Webology (ISSN: 1735-188X)
Volume 18, Number 6, 2021

Harnessing The Potential Of Opencl For Heterogeneous Computing

PALYAM NATA SEKHAR [1] , DR. ARPANA BHARANI [2]

[1] Research Scholar, Department of Computer Science, Dr. A. P. J. Abdul Kalam University, Indore, Madhya Pradesh.

[2] Supervisor, Department of Computer Science, Dr. A. P. J. Abdul Kalam University, Indore, Madhya Pradesh.

ABSTRACT

Parallel programming is essential in today's computing landscape, with the rise of heterogeneous architectures and the demand for high-performance computing. OpenCL, as a versatile parallel programming model, has gained prominence in enabling developers to harness the power of heterogeneous computing systems. Through practical examples and optimization strategies, we provide insights into crafting high-performance OpenCL applications. Real-world use cases underscore OpenCL's effectiveness in harnessing heterogeneous computing environments. By comparing OpenCL to other parallel models, we highlight its strengths and versatility. We conclude by discussing challenges and future prospects, emphasizing OpenCL's pivotal role in modern parallel programming.

Keywords: Parallel Programming, Heterogeneous, Performance, Vector, Memory.

I. INTRODUCTION

In general-purpose parallel programming on multiple kinds of processors, OpenCL is intended as an open standard. OpenCL aims to make it easier for software developers to access heterogeneous processing architectures by providing a uniform foundation. C is the programming language of choice for the OpenCL standard, which provides a set of APIs. OpenCL principles rather than technical specifics are all that is required for this thesis. The OpenCL standard contains all of the framework's technical specifications.

Software platform OpenCL (Open Computing Language) aims to offer an interface for programming computational devices such as GPUs, FPGA platforms and certain CPU models. All devices that accept Khronos may execute software written in C99, an open specification maintained by the Khronos Group. This enables parallel processing across a variety of platforms. Developers working with GPGPU applications will find OpenCL particularly useful since it works with all of the main GPU manufacturers and doesn't need learning a new programming language

for every target device. OpenCL, on the other hand, enables one's programme to run on a wide variety of platforms. As a result, OpenCL programmes can run on a wide range of computing platforms.

First, its ability to operate on practically any major computing device and second, its ability to provide utilities for an application to recognise what hardware is available and to modify task allocation among devices to best use the available hardware enables it to achieve this goal. In the next section (and the next chapter), we'll look at a specific application that is ideally suited to, and in dire need of, the benefits of HC.

II. OPENCL APPLICATION FLOW

Figure 1 depicts the OpenCL application flow, with the stages numbered for reference in the subsequent discussion. There are two distinct components to the flow. Runtime objects are created by the platform layer and the kernel is executed in a context provided by the various platforms.

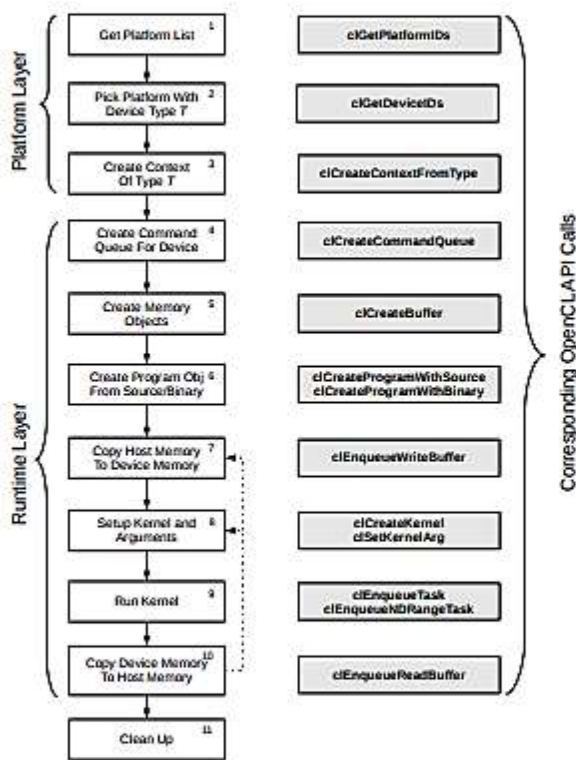

Figure 1 OpenCL Application Flow

<u>Platform Layer</u>

An OpenCL programme first checks to see whether any of the supported platforms are accessible (step 1). A context is then created when the platform list has been retrieved (step 2), and the

required device type has been selected (step 3). There are three kinds of OpenCL devices that may be used: CL DEVICE TYPE CPU, CL DEVICE TYPE GPU, and CL DEVICE TYPE ACCELERATOR. The accessible devices are then multiplied by the specified number in the context (step 3). unless specifically liberated from context, the gadgets are made unique to the context.

Runtime Layer

As part of the runtime layer, these duties are outlined in the following paragraphs. You don't have to do the activities in the sequence outlined below.

Commands mentioned in this article are used to communicate between the host and the devices. A command queue is built for each device specified in the context to send these instructions to the devices (step 4). An OpenCL event object may be generated as an option whenever a command is given.. These event objects enable the application to monitor the command's completion and may be used to explicitly synchronise the process. To allocate space on the devices, memory objects are generated (step 5). When these memory objects are generated, the application determines if the host has permission to read or write to them.

Loading the source code or the binary implementation of one or more kernels creates the programme objects (step 6). You may utilise either the device-specific executable (DSE), or the current OpenCL implementation's intermediate representation (IR). Once the programme objects have been produced, the device-specific executable may be generated from them. If an executable, source code, or IR was used to construct the programme object, the OpenCL implementation determines what action to take in the build stage of the programme object. With the OpenCL API, a binary implementation may be saved to a file and utilised in subsequent executions of the programme without having to recompile. The format of the output file is not part of the OpenCL definition, and the implementation of OpenCL chooses a suitable format for the application. The kernel object is formed from the executable after it has been compiled into the programme object. The kernel object is a representation of one of the program's functions.

Memory copy instructions are sent against connected memory objects to move the input data to device memory before the kernel is executed (step 7). The memory transfer might be non-blocking or blocking, in which control is restored to the programme once the memory transfer is planned. Events are used to synchronise non-blocking transfers. In step 8, the kernel parameters are specified and the kernel is scheduled for execution via the command queue (as seen in Figure 1). (step 9). Host memory is moved from the device to the host after the kernel has finished running (step 10). The same kernel may be scheduled to run again in an iterative process. After the kernel has run, fresh data may be sent to the device and new data can be sent back to the host. Once the calculation is complete, all OpenCL objects are freed (step 11). OpenCL implementations may be used in the same application with different implementations.

III. OPENCL PROGRAMMING MODEL

If the host CPU and any linked OpenCL "devices" employ the same language, programming interfaces, and hardware abstractions, then task- or data-parallel programs may run faster in a heterogeneous computing environment. OpenCL interfaces presume host and device heterogeneity, which is necessary since peripherals often do not share memory with the host CPU and utilize a distinct machine instruction set. OpenCL provides APIs for a variety of tasks, such as hardware discovery and management, memory allocation, data transfer between the host and device, compiling OpenCL programs and "kernel" functions for execution on target devices, launching kernels on target devices, querying execution progress and checking for errors, and launching kernels.

OpenCL applications may use either offline or in-process compilation to generate binary objects. The developer doesn't even need to have physical possession of the target device for their programs to function normally there. Even though instruction sets, drivers, and supporting libraries may be drastically altered between hardware generations, run-time compilation is unaffected.

Using OpenCL's run-time compilation capabilities ensures that your app always uses the most up-to-date libraries and libraries for the target device, without requiring you to recompile your program.

OpenCL is designed to work with a wide variety of microprocessor architectures, therefore it must be compatible with many different programming styles. Although OpenCL ensures kernels are portable and accurate across diverse hardware, it does not promise that any given kernel will run at its absolute fastest on every architecture. Different platforms may benefit from different programming techniques due to differences in the underlying hardware.

Creating a "context" for one or more devices is required before an application may do tasks such as compiling OpenCL code, allocating memory, or launching kernels. Instead of being tied to a certain hardware platform, memory allocations are context-aware. If devices with insufficient memory are included in the context creation process, the total amount of memory allocated will be capped at the capability of the least capable device. Similarly, if the devices that will be used to execute the OpenCL applications in the new context do not support certain features, those devices should be removed from the context.

OpenCL applications may be compiled dynamically by supplying the source code to OpenCL compilation methods as arrays of strings once a context has been constructed. Handles for individual "kernels" inside a built OpenCL application are then available. The OpenCL context allows the "kernel" functions to be executed on devices. In order to run on a target device, OpenCL hostdevice memory I/O operations and kernels must be enqueued into a command queue specific to that device.

IV. OPENCL AND MODERN PROCESSOR ARCHITECTURES

Existing programming languages do not adequately support, or even prevent the use of, some characteristics of modern microprocessor architecture. Companies have developed their own programming tools, language extensions, vector intrinsics, and subroutine libraries to compensate for the general lack of programmability in the hardware business. To further demonstrate the compatibility between the OpenCL programming paradigm and the wide range of available target hardware, we conduct an in-depth comparison of the architecture of three example microprocessor generations and draw connections between these features and core OpenCL abstractions and capabilities.

Multi-core CPUs

High-frequency processor cores are the building blocks of today's central processing units (CPUs), allowing for out-of-order execution and branch prediction. Central processing units (CPUs) are very adaptable because of the variety of tasks they may do concurrently. A high cache is necessary because the CPU cannot tolerate the latency of the main memory. For jobs that are time-sensitive yet have little parallelism, CPUs need huge caches. To improve the performance of intensive arithmetic and multimedia programs, many CPUs make limited use of single-instruction multiple-data (SIMD) arithmetic units. Developers of legacy languages like C and Fortran may resort to vectorized subroutine libraries, proprietary vector intrinsic functions, or rewrite source code and make use of auto vectorizing compilers to make up for the lack of SIMD units. OpenCL has been implemented by AMD, Apple, and IBM for multi-core CPUs and is compatible with SIMD instruction set extensions like x86 SSE and Power/VMX. Float4 types must be used explicitly during development for OpenCL kernels to take use of SSE on modern x86 processors. Many CPU implementations use a single hardware cache for all memory spaces, which may make it more expensive for a kernel to make extensive use of constant and local memory spaces than it would be for a kernel to make exclusive use of global memory references.

The Cell Processor

The Cell Broadband Engine design (CBEA) is a heterogeneous chip architecture that consists of many Synergistic Processor Elements (SPE), a Memory Interface Controller (MIC), and I/O units. The PPE is Power-compliant and has 64 bits of memory. The PPE is an IBM Power architecture-based general-purpose processor used to coordinate the work of SPEs by running standard operating systems and control-intensive applications. The SPE is an integral part of Cell systems; it is a SIMD streaming processor optimized for processing large amounts of data. Multiple SPEs may be used to realize the task parallelism of an application, and SIMD instructions and dual execution pipelines can be used to realize the data and instruction parallelism of SPEs. Each SPE has its own dedicated cache-like fast memory (called local store) that is maintained by the CPU. Transfer speeds are optimized when both the source and the destination are aligned to 128 bytes,

which is possible when applications employ DMA requests to transmit data between system memory and local storage.

Application developers may mask memory latency with methods like double buffering since data transmission and instruction execution can occur concurrently. The Cell processor's architecture and an example application's porting to it have been described in depth in the work by Shi et al.

In order to facilitate the use of Linux on the Cell and Power CPUs, IBM has created an OpenCL toolbox for Linux. IBM's OpenCL implementation incorporates software strategies to support the embedded profile, which is necessary due to the Cell SPUs' architectural differences from regular CPUs. Use global memory accesses with OpenCL float4 types or other operands that are multiples of 16 bytes for best performance on the Cell processor. When working with larger vector types, such as float16, performance may be further enhanced by using the compiler's unroll loops option. The 256 kilobytes of on-board memory of a Cell SPU are used for storing both the source code and "local" and "private" OpenCL variables. Since separate data storage is required for each assignment, the possible size of workgroups is reduced. The Cell DMA engine's full potential may be realized via the use of double buffering strategies in combination with asynchronous workgroup copy() operations to load data from global memory into local storage.

Graphics Processing Units

Modern GPUs with hundreds of processing units were designed for throughput-oriented, latency-insensitive applications. To hide the impact of global memory delay, GPUs include moderate-sized on-chip caches, enabling thousands of threads to run in parallel over the whole GPU. Conventional GPUs are organized as clusters of SIMD processing units managed by a central instruction decoder and using a shared, high-speed on-chip cache.

The machine instructions are executed simultaneously by the SIMD clusters, which also deal with branch divergence by keeping an eye on both forks of the branch and, if required, masking off the outputs of the processing units that aren't actively doing anything. Because of its SIMD design and in-order instruction execution, GPUs have more arithmetic units per square inch than CPUs do. OpenCL has been implemented on both AMD and NVIDIA graphics processing units (GPUs). An excessive amount of OpenGL work-items and work-groups are required for these gadgets to completely overload the hardware and conceal latency. Due to the scalar processor nature of the individual PEs, NVIDIA GPUs can efficiently handle most OpenCL data types. Float4 and other four-element vector types provide the best performance for OpenCL applications on AMD graphics processors because of their vector architecture. It is possible to optimize the resultant vectorized OpenCL kernel code for excellent performance on x86 CPUs and on AMD and NVIDIA GPUs, although the code is less legible than its scalar counterpart. Kernel optimality is impacted by many features of the GPU's low-level architecture, such as the amount of cached memory and the sorts of memory access that cause bank conflicts. The OpenCL documentation provided by your chosen vendor will likely provide some suggestions for low-level optimization.

The following code snippets are meant to illustrate fundamental OpenCL programming concepts rather than any specific implementations.

V. CONCLUSION

OpenCL stands as a robust and indispensable parallel programming paradigm, vital for tapping into the full potential of modern hardware architectures. Its significance extends beyond theoretical frameworks, as it empowers developers and researchers to address complex computational challenges across diverse domains. As we look to the future, OpenCL remains a cornerstone, supporting the ongoing evolution of parallel programming in an ever-changing technological landscape.

REFERENCES: -

1. Gioiosa, Roberto & Mutlu, Burcu & Lee, Seyong & Vetter, Jeffrey & Picierro, Giulio & Cesati, Marco. (2020). The Minos Computing Library: efficient parallel programming for extremely heterogeneous systems. 1-10. 10.1145/3366428.3380770.

2. Fang, Jianbin & Huang, Chun & Tang, Tao & Wang, Zheng. (2020). Parallel Programming Models for Heterogeneous Many-Cores : A Survey.

3. Fang, Jianbin & Huang, Chun & Tang, Tao & Wang, Zheng. (2020). Parallel Programming Models for Heterogeneous Many-Cores: A Comprehensive Survey. CCF Transactions on High Performance Computing. 2. 10.1007/s42514-020-00039-4.

4. Kaeli, David & Mistry, Perhaad & Schaa, Dana & Zhang, D.P.. (2015). Heterogeneous Computing with OpenCL 2.0: Third Edition.

5. Gaster, Benedict & Howes, Lee & Kaeli, David & Mistry, Perhaad & Schaa, Dana. (2013). Heterogeneous Computing with OpenCL. 10.1016/C2012-0-03322-4.

6. Nielsen, Allan & Engsig-Karup, Allan & Dammann, Bernd. (2012). Parallel Programming using OpenCL on Modern Architectures.

7. Gaster, Benedict & Howes, Lee & Kaeli, David & Mistry, Perhaad & Schaa, Dana. (2012). Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition.

8. Gaster, Benedict & Kaeli, David & Howes, Lee & Mistry, Perhaad. (2011). Heterogeneous Computing With OpenCL. 10.1016/C2011-0-69669-3.

9. Xu, J. (2011). OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems.

10. Barak, Amnon & Ben-Nun, Tal & Levy, Ely & Shiloh, Amnon. (2010). A package for OpenCL based heterogeneous computing on clusters with many GPU devices. 1 - 7. 10.1109/CLUSTERWKSP.2010.5613086.