IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 20 February 2024, accepted 26 February 2024, date of publication 4 March 2024, date of current version 8 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3372990

SURVEY

MIMD Programs Execution Support on SIMD Machines: A Holistic Survey

DHEYA MUSTAFA[1], (Member, IEEE), RUBA ALKHASAWNEH[2], (Member, IEEE), FADI OBEIDAT[3], AND AHMED S. SHATNAWI[4]

[1]Department of Computer Engineering, Faculty of Engineering, The Hashemite University, Zarqa 13133, Jordan
[2]Department of Communication and Computer Engineering, Faculty of Engineering, Al-Ahliyya Amman University, Amman 19111, Jordan
[3]Synopsys Inc., Austin, TX 80309, USA
[4]Department of Software Engineering, Jordan University of Science and Technology, Irbid 21110, Jordan

Corresponding author: Dheya Mustafa (dheya@hu.edu.jo)

ABSTRACT The Single Instruction Multiple Data (SIMD) architecture, supported by various high-performance computing platforms, efficiently utilizes data-level parallelism. The SIMD model is used in traditional CPUs, dedicated vector systems, and accelerators such as GPUs, vector extensions, and Xeon Phi. It provides performance throughput in computation-intensive and data-parallel applications. Despite the similarity of data-processing principles between these architectures, porting various programming models between the reviewed platforms is challenging. Furthermore, enhancing the programmability of these architectures is an important feature for utilizing their emerging computing power and simplifying programming complexity. This paper reviews the basic principles of optimization techniques to run asynchronous Multiple Instruction Multiple Data (MIMD) on SIMD accelerators. It also surveys several GPU programming paradigms and application programming interfaces (APIs) and classifies these frameworks into different groups based on their criteria. In addition, a review of studies that performed a comparison of the collaborative execution of GPUs with CPUs and Xeon Phi is presented in this paper. This study will be beneficial for developers and researchers in the field of computer architecture and parallel computing of intensive scientific applications, specifically for early-stage high-performance computing researchers, to obtain a brief overview of performance optimization opportunities as well as the challenges of existing SIMD platforms.

INDEX TERMS Accelerators, asynchronous applications, GPUs, irregular applications, SIMD.

I. INTRODUCTION

Flynn's taxonomy categorized parallel computers based on how concurrently they handle instructions and data sequences (aka streams). Four classes are produced as a result: Single instruction single data (SISD), multiple instruction single data (MISD), single instruction multiple data (SIMD), and multiple instruction multiple data (MIMD) [1]. The advent of SIMD architectures with wider lanes and more applicable programming models is providing new opportunities along with challenges to improve performance and energy efficiency.

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

Cray-1 [2], created by Cray Research and Seymour Cray, was the first commercially successful, high-performance innovation supercomputer with vector instructions. Vector instructions became an obsolete technology in the 1990s. With the introduction of the MMX instruction set (1997) to the x86 architecture and Intel Pentium processors, vector instructions began to reappear. SIMD accelerators have been utilized by modern microprocessor designers [3], [4], [5], [6], [7], [8] to exploit data-level parallelism (DLP), in which multiple data elements are processed simultaneously. Examples of recent commercial SIMD accelerators include 512-bit SIMD implementations from Intel [7], [9] and Fujitsu [10]. However, performance is limited by control\data flow divergence, irregular\sparse data structure,

poor cache and memory performance, and low hardware utilization.

Graphic Processing Units (GPUs), similar to CPUs, greatly exploit DLP using an enormous multi-threaded, multi-core architecture that can be executed in a lock-step model. Modern GPUs exploit explicit thread-level parallelism in software to increase throughput [11], [12]. However, improving performance is the responsibility of software developers. Due to their massive computing power (exceeding the Teraflop scale) and energy efficiency, GPUs were quickly adopted as general-purpose accelerators for high-performance computing. Much effort has been made for this purpose, including more computing engines with support for control-flow and improved programming models, exploiting this power for many general-purpose applications. Furthermore, to reduce data transfer latency between the CPU and GPU for optimal offloading, they are integrated on the same die.

Handling control-flow divergence efficiently is one of the major challenges for modern GPU architectures to exploit their unbridled power [13]. Computation density can be improved on modern GPUs by batching threads running the same program in groups and executing them together in lock step [14], [15], [16], [17], [18]. Irregular applications contain data-dependent\indirect memory accesses, which makes implementing this broad class of applications in SIMD architectures very challenging [19].

This study presents a pervasive literature survey of the basics of various methodologies that tackle the aforementioned issues. Furthermore, it reviews many optimization schemes that support MIMD program execution on SIMD (MoS) architectures across various platforms and programming models and categorizes them into different classes according to many representative attributes.

This survey is much more comprehensive and up-to-date than prior surveys, which either focused on a specific application [20], programming model, or platform [21], [22] or were not much detailed. In addition, it encompasses a broad spectrum of MIMD execution systems developed during the last thirty years, spanning several SIMD platforms and programming paradigms.

The main contributions in this paper are:

- Providing an extensive and an up-to-date survey on analyzing, categorizing and comparing various approaches to executing MIMD programs on SIMD architectures. This can show areas for development and assist the community in understanding the use cases of various execution mechanisms.
- Describing contemporary SIMD architectures and highlighting their major attributes, especially complex low-level hardware characteristics that are important for computation efficiency.
- Analyzing and categorizing applications based on the opportunities and challenges to accelerate them using SIMD platforms.

Figure 1 shows the outline of the paper. The organization of the rest of the paper is as follows: Section II sets the scope of the survey and provides a brief background on the problem. Section III briefly describes the main representatives of modern SIMD architecture platforms. An overview of GPU programming models is reviewed in Section IV. In Section V, we classify MIMD on SIMD execution frameworks into several categories and discuss these categories in detail. We discuss irregular applications in section VI, followed by surveying evaluation studies in Section VII. Section VIII summarizes the main observations. Finally, we conclude the paper in Section IX.

II. BACKGROUND

SIMD machines have a centralized control structure where one control unit is shared across the processing elements (PEs) [23], [24], [25], [26]. This control unit, executed on a single thread, broadcasts instructions to the PEs for execution. Examples of SIMD machines include Cray processors, extensions (Intel SSE and AVX), Connection Machines, MasPar machines, and modern GPUs. GPUs implement a group of loosely coupled relatively narrow SIMD engines and avoid hardware-intensive features (i.e. memory protection). GPUs utilize the Single Instruction Multiple Threads (SIMT) model. GPUs warps include groups of threads executing the same instructions in a SIMD fashion. Vectorization on classical SIMD is handled at the instruction-level (vector instructions are generated), whereas in GPUs (SIMT), vectorization is implicitly hidden but should be taken into account to generate fast code. The problem of SIMD is that the result is more scalable than MIMD but more complex and restrictive, and the execution model is difficult to program.

Contrariwise, MIMD machines implement a separate control unit for each PE to make them able to execute a program segment locally. General parallel applications with control parallelism [27], [28] do not perform well on classical SIMD machines. Thus, SIMD machines were avoided as a platform for general-purpose parallel processing [29], [30].

After surveying a huge amount of related work in the literature, applications can be classified based on their structure into synchronous, loosely synchronous, asynchronous/irregular, and stencil applications. Synchronous applications, such as matrix algebra, naturally fit into SIMD architecture, while implementing irregular applications for SIMD is an essential problem. Table 1 summarizes these categories with their main challenges.

To fully utilize the powerful computation resources that SIMD architectures provide, an application should increase on-chip data locality and reduce global memory access, memory divergence, and execution divergence. Execution divergence is caused by conditional code blocks, where some SIMD lanes may execute along different paths (if-path, else-path) depending on the conditional result of each lane. Code segments in different execution paths must be partitioned into

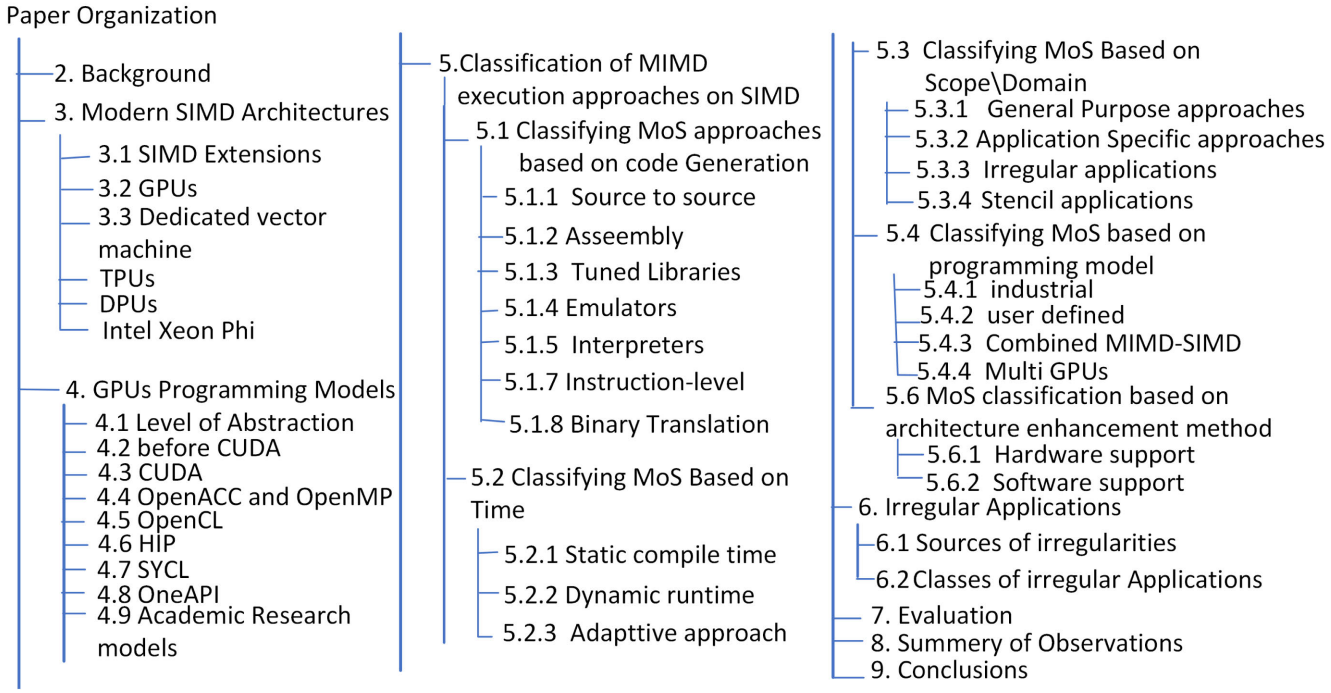


FIGURE 1. Paper organization.

TABLE 1. Classification of Problem structures and their implementation on MIMD and SIMD machine.

Applications Structure	Implementation on MIMD	Implementation on SIMD	Examples
Synchronous	Novice	Normal	Finite difference, Matrix Algebra, QCD
loosely synchronous	Normal	Hard, data dependencies	irregular finite elements, Molecular dynamics, Unstructured mesh
Stencil	Hard -effective caching is extremely important -Parallelization can be applied to the inner-level nested loops	Very Hard -Unaligned none-one stride accesses (high cache miss-rate), memory stream alignment issue	Heat3D, Sobel, Jacobi, computational electromagnetics, image processing, and partial differential equation solvers
Asynchrounous/ Irregular	Hard require runtime to perform dynamic loop scheduling, and load balancing,	Extremely Hard, -require runtime support; -memory alignment is not available at compile-time, and -control divergence	Event-driven simulation, N-queen problem, Region growth, stream

several sub-groups and serialized, leading to low resource utilization.

The main qualities of a general-purpose approach for running MIMD on SIMD are reusability, applicability, scalability, portability, difficulty, utilization, and efficiency. This problem has been studied for several decades, such as the early work on combinators [31], [32], the MIMD simulator [33], and the interpretation of Prolog and FLAT GHC [34], [34], [35]. Other studies proposed building hybrid parallel machines that operate in both Single Program-Multiple Data (SPMD) and SIMD modes of parallelism [36], [37]. In SPMD mode, the same program is independently executed on all processors using distinct data sets, and each processor may use a different control path to run the program. Different parts of a single program might be divided into parts, which are then mapped more closely to different modes of parallelism [38], [39]. For example, both PASM [40], [41], [42], [43], and TRAC [44], [45] are hybrid SIMD\MIMD machines that support both models. OPSILA [46], [47], [48] is a hybrid SIMD\SPMD machine, while Whole-Function Vectorization [49] and Intel's ISPC compiler [50] are both examples of SPMD-on-SIMD.

III. MODERN SIMD ARCHITECTURES

This section describes the recent popular platforms that support SIMD architecture. SIMD architectures include multiple identical processing units that operate on a vector of operands. This includes Intel SSE, AVX, Xeon Phi, ARM Scalable Vector Extension (SVE), NVIDIA, AMD and Intel GPUs. We highlight their major architectural attributes that are essential for porting high-performance applications and challenges as well.

A. SIMD EXTENSIONS

Table 2 lists several ISAs for SIMD extensions with their major attributes. Intel's Advanced Vector Extension (AVX) and AVX-512 have been expanded to operate on 256 and 512-bit SIMD registers compared to the 128-bit registers of the Streaming SIMD extensions (SSEs) [51]. Existing multimedia extensions, such as MMX/SSE for Intel and VMX/AltiVec for IBM, Apple, and Motorola, are SIMD units with fixed-length vectors. SIMD units are also used in DSP processors, graphics engines, and game consoles. Both Intel and AMD CPU architectures support vectorization. The AMD Rome processor, which was used in datacenters, supports x86 vector instructions up to 256-bit AVX2 [52].

The ARM architecture has extended the aging NEON SIMD extensions with Scalable Vector Extension (SVE) vector architecture, which features a contemporary application of concepts first presented in the 1970s with the Cray supercomputers' vector architecture [51], [53]. ARM SVE architecture is used in Fujitsu's ARM A64FX CPU with 512-bit vector length to produce the Fukagu supercomputer (Top500 list, November 2021).

Due to the open-source RISC-V softcore design flexibility, it is being used in many applications. Researchers are able to integrate customized SIMD instructions and optimize micro-architecture to increase the efficiency of the processor [54], [55]. The RISC-V V-extension (RVV) is meant for high-performance cores and provides support for 8-bit, 16-bit, and 32-bit integer SIMD instructions. The RISC-V P-extension is used in low-power designs.

The data lane width of SIMD accelerators is extending with most new generations, with more demands for performance. This brings up the compatibility issue for different generations. For example, Intel Advanced Vector Extensions (AVX) and AVX2 support 256-bit vectors [6]. whereas AVX-512 [56] and Many Integrated Core architecture [57] support 512-bit vector operations. Another example is that Intel MMX had 64-bit vectors, which increased to 128-bit SSE extensions [6]. Liquid [58] and Vapor [59] SIMD proposed a combined compiler translation and runtime support method to solve compatibility issues. ARM SVE offers a vector length-agnostic model to support SIMD architectures of different lane lengths. This enables machine code portability on different hardware generations with varying vector widths as needed [60].

The memory alignment constraint of SIMD units significantly affected the simdization efficiency. A major obstacle to simdization is the unavailability of alignment information at compile-time (aka runtime alignment). A large body of work has considered vectorizing irregular applications on SSE. Kim and Han [61] proposed a compiler technique to simidize irregular kernels, primarily intra-iterations, containing indirection-based memory accesses on Cell SPU, which has a shorter SIMD lane width compared to the AVX.

B. GRAPHICAL PROCESSING UNITS (GPUS)

Graphics Processing Units (GPUs) were first created to speed up the creation of graphical output with the potential for extremely high parallelism, including digital photos, text, video games, 2D and 3D geometric models, and digital images. A GPU Device is organized as a grid of Compute Units. Each Compute Unit is organized as a grid of Processing Elements. The semi-modular design of both NVIDIA and AMD GPUs groups the processing components, or units that carry out computations, into many computing units. The processing components inside each computing unit share some resources, such as schedulers, dispatch units, registers, caches, fast local storage, load/store units, and texture units. GPU die contains dozens of compute units, each of which has dozens of processing elements, for a grand total of thousands processing elements.

NVIDIA GPUs are based on the SPMD programming model at the block level (work groups in OpenCL and groups in ATI GPUs [62]) and the SIMD model at the warp level. Blocks and threads coordinate their actions and communicate data. A block's threads are arranged into 32-thread warps, which can be interspersed with hardware multi-threading to withstand stalls within the warp and allow memory latency overlap with practical processing. A streaming multiprocessor (SM) operating in SIMD mode processes the threads within a warp. The thread blocks allotted to each SM are divided into 32-thread warps, which are subsequently scheduled to run on streaming processors when they're available [63]. Typically, all NVIDIA GPUs attributes discussed also hold for AMD GPUs except warp size, where AMD uses 64 instead of 32. Contemporary NVIDIA [18] and AMD [62] GPUs achieves more than 2.0 GHz rate. The potential efficiency of GPUs is further increased by avoiding hardware-intensive features (i.e., interrupt handling, memory protection, and large caches) that reduce peak arithmetic performance per unit circuit complexity.

The Intel Arc A770 GPU boasts 512 execution units (vector engines) and 2.4 GHz clock speeds, ensuring smooth and responsive graphics rendering. Intel's Arc A770 enjoys decent local memory latency, though it is slightly behind current cards from AMD and NVIDIA. It's similar to NVIDIA's Maxwell based GTX 980 Ti.

C. DEDICATED VECTOR ARCHITECTURE MACHINE

The NEC SX-Aurora TSUBASA was reviewed as the latest representative of the SX vector supercomputer with dedicated vector processors [64], [65]. The Vector engine processor, with eight cores, reaches a peak performance of 2.43 TFlop/s (DP), as each core offers 304 GFlop/s of DP throughput at a 1.584 GHz frequency. The memory subsystem in the vector processor can deliver 1.22 TB/s memory bandwidth [66]. With such high memory bandwidth, the system can perform well in memory-bound applications.

A memory subsystem, a vector processing unit (VPU) for computations, and a scalar processing unit (SPU) that

TABLE 2. SIMD instruction set extensions.

ISA	VECTOR LENGTH (2)	register size(width)	SIMD width(1)	Max flops/ cycle	Processors
3DNow	2 SP	64-bit	2	4	AMD Athlon, Opteron
SSE	4 SP	128-bit	4	8	Intel Pentuim III, and IV, AMD Opteron and Athlon XP
SSE2	2 DP	128 bit	4	8	Intel Pentuim IV, AMD Opteron
Altivec	4 SP	128-bit	4		Motorola PowerPC G4, IBM PowerPC G5
Double FPU	2 DP	64 bits	2	4	IBM PowerPC 440 FP2 (BlueGene/L)
AVX	2 SP	128-256 bits	8	16	Intel Sandy Bridge/Ivy Bridge, AMD Ryzen/EPYC
AVX2	2	256 bits	8	32	Intel Haswell, Skylake, AMD Ryzen/EPYC
AVX512	4	512 bits	16	64	Skylake (server, Knights Corner, Knights Landing
SVE	Configurable	128-2048 bits	8(256bits)	32	ARM Neoverse V1
SVE2	Configurable	128-2048 bits	4(128bits)	4-8	ARM Cortex-A510
NEON	2 DP	128-bit	4	8	ARM Cortex-A57
RVV, RVP	Configurable	16384-bits	512bit	32	RISV-V

(1) the number of single precision operands. (2) SP: single precision, DP: double precision.

functions as a standard CPU are all included in an SX-Aurora vector core. Since SX-Aurora is an independent processor, SPUs may do scalar computations with comparatively high performance. Using a comparatively straightforward instruction pipeline of its own, the VPU decodes and reorders vector instructions that arrive from the SPU. Following that, instructions are carried out on 32 identical vector-parallel pipelines (VPP), each of which has three FMA (Fused Multiply-Add) units, two ALUs (Arithmetic and Logic Units), one unit specifically designated for high-latency commands (such as sqrt and division) and a memory subsystem for communication.

D. TENSOR PROCESSING UNIT (TPU)

Tensor-processing unit (TPU) is one of the most promising new digital hardware accelerators. It speeds up machine learning (ML) applications by performing parallel computing through a deeply-pipelined network of processing elements (PEs), especially systolic arrays [67]. Systolic arrays in TPUs reduce energy consumption and improve performance by recycling data fetched from memory and registers and reducing irregular intermediary memory visits [68]. However, because Fully Connected (FC) layers often contain a large number of weights, TPUs struggle to retain an equivalent level of performance when executing FC layers. Due to the restriction on weight reuse and the need for many execution rounds, this leads to ineffective hardware usage and significant energy consumption [69]. It should be mentioned that systolic arrays in practice have symmetrical sizes to enhance the execution of convolutional layers. They can, however, speed up FC operations at the expense of convolutional layer execution performance if they are constructed with asymmetric dimensions.

According to a recent study [69], TPUs perform better than GPUs and CPUs while running convolutional layers because of their spatial reuse properties and capacity to remove unnecessary features from the neural network. TPUs, on the other hand, perform worse than GPUs due to reduced weight reuse and are less popular for running FC layers. High memory traffic and energy usage result from this, especially as the model size grows [70].

E. DATA PROCESSING UNITS (DPUS)

The NVIDIA BlueField-3 DPU is a 400 Gbps infrastructure computation platform featuring line-rate processing for cybersecurity, software-defined networking, and storage. BlueField-3 provides hardware-accelerated software-defined solutions for the most demanding applications by combining powerful processing, fast networking, and comprehensive programmability. BlueField-3 redefines the art of the possible, from accelerated AI to hybrid clouds, high-performance computing to 5G wireless networks [71].

F. INTEL XEON PHI

Wide vector units and the Intel®Many Integrated Core (Intel®MIC) architecture served as the foundation for the Intel®Xeon Phi™processor. It had a lot of small, power-efficient in-order cores and several instructions without equivalents in SSE or AVX. It was meant for highly parallel applications to use the Intel®Xeon Phi™processor [72].

Although the Intel Xeon Phi architecture was officially discontinued in 2020, studying its huge related work is still beneficial. It will provide researchers with vital advice on how the evolution of hardware and software learned from the mistakes made by its predecessors. The Intel Xe graphics cards were announced in November 2019 as a result of Xeon Phi's evolution [73]. It was stated that the new design will prevent Xeon Phi's faults from happening again and offer a single programming model (oneAPI) with exceptional performance.

Table 3 summarizes architectural specifications for different SIMD platforms.

IV. PROGRAMMING MODELS

GPGPU computing has emerged as a powerful and attractive heterogeneous platform for general-purpose parallel computing. The GPU execution model requires balanced computing resources from both microprocessors and coprocessors to utilize the system effectively. An overview of the various

TABLE 3. Architectural differences among different SIMD platforms.

Platform	Boost Frequency (GHz)	# cores/ shading	SP performance/core (Gflop/s)	DP performance / core (Gflop/s)	Memory bandwidth (GB/s)	Memory capacity (GB)
TSUBASA 10AE	1.584	8	4860(608)	2430(304)	1228.8	48
GeForce RTX 3090 Ti	1.86	84/10752	40000(476.1)	625	1008	24(GDDR6X)
AMD Radeon™ RX 7800 XT	2.43	60/3840	37300(621.6)	1200	624	16(GDDR6)
Intel Arc A770 Limited Edition	2.4	32/4096	17200(537.5)	NA	660	16(GDDR6)
TPUv4	1.050	configurable	275000 HP	NA	1200	32(HBM2)
BlueField-3 DPU	2.133	16/256			80	32 DDR5
Intel Xeon Phi 7290	1.50	72	96	48	115.2 (DDR)	up-to 384 16(MCDRAM), 96(DDR)
Intel Xeon Gold 6126	2.60	12	166.4	83.2	128	up-to 768 (96)

software ecosystems for accelerator technology is given in this section. We provide an overview of the most relevant programming models available for the accelerators, their interoperability, benefits, and drawbacks, as well as some helpful tips. Each programming model has specific documentation that the reader should consult.

A. LEVEL OF ABSTRACTION

GPU programmers prefer high-level languages and libraries that provide convenient GPGPU abstractions and hide the complex execution environment using compiler optimization techniques, debuggers, and profilers, alleviating the programmer of the GPU-specific low-level hardware information.

GPU programming models can be classified based on the level of abstraction into high-level and low-level. CUDA is the most common low-level approach of NVIDIA GPUs, whereas OpenACC and OpenMP are high-level approaches. A low-level approach takes significant effort from the programmer to use low-level hardware primitives in writing code, exploring all possible GPU architecture capabilities. It is often faced with issues of portability, reusability, applicability, or abstraction. On the other hand, OpenACC and OpenMP programming models require the programmer only to annotate the parallel blocks of code using pragmas. Using such high-level, excessively transparent GPU frameworks can lead to a significant drop in the underlying computational efficiency.

The choice of proper GPU programming model depends on the characteristics of the application, the target hardware architecture, and the level of control and optimization required. OpenMP is more approachable for general-purpose parallelism on multi-core CPUs, while CUDA provides finer control for maximizing performance on NVIDIA GPUs in highly parallel workloads. Choosing the right tool for the job often involves considering the specific requirements and constraints of the given computational task. Although the GPU developers would prefer high-level languages for convenience, most GPU programs are still written in CUDA\OpenCL as the high-level languages typically cannot provide sufficient performance.

B. BEFORE CUDA

Before the adoption of the CUDA programming model [18], GPUs were programmed using difficult, low-level graphics programming interfaces and required broad expertise in the underlying hardware. Several shading environments, such as Cg (C for graphics), HLSL (high-level shading language), Sh, and OpenGL, were proposed to facilitate writing GPU code in a simpler C\C++-like programming language. However, all of these languages contain graphics-specific constructs and are tied to the specialized nature of GPUs. OpenGL and DirectX are the two major standards of graphics hardware shading APIs that were designed to program graphics operations.

Several high-level programming languages were proposed with runtime support to simplify GPU programming. BrookGPU is an extension to the ANSI C standard that can be used to program GPUs such as NVIDIA or ATI [74].

C. CUDA

CUDA is the most popular API that provides a general-purpose programming model for NVIDIA GPUs, exploring all possible abilities of the SIMT GPU architecture. NVIDIA's CUDA provides improved programmability with a familiar development environment of massively parallel programs for GPUs using the C programming language, in addition to CUDA tools [18], [75], [76]. CUDA threads execute on the streaming processors in the SPMD model. CUDA has provided a significant boost for the GPGPU efforts.

Performance optimization of CUDA-based GPGPU applications has been studied extensively. Several studies have concluded that obtaining high performance with CUDA programs is not trivial, and the performance of GPU applications can differ significantly based on how well optimization strategies were applied [77], [78], [79], [80], [81]. Petrovic et al. [82] described a dynamic auto-tuning framework for CUDA.

D. OPENACC AND OPENMP

OpenMP [83] and OpenACC [84] frameworks provide high-level declarative parallel programming models for

GPUs. They used compiler pragmas to simplify the code parallelization. The programmer only has to annotate the code segments to be parallelized using pragmas, and compilers effectively generate parallel code for the GPU, which is not always a doable task. For this reason, to get maximum performance on OpenACC and OpenMP, a programmer intervention will often be required to write code that conceptually repeats CUDA or OpenCL code when creating a program.

The primary benefit of OpenACC is that it usually eliminates the need to change legacy code. Sometimes, in addition to adding a few pragmas, code needs to be significantly altered to improve performance. This feature makes it possible to parallelize complicated tasks at a lower cost than with alternative programming methods.The low implementation cost targeting NVIDIA/AMD devices makes it beneficial to employ in most situations.

A further advantage of OpenACC over the most widely used programming models is its portability. The majority of the time, moving an application from one architecture to another just requires modifying the compiler flag. For example, to force the NVIDIA compiler to parallelize code on the CPU rather than the NVIDIA GPU, change `nvc -acc=gpu` to `nvc -acc=multicore`. You can specify `-acc=host` to generate an executable that will run serially on the host CPU. It is possible to adjust additional flags, such as `-gpu=cc70` to exclusively target a Volta GPU, and enhance performance on that desired architecture. This will parallelize code on the GPU and focus on the particular GPU microarchitecture, in this example, Volta. The majority of compilers that support AMD or NVIDIA GPUs also support CPUs. Code must be ported from the NVIDIA GPU to the AMD GPU by altering the compiler's settings and compilation flags (i.e., from nvc to gcc).

OpenMP has been working on extending its support for GPU offloading. The ''target'' directive, introduced in OpenMP 4.0, allows developers to offload computations to accelerators like NVIDIA GPUs. With OpenMP's target offload model, developers can specify code sections to run on the GPU, taking advantage of the GPU's parallel processing power to speed up certain computations. This provides a convenient way to harness GPU parallelism for specific tasks, such as data-parallel operations.

E. OPENCL (OPEN COMPUTING LANGUAGE)

OpenCL is an open standard API for managing parallel computations and resources. It supports a heterogeneous device programming environment through uniform abstraction of widely divergent hardware architectures (high-performance computing servers, multi-core CPUs, GPUs, handheld devices, FPGAs, etc.) as computational units [85]. The OpenCL standard allocates different levels of memory. It also supports cross-vendor software portability.

A key issue is performance portability across different architectures, because performance may be sensitive to input size, structure, or application settings; a code optimized for some input may run suboptimally when the input is changed [86], [87]. Examples of optimization knobs include the workgroup size, tile sizes, loop unrolling factors, and vector data types [88].

F. HETEROGENEOUS-COMPUTE INTERFACE FOR PORTABILITY (HIP)

AMD created the open-source C++ runtime API and kernel language known as Heterogeneous-Computing Interface for Portability (HIP) [89]. With just one source code, developers can use it to construct portable apps for GPUs made by AMD and NVIDIA [90]. Its goal is to expose more low-level hardware functionality while maintaining near-native performance on CUDA machines [91]. Functions like hipMalloc, hipMemcpy, and hipFree are part of the HIP API. Programmers who are already familiar with CUDA will find it easy to pick up the HIP API [90] and begin writing code right away. The 'hipLaunchKernel's macro call is used to launch compute kernels. The source-to-source conversion from CUDA to HIP is automated by the HIPify tools. HIP code can be executed on AMD or NVIDIA hardware using the HCC or NVCC compilers, respectively.

G. SYCL

An open standard parallel programming model called SYCL [92] is based on the contemporary object-oriented programming language C++11 and enables the writing of codes that may be executed on heterogeneous systems made up of hardware from many vendors. This means that significant sections of the application will operate in parallel on the host CPU or on accelerator devices using a single source code base. The Intel open source compiler for SYCL-based programs, DPC++, is derived from ISO C++, Khronos SYCL, and community extensions [93].

Any device in the system can be programmed using a single programming model thanks to SYCL. This is a compelling proposal for the performance portability initiative [94]. In heterogeneous systems, where a computer node may be made up of multicore CPUs, GPUs, FPGAs, and other problem-oriented devices like ASICs (Application-Specific Integrated Circuits), it adds the data parallel programming model to C++ programs.

H. ONEAPI PROGRAMMING ENVIRONMENTS

Recently, Intel released OneAPI, a programming paradigm that uses DPC++ based on SYCL2020 to provide such a solution on a single language platform. With teroperability, it facilitates data transfer and operation control over numerous devices with multiple accelerators under a single programming framework. Code in the oneAPI framework is written in DPC++ using SYCL 2020 for applicability to different accelerating devices, such as a GPU or an FPGA.

Intel's framework and unified, open programming paradigm, OneAPI [95], is used to create applications

for heterogeneous systems, such as CPUs, GPUs, FPGAs, and other devices. The platform, execution, and memory models of SYCL serve as the foundation for these models. The DPC++ compiler, libraries, debuggers, and other tools are all combined into one package by the OneAPI framework.

I. MORE ENVIRONMENTS FROM ACADEMIC RESEARCH

The following subsections describe several programming models from academic research groups.

1) OPENMP-TO-CUDA TRANSLATION FRAMEWORK

Lee & Eigenmann described an automatic OpenMP-to-CUDA translation framework [96], [97], [98], [99], to facilitate writing parallel applications for GPGPU architectures easily with OpenMP. Their approach supports special annotations added by a programmer, which are used for fine-tuning along with automatically extracted information from the OpenMP directives. It supports several optimization techniques that handle the architectural differences between GPUs and traditional shared memory systems, served by OpenMP [99], [100], [101].

2) HYBRID MPI-CUDA

This framework [102] is proposed for computing many pairwise alignments for large sequence datasets on heterogeneous CPU-GPU clusters. Work is distributed to the compute nodes through a cluster-level dispatcher, which also aggregates the results. MPI-CUDA applications are really very common. Probably most of the applications exploiting GPUs in HPC clusters are of this kind.

3) GEMTC (GPU ENABLED MANY-TASK COMPUTING)

The GeMTC framework is a combined execution model and runtime system that supports programming accelerators with many concurrent and independent tasks of potentially short or variable duration [103].

4) FASTFLOW

FastFlow provides a simplified high-level abstraction model to program heterogeneous parallel platforms using general-purpose C++ programming. This framework facilitates writing portable parallel applications and exploiting performance on a wide range of platforms [104].

5) GPU RUNTIME CODE GENERATION TECHNIQUES

Several techniques use high-level programming languages for GPUs, along with source-to-source code generation support. PyCUDA and PyOpenCL connected Python [105] with CUDA and OpenCL compute abstractions [18], [106]. CorePy [107], jCUDA [108] provided a source-to-source translation. hiCUDA [109] used OpenMP for GPU programming. The bulk-synchronous parallel model [110] was ported to the GPU architecture [111].

V. CLASSIFICATION OF MIMD EXECUTION ON SIMD (MOS) APPROACHES

Methodologies that implement a MIMD program on SIMD (MoS) can be classified into several categories based on different important factors. This includes, but is not limited to, the target architecture programming model, the application scope, and the time at which optimization takes place. This section provides a detailed discussion of the proposed novel classification taxonomy, which is based on five criteria: code generation method, optimization scope, tuning stage, programming model, and methods that require hardware modification. It should be emphasized that this taxonomy is not mutually exclusive, and one MoS framework can belong to more than one class. All classes are summarized in the taxonomy diagram depicted in Figure 2.

A. CLASSIFYING MOS APPROACHES BASED ON CODE GENERATION METHOD

The code generation method implemented in MoS frameworks was used as an important factor in classification. There are efforts to support code portability between CPUs and GPUs and simplify GPU programming using convenient traditional parallel programming languages. Several sophisticated code generation strategies were investigated in previous studies, such as source-to-source translation [98], [112], developing non-GPU front-ends [113] and machine-independent optimizing compilers (LLVM) [114]. Ren et al. [115] introduced a SIMD code generation and optimization engine for irregular data-traversal applications. The main classes of MIMD execution frameworks on SIMD based on code generation methods are source-to-source translation, assembly, tuned libraries, emulators, instruction-level, and binary translation.

1) SOURCE-TO-SOURCE TRANSLATORS

A large number of automatic CPU-to-GPU source parallelization translation tools have been developed for academic and commercial use [84], [98], [109], [116], [117], [118], [119], [120], [121], [122], [123], [124]. This includes directive-based models [84], [98], [98], [109], [117], [120], [125], [126], or polyhedral-based [119], [124].

Kapasi et al. [127] introduced conditional streams, a code transformation that separates a single kernel with conditional code into multiple kernels and connects them via inter-kernel buffers to increase the utilization of a SIMD pipeline, such as the stream register file in Imagine [128] and Merrimac [129].

The CUDA-lite [116] translator relies on annotations provided by a programmer to perform optimal tiling of global memory data transformations. On the other side, MCUDA [130] translates the CUDA data-parallel applications into conventional shared memory, multi-core systems. Polyhedral [131] and IR-to-IR (intermediate representation) [132] transformations belong to this category. OpenMPC [96] translates OpenMP to Cuda. hiCUDA [109],

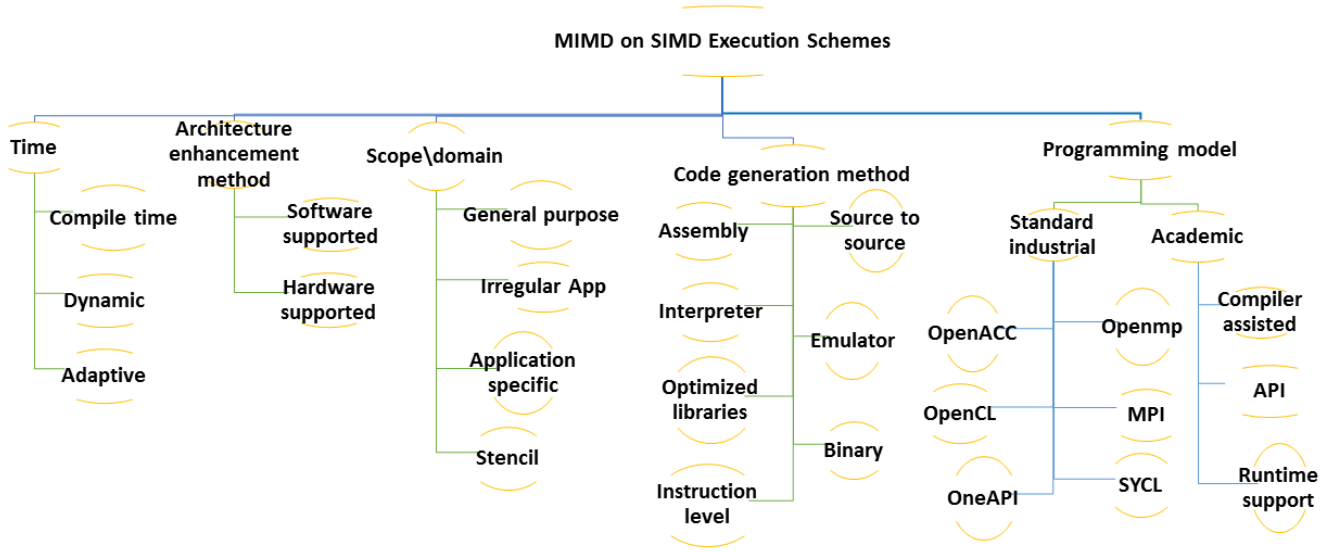


FIGURE 2. Schematic view of MIMD execution on SMD schemes, categorized based on programming model, code generation method, scope, time, and architecture enhancement method.

PPCG [121], and MINT [120] translate C to CUDA. These are directive-based source translators, which only require the programmer to annotate code with basic pragmas about exploiting parallelism and data transfer. PGI [117] translates Fortran, C, and C++ to CUDA.

2) ASSEMBLY CODE APPROACH

Dietz and Roberts [133] used GCC to compile a MIMD application program into a standard assembly language (MIPSEL). Then, a series of transformations is applied to convert this assembly code into a new instruction set that manages GPU local memory as registers. From this code, an optimizing assembler generates a customized interpreter in either NVIDIA CUDA or portable OpenCL.

3) TUNED LIBRARIES APPROACH

With this approach, tuning is done for specific applications on specific platforms. Then, tuned libraries can be used to accelerate other applications. For example, the basic linear algebra subroutines (BLAS) were implemented and tuned on GPUs to form the cuBLAS Library, which is then used to accelerate several applications using GPUs [134]. Several libraries were implemented in CUDA to accelerate a wide range of applications, such as the Fourier transform library (cuFFT), 2D and batched 1D finite-difference/stencil programs (cuSten), and sparse matrices solution methods (cuS-PARSE) [135], [136]. The ARM Compute Library or the ARM Performance Libraries are tuned libraries for the ARM SIMD extension, Neon, which can be imported to accelerate your program. Tuned Libraries as any optimized code can be sensitive to any innocent change in the target architecture, and require retuning.

```
init
do {
   if mask_inst1 then execute inst1
   if mask_inst2 then execute inst2
   if mask_inst3 then execute inst3
   .....
   if mask_instn then execute instn
  } while()
```

FIGURE 3. MIMD SIMD-izing loop.

4) EMULATORS APPROACH

MIMD-like software can be executed on SIMD hardware using a runtime support system implemented as an emulator. Emulating (aka SIMD-izing) an algorithm is very straightforward. Every MIMD program can be transformed into a SIMD program using a loop iterating over all instructions [137], [138]. Figure 3 illustrates the SIMD-ing loop.

Dietz and Cohen used microcode to emulate MIMD on a SIMD microengine [139]. One important detrimental performance issue is control divergence, which can be mitigated with various optimizations utilizing the static nature of the SIMD machine. MIMD multiprocessor simulator that runs on the CRAY-1 is another example of this approach [140].

5) INTERPRETER APPROACH

The implementation of a MIMD program interpreter is simply a SIMD program that interpretively executes a sequence of MIMD instructions on the SIMD platform [26]. Blank presented a MIMD interpreter on a SIMD MasPar MP-1A machine [141]. Nilson and Tanaka [142] presented a MIMD interpreter on a SIMD machine with an adaptive

algorithm. It dynamically optimizes the interpreter loop. Abu-Ghazaleh et al. [143] derived an interpreter with near-optimal variable issue control loops using three heuristic algorithms.

The graphinators execution model [144] simulates a MIMD on SIMD machines by repetitively cycling over the entire instructions set by each SIMD processor. It is dedicated merely to the functional language. Shu and Wu [33] proposed a general method for vectorization and parallelization across multiple application classes, including different irregular application subsets, with a systematic method of detecting their data access patterns.

6) INSTRUCTION-LEVEL APPROACH

Several researchers have studied the instruction-level approach [139], [142], [145], [146], [147], [148]. This approach treats a program as data interpreted by a SIMD machine in parallel across all of the processors. Similar to the interpreter, the SIMD control unit iterates through all instruction sets for each instruction execution to handle all diverging paths. Furthermore, programs with communication require adequate synchronization to verify execution correctness.

To support control divergence and data-dependent branches in a SIMD machine, most approaches used guarded instructions [149]. A SIMD-guarded instruction is executed only if the conditional mask is set to true. The mask is controlled by another instruction. This can reduce potential branch divergences in existing GPU architectures [18], [150].

7) BINARY TRANSLATION

This general approach transforms loops containing short-SIMD instructions to machine-independent IR, applies SIMD optimization techniques at the IR level, and then generates long-SIMD instructions. This dynamic technique enables short-SIMD binaries portability across newer, wider SIMD generations [151]. Spill-aware superword level parallelism (saSLP) [152] exploits the x86 AVX2 host's parallelism, gathers instructions, and registers capacity. To support that, it combines short ARMv8 instructions and registers in the guest binaries.

Levinthal and Porter [153] proposed a dynamic mechanism to handle SIMD branch divergence. When a diverging branch occurs, threads are rescheduled on the fly into new warps. This mechanism uses a stack of execution masks to handle branch divergence on GPUs efficiently. Moy and Lindholm [16] proposed a hazard avoidance technique in a SIMD pipeline through thread serialization at divergence.

B. CLASSIFYING MOS BASED ON TIME

MoS tools can be classified based on the time when a technique is applied. This important factor influences the implementation of optimizations to improve programmability and enable the adaptive execution of programs in SIMD architectures [100]. It may take place at compile-time, runtime, or in any combination. Therefore, we classified MoS tools into static compile-time, runtime, and adaptive techniques.

1) STATIC COMPILE-TIME APPROACH

The compile-time transformations for different SIMD architectures in different studies are fundamentally similar; vector systems and GPUs use similar transformations [98], [154], [155], [156], [157]. Nevertheless, the underlying architectural differences between SIMD platforms raise different challenges as well as approaches to applying these techniques. For example, stride-one memory accesses are enabled by parallel loop transformations such as loop swap and loop collapsing. As a result, the GPU threads can utilize coalesced memory access, optimizing the off-chip memory performance. In contrast, vectorizing compilers use loop interchange to vectorize loops within a single thread.

Min et al. described compiler techniques to enable OpenMP program portability on a software distributed shared memory system [158], [159], and distributed memory systems using MPI message-passing [160]. Other studies proposed mapping OpenMP to cell architectures [161], [162].

A polyhedral model has implemented compile-time transformations that automatically optimize CUDA programs [81]. Main transformations include efficient global memory access and affine loop nest optimizations [131].

ElTantawy and Aamodt [163] implemented static analysis and transformation techniques as an LLVM compiler passes. The scheme examines the program control-flow graph (CFG) to detect and avoid SIMT deadlocks, where the execution of threads in a SIMT architecture becomes stalled or blocked, waiting for each other to release resources and preventing forward progress [164], [165]. Revec [166] is a transparent compiler optimization pass happening at the compiler IR level to rewrite old vectorized code using new vector instructions. It enables the performance portability of hand-vectorized code and execution portability across newer SIMD extension generations.

2) DYNAMIC RUNTIME APPROACH

Dietz and Cohen [139] exploited SIMD parallelism in reductions and stencil computations using an API and a runtime system. The system utilizes knowledge about underlying communication patterns for scheduling the computation and reorganizing the data to improve SIMD parallelism.

Several research studies proposed parallelization techniques utilizing internal kernel launches, where a GPU thread can call another GPU kernel [167]. One example is dynamic thread block launch [168], where the child threads are dynamic instances of the original kernel launched by the CPU (i.e., the parallelism occurs within recursive code). Nested parallelism in CUDA [169] and lazy nested parallelism [167] exploit nested parallelism, where the child threads belong to parallel loops nested within the original kernel launched by the CPU.

Dynamic warp formation is a hardware method to execute SIMD branches on GPUs efficiently without data movement between register lanes [165], [170]. Ren et al. [115] developed a runtime scheduler and an intermediate language to generate a SIMD code for irregular data-traversal applications.

3) ADAPTIVE HARDWARE RECONVERGENCE APPROACH

An adaptive hardware reconvergence scheme maintains MIMD synchronization without changing the program CFG. It eliminates most restrictions inherent in a compiler-only approach. For example, AWARE is an adaptive hardware reconvergence technique that extends the Multi-Path (MP) execution model [171] and improves performance by imposing fewer scheduling constraints.

To enable compiler-driven, adaptive execution of standard OpenMP programs on the underlying GPGPU architectures, Lee [172] proposed a new API called OpenMPC. OpenMPC implements a tuning framework that helps users generate CUDA programs in several optimization variants without deep knowledge of the CUDA programming and memory models.

Irregular applications are not amenable to compile-time optimizations because memory access patterns cannot be analyzed at compile-time. Dynamic performance optimizations are preferred for this situation. Other key issues in parallel applications are computational load balancing and communication cost reduction. To address these issues, Lee [172] has developed an adaptive load-mapping and communication-algorithm-selection system. In [58] and [59], a combined compiler translation and runtime support were proposed to utilize SIMD extension performance.

C. CLASSIFYING MOS BASED ON THE SCOPE/DOMAIN

We classified MoS techniques into four categories based on the scope of applications under optimization: general-purpose, application-specific, stencil applications, or irregular applications.

1) GENERAL PURPOSE APPROACH

Most applications pursue similar patterns [173]. Many SIMD (and SIMT) parallelization efforts have concentrated on specific patterns, such as stencil computations [174], [175], [176] or irregular reductions [61], [177], [178]. Suitable data reorganization approaches for efficient execution can be selected based on the knowledge of individual patterns.

Instead of studying individual problems, Shu and Wu proposed an approach that handles general asynchronous and loosely synchronous problems [33]. Chen et al. [19] proposed a general optimization framework that views irregular applications with indirect memory accesses as sparse matrix computations and effectively optimizes them using tiling to enhance locality, identifying data access patterns, and removing write conflict at both SIMD and MIMD levels. Lee et al. [98] proposed a translator with effective compile-time optimizations for both regular and irregular applications.

2) APPLICATION-SPECIFIC APPROACH

Application-oriented schemes for porting loosely synchronous and asynchronous on SIMD machines are popular [179], [180], [181], [182]. Pennycook et al. implemented a single application (Moldyn) on the Xeon Phi, using both MIMD and SIMD parallelism [72]. Other studies have described domain-specific techniques to execute irregular applications on GPUs. Such techniques include N-body simulation [183], graph algorithms [184], [185], [186], [187], graphics rendering [188], and data-flow analysis [189]. To fill the abstraction gap between application-specific algorithms and the CUDA programming model, Sundaram et al. [190] proposed a method for the scalable execution of domain-specific templates on GPUs.

3) STENCIL APPLICATIONS

Acceleration of stencil computations using SIMD vectorization and GPUs is a popular approach [7], [15], [17]. Eichenberger et al. [191] and Nuzman et al. [192] addressed memory alignment for vectorizing stencils using data reorganization methods. A stencil is memory-bounded if its computations have a low arithmetic density over the neighbors (the ratio of total FLOPs divided by the number of total bytes transferred to or from memory). One major issue with stencils is the high cache miss rate, which degrades performance significantly when input sizes get greater than the cache size.

Henretty et al. [175] proposed a system that utilizes short-vector SIMD optimizations and improves data locality. Yuan et al. [193] proposed temporal vectorization for the stencil computation in the iteration space of a loop nest. The scheme gathers data accesses with different time coordinates in one vector. Imperfectly nested loop transformation, optimization, and vectorization were implemented in a polyhedral compiler [194]. Armejach et al. [60] optimized stencil applications for SVE. Habich et al. [195] proposed a block-striped data access pattern heavily depending on the Gather operation on GPUs to optimize the overhead of accessing non-consecutive memory locations. Parallelizing stencil applications on GPUs is strongly correlated to SIMD extensions [136], [174], [196], [197], [198], [199]. Mint [120] translates stencil applications implemented in C to Cuda.

4) IRREGULAR APPLICATIONS

The key issue with irregular code is that both control-flow and memory access patterns may be data-dependent, where the program's runtime behavior is controlled by the input values and cannot be statically predicted. Lee studied compiler-driven runtime tuning systems for the dynamic adaptation of applications onto the underlying execution platform [172]. The proposed runtime tuning system emphasizes parallel, irregular applications. The coalesced memory access

optimization has been discussed for irregular applications on GPU [178], [200], and on Xeon Phi [57], [201], [202], [203], [204], [205].

D. CLASSIFYING MOS BASED ON THE PROGRAMMING MODEL

There are huge efforts to support code portability between CPUs and GPUs by using traditional parallel programming languages for GPU programming. We categorized programing models into four classes: industrial [206], [207], user-defined/research programming models [50], [208], [209], combined MIMD-SIMD, and multi-GPUs programing models.

1) STANDARD INDUSTRIAL PROGRAMMING MODELS

OpenMP [206] is a well-established, convenient industry-standard model widely used for parallel programming on multicore shared memory systems. The OpenMP API has been ported to different computational systems. The motivation is to simplify GPU programming using OpenMP, which is an easier model compared to the CUDA programming model. One major challenge of using high-level OpenMP is that hidden architectural differences among the underlying platforms require different optimization schemes.

In [207], the authors surveyed works that concentrated on making GPUs able to efficiently run parallel programs that were developed targeting MIMD systems. They used either shared memory or message-passing communication with MPI [210]. An automatic OpenMP to GPGPU framework [98] presented a source-to-source translator and related compile-time optimization techniques. In this approach, work is partitioned among participating threads using parallel constructs and data environment directives, annotated with OpenMP pragmas. They are also utilized to map data into underlying memory systems.

2) RESEARCH/USER-DEFINED PROGRAMMING MODELS

Several research studies have proposed new programming models for SIMD architecture. For example, C-For-Metal (CM) [208] is a programming language targeting Intel GPUs with an explicit interface to explore the underlying hardware characteristics, such as SIMD size control, fine-grained register management, and cross-lane data sharing. The CM compiler is responsible for generating appropriate vector instructions from the high-level explicit constructs. While it is harder to program, this model provides higher throughput for programs with mixed serial/parallel execution and irregular memory accesses [211]. Tian et al. [209] supported function calls by an extension to the directive vectorization methods. The ISPC compiler [50] provided a compiler-based scheme supporting control-flow, data structure, and function calls.

3) COMBINED MIMD-SIMD PARALLELISM

This approach considered the parallelism of nested loop structures, in which a parallel outer loop may include irregular data access with control divergence and inner loops have regular data access. Gerzhoy et al. [212] parallelized the outer loop and scheduled it on a multicore CPU, while the inner regular loop(s) can be scheduled on the GPU cores, increasing its utilization.

In most recent accelerators [213], [214], [215], [216], which generally target standard convolution operation, PEs mainly operate in a SIMD mode. The convolution windows in convolution operations follow a regular pattern, and the number of operations for each of these windows stays constant. GANAX [217] is a unified SIMD-MIMD architecture that operates on convolution windows with varying computation patterns in a MIMD mode and works in a SIMD mode on convolution windows with a regular pattern.

4) MULTI-GPU PROGRAMMING MODEL

To decrease the programming effort, Domonkos and Jakab proposed a scalable multi-GPU programming model that enables architectural optimizations and virtualization of hardware resources [218].

GPU-SM [219] proposed programming multi-GPU systems similar to NUMA shared memory systems, maintaining minimal performance overheads. GPU-STM (software transactional memory) [220] is a novel system that accelerates programs with dynamic data sharing on GPU architecture. Scalability over the massive multithreading of GPUs and preventing livelocks are two major challenges. PyTorch-Direct, [221] enables GPU-centric data accessing paradigm directly without CPU intervention. Groute, [222] is an asynchronous multi-GPU programming model and runtime environment implemented over CUDA for NVIDIA-based multi-GPU nodes. To reduce communication time and further increase GPU utilization, Choi et al. [223] supported computation-communication overlap and integrated GPU-aware communication into asynchronous tasks.

E. MOS CLASSIFICATION BASED ON ARCHITECTURE ENHANCEMENT METHOD

Some studies proposed a slight modification to the current SIMD architecture to enhance MMD execution performance. Others used software layers, leaving hardware intact, or a mix of both.

1) HARDWARE SUPPORTED MODIFICATIONS (HARDWARE SOLUTION)

Several studies have recognized the performance implications of branch divergence in GPUs [165], [171], [224], [225], [226], [227], [228], [229]. However, less work has investigated the functional implications. Eltantawy and Aamodt [230] proposed a hardware warp scheduling policy that temporarily deprioritizes warps executing busy wait codes. Furthermore, they proposed a hardware mechanism that detects busy-wait synchronization on GPUs.

Temporal-SIMT [231], [232], proposed a flexible hardware-supported placement of barriers that minimizes the

TABLE 4. Sources classification.

Category	Techniques	References
Compiler optimization Techniques	Branch prediction	[81], [157], [191]
	Instruction fetch	[157]
	Automatic codes Generation on SIMD units	[115], [157]
	Data\code partitioning across the multiple processor elements	[234]
	Automatic Simdization techniques	[191], [248]–[253]
General directive-based models	directive-based GPU programming models	[84], [96], [98], [109], [116]–[120]
	directives, library routines, and designated compilers	[84], [254]
Execution Schemes	instruction-level approach	[139], [142], [145]–[147], [147]
	Interpreter	[143], [207]
	MIMD microcode on SIMD hardware	[210]
	Source-to-source translation	[84], [96], [98], [108], [109], [112], [116], [117], [117]–[124], [255]
Runtime systems	Dynamic Warp Formation	[165], [170]
	MIMD emulator on SIMD	[33], [103], [137]–[139], [222], [256]
	ADAPTIVE	[172]
	GPU runtime code generation	[106]–[108], [111], [248], [257]
	simdize loops with runtime support	[58], [59], [115], [156], [242], [248], [258]
programming model	OpenMP or MPI programming model	[96]–[99], [102]
	SIMD, SPMD programming language, compiler	[259]
	C-for-Metal (user defined)	[208]
	HIF	
	SYCL, ONEAPI	
	OpenACC,	[84]
	OpenMP for Accelerators	[254]
	C to Cuda translators	[109], [120], [121]
	User-assisted tuning techniques	[88]
	Static compiler approach	[163]

TABLE 4. *(Continued.)* Sources classification.

HW vs SW realization	Implicit hardware-managed cache hierarchies.	[239], [260]
	software-managed streaming memories	[58], [59], [241]
Irregular applications	control divergence	[19], [98], [139], [212], [212], [221], [261]–[266]
	Memory divergence	[201], [267]
	compile-time	[98], [262]
	dynamic schemes	[115], [239], [242], [256], [258], [263]
Scope, domain	application-oriented (graph, data-flow analysis, N-body simulation and graphics rendering)	[179]–[189]
	general purpose	[87]
	general asynchronous and loosely synchronous	[33], [81], [98]
	The regular affine loop index	[81], [131]
Application-specific Solutions, tuned libraries	Biology	[102]
	sparse matrix-matrix computations	[268]
	Kinetic Fluid Simulation	[269]
	nonlinear programming	[270], [271]
	Robotics	[179]–[182], [272]
	Deep neural network, tensor signal processing	[131], [260]

divergence effect on the SIMD units utilization. However, SIMT deadlocks can still happen when using explicit reconvergence points in Temporal-SIMT. Dynamic Warp Formation [165] avoids SIMT deadlocks through flexible thread scheduling, at the cost of higher hardware complexity.

To eliminate branch divergence, Krashinsky et al. [233] proposed vector-fetch and thread-fetch instructions mechanisms in the vector-thread architecture. With Vectorfetch, all processors get the same instruction block for execution. At execution divergence, a single processor can issue a thread fetch to get an atomic instruction block. If each processor is fetching a different atomic instruction block, instruction bandwidth can significantly degrade performance.

Fung et al. [170] added a stack to execute distinct program paths on different SIMD PEs after a diverging branch, which can become a bottleneck in this approach. Levinthal and Porter [153] used a stack of execution masks to handle SIMD branch execution efficiently on GPUs. When a diverging branch occurs, the mechanism reschedules threads into new

warps on the fly. Ramamurthy [164] described hardware to avoid possible deadlocks that modify the behavior of the reconvergence stack when executing lock or unlock instructions. This approach supports very restricted cases and fails if the locking happens in diverged code. Multiple SIMD Multiple Data (MSMD) [234] proposed flexible SIMD datapaths to handle divergence through repartitioning those datapaths among multiple control-flow paths.

To support synchronization on GPUs smoothly, both hardware and software transactional memory approaches have been proposed [220], [235]. In [236], the authors described hardware support with synchronization APIs for a blocking synchronization scheme on GPGPU. This scheme could mitigate SIMT deadlocks in restricted cases [164]. Li et al. proposed a fine-grained inter-thread synchronization mechanism using GPUs' shared memory [237]. Avoiding SIMT-induced deadlocks must be handled by programmers.

Thuerck [238] proposed a hardware and compiler-supported extension to CUDA's programming that handles data and control-flow irregularities. Major architectural enhancements were implemented and tested in a simulation environment. Libra [239] is a customizable, dynamic configurable SIMD accelerator that can adapt its execution scheme to the running program. With shared control architecture [240], the control unit is shared across all PEs executing the same operation.

2) SOFTWARE-SUPPORTED ENHANCEMENTS

To easily adapt MIMD code to a SIMT platform, the software lock stealing and virtualization approach is proposed [241]. It changes the SIMT execution model to be effectively compatible with the MIMD execution model. Their techniques reduced the memory cost of fine-grain locks and avoided circular locking among GPU threads [241]. Locks can't be obtained in a loop due to SIMT deadlocks.

Several studies improved SIMD systems for irregular applications to adapt to varying resource requirements by offering a convenient way to set the SIMD vector length at runtime, using both software [58], [59] and hardware approaches [60]. Other studies in-vestigated hand-optimizing irregular applications on SSE and other vector units [25], [175]. Ren et al. [242] designed a virtual machine as well as a domain-specific bytecode mechanism to SIMD-ize programs that traverse irregular data structures.

Several GPU verification tools focus on finding data races and divergence freedom in GPU kernels [243], [244], [245], [246]. However, they did not consider SIMT deadlock issues due to conditional loops. Habermaier and Knapp [247] provided formal semantics for NVIDIA's stack-based reconvergence mechanism and a definition for the scheduling unfairness issue without providing ways to detect or prevent it.

F. CLASSIFYING SOURCES

Sources classification according to several criteria discussed in previous sections is summarized in Table 4. Categories are overlapped, and techniques used in each category are highlighted. This table serves as an index of topics for readers to easily retrieve articles related to a specific topic.

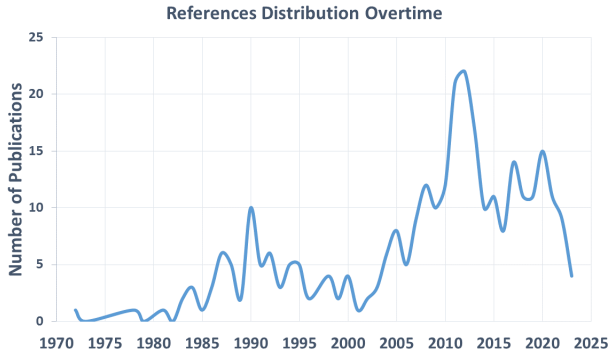


FIGURE 4. Distribution of surveyed publications over the years.

Figure 4 shows the distribution of the selected publications over the years. While publications were carefully selected based on relevance, time distribution provides a good indicator of the progress of SIMD hardware and the associated programming model. For example, you can see two peaks in Figure 4 that reflect the rising attention this subject has received. The first one started in 2008 (CUDA toolkit release date) and continued through 2010 (Xeon Phi architecture release date) until it reached its peak in 2012. It is important to notice that around the 1990s, the subject received a decent amount of interest (the advent of connection machines and multimedia extensions).

VI. REVISITING IRREGULAR APPLICATIONS: CASE STUDY

SIMD accelerators favor classical scientific computing, such as regular structure, large trip count loops, and few data dependencies. However, a high percentage of applications are irregular with indirect memory accesses and sparse data structures, such as a graph, an unstructured mesh, or a sparse matrix. Several studies have described domain-specific techniques to address irregular dataflow on GPUs [183], [184], [185], [186], [187], [188], [189]. The following subsections provide a detailed review of irregular applications.

A. SOURCES OF IRREGULARITIES

Irregular applications stem from both scientific fields (molecular simulations, climate modeling, rational materials design, and bioinformatics) and problem-solving approaches (i.e., simulated annealing, parameter sweeps, uncertainty quantification, and branch-and-bound optimizations). Extensive work has studied improving data locality in irregular applications using both compiler and dynamic schemes [262], [263], [264], [265], [266], [267].

With iteratively irregular applications, several iterations follow the same access pattern. The vital challenge in parallelizing these applications is that the memory data access pattern can't be analyzed at compile-time and can only be known at runtime. Inspecting irregular accesses at runtime resolves them accurately and analyzes the precise mapping

of accesses to iterations. Hence, irregular loop iterations can be partitioned depending on the position of the shared data accessed, optimizing memory performance.

B. CLASSES OF IRREGULAR APPLICATIONS

Irregular applications can be classified into three important subclasses: irregular reductions (MOLDYN kernel from CHARMM [273]), graph algorithms (depth-first tree search), and sparse matrix-vector multiplication (Equake [273] and CG [274]). Two NAS benchmarks, CG and IS, have irregular accesses. Complex instances of inherently asynchronous problems can be observed in non-numeric applications, such as the highly irregular N-Queen problem, which does not suit SIMD implementation. This classification is not exclusive. the following subsections discuss the three classes.

1) IRREGULAR REDUCTIONS

Unstructured grids or molecular dynamics cause irregular reductions. The connectivity of nodes in unstructured grids cannot be determined by node positions (coordinates). Hence, nodes are connected by edges explicitly [275]. Molecular interactions follow a similar pattern [276].

MolDyn is a molecular dynamics kernel that computes the interaction between particles to influence their positions, forces, and velocities. The force acting on each particle is calculated from the sum of each of the forces the other particles impart on it. It involves irregular accesses and is limited by the significant computational complexity of the algorithm. The other time-consuming part is computing the neighbors' graph of interacting particles [72], [273].

2) IRREGULAR DATA STRUCTURES: GRAPHS AND TREES

Graph mining has seen a surge in interest, especially in the context of machine learning and social network graphs [12], [36], [178], [242]. Graphs are an essential data structure for representing dependencies among entities. Random forests, regular expression matching, and B+ trees are popular algorithms that work on irregular data structures [115]. Graph applications include breadth-first search, single-source shortest path computation, Delaunay mesh refinement, pointer analysis, and survey propagation.

Many irregular graph applications make unpredictable, data-dependent accesses to complex data structures (graphs and trees) [184], [185], [186], [187], [272]. The memory-access patterns are usually not analyzable at compile-time, which leads to uncoalesced memory accesses that drastically reduce performance [277]. Similar to irregular reductions, a graph can be viewed as a sparse adjacency matrix because graphs are usually sparse. Therefore, several graph algorithms can be summarized as a computation of sparse matrix-vector multiplication (SpMV) [175].

3) SPARSE MATRIX-VECTOR MULTIPLICATION (SPMV)

The SMVP is the most time-consuming subroutine of the Equake (earthquake simulation) application, which is called at every timestep to compute the product of the grid coordinates and displacements in time. While this computation accesses the displacement vector irregularly, the memory access pattern stays the same through all iterations. All threads in the previous timestep read the vector blockwise for the next step.

The most time-consuming part of CG (conjugate gradient) is the `conjgrad` subroutine, which spends most of its time computing the product of a sparse matrix A (accessed blockwise) and a vector p (irregularly read).

Some studies optimized SpMV for non-SIMD architectures. Strout et al. [278], [279] used rows/columns permutation and tiling to optimize the data locality for irregular kernels like Gauss-Seidel on traditional CPU architectures. Other studies used a blocked coloring method to optimize the ICCG (Incomplete cholesky Conjugate Gradient) solver for multi-threaded execution [280], vectorize the computation of CG [281], or vectorize unstructured 3D mesh computations [282]. Such techniques can be used on SIMD platforms.

VII. EVALUATION OF MOS: METRICS AND BENCHMARKS

A key component of common parallelization paradigms like MPI and OpenMP is performance analysis. Given the large number of threads involved and the variations in data movement and storage, it plays a vital role when working with mixed traditional paradigms and CUDA. In these situations, profile analysis tools (OpenMP, MPI, and CUDA) that can track thread behavior at various levels are crucial.

The majority of surveyed MoS studies used execution time as a primitive performance metric for empirically evaluating their proposed methodologies. Other quantitative, derived metrics include speedup, scalability, and efficiency (ratio of achieved throughput to peak performance). Furthermore, some evaluations used bandwidth and cache-related performance counters, especially for memory-bounded applications. Few studies implemented and evaluated its methods in a simulation environment with a performance model [212]. Table 5 lists the most popular SIMD platforms covered in the surveyed studies.

Burtscher et al. [293] quantified memory-access (MA) and control-flow (CF) irregularities in GPUs. They found that applications demonstrate a variable but consistent degree of irregularity. The degree of control-flow and memory-access irregularities is defined based on the performance counter metrics as follows:

$$\text{irregularity}_{CF} = \frac{\text{divergent branches}}{\text{executed instructions}} \tag{1}$$

$$\text{irregularity}_{MA} = \frac{\text{replayed instructions}}{\text{issued instructions}} \tag{2}$$

where: $irregularity_{CF}$ is the percentage of divergent branches in the executed instructions. $irregularity_{MA}$ is the percentage of the issued instructions that are replayed. A higher percentage value infers more irregularity. Performance scaling is bottlenecked by low arithmetic intensity (AI),

TABLE 5. Indexing sources based on the SIMD platforms covered in surveyed studies.

Target platform	Related studies that considered That platform
NVIDIA GPU	[63], [77]–[81], [87], [96], [98], [99], [102], [103], [106], [109], [111]–[113], [115]–[117], [125], [133], [163], [165], [167]–[169], [171], [172], [177], [178], [190], [197]–[200], [207], [218], [220], [221], [225], [228]–[230], [232], [235]–[237], [241], [243]–[247], [268]–[270], [283]–[285]
Xeon Phi	[19], [57], [72], [194], [203]–[205], [256], [267], [282], [286], [287], [287]
Intel SSE family, AVX family	[59], [115], [115], [151], [152], [175], [193], [195], [242], [255], [267], [282], [288], [289]
ARM SVE, NEON	[58]–[60], [151]
AMD 3DNow!	[288]
Motorola's AltiVec, PowerPC AltiVec	[5], [17]
IBM's BlueGene/L SIMD	[288]
TSUBASA	[64], [65], [290]
Multi-GPUs	[218]–[223]
GPU+ Xeon Phi+ CPU	[202], [291], [292]

TABLE 6. Benchmarks used in the surveyed methodologies evaluation with varying degrees of vector parallelism and irregularities.

Benchmark	References
Stencil Computation (Jacobi, Heat3D, Sobel, Gauss-Seidel)	[59], [60], [98], [125], [139], [174]–[176], [191]–[194], [196]–[199], [256], [285]
NAS Parallel Benchmarks suite	[98], [151], [172], [203], [285], [294]
STREAM	[12], [65], [295]
Rodinia Benchmark Suite	[102], [163], [172], [230], [285], [296]
Irregular Reduction(Euler, Moldyn Kmeans Molecular Dynamics)	[19], [256], [267]
SIMD image processing library	[152]
SPEC OMP2001, SPEC OMP2012	[57], [212]
SPECfp2000, SPECfp95, Media-Bench	[58]
LonestarGPU benchmark suite	[293]
Cuda	[170]
OpenCL SDK	[163]
SPEC CPU2006	[170]
SPLASH2	[170]
KILO TM	[163]
Polybench1.0	[59]
MATPOWER, PGLIB	[270]
Naive Bayes Classifier	[256]
Matrix-matrix mul	[65]
Graph Traversal	[19], [115], [267], [285]
SPMUL	[19]

calculated as:

$$\text{AI} = \frac{\text{Flops}}{\text{Bytes read from DRAM}} \tag{3}$$

The ratio of active warps to the maximum number of warps that the specified NVIDIA architecture can support is known as CUDA occupancy. Too low occupancy typically means low performance, as the memory latency masking is limited.

Power efficiency metrics include the energy-delay product (EDP) and energy-delay2 product (EDP2) that emphasize performance [289]. If the EDP for a given program execution is broken up into a series of distinct instruction intervals, then the overall EDP can be computed by the formula:

$$\text{EDP} = \left(\sum_{i=1}^{n} E_i\right)\left(\sum_{i=1}^{n} t_i\right) \tag{4}$$

where E_i and t_i represent the energy and time spent for the ith interval. The lower the EDP, the better the power efficiency.

The performance achieved per consumed watt of a particular architecture is measured by the metric MFlops per watt ratio, which combines throughput and power efficiency by the formula:

$$\text{MFLOPs/Watt} = \frac{\text{Throughput}}{\text{Power consumption}} \tag{5}$$

Benchmarking is the de facto standard for evaluating hardware architectures in academia and industry. Table 6 lists popular benchmarks that contain varying degrees of vector parallelism and irregularities and were used in the evaluation of the surveyed methodologies.

Daga et al. [297] and Spafford et al. [298] have measured the performance benefits of heterogeneous microprocessors, mainly AMD's Fusion architecture using traditional GPU benchmarks [299], [300]. They investigated how the communication bottlenecks were addressed in CPU-GPU integration [301].

Some studies reported a qualitative evaluation of factors such as ease of use, applicability, scope of use, and usability. Applicability is related to the spectrum of applications that are amenable to the methodology. Usability is related to the source code programming languages. For example, directive-based tools provide more flexibility in adding annotations to the CPU source code, which makes them more applicable in handling complex CPU algorithms. Nevertheless, the user must annotate the parallelization block and maintain the complicated memory hierarchy by themselves.

Lee and Vetter [285] conducted a comprehensive evaluation of existing directive-based GPU programming models (PGI Accelerator [117], HMPP [118], OpenMPC [96], R-Stream, OpenACC [84]) from both industry and academia. Khalilov & Timoveev [302] evaluated CUDA, OpenACC, and OpenMP programming models.

VIII. SUMMARY OF OBSERVATIONS AND FUTURE DIRECTIONS

This section summarizes our key observations based on our survey of many different MIMD on SIMD execution approaches:

- Several MoS optimization techniques that were implemented in the earlier work and were found beneficial are the most important today. However, the performance sensitivity to micro-architectural details of modern SIMD platforms requires precise attention by a software engineer implementing an MoS technique.
- While SIMD platforms are omnipresent, their unbridled computation power is still not fully utilized in mainstream program performance.

- GPGPU offers a low-cost, power-efficient, massively parallel platform to software developers. However, programmers struggle with its challenging, complex programming model.
- While GPU architecture provides more scalable performance, it is more complex and restrictive, and the execution model is difficult to program.
- To decrease the programming efforts, compiler extensions or metaprogramming techniques can be used to abstract the complex execution environment from the programmer.
- Compiler-based systems use a high-level (OpenMP) interface with automatic translation to simplify creating CUDA programs. They also implement many compiler transformations and optimization techniques to fill the performance gap between hand-optimized CUDA programs and auto-generated codes.
- A significant performance gap is observed between high-level (OpenMP, OpenACC) and low-level (CUDA) programming models, especially with the growing complexity of code.
- Source-to-source vectorization reforms the applicability of auto-vectorization and the difficulty of manual vectorization with comparable performance.
- Runtime analysis is required for irregular applications to resolve the challenging memory access patterns. Such patterns cannot be analyzed accurately at the compile-time, and conservatively overestimates data consumption.
- GPGPU performance can be significantly impaired (degraded) by control divergence, memory bandwidth, and limited parallelism. They are critical factors that may significantly bottleneck the performance of vectorized code.
- Handling divergence mechanism can be implemented with a sequence of hardware actions performed inside SIMD processing in any computational platform.
- For a more comprehensive evaluation, Researchers should investigate new and potentially more complex applications that can be enabled by nested parallelism using CPU-GPU integration and multi-GPUs.
- Dedicated accelerators for AI applications are increasingly gaining wide interests. Many software techniques discussed in this survey might need to be implemented in hardware for maximized performance.
- Future accelerators are expected to feature more specialized hardware designs tailored specifically for natural language processing tasks. This could involve custom architectures optimized for the unique requirements of LLMs, allowing for faster and more energy-efficient processing. As the field continues to advance, innovations in hardware will play a crucial role in supporting the next generation of large and complex language models.

IX. CONCLUSION

This research presented a comprehensive survey of parallel programming models and execution frameworks to facilitate executing parallel programs on a SIMD architecture (MoS). The rapid evolvement of SIMD architectures, as well as the huge amount of literature regarding this hot topic for more than three decades, made the survey challenging and necessitated. A comparative survey of various MoS tools, benchmarks, and SIMD platforms was conducted. Several GPU programming paradigms and API were surveyed and classified into different groups based on their criteria. This study aims to provide a comprehensive reference for new researchers and developers in the field of computer architecture and parallel computing-intensive scientific applications.

ACKNOWLEDGMENT

The authors acknowledge anonymous reviewers for their valuable suggestions.

REFERENCES

[1] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 67–70, 1996.
[2] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.
[3] AMD. (2000). *3DNOW! Technology Manual*. Motorola, Chicago, IL, USA. Accessed: Sep. 2022.
[4] ARM. *Neon Programmers' Guide*. Accessed: 2023. [Online]. Available: https://documentation-service.arm.com/static/63299276e68c6809a6b41308
[5] S. Fuller, "Motorola's AltiVe technology," *White Paper*, vol. 6, p. 998, May 1998.
[6] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Intel Corp., Santa Clara, CA, USA, 2016.
[7] *Intel® 64 and IA-32 Architectures Software Developer Manuals Volume 2A: Instruction Set Reference*, Intel Corp., Santa Clara, CA, USA, 2016.
[8] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017.
[9] A. Sodani, "Knights landing (KNL): 2nd generation Intel® Xeon Phi processor," in *Proc. IEEE Hot Chips 27 Symp. (HCS)*, Aug. 2015, pp. 1–24.
[10] T. Yoshida, "Introduction of Fujitsu's HPC processor for the post-K computer," in *Proc. Hot Chips 28th Symp.*, Aug. 2016, pp. 1–5.
[11] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proc. 29th Annu. Conf. Comput. Graph. Interact. Techn.*, Jul. 2002, pp. 703–712.
[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *Proc. ACM SIGGRAPH Papers*, Aug. 2004, pp. 777–786.
[13] M. Shebanow, *Programming Massively Parallel Processors Lecture 12*, document ECE 498 Al, 2007.
[14] R. A. Lorie and H. R. Strong, "Method for conditional branch execution in SIMD vector processors," U.S. Patent 4 758 435, Mar. 6, 1984.
[15] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41–51, Mar. 2005.
[16] S. Moy and E. Lindholm, "Method and system for programmable pipelined graphics processing with branching instructions," U.S. Patent 6 047 947, Sep. 20, 2005.
[17] D. Luebke and G. Humphreys, "How GPUs work," *Computer*, vol. 40, no. 2, pp. 96–100, Feb. 2007.
[18] *NVIDIA CUDA (Compute Unified Device Architecture) Programming Guide 3.1*, NVIDIA Corp., Santa Clara, CA, USA, 2010.
[19] L. Chen, P. Jiang, and G. Agrawal, "Exploiting recent SIMD architectural advances for irregular applications," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2016, pp. 47–58.
[20] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, Mar. 2020.

[21] M. Khairy, A. G. Wassal, and M. Zahran, "A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity," *J. Parallel Distrib. Comput.*, vol. 127, pp. 65–88, May 2019.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[23] W. D. Hillis, *The Connection Machine*. Cambridge, MA, USA: MIT Press, 1989.

[24] J. Nickolls, "The design of the MasPar MP-1: A cost effective massively parallel computer," in *Proc. 35th IEEE Comput. Soc. Int. Conf. Intellectual Leverage*, Mar. 1990, pp. 25–28.

[25] T. Bridges, "The GPA machine: A generally partitionable MSIMD architecture," in *Proc. 3rd Symp. Frontiers Massively Parallel Comput.*, Jan. 1990, pp. 196–203.

[26] C. C. Weems, E. M. Riseman, and A. R. Hanson, "Image understanding architecture: Exploiting potential parallelism in machine vision," *Computer*, vol. 25, no. 2, pp. 65–68, Feb. 1992.

[27] J. D. Allen and D. E. Schimmel, "The impact of pipelining on SIMD architectures," in *Proc. 9th Int. Parallel Process. Symp.*, Apr. 1995, pp. 380–387.

[28] G. Fox, "What have we learnt from using real parallel machines to solve real problems?" Caltech Concurrent Comput. Program, California Inst. Technol., Pasadena, CA, USA, Tech. Rep. C3P-522, 1989.

[29] J. L. Hennesy and D. A. Patterson, *Computer Architecture a Quantitative Approach*, 6th ed. San Mateo, CA, USA: Morgan Kaufmann, 2017.

[30] B. Parhami, "SIMD machines: Do they have a significant future?" *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 4, pp. 19–22, Sep. 1995.

[31] B. C. Kuszmaul, "Simulating applicative architectures on the connection machine," M.S. thesis, MIT, Cambridge, MA, USA, 1986.

[32] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 170–183, 1986.

[33] W. Shu and M.-Y. Wu, "Asynchronous problems on SIMD parallel computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 7, pp. 704–713, Jul. 1995.

[34] P. Kacsuk and A. Bale, "DAP Prolog: A set-oriented approach to prolog," *Comput. J.*, vol. 30, no. 5, pp. 393–403, Oct. 1987.

[35] M. Nilsson and H. Tanaka, "Massively parallel implementation of flat GHC on the connection machine," in *Proc. Int. Conf. 5th Generat. Comput. Syst.*, 1988, pp. 1031–1039.

[36] F. Darema-Rodgers, D. A. George, V. A. Norton, and G. F. Pfister, "Environment and system interface for VM/EPEX," IBM Thomas J. Watson Res. Center, New York, NY, USA, Tech. Rep. RCLL381, 1985.

[37] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," *Parallel Comput.*, vol. 7, no. 1, pp. 11–24, Apr. 1988.

[38] L. H. Jamieson, "Characterizing parallel algorithms," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. Cambridge, MA, USA: MIT Press, 1987, pp. 65–100.

[39] R. F. Freund, "Optimal selection theory for superconcurrency," in *Proc. ACM/IEEE Conf. Supercomputing*, Nov. 1989, pp. 699–703.

[40] E. C. Bronson, T. L. Casavant, and L. H. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 2, pp. 195–205, Apr. 1990.

[41] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, "Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting," *J. Parallel Distrib. Comput.*, vol. 11, no. 3, pp. 239–251, Mar. 1991.

[42] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, H. E. Smalley, and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, no. 12, pp. 934–947, Dec. 1981.

[43] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, Eds. Washington, DC, USA: IEEE Computer Society Press, 1987, pp. 387–407.

[44] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons*. New York, NY, USA: Wiley, 1987.

[45] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas reconfigurable array computer," in *Proc. Nat. Comput. Conf.*, May 1980, pp. 631–641.

[46] M. Auguin and F. Boeri, "The OPSILA computer," in *Parallel Languages and Architectures*, M. Consard, Ed. Holland, U.K.: Elsevier, 1986, pp. 143–153.

[47] M. Auguin, F. Boeri, J. P. Dalban, and A. Vincent-Carrefour, "Experience using a SIMD/SPMD multiprocessor architecture," *Microprocess. Microprogram.*, vol. 21, nos. 1–5, pp. 171–177, Aug. 1987.

[48] P. Duclos, F. Boeri, M. Auguin, and G. Giraudon, "Image processing on a SIMD/SPMD architecture: OPSILA," in *Proc. 9th Int. Conf. Pattern Recognit.*, Jan. 1988, pp. 430–433.

[49] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Apr. 2011, pp. 141–150.

[50] M. Pharr and W. R. Mark, "ISPC: A SPMD compiler for high-performance CPU programming," in *Proc. Innov. Parallel Comput. (InPar)*, May 2012, pp. 1–13.

[51] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 530–543, Feb. 2020.

[52] A. Poenaru, "Modern vector architectures for high-performance computing," M.S. thesis, Dept. Comput. Sci., Univ. Bristol, Bristol, U.K., 2022.

[53] *ARM C Language Extensions for SVE*, ARM Ltd., Cambridge, U.K., 2020.

[54] P. Papaphilippou, K. Paul H. J., and W. Luk, "Simodense: A RISC-V softcore optimised for exploring custom SIMD instructions," in *Proc. 31st Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2021, pp. 391–397.

[55] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[56] J. Reinders. (2017). *Intel AVX-512 Instructions*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html

[57] D. Mustafa, "Performance evaluation of massively parallel systems using SPEC OMP suite," *Computers*, vol. 11, no. 5, p. 75, May 2022.

[58] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, Feb. 2007, pp. 216–227.

[59] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks, "Vapor SIMD: Auto-vectorize once, run everywhere," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Apr. 2011, pp. 151–160.

[60] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Stencil codes on a vector length agnostic architecture," in *Proc. 27th Int. Conf. Parallel Architectures Compilation Techn.* New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 1–12.

[61] S. Kim and H. Han, "Efficient SIMD code generation for irregular kernels," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 55–64, Sep. 2012.

[62] ATI. (2008). *ATI Stream Computing SDK User Guide V1.3-Beta*. [Online]. Available: http://developer.amd.com/gpu-assets/Stream-Computing-Overview.pdf

[63] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M.-W. Hwu, "An adaptive performance modeling tool for GPU architectures," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 105–114, May 2010.

[64] Y. Yamada and S. Momose, "Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA," in *Proc. Int. Symp. High Perform. Chips*, Aug. 2018, pp. 1–25.

[65] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance evaluation of a vector supercomputer SX-aurora TSUBASA," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, vol. 54, Nov. 2018, pp. 685–696.

[66] R. Egawa, K. Komatsu, S. Momose, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, "Potential of a modern vector supercomputer for practical applications: Performance evaluation of SX-ACE," *J. Supercomput.*, vol. 73, no. 9, pp. 3948–3976, Sep. 2017.

[67] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and R. Boyle, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 1–12.

[68] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, May 2018.
[69] A. Ravikumar, H. Sriraman, P. M. S. Saketh, S. Lokesh, and A. Karanam, "Effect of neural network structure in accelerating performance and accuracy of a convolutional neural network with GPU/TPU for image analytics," *PeerJ Comput. Sci.*, vol. 8, p. e909, Mar. 2022.
[70] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, pp. 67–78, Jun. 2020.
[71] I. Burstein, "Nvidia data center processing unit (DPU) architecture," in *Proc. IEEE Hot Chips 33th Symp. (HCS)*, Aug. 2021, pp. 1–20.
[72] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon Phi coprocessors," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 1085–1097.
[73] D. Blythe, "The Xe GPU architecture," in *Proc. IEEE Hot Chips 32th Symp. (HCS)*, Aug. 2020, pp. 1–27.
[74] (2023). *BrookGPU Home Page*. [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/
[75] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *Proc. 5th IEEE Int. Symp. Biomed. Imag., Nano Macro*, May 2008, pp. 836–838.
[76] M. Gerndt, "Automatic performance analysis tools for the Grid," *Concurrency Comput., Pract. Exper.*, vol. 17, no. 24, p. 99115, 2005.
[77] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-M.-W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, Apr. 2008, pp. 195–204.
[78] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–10.
[79] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Multi-kernel auto-tuning on GPUs: Performance and energy-aware optimization," in *Proc. 23rd Euromicro Int. Conf. Parallel, Distrib., Network-Based Process.*, Mar. 2015, pp. 438–445.
[80] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M.-W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2008, pp. 73–82.
[81] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, Jun. 2008, pp. 225–234.
[82] F. Petrovic, D. Strelák, J. Hozzová, J. Ol'ha, R. Trembecký, S. Benkner, and J. Filipovic, "A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with kernel tuning toolkit," *Future Gener. Comput. Syst.*, vol. 108, pp. 161–177, Jul. 2020.
[83] *The OpenMP Application Programming Interface*, OpenMP Archit. Rev. Board, Beaverton, OR, USA, 2018, p. 666.
[84] *The OpenACC Application Programming Interface*, OpenACC, Bend, OR, USA, 2020.
[85] A. Munshi. (2012). *The OpenCL Specification Version 1.2. Khronos OpenCL Working Group*. [Online]. Available: https://www.khronos.org/files/opencl-spir-12-provisional.pdf
[86] S. G. Gonzalo, S. D. Hammond, C. R. Trott, and W. M. Hwu, "Revisiting online autotuning for sparse-matrix vector multiplication kernels on next-generation architectures," in *Proc. IEEE 19th Int. Conf. High Perform. Comput. Commun., IEEE 15th Int. Conf. Smart City, IEEE 3rd Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Dec. 2017, pp. 72–80.
[87] D. Strelák, C. Ó. S. Sorzano, J. M. Carazo, and J. Filipovic, "A GPU acceleration of 3-D Fourier reconstruction in cryo-EM," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 5, p. 948, 2019, Art. no. 959.
[88] C. Nugteren and V. Codreanu, "CLTune: A generic auto-tuner for OpenCL kernels," in *Proc. IEEE 9th Int. Symp. Embedded Multicore/Many-Core Syst.-Chip*, Sep. 2015, pp. 195–202.
[89] AMD. (2023). *HIP*. [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/latest/
[90] AMD. (2022). *Hip Programming Guide*. Accessed: Mar. 31, 2023. [Online]. Available: https://docs.amd.com/projects/HIP/en/docs-5.3.0/user_guide/programming_manual.html
[91] Y. M. Tsai, T. Cojean, T. Ribizel, and H. Anzt, "Preparing GINKGO for amd GPUs—A testimonial on porting CUDA code to HIP," in *Euro-Par 2020: Parallel Processing Workshops*, B. Balis, Ed. Cham, Switzerland: Springer, 2021, pp. 109–121.
[92] *SYCL™ 2020 Specification (Revision 7)*, Khronos Group, Beaverton, OR, USA, 2020.
[93] B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, X. Tian, and J. Reinders, *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. New York, NY, USA: Apress, 2020.
[94] R. Reyes, G. Brown, R. Burns, and M. Wong, "SYCL 2020: More than meets the eye," in *Proc. Int. Workshop OpenCL*, vol. 4, Apr. 2020, pp. 1–14.
[95] *Intel® OneAPI Toolkits*, Intel Corp., Santa Clara, CA, USA, 2023.
[96] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1–11.
[97] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Effects of compiler optimizations in OpenMP to CUDA translation," in *Proc. Int. Workshop OpenMP*. Berlin, Germany: Springer, 2012, pp. 169–181.
[98] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *Proc. ACM Symp. PPoPP*, 2009, pp. 101–110.
[99] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP for efficient programming and tuning on GPUs," *Int. J. Comput. Sci. Eng.*, vol. 8, no. 1, pp. 4–20, 2013.
[100] D. Mustafa, "A survey of performance tuning techniques and tools for parallel applications," *IEEE Access*, vol. 10, pp. 15036–15055, 2022.
[101] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? On the scalability of OpenMP implementations," in *Euro-Par 2015: Parallel Processing*. Berlin, Germany: Springer, 2015, pp. 451–463.
[102] D. Li, K. Sajjapongse, H. Truong, G. Conant, and M. Becchi, "A distributed CPU-GPU framework for pairwise alignments on large-scale sequence datasets," in *Proc. IEEE 24th Int. Conf. Appl.-Specific Syst., Architectures Processors*, Jun. 2013, pp. 329–338.
[103] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and evaluation of the gemtc framework for GPU-enabled many-task computing," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, Jun. 2014, pp. 153–164.
[104] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: High-level and efficient streaming on multi-core," in *Programming Multi-Core and Many-Core Computing Systems* (Parallel and Distributed Computing). Hoboken, NJ, USA: Wiley, Mar. 2014, pp. 1–3.
[105] G. Rossum. (1994). *The Python Programming Language*. [Online]. Available: https://www.python.org/
[106] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Comput.*, vol. 38, no. 3, pp. 157–174, Mar. 2012.
[107] C. Mueller, B. Martin, and A. Lumsdaine, "CorePy: High-productivity Cell/BE programming," in *Proc. 1st STI/Georgia Tech Workshop Softw. Appl. Cell/BE Processor*, 2007.
[108] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA," in *Euro-Par 2009 Parallel Processing*. Cham, Switzerland: Springer, 2009, pp. 887–899.
[109] T. D. Han and T. S. Abdelrahman, "Hi CUDA: A high-level directive-based language for GPU programming," in *Proc. 2nd Workshop Gen. Purpose Process. Graph. Process. Units*, New York, NY, USA, Mar. 2009, pp. 52–61.
[110] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
[111] Q. Hou, K. Zhou, and B. Guo, "BSGP: Bulk-synchronous GPU programming," in *Proc. ACM SIGGRAPH Papers*, Aug. 2008, pp. 1–12.
[112] G. Noaje, C. Jaillet, and M. Krajecki, "Source-to-source code translator: OpenMP C to CUDA," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Sep. 2011, pp. 512–519.
[113] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O. Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU threads for OpenMP 4.0 in LLVM," in *Proc. LLVM Compiler Infrastructure HPC*, Nov. 2014, pp. 12–21.
[114] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, Mar. 2004, pp. 75–86.
[115] B. Ren, T. Mytkowicz, and G. Agrawal, "A portable optimization engine for accelerating irregular data-traversal applications on SIMD architectures," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, pp. 1–31, Jun. 2014.

[116] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu, "CUDA-Lite: Reducing GPU programming complexity," in *Proc. Int. Workshop Lang. Compil. Parallel Comput. (LCPC)*, 2008, pp. 1–15.

[117] PGI Accelerator. (2009). *The Portland Group, PGI Fortran and C Accelarator Programming Model*. [Online]. Available: https://docs.nvidia.com/hpc-sdk/pgi-compilers/2013/pgirn133.pdf

[118] HMPP. (2009). *HMPP Workbench, a Directive-Based Compiler for Hybrid Computing*. Accessed: Apr. 2, 2012. [Online]. Available: https://www.caps-entreprise.com/hmpp.html

[119] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, New York, NY, USA, Mar. 2010, pp. 51–61.

[120] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing CUDA performance in 3D stencil methods with annotated C," in *Proc. Int. Conf. Supercomputing*, New York, NY, USA, May 2011, pp. 214–224.

[121] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 1–23, Jan. 2013.

[122] P. Yang, F. Dong, V. Codreanu, D. Williams, J. B. T. M. Roerdink, B. Liu, A. Anvari-Moghaddam, and G. Min, "Improving utility of GPU in accelerating industrial applications with user-centered automatic code translation," *IEEE Trans. Ind. Informat.*, vol. 14, no. 4, pp. 1347–1360, Apr. 2018.

[123] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Automatic generation of multicore chemical kernels," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 119–131, Jan. 2011.

[124] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, and P. Villalon, "Par4All: From convex array regions to heterogeneous computing," in *Proc. 2nd Int. Workshop Polyhedral Compilation Techn., Impact*, Dec. 2012, pp. 1–21.

[125] P. H. Lin, C. Liao, D. J. Quinlan, and S. Guzik, "Experiences of using the OpenMP accelerator model to port DOE stencil applications," in *Proc. Int. Workshop OpenMP*. Cham, Switzerland: Springer, 2015, pp. 45–59.

[126] L. Wan, X. Cui, Y. Li, W. Zheng, and X. Yuan, "HeteroPP: A directive-based heterogeneous cooperative parallel programming framework," *Concurrency Comput., Pract. Exper.*, vol. 5, p. e8014, Jan. 2024.

[127] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. MAttson, J. D. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures," in *Proc. 33rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2000, pp. 159–170.

[128] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proc. 31st Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1998, pp. 3–13.

[129] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with streams," in *Proc. ACM/IEEE Conf. Supercomputing*, Nov. 2003, p. 35.

[130] J. A. Stratton, S. S. Stone, and W.-M.-W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Languages and Compilers for Parallel Computing*. Berlin, Germany: Springer, 2008, pp. 16–30.

[131] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, P. Di, K. Zhang, and X. Jin, "AKG: Automatic kernel generation for neural processing units using polyhedral transformations," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, Jun. 2021, pp. 1233–1248.

[132] J. Wang, X. Deng, K.-T.-A. Wang, and Z. Ye, "Adapting SYCL's SIMT programming paradigm for accelerators via program reconstruction," in *Proc. 50th Int. Conf. Parallel Process. Workshop*, Lemont, IL, USA, Aug. 2021, p. 6.

[133] H. G. Dietz and F. Roberts. (2012). *Execution Of MIMD MIPSEL Assembly Programs Within CUDA/OpenCL GPUs*. [Online]. Available: http://aggregate.ece.engr.uky.edu/MOG/icpp12.pdf

[134] NVIDIA. (Nov. 2015). *CuBLAS User Guide*. Accessed: Jan. 2016. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[135] *CUDA Toolkit Documentation. NVIDIA Developer Zone*, NVIDIA Corp., Santa Clara, CA, USA, 2019.

[136] A. Gloster and L. Ó. Náraigh, "CuSten—CUDA finite difference and stencil library," *SoftwareX*, vol. 10, Jul. 2019, Art. no. 100337.

[137] P. Sanders, "Emulating MIMD behavior on SIMD machines," in *Proc. EUROSIM*, 1994, pp. 313–320.

[138] P. Sanders, "Optimizing the emulation of MIMD behavior on SIMD machines," *Math. Res.*, vol. 96, pp. 320–321, Oct. 1996.

[139] H. G. Dietz and W. E. Cohen, "A massively parallel MIMD implemented by SIMD hardware," Dept. School Elect. Eng., Purdue Univ., West Lafayette, IN, USA, Tech. Rep. TR-EE 92-4, 1992.

[140] T. Axelrod, P. Dubois, and P. Eltgroth, "A simulator for MIMD performance prediction: Application to the S-1 MkIIa multiprocessor," *Parallel Comput.*, vol. 1, nos. 3–4, pp. 237–274, Dec. 1984.

[141] T. Blank, "The MasPar MP-1 architecture," in *Proc. 35th IEEE Comput. Soc. Int. Conf. Intellectual Leverage*, Feb. 1990, pp. 20–24.

[142] M. Nilsson and H. Tanaka, "MIMD execution by SIMD computers," *J. Inf. Process.*, vol. 13, no. 1, pp. 58–61, 1990.

[143] N. Abu-Ghazaleh, P. A. Wilsey, X. Fan, and D. Hensgen, "Variable instruction issue for efficient MIMD interpretation on SIMD machines," in *Proc. 8th Int. Parallel Process. Symp.*, Apr. 1994, pp. 304–310.

[144] P. Hudak and E. Hohr, "Graphinators and the duality of SIMD and MIMD," in *Proc. ACM Conf. LISP Funct. Program.*, New York, NY, USA, Jan. 1988, pp. 224–234.

[145] R. J. Collins, "Multiple instruction multiple data emulation on the connection machine," Dept. Comput. Sci., Univ. California, Oakland, CA, USA, Tech. Rep. CSD-910004, 1991.

[146] M. S. Littmari and C. D. Metcalf, "An exploration of asynchronous data-parallelism," Dept. Comput. Sci., Yale Univ., New Haven, CT, USA, Tech. Rep. YALEU/DCS/TR-684, 1988.

[147] P. A. Wilsey and D. A. Hensgen, "Exploiting SIMD computers for general purpose computation," in *Proc. 6th Int. Parallel Process. Symp.*, Mar. 1992, pp. 675–679.

[148] P. A. Wilsey, D. A. Hensgen, N. B. Abu-Ghazaleh, C. E. Slusher, and D. Y. Hollinden, "The concurrent execution of non-communicating programs on SIMD processors," in *Proc. 4th Symp. Frontiers Massively Parallel Comput.*, Jan. 1992, pp. 29–30.

[149] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The illiac IV system," *Proc. IEEE*, vol. 60, no. 4, pp. 369–388, Apr. 1972.

[150] *ATI CTM Guide*, AMD Inc, Santa Clara, CA, USA, 2006.

[151] D.-Y. Hong, Y.-P. Liu, S.-Y. Fu, J.-J. Wu, and W.-C. Hsu, "Improving SIMD parallelism via dynamic binary translation," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 3, pp. 1–27, May 2018.

[152] Y.-P. Liu, D.-Y. Hong, J.-J. Wu, S.-Y. Fu, and W.-C. Hsu, "Exploiting SIMD asymmetry in ARM-to-x86 dynamic binary translation," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, pp. 1–24, Mar. 2019.

[153] A. Levinthal and T. Porter, "Chap—A SIMD graphics processor," *ACM SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 77–82, 1984.

[154] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 4, pp. 491–542, Oct. 1987.

[155] D. Levine, D. Callahan, and J. Dongarra, "A comparative study of automatic vectorizing compilers," *Parallel Comput.*, vol. 17, nos. 10–11, pp. 1223–1244, Dec. 1991.

[156] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, "An integrated simdization framework using virtual vectors," in *Proc. 19th Annu. Int. Conf. Supercomputing*, Jun. 2005, pp. 169–178.

[157] A. E. Eichenbergert, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the CELL processor," in *Proc. 14th Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Sep. 2005, pp. 161–172.

[158] S. Min, A. Basumallik, and R. Eigenmann, "Optimizing OpenMP programs on software distributed shared memory systems," *Int. J. Parallel Program.*, vol. 31, pp. 225–249, Jun. 2003.

[159] S.-J. Min and R. Eigenmann, "Optimizing irregular shared-memory applications for clusters," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, Jun. 2008, pp. 256–265.

[160] A. Basumallik and R. Eigenmann, "Towards automatic translation of OpenMP to MPI," in *Proc. 19th Annu. Int. Conf. Supercomputing*, Jun. 2005, pp. 189–198.

[161] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on cell," *Int. J. Parallel Program.*, vol. 36, no. 3, pp. 289–311, Jun. 2008.

[162] H. Wei and J. Yu, "Loading OpenMP to cell: An effective compiler framework for heterogeneous multi-core chip," in *Proc. Int. Workshop OpenMP (IWOMP)*, 2007, pp. 129–133.

[163] A. ElTantawy and T. M. Aamodt, "MIMD synchronization on SIMT architectures," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–14.

[164] A. Ramamurthy, "Towards scalar synchronization in SIMT architectures," Dept. Elect. Comput. Eng., Univ. British Columbia, Vancouver, BC, Canada, Tech. Rep., 2011, doi: 10.14288/1.0072253.
[165] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 407–420.
[166] C. Mendis, A. Jain, P. Jain, and S. Amarasinghe, "Revec: Program rejuvenation through revectorization," in *Proc. 28th Int. Conf. Compiler Construct.*, Feb. 2019, pp. 29–41.
[167] G. Ozen, "Compiler and runtime based parallelization & optimization for GPUs," Ph.D. dissertation, Dept. Comput. Archit., Universitat Politècnica Catalunya, Barcelona, Spain, 2017.
[168] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 528–540.
[169] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 93–106.
[170] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, pp. 1–37, Jun. 2009.
[171] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A scalable multi-path microarchitecture for efficient GPU control flow," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 248–259.
[172] S. Lee, "Toward compiler-driven adaptive execution and its application to GPU architectures," Ph.D. dissertation, Dept. Elect. Comput. Eng., Purdue Univ., West Lafayette, IN, USA, 2011.
[173] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams, "The landscape of parallel computing research: A view from Berkeley," Dept. EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. EECS-2006-183, 2006.
[174] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. ACM/IEEE Conf. Supercomputing*, Nov. 2008, pp. 1–12.
[175] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector SIMD architectures," in *Proc. CC/ETAPS*. Berlin, Germany: Springer-Verlag, 2011, pp. 225–245.
[176] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," in *Proc. 27th Int. ACM Conf. Int. Conf. supercomputing*, Jun. 2013, pp. 13–24.
[177] X. Huo, V. Ravi, W. Ma, and G. Agrawal, "An execution strategy and optimized runtime support for parallelizing irregular reductions on modern GPUs," in *Proc. Int. Conf. Supercomputing*, May 2011, pp. 2–11.
[178] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2011, pp. 369–380.
[179] T. W. Clark, R. V. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott, "Evaluating parallel languages for molecular dynamics computations," in *Proc. Scalable High Perform. Comput. Conf.*, Williamsburg, VA, USA, Apr. 1992, pp. 1–9.
[180] S. Tomboulian and M. Pappas, "Indirect addressing and load balancing for faster solution to Mandelbrot set on SIMD architectures," in *Proc. 3rd Symp. Frontiers Massively Parallel Comput.*, Oct. 1990, pp. 443–450.
[181] R. V. Hanxleden and K. Kennedy, "Relaxing SIMD control flow constraints using loop transformations," Center Res. Parallel Comput., Houston, TX, USA, Tech. Rep. CRPC-TR92207, 1992.
[182] M. Willebeek-Lemair and A. P. Reeves, "Solving nonuniform problems on SIMD computers: Case study on region growing," *J. Parallel Distrib. Comput.*, vol. 8, no. 2, pp. 135–149, Feb. 1990.
[183] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based Barnes Hut *n*-body algorithm," in *GPU Computing Gems Emerald Edition*. San Mateo, CA, USA: Morgan Kaufmann, 2011, pp. 75–92.
[184] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. 14th Int. Conf. High Perform. Comput.*, 2007, pp. 197–208.
[185] L. Luo, M. Wong, and W.-M. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. 47th Design Autom. Conf.*, Jun. 2010, pp. 52–55.
[186] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2012, pp. 117–127.
[187] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on GPUs," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. parallel Program.*, Feb. 2013, pp. 147–156.
[188] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proc. Conf. High Perform. Graph.*, 2010, pp. 29–37.
[189] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU implementation of inclusion-based points-to analysis," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 107–116.
[190] N. Sundaram, A. Raghunathan, and S. T. Chakradhar, "A framework for efficient and scalable execution of domain-specific templates on GPUs," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–12.
[191] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD architectures with alignment constraints," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2004, pp. 82–93.
[192] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in *Proc. 27th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2006, pp. 132–143.
[193] L. Yuan, H. Cao, Y. Zhang, K. Li, P. Lu, and Y. Yue, "Temporal vectorization for stencils," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Louis, MO, USA, Nov. 2021, pp. 01–14.
[194] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2013, pp. 127–138.
[195] D. Habich, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner, "To use or not to use the SIMD gather instruction?" in *Proc. 18th Int. Workshop Data Manag. New Hardw.*, Jun. 2022, pp. 1–5.
[196] L. Chen, X. Huo, and G. Agrawal, "Scheduling methods for accelerating applications on architectures with heterogeneous cores," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 48–57.
[197] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proc. 26th ACM Int. Conf. Supercomputing*, Jun. 2012, pp. 311–320.
[198] J. Meng and K. Skadron, "A performance study for iterative stencil loops on GPUs with ghost zone optimizations," *Int. J. Parallel Program.*, vol. 39, no. 1, pp. 115–142, Feb. 2011.
[199] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *SC : Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1–13.
[200] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2013, pp. 57–68.
[201] P. Jiang and G. Agrawal, "Conflict-free vectorization of associative irregular applications with recent SIMD architectural advances," in *Proc. Int. Symp. Code Gener. Optim.*, Feb. 2018, pp. 175–187.
[202] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative performance analysis of Intel Xeon Phi, GPU, and CPU," 2013, *arXiv:1311.0378*.
[203] A. Ramachandran, J. Vienne, R. V. D. Wijngaart, L. Koesterke, and I. Sharapov, "Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi," in *Proc. 42nd Int. Conf. Parallel Process.*, Oct. 2013, pp. 736–743.
[204] S. Jha, "Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach," in *Proc. PVLDB*, 2015, pp. 642–653.
[205] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving Intel Xeon Phi," in *Proc. 5th ACM/SPEC Int. Conf. Perform. Eng.*, Mar. 2014, pp. 137–148.
[206] OpenMP Architecture Review Board. (2021). *The OpenMP API Specification for Parallel Programming*. [Online]. Available: https://www.openmp.org/specifications/
[207] H. G. Dietz and B. D. Young, "MIMD interpretation on a GPU," in *Languages and Compilers for Parallel Computing* (Lecture Notes in Computer Science), vol. 5898, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds. Berlin, Germany: Springer, 2010.

[208] G.-Y. Lueh, K. Chen, G. Chen, J. Fuentes, W.-Y. Chen, F. Fu, H. Jiang, H. Li, and D. Rhee, "C-For-Metal: High performance simd programming on Intel GPUs," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2021, pp. 289–300.

[209] X. Tian, H. Saito, M. Girkar, S. V. Preis, S. S. Kozhukhov, A. G. Cherkasov, C. Nelson, N. Panchenko, and R. Geva, "Compiling C/C++ SIMD extensions for function and loop vectorizaion on multicore-SIMD processors," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, May 2012, pp. 2349–2358.

[210] MPI Forum, "MPI: A message passing interface," in *Proc. Supercomputing*, Dec. 1993, pp. 878–883.

[211] W.-M. Hwu, D. Kiirk, S. Ryoo, C. Rodriigues, J. Stratton, and K. Huang, "Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX," in *Proc. IEEE Hot Chips 19th Symp. (HCS)*, Aug. 2007, pp. 1–11.

[212] D. Gerzhoy, X. Sun, M. Zuzak, and D. Yeung, "Nested MIMD-SIMD parallelization for heterogeneous microprocessors," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 1–27, Dec. 2019.

[213] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[214] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS*, 2014, pp. 269–284.

[215] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan. 2016, pp. 127–138.

[216] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proc. ASPLOS*, 2017, pp. 751–764.

[217] A. Yazdanbakhsh, K. Samadi, N. S. Kim, H. Esmaeilzadeh, H. Falahati, and P. J. Wolfe, "GANAX: A unified MIMD-SIMD acceleration for generative adversarial networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 650–661.

[218] B. Domonkos and G. Jakab, "A programming model for GPU-based parallel computing with scalability and abstraction," in *Proc. 25th Spring Conf. Comput. Graph.*, Apr. 2009, pp. 103–111.

[219] J. Cabezas, M. Jordà, I. Gelado, N. Navarro, and W.-M. Hwu, "GPU-SM: Shared memory multi-GPU programming," in *Proc. 8th Workshop Gen. Purpose Process. GPUs*, Feb. 2015, pp. 13–24.

[220] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian, "Software transactional memory for GPU architectures," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, Feb. 2014, pp. 1–14.

[221] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-M. Hwu, "PyTorch-Direct: Enabling GPU centric data access for very large graph neural network training with irregular accesses," 2021, *arXiv:2101.07956*.

[222] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: Asynchronous multi-GPU programming model with applications to large-scale graph processing," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, pp. 1–27, Sep. 2020.

[223] J. Choi, D. F. Richards, and L. V. Kale, "Improving scalability with GPU-aware asynchronous tasks," 2022, *arXiv:2202.11819*.

[224] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, Jun. 2010, pp. 235–246.

[225] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 591–602.

[226] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "SIMD re-convergence at thread frontiers," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 477–488.

[227] W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 25–36.

[228] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 308–317.

[229] M. Rhu and M. Erez, "CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures," in *Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2012, pp. 61–71.

[230] A. ElTantawy and T. M. Aamodt, "Warp scheduling for fine-grained synchronization," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 375–388.

[231] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic, "Convergence and scalarization for data-parallel architectures," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2013, pp. 1–11.

[232] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011.

[233] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, Mar. 2004, pp. 52–63.

[234] Y. Wang, S. Chen, J. Wan, J. Meng, K. Zhang, W. Liu, and X. Ning, "A multiple SIMD, multiple data (MSMD) architecture: Parallel execution of dynamic and static SIMD fragments," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 603–614.

[235] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for GPU architectures," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 296–307.

[236] A. Yilmazer and D. Kaeli, "HQL: A scalable synchronization mechanism for GPUs," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 475–486.

[237] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on GPUs," in *Proc. 29th ACM Int. Conf. Supercomputing*, New York, NY, USA, Jun. 2015, pp. 109–118.

[238] D. Thuerck, "Supporting irregularity in throughput-oriented computing by SIMT-SIMD integration," in *Proc. IEEE/ACM 10th Workshop Irregular Appl., Architectures Algorithms (IA3)*, Nov. 2020, pp. 31–35.

[239] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring SIMD execution using heterogeneous hardware and dynamic configurability," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 84–95.

[240] N. B. Abu-Ghazaleh and P. A. Wilsey, "Shared control—Supporting control parallelism using a SIMD-like architecture," in *Proc. Eur. Conf. Parallel Process.* Berlin, Germany: Springer, 1998, pp. 1089–1099.

[241] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian, "Lock-based synchronization for GPU architectures," in *Proc. ACM Int. Conf. Comput. Frontiers*, May 2016, pp. 205–213.

[242] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "SIMD parallelization of applications that traverse irregular data structures," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2013, pp. 1–10.

[243] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A verifier for GPU kernels," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, Oct. 2012, pp. 113–132.

[244] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic verification and test generation for GPUs," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 215–224.

[245] R. Sharma, M. Bauer, and A. Aiken, "Verification of producer-consumer synchronization in GPU programs," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2015, pp. 88–98.

[246] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: A low-overhead mechanism for detecting data races in GPU programs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 135–146, 2011.

[247] A. Habermaier and A. Knapp, "On the correctness of the SIMT execution model of GPUs," in *Programming Languages and Systems*. Berlin, Germany: Springer, 2012, pp. 316–335.

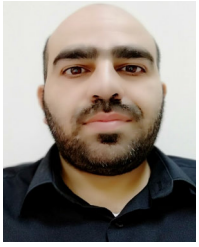
[248] A. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the Intel architecture," *Int. J. Parallel Program.*, vol. 30, pp. 65–98, Apr. 2002.

[249] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *Int. J. Parallel Program.*, vol. 28, no. 4, pp. 363–400, Aug. 2000.

[250] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. New York, NY, USA: ACM Press, 1990.

[251] C. G. Lee and M. G. Stoodley, "Simple vector microprocessors for multimedia applications," in *Proc. 31st Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1998, pp. 25–36.
[252] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks, "Vectorizing for a SIMdD DSP architecture," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, Oct. 2003, pp. 2–11.
[253] K. Hou, H. Wang, and W.-C. Feng, "ASPaS: A framework for automatic SIMDization of parallel sorting on x86-based many-core processors," in *Proc. 29th ACM Int. Conf. Supercomputing*, Jun. 2015, pp. 383–392.
[254] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. Supinski, "OpenMP for accelerators," in *Proc. IWOMP*, 2011, pp. 108–121.
[255] P. Flynn, X. Yi, and Y. Yan, "Exploring source-to-source compiler transformation of OpenMP SIMD constructs for Intel AVX and arm SVE vector architectures," in *Proc. 13th Int. Workshop Program. Models Appl. Multicores Manycores*, Apr. 2022, pp. 11–20.
[256] X. Huo, B. Ren, and G. Agrawal, "A programming system for xeon phis with runtime SIMD parallelization," in *Proc. 28th ACM Int. Conf. Supercomputing*, Jun. 2014, pp. 283–292.
[257] L. Wan, W. Zheng, and X. Yuan, "HCE: A runtime system for efficiently supporting heterogeneous cooperative execution," *IEEE Access*, vol. 9, pp. 147264–147279, 2021.
[258] W. Shu and M.-Y. Wu, "Solving dynamic and irregular problems on SIMD architectures with runtime support," in *Proc. Int. Conf. Parallel Process.*, vol. 2, Aug. 1993, pp. 167–174.
[259] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 222–234, Feb. 1993.
[260] K. Yuksel and W. Skarbek, "Deep alignment network: From MIMD to SIMD platform," *Proc. SPIE*, vol. 10808, pp. 67–74, Oct. 2018.
[261] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. van Essen, "Aluminum: An asynchronous, GPU-aware communication library optimized for large-scale training of deep neural networks on HPC systems," in *Proc. IEEE/ACM Mach. Learn. HPC Environments (MLHPC)*, Nov. 2018, pp. 1–13.
[262] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proc. 6th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, Nov. 1994, pp. 252–262.
[263] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: ACM Press, May 1999, pp. 229–241.
[264] H. Han and C. W. Tseng, "A comparison of locality transformations for irregular codes," in *Proc. 5th Int. Workshop Lang., Compil., Run-Time Syst. Scalable Comput.* London, U.K.: Springer-Verlag, 2000, pp. 70–84.
[265] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," in *Proc. Int. Conf. Comput. Sci. (ICCS)*. London, U.K.: Springer-Verlag, May 2001, pp. 127–136.
[266] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, Oct. 1999, pp. 192–202.
[267] G. Zhu, P. Jiang, and G. Agrawal, "A methodology for characterizing sparse datasets and its application to SIMD performance prediction," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2019, pp. 445–456.
[268] A. Bustamam, K. Burrage, and N. A. Hamilton, "Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 9, no. 3, pp. 679–692, May 2012.
[269] Y. Chen, W. Li, R. Fan, and X. Liu, "GPU optimization for high-quality kinetic fluid simulation," *IEEE Trans. Vis. Comput. Graphics*, vol. 28, no. 9, pp. 3235–3251, Sep. 2022.
[270] Y. Kim, F. Pacaud, K. Kim, and M. Anitescu, "Leveraging GPU batching for scalable nonlinear programming through massive Lagrangian decomposition," 2021, *arXiv:2106.14995*.
[271] J. Austin, R. Corrales-Fatou, S. Wyetzner, and H. Lipson, "Titan: A parallel asynchronous library for multi-agent and soft-body robotics using NVIDIA CUDA," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2020, pp. 7754–7760.
[272] W. Zhang, Z. Yan, Y. Lin, C. Zhao, and L. Peng, "A high throughput B+tree for SIMD architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 707–720, Mar. 2020.
[273] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, "CHARMM: A program for macromolecular energy, minimization, and dynamics calculations," *J. Comput. Chem.*, vol. 4, no. 2, pp. 187–217, Jun. 1983.
[274] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA, Washington, DC, USA, Tech. Rep. NAS-99-011.
[275] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, "Design and implementation of a parallel unstructured Euler solver using software primitives," *AIAA J.*, vol. 32, no. 3, pp. 489–496, Mar. 1994.
[276] Y.-S. Hwang, R. Das, J. H. Saltz, M. Hodoscek, and B. R. Brooks, "Parallelizing molecular dynamics programs for distributed-memory machines," *IEEE Comput. Sci. Eng.*, vol. 2, no. 2, pp. 18–29, May 1995.
[277] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs," in *Proc. 6th Workshop Gen. Purpose Processor Using Graph. Process. Units*, Mar. 2013, pp. 96–107.
[278] J. Shen and J. A. McCammon, "Molecular dynamics simulation of superoxide interacting with superoxide dismutase," *Chem. Phys.*, vol. 158, nos. 2–3, pp. 191–198, Dec. 1991.
[279] W. Shu, "Chare kernel and its implementation on multicomputers," Dept. Comput. Sci., Univ. Illinois Urbana-Champaign, Champaign, IL, USA, ProQuest Dissertations Publishing, 9021756, 1990.
[280] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 474–483.
[281] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2014, pp. 945–955.
[282] L. Thébault, E. Petit, and Q. Dinh, "Scalable and efficient implementation of 3D unstructured meshes computation: A case study on matrix assembly," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, Jan. 2015, pp. 120–129.
[283] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Dec. 2010, pp. 1–10.
[284] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: A GPU implementation of a general sparse linear solver," *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 24, no. 3, pp. 205–223, Jun. 2009.
[285] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–11.
[286] C. R. Ferreira and M. Bader, "Load balancing and patch-based parallel adaptive mesh refinement for tsunami simulation on heterogeneous platforms using Xeon Phi coprocessors," in *Proc. Platform Adv. Sci. Comput. Conf.*, Jun. 2017, p. 12.
[287] C. Rosales, "Porting to the Intel Xeon Phi: Opportunities and challenges," in *Proc. Extreme Scaling Workshop (XSW)*, Aug. 2013, pp. 1–7.
[288] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of SIMD extensions," *Proc. IEEE*, vol. 93, no. 2, pp. 409–425, Feb. 2005.
[289] A. Barredo, J. M. Cebrian, M. Valero, M. Casas, and M. Moreto, "Efficiency analysis of modern vector architectures: Vector ALU sizes, core counts and clock frequencies," *J. Supercomput.*, vol. 76, no. 3, pp. 1960–1979, Mar. 2020.
[290] I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi, "Analysis of relationship between SIMD-processing features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA vector processors," in *Proc. Int. Conf. Parallel Comput. Technol.* Cham, Switzerland: Springer, 2019, pp. 125–139.
[291] J. Langguth, Q. Lan, N. Gaur, and X. Cai, "Accelerating detailed tissue-scale 3D cardiac simulations using heterogeneous CPU-Xeon Phi computing," *Int. J. Parallel Program.*, vol. 45, no. 5, pp. 1236–1258, Oct. 2017.
[292] B. Plazolles, D. E. Baz, M. Spel, V. Rivola, and P. Gegout, "Parallel monte-carlo simulations on GPU and Xeon Phi for stratospheric balloon envelope drift descent analysis," in *Proc. Intl. IEEE Conf. Ubiquitous Intell. Comput., Adv. Trusted Comput., Scalable Comput. Commun., Cloud Big Data Comput., Internet People, Smart World Congr. (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, Jul. 2016, pp. 611–619.

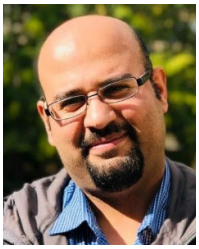
[293] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2012, pp. 141–151.
[294] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proc. 11th ACM SIGPLAN Symp. Princ. Pract. parallel Program.*, New York, NY, USA, Mar. 2006, pp. 119–128.
[295] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," in *Proc. IEEE Comput. Soc. Tech. Committee Comput. Archit. (TCCA) Newslett.*, Dec. 1995, pp. 19–25.
[296] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.
[297] M. Daga, A. M. Aji, and W.-C. Feng, "On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing," in *Proc. Symp. Appl. Accel. High-Perform. Comput.*, Jul. 2011, pp. 141–149.
[298] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures," in *Proc. 9th Conf. Comput. Frontiers*, May 2012, pp. 103–112.
[299] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, Mar. 2010, pp. 63–74.
[300] J. Dongarra and P. Luszczek, "Introduction to the HPC challenge benchmark suite," Dept. Comput. Sci., Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. UT-CS-05-544, 2005.
[301] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2011, pp. 134–144.
[302] M. Khalilov and A. Timoveev, "Performance analysis of CUDA, OpenACC and OpenMP programming models on Tesla V100 GPU," *J. Phys., Conf. Ser.*, vol. 1740, no. 1, Jan. 2021, Art. no. 012056.



DHEYA MUSTAFA (Member, IEEE) received the bachelor's degree in computer engineering from Jordan University of Science and Technology, in 2004, the master's degree in computer engineering from the University of Kentucky, in 2009, and the Ph.D. degree in electrical and computer engineering from Purdue University, in 2013. He is currently an Associate Professor with the Department of Computer Engineering, The Hashemite University, Jordan. Previously, he was with Intel Corporation, Austin, TX, USA, from 2012 to 2016, as a Component Design Engineer, focusing on pre- and post-silicon system validation. His research interests include performance tuning and evaluation, parallel computing, high-performance computing, heterogeneous architectures and systems, artificial intelligence, natural language processing, and STEM education technology.

RUBA ALKHASAWNEH (Member, IEEE) received the B.S. degree in computer engineering from Jordan University of Science and Technology, the M.S. degree in computer engineering from Yarmouk University, and the Ph.D. degree in engineering from Virginia Commonwealth University (VCU), in 2011. She was an Adjunct Instructor with multiple online institutions, from 2014 to 2020. She is currently an Assistant Professor with Al-Ahliyya Amman University, Amman, Jordan. Previously, she was with Intel Corporation, Austin, TX, USA, from 2012 to 2020, as a System Validation Engineer, focusing on pre/post-silicon validation by developing test plans and content to validate leading-edge computer chips. Her research interests include semiconductor design and validation, STEM education, AI, automation, and embedded systems.

FADI OBEIDAT received the B.S. degree in computer engineering from Jordan University of Science and Technology (JUST), the M.S. degree in computer engineering from Prince Faisal Information Technology Center, Yarmouk University, and the Ph.D. degree in computer engineering from Virginia Commonwealth University (VCU). He is currently an Application Engineering Manager and an Emulation Consultant with Synopsys Inc. Previously, he was a Prototyping/an Emulation Architect with Cadence Design Systems (2020–2022), an Emulation Consultant with Synopsys (2014–2020), and an Emulation Architect with intel Corporation (2010–2014). He has an extensive experience in deploying FPGAs and emulation technologies to accelerate system-on-chip (SoC) design and verification. He had conducted research in the following areas embedded systems, unmanned arial vehicles (UAVs), performance modeling, emulation, and engineering education.



AHMED S. SHATNAWI received the M.S. degree in software engineering from George Mason University, in 2012, and the Ph.D. degree in engineering from the University of Wisconsin Milwaukee, in 2017. He is currently an Associate Professor with the Software Engineering Department, Jordan University of Science and Technology. His research interests include intersection of software engineering, information security, and finding better ways to design safe, secure, and reliable software systems.