Comparative Study of Directive-based Programming Models on CPUs and GPUs for Scientific Applications

C. Navya, H. A. Sanjay, and Sanket Salvi

1 Introduction

The world of high performance computing is an interesting field of study from two decades. As the need for large and complex applications have been increasing across many industries, the need for high performance computing has also increased. With the need to give more assets to the users, there is the chance of utilizing all equipment assets proficiently. However, the CPU technology is not capable of scaling in performance sufficiently to address the demand. GPUs can give amazing execution by utilizing the available GPU cores. Graphics processing unit (GPU) can be used for parallel programming. Parallel programming is a process of dividing complex computational tasks into smaller tasks that can run concurrently. The main objective of HPC is to achieve better performance for large and complex scientific applications by adopting parallel programming paradigm. Multi-cores CPUs will improve the performance of the HPC application up to some extent and cannot be proficiently extrapolated forward in time. In both cases, it is difficult to overcome the critical part of the program to use the computational force efficiently.

For heterogeneous parallel programming, directive-based accelerators are used. Compared to low-level programming (PThreads, CUDA, and OpenCL) using directives-based language such as OpenMP and OpenACC. For parallel programming, OpenACC is the new programming standard, where application program interface characterizes a gathering of compiler orders to depict circles and locales of

C. Navya (✉) · H. A. Sanjay · S. Salvi
Nitte Meenakshi Institute of Technology, Bangalore, India
e-mail: navya.c@nmit.ac.in

H. A. Sanjay
e-mail: sanjay.ha@nmit.ac.in

S. Salvi
e-mail: sanket.salvi@nmit.ac.in

N. R. Shetty et al. (eds.), *Emerging Research in Computing, Information, Communication and Applications*, Lecture Notes in Electrical Engineering 790,
https://doi.org/10.1007/978-981-16-1342-5_61

code in FORTRAN, C, and C + + with the goal that they can be offloaded from CPU to quickening agent, which gives versatility across working frameworks, CPUs, and accelerators. OpenACC is a programming standard for parallel programming developed by PGI, Cray, CAPS, and Nvidia.

The main objective of this work is to utilize the available GPUs efficiently for parallel programming and to compare the time taken for executing scientific applications on CPUs and GPUs using OpenACC and OpenMP. The work focuses on parallelizing benchmark applications like fast Fourier transform, Laplace transform, molecular dynamics, and matrix multiplication using OpenMP and OpenACC directive-based languages. The experiment section demonstrates execution of these parallel applications on various programming environment. The result section demonstrates benchmark application implemented using OpenACC will perform better against to the benchmark application implemented using OpenMP.

The remaining part of this paper is organized as follows: Sect. 2 highlights various national and international efforts, which focuses on understanding the concepts of high performance computing, parallel programming, and directive-based programming models used for parallel programming. Sect. 3 focuses on architectural analysis of programming models. Section 4 discusses the proposed parallel implementation of various benchmark applications. Section 5 describes the performance evaluation, the experimental setup, and the results of the experiment along with the graphs. Finally, Sect. 6 provides conclusion and future work to enhance the proposed system.

2 Related Work

This section describes efforts made by several researchers to deal with issues and challenges in applying the methods for real-time applications. In Li, [1] has shown a performance examination among CUDA and OpenACC. The exhibition investigation manages programming models and fundamental compilers. The examination of execution holes has appeared in nineteen parts of ten benchmarks. They will in general use piece execution time and information affectability as primary principles reported once ends were made. A comparison of knowledge sensitivity may be a new index to explore an easily ignored downside that, however, each programming model is sensitive to changes of knowledge sizes. The vibes of the PRoDS equation brings the USA. A target examination rather than an emotional correlation. The OpenACC programming model is significantly more delicate to information than CUDA with advancements, while CUDA is much touchier than OpenACC to improvements. Generally speaking, the OpenACC execution is practically equivalent to CUDA under a decent correlation, and OpenACC might be a fair contrast to CUDA especially for amateurs in elevated-level equal programming.

The work [2] shows that OpenACC and OpenMP each lacks the power to completely generate a tailor-made multidimensional grid and threads for GPUs. Whereas this downside is often overcome by flattening the loop via the collapse clause, their experiments have shown that the performance of such associate improvement would possibly still be slower than the CUDA 2-dimensional grid, wherever we have a tendency to achieve our greatest performance result.

In Ledur et al. [3] and Memeti et al. [4], authors show the characteristics of OpenCL, OpenMP, OpenACC, and CUDA with respect to programming productiveness, enforcement, and energy. In Wang et al. [5], author shows how realistic it is to use a single OpenACC source code for a set of hardware's with different underlying micro-architecture, Nvidia Kepler, and Intel Knight Corner. In this project, we are considering Nvidia Tesla and Nvidia Quadro. Performance portability of OpenACC is related to the arithmetic intensity, and a big performance gap still exists in specific benchmarks between platforms.

The work [6] demonstrated that GPU-based parallel computer architecture has been showing extended notoriety as a building piece for first class handling and for future Exascale enrolling. They have assessed existing request-based models by porting application pieces from particular authentic spaces to utilize CUDA GPUs, which, along these lines, permits us to perceive fundamental issues in the supportiveness, adaptability, sensibility, and investigate limit of the current models.

3 Architectural Analysis of Parallel Programming Models

Generic directive-based programming frameworks comprise of directives, library routines, and designated compilers. In the request-based GPU programming models, a game plan of solicitations is utilized to stretch out data open to the selected compilers, for example, heading on arranging of circles onto GPU and information sharing rules. The most essential great circumstance of utilizing request-based GPU programming models is that they give bizarre state thought on GPU programming, following the consigned compiler covers the majority of the eccentric subtle parts explicit to the principal GPU structures. Another ideal position is that the solicitation frameworks make it simple to do steady parallelization of occupations, as OpenMP, with the ultimate objective that a client can choose domains of a host undertaking to be offloaded to a GPU gadget in a steady manner and a short time later the compiler, thus, makes related host gadget programs. There exists a couple of order-based GPU programming models. These models give unmistakable degrees of consultation and programming attempts expected to follow their models and smooth out the execution furthermore move.

3.1 OpenACC Programming Model

The OpenACC programming model adopts a renowned approach. The first application is explained with orders and calls to a runtime application program interface. The compiler is coordinated to create pieces that execute on the connected GPU or GPUs.

The OpenACC execution model is like that of CUDA; a fundamental program runs on the CPU and starts errands (computational pieces or information moves) on the GPU. The principle program handles synchronization, either through unequivocal client control or certainly. OpenACC embraces the natural feeble memory model utilized in most GPU programming models: The GPU and CPU memory spaces are particular. OpenACC is an API that gives a lot of array orders, runtime libraries, and condition factors that can be used to create equal projects in Fortran, C, and C + to run on quickening agents, including GPUs. Engineers can begin composing their calculations consecutively and introduce directives OpenACC in the algorithm. It resembles giving indications for the compiler to turn the code parallel.

3.2 OpenMP Programming Model

The OpenMP standard was created, and it is kept up by the gathering OpenMP architecture review board shaped from some big organizations, for example, Intel, SGI, SUN Microsystems, IBM, and others, that toward the finish of 1997, assembled power to make a conventional equal programming for shared memory models. The OpenMP API and spotlights on a lot of orders that underpins the making of equal projects with shared memory through the execution of a programmed and improved arrangement of strings. Its highlights would now be able to be utilized in dialects FORTRAN 77, FORTRAN 90, C, and C ++. The benefits of utilizing OpenMP can be shown on straightforwardness and little change in the codes, the powerful help for equal programming, simplicity of comprehension, and utilization of mandates, one help settled parallelism, and the chance of dynamic alteration of the quantity of strings utilized.

3.3 Difference Between OpenACC and OpenMP

The OpenMP program will contain pragma omp directives. When the compiler encounters a pragma directive, it will start executing the program parallel. It will divide the tasks among multiple cores of the CPU, and the program will start executing all the tasks on different CPUs simultaneously. And hence, the parallel programming of a given application can be achieved.

OpenACC will contain pragma acc directives. When the compiler encounters a pragma directive, it will start executing the program parallely. It will divide the program tasks among multiple cores of GPUs. GPU will contain thousands of cores which can be used to execute the program faster, and time taken for executing a program will be less. And, all GPU cores will be used to execute the program simultaneously to achieve parallel programming.

4 Proposed Methodology

The implementation part describes the parallel implementation of benchmark application by considering available GPU cores efficiently using OpenACC and available CPU cores using OpenMP. Performance of OpenACC programs running on GPU is compared with that of OpenMP programs running on CPU. The runtime of each core is evaluated on the test dataset. The benchmark HPC applications that are used to do the comparison are matrix multiplication, fast Fourier transforms, Laplace transforms, and molecular dynamics.

Parallelizing process may incorporate a few or all of the following:

- Analyzing segments of the code that can be performed simultaneously.
- Outlining the simultaneous bits of code onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data information related with the program.
- Overseeing access to information shared by various processors.
- Integrating the processors at different phases of the parallel program execution.

4.1 Parallel Implementation of Matrix Multiplication:

In math, framework augmentation is a parallel operation that takes a couple of grids and produces another network. Numbers, for example, the genuine or complex numbers can be duplicated by number juggling. Then again, networks are varieties of numbers, so there is no one of a kind approach to characterize "the" increase of grids. Accordingly, when all is said in done, the expression "lattice duplication" alludes to various diverse approaches to increase lattices. The key components of any network duplication include: the quantity of columns and segments the first grids have (called the "size", "request," or "measurement") and determining how the passages of the networks create the new grid. Like vectors, networks of any size can be duplicated by scalars, which add up to reproducing each section of the grid by the same number. Like the entry-wise meaning of including or subtracting lattices, augmentation of two grids of the same size can be characterized by reproducing the relating passages, and this is known as the Hadamard item. Another definition is the Kronecker result of two frameworks, to acquire a square network. One can shape numerous different definitions. On the other hand, the most helpful definition can be inspired by straight mathematical statements and direct changes on vectors, which have various applications in connected arithmetic, material science, and designing. This definition is regularly called the lattice product [2, 3]. In words, if An is a n m grid and B is a m p network, their framework item AB is a n p framework, in which the m passages

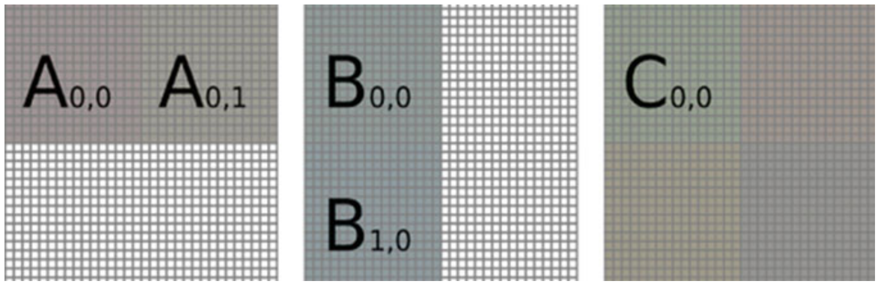

Fig. 1 Shows the parallelization of matrix multiplication

over the columns of An are reproduced with the m sections down the segments of B (the exact definition is underneath).

Algorithm

```
Input: matrices A[m,p] and B[p,n]
Let C[m,n] be a new matrix
for i from 1 to m:
 for j from 1 to p:
       Let sum = 0
             For k from 1 to n:
             Set sum <- sum + A[i,k] × B[k,j]
                   Set Cij<- sum
Return C
```

A local execution doles out one string to process one component of grid C. Each string loads one line of network An and one segment of framework B from worldwide memory, does the internal item, and stores the outcome back to lattice C in the worldwide memory. To build the "calculation-to-memory proportion," the tiled grid duplication can be applied. One string square processes one tile of lattice C (Fig. 1).

4.2 Parallel Implementation of Fast Fourier Transforms:

A FFT figures the DFT and creates precisely the identical result as assessing the DFT definition straightforwardly; practically, essential contrast is that a FFT is much quicker. Let $\times$ 0, …., x $N - 1$ be complex numbers. The DFT is defined by the formula.

Evaluating this definition particularly obliges O ($N2$) operations: There are N yields Xk, and each yield obliges a total of N terms. A FFT is any framework to enroll the same results in O ($N \log N$) operations. More conclusively, all known FFT computations oblige (N log N) operations, but there is no known confirmation that a lower disperse quality score is impossible.

To layout the venture trusts of a FFT, consider the count of complex growths and increments. Surveying the DFT's aggregates particularly incorporates $N2$ complex increments and N ($N1$) complex additions. The undoubtedly comprehended radix 2 Cooley-Turkey count, for N a power of 2, can enlist the same result with just (N/2) log2 (N) psyche boggling duplications and Nlog2(N) complex additions. For all intents and purposes, genuine execution on front line PCs is by and large controlled by components other than the rate of calculating operations, and the examination is a confounded subject, yet the general change from O ($N2$) to O (N log N) remains.

Algorithm

```
1.     For k=0 to l // where l=log (N)/log (2)
2.          Divide the set into intervals
3.              Get the corresponding twiddle factors, W
4.                in each interval
5.   for each pair of points J, J+Half-Size
6.   Analyze using butterfly representation
```

The portion of work that can be parallelized in this problem is finding out twiddle factors and FFT computation. Mapping of concurrent portions of work can be achieved by using the directive pragma omp for which splits parallel iteration spaces across threads in OpenMP and by using the pragma acc kernels directive in OpenACC to execute the portion of code on GPU. The data samples and the twiddle factors, in form of a complex number, are shared (global) across all the threads. Intermediate results generated during processing are stored in variables that are private (local) to the threads.

Synchronization is needed while moving back and forth in recursive task division, which is achieved through barrier directive in OpenMP and barrier directive in OpenACC.

4.3 Parallel Implementation of Laplace Transforms:

The Laplace change is a by and large used fundamental change as a piece of number juggling and electrical structure named after Pierre-Simon Laplace that changes a portion of time into a segment of complex rehash. The retrogressive Laplace change takes a flighty recurrent region breaking point and yields a cutoff depicted in the time space. The Laplace change is related to the Fourier change, yet while the Fourier change imparts a limit or banner as a superposition of sinusoids, the Laplace change conveys a limit, even more generally, as a superposition of minutes. Given a direct logical or utilitarian delineation of an information or respect a system, the Laplace change gives an alternative functional depiction that consistently revamps the technique of examining the lead of the structure or in mixing another system taking into account a course of action of determinations. Along these lines, for example, Laplace change from the time locale to the recurrent space changes differential connections into arithmetical numerical enunciations and convolution into steady.

Algorithm

```
1.    Set particle positions.
2.          Assign particle velocities.
3.          Repeat
                  Calculate force on each particle.
                  Update particle positions and velocity.
                  Measure properties ,Store results.
                  until the preset time steps
5.      Analyze properties, print results
```

The portion of work that can be parallelized in this problem is taking the size of matrix and calculating the time taken for each matrix. Mapping of concurrent portions of work can be achieved by using the directive pragma omp for which splits parallel iteration spaces across threads in OpenMP and by using the pragma acc kernels directive in OpenACC to execute the portion of code on GPU. The input data, viz. taking the size of matrix and taking their values, these values are shared (global) across all the threads. Intermediate results generated during processing are stored in variables that are private (local) to the threads. Synchronization is needed while calculating the time taken for each matrix. These calculations can be done within the pragma omp master directive in OpenMP and pragma acc master directive in OpenACC.

4.4 Parallel Implementation of Molecular Dynamics

Molecular dynamics (MD) is a PC reenactment of physical improvements of molecules and particles in the association of N-body diversion. The particles are allowed to interface for a period of time, giving a point of view of the development of the atoms. In the most generally perceived adjustment, the direction of particles and iotas are constrained by numerically handling Newton's examinations of development for a game plan of imparting particles, where qualities between the particles and potential imperativeness are described by interatomic conceivable outcomes or subatomic mechanics power fields. The strategy was at first envisioned inside speculative material science in the late 1950s yet is associated today generally in compound material science, materials science, and the showing of biomolecules.

Since atomic frameworks comprise countless, it is hard to find the properties of such complex systems legitimately; MD reenactment circumvents this issue by using numerical methodologies. In any case, long MD propagations are deductively seriously adjusted, creating consolidated mix-ups in numerical joining that can be limited with genuine selection of computations and boundaries, yet not cleared out totally.

For structures which agree to the ergodic theory, the advancement of a singular nuclear stream reenactment might be used to concentrate doubtlessly noticeable thermodynamic properties of the system: the time midpoints of an ergodic structure contrast with micro-canonical gathering midpoints. MD has moreover been named "quantifiable mechanics by numbers" and "Laplace's vision of Newtonian mechanics" of foreseeing the future by vivifying nature's powers and allowing understanding into sub-nuclear development on an atomic scale.

Algorithm

```
1.   Set particle positions.
2.       Assign particle velocities.
3.       repeat
       1.    Calculate force on each particle.
       2.    Update particle positions and velocity.
       3.    Measure properties, Store results.
4.     until the preset time steps
5.     Analyze properties, print results
```

The portion of work that can be parallelized in this problem is calculating the force on each particle. Mapping of concurrent portions of work can be achieved by using the directive pragma omp for which splits parallel iteration spaces across threads in OpenMP and by using the pragma acc kernels directive in OpenACC to execute the portion of code on GPU. The input data, viz. number of particles, their positions, and initial velocities, are shared (global) across all the threads. Intermediate results generated during processing are stored in variables that are private (local) to the threads. Synchronization is needed while measuring properties such as total kinetic energy and total potential energy. These calculations can be done within the pragma omp master directive in OpenMP and pragma acc master directive in OpenACC.

5 Experimental Setup and Results

This section is aimed to give a brief description of the experimental setup that is required for this project work to obtain the required results. First, the experimental setup is based on the system specifications to meet the requirements. Different applications are used to determine the performance of CPU and GPU and conclude with results and screenshots. Once the framework necessities are settled, then we need to figure out if a specific programming bundle fits framework prerequisites or not. The

Table 1 Tesla GPU configuration

	CPU configuration	GPU configuration
System type	HP Pro 3330 NT PC	PowerEdge R270
Processor	Intel Core i3-322Q CPU@3.30GHZx4	Intel xeon® CPU E5-26,200 @2.00GHZ\1S
RAM	2 GB RAM	32 GB RAM
Operating system type	64-bit	6 4-bit
Hard disk	500 GB	500GBx3
Graphics card	NA	Nvidia Tesla M2075 dual slot graphics card

Table 2 Quadro GPU configuration

	CPU configuration	GPU configuration
System type	HP Pro 3330 NT PC	DELL Precision R5500
Processor	Intel Core i3-3220 CPU @3.30GHZx4	Intel xeon® CPU E5620 2.40 GHz
RAM	2 GB RAM	32 GB RAM
Operating system type	64-bit	64-bit
Hard disk	500 GB	500GBx3
Graphics card	NA	Nvidia Qua dip K2000 dual slot graphics card

software bundle includes Nvidia drivers, PGI compiler, CUDA toolkit (Version 6.5), and Fedora operating system.

The following is the configuration of the system which is used in our experiment setup to run the applications on CPU and GPU (Tables 1 and 2).

PGI compiler is the compiler required to compile and run the OpenACC and OpenMP programs. The following graphs show the comparison of OpenMP and OpenACC on CPUs and GPUs for scientific applications on Tesla and Quadro graphics cards, in which *x*-axis shows the time in seconds and *y*-axis shows the number of particles in which blue color shows the results of OpenMP and red color shows the results of OpenACC. Performance is measured by varying the problem size of the benchmark applications.

Fig 2 shows the comparison OpenMP and OpenACC for matrix multiplication. The results on Tesla graphics card shows better performance because it contains more number of CPU cores (24) and more number of GPU cores (448) when compared to Quadro graphics card which contains less CPU cores (8) and less GPU cores (240). As the size of the matrix increases, OpenACC implementation will perform better (Fig. 3).

The figure shows the comparison OpenMP and OpenACC for FFT. On both the devices, we are able to observer large gap in the performance between OpenACC and OpenMP implementation as the problem size increases.

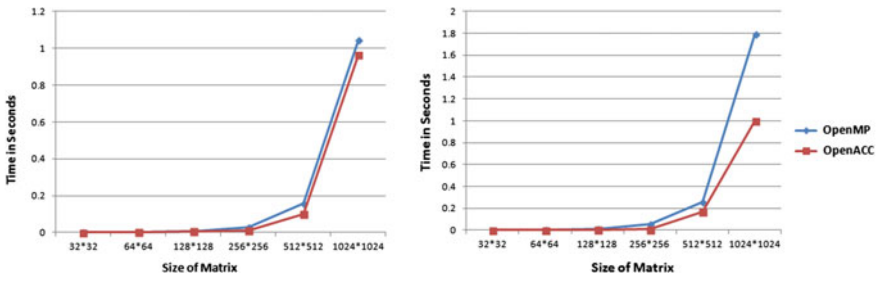

Fig. 2 Matrix multiplication performance on Tesla and Quadro graphics cards

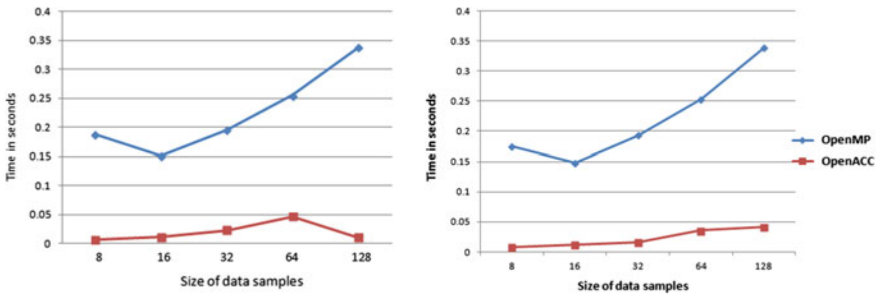

Fig. 3 Fast Fourier transform (FFT) performance on Tesla and Quadro graphics cards

Fig. 4 shows the comparison OpenMP and OpenACC implementation for Laplace transform. The results demonstrate performance improvement with OpenACC implementation as the size of the data samples increases.

(240) (Fig. 5).

The figure shows the comparison OpenMP and OpenACC for molecular dynamics. The results on Tesla graphics card shows better performance because it contains more number of CPU cores (24) and more number of GPU cores (448) when compared to Quadro graphics card which contains less CPU cores (8) and less

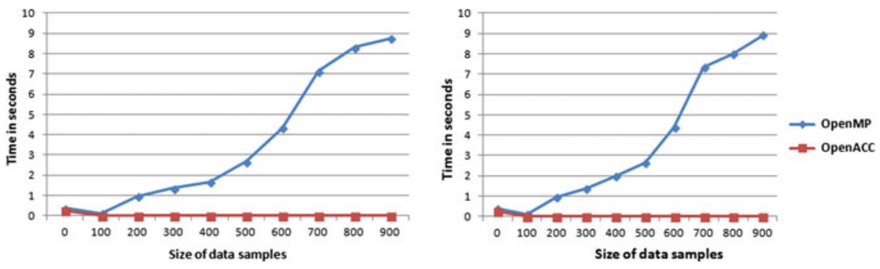

Fig. 4 Laplace transform performance on Tesla and Quadro graphics cards

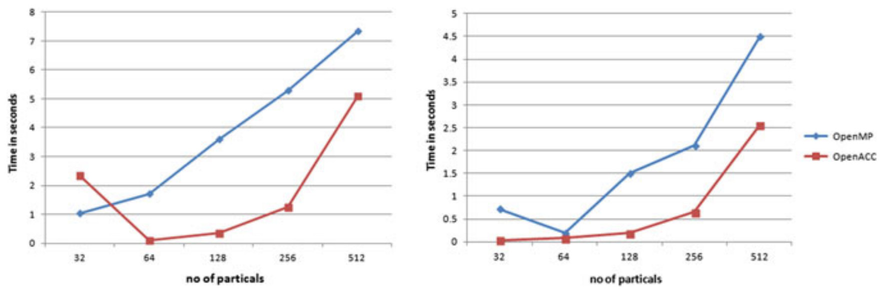

Fig. 5 Molecular dynamics performance on Tesla and Quadro graphics cards

GPU cores (240). Also, results demonstrate improvement in the performance as the number of particles increases.

6 Conclusion

Parallel programming is increasing in the near future, not only in massive computing software, but also in systems of small and medium businesses to generate more speed and providing the programmer more options to exploit the hardware resources. In this project, the performance of parallel program is better than the performance of serial program. When performance measurement was done, it was observed that time taken by OpenACC is better for smaller size data as well as when the size of data increases, when compared with OpenMP. Hence, from our work, we can conclude that OpenACC is a good option for data parallel task on GPUs. OpenMP is less complex and can give nearly equivalent performance to OpenACC on CPUs. Developers who need to utilize the parallelism must change their programming paradigms to address the issues that emerge as speed and better execution programming applications so as to build the limit computations and handling conceivable.

References

1. Li X, Shih PC (2018) An early performance comparison of CUDA and openACC. MATEC Web Conf 208:05002. https://doi.org/10.1051/matecconf/201820805002
2. Gayatri R, Yang C, Kurth T, Deslippe J (2018) A case study for performance portability using openmp 4.5. In: WACCPD@SC
3. Ledur CL, Zeve CM, dos Anjos JC (2013) Comparative analysis of openACC, openMP and CUDA using sequential and parallel algorithms
4. Memeti S, Li L, Pllana S, Kolodziej J, Kessler C (2017) Benchmarking openCL, openACC, openMP, and CUDA: programming productivity, performance, and energy consumption. In: Proceedings of the 2017 workshop on adaptive resource management and scheduling for cloud computing. ARMS-CC '17, Association for Computing Machinery, New York, NY, USA, pp 1–6. https://doi.org/10.1145/3110355.3110356

5. Wang Y, Qin Q, SEE SCW, Lin J (2013) Performance portability evaluation for openACC on intel knights corner and nvidia kepler
6. Lee S, Vetter JS (2013) Early evaluation of directive-based GPU programming models for productive exascale computing. In: SC ’12: proceedings of the international conference on high performance computing, networking, storage and analysis, pp 1–11