over the last few years, with Double Negative's proprietary fluid solver 'Squirt' being a key part of their success. The image of the volcanic explosion on the front cover of this report was generated by Double Negative using this software. The computationally expensive process of solving the Navier-Stokes equations required by these fluid-based effects have led Double Negative to investigate the potential of GPUs. Double Negative recently added a GPU-based Poisson solver to their fluid simulations, resulting in as much as a 20X speedup of this component of their workflow. These results motivated Double Negative to install their first GPU-based render-farm.

Double Negative is now developing a specialist language and compiler to make GPUs easier to use for their visual effects artists. Results are again impressive, with a 26X speedup on an NVIDIA C2050 GPU compared to the original single-threaded C++ code running on one core of a 3.33GHz six-core Intel X5680 CPU (the speedup would be about 4.4X if all six cores of the host could be used at once with perfect scaling on the CPU) [14]. These increases in performance have motivated Double Negative to investigate using GPUs for other applications, such as image manipulation tools and deformers, in a bid to accelerate as much as possible of their visual effects workflow.

Accelerating database operations using GPUs: 3DMashUp

It may be clear that many-core architectures will shine on computationally intensive codes, but what about more data-driven applications?

Researchers are now looking at using GPU architectures to improve the performance of computationally intensive database operations. One of the first companies developing this technology is the California-based 3DMashUp. 3DMashUp has developed a solution using a PostgreSQL-based relational database that embeds GPU code alongside data within the database itself. Users can query the GPU-aware database, transparently causing the GPU code to be executed on the data, all the while remaining within the context of the database.

This approach has a number of benefits, one of which is that the data and the code can remain resident in the database at all times. The overhead of calling OpenCL GPU code from within the database is just 4 microseconds, making this a convenient and efficient way to exploit GPUs for large database users with computationally expensive processing requirements. It also brings GPU acceleration to mainstream markets. For example, users who have very large image databases may now process these images using GPUs just by integrating the appropriate 3DMashUp methods and installing the latest GPUs in their system. Users of the database do not have to care about GPU programming at all in this scenario; their tasks 'just run faster'.

The 3DMashUp solution also transparently handles mapping the most performance-critical data into the GPU's memory as and when it is needed, removing the need for users to have to concern themselves with these low-level programming details.

This approach appears to be very promising and is one of the few examples where the benefits of GPU acceleration can be invisibly provided to end users, in this case with computationally intensive, large data problems that use databases to store and manage that data. For more information see the 3DMashUp website [26].

GPUs in Depth

While several different many-core architectures have emerged during the last few years, we will focus on GPU-based many-core systems for the following discussion. However, almost all the principles and terminology discussed in this context apply equally to the other many-core architectures, including x86 CPUs and consumer electronics products as previously described.

A modern GPU is a many-core device, meaning it will contain hundreds or even thousands of simple yet fully programmable cores, all on a single chip. These cores are often grouped together into homogeneous sets with varying names. The emerging C-based many-core programming language OpenCL [67] calls these simple cores 'Processing Elements' or *PEs*, and the groupings of these PEs 'Compute Units' or *CUs*. Another common feature of all many-core architectures is a multi-level memory hierarchy. Typically each Processing Element will have a small amount of its own private memory. There is often a larger memory per Compute Unit that can be used as shared memory between all that Compute Unit's Processing Elements. There will also usually be a global memory which can be seen by all the Compute Units and thus by all the Processing Elements. Finally, there is usually a separate memory for the host processor system. OpenCL refers to these four levels in the memory hierarchy as *Private*, *Local*, *Global* and *Host*, respectively. Figure 5 illustrates the OpenCL memory hierarchy terminology. We will adopt OpenCL's terminology for the rest of this report.

The GPU itself is integrated into a 'host system'. This might mean the GPU is on its own add-in board, plugged into a standard PCI Express expansion slot within a server or desktop computer. Alternatively, the GPU may be integrated alongside the host CPU, as found inside high-end laptops and smartphones today. Increasingly in the future we will see the many-core GPU tightly integrated onto the same chip as the host CPU; in June 2011 AMD officially launched their first 'Fusion' CPU, codenamed Llano, that integrates a quad-core x86 CPU with a many-core GPU capable of running OpenCL programs [104]. NVIDIA already has consumer-level 'Fusion' devices in its Tegra CPU+GPU product line, but at SuperComputing 2010 in New Orleans they announced 'Project Denver'. This is NVIDIA's programme for a high-end Fusion-class device, integrating their cutting-edge GPUs with new, high-end ARM cores [84]. The first Project Denver products will not arrive for several years, but NVIDIA has indicated that this 'fusion' approach of integrating their GPUs alongside their own ARM-based multi-core CPUs is central to their future product roadmap.

The reader may come across three very different configurations of GPU-accelerated systems, illustrated in Figure 6. These three different ways of integrating GPUs within a system may have very different performance characteristics, but their usage models are almost identical and we shall treat them in the same way for the purposes of this report. An important implication of this wide range of different approaches is that GPU-accelerated computing is not just for supercomputers — almost all systems are capable of exploiting the performance of

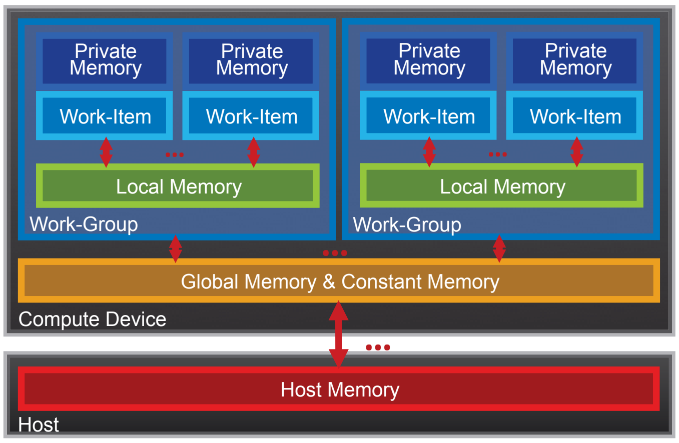


Figure 5: The memory hierarchy of a generic GPU (source: Khronos).

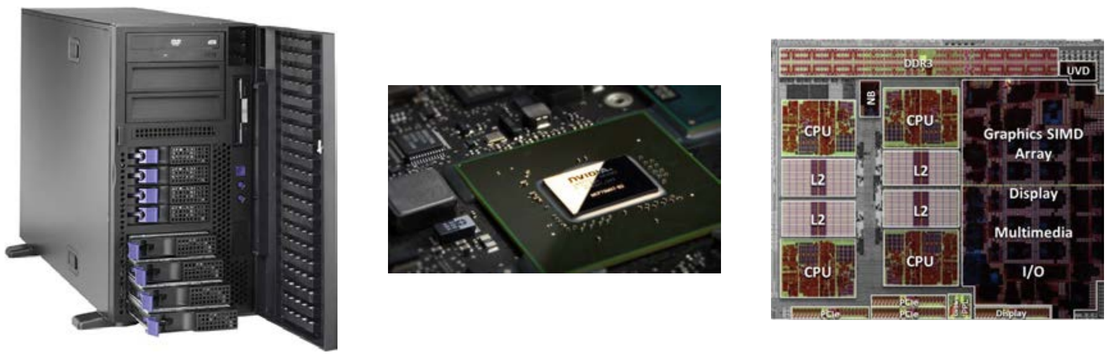


Figure 6: Three different methods of integrating GPUs: add-in graphics cards for a workstation, an integrated but discrete GPU within a laptop, and a fully integrated, single-chip CPU and GPU (source: NVIDIA, AMD).

GPUs, including laptops, tablets and even smartphones.

What are the performance benefits of the current many-core GPU systems? Just how fast are they? GPUs are still so new to the area of scientific computing that no widely used standard benchmarks exist for them. This makes it very difficult for us to compare their performance to each other, or to traditional CPUs. Before GPU-oriented benchmarks mature, we have to 'make do' with much less accurate indicators of performance, such as peak hardware capabilities (FLOPS, bandwidth etc). There are some early application performance comparisons we can use too; the 'Success Stories' section of this report gives some real-world examples.

Let us consider three different systems as points of reference: a laptop, a desktop and a server. The laptop on which this report was written is a contemporary Apple MacBook Pro and has a dual-core CPU running at 2.4 GHz. Its peak performance is around 40 GFLOPS (GigaFLOPS), measured in single-precision floating point operations per second — 1 GFLOP is one billion (10^9) floating point operations per second. Double-precision floating point performance is typically one half that of single precision performance on CPUs. The other important hardware characteristic for performance is memory bandwidth — the same laptop has a peak bandwidth to its main memory of about 8 GBytes/s. Under the author's desk is a PC containing a reasonably fast CPU: a dual-core x86 running at 3.16 GHz. This desktop's peak speed is nearly 60 GFLOPS single-precision and its peak memory bandwidth is about 12 GBytes/s. The fastest mainstream x86 server CPUs available on the market in the Spring of 2011 were Intel's six-core Westmere-EX devices which, at their fastest speeds of 2.4 GHz, are capable of peak speeds of around 230 GFLOPS per CPU for single precision, and of over 60 GBytes/s of main memory bandwidth. The Westmere figures are truly impressive, although these high-end CPUs are both expensive (with a list price per CPU of $4,616 at launch) and power-hungry (130W for the CPU alone). Even so, a single one of these high-end CPUs would have qualified as a supercomputer in its own right just 15 or 20 years ago [79].

But many-core performance, as embodied by the GPUs, is on a different level. For example, the desktop machine mentioned above also contains an NVIDIA GTX 285 GPU. This has a peak speed of over 1,000 GFLOPS (1 TeraFLOP) and peak memory bandwidth of nearly 160 GBytes/s. Hence the GPU, which only cost a few hundred pounds, has about 17 times the peak single-precision performance and over 13 times the memory bandwidth of the host CPU in the same system. Even compared to the quad-core server CPUs found in most servers today the GPU is fast, with 10 times the peak FLOPS and over 5 times the peak memory bandwidth.

So now the opportunity should be clear: many-core GPUs can potentially offer an order of magnitude greater performance in peak compute and memory bandwidth. The challenge is translating this into real-world benefits. GPU-based computing is still in its infancy, but early indications are that *for the right kinds of applications* (an important proviso), real-world speedups in the range of 5 to 10 times are being achieved using GPUs.

Gaining performance: massive parallelism

The examples we have presented demonstrate that GPUs are very fast and can provide up to a tenfold improvement in performance over traditional CPUs. The trick to getting the most from these

Box 2: Linear Algebra

Linear algebra libraries have been a staple in scientific software since the development of the Basic Linear Algebra Subprograms (BLAS) by Kincaid and Krogh [70] amongst others in the 1970's. Today the speed of routines for vector and matrix arithmetic is critically important to the performance of a wide range of applications. Not surprisingly therefore, significant effort has gone into optimising linear algebra libraries for GPUs.

Most vendors supply optimised BLAS and LAPACK (Linear Algebra PACKage) libraries for their hardware. CPU vendor libraries include Intel's MKL [59], AMD's ACML [7] and IBM's ESSL [55]. GPU vendor libraries include NVIDIA's CuBLAS [93] and AMD's GPU ACML [8]. These libraries speed up the BLAS routines by optimising the original BLAS code for the targetted hardware. This approach has been reasonably successful up to now, but there are questions about its long-term scalability. The author has first-hand experience in this area, having led one of the first teams to develop a many-core optimised BLAS/LAPACK library in 2003 while at ClearSpeed. The concern is around this approach to programming, and whether it is realistic to be able to write a serial program that calls a software library that 'magically' hides all the complexity of using massive parallelism in an optimal fashion. Experience has shown that, to get the best performance, one usually has to port most of the application to the massively parallel part of the system, and not just rely on calls to libraries with parallel implementations.

To address this issue, the PLASMA and MAGMA research projects have been looking at the implications of heterogeneous and many-core architectures for future BLAS and LAPACK libraries [5]. These projects have shown that it is possible to achieve a significant fraction of peak performance for LAPACK-level routines, such as Cholesky and LU solvers, on heterogeneous, multi- and many-core systems, going as far as using multiple GPUs in a single accelerated system to achieve more than 1.1 TeraFLOPS of single precision performance [71].

many-core architectures is *parallelism*, and lots of it. The GPU hardware itself is massively parallel, with today's GPUs already including hundreds or thousands of simple cores, and GPUs with tens of thousands of cores due to appear within the next few years. The most popular programming models for GPUs, NVIDIA's CUDA [96] and the cross-platform open standard, OpenCL [67], encourage programmers to expose the maximum parallelism possible in their code in order to achieve the best performance, both now and in the future. In general these are just good software design practices and we encourage all software developers to start thinking this way if they have not already — within a few years almost all processors will be of a many-core design and will require massively parallel software for best performance.

This requirement for massive parallelism may at first appear intimidating. But fortunately the kind of parallelism required does not have to be coarse-grained 'task' parallelism; it can simply be fine-grained 'data' parallelism. This is usually much easier to find and often corresponds to traditional vector processing with which many software developers are already familiar. Computationally intensive applications almost always have a large degree of data parallelism inherent in the underlying problem being solved, ranging from the particle parallelism in classical N-body problems to the row and column parallelism in matrix-based linear algebra. But it remains a fundamental point: *to get the best performance from many-core architectures you will want to expose the maximum parallelism possible in your application*.

This is a very important point and is worth restating: we will go so far as to make the potentially controversial statement that *most* existing software will need to be completely redeveloped, possibly even requiring new algorithms, in order to scale well on the increasingly parallel hardware we will all be using from now on.

There is an equally important implication that we will also state explicitly: all computer hardware is likely to remain on the increasingly parallel trend for at least the next ten to twenty years. Hardware designers predict that processors will become an order of magnitude more parallel every five years, meaning that in fifteen years' time, processors will be of the order of a thousand times more parallel than the already very parallel processors that exist today!

We will come back to what this means for all of us as users or developers of software in the 'Next Steps' section later in this report.

GPU parallel programming models

With the rapidly increasing popularity of GPU-based computing, multiple different programming models have emerged for these new platforms. These include vendor-specific examples, such as NVIDIA's CUDA, The Portland Group's PGI Accelerator^{TM} [113], CAPS enterprise's HMPP Workbench [23] and Microsoft's DirectCompute [81], as well as open standards such as OpenCL. We shall now look at the two most widely used GPU programming models in more detail: CUDA and OpenCL.

CUDA

NVIDIA's CUDA is currently the most mature and easiest to use parallel programming model for GPUs, first appearing on the market in 2006 [91]. CUDA was designed to be familiar to as many software developers as possible, and so its first instantiation began as a variant of the C programming language, called CUDA C. Indeed, in developing CUDA C code it is often possible to incrementally port an existing C application, using *profiling* to identify those parts of the code that consume most of the run-time (the 'hot spots') and converting these to run in parallel on the many-core GPU.

This 'incremental porting' approach can be very effective and certainly lowers the barrier to entry for many users. But in our experience, most software developers find that they can only get so far with this approach. The reason for this limitation usually comes back to the design of the underpinning algorithm. If it was not designed to be massively parallel, then it is likely that the current software implementation cannot easily be modified in an incremental way in order to make it massively parallel. Of course there are always exceptions, but often the incremental porting approach will only take you so far.

Coming back to CUDA C, to demonstrate how familiar it can be, consider the two code examples in Figure 7. On the left is some ordinary ANSI C code for performing a simple vector addition. Because it is written in a serial programming language, this code uses a loop and an array index to run along the two input vectors, producing one output element at each loop iteration.

In the CUDA C example on the right, you can see the loop has completely disappeared. Instead, many copies of the function are executed in parallel, one copy for each parallel data item we wish to process. In this instance we would launch 'size' copies of the vector addition function, all of which could run in parallel. At execution time the run-time system decides how to map efficiently the expressed parallelism onto the available parallel hardware resources.

It is important not to underestimate the level of challenge when writing code for GPUs. The trivial examples in Figure 7 are very small in order to illustrate a point. In a real example, modifying the algorithm in order to express enough parallelism to achieve good performance on a GPU often requires significant intellectual effort. And while the parallel kernels themselves may often look quite similar to a highly optimised sequential equivalent, significant additional code is required in order to initialise the parallel environment, orchestrate the GPUs in the system, and communicate data efficiently, often by overlapping the data movement with computation.

The CUDA C code on the right hand side of Figure 7 is an example of a 'kernel' — this is common terminology for the parallel version of a routine that has been modified in order to run on a many-core processor. CUDA uses slightly different terminology for the

```
// ANSI C vector addition example
void vectorAdd(const float *a,
               const float *b,
               float *c,
               unsigned int size)
{
    unsigned int i;
    for (i=0; i<size; i++)
        c[i] = a[i] + b[i];
}
```

```
// CUDA C vector addition example
__global__ void vectorAdd(const float *a,
                          const float *b,
                          float *c)
{
    // Vector element index
    int nIndex =
         blockIdx.x * blockDim.x + threadIdx.x;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

Figure 7: Simple vector addition examples in C and CUDA C – note the absence of the 'for' loop in the CUDA version.

Processing Elements (which it calls ‘threads’) and Compute Units (which it calls ‘thread blocks’) but generally it is considered to be fairly straightforward to port code between CUDA C and similar programming models, such as OpenCL.

CUDA threads execute independently and thus ideally on independent data — this is why data parallelism is such a natural fit for these kinds of architectures. Threads in a thread block are grouped to execute essentially the same program at the same time, but on different pieces of data. This data-parallel approach is known as Single Instruction Multiple Data (SIMD) [38]. On the other hand, different thread blocks may execute completely different programs from one another if the need arises, although most applications tend to run the same program on all the thread blocks at the same time, essentially turning the computation into one large SIMD or vector calculation.

CUDA’s maturity brings a number of benefits for software developers, including a growing number of software development tools including debuggers and profilers [97]. In 2009, CUDA Fortran arrived, as a joint development between NVIDIA and the Portland Group [95]. CUDA Fortran takes the principles of CUDA C and weaves them into a state-of-the-art commercial Fortran 2003 compiler.

OpenCL

OpenCL bears many similarities to CUDA and indeed NVIDIA is one of the main contributors to the OpenCL standard and so this should be no surprise. The biggest differences are in the way OpenCL is being developed. Whereas CUDA is a proprietary solution being driven by a single vendor, OpenCL is an open standard, instigated by Apple, but now being driven by a consortium of over 35 companies, including all the major processor vendors such as Intel, IBM and AMD. The consortium is being organised and run by the Khronos Group [67].

OpenCL is a much more recent development than CUDA and is correspondingly less mature. However, OpenCL also includes a number of more recent advances for supporting heterogeneous computing in systems combining multiple CPUs and GPUs. The first version of the OpenCL standard was released in December 2008 [66], and OpenCL has been developing rapidly since then. It has already been integrated into recent versions of Apple’s OS X operating system. AMD and NVIDIA have released implementations for their GPUs, the former also including a version that will run on a multi-core x86 host CPU. IBM has demonstrated a version of OpenCL running on its Cell processor and recently released a version for their high-end POWER architecture [56]. Intel released its first OpenCL implementation for its multi-core x86 CPUs in late 2010 [27, 60]. Embedded processor companies are also developing their own OpenCL solutions, including ARM [11, 99], Imagination Technologies [57] and Zii Labs [122]. These last three companies provide the CPUs and GPUs in most of the popular consumer electronics gadgets, such as smartphones and portable MP3 players.

While OpenCL is less mature than NVIDIA’s CUDA and has some of the drawbacks of committee designed standards, its benefits are the openness of the standard, the vast resource being ploughed into its development by many companies, and most importantly, its cross-platform capabilities.

OpenCL is quite a low-level solution. It exposes features that many software developers may not have had to deal with before. CUDA has similar features but includes a higher-level application programmer interface (API) that conveniently handles much of the low-level detail. But in OpenCL this is all left up to the programmer. One example is in the explicit use of queues for sending commands such as ‘run this kernel’ from the host processor to the many-core GPU. It is expected that as OpenCL matures, various solutions will emerge to abstract away this lower-level detail, leaving most programmers to operate at a higher level. An interface has already been developed that provides this facility for C++ programs.

One of OpenCL’s other important characteristics is that it has been designed to support heterogeneous computing from Day One; that is, it supports running code simultaneously on multiple, different kinds of processors, all within a single OpenCL program. When re-engineering software this is an important consideration: adopting a programming environment that supports a wide range of heterogeneous parallel hardware will give developers the greatest flexibility when deploying their re-engineered codes in the future.

For example, an OpenCL program could decide to run one task on one of the host processor cores, while running another task using a many-core GPU, and do this all in parallel. These multiple OpenCL tasks can easily coordinate between themselves, passing data and signals from one to the other. Because almost all processors will

```
// OpenCL vector addition example
__kernel void vectorAdd(__global const float *a,
                        __global const float *b, __global float *c)
{
     // Vector element index
    int nIndex = get_global_id(0);
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

Figure 8: OpenCL vector addition kernel example. Note the similarity with the CUDA example on the right hand side of Figure 7.

combine elements of both heterogeneity and many-core in the future, this is a critical feature to support. The ability to run OpenCL code on almost any multi-core mainstream processor makes it extremely attractive as a method for exploiting many-core parallelism, be it on today's increasingly multi-core CPUs or tomorrow's heterogeneous many-core processors.

For completeness, Figure 8 shows the simple vector addition example once again, this time in OpenCL.

To learn more about OpenCL, Scarpino's 'A gentle introduction to OpenCL' [105] is a good place to start. Two OpenCL text books are due to appear at about the same time as this report: Munshi et al's 'OpenCL Programming Guide' [86] and Gaster et al's 'Heterogeneous Computing with OpenCL' [40].

Other many-core programming models

While we have focused our descriptions on the many-core programming models that are emerging to support GPUs, there exist other models that were created primarily to support multi-processor systems. There are many such models, but the most commonly used fall mainly into two categories — *message passing* and *shared memory*.

Message passing programming models provide methods for running collections of tasks in parallel, and for communicating data between these tasks by sending messages over communication links.

Shared memory programming models assume a system of parallel processors connected together via a memory that can be seen by all of the processors.

Both of these models are used in the mainstream today and we shall briefly describe the most common examples of each.

Message passing, MPI — MPI 'is a message-passing application programmer interface (API), together with protocol and semantic specifications for how its features must behave in any implementation' [49]. MPI has been proven to scale to large high-performance computing systems consisting of hundreds of thousands of cores and underpins most parallel applications running on large-scale systems. It is most commonly used to communicate between tasks running on multiple servers, with the MPI protocol used to send messages over the network connecting the servers together.

As well as providing basic primitives for point-to-point communications, MPI includes collective communication operations such as broadcast and reduction. The more recent MPI version 2 (MPI-2) adds support for parallel I/O and dynamic process management [112].

Many implementations of MPI exist, with the standard specifying bindings for C, C++ and Fortran.

Shared memory, OpenMP — OpenMP is the most commonly used high-level shared-memory programming model in scientific computing [25]. Its API is implemented in the form of compiler directives for C, C++ and Fortran. Most major hardware vendors support it, as do most major compilers, including GNU gcc, Intel's ICC and Microsoft's Visual Studio.

OpenMP supports task-level parallelism (also known as Multiple Instruction Multiple Data or MIMD) in addition to the data-level parallelism (SIMD) that we have already met [38]. Sections of code that can be executed in parallel are identified with the addition of special compiler markers, or *pragmas*, that tell an OpenMP-compatible compiler what to do. In theory these pragmas are simply ignored by non-OpenMP compilers, which would generate valid serial executables from the same source code.

For comparison, Figure 9 shows our by now familiar simple vector

```
!$omp parallel default(none)        &
!$omp shared(a, b, c) private(i)
!$omp do
      do i = 1, size
          c(i) = a(i) + b(i)
      end do
!$omp end do
!$omp end parallel
```

Figure 9: Vector addition example in Fortran extended with OpenMP.

addition, this time in OpenMP, and for variety, in Fortran (the OpenMP Fortran pragmas are the lines beginning with '`!$omp`').

Work is already under way developing the next major version of OpenMP [109] and early signs indicate that consideration is being given to extensions for supporting many-core architectures. Thus if the reader is already using OpenMP we would recommend following the developments in this area from the standards committee and also active OpenMP vendors such as Cray.

Hybrid solutions — It is possible to combine two or more of these parallel programming models for hybrid solutions. In the parallel programming mainstream today, one of the most common approaches is to use a hybrid system of OpenMP within each multi-core server node and MPI between nodes. Much has been written about this particular hybrid approach [109].

An increasingly common approach is to combine MPI with one of the new GPU-oriented parallel programming systems, such as CUDA or OpenCL. This enables the creation of systems from multiple nodes, each node including one or more GPUs. Several of the fastest computers in the world have recently been constructed in just this fashion, including TSUBAME 2.0 at Tokyo Tech in Japan [50]. Indeed the second fastest computer in the world in June 2011 was a GPU-powered cluster: China's Tianhe-1A system at the National Supercomputer Center in Tianjin, with a performance of 2.57 PetaFLOPS (2.57×10^{15} floating point operations per second, or 2.57 million GFLOPS) [16, 114]. As of the International Supercomputing Conference in June 2011, three of the top ten supercomputers in the Top500 list are GPU accelerated. The fraction of systems in the Top500 that achieve their performance by GPU acceleration is set to increase rapidly.

Current Challenges

Like all breakthroughs in technology, the change from multi-core to many-core computer architectures will not be smooth for everyone. There are significant challenges during the transition, some of which are outlined below.

1. Porting code to massively parallel heterogeneous systems is often (but not always) harder than ports to new hardware have been in the past. Often completely new algorithms are required.
2. Many-core technologies are still relatively new, with implications for the maturity of software tools, the lack of software developers with the right skills and experience, and the paucity of ported application software and libraries.
3. Even though cross-platform programming languages such as OpenCL are now emerging, these have so far focused on source code portability and cannot guarantee performance portability. This is of course not a new issue; any highly optimised code written in a mainstream language such as C, C++ or Fortran has performance portability issues between different architectures. However, differences between GPU architectures are even greater than those between CPU architectures, and so performance portability is set to become a greater challenge in the future.
4. There are multiple competing open and de facto standards which inevitably confuse the situation for potential adopters.
5. Many current GPGPU products still carry some of their consumer graphics heritage, including the lack of important hardware reliability features such as Error Correcting Codes (ECC) on their memories. Even where these features do exist, they currently incur prohibitive performance penalties that are not present in the corresponding CPU solutions.
6. There is a lot of hype around GPU computing, with many over-inflated claims of performance speedups of 100 times or more. These claims increase the risk of setting expectations too high, with subsequent disappointment from trial projects.
7. The lack of industry standard benchmarks makes it difficult for users to compare competing many-core products simply and accurately.

Of all these challenges, the most fundamental is the design and development of new algorithms that will naturally lend themselves to the massive parallelism of GPUs today, and to the ubiquitous heterogeneous multi-/many-core systems of tomorrow. If as a community we can design adaptive, highly scalable algorithms, ideally with heterogeneity and even fault-tolerance in mind, we will be well placed to exploit the rapid development of parallel architectures over the next two decades.

Next Steps

Audit your software

One valuable practical step we can each take is to perform an audit of the software we use that is performance-critical to our work. For software developed by third parties, find out what their policy is towards supporting many-core processors such as GPUs. Is their software already parallel? If so, how scalable is it? Does it run effectively on a quad-core CPU today? What about the emerging 8, 12 and 16 core CPUs? Do they have a demonstration of their software accelerated on a GPU? What is their roadmap for the software? The software licensing model is something that you also need to be conscious of — is the software licensed per core, processor, node, user, . . . ? Will you have to pay more for an upgrade that supports many-core processors? Some vendors will supply these features as no-cost upgrades; others will charge extra for them.

It is also important to be specific about parallel acceleration of the particular features you use in the software in question. For example, at the time of writing there are GPU accelerated versions of dense linear algebra solvers in MATLAB, but not of sparse linear algebra solvers [73]. Just because an application claims to be 'GPU-accelerated', it does not necessarily follow that your particular use of that application will gain the performance benefit of GPUs. Your mileage will definitely vary, so check with the supplier of your software to verify before committing.

Plan for parallelism

If you develop your own software, start thinking about what your own path to parallelism should be. Are the users of your software likely to stick primarily to multi-core processors in mainstream laptops, desktops and servers? If so you should be thinking about adopting OpenMP, MPI or another widely supported approach for parallel programming. You should probably avoid proprietary approaches that may not support all platforms or be commercially viable in the long term. Instead, use open standards available across multiple platforms and vendors to minimise your risk.

Also consider when many-core processors will feature in your roadmap. These are inevitable — even the mainstream CPUs will rapidly become heterogeneous many-cores, so this really is a 'when' not an 'if'. If you do not want to support many-core processors in the near or medium term, OpenMP and MPI will be good choices. If, however, you may want to support many-core processors within the next few years, you will need a plan to adopt either OpenCL or CUDA sooner rather than later. OpenCL might be a viable alternative to OpenMP on multi-core CPUs in the short term.

If you are going to develop your own many-core aware software there is a tremendous amount of support that you can tap into.

Attend a workshop

In the UK each year there are several training workshops in the use of GPUs. The national supercomputing service HECToR [53] is starting to provide GPU workshops on CUDA and OpenCL programming; the timetable for these is available online [52]. Prof Mike Giles at the University of Oxford regularly runs CUDA programming workshops; for the date of the next one see [43]. His webpage also includes excellent links to other GPU programming resources. A search for GPU training in the UK should turn up offerings from several universities. There are also commercial training providers in the UK that are worth considering, such as NAG [89]. Daresbury Labs has a team who track the latest processor technologies and who are already experienced in developing software for GPUs. This group holds occasional seminars and workshops, and is willing to offer advice and guidance for newcomers to many-core technologies [30]. GPU vendors will often provide assistance if asked, especially if their assistance could lead to new sales.

There are also many conferences and seminars emerging to address many-core computing. The UK now has a regular GPU developers workshop. Previous years have seen the workshop held in Oxford [1] and Cambridge [2]. The 2011 workshop is due to be held at Imperial College.

A useful GPU computing resource is GPUcomputing.net [47]. In particular it has a page dedicated to GPU computing in the UK [45]. This site is mostly dominated by the use of NVIDIA GPUs, reflecting NVIDIA's lead in the market, but over time the site should see a greater percentage of content coming from work on a wider range of platforms.

To get started with OpenCL, one good place to start is AMD's

OpenCL webpage [9]. NVIDIA's OpenCL webpage [98] is also useful, as is their OpenCL 'jump start' guide [92].

We would also recommend reading some of the latest work in the area of parallel algorithms. A good place to start is 'The view from Berkeley' [12], which builds on earlier work by Per Brinch Hansen [21]. Tim Mattson's 'Patterns for Parallel Programming' is another notable book in this area [74]. These works recognise that most forms of scientific computation can be decomposed into a small set of common kernels often known as 'dwarfs', 'motifs' or 'templates'. Decomposing algorithms into sub-algorithms that may in turn be classified as a particular kind of dwarf makes it easier to exploit the substantial works of the past and present, thus simplifying the task of identifying parallel approaches for your own algorithms where they are already known.

Finally, there is an online list of applications that have already been ported to NVIDIA's GPUs [94].

Appendix 1: Active Researchers and Practitioner Groups

This is a rapidly growing field and there are already many active researchers and developers using GPU computing in the UK and around the world. The GPU developer conference held at the University of Cambridge in December 2010 saw around 100 attendees, and the Many-Core and Reconfigurable Supercomputing Conference at Bristol in April 2011 saw over 130 delegates, all actively working within this area.

In the UK

The list below is not intended to be exhaustive but instead to give a cross section of the GPU computing community in the UK and beyond. Our apologies are offered in advance to anyone we left off this list.

University of Bristol: McIntosh-Smith is a computer architect with extensive industrial experience designing many-core heterogeneous architectures. He has more recently been developing software and algorithms for GPUs, and has been working on new, massively parallel algorithms for a range of different applications including molecular docking, computational chemistry, climate modelling, and linear algebra [78].

University of Oxford: Giles is the leading expert in the UK in the use of many-core GPUs for financial applications and also for unstructured grid problems. Giles runs regular GPU programming workshops in NVIDIA's proprietary CUDA programming language [43]. Also at Oxford, Karastergiou has been investigating the use of GPUs for the real-time processing of data for the future international telescope project, the Square Kilometre Array (SKA) [64].

University of Warwick: Jarvis's group has performed extensive work modelling the performance of large-scale HPC systems, more recently looking at the effects of adding GPUs into these systems. The group has a particular focus on wavefront codes [62].

Imperial College: Kelly is an expert in high-level tools for automatically generating optimised code for many-core architectures. His research interests include parallelising compilers and auto-tuning techniques [65].

UCL: Atkinson is one of the leaders in applying GPUs to medical imaging problems, in particular Magnetic Resonance Imaging (MRI) [13].

Edinburgh Parallel Computing Centre: EPCC is one of the leading providers of high-performance computing training in the UK [34], and can provide consultancy to help parallelise algorithms and software for parallel computer architectures.

University of Manchester: Manchester has a vibrant community of GPU users and runs a 'GPU club' that meets on a semi-regular basis, attracting around one hundred attendees to recent events [72].

Queen's University Belfast: Gillan and Scott lead a group of experts in using many-core architectures for image processing applications [46, 108]. This group provides expertise and consultancy to third parties who need to adapt their codes for GPUs and for reconfigurable systems based on Field-Programmable Gate Arrays (FPGAs).

University of Cambridge: Pullan and Brandvik were two of the earliest adopters of many-core architectures in the UK. They are experts in using GPUs for CFD and in porting structured grid codes to many-core architectures [20]. Also at Cambridge, Gratton has been investigating the use of AMD and NVIDIA GPUs for accelerating cosmology codes [48]. All three are part of Cambridge's Many-Core Computing Group which also has expertise in the use of GPUs for medical imaging and genomic sequence processing [22].

Daresbury Laboratories: the Distributed Computing (DisCo) group at Daresbury Labs regularly runs workshops in the use of many-core architectures and can provide support in porting codes to GPUs. They also have access to various GPU hardware platforms that they can make available for remote testing [30].

AWE: Turland leads a group using GPUs to accelerate industrial wavefront simulation codes. His group has been developing OpenCL-based methods that abstract away from specific architectures and vendor programming models, allowing them to develop many-core applications that can perform well on a wide range of different hardware [15].

BAE Systems: Appa was one of the earliest adopters of many-core architectures of GPUs in the world. He now leads the team within the mathematical modelling group at BAE Systems that is using GPUs to accelerate CFD computations to great effect [37].

The Numerical Algorithms Group: NAG has a team developing commercial many-core software libraries for linear algebra and random number generators, amongst others. NAG also provides commercial many-core training courses and software development consultancy services [89].

The MathWorks: The MATLAB developer is doing a lot of work to target GPUs from within their software, much of which is taking place at their Cambridge development office [73].

Allinea: Allinea is a UK-based company developing some of the leading multi-core and many-core software development tools, including debuggers and profilers for GPUs [6]. Developers of large-scale software using hundreds or thousands of CPUs and GPUs could benefit significantly from these kinds of parallel debuggers.

Internationally

Innovative Computing Lab, University of Tennessee: Dongarra is a world expert in linear algebra libraries. His group is spearheading the development of highly scalable, dense linear algebra libraries for heterogeneous many-core processors and massively parallel systems [35, 36].

University of Illinois, Urbana-Champaign: Hwu is another leading global figure in GPU-based computing. His particular interest is in using GPUs to accelerate medical imaging applications [54]. UIUC is also one of the leading academic institutions in the use of GPUs to accelerate molecular dynamics and visualisation codes, such as NAMD and VMD [115].

Oak Ridge National Lab: Vetter's Future Technologies group at ORNL is planning one of the largest supercomputers in the world based on GPU technology [116]. Vetter's group has also developed one of the first GPU benchmarks, the SHOC suite, which is useful for comparing the performance of different many-core architectures [29, 117].

Tokyo Institute of Technology: Matsuoka is one of the leading HPC experts in Japan and is a specialist in building highly energy-efficient supercomputers based on GPU technologies. He is the architect of the TSUBAME 2.0 GPU-based supercomputer, the fifth fastest in the world as of June 2011. TSUBAME 2.0 is also one of the most energy-efficient production systems in the world [50]. Matsuoka's group also develops GPU software, and has developed one of the fastest FFT libraries for GPUs [90].

Stanford: Pande's group developed one of the first GPU-accelerated applications, the Folding@Home screensaver-based protein folding project [100]. This group went on to develop the OpenMM open source library for molecular mechanics. OpenMM takes advantage of GPUs via CUDA and OpenCL and is a good starting point for porting molecular dynamics codes to GPUs [39, 101].

NVIDIA: NVIDIA was the first GPU company to see the real potential of GPU-based computing. Introducing the CUDA parallel programming language in 2006, NVIDIA has the most applications ported to their many-core architecture. Their ported applications webpage is a good place to find out what software is available for GPUs [94].

The Portland Group: PGI provides software development tools including a parallel Fortran compiler that generates code for GPUs [103].

Acceleware: Acceleware provides multi-core and GPU-accelerated software solutions for the electromagnetics and oil and gas industries [4].

SciComp: SciComp is a financial software services company that provides a suite of software tools that exploit GPUs and multi-core CPUs to deliver significant performance advantages over traditional approaches [106].

Appendix 2: Software Applications Available on GPUs

The table below lists several important applications along with their current multi-core and GPU capabilities. These will be changing all the time so do check for the latest information when considering any application. This list is far from exhaustive; readers are encouraged to investigate further if their favourite codes are not included.

Software	**Multi-Core**	**Many-Core**
MATLAB	Some built-in ability to exploit parallelism at the level of libraries such as BLAS and LAPACK. Additionally provides a ‘Parallel computing toolbox’ to exploit multi-core CPUs [111].	Has a beta release of a GPU-accelerated version of MATLAB [73]. Various third-party solutions available, such as AccelerEyes’ Jacket [3].
Mathematica	Built-in support for parallelism since *Mathematica* 7 [119]; also provides grid*Mathematica* for large-scale parallel computation [118].	*Mathematica* 8 now includes support for GPUs via CUDA and OpenCL [120].
NAG	NAG has a software library product for shared memory and multi-core parallel systems [87].	Has a beta release of random number generators using CUDA on NVIDIA GPUs [88].
R	Multiple R packages are available for exploiting parallelism on multi-core systems [32].	Numerous open source projects porting R to GPUs, including R+GPU available in the gputools R package [80].
SciFinance	This code synthesis tool for building derivatives pricing and risk models can already generate multi-core code using OpenMP [107].	Already supports the generation of CUDA code for PDEs and SDEs [107].
BLAS, LAPACK math libraries	Almost all BLAS and LAPACK libraries already support multi-core processors: Intel’s MKL, AMD’s ACML, IBM’s ESSL, ATLAS, ScaLAPACK, PLASMA.	This is an area where much software is already available for GPUs: NVIDIA’s CUBLAS, AMD’s ACML-GPU, Acceleware, EM Photonics’ Cula Tools and MAGMA.

References

[1] 1st CUDA Developers' Conference. www.smithinst.ac.uk/Events/CUDA2009, December 2009.

[2] 2nd UK GPU Computing Conference. www.many-core.group.cam.ac.uk/ukgpucc2, December 2010.

[3] AccelerEyes — MATLAB GPU computing. www.accelereyes.com, April 2011.

[4] Acceleware Ltd — Multi-Core Software Solutions. www.acceleware.com, July 2011.

[5] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[6] Allinea Software. allinea.com, May 2011.

[7] AMD Corp — AMD Core Math Library (ACML). developer.amd.com/cpu/Libraries/acml/Pages, April 2011.

[8] AMD Corp — AMD Core Math Library for graphic processors (ACML-GPU). developer.amd.com/gpu/acmlgpu/pages, April 2011.

[9] AMD Corp — AMD Developer Central: getting started with OpenCL. developer.amd.com/zones/OpenCLZone/Pages/gettingstarted.aspx, April 2011.

[10] AMD Corp — AMD Developer Central: Magny-Cours zone. developer.amd.com/zones/magny-cours/pages/default.aspx, April 2011.

[11] ARM Ltd — ARM heralds new era in embedded graphics with next-generation Mali GPU. www.arm.com/about/newsroom/arm-heralds-new-era-in-embedded-graphics-with-next-generation-mali-gpu.php, April 2010.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[13] David Atkinson. www.cs.ucl.ac.uk/staff/d.atkinson, July 2011.

[14] Dan Bailey. Fast GPU preconditioning for fluid simulations in film production (GTC 2010 presentation archive). www.nvidia.com/object/gtc2010-presentation-archive.html#session2239, April 2011.

[15] Dave Barrett, Ron Bell, Claire Hepwood, Gavin Pringle, David Turland, and Paul Graham. An investigation of different programming models and their performance on innovative architectures. In *The Many-Core and Reconfigurable Supercomputing Conference (MRSC)*, 2011.

[16] BBC — China's Tianhe-1A crowned supercomputer king. www.bbc.co.uk/news/technology-11766840, November 2010.

[17] Thomas Bradley, Jacques du Toit, Robert Tong, Mike Giles, and Paul Woodhams. Parallelization techniques for random number generations. In Wen mei W. Hwu, editor, *GPU Computing Gems Emerald Edition*, chapter 16, pages 231–246. Morgan Kaufmann, 2011.

[18] Tobias Brandvik and Graham Pullan. SBLOCK: a framework for efficient stencil-based PDE solvers on multi-core platforms. In *10th IEEE International Conference on Computer and Information Technology*, pages 1181–1188, 2010.

[19] Tobias Brandvik and Graham Pullan. An accelerated 3D Navier-Stokes solver for flows in turbomachines. *Journal of Turbomachinery*, 133(2):021025, 2011.

[20] Tobias Brandvik and Graham Pullan. Turbostream: ultra-fast CFD for turbomachines. www.turbostream-cfd.com, April 2011.

[21] Per Brinch Hansen. Model programs for computational science: a programming methodology for multicomputers. *Concurrency — Practice and Experience*, 5(5):407–423, August 1993.

[22] University of Cambridge — Cambridge many-core group projects. www.many-core.group.cam.ac.uk/projects, May 2011.

[23] CAPS entreprise — HMPP Workbench. www.caps-entreprise.com/hmpp.html, August 2011.

[24] CFD Online. www.cfd-online.com, April 2011.

[25] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[26] Tim Childs. 3DMashUp. www.scribd.com/3dmashup, April 2011.

[27] Peter Clarke. Intel releases alpha OpenCL SDK for Core. www.eetimes.com/electronics-news/4210812/Intel-releases-alpha-OpenCL-SDK-for-Core, November 2010.

[28] P. I. Crumpton and M. B. Giles. Multigrid aircraft computations using the OPlus parallel library. In A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, editors, *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, pages 339–346. North-Holland, 1996.

[29] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[30] DisCo: Daresbury Laboratory distributed computing group. www.cse.scitech.ac.uk/disco, September 2010.

[31] Double Negative Visual Effects. www.dneg.com, April 2011.

[32] Dirk Eddelbuettel. CRAN task view: high-performance and parallel computing with R. cran.r-project.org/web/views/HighPerformanceComputing.html, February 2011.

[33] Erich Elsen, Patrick LeGresley, and Eric Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227:10148–10161, December 2008.

[34] EPCC. www.epcc.ed.ac.uk, May 2011.

[35] Jack Dongarra et al. PLASMA. icl.cs.utk.edu/plasma/index.html, November 2010.

[36] Jack Dongarra et al. MAGMA. icl.cs.utk.edu/magma/index.html, April 2011.

[37] Mark Fletcher. BAE Systems packs massive power thanks to GPU coprocessing. mcad-online.com/en/bae-systems-packs-massive-power-thanks-to-gpu-co-processing.html?cmp_id=7&news_id=1078, May 2010.

[38] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, September 1972.

[39] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6):864–872, 2009.

[40] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2011.

[41] Nick Gibbs, Anthony R. Clarke, and Richard B. Sessions. Ab initio protein structure prediction using physicochemical potentials and a simplified off-lattice model. *Proteins: Structure, Function, and Bioinformatics*, 43(2):186–202, 2001.

[42] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. Performance analysis and optimisation of the OP2 framework on many-core architectures. *The Computer Journal*, 2011 (to appear).

[43] Mike Giles. Course on CUDA programming. people.maths.ox.ac.uk/gilesm/cuda, August 2011.

[44] Mike Giles. OP2 project page. people.maths.ox.ac.uk/gilesm/op2, April 2011.

[45] Mike Giles and Simon McIntosh-Smith. GPUcomputing.net: GPU Computing UK. www.gpucomputing.net/?q=node/160, April 2011.

[46] Charles J. Gillan. www.ecit.qub.ac.uk/Card/?name=c.gillan, May 2011.

[47] GPUcomputing.net. www.gpucomputing.net, April 2011.

[48] Steve Gratton. Graphics card computing for cosmology. www.ast.cam.ac.uk/~stg20/gpgpu/index.html, July 2011.

[49] William Gropp. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, second edition, 2000.

[50] GSIC, Tokyo Institute of Technology — TSUBAME2. www.gsic.titech.ac.jp/en/tsubame2, April 2011.

[51] Mark Harris. General-purpose computation on graphics hardware. gpgpu.org/about, July 2011.

[52] HECToR training. www.hector.ac.uk/cse/training, April 2011.

[53] HECToR: UK National Supercomputing Service. www.hector.ac.uk, April 2011.

[54] Wen-mei W. Hwu. The IMPACT research group at UIUC. impact.crhc.illinois.edu/people/current/hwu/hwu_research.php, May 2011.

[55] IBM — Engineering and scientific subroutine library (ESSL) and parallel ESSL. www-03.ibm.com/systems/software/essl, April 2011.

[56] IBM — OpenCL development kit for Linux on Power. www.alphaworks.ibm.com/tech/opencl, April 2011.

[57] Imagination Technologies — Imagination Technologies announces POWERVR SGX545 graphics IP core with full DirectX 10.1, OpenGL 3.2 and OpenCL 1.0 capabilities. www.imgtec.com/news/Release/index.asp?NewsID=516, January 2010.

[58] Intel Corp — Intel AVX: new frontiers in performance improvements and energy efficiency. software.intel.com/en-us/articles/intel-avx-new-frontiers-in-performance-improvements-and-energy-efficiency, April 2011.

[59] Intel Corp — Intel math kernel library. software.intel.com/en-us/articles/intel-mkl, April 2011.

[60] Intel Corp — Intel OpenCL SDK. software.intel.com/en-us/articles/intel-opencl-sdk, April 2011.

[61] Intel Corp — Intel Research: single-chip cloud computer. techresearch.intel.com/ProjectDetails.aspx?Id=1, April 2011.

[62] Stephen Jarvis. High Performance Computing group, University of Warwick. www2.warwick.ac.uk/fac/sci/dcs/research/pcav/areas/hpc, May 2011.

[63] Jon Peddie Research — Jon Peddie Research announces 2nd quarter PC graphics shipments. jonpeddie.com/press-releases/details/jon-peddie-research-announces-2nd-quarter-pc-graphics-shipments, July 2010.

[64] Aris Karastergiou. www.physics.ox.ac.uk/users/Karastergiou/Site/home.html, July 2011.

[65] Paul H. J. Kelly. www.doc.ic.ac.uk/~phjk, May 2011.

[66] Khronos Group — Khronos Group releases OpenCL 1.0 specification. www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification, December 2008.

[67] Khronos Group — OpenCL: the open standard for parallel programming of heterogeneous systems. www.khronos.org/opencl, November 2010.

[68] Khronos Group — OpenGL: the industry's foundation for high performance graphics. www.opengl.org, April 2011.

[69] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: a Hands-on Approach*. Morgan Kaufmann, 2010.

[70] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, September 1979.

[71] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. A scalable high performant Cholesky factorization for multicore with GPU accelerators. In *Proceedings of the 9th International Conference on High-Performance Computing for Computational Science*, pages 93–101. Springer-Verlag, 2010.

[72] University of Manchester, Computational Science Community Wiki. wiki.rcs.manchester.ac.uk/community/GPU/Club, May 2011.

[73] The MathWorks — MATLAB GPU computing with NVIDIA CUDA-enabled GPUs. www.mathworks.com/discovery/matlab-gpu.html, July 2011.

[74] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[75] Simon McIntosh-Smith and Richard B. Sessions. An accelerated, computer assisted molecular modeling method for drug design. In *International Supercomputing*, June 2008. www.cs.bris.ac.uk/~simonm/publications/ClearSpeed_BUDE_ISC08.pdf.

[76] Simon McIntosh-Smith, Terry Wilson, Jon Crisp, Amaurys Ávila Ibarra, and Richard B. Sessions. Energy-aware metrics for benchmarking heterogeneous systems. *SIGMETRICS Performance Evaluation Review*, 38:88–94, March 2011.

[77] Simon McIntosh-Smith, Terry Wilson, Amaurys Ávila Ibarra, Richard B. Sessions, and Jon Crisp. Benchmarking energy efficiency, power costs and carbon emissions on heterogeneous systems. *The Computer Journal*, 2011 (to appear).

[78] Simon McIntosh-Smith. www.cs.bris.ac.uk/home/simonm, April 2011.

[79] Hans Meuer. Top500 supercomputer sites. www.top500.org, April 2011.

[80] University of Michigan, Molecular and Behavioral Neuroscience Institute — R+GPU. brainarray.mbni.med.umich.edu/brainarray/rgpgpu, May 2011.

[81] Microsoft Corp — DirectCompute. blogs.msdn.com/b/chuckw/archive/2010/07/14/directcompute.aspx, April 2011.

[82] Microsoft Corp — Microsoft DirectX. www.gamesforwindows.com/en-US/directx, April 2011.

[83] Gordon Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, pages 114–117, April 1965.

[84] Samuel K. Moore. With Denver project NVIDIA and ARM join CPU-GPU integration race. spectrum.ieee.org/tech-talk/semiconductors/processors/with-denver-project-nvidia-and-arm-join-cpugpu-integration-race, January 2011.

[85] Trevor Mudge. Power: a first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[86] Aaftab Munshi, Benedict Gaster, Timothy Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, 2011.

[87] NAG Ltd — NAG library for SMP & multicore. www.nag.co.uk/numeric/FL/FSdescription.asp, May 2011.

[88] NAG Ltd — NAG numerical routines for GPUs. www.nag.co.uk/numeric/GPUs, May 2011.

[89] NAG Ltd — NAG training. nag.co.uk/support_training.asp, April 2011.

[90] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 30:1–30:10. ACM, 2009.

[91] NVIDIA Corp — NVIDIA unveils CUDA – the GPU computing revolution begins. www.nvidia.com/object/IO_37226.html, November 2006.

[92] NVIDIA Corp — NVIDIA OpenCL JumpStart Guide. developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf, April 2009.

[93] NVIDIA Corp — CUBLAS library. developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUBLAS_Library.pdf, August 2010.

[94] NVIDIA Corp — CUDA-accelerated applications. www.nvidia.com/object/cuda_app_tesla.html, April 2011.

[95] NVIDIA Corp — CUDA Fortran. www.nvidia.com/object/cuda_fortran.html, April 2011.

[96] NVIDIA Corp — CUDA zone. www.nvidia.com/object/cuda_home_new.html, April 2011.

[97] NVIDIA Corp — NVIDIA developer zone: tools & ecosystem. developer.nvidia.com/tools-ecosystem, April 2011.

[98] NVIDIA Corp — NVIDIA OpenCL. www.nvidia.com/object/cuda_opencl_new.html, April 2011.

[99] Tom Olson. Why OpenCL will be on every smartphone in 2014. blogs.arm.com/multimedia/263-why-opencl-will-be-on-every-smartphone-in-2014, August 2010.

[100] Vijay Pande. Folding@home. folding.stanford.edu, May 2011.

[101] Vijay Pande. OpenMM. simtk.org/home/openmm, April 2011.

[102] Kevin Parrish. Intel to ship 10-core CPUs in first half of 2011. www.tomshardware.com/news/Xeon-westmere-EX-Nehalem-EX-10-cores-160-threads,12230.html, February 2011.

[103] The Portland Group. www.pgroup.com, July 2011.

[104] Don Scansen. Closer look inside AMD's Llano APU at ISSCC. www.eetimes.com/electronics-news/4087527/Closer-look-inside-AMD-s-Llano-APU-at-ISSCC, February 2010.

[105] Matthew Scarpino. A gentle introduction to OpenCL. *Dr. Dobb's Journal*, July 2011. drdobbs.com/high-performance-computing/231002854.

[106] SciComp inc — SciGPU achieves stunning acceleration for PDE and MC derivative pricing models. www.scicomp.com/parallel_computing/GPU_OpenMP, July 2011.

[107] SciComp Inc — SciFinance achieves astounding accelerations with parallel computing. www.scicomp.com/parallel_computing/GPU_OpenMP, May 2011.

[108] Stan Scott. www.qub.ac.uk/research-centres/HPDC/Profile/?name=ns.scott, May 2011.

[109] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9:83–98, August 2001.

[110] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. www.gotw.ca/publications/concurrency-ddj.htm.

[111] The MathWorks — MATLAB's parallel computing toolbox. www.mathworks.com/products/parallel-computing/?s_cid=HP_FP_ML_parallelcomptbx, May 2011.

[112] The MPI Forum — MPI: a message-passing interface standard, Version 2.2. www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, September 2009.

[113] The Portland Group — PGI Accelerator Compilers. www.pgroup.com/resources/accel.htm, August 2011.

[114] Top500.org — Tianhe-1A. www.top500.org/system/10587, April 2010.

[115] University of Illinois at Urbana-Champaign — GPU acceleration of molecular modeling applications. www.ks.uiuc.edu/Research/gpu, April 2011.

[116] Jeffrey Vetter. Toward exascale computational science with heterogeneous processing. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, page 1. ACM, 2010.

[117] Jeffrey Vetter. The SHOC benchmark suite. ft.ornl.gov/doku/shoc, April 2011.

[118] Wolfram Research — grid*Mathematica*. www.wolfram.com/products/gridmathematica, May 2011.

[119] Wolfram Research — *Mathematica* 7 built-in parallel computing. www.wolfram.com/products/mathematica/newin7/content/BuiltInParallelComputing, May 2011.

[120] Wolfram Research — *Mathematica* 8 CUDA and OpenCL support. www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support, May 2011.

[121] Dong Hyuk Woo and H-H. S. Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *IEEE Computer*, 41(12):24–31, December 2008.

[122] ZiiLABS Pte Ltd — ZiiLABS OpenCL SDK. www.ziilabs.com/technology/opencl.aspx, April 2011.

The GPU Computing Revolution

From Multi-Core CPUs to Many-Core Graphics Processors

A Knowledge Transfer Report from the London Mathematical Society and the Knowledge Transfer Network for Industrial Mathematics

by Simon McIntosh-Smith

LONDON MATHEMATICAL SOCIETY

Knowledge Transfer Network

Industrial Mathematics