Published in final edited form as:
Comput Sci Eng. 2010 May ; 12(3): 66–72. doi:10.1109/MCSE.2010.69.

OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems

John E. Stone[*], **David Gohara**[†], and **Guochun Shi**[‡]

[*] Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL, 61801

[†] Dept. Biochemistry and Molecular Biophysics, Washington University in St. Louis, St. Louis, MO, 63110

[‡] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL, 61801

Abstract

We provide an overview of the key architectural features of recent microprocessor designs and describe the programming model and abstractions provided by OpenCL, a new parallel programming standard targeting these architectures.

Introduction

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 co-processors for arithmetic intensive data-parallel workloads [1–4]. 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[5]. The trend towards 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 supporting heterogeneous computing on hardware platforms produced by multiple vendors. Many existing science and engineering applications have been adapted to make effective use of multi-core CPUs and massively parallel GPUs using toolkits such as Threading building blocks (TBB), OpenMP, CUDA [6], and others like them [7,8]. Existing programming toolkits have either been limited to a single microprocessor family or did not support heterogeneous computing. OpenCL provides easy-to-use abstractions and a broad set of programming APIs based on past successes with CUDA, TBB, and other programming toolkits. OpenCL defines a set of core functionality that is supported by all devices, as well as optional functionality that may only be implemented on high-function devices, and includes an extension mechanism that allows vendors to expose unique hardware features and experimental programming interfaces for the benefit of application developers. Although OpenCL cannot mask significant differences in hardware architectures, it does guarantee portability and correctness. This makes it much easier for a developer to begin with a correctly functioning OpenCL program tuned for one architecture, and produce a correctly functioning program optimized for another architecture.

The OpenCL programming model

OpenCL provides a common language, programming interfaces, and hardware abstractions enabling developers to accelerate applications with task-parallel or data-parallel computations in a heterogeneous computing environment consisting of the host CPU and any attached OpenCL “devices”. OpenCL devices may or may not share memory with the host CPU, and

typically have a different machine instruction set, so the OpenCL programming interfaces assume heterogeneity between the host and all attached devices. The key programming interfaces provided by OpenCL include functions for enumerating available target devices (CPUs, GPUs, and Accelerators of various types), managing "contexts" containing the devices to be used, managing memory allocations, performing host-device memory transfers, compiling OpenCL programs and "kernel" functions to be executed on target devices, launching kernels on target devices, querying execution progress, and checking for errors.

Although OpenCL programs can be compiled and linked into binary objects using conventional off-line compilation methodology, OpenCL also supports run-time compilation enabling OpenCL programs to run natively on the target hardware, even on platforms unavailable to the original software developer. Run-time compilation eliminates dependencies on instruction sets, allowing hardware vendors to make significant changes to instruction sets, drivers, and supporting libraries, from one hardware generation to the next. Applications that make use of the run-time compilation features of OpenCL will automatically take advantage of the latest hardware and software features of the target device without any need for recompilation of the main application itself.

OpenCL targets a broad range of microprocessor designs, requiring that it support a multiplicity of programming idioms that are matched to the target architectures. Although OpenCL provides guarantees of portability and correctness of kernels across a variety of hardware, it does not guarantee that a particular kernel will achieve peak performance on different architectures. The nature of the underlying hardware may make some programming strategies more appropriate for particular platforms than for others. As an example, a GPU-optimized kernel may achieve peak memory performance when the work items in a single work group collectively perform loads and stores, where a Cell-optimized kernel may perform better with the use of of a double buffering strategy combined with calls to *async_workgroup_copy()*. Applications select the most appropriate kernel for the target devices by querying the capabilities and hardware attributes of the installed devices at runtime.

The OpenCL programming model abstracts CPUs, GPUs, and other accelerators as "devices" that contain one or more "compute units" (e.g., cores) composed of one or more SIMD "processing elements" (PEs) that execute instructions in lock-step. OpenCL defines four types of memory systems that devices may incorporate, a large high-latency "global" memory, small low-latency read-only "constant" memory, shared "local" memory accessible from multiple PEs within the same compute unit, and "private" memory or device registers accessible within each PE. Local memory may be implemented using either high-latency global memory, or may be implemented with fast on-chip SRAM or shared register file. Applications can query device attributes to determine the properties of the available compute units and memory systems, using 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. Memory allocations are associated with a context rather than a specific device. Devices with inadequate memory capacity should be excluded when creating a context, otherwise the maximum memory allocation will be limited by the least-capable device. Similarly, it may be necessary to exclude some devices from a context in the case that they do not support features required by OpenCL programs to be run on the newly created 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 specific "kernels" contained by the program. The "kernel" functions can then be launched on devices within the OpenCL context. OpenCL host-device memory I/O operations and

kernels are executed on a target device by enqueing them into a command queue associated with the target device.

OpenCL and Modern Processor Architectures

State-of-the-art microprocessors contain a number of architectural features that have historically been poorly supported or are difficult to utilize in existing programming languages. This has led vendors to create their own programming tools, language extensions, vector intrinsics, and subroutine libraries to close the gap in programmability created by these hardware features. To help clarify the relationship between 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 features of the OpenCL programming model.

Multi-core CPUs

Modern CPUs are typically composed of a small number of 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 variety of applications including latency-sensitive sequential workloads, and coarse-grained task-parallel or data-parallel workloads. Since they are typically used for latency sensitive workloads with minimal parallelism, CPUs make extensive use of large caches to hide main memory latency. Many CPUs also incorporate small scale use of single-instruction multiple-data (SIMD) arithmetic units to boost the performance of dense arithmetic and multimedia workloads. These SIMD units are not directly exposed by conventional programming languages like C and Fortran, so their use requires calling vectorized subroutine libraries or proprietary vector intrinsic functions, or trial-and-error source level restructuring and autovector-izing compilers. AMD, Apple, and IBM provide OpenCL implementations that target multi-core CPUs, and support the use of SIMD instruction set extensions such as x86 SSE and Power/VMX. 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 makes explicit use of constant and local memory spaces may actually incur more overhead than a simple kernel that only uses global memory references.

The Cell Processor

The Cell Broadband Engine Architecture (CBEA) is a heterogeneous chip architecture consisting of one 64-bit Power-compliant Processor Element (PPE), multiple Synergistic Processor Elements (SPE), a Memory Interface Controller and I/O units, connected with an internal high speed bus [9]. The PPE is a general purpose processor based on the IBM Power-architecture and it is designed to run conventional operating system and control-intensive code to coordinate tasks running on SPEs. The SPE, a SIMD streaming processor, provides most of the computing power for the Cell systems with its design optimized for massive data processing. An application's task parallelism can be realized using multiple SPEs while the data parallelism and instruction parallelism can be realized using the SIMD instructions and dual execution pipelines in SPEs. Each SPE has a small software-managed cache-like fast memory local to an SPE, called local store. Applications can load data from system memory to local store or the other way around using DMA requests, with the best bandwidth achieved when both source and destination are aligned to 128 bytes. The data transfer and instructions execution can happen simultaneously, enabling application programmers to hide memory latency using techniques like double buffering. Shi e al. have provided a detailed description of the architecture and a sample application ported to the Cell processor [1].

IBM has 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 of the 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, e.g. an OpenCL *float4* type. The use of larger vector types such as *float16* enables the compiler to unroll loops, further increasing performance. The 256 kB Cell SPU local store, is shared among the program text, and OpenCL "local", and "private" variables. This places practical limits on the size of work-groups since private data storage is required for each work-item. The Cell DMA engine performs most effectively with the use of 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 a low to moderate frequency, designed for throughput-oriented latency insensitive workloads. In order to hide global memory latency, GPUs contain small or moderate sized on-chip caches, and they make extensive use of 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, and branch divergence is handled by executing both paths of the branch and masking off results from inactive processing units as necessary. The use of SIMD architecture and in-order execution of instructions allows GPUs to contain a larger number of arithmetic units in the same area as compared to traditional CPUs.

Massively parallel arithmetic-heavy hardware design enables state-of-the-art GPUs to achieve single-precision floating point arithmetic rates approaching 2 TFLOPS (trillions of instructions per second). Due to the demands of graphics workloads, GPUs are designed with global memory systems capable of bandwidths approaching 200 GB/sec. GPU global memory is organized in multiple banks, with peak performance attained when accesses are aligned on appropriate address boundaries, and groups of SIMD units cooperatively load data from a contiguous block of memory addresses, known as a "coalesced" memory access. When a memory access is not aligned on an appropriate address boundary and in consecutive sequence, the memory access must be split into multiple transactions resulting in a significant reduction in effective bandwidth, and increasing latency.

Although GPUs are powerful computing devices in their own right, they must currently be managed by the host CPUs. GPUs are typically attached to the host by a PCI-Express bus, and in most cases have their own independent on-board memory system. In order to exchange input and output data with the GPU, the host CPU schedules DMA transfers between the host and GPU memory systems. OpenCL provides APIs for CPU-directed data transfers between independent host and GPU memory systems. Recent GPUs are capable of direct access to host memory over PCI-e, and in some cases may allow their on-board to be mapped into the host address space, providing the necessary hardware support for zero-copy access to data that are read or written only once during kernel execution. At the present time, OpenCL does not include mechanisms for zero-copy memory access, though it could be provided as an extension or as part of a future version.

Both AMD and NVIDIA have released OpenCL implementations supporting their respective GPUs. These devices require a large number of OpenGL work-items and work-groups to fully saturate the hardware and hide latency. NVIDIA GPUs use a scalar processor architecture for the individual PEs seen by OpenCL, enabling them to work with high efficiency on most OpenCL data types. AMD GPUs use a vector architecture, and typically achieve best

performance such that OpenCL work-items operate on 4-element vector types such as *float4*. In many cases, a vectorized OpenCL kernel can be made to perform well on x86 CPUs and on AMD and NVIDIA GPUs, though the resulting kernel code may 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 discuss an example kernel from the Adaptive Poisson-Boltzmann Solver (APBS) [10]. APBS is a package for calculating biomolecular solvation through the use of the Poisson-Boltzmann equation (PBE). The PBE is a popular continuum model that describes electrostatic interactions between molecular solutes. As part of the solution of the PBE, potentials are discretized onto a grid sized larger than the bounding volume containing the molecule of interest. Under Dirichlet boundary conditions, the potential contribution of grid points on the faces of the grid can be solved using the Single Debye-Hückel (SDH) method or 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

$$V_i = \alpha \sum_j \frac{q_i}{r_{ij}} \frac{e^{-\kappa(r_{ij}-\sigma_j)}}{1.0+\kappa\sigma_j}, \tag{1}$$

with the sum taken over all atoms, where α 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 σ_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 since they are computed independently and there are no output conflicts. A serial MDH algorithm is summarized in Alg. 1, and a C implementation is shown in Fig. 2.

Since the potential at each grid point can be calculated independently, the above example can be trivially parallelized on a CPU using standard methods (pthreads or OpenMP, for example). This type of calculation can also be performed with OpenCL with almost no modification. The kernel is simply the inner loop over the atoms, as shown in Fig. 3. In effect, the OpenCL dispatcher becomes the outer loop over grid points. The OpenCL global work-group size is set to the total number of grid points. Each work-item dispatched by OpenCL is responsible for calculating a potential using the above kernel. Each OpenCL work-item obtain its grid point index its global work-item index.

The code shown in Fig. 3 runs approximately 20× 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× faster. However, the kernel in Fig. 3 does not 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 with some changes to the algorithm. By taking advantage of the fast on-chip "local" memory present on each OpenCL compute unit, data can be staged in local memory and then efficiently broadcast to all of the work-items within the same work-group. This greatly amplifies the effective memory bandwidth available to the algorithm, improving performance. The global work-group size remains the same,

however, the local work-group size is increased from 1 to some multiple of the hardware SIMD width. In the example shown in Fig. 4 the work-group size is limited by the amount of data that can be loaded into shared memory (typically 16 kB, on NVIDIA GPUs). The on-chip shared memory is partitioned so that each work-item of a work-group loads a block of the position, charge and size data into shared memory at a specific offset. Local memory barriers are placed to ensure that data is not overwritten in the shared memory before all of the work-items in a work-group have accessed it. This coordinated loading and sharing of data reduces the number of slow global memory accesses.

Another optimization involves the use of vector types such as *float4* or *float16* making it easier for the OpenCL compiler to effectively fill VLIW instruction slots, and enabling wider memory transfer operations. The use of vector types causes an individual work-item to process multiple grid points at a time, reducing the global work dimensions accordingly, and with a corresponding increase in register usage. By calculating multiple grid points per work-item, the ratio of arithmetic operations to memory operations is increased, since the same atom data is referenced multiple times. On the AMD and NVIDIA GPUs, the use of vector types yields a 20% increase in performance. On the Cell processor, the use of *float16* vectors yields a factor of 11× increase in performance for this kernel.

One variation of these concepts (many others are possible) is shown in Fig. 4. When compared to the original serial CPU code, the approximate performance increase for an IBM Cell blade (using *float16*) is 17× faster, an AMD Radeon 5870 GPU is 129× faster, and an NVIDIA GeForce GTX 285 GPU is 141× faster. With further platform-specific tuning, each of these platforms could undoubtably achieve even higher performance. More important is that the numerical result is exact, within the floating point rounding mode, to the methods used on the CPU (both serial and parallel).

Summary and Concluding Remarks

Our initial experiences in adapting molecular modeling applications such as APBS [10] and VMD [11] to OpenCL 1.0 have been generally positive. In the coming year, we expect that OpenCL will incorporate new features and that previously optional features will be promoted to core features of OpenCL 1.1 and later versions. As OpenCL matures we hope to see increased support for thread-safety, increased interoperability with OpenGL, and extension with advanced features found in APIs like CUDA. With the arrival of future hardware platforms that operate on wider vectors (e.g. Intel AVX), we are eager for OpenCL implementations to incorporate a greater degree of autovectorization, enabling efficient kernels to be written with less vector width specificity. We feel that OpenCL holds great promise as a standard low-level parallel programming interface for heterogeneous computing devices.

Acknowledgments

David Gohara would like to thank Nathan Baker and Yong Huang for development and support of the APBS project. He also thanks Ian Ollmann and Aaftab Munshi for assistance with OpenCL. APBS development is funded by NIH grant R01-GM069702. Performance experiments were made possible with hardware donations and OpenCL software provided by AMD, IBM, NVIDIA, and with support from NSF CNS grant 05-51665, the National Center for Supercomputing Applications, and NIH grant P41-RR05969.

References

1. Shi, Guochun; Kindratenko, Volodymyr; Pratas, Frederico; Trancoso, Pedro; Gschwind, Michael. Application acceleration with the Cell broadband engine. Computing in Science and Engineering 2010;12(1):76–81.

2. Cohen, Jonathan; Garland, Michael. Solving computational problems with GPU computing. Computing in Science and Engineering 2009;11(5):58–63.

3. Bayoumi, Amr; Chu, Michael; Hanafy, Yasser; Harrell, Patricia; Refai-Ahmed, Gamal. Scientific and engineering computing using ATI stream technology. Computing in Science and Engineering 2009;11 (6):92–97.

4. Barker, Kevin J.; Davis, Kei; Hoisie, Adolfy; Kerbyson, Darren J.; Lang, Mike; Pakin, Scott; Sancho, Jose C. SC' 08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Piscataway, NJ, USA: IEEE Press; 2008. Entering the petaflop era: the architecture and performance of Roadrunner; p. 1-11.

5. Munshi, Aaftab. OpenCL Specification Version 1.0. Dec. 2008 http://www.khronos.org/registry/cl/

6. Nickolls, John; Buck, Ian; Garland, Michael; Skadron, Kevin. Scalable parallel programming with CUDA. ACM Queue 2008;6(2):40–53.

7. Stone, John E.; Phillips, James C.; Freddolino, Peter L.; Hardy, David J.; Trabuco, Leonardo G.; Schulten, Klaus. Accelerating molecular modeling applications with graphics processors. J Comp Chem 2007;28:2618–2640. [PubMed: 17894371]

8. Stone, John E.; Saam, Jan; Hardy, David J.; Vandivort, Kirby L.; Hwu, Wenmei W.; Schulten, Klaus. High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs. Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units, ACM International Conference Proceeding Series; 2009. p. 9-18.

9. Hofstee, H Peter. Power efficient processor architecture and the cell processor. HPCA' 05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture; Washington, DC, USA: IEEE Computer Society; 2005. p. 258-262.

10. Baker, Nathan A.; Sept, David; Joseph, Simpson; Holst, Michael J.; McCammon, J Andrew. Electrostatics of nanosystems: Application to microtubules and the ribosome. Proc Natl Acad Sci USA 2001;98(18):10037–10041. [PubMed: 11517324]

11. Humphrey, William; Dalke, Andrew; Schulten, Klaus. VMD – Visual Molecular Dynamics. J Mol Graphics 1996;14:33–38.

Biographies

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. His research interests include scientific visualization and high performance computing. He is the lead developer of the molecular visualization program VMD. Mr. Stone earned his M.S. and B.S. degrees in Computer Science from the Missouri University of Science and Technology. Contact him at johns@ks.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. He has a Ph.D. 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. Mr. Shi has an M.S. in computer science from the University of Illinois at Urbana-Champaign. Contact him at gshi@ncsa.uiuc.edu.

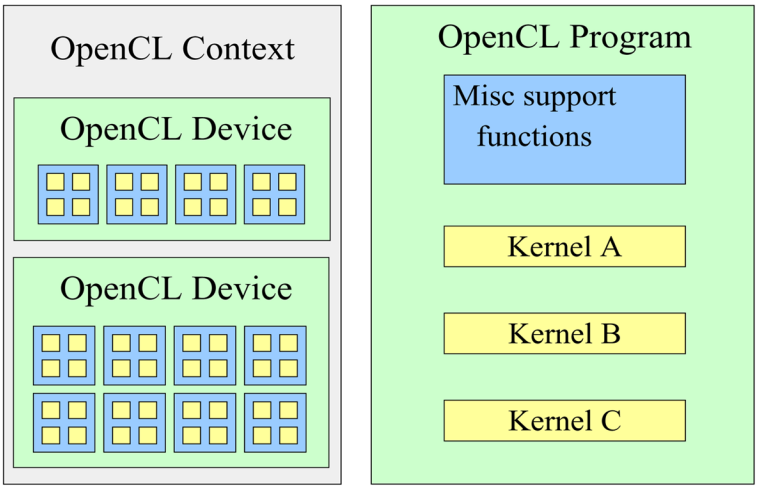

Figure 1.
OpenCL describes hardware in terms of a hierarchy of devices, compute units, and clusters of SIMD processing elements. Before becoming accessible to an application, devices must first be incorporated into an OpenCL context. OpenCL programs contain one or more kernel functions as well as supporting routines that may be used by kernels.

```
for (int igrid=0; igrid<ngrid; igrid++) {
    float v=0.0f;
    for (int jatom=0; jatom<natoms; jatom++) {
        dx = gx[igrid] - ax[jatom];
        dy = gy[igrid] - ay[jatom];
        dz = gz[igrid] - az[jatom];
        dist = sqrt(dx*dx + dy*dy + dz*dz);
        v += pre1 * (charge[jatom] / dist) *
             exp(-xkappa * (dist - size[jatom])) /
             (1.0f + xkappa * size[jatom]);
    }
    val[igrid] = v;
}
```

Figure 2.
The innermost loops of a sequential C implementation of the MDH method.

```
__kernel void mdh(__global float *ax, __global float *ay,
                  __global float *az, __global float *charge,
                  __global float *size, __global float *gx,
                  __global float *gy, __global float *gz,
                  __global float *val,
                  float pre1, float xkappa, cl_int natoms) {
    int igrid = get_global_id(0);
    float v = 0.0f;
    for (int jatom = 0; jatom < natoms; jatom++) {
        float dx = gx[igrid] - ax[jatom];
        float dy = gy[igrid] - ay[jatom];
        float dz = gz[igrid] - az[jatom];
        float dist = sqrt(dx*dx + dy*dy + dz*dz);
        v += pre1 * (charge[jatom] / dist)  *
             exp(-xkappa * (dist - size[jatom])) /
             (1.0f + xkappa * size[jatom]);
    }
    val[igrid] = v;
}
```

Figure 3.

A simple OpenCL kernel for the MDH method is very 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.

Figure 4.

The optimized OpenCL kernel for the MDH method is similar to the simple OpenCL kernel, 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.

Algorithm 1

The MDH algorithm calculates the total potential at each grid point on a grid face, as described in Eq. 1.

```
1:  for i = 1 to M do {loop over grid points on face}
2:      grid potential ⇐ 0.0
3:      for j = 1 to N do {loop over all atoms}
4:          grid potential ⇐ grid potential + (potential from atom j)
5:      end for
6:  end for
7:  return grid potential
```