The strong need for increased computational performance in science and engineering has led to the use of heterogeneous computing, with GPUs and other accelerators acting as coprocessors for arithmetic intensive data-parallel workloads.

OpenCL is a new industry standard for task-parallel and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs. This trend toward heterogeneous computing and highly parallel architectures has created a strong need for software development infrastructure in the form of parallel programming languages and subroutine libraries that can support heterogeneous computing on multiple vendors’ hardware platforms. To address this, developers adapted many existing science and engineering applications to take advantage of multicore CPUs and massively parallel GPUs using toolkits such as Threading Building Blocks (TBB), OpenMP, Compute Unified Device Architecture (CUDA), and others.

Existing programming toolkits, however, were either limited to a single microprocessor family or didn’t support heterogeneous computing. OpenCL provides easy-to-use abstractions and a broad set of programming APIs based on past successes with CUDA and other programming toolkits. OpenCL defines core functionality that all devices support, as well as optional functionality for high-function devices; it also includes an extension mechanism that lets vendors expose unique hardware features and experimental programming interfaces for application developers’ benefit. Although OpenCL can’t mask significant differences in hardware architecture, it does guarantee portability and correctness. This makes it much easier for developers to start with a correctly functioning OpenCL program tuned for one architecture and produce a correctly functioning program optimized for another architecture.

The OpenCL Programming Model

In OpenCL, a program is executed on a computational device, which can be a CPU, GPU, or another accelerator (see Figure 1). Devices contain one or more compute units (processor cores). These units are themselves composed of one or more single-instruction multiple-data (SIMD) processing elements (PE) that execute instructions in lock-step.

OpenCL Device Management

By providing a common language and common programming interfaces and hardware abstractions, OpenCL lets developers accelerate applications with task- or data-parallel computations in a heterogeneous computing environment consisting of the host CPU and any attached OpenCL devices. Such devices might or might not share memory with the host CPU, and typically have a different machine instruction set. The OpenCL programming interfaces therefore assume heterogeneity between the host and all attached devices.

OpenCL’s key programming interfaces include functions for:

- enumerating available target devices (CPUs, GPUs, and various accelerators);
- managing the target devices’ contexts;
- managing memory allocations;
- performing host-device memory transfers;
- compiling the OpenCL programs and kernel functions that the devices will execute;
- launching kernels on the target devices;
- querying execution progress; and
- checking for errors.

Although developers can compile and link OpenCL programs into binary objects using offline compilation methodology, OpenCL
encourages runtime compilation that lets OpenCL programs run natively on the target hardware—even on platforms unavailable to the original software developer. Runtime compilation eliminates dependencies on instruction sets, letting hardware vendors significantly change instruction sets, drivers, and supporting libraries from one hardware generation to the next. Applications that use OpenCL’s runtime compilation features will automatically take advantage of the target device’s latest hardware and software features without having to recompile the main application itself.

Because OpenCL targets a broad range of microprocessor designs, it must support a multiplicity of programming idioms that match the target architectures. Although OpenCL guarantees kernel portability and correctness across a variety of hardware, it doesn’t guarantee that a particular kernel will achieve peak performance on different architectures; the hardware’s underlying nature might make some programming strategies more appropriate for particular platforms than for others. As an example, a GPU-optimized kernel might achieve peak memory performance when a single work-group’s work-items collectively perform loads and stores, whereas a Cell-optimized kernel might perform better using a double buffering strategy combined with calls to `async_workgroup_copy()`. Applications can query device attributes to determine properties of the available compute units and memory systems and use them accordingly.

Before an application can compile OpenCL programs, allocate device memory, or launch kernels, it must first create a context associated with one or more devices. Because OpenCL associates memory allocations with a context rather than a specific device, developers should exclude devices with inadequate memory capacity or launch kernels, it must first create a context associated with one or more devices. Because OpenCL associates memory allocations with a context rather than a specific device, developers should exclude devices with inadequate memory capacity when creating a context, otherwise the least-capable device will limit the maximum memory allocation. Similarly, they should exclude devices from a context if they don’t support features that OpenCL programs require to run on that context.

Once a context is created, OpenCL programs can be compiled at runtime by passing the source code to OpenCL compilation functions as arrays of strings. After an OpenCL program is compiled, handles can be obtained for the kernel functions contained in the program. The kernels can then be launched on devices within the OpenCL context. OpenCL host-device memory I/O operations and kernels are executed by enqueuing them into one of the command queues associated with the target device.

**OpenCL and Modern Processor Architectures**

State-of-the-art microprocessors contain several architectural features that, historically, have been poorly supported or difficult to use in existing programming languages. This has led vendors to create their own programming tools, language extensions, vector intrinsics, and subroutine libraries to close the programmability gap created by these hardware features. To help clarify the relationship between the OpenCL programming
model and the diversity of potential target hardware, we compare the architectural characteristics of three exemplary microprocessor families and relate them to key OpenCL abstractions and OpenCL programming model features.

**Multicore CPUs**
Modern CPUs are typically composed of a few high-frequency processor cores with advanced features such as out-of-order execution and branch prediction. CPUs are generalists that perform well for a wide range of applications, including latency-sensitive sequential workloads and coarse-grained task- or data-parallel workloads.

**The SIMD clusters execute machine instructions in lock-step; branch divergence is handled by executing both branch paths and masking off results from inactive processing units as necessary.**

Because they’re typically used for latency sensitive workloads with minimal parallelism, CPUs require large caches to hide main-memory latency. Many CPUs also incorporate small-scale use of SIMD arithmetic units to boost the performance of dense arithmetic and multimedia workloads. Because conventional programming languages like C and Fortran don’t directly expose these units, their use requires calling vectorized subroutine libraries or proprietary vector intrinsic functions, or trial-and-error source-level restructuring and auto-vectorizing compilers. AMD, Apple, and IBM provide OpenCL implementations that target multicore CPUs, and support the use of SIMD instruction set extensions such as x86 SSE and Power/VMX (vector multimedia extensions). The current CPU implementations for x86 processors often make best use of SSE when OpenCL kernels are written with explicit use of float4 types. CPU implementations often map all memory spaces onto the same hardware cache, so a kernel that explicitly uses constant and local memory spaces might actually incur more overhead than a simple kernel that uses only global memory references.

**The Cell Processor**
The Cell Broadband Engine Architecture (CBEA) is a heterogeneous chip architecture consisting of one 64-bit Power-compliant PE (PPE), multiple Synergistic PEs (SPEs), a memory-interface controller, and I/O units, connected with an internal high-speed bus. The PPE is a general-purpose processor based on the Power architecture and it’s designed to run conventional OS and control-intensive code to coordinate tasks running on SPEs.

The SPE is a SIMD streaming processor optimized for massive data processing that provides most of the Cell systems’ computing power. Developers can realize an application’s task parallelism using multiple SPEs, while achieving data and instruction parallelism using the SIMD instructions and the SPEs’ dual execution pipelines. Each SPE has local store, a small, local software-managed cache-like fast memory. Applications can load data from system memory to local store or vice versa using direct memory access (DMA) requests, with the best bandwidth achieved when both source and destination are aligned to 128 bytes. Cell can execute data transfer and instructions simultaneously, letting application developers hide memory latency using techniques such as double buffering. (We describe the architecture and a sample application ported to the Cell processor elsewhere.)

IBM has recently released an OpenCL toolkit supporting both the Cell and Power processors on the Linux platform. The IBM OpenCL implementation supports the embedded profile for the Cell SPUs, and uses software techniques to smooth over some architectural differences between the Cell SPUs and conventional CPUs. On the Cell processor, global memory accesses perform best when operands are a multiple of 16 bytes (such as an OpenCL float4 type). The use of larger vector types such as float16 lets the compiler unroll loops, further increasing performance. The program text and OpenCL local and private variables share the 256 Kbyte Cell SPU local store, which limits the practical work-group size because each work-item requires private data storage. The Cell DMA engine performs most effectively using double buffering strategies combined with calls to async_workgroup_copy() to load data from global memory into local store.

**Graphics Processing Units**
Contemporary GPUs are composed of hundreds of processing units running at low to moderate frequency, designed for throughput-oriented latency
Insensitive workloads. To hide global memory latency, GPUs contain small or moderate sized on-chip caches and extensively use hardware multithreading, executing tens of thousands of threads concurrently across the pool of processing units. The GPU processing units are typically organized in SIMD clusters controlled by a single instruction decoder, with shared access to fast on-chip caches and shared memories. The SIMD clusters execute machine instructions in lock-step; branch divergence is handled by executing both branch paths and masking off results from inactive processing units as necessary. Using SIMD architecture and in-order instruction execution allows GPUs to contain many more arithmetic units in the same area than traditional CPUs.2,3

Both AMD and Nvidia have released OpenCL implementations supporting their respective GPUs. These devices require many OpenCL work-items and work-groups to fully saturate the hardware and hide latency. Nvidia GPUs use a scalar processor architecture for individual PEs exposed by OpenCL, making them highly efficient on most OpenCL data types. AMD GPUs use a vector architecture, and typically achieve best performance when OpenCL work-items operate on four-element vector types (such as float4). In many cases, a vectorized OpenCL kernel can perform well on x86 CPUs and on AMD and Nvidia GPUs, but the resulting kernel code might be less readable than the scalar equivalent. Differences in low-level GPU architecture—including variations on what memory is cached and what memory access patterns create bank conflicts—affect kernel optimality. Vendor-provided OpenCL literature typically contains low-level optimization guidelines.

In the examples that follow, we refrain from detail and focus on general OpenCL programming concepts.

### An Example OpenCL Kernel

To illustrate the process of moving serial code to OpenCL, we’ll use an example kernel from the Adaptive Poisson-Boltzmann Solver.10 APBS is a package for calculating biomolecular solvation using the Poisson-Boltzmann equation (PBE). The PBE is a popular continuum model that describes electrostatic interactions between molecular solutes. As part of solving the PBE, potentials are discretized onto a grid that’s larger than the bounding volume containing the target molecule. Under Dirichlet boundary conditions, we can solve the potential contribution of grid points on the grid’s faces using the single Debye–Hückel (SDH) method or the multiple Debye–Hückel (MDH) method. For the MDH method, the potential at a grid point \( i \) located at position \( \mathbf{r}_i \) is given by

\[
V_i = \alpha \sum_j \frac{q_j e^{-\alpha(r_{ij} - c_j)}}{r_{ij} 1.0 + \kappa \sigma_j},
\]

with the sum taken over all atoms, where \( \alpha \) is a prefactor that accounts for the system of units and solution dielectric values, atom \( j \) is located at \( \mathbf{r}_j \) and has partial charge \( q_j \) and size \( \sigma_j \), and the pairwise distance is \( r_{ij} = |\mathbf{r}_j - \mathbf{r}_i| \).

The potential at each grid point is effectively the sum of all atomic potential contributions in the molecule. The MDH method is inherently data-parallel when decomposed over grid points because they’re computed independently and there are no output conflicts. Figure 2 shows a serial MDH algorithm, and Figure 3 shows a C implementation.
Because the potential at each grid point can be calculated independently, we can use standard methods (such as pthreads or OpenMP) to parallelize the earlier example on a CPU. We can also perform this type of calculation with OpenCL with almost no modification. The kernel is simply the inner loop over the atoms, as Figure 4 shows. In effect, the OpenCL dispatcher becomes the outer loop over grid points. We set the OpenCL global work-group size to the total number of grid points, and each work-item that OpenCL dispatches is responsible for calculating a potential using the above algorithm, improving performance. On the Cell processor, using vector types such as float4 or float16, which makes it easier for the OpenCL compiler to effectively fill very long instruction word (VLIW) slots and enables wider memory transfer operations. With vector types, individual work-items process multiple grid points at a time. This reduces the global work dimensions accordingly, with a corresponding increase in register usage. Calculating multiple grid points per work-item increases the ratio of arithmetic operations to memory operations because the same atom data is referenced multiple times. On the AMD and Nvidia GPUs, the use of vector types yields a 20 percent increase in performance. On the Cell processor, using float16 vectors yields a factor 11 times increase in kernel performance.

Another optimization involves using vector types such as float4 or float16, which makes it easier for the OpenCL compiler to effectively fill very long instruction word (VLIW) slots and enables wider memory transfer operations. With vector types, individual work-items process multiple grid points at a time. This reduces the global work dimensions accordingly, with a corresponding increase in register usage. Calculating multiple grid points per work-item increases the ratio of arithmetic operations to memory operations because the same atom data is referenced multiple times. On the AMD and Nvidia GPUs, the use of vector types yields a 20 percent increase in performance. On the Cell processor, using float16 vectors yields a factor 11 times increase in kernel performance.

Figure 4's code runs approximately 20 times faster on an Nvidia GeForce GTX 285 GPU than the serial code on a 2.5 GHz Intel Nehalem CPU. For reference, on 16 cores, the parallel CPU performance is almost 13 times faster. However, the kernel in Figure 4 doesn’t take advantage of locality of concurrent accesses to the atom data. In this form, each work-item (grid point) is responsible for loading each atom’s data (x, y, z, charge, and size) resulting in global memory transactions that could be avoided by changing the algorithm. By taking advantage of each OpenCL compute unit’s fast on-chip local memory, data can be staged in local memory and then efficiently broadcast to all work-items within the same work-group. This greatly amplifies the effective memory bandwidth available to the algorithm, improving performance. Although the global work-group size remains the same, the local work-group size increases from 1 to some multiple of the hardware SIMD width.

As Figure 5’s example shows, the work-group size is limited by the amount of data that can be loaded into shared memory (typically 16 Kbytes on Nvidia GPUs). The on-chip local memory is partitioned so that each of a work-group’s work-items loads a block of the position, charge, and size data into local memory at a specific offset. Local memory barriers ensure that data isn’t overwritten before all work-items in a work-group have accessed it. This coordinated data loading and sharing reduces the number of slow global memory accesses.

领先地位

Figure 4. A simple OpenCL kernel for the multiple Debye-Hückel method. This kernel is similar to the original sequential C loops, except that the outer loop over grid points has been replaced by a parallel instantiation of independent grid points as OpenCL work-items, and the igrid index is determined by the OpenCL work-item index.
A more important outcome still is that the numerical result is exact—within the floating point rounding mode and summation order—to the CPU methods (both serial and parallel).

Our initial experiences in adapting molecular modeling applications such as APBS and VMD\textsuperscript{11} to OpenCL 1.0 have been generally positive. In the coming year, we expect OpenCL to incorporate new features and promote previously optional features to core features of OpenCL 1.1 and later versions. Recent GPUs allow direct access to host memory over peripheral component interconnect express (PCI-E), and some enable global GPU memory

![Figure 5. The optimized OpenCL kernel for the multiple Debye-Hückel method. This kernel is similar to the simple OpenCL kernel in Figure 4, but each work-group collectively loads and processes blocks of atom data in fast on-chip local memory. OpenCL barrier instructions enforce coordination between the loading and processing phases to maintain local memory consistency. The inner loop uses vector types to process multiple grid points per work-item, and atom data is processed entirely from on-chip local memory, greatly increasing both arithmetic intensity and effective memory bandwidth.](image-url)
mapping into the host address space, allowing fast zero-copy access to data that’s read or written only once during kernel execution. Although zero-copy APIs exist in CUDA, OpenCL currently doesn’t include this capability. As OpenCL matures, we hope to see increased support for thread-safety, increased OpenGL interoperability, and extension with advanced features found in APIs such as CUDA.

OpenCL provides correctness guarantees such that code written and optimized for one device will run correctly on any other device, although not necessarily with peak performance. The only exception to this is when optional OpenCL features are used that aren’t available on the target device. As future hardware platforms that operate on wider vectors—such as Intel’s advanced vector extensions (AVX)—arrive, we’re eager to see OpenCL implementations incorporate a greater degree of autovectorization, enabling efficient kernels to be written with less vector width specificity, and improving the performance portability of OpenCL kernels across the board. Our experiences show that OpenCL holds great promise as a standard low-level parallel programming interface for heterogeneous computing devices.

Acknowledgments
We are grateful to Nathan Baker and Yong Huang for development and support of the APBS project, and to Ian Ollmann and Aaftab Munshi for assistance with OpenCL. The US National Institutes of Health funds APBS development through grant R01-GM069702. Performance experiments were made possible with hardware donations and OpenCL software provided by AMD, IBM, and Nvidia, and with support from the US National Science Foundation’s Computer and Network Systems grant 05-51665, the US National Center for Supercomputing Applications, and NIH grant P41-RR05969.

References

John Stone is a senior research programmer in the Theoretical and Computational Biophysics Group at the Beckman Institute for Advanced Science and Technology, and associate director of the CUDA Center of Excellence, both at the University of Illinois at Urbana-Champaign. He is lead developer of the molecular visualization program VMD. His research interests include scientific visualization and high-performance computing. Stone earned his MS in computer science from the Missouri University of Science and Technology. Contact him at johns@uiuc.edu.

David Gohara is director of research computing in the Department of Biochemistry and Biophysics and a Senior Programmer in the Center for Computational Biology at The Washington University School of Medicine in St. Louis. His research interests are in high-performance computing for improving computational methods used for the study of biological processes. Gohara has a PhD in biochemistry and molecular biology from the Pennsylvania State University, and did his postdoctoral research in x-ray crystallography at Harvard Medical School. Contact him at gohara@biochem.wustl.edu.

Guochun Shi is a research programmer at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. His research interests are in high-performance computing. Shi has an MS in computer science from the University of Illinois at Urbana-Champaign. Contact him at gshi@ncsa.uiuc.edu.
Nonmember rate of $32 for S&P magazine!

*IEEE Security & Privacy* is THE premier magazine for security professionals.

Top security professionals in the field share information on which you can rely:

- Silver Bullet podcasts and interviews
- Intellectual Property Protection & Piracy
- Designing for Infrastructure Security
- Privacy Issues
- Legal Issues & Cybercrime
- Digital Rights Management
- The Security Profession

Visit our Web site at www.computer.org/security/

Subscribe now! www.computer.org/services/nonmem/spbnr