mathematics

Review

Programming for High-Performance Computing on Edge Accelerators

Pilsung Kang

Department of Software Science, Dankook University, Yongin 16890, Republic of Korea; pilsungk@dankook.ac.kr

Abstract: The field of edge computing has grown considerably over the past few years, with applications in artificial intelligence and big data processing, particularly due to its powerful accelerators offering a large amount of hardware parallelism. As the computing power of the latest edge systems increases, applications of edge computing are being expanded to areas that have traditionally required substantially high-performant computing resources such as scientific computing. In this paper, we review the latest literature and present the current status of research for implementing high-performance computing (HPC) on edge devices equipped with parallel accelerators, focusing on software environments including programming models and benchmark methods. We also examine the applicability of existing approaches and discuss possible improvements necessary towards realizing HPC on modern edge systems.

Keywords: edge computing; parallel systems; high-performance computing; GPU (Graphics Processing Unit); accelerators; programming model; benchmarks

MSC: 68N01

Citation: Kang, P. Programming for High-Performance Computing on Edge Accelerators. *Mathematics* **2023**, *11*, 1055. https://doi.org/10.3390/math11041055

Academic Editor: Shadi G. Alawneh

Received: 30 December 2022
Revised: 14 February 2023
Accepted: 15 February 2023
Published: 20 February 2023

1. Introduction

Recently, the high-performance computing (HPC) community has overcome the limit of exa-scale (10^{18} floating point double precision operations per second) [1] computing and is now aiming to reach the next level of zeta-scale (10^{21} operations per second) computing. To accomplish this goal, innovative technology developments across the entire stack of HPC hardware and software environments are imperative in order to squeeze out as much performance as possible at every level of the stack. In particular, one of the main focuses is on cloud computing—an infrastructure that can flexibly scale up large-scale processing units and storage systems [2,3]. Accordingly, research on edge computing that can address the data bottleneck issue in the cloud environment is also active [4–7].

At the same time, recent edge devices are rapidly becoming powerful, to the extent that some of their performance manages to reach tens of TOPS (tera operations per second), mainly due to the development of parallel hardware architecture that offers a large amount of parallelism such as the GPU (graphics processing unit) accelerator [8]. Research on the use of this high-performance edge hardware is now gradually expanding from artificial intelligence (AI) and big data applications [9,10] to scientific calculations and simulations [11–14].

However, most of the research on edge systems is limited to applications of AI technology such as smart agriculture [15,16] or autonomous vehicles [17]. While there are a few instances of HPC applications of edge devices, they are mainly focused on the hardware level such as sensors for collecting high-resolution data or networking infrastructure [18] for transmitting data to the high-performance server. What is lacking is a solid software environment to effectively realize HPC from the powerful edge systems.

The current time seems to be an early stage of applying modern edge system's computing power to implementing scientific applications that require a large amount of computing resources. Hence, in the development of HPC on edge devices, it is essential to construct a software environment optimized for the edge device architecture. In this paper, we review the current status of edge computing in the context of HPC. Specifically, as illustrated in Figure 1, we examine the software environment for implementing HPC on the edge device with hardware accelerators, focusing on programming models and benchmark software used to develop HPC applications on modern edge systems.

We make the following contributions in this paper.

- We examine the current status of HPC research on edge devices from the software perspective. Specifically, we review the parallel programming models and benchmark tools that constitute the HPC environment for accelerator-based edge systems.
- We present potential directions for the software environment that can strengthen the development of HPC applications on edge. We discuss ideas to improve the models to program accelerator-based edge devices for better performance. We also consider more high-level programming models and accompanying benchmark software towards implementing HPC on edge.

The remainder of this paper is structured as follows. Section 2 reviews different HPC programming models used for accelerator-based edge systems and discusses performance issues specific for edge devices. Section 3 describes existing benchmark programs used in evaluating edge accelerators. Section 4 briefly discusses other research efforts related to implementing HPC on edge accelerators. Finally, Section 5 summarizes our work and makes conclusions.

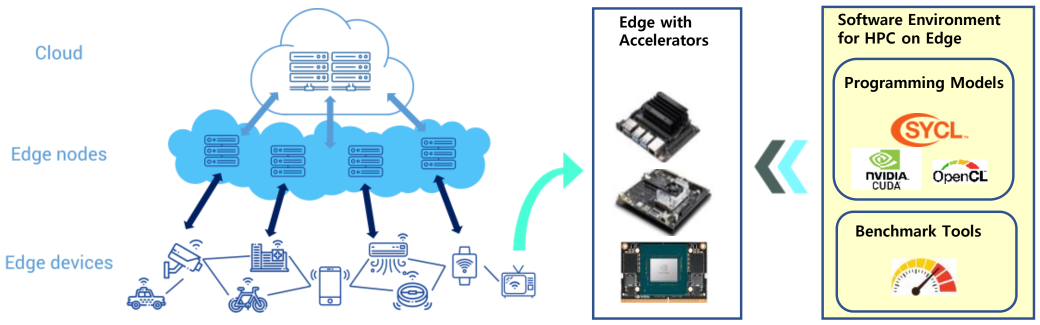

Figure 1. An overview of software environments for HPC on edge.

2. Programming Models for HPC on Edge Accelerators

A programming model is typically a language (or programming interfaces) with its runtime libraries for programming a computing system. A given computing system can be just a single compute device, a composite node of different devices, or a group of multiple compute nodes. Usually, traditional HPC programming models are applied to the edge computing environment; that is, MPI (Message-Passing Interface) is used for a group of networked edge nodes (distributed memory), and OpenCL [19] or NVidia CUDA [20] is used for accelerator-based devices.

In this section, we focus on the HPC models for programming a composite node of different devices because this type of model is mostly used for the accelerator-based edge devices, in which the CPU of an edge device acts as the “host” and the accelerator acts as the “device”. Table 1 is a brief summary of the different programming models considered in this paper for HPC on edge.

There are other “programming models” in a broader sense. For instance, Li and Dong [21] propose a programming model that implements runtime code offloading from the edge server to the client (device) to improve latency and code development productivity. However, their model is rather an execution model related to code distribution and execution between the edge server and the device at runtime and has a different orientation from the accelerator programming model for high performance, which is the subject of this study.

Table 1. Comparison of programming models for HPC on edge.

	CUDA	**OpenCL**	**SYCL**	**OpenMP**
Level of Expression	middle	low	high	high
Supported Architecture	NVidia GPUs	CPU/GPU/FPGA	CPU/GPU/FPGA	CPU/GPU
Framework Implementation	C/C++/Fortran extension	C/C++ extension	C++ extension	compiler directives
Open or Proprietary	proprietary	open	open	open
Major Feature	popularity for NVidia GPUs	heterogeneous system support	single-source model	ease of programming

2.1. NVidia CUDA

First introduced in 2007, NVIDIA CUDA is a parallel computing platform and programming model that allows software developers to use a GPU's computational power to accelerate applications. It includes a set of programming tools, libraries, and technologies that enable developers to write programs that can execute on NVIDIA GPUs to perform tasks such as scientific simulations, deep learning, and image and video processing. In support of scientific computing where floating point operations are fundamental, CUDA has traditionally supported FP32 (single precision) and FP64 (double precision). However, with the increasing needs for lower precision operations by mostly AI neural network applications, CUDA version 7.5 started supporting FP16 (half precision), a 16-bit floating point format standardized in 2008 by the IEEE [22].

Traditionally, in GPU-based high-performance systems, the memory used by the host CPU and the memory used by the device (GPU) are separated, so the programmer has to explicitly manage the address spaces of both the host and device to use GPU accelerators. This method of memory allocation and management can be called the traditional "memcopy" method, and is still largely used through the `cudaMalloc` and `cudaMemcpy` APIs (Application Programming Interfaces) of CUDA.

To address the inconvenience of managing distinct memories of the host and the device, NVidia CUDA version 6 started supporting the Unified Virtual Memory (UVM), which integrates two independent memories into one coherent address space. The UVM has since greatly simplified programming NVidia GPU systems via the `cudaMallocManaged` allocation API. However, in SoM (system-on-module) edge devices such as the Jetson series, one physical memory is typically shared between the host and the device, so there can be an unnecessary overhead of UVM, because UVM was originally intended for unifying physically separate memories. This overhead can become significant for those devices such as Jetson Nano that do not come with NVidia Compute Capability 7.2 or later with I/O coherency support that avoids CPU cache management operations when the physical memory is shared with the GPU.

NVidia CUDA offers another memory management method called pinned (or zero-copy) memory. Through the `cudaHostAlloc` and `cudaHostGetDevicePointer` APIs of this method, the GPU device can directly access the host memory allocated via DMA. Using this method can improve performance in SoM-based edge devices because the allocated memory at the host side is located in one physically shared memory. However, using this method under the UVM involves unnecessary steps of obtaining pointers via `cudaHostGetDevicePointer`, which can also incur unnecessary overhead.

There are a few reports about performance comparisons between the different memory allocation and management schemes of NVidia CUDA. Li et al. [23] presented one of the initial reports about the UVM performance on *physically unified memory* of embedded devices. They report a 10% performance loss on average for UVM compared to the traditional memcopy method on Jetson TK1, due to redundant memory transfers and page faults between CPU parts and GPU parts of the physically shared 2GB memory when UVM is used. In [24], the authors provide a similar report that, for a set of benchmark applications including 2D heat transfer simulation and adaptive numerical integration, the UVM mechanism performs worse by up to 8% than the traditional memcopy method, while the UVM scheme enables simpler programming. Since their evaluation was performed on NVIDIA desktop GPUs such as GTX 970 (released in 2015) which has independent memory units between the host and the device, the evaluation results would not directly translate to the SoM-based edge devices.

In [25], Choi et al. provide a more recent evaluation of the performance of different memory management schemes of CUDA (version 9) for the NVidia Jetson TX2 (Pascal architecture) with a set of common HPC benchmarks such as vector calculation and CFD (computational fluid dynamics) programs. In contrast to other evaluation efforts, they examine not only the execution time of benchmark programs but also the amount of used memory by each memory management scheme during the benchmark runtime. Based on the experimental results, they conclude that the UVM scheme performs efficiently across different application benchmarks by avoiding the memory copy overhead. In addition, applications with good temporal locality can benefit most from the UVM scheme in terms of memory usage, while the pinned memory can show better execution time depending on the used applications. There report indicates that more recent versions of CUDA provide improved performance by taking into account the SoM-based systems. In [26], the authors provide an in-depth analysis of the UVM system on the high-end Titan V GPU using CUDA 11.2. Their performance analysis includes various components such as data movement, page faults, and prefetching and emphasizes that data movement contributes relatively little to the overall cost, in contrast to expectation. Instead, they report that appropriately managing page faults in UVM is more fundamental to performance. Although their evaluation is based on a discrete GPU with separate memory, it provides compelling insights into the performance behavior of the CUDA UVM for edge devices.

Discussions

To fully exploit the architectural features of the NVidia SoM device with the GPU accelerator, the UVM can be optimized to remove the overhead of UVM's method of data movement between the host and device. To actually implement the optimization, the programmer can first examine the inner workings of the `cudaMallocManaged` function and its execution flow along with the device PTX (Parallel Thread eXecution) code, so that the optimization points can be identified across the execution path of the on-demand page transfer in the CUDA driver. In addition, the source of NVidia's previously published GPU driver module [27,28] can be reviewed for analysis and relevant code can be identified. In a similar manner, the `cudaHostGetDevicePointer` function call path for zero-copy operation can be examined for the further optimization of unnecessary operations between the host and the device in managing the shared memory.

Lastly, the traditional memcopy scheme can be improved as well. The memory allocation and transfer model between host and device that has been used since the early days of NVidia CUDA allocates space separately to each separated memory and transfers data. Even after UVM was introduced, it continues to be used for CUDA programming, and there are codes implemented as an early programming model. In SoM-based edge devices, one memory is shared between hosts and devices, so memory allocation and data movement in this way incur unnecessary overhead. Hence, there is room for improvements for the old memory allocation and management scheme by examining the old APIs such as `cudaMalloc`, `cudaMemcpy`, and `cudaFree` functions to avoid unnecessary overhead in SoM architecture. For the legacy functions, the NVidia PTX machine code can be analyzed to check the execution path at the CUDA driver level. Overall, Figure 2 shows the possible improvements of CUDA with respect to the memory model towards realizing HPC on the SoM-based edge devices.

2.2. OpenCL

OpenCL^{TM} (Open Computing Language) is an open standard for programming applications on parallel systems that comprise different types of hardware architectures. It was first proposed by Apple in 2008, with the goal of creating an open standard for parallel programming on heterogeneous devices. The initial proposal was later merged with a similar proposal from other companies, including AMD, IBM, and Intel, to form the Khronos Group's OpenCL working group. The first version of OpenCL was released in 2009, and it has since been updated several times, with the latest version being OpenCL 3.0. Over the years, OpenCL has been widely adopted in various fields, such as scientific computing, computer vision, and machine learning.

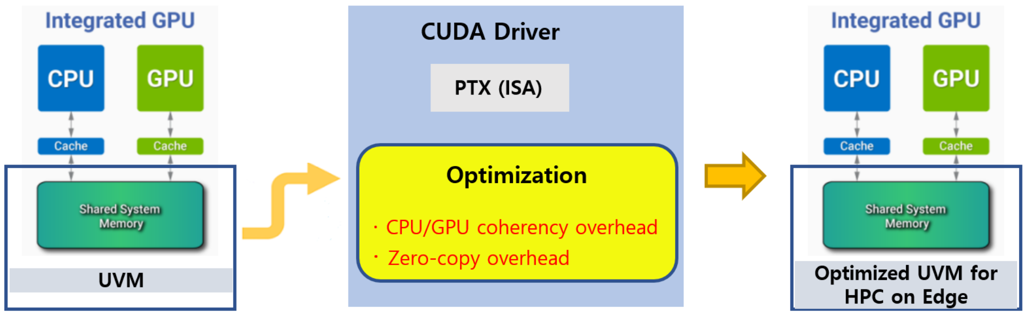

Figure 2. UVM (Unified Virtual Memory) optimization of the CUDA model for HPC on edge.

One of the most salient features of OpenCL is its ability to provide functional portability, enabling a single codebase to be utilized on a wide range of hardware and architectural configurations. OpenCL is defined as an extension of the C and C++ programming languages, and APIs are also provided to orchestrate the execution of programs on the device.

The OpenCL architecture comprises a host system that communicates with one or more *compute devices*. Each compute device encompasses one or more *compute units*, which are further subdivided into one or more *processing elements* (*PEs*) for parallel processing. As an exemplar, when utilizing an NVidia GPU card in the OpenCL framework, the card is classified as a compute device, the streaming multiprocessors (SMs) are considered as compute units, and the CUDA cores are identified as PEs. The fundamental unit of work within the device is referred to as a *work-item*, which is mapped to a specific PE. These work-items are then grouped together to form a *workgroup*, the size and dimensionality of which are determined by the programmer. The code that specifies the execution of a work-item on a PE is referred to as a *kernel*. Hence, when a kernel is launched on a compute device, it is executed in a concurrent manner across all the PEs within the workgroup, allowing for seamless parallel processing.

Figure 3 shows a logical view of the memory hierarchy as seen by the conventional OpenCL program. A compute device possesses a shared memory space, referred to as *global memory*, that can be accessed and utilized by all the workgroups operating within the device. To perform any calculation on a compute device, the (initial) data need to be moved from the host to the device first. Later, when the computation is done, the final data on the device need to be moved back from the device to the host. Global memory is the slowest type of memory in the device.

Each workgroup is equipped with its own private memory space, referred to as *local memory*, that is shared among all the PEs within the group and provides faster access than global memory. Additionally, each thread has its own dedicated storage area, known as *private memory*, which is the fastest among all the memory types. Similar to other programming models for heterogeneous systems, optimizing memory access and effectively utilizing the memory hierarchy of the memory model in OpenCL is crucial to achieving optimal performance in an application.

Although OpenCL provides a unified interface for various heterogeneous architectures such as CPU, GPU, and FPGA, the OpenCL memory model separates host memory and device memory, making it unsuitable for SoM-based edge devices where a single physical memory is shared between the host and the compute device. If the OpenCL memory model is mapped and operated as it is to the SoM-based edge device, the memory model does not fit well to the device, which can result in unfavorable overhead due to unnecessary data movements within the device. For instance, when programming NVidia's edge devices such as Jetson using OpenCL, the compiler uses the CUDA runtime as the backend, which in turn may use a separate address space between the host and the device without taking advantage of the shared physical memory.

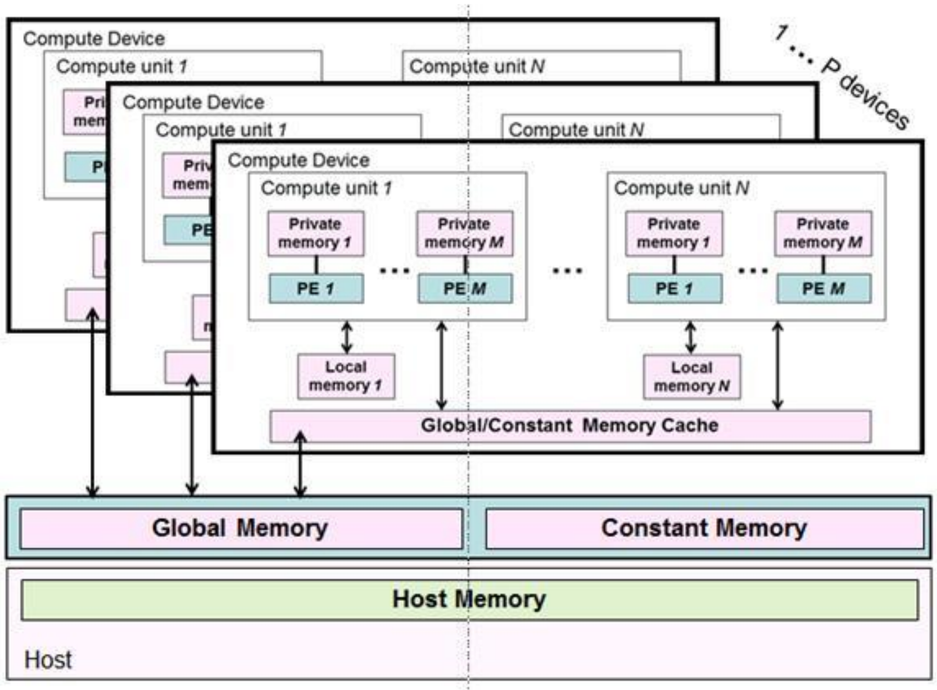

Figure 3. OpenCL memory model [29].

Since version 2.0, the OpenCL specification provides Shared Virtual Memory (SVM) [29] that allows the device to directly access host-allocated buffers, enabling the host and the device to share pointer-based data structures in OpenCL kernels. Similar to NVidia CUDA's UVM, OpenCL's SVM mechanism greatly simplifies programming accelerator-based systems by removing the inconvenience of explicitly specifying memory allocation and data movement across the host and the device as well as its accompanying overhead. However, this approach can introduce memory coherency problems between the host and the device when accessing the same shared memory buffer, thus leading to adverse performance degradation [30].

As concerns the floating point storage format, OpenCL includes FP16 as an optional extension in addition to FP32 and FP64 from version 1.0. In terms of the programming language, OpenCL 2.2 introduced C++ support in 2017 besides C for writing kernel code for enhanced parallel programming productivity with high-level abstractions when programming HPC applications.

Discussions

The OpenCL model may update its specification in the future to consider physically shared memory between the host and the device, which is common among the SoM-based edge systems. In the meantime, there can be improvements at the backend for OpenCL programs running on the SoM-based systems. For instance, memory allocation and data movement APIs of OpenCL such as `clCreateBuffer`, `clEnqueueWriteBuffer`, and `clEnqueueReadBuffer` can be translated to avoid redundant data movements within the shared memory at execution. For NVidia edge devices, the CUDA compiler or the backed driver can be modified for these OpenCL function calls for optimized operations in a consistent way. Open source implementations such as PoCL (Portable Computing Language) [31] can be updated to support the memory hierarchy of the SoM-based edge systems, such that unnecessary memory allocation and data movements are removed for these systems.

2.3. SYCL

NVidia CUDA preemptively implemented accelerator-based high-performance computing based on massively parallel GPU hardware and is currently the most widely used GPU computing model in the HPC field. However, since CUDA is proprietary NVidia technology, many heterogeneous architectures cannot use CUDA directly. OpenCL overcomes these drawbacks and provides a consistent programming model for heterogeneous architectures, but its low-level API places the burden on programmers to explicitly describe all system behavior. SYCL, established by the Khronos standards group, provides a single-source programming model by abstracting OpenCL [32] using the C++17 semantics. Therefore, SYCL has the potential to become a more programmer-friendly option with its high-level expressive power.

Burns et al. [33] studied OpenCL and SYCL models for RISC-V architecture, which was introduced in 2010 but is under active research in edge computing due to its architectural characteristics for low-power and parallel performance using vectorized instructions [34]. Their work is a joint project with Kyoto University towards an integrated programming model for RISC-V vector instructions. Kuncham et al. [35] report comparative evaluation results of the SYCL and CUDA models, showing similar performance behavior on the NVidia's high-performance GPU cards such as V100. SYCL has the advantage of widely supporting heterogeneous architectures, but its performance in edge systems needs to be confirmed.

Discussions

We expect that the SYCL layer may be implemented on top of an OpenCL backend that is optimized for the SoM-based edge device. Since an LLVM-based SYCL compiler source is available [36], we also expect that high-performance applications written in SYCL can be translated to generate highly optimized code based on an OpenCL backend for edge devices regarding host side and device side memory allocation along with data movement operations between them.

2.4. OpenMP

OpenMP [37] is a widely-adopted model for programming shared memory parallel systems. It supports the C, C++, and Fortran languages with a set of compiler directives that enable users to specify parallelism in an easy manner for different types of tasks such as work-sharing, synchronization, and data environment. OpenMP's directive-based constructs are simple to use and especially effective for expressing data-parallel algorithms compared to other programming models. It also provides runtime library routines that can be used for a variety of purposes in controlling the execution of parallel programs, which includes setting and querying the number of threads, nested parallelism, and dynamic thread features.

Traditionally, OpenMP has been used to program multi-CPU or multi-core systems, where multiple threads are generated and concurrently executed to cooperatively solve a large computational problem. However, heterogeneous parallel programming with different types of accelerator processors has become possible with the introduction of accelerator offloading since OpenMP version 4.0. For example, the programmer can specify the target GPU device to offload the region of computation by using the `omp target` and other supporting directives. Furthermore, OpenMP 5.0 introduced support for unified shared memory between the accelerator devices and the CPU host in a hybrid system, which is equivalent to NVidia's UVM in that the host and the device can access each other's memory in a unified address space without explicitly copying data between them. In OpenMP 5.0, it is also possible to use device-specific function implementations and override device offloading during runtime, thereby fully supporting accelerator devices.

However, there are only a handful of research works on HPC on edge systems using the OpenMP model. Chapman et al. [38] adapt the existing OpenMP model to enable the specification of high-performance applications on embedded systems such as MPSoC (multi-processor system on a chip). The particular requirements needed for applying

OpenMP to embedded systems are primarily the ability to express nested parallelism and the constraints of real-time and resource management. They implement OpenMP with the requirements and evaluate the implementation on the Texas Instruments TMS320C64x+ multi-core processor. Their work is one of the early efforts in applying the conventional OpenMP model to high-performant embedded systems. Liang et al. [39] implemented a mobile programming environment for developing OpenMP applications that can run on the CPU and the GPU of mobile devices, so that scientific research can be performed with the ease of mobility of devices such as cell phones. Their work allows one to translate an OpenMP source code into a combination of a pthread module for the CPU and an OpenCL kernel for the GPU of the mobile device.

Gayatri et al. [40] investigate the performance portability of OpenMP across different architectures of CPUs and GPUs. They implement a material science application in OpenMP and evaluate the application performance on different CPU architectures including IBM Power, Intel Haswell, and Xeon Phi systems, and on different GPU architectures including NVidia P100 and V100 systems. Their evaluation results show that the GPU offloading scheme of OpenMP 4.5 shows comparable performance against OpenACC [41], which is an alternative heterogeneous parallel programming model based on compiler directives. However, they observe that the same OpenMP code optimized for GPU is ill-suited for CPU execution, which necessitates a different optimization strategy for CPUs.

Liang et al. [42] use OpenMP to parallelize erasure coding, which is a method of data protection in storage systems, on the quad-core ARM Cortex-A57 processor of Jetson Nano, so that the throughput of encoding and decoding operations of erasure coding can fully exploit the network bandwidth of 5G and Wi-Fi 6, while effectively reducing the latency for small-sized data at the network edge. Their work can be regarded as one of the few OpenMP applications where parallelization is performed for the CPU instead of the parallel accelerators of the modern edge devices.

Discussions

OpenMP has been traditionally used only for programming shared memory systems with its fork-join style of parallel thread execution, and support for accelerator offloading introduced in OpenMP 4.0 is relatively new compared to other programming models such as OpenCL. Therefore, as observed in [40], conventional programming patterns and optimization techniques for CPU-based systems can be inadequate for maximizing the accelerator performance. As a result, more in-depth examinations on the performance behavior of accelerator offloading such as [43] are required for wider adoption of OpenMP for programming heterogeneous systems. This argument applies to accelerator-based edge devices as well, where there are many unclear aspects of using OpenMP such as performance portability.

3. Benchmarks for HPC on Edge Accelerators

In the field of parallel computing, the NPB benchmark [44] suite derived from CFD simulations has traditionally been used, and reference versions implemented with major parallel programming models such as MPI and OpenMP have been available for quite a while [45]. The NPB benchmark originally consisted of five major numerical kernels such as conjugate gradient (CG) and Fourier Transform (FT), and NPB also offered three pseudo-applications such as the Lower–Upper symmetric Gauss–Seidel (LU) solver performing matrix-based operations, which are key tools in computational science. Later, from NPB 3.1, other benchmark components were gradually added such as Data Cube (DC) based on a data mining application [46]. In addition, there are recent efforts to implement not only the main NPB kernel but also other numerical solvers using the CUDA and the OpenCL models for heterogeneous architectures. In this section, we briefly review research works on benchmarks in the perspectives of HPC on edge.

Varghese et al. classify various benchmark methods for edge systems identifying different target systems under test, analysis methods, and benchmark runtimes [47]. According to the survey, most edge computing benchmarks are centered around AI technology and its applications, and traditional methods such as NPB are usual for evaluating edge systems for HPC applications.

Seo et al. [48] implemented SNU-NPB, an NPB benchmark suite written in OpenCL, and analyzed its performance. They observe that the performance characteristics of NPB in OpenCL are quite different from the OpenMP version based on their experimental results, concluding that optimizations for different OpenCL compute devices are required for better performance. According to their report, like most other standardized programming models, OpenCL's advantage of source level portability across different architectures can harm performance portability.

Do et al. [49] developed an improved version of SNU-NPB, called SNU-NPB 2019. In SNU-NPB 2019, they not only rewrite their previous OpenCL version of the NPB but also provide a CUDA implementation by applying a group of 15 parallelization and optimization techniques for latest GPU architectures. Applied optimization techniques include loop parallelization, global memory access coalescing, and communication/computation overlapping. In addition, compared to the previous SNU-NPB, large problem sizes such as classes D and E are covered, and GPU memory capacity limitations in the previous implementation have been corrected. As a result, they report order-of-magnitude improvements of SNU-NPB 2019 over the previous version in terms of performance for many NPB benchmark kernels. Their work evaluates the performance of high-performance GPUs from NVidia and AMD, and the presented optimization techniques can be useful for optimizing the NPB benchmarks on edge systems equipped with latest accelerators.

In [50], the authors apply NVidia CUDA to implement the NPB benchmark kernels and pseudo-applications and report up to 267% performance improvements compared to SNU-NPB 2019 for a few NPB components on high-end GPU systems such as NVidia Titan X and V100 Volta. Although the authors adopted a large amount of the optimization techniques presented by SNU-NPB 2019 in implementing their CUDA C++ version of the NPB benchmark, they rewrote the main computations to allow a higher degree of parallelism and simplify the instruction flow for better performance. In addition, their implementation provides parametrization support for configuring the number of threads per CUDA block, which can be useful in terms of programming flexibility. Still, whether similar performance improvements can be observed for GPU-based edge systems as well as high-end GPU systems needs careful examination.

Kang and Lim [51] evaluate GPU accelerator-based edge systems from the perspective of HPC applications. Although the performance of the edge device itself is slow, they observe that the performance of a small cluster of networked edge devices can be better than a multi-core CPU in terms of the price–performance ratio. Their evaluation report can be a useful hint that the vertical optimization of memory hierarchy is necessary to improve performance for SoM-based systems. Table 2 summarizes research works related to the NPB benchmarks for HPC on the edge environments.

Discussions

Although there are standard benchmark tools such as NPB for parallel systems, the programming models used to implement the benchmarks are targeted to traditional HPC systems. These HPC systems have a different memory hierarchy than major edge systems that are usually organized in the SoM form factor where a single physical memory is shared between the CPU and the accelerator. Hence, it is unclear if these benchmark tools can accurately measure the edge system's performance. In addition, there is a growing demand for standard benchmarks written in a high-level programming model for heterogeneous systems including the edge devices. Hence, a benchmark suite written in the SYCL model will be highly beneficial to the community due to its highly expressive power with a single source that can cover a wide variety of different hardware architectures, including the accelerator-based edge systems. Furthermore, an OpenCL or a CUDA backend optimized for edge devices can be supported with the SYCL APIs. Altogether, the standard NPB benchmark suite constructed with a high-performant backend will be greatly favorable for evaluating latest versions of edge accelerators from various perspectives, such as memory random access, inter-thread communication, and massive parallelism.

Table 2. Summary of NPB Benchmark Research for HPC on Edge.

Research	Model	Year	Benchmark Target	Major Contributions
Seo [48]	OpenCL	2011	High-end GPUs	• Implements NPB in CUDA C • Observes OpenCL's source-level portability harms performance portability
Do [49]	OpenCL & CUDA	2019	High-end GPUs	• Implements NPB in CUDA C & OpenCL • Presents 15 parallelization and optimization techniques for GPU
Kang [51]	CUDA	2020	Edge GPUs	• Applies NPB to evaluate edge GPU • Observes edge cluster can be effective for HPC in price-performance perspective
Arajuo [50]	CUDA	2021	High-end GPUs	• Implements NPB in CUDA C++ • Defines design principles for programming GPU applications in HPC

4. Other Research Works Related to HPC on Edge

In this section, we briefly review a few other research works for constructing the software environment towards HPC on edge. In [14], floating point representation is studied to implement scientific calculations requiring high precision in an edge system. The posit type allows the system to offer more precision and a greater dynamic range with a given number of bits compared to the popular IEEE-754 standard. However, it must be supported by the hardware for widespread adoption, and it is yet unclear when major edge system vendors will provide support for the posit number system. In [52], the authors implement a dense matrix multiplication operation in VPU (Vision Processing Unit) on Intel's edge system. Although matrix calculations are common in HPC applications, their work is focused on AI applications using neural network computations.

Cecilia et al. [13] evaluate the performance behavior of an edge system compared to an HPC desktop system by executing computationally intensive applications such as clustering algorithms. They evaluate an Nvidia's AGX Xavier device and report a significant amount of energy savings up to 150% compared to the HPC system, although it performs 11× slower. Their work shows that edge systems can be useful for some computationally heavy workloads depending on applications.

5. Summary and Conclusions

As the HPC community is striving for innovative technology that can achieve extreme performance for the entire system stack to realize a zeta-scale HPC environment, edge systems at the cloud environment have great potential in HPC applications due to the accelerators offering a substantial amount of parallelism. In this study, we considered the relevant literature dealing with implementing HPC on the edge platforms, focusing on the programming models and benchmark applications that constitute the software development environments. Major studies on edge computing from the perspective of HPC applications can be summarized as follows. Low-power, high-performance architectures such as GPUs and RISC-V processors are under active research and development, but research on software environments such as programming models, benchmark tools for evaluation, and compiler techniques for performance optimization is still in its nascent stage.

We also discussed how existing HPC programming models can be improved further for better performance. The programming model for accelerator-based edge systems is significant for optimized performance. However, traditional HPC methods are being applied to the edge systems without fully considering the different architectural characteristics of the major edge systems. Most of all, careful consideration of the memory model is essential for programming models to achieve highly optimized performance from the state-of-the-art edge systems.

Recently, support from cloud companies is strengthening to provide high-performance, low-latency edge services such as AWS for the Edge [53] and Azure Edge Zone [54] at the enterprise level. These changes suggest that novel computing environments are being established in which applications of edge systems, which have been somewhat limited to the AI and big data processing areas, are being expanded. We expect that HPC applications based on edge platforms will be provided as cloud services in the foreseeable future, for which support for adequate programming environments for edge devices is crucial for high performance.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT), grant number 2020R1F1A1067619.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Frontier Supercomputer Debuts as World’s Fastest, Breaking Exascale Barrier. 2022. Available online: https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier (accessed on 25 December 2022).
2. Guidi, G.; Ellis, M.; Buluç, A.; Yelick, K.; Culler, D. 10 Years Later: Cloud Computing is Closing the Performance Gap. In Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering, Virtual, 19–23 April 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 41–48. [CrossRef]
3. Reed, D.; Gannon, D.; Dongarra, J. Reinventing High Performance Computing: Challenges and Opportunities. *arXiv* **2022**, arXiv:2203.02544.
4. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
5. Varghese, B.; Wang, N.; Barbhuiya, S.; Kilpatrick, P.; Nikolopoulos, D.S. Challenges and Opportunities in Edge Computing. In Proceedings of the 2016 IEEE International Conference on Smart Cloud (SmartCloud), New York, NY, USA, 18–20 November 2016; pp. 20–26. [CrossRef]
6. Cao, K.; Liu, Y.; Meng, G.; Sun, Q. An Overview on Edge Computing Research. *IEEE Access* **2020**, *8*, 85714–85728. [CrossRef]
7. Saraf, P.D.; Bartere, M.M.; Lokulwar, P.P. A Review on Evolution of Architectures, Services, and Applications in Computing Towards Edge Computing. In *International Conference on Innovative Computing and Communications*; Springer: Singapore, 2022; pp. 733–744.
8. Karumbunathan, L. Solving Entry-Level Edge AI Challenges with NVIDIA Jetson Orin Nano. 2022. Available online: https://developer.nvidia.com/blog/solving-entry-level-edge-ai-challenges-with-nvidia-jetson-orin-nano (accessed on 25 December 2022).
9. Li, E.; Zeng, L.; Zhou, Z.; Chen, X. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Trans. Wirel. Commun.* **2020**, *19*, 447–457. [CrossRef]
10. Chen, J.; Ran, X. Deep Learning With Edge Computing: A Review. *Proc. IEEE* **2019**, *107*, 1655–1674. [CrossRef]
11. Riha, L.; Le Moigne, J.; El-Ghazawi, T. Optimization of Selected Remote Sensing Algorithms for Many-Core Architectures. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2016**, *9*, 5576–5587. [CrossRef]
12. Tu, W.; Pop, F.; Jia, W.; Wu, J.; Iacono, M. High-Performance Computing in Edge Computing Networks. *J. Parallel Distrib. Comput.* **2019**, *123*, 230. [CrossRef]
13. Cecilia, J.M.; Cano, J.C.; Morales-Garcia, J.; Llanes, A.; Imbernon, B. Evaluation of Clustering Algorithms on GPU-Based Edge Computing Platforms. *Sensors* **2020**, *20*, 6335. [CrossRef]
14. Poulos, A.; McKee, S.A.; Calhoun, J.C. Posits and the State of Numerical Representations in the Age of Exascale and Edge Computing. *Softw. Pract. Exp.* **2022**, *52*, 619–635. [CrossRef]
15. Zamora-Izquierdo, M.A.; Santa, J.; Martínez, J.A.; Martínez, V.; Skarmeta, A.F. Smart Farming IoT Platform based on Edge and Cloud Computing. *Biosyst. Eng.* **2019**, *177*, 4–17. [CrossRef]
16. Kalyani, Y.; Collier, R. A Systematic Survey on the Role of Cloud, Fog, and Edge Computing Combination in Smart Agriculture. *Sensors* **2021**, *21*, 5922. [CrossRef] [PubMed]
17. Liu, S.; Liu, L.; Tang, J.; Yu, B.; Wang, Y.; Shi, W. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proc. IEEE* **2019**, *107*, 1697–1716. [CrossRef]
18. Facing the Edge Data Challenge with HPC + AI. 2022. Available online: https://developer.nvidia.com/blog/facing-the-edge-data-challenge-with-hpc-ai (accessed on 25 December 2022).
19. Stone, J.E.; Gohara, D.; Shi, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Comput. Sci. Eng.* **2010**, *12*, 66–73. [CrossRef] [PubMed]
20. Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. Scalable Parallel Programming with CUDA. *Queue* **2008**, *6*, 40–53. [CrossRef]

21. Li, B.; Dong, W. EdgeProg: Edge-centric Programming for IoT Applications. In Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), Singapore, 29 November–1 December 2020; pp. 212–222. [CrossRef]
22. IEEE. *IEEE Standard for Floating-Point Arithmetic*; IEEE Std 754-2008; IEEE: Piscataway, NJ, USA, 2008; pp. 1–70. [CrossRef]
23. Li, W.; Jin, G.; Cui, X.; See, S. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen, China, 4–7 May 2015; pp. 1092–1098. [CrossRef]
24. Jarząbek, Z.; Czarnul, P. Performance Evaluation of Unified Memory and Dynamic Parallelism for Selected Parallel CUDA Applications. *J. Supercomput.* **2017**, *73*, 5378–5401. [CrossRef]
25. Choi, J.; You, H.; Kim, C.; Young Yeom, H.; Kim, Y. Comparing Unified, Pinned, and Host/Device Memory Allocations for Memory-intensive Workloads on Tegra SoC. *Concurr. Comput. Pract. Exp.* **2021**, *33*, e6018. [CrossRef]
26. Allen, T.; Ge, R. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, MO, USA, 14–19 November 2021; Association for Computing Machinery: New York, NY, USA, 2021. [CrossRef]
27. NVidia. Accelerated Linux Graphics Driver README and Installation Guide, Chapter 44. Open Linux Kernel Modules. 2022. Available online: http://download.nvidia.com/XFree86/Linux-x86_64/515.43.04/README/kernel_open.html (accessed on 25 December 2022).
28. NVidia. NVIDIA Linux Open GPU Kernel Module Source. 2022. Available online: https://github.com/NVIDIA/open-gpu-kernel-modules (accessed on 25 December 2022).
29. Khronos OpenCL Working Group. The OpenCL Specification Version v3.0.12. 2022. Available online: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf (accessed on 25 December 2022).
30. Cavicchioli, R.; Capodieci, N.; Bertogna, M. Memory Interference Characterization between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms. In Proceedings of the 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, Cyprus, 12–15 September 2017; pp. 1–10. [CrossRef]
31. Portable Computing Language (PoCL). Available online: http://portablecl.org (accessed on 25 December 2022).
32. Khronos OpenCL Working Group. SYCL 1.2.1 Specification. 2022. Available online: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf (accessed on 25 December 2022).
33. Burns, R.; Davidson, C.; Dodds, A. Enabling OpenCL and SYCL for RISC-V Processors. In Proceedings of the International Workshop on OpenCL, Munich, Germany, 27–29 April 2021; Association for Computing Machinery: New York, NY, USA, 2021. [CrossRef]
34. Asanović, K.; Patterson, D.A. *Instruction Sets Should Be Free: The Case For RISC-V*; Technical Report UCB/EECS-2014-146; EECS Department, University of California, Berkeley: Berkeley, CA, USA, 2014.
35. Reddy Kuncham, G.K.; Vaidya, R.; Barve, M. Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU. In Proceedings of the 2021 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 20–24 September 2021; pp. 1–7. [CrossRef]
36. Intel DPC++ SYCL for CUDA User Manual. Available online: https://github.com/intel/llvm/blob/sycl/sycl/doc/UsersManual.md (accessed on 25 December 2022).
37. Dagum, L.; Menon, R. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46–55. [CrossRef]
38. Chapman, B.; Huang, L.; Biscondi, E.; Stotzer, E.; Shrivastava, A.; Gatherer, A. Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing, Rome, Italy, 23–29 May 2009; pp. 1–8. [CrossRef]
39. Liang, T.Y.; Li, H.F.; Chen, Y.C. An OpenMP Programming Environment on Mobile Devices. *Mob. Inf. Syst.* **2016**, *2016*, 4513486. [CrossRef]
40. Gayatri, R.; Yang, C.; Kurth, T.; Deslippe, J. A Case Study for Performance Portability Using OpenMP 4.5. In *Accelerator Programming Using Directives, Proceedings of the 5th International Workshop, WACCPD 2018, Dallas, TX, USA, 11–17 November 2018*; Springer: Cham, Switzerland, 2019; pp. 75–95.
41. OpenACC. Available online: http://www.openacc-standard.org (accessed on 25 December 2022).
42. Liang, L.; He, H.; Zhao, J.; Liu, C.; Luo, Q.; Chu, X. An Erasure-Coded Storage System for Edge Computing. *IEEE Access* **2020**, *8*, 96271–96283. [CrossRef]
43. Huber, J.; Cornelius, M.; Georgakoudis, G.; Tian, S.; Diaz, J.M.M.; Dinel, K.; Chapman, B.; Doerfert, J. Efficient Execution of OpenMP on GPUs. In Proceedings of the 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Seoul, Republic of Korea, 2–6 April 2022; pp. 41–52. [CrossRef]
44. Bailey, D.; Barszcz, E.; Barton, J.; Browning, D.; Carter, R.; Dagum, L.; Fatoohi, R.; Frederickson, P.; Lasinski, T.; Schreiber, R.; et al. The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.* **1991**, *5*, 63–73. [CrossRef]
45. NAS Parallel Benchmarks. Available online: https://www.nas.nasa.gov/software/npb.html (accessed on 25 December 2022).
46. NAS Parallel Benchmarks Changes. Available online: https://www.nas.nasa.gov/software/npb_changes.html (accessed on 25 December 2022).
47. Varghese, B.; Wang, N.; Bermbach, D.; Hong, C.H.; Lara, E.D.; Shi, W.; Stewart, C. A Survey on Edge Performance Benchmarking. *ACM Comput. Surv.* **2021**, *54*, 1–33. [CrossRef]

48. Seo, S.; Jo, G.; Lee, J. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 6–8 November 2011; pp. 137–148. [CrossRef]
49. Do, Y.; Kim, H.; Oh, P.; Park, D.; Lee, J. SNU-NPB 2019: Parallelizing and Optimizing NPB in OpenCL and CUDA for Modern GPUs. In Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC), Orlando, FL, USA, 3–5 November 2019; pp. 93–105. [CrossRef]
50. Araujo, G.; Griebler, D.; Rockenbach, D.A.; Danelutto, M.; Fernandes, L.G. NAS Parallel Benchmarks with CUDA and beyond. *Softw. Pract. Exp.* **2021**, *53*, 53–80. [CrossRef]
51. Kang, P.; Lim, S. A Taste of Scientific Computing on the GPU-Accelerated Edge Device. *IEEE Access* **2020**, *8*, 208337–208347. [CrossRef]
52. Ionica, M.H.; Gregg, D. The Movidius Myriad Architecture's Potential for Scientific Computing. *IEEE Micro* **2015**, *35*, 6–14. [CrossRef]
53. Amazon AWS for the Edge. Available online: https://aws.amazon.com/edge (accessed on 25 December 2022).
54. Microsoft Azure Edge Zone. Available online: https://docs.microsoft.com/azure/networking/edge-zones-overview (accessed on 25 December 2022).