CCF Transactions on High Performance Computing manuscript No.
(will be inserted by the editor)

Parallel Programming Models for Heterogeneous Many-Cores : A Comprehensive Survey

Jianbin Fang · Chun Huang ✉ · Tao Tang · Zheng Wang

Received: date / Accepted: date

Abstract Heterogeneous many-cores are now an integral part of modern computing systems ranging from embedding systems to supercomputers. While heterogeneous many-core design offers the potential for energy-efficient high-performance, such potential can only be unlocked if the application programs are suitably parallel and can be made to match the underlying heterogeneous platform. In this article, we provide a comprehensive survey for parallel programming models for heterogeneous many-core architectures and review the compiling techniques of improving programmability and portability. We examine various software optimization techniques for minimizing the communicating overhead between heterogeneous computing devices. We provide a road map for a wide variety of different research areas. We conclude with a discussion on open issues in the area and potential research directions. This article provides both an accessible introduction to the fast-moving area of heterogeneous programming and a detailed bibliography of its main achievements.

Keywords Heterogeneous Computing · Many-Core Architectures · Parallel Programming Models

1 Introduction

Heterogeneous many-core systems are now commonplace [159, 160]. The combination of using a host CPU together with specialized processing units (e.g., GPGPUs, XeonPhis, FPGAs, DSPs and NPUs) has been shown in many cases to achieve orders of magnitude performance improvement. As a recent example, Google's Tensor Processing Units (TPUs) are application-specific integrated circuits (ASICs) to accelerate machine learning workloads [162]. Typically, the host CPU of a heterogeneous platform manages the execution context while the computation is offloaded to the accelerator or coprocessor. Effectively leveraging such platforms not only enables the achievement of high performance, but increases energy efficiency. These goals are largely achieved using simple, yet customized hardware cores that use area more efficiently with less power dissipation [69].

The increasing importance of heterogeneous many-core architectures can be seen from the TOP500 and Green500 list, where a large number of supercomputers are using both CPUs and accelerators [23, 44]. A closer look at the list of the TOP500 supercomputers shows that seven out of the top ten supercomputers are built upon heterogeneous many-core architectures (Table 1). On the other hand, this form of many-core architectures is being taken as building blocks for the next-generation supercomputers. e.g., three US national projects (Aurora [36], Frontier [38], and El Capitan [37]) will all implement a heterogeneous CPU-GPU architecture to deliver its exascale supercomputing systems.

The performance of heterogeneous many-core processors offer a great deal of promise for future computing systems, yet their architecture and programming model significantly differ from the conventional multi-core processors [59]. This change has shifted the burden onto programmers and compilers [132]. In particular, programmers have to deal with heterogeneity, massive processing cores, and a complex memory hierarchy.

J. Fang, C. Huang (✉), T. Tang
Institute for Computer Systems, College of Computer, National University of Defense Technology
E-mail: {j.fang, chunhuang, taotang84}@nudt.edu.cn

Z. Wang
School of Computing, University of Leeds
E-mail: z.wang5@leeds.ac.uk

Table 1 The on-node parallel programming models for the top10 supercomputers (as of November 2019).

Rank	Name	Compute node architecture	Heterogeneous?	Programming models
#1	Summit	IBM POWER9 22C CPU (x2)+ NVIDIA Volta GV100 (x6)	YES	CUDA/OpenMP
#2	Sierra	IBM POWER9 22C CPU (x2)+ NVIDIA Volta GV100 (x4)	YES	CUDA/OpenMP
#3	TaihuLight	Sunway SW26010 260C	YES	Athread/OpenACC
#4	Tianhe-2A	Intel Xeon E5-2692v2 12C CPU (x2)+ Matrix-2000 (x2)	YES	OpenCL/OpenMP
#5	Frontera	Xeon Platinum 8280 28C CPU	NO	OpenMP
#6	Piz Daint	Xeon E5-2690v3 12C CPU (x1)+ NVIDIA Tesla P100 (x1)	YES	CUDA
#7	Trinity	Intel Xeon E5-2698v3 16C CPU & Intel Xeon Phi 7250 68C	NO	OpenMP
#8	ABCI	Intel Xeon Gold 6148 20C CPU (x2)+ NVIDIA Tesla V100 SXM2 (x4)	YES	CUDA
#9	SuperMUC-NG	Intel Xeon Platinum 8174 24C CPU	NO	OpenMP
#10	Lassen	IBM POWER9 22C CPU (x2)+ NVIDIA Tesla V100 (x4)	YES	CUDA/OpenMP

Thus, programming heterogeneous many-core architectures are extremely challenging.

How to program parallel machines has been a subject of research for at least four decades [122]. The main contextual difference between now and the late 80s/early 90s is that heterogeneous parallel processing will be shortly a mainstream activity affecting standard programmers rather than a high-end elite endeavour performed by expert programmers. This changes the focus from one where raw performance was paramount to one where programmer productivity is critical. In such an environment, software development tools and programming models that can reduce programmer effort will be of considerable importance.

In this work, we aim to demystify heterogeneous computing and show heterogeneous parallel programming is a trustworthy and exciting direction for systems research. We start by reviewing the historical development and the state-of-the-art of parallel programming models for heterogeneous many-cores by examining solutions targeted at both low-level and high-level programming (Section 2). We then discuss code generation techniques employed by programming models for improving programmability and/or portability (Section 3), before turning our attention to software techniques for optimizing the communication overhead among heterogeneous computing devices (Section 4). Finally, we outline the potential research directions of heterogeneous parallel programming models (Section 5).

2 Overview of Parallel Programming Models

Parallel programming model acts as the bridge between programmers and parallel architectures. To use the shared memory parallelism on multi-core CPUs, parallel programming models are often implemented on threading mechanisms such as the POSIX threads [47]. When it comes to heterogeneous many-cores, we have to deal with the heterogeneity between host and accelerators. And parallel programming models have to introduce relevant abstractions of controlling them both, which is the focus of this survey work.

Figure 1 summarizes the family of parallel programming models for heterogeneous many-core architectures. Based on the performance-programmability tradeoff, we categorize them into *low-level programming models* (Section 2.1) and *high-level programming models* (Section 2.2). The expected application *performance* increases from high-level programming models to low-level programming models, whereas the *programmability* decreases.

The low-level programming models are closest to the many-core architectures, and expose the most hardware details to programmers through data structures and/or APIs. These models are typically bound to specific hardware architectures, and are also known as *native programming models*. In contrast, the high-level programming models raise the languages' abstraction level, and hide more architecture details than the low-level models. Thus, the high-level models often enable better programmability.

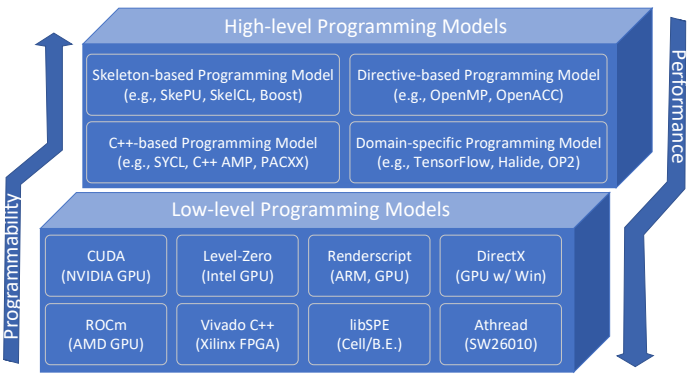

Fig. 1 The family of parallel programming models for heterogeneous many-core architectures.

2.1 Low-level Parallel Programming Models

2.1.1 Prelude of GPGPU Programming

The GPU Shading Lanuages At early 2000s, commodity graphics hardware was rapidly evolving from a fixed function pipeline into a programmable vertex and fragment processor. Originally, these programmable GPUs could only be programmed using assembly language. Later, Microsoft and NVIDIA introduced their C-like programming languages, HLSL and Cg respectively, that compile to GPU assembly language [14,144]. The shading languages make it easier to develop programs incrementally and interactively. This is achieved by using a high-level shading language, e.g., Cg, based on both the syntax and the philosophy of C.

Although these shading languages can hide details of the graphics pipeline (e.g., the number of stages or the number of passes), they are specialized for real-time shading and remain very graphics-centric [64]. In particular, these high-level shading languages do not provide a clean abstraction for general-purpose computing on graphics hardware. Programs operate on vertices and fragments separated by a rasterization stage; memory is divided up into textures and framebuffers; the interface between the graphics hardware and host is through a complex graphics API. This often prevents applying the graphics hardware onto new applications.

Brook for GPGPUs The graphics processors feature instruction sets general enough to perform computation beyond the rendering domain. Applications such as linear algebra operators [131], numerical simulation [111], and machine learning algorithms [176] have been ported to GPUs and achieved a remarkable speedup over traditional CPUs. These research works demonstrate the potential of graphics hardware for more general-purpose computing tasks, i.e., GPGPUs.

The first work to explore this idea is the `Brook` programming system [64]. By introducing the concepts of *streams*, *kernels* and *reduction operators*, Brook abstracts the GPU as a streaming processor. This abstraction is achieved by virtualizing various GPU hardware features with a compiler and runtime system.

The Brook language is an extension to the standard ANSI C and is designed to incorporate the idea of data parallel computing and arithmetic intensity. A Brook program consists of legal C code plus syntactic extensions to denote streams and kernels. The Brook programming system consists of `BRCC` and `BRT`. `BRCC` is a source-to-source compiler which translates Brook codes (`.br`) into C++ codes (`.cpp`). `BRT` is a runtime software which implements the backend of the Brook primitives for target hardware. *We regard Brook as an origin work for programming GPGPUs, and other parallel programming models for heterogeneous many-cores inherit many features from it.*

2.1.2 Vendor-Specific Programming Models

Vendor-specific programming models are bound to vendors and their manufactured hardware. The typical heterogeneous many-core architectures include Cell/B.E., NVIDIA GPU, AMD GPU, Intel XeonPhi, FPGA, DSP, and so on [63]. Hardware vendors introduces their unique programming interfaces, which are restricted to their own products. This section examines each many-core architecture and its native programming models.

libSPE for IBM Cell Broadband Engine The Cell Broadband Engine Architecture (CEBA) and its first implementation Cell/B.E. is a pioneering work of heterogeneous computing [105, 121]. Cell/B.E. [163] was designed by a collaboration effort between Sony, Toshiba, and IBM (STI), and takes a radical departure from conventional multi-core architectures. Instead of integrating identical commodity hardware cores, it uses a conventional high performance PowerPC core (PPE) which controls eight simple SIMD cores (i.e., Synergistic Processing Elements, SPEs). Each SPE contains a synergistic processing unit (SPU), a local store, and a memory flow controller (MFC). Prior works have demonstrated that a wide variety of algorithms on the Cell/B.E. processor can achieve performance that is equal to or significantly better than a general-purpose processor [52, 69, 173, 199]. Its architecture variant, i.e., `PowerXCell 8i`, has been used to build the first petascale supercomputer, `Roadrunner` [55, 128, 130].

Programming the CEBA processor is challenging [74]. IBM developed IBM SDK for Multicore Acceleration with a suite of software tools and libraries [50]. The SDK provides various levels of abstractions, ranging from the low-level management library to the high-level programming models, to tap the Cell's potential. Managing the code running on the SPEs of a CEBA-based system can be done via the `libspe` library (SPE runtime management library) that is part of the SDK package [50]. This library provides a standardized low-level programming interface that manages the SPE threads, and enables communication and data transfer between PPE threads and SPEs. Besides, the SDK contains high-level programming frameworks to assist the development of parallel applications on this architecture.

CUDA for NVIDIA GPUs NVIDIA implemented the unified shader model, where all shader units of graphics hardware are capable of handling any type of shading tasks, in the Tesla and its subsequent designs [13]. Since G80, NVIDIA's GPU architecture has evolved from Tesla [141, 200], Fermi [2], Kepler [7], Maxwell [6],

Pascal [8], Volta [10], to Turing [15]. Each generation of NVIDIA's microarchitecture introduces new features based on its previous one, e.g., the Volta architecture features tensor cores that have superior deep learning performance over regular CUDA cores.

NVIDIA introduces CUDA to program its computing architecture for general-purpose computation [29]. The CUDA programming model works with programming languages such as C, C++, and Fortran. A CUDA program calls parallel kernels, with each executing in parallel across a set of threads. The programmer organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, and per-thread private memory. This accessibility makes it easier for us to use GPU resources.

With CUDA, NVIDIA GPUs have been used to speed up both regular applications [99, 182] and irregular ones [117, 150], with an impressive performance increase over multi-core CPUs. Nevertheless, Lee *et al.* argue that the performance gap can be narrowed by applying optimizations for both CPUs and GPUs [135]. Table 1 shows that five of the top ten supercomputers use NVIDIA GPUs as the accelerators. The GPU-enabled architectures will continue to play a key role in building future high-performance computing systems.

CAL/ROCm for AMD GPUs AMD/ATI was the first to implement the unified shader model in its TeraScale design, leveraging flexible shader processors which can be scheduled to process a variety of shader types [13]. The TeraScale is based upon a very long instruction word (VLIW) architecture, in which the core executes operations in parallel. The Graphics Core Next (GCN) architecture moved to a RISC SIMD microarchitecture, and introduced asynchronous computing [12]. This design makes the compiler simpler and leads to a better utilization of hardware resources. The RDNA (Radeon DNA) architecture is optimized for efficiency and programmability, while offering backwards compatibility with the GCN architecture [16]. As the counterpart to the gaming-focused RDNA, CDNA is AMD's compute-focused architecture for HPC and ML workloads.

Close-To-the-Metal (CTM) is a low-level programming framework for AMD's GPUs. This framework enables programmers to access AMD GPUs with a high-level abstraction, Brook+ [1], which is an extension to the Brook GPU specification on AMD's compute abstraction layer (CAL) [3]. Then AMD renamed the framework as AMD APP SDK (Accelerated Parallel Programming) built upon AMD's CAL, with an OpenCL programming interface. In November 2015, AMD released its "Boltzmann Initiative" and the ROCm open computing platform [34]. ROCm has a modular design which lets any hardware-vendor drivers support the ROCm stack [34]. It also integrates multiple programming languages, e.g., OpenCL and HIP, and provides tools for porting CUDA codes into a vendor-neutral format [25]. At the low level, ROCm is backed by a HSA-compliant language-independent runtime, which resides on the kernel driver [35].

On AMD GPUs, a kernel is a single sequence of instructions that operates on a large number of data parallel work-items. The work-items are organized into architecturally visible work-groups that can communicate through an explicit local data share (LDS). The shader compiler further divides work-groups into microarchitectural wavefronts that are scheduled and executed in parallel on a given hardware implementation. Both AMD and NVIDIA use the same idea to hide the data-loading latency and achieve high throughput, i.e., grouping multiple threads. AMD calls such a group a *wavefront*, while NVIDIA calls it a *warp*.

MPSS/COI for Intel XeonPhis Intel XeonPhi is a series of x86 manycore processors, which inherit many design elements from the Larrabee project [174]. It uses around 60 cores and 30 MB of on-chip caches, and features a novel 512-bit vector processing unit within a core [92]. This architecture has been used to build the Tianhe-2 supercomputer, which was ranked the world's fastest supercomputer in June 2013 [42].

The main difference between an XeonPhi and a GPU is that XeonPhi can, with minor modifications, run software that was originally targeted to a standard x86 CPU. Its architecture allows the use of standard programming languages and APIs such as OpenMP. To access the PCIe-based add-on cards, Intel has developed the Manycore Platform Software Stack (MPSS) and the Coprocessor Offload Infrastructure (COI) [26].

Intel COI is a software library designed to ease the development of software and applications that run on Intel XeonPhi powered device. The COI model exposes a pipelined programming model, which allows workloads to be run and data to be moved asynchronously. Developers can configure one or more pipelines to interact between sources and sinks. COI is a C-language API that interacts with workloads through standard APIs. It can be used with any other programming models, e.g., POSIX threads.

Level-Zero for Intel XPUs Intel GPUs have been getting more powerful and are strong contenders in the graphics and GPGPU space [66]. Apart from the integrated graphics, Intel has revealed its discrete GPU architecture (Xe) targeted for datacenter and HPC ap-

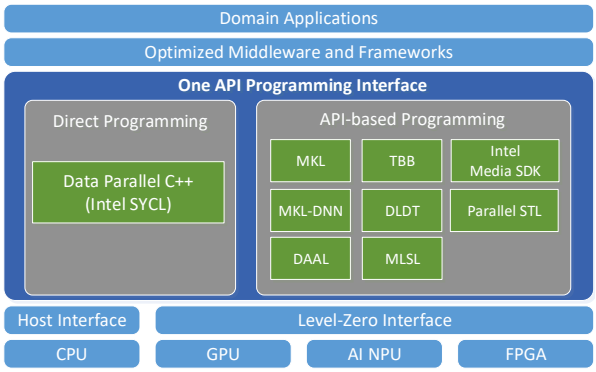

Fig. 2 Intel's OneAPI software stack. Reproduced from [27].

plications, e.g., being as the primary compute engine for the Aurora supercomputer [36]. Figure 2 shows Intel's `OneAPI` software stack to unify programming across its compute product portfolio (CPUs, GPUs, NPUs and FPGAs) with a single set of APIs [27]. At the low level, the `Level-Zero` API is to provide direct-to-metal interfaces to offload accelerator devices. This interface is intended for providing explicit controls needed by higher-level runtime APIs and libraries, e.g., `OneAPI`. Its design is initially influenced by other low-level APIs.

DirectCompute for GPUs Microsoft's DirectCompute is an application programming interface that supports running compute kernels on GPGPUs on Windows. DirectCompute is part of the Microsoft DirectX collection of APIs.

DirectCompute exposes GPU's compute functionality as a new type of shader - the compute shader, which is very similar to the existing vertex, pixel and geometry shaders, but with much more general purpose processing capabilities [4]. The compute shader is not attached specifically to any stage of the graphics pipeline, but interacts with the other stages via graphics resources such as render targets, buffers and textures. Unlike a vertex shader, or a pixel shader, the compute shader has no fixed mapping between the data it is processing and threads that are doing the processing. The compute shader also allows unordered memory access, in particular the ability to perform writes to any location in a buffer. The DirectCompute architecture shares a range of computational interfaces with NVIDIA's CUDA.

Renderscript for GPUs Many mobile phones have followed the same trend of desktop architectures, integrating different types of processing units onto the same chip. These new mobile phones include multi-core CPUs (e.g., Qualcomm Snapdragon) as well as GPUs (e.g., ARM Mali or Qualcomm Adreno).

Google released Renderscript as an official GPU computing framework for Android in 2011 [33]. Renderscript provides high performance and portability across mobile hardware architectures. It provides three primary tools: a simple 3D rendering API, a compute API similar to CUDA, and a C99-derived language.

2.1.3 A Unified Programming Model

The use of heterogeneous many-core architectures in high-performance computing has attracted increasingly more interests, particularly due to the growth of graphics processing units. Much of this growth has been driven by NVIDIA's CUDA ecosystem for developing GPGPU applications on NVIDIA GPUs. However, with the increasing diversity of GPUs, including those from AMD, ARM, Intel and Qualcomm, OpenCL (Open Computing Language) has emerged as an open and vendor-agnostic standard for programming GPUs as well as other accelerating devices such as APUs and FPGAs.

OpenCL is an open programming standard that is maintained by the Khronos group. Its API consists of a C library supporting device programming in the C99 or C++ language. An OpenCL application is composed of two parts: one or more kernels and an OpenCL host program. The kernel specifies functions to be executed in a parallel fashion on the processing cores. The host sets up the execution environment, reads in data, and instantiates and enqueues the kernels to run.

Code Portability vs. Performance Portability OpenCL stands out in its portability by defining an abstract execution model and a platform model. Porting OpenCL to a new many-core device is a matter of providing an implementation of the runtime library that conforms to the standard, achieving the goal of code portability [90]. OpenCL applications written for one vendor's platform should run correctly on other vendors' platforms, if they are not using any vendor-proprietary or platform-specific extensions. The code portability of OpenCL is ensured by Khronos' certification program, which requires OpenCL vendors to pass rigorous conformance tests on their platform before they claim it is OpenCL "conformant" [11, 40].

Different from code portability, OpenCL cannot guarantee the goal of performance portability. This is because the hardware implementation of OpenCL is vendor dependent. Different hardware vendors have their unique device architectures. As a result, an OpenCL application written and optimized for one vendor's platform is unlikely to have the same performance as on other vendors' platforms [60]. To achieve portable performance, researchers have investigated various techniques, which are discussed in Section 3.

Closed and Open-source Implementation Table 2 shows that there exist a variety of OpenCL implemen-

tations. AMD is working on the "Boltzmann Initiative" and the ROCm open compute platform, which contains its OpenCL implementation [18]. Furthermore, the Gallium Compute Project maintains an implementation of OpenCL mainly for AMD Radeon GCN (formerly known as CLOVER [21]), and it builds on the work of the Mesa project to support multiple platforms. Recently, Intel has turned to implementing OpenCL for its CPUs, GPUs and FPGAs, and made its partial implementation open to the public [22]. BEIGNET was an open-source implementation of OpenCL released by Intel in 2013 for its GPUs (Ivy Bridge and newer), but is now deprecated [19]. IBM once released its OpenCL implementation for programming CEBA [62].

In recent years, the mobile system-on-chips (SOCs) have advanced significantly in computing power. GPUs in the mobile SOCs are very powerful. To leverage the computing power, mobile vendors enable OpenCL onto their hardware. ZiiLABS enabled OpenCL on the ZiiLABS platforms and released the ZiiLABS OpenCL SDK [45]. This implementation aims to unlock the full potential of the StemCell (SIMD) array architecture to deliver new levels of performance. ARM has also implemented OpenCL for Mali GPUs [100]. Qualcomm provides the Qualcomm Adreno SDK to take full advantage of the graphics and computation power provided by the Adreno GPU [11]. To facilitate seamless migration of applications between TI SoCs, TI has customized OpenCL implementation for its SOCs (ARM CPUs+TI DSP) [43].

There are several open-source implementations developed and maintained by the academia. `POCL` is an implementation built on Clang and LLVM. It supports CPUs, TTA, NVIDIA GPUs, and the HSA-based architectures [120]. Based on POCL, the researchers from National University of Defense Technology have built an OpenCL implementation (`MOCL`) for the Matrix-2000 many-core architecture [207] and the FT-1500A multi-core CPU [94]. With the help of the generic C++ compilers, `FreeOCL` can supports a large range of multi-core CPUs [20]. `SnuCL` is an open-source OpenCL framework developed at Seoul National University. This framework stands out that it extends the original OpenCL semantics to the heterogeneous cluster environment [123].

One OpenCL to Rule them All? It has been around ten years since the birth of the OpenCL standard in 2009 [30,146]. We had expected that, OpenCL became the unified de facto standard programming model for heterogeneous many-core processors, like OpenMP for multi-core CPUs [41] and MPI for large-scale clusters [28]. However, this is not the case. The fact is that, CUDA has been the de facto programming environment on GPGPUs for years. Table 1 shows that five of the top ten supercomputers use GPGPU architectures and take CUDA as their programming methodology.

We believe that there are several factors behind OpenCL's tepid popularity. The first factor is due to the diversity of many-core architectures in terms of processing cores and memory hierarchies. For example, using scratch-pad memory has been very popular in DSPs, game consoles (IBM Cell/B.E.), and graphic processor, while caches are typically used in Intel XeonPhi. To be compatible with the diverse many-cores, the next version of the OpenCL standard will let more OpenCL features become optional for enhanced deployment flexibility. The optionality includes both API and language features e.g., floating point precisions. By doing so, we can enable vendors to ship functionality precisely targeting their customers and markets [184]. The second factor is due to the commercial interests. The vendors would prefer using a specialized programming model for their hardware and building a complete software ecosystem. From the perspective of programmers, we often choose to use the CUDA programming interface for NVIDIA GPU. This is because NVIDIA optimize its CUDA software stack, and thus applications in CUDA can often enable us to yield a better performance [91].

To summarize, *OpenCL is eligible, yet not practical to be a unified programming model for heterogeneous many-core architectures*. For future, we argue that, neither OpenCL nor CUDA dominates the programming range for heterogeneous many-core architectures, but it is likely that they coexist. Our investigation work shows that the low-level programming models share a common working paradigm, and vendors would choose to support their unique one. Thereafter, they should develop a complete ecosystem (e.g., domain libraries) around this programming model.

2.2 High-level Parallel Programming Models

To lower programming barrier, various strategies have been proposed to improve the abstraction level, i.e., high-level programming models. This is achieved by redefining programmers' role and letting compilers and runtime systems take more work. Ultimately, the high-level parallel programming models aim to free programmers from mastering the low-level APIs or performing architecture-specific optimizations. There are five types of high-level programming models: (1) the C++-based programming models, (2) the skeleton-based programming models, (3) the STL-based programming models, (4) the directive-based programming models, and (5) the domain-specific programming models.

Table 2 The OpenCL Implementations: Open- and closed-source

	Developer	SDK	Hardware	Operating System	Version	Open-Source
AMD OpenCL	AMD	ROCm	AMD CPU/GPU/APU	Linux/Windows	2.0	Y
NVIDIA OpenCL	NVIDIA	CUDA	NVIDIA GPU	Linux/Windows	1.2	N
Intel OpenCL	Intel	Intel SDK	Intel CPU/GPU/MIC/FPGA	Linux/Windows	2.1	Y
IBM OpenCL	IBM	IBM SDK	IBM CPU/CEBA	Linux	1.1	N
ARM OpenCL	ARM	ARM	ARM CPU/Mali GPU	Linux	1.2, 2.0	N
Qualcomm OpenCL	Qualcomm	Adreno GPU SDK	Qualcomm Adreno GPU	Andriod	2.0	N
TI OpenCL	TI	Processor SDK	TI C66x DSP	Linux	1.1	Y
ZiiLABS OpenCL	ZiiLABS	ZIILABS SDK	ZMS StemCell processors	Andriod	N/A	N
POCL	Tampere U.	POCL	CPU/ASIP/NVIDIA GPU/HSA GPUs	Linux/Windows	1.2, 2.0	Y
Clover OpenCL	Denis Steck	Mesa	AMD GPU	Linux	1.1	Y
FreeOCL	zuzuf	FreeOCL	CPU	Linux/Windows	1.2	Y
MOCL	NUDT	MOCL	Matrix-2000	Linux	1.2	Y
SnuCL	SNU	SNUCL	CPU/GPU/Cluster	Linux	1.2	Y

2.2.1 C++-based Programming Models

SYCL is a cross-platform abstraction layer that builds on OpenCL's concepts, portability and efficiency for programming heterogeneous platforms in a single-source style with standard C++. This programming model enables the host and kernel code for an application to be contained in the same source file, and achieves the simplicity of a cross-platform asynchronous task graph [17]. Intel has been developing oneAPI that includes DPC++ (an implementation of SYCL with extensions) for its CPUs, GPUs and FPGAs [27]. C++ AMP (C++ Accelerated Massive Parallelism) is a heterogeneous programming model based on C++. Built upon DirectX11, this model uses *parallel_for_each* to instantiate device code, and provides users with a C++ programming interface. Thus, the programmer can write GPU code in a more concise and controlled way [102]. Inspired by C++ AMP and C++14, HC (Heterogeneous Computing) C++ API is a GPU-oriented C++ compiler developed by AMD [24]. HC removes the "restrict" keyword, supports additional data types in kernels, and provides fine-grained control over synchronization and data movement.

`PACXX` is a unified programming model for programming heterogeneous many-core systems, to mitigate the issue of long, poorly structured and error-prone codes in low-level programming models [106]. In PACXX, both host and device programs are written in the C++14 standard, with all modern C++ features including type inference (auto), variadic templates, generic lambda expressions, as well as STL containers and algorithms. PACXX consists of a custom compiler (based on the Clang front-end and LLVM IR) and a runtime system, which facilitate automatic memory management and data synchronization [107, 108].

Other C++-based parallel programming models include boost.compute [180], HPL [188,189], VexCL [82], hlslib for FPGAs [95], alpaka [205], and so on. By respecting the philosophy of modern C++, these high-level programming models integrate the host code and the device code into a single C++ file, and use a unified data structure to manage the buffer to avoid manual management of data synchronization. By doing so, programmers can transparently use the parallel computing resources of many-core architectures, without having to master their details. An important feature of the C++-based parallel programming model is that, it can form a natural match with other C++ programs, which facilitates the development and composition of large-scale scientific computing applications, significantly improving programmability and productivity.

2.2.2 Skeleton-based Programming Models

Skeleton programming is a programming paradigm based on algorithmic skeleton [72]. A skeleton is a predefined "high-order function", such as *map*, *reduce*, *scan*, *farm*, and *pipeline*, which implements a common pattern of computation and/or data dependence. Skeleton programming provides a high level of abstraction and portability, with a quasi-sequential programming interface, as their implementations encapsulate all the underlying details and architecture features, including parallelization, synchronization, communication, buffer management, accelerator usage and many others.

SkePU is a skeleton programming framework based on the C++ template for multi-core CPU and multi-GPU systems. This framework contains six data-parallel and one task-parallel skeletons, and two generic container types. The backend support of SkePU includes OpenMP, OpenCL and CUDA, where Clang is used to facilitate the source-to-source code transformation [78]. The interface of SkePU2 is improved based on C++11 and variadic templates, and user-defined functions are expressed with the lambda expression [89].

SkelCL is another skeleton library for heterogeneous platforms [178]. It provides programmers with vector data type, high-level data distribution mechanism to enable the automatic buffer management, implicit data

transfers between host and accelerator. This aims to significantly simplify the programming of heterogeneous many-core systems. The SkelCL backend translates each skeleton into an OpenCL kernel, and enables the SkelCL codes to run on both multi-core CPUs and heterogeneous GPUs. This framework also supports automatic mapping of tasks onto multiple devices [177].

Other skeleton-based programming models include Muesli [71], Marrow [145], ParallelME [49,65], etc. Compared with the C++-based programming models, the skeleton programming model is lack of generality, i.e., some computing or communication patterns are difficult to be represented by the builtin skeletons. Thus we have to extend the existing skeletons when necessary. Programmers are also responsible for synthesizing their target applications with builtin skeletons.

2.2.3 STL-based Programming Models

TBB (Threading Building Blocks) is a C++ template library developed by Intel for parallel programming of its multi-core CPUs. It implements a set of algorithm components (e.g., *parallel_for*, *parallel_reduce*) and a set of containers (e.g., *concurrent_queue*, *concurrent_vector*). The TBB runtime system is responsible for managing and scheduling threads to execute tasks, and balancing the computing loads between multi-cores by task stealing. By doing so, TBB aims to unbind programmers and the underlying hardware [126]. HPX (High Performance ParallelX) is a generic C++ runtime system for parallel and distributed systems [114, 115]. By providing a unified programming interface, HPX can transparently use the underlying hardware resources with an asynchronous multi-tasking mechanism. HPX aims to be easy-to-use, and achieves high scalability and portability. HPX enables the support of heterogeneous computing, by introducing the concepts of *target*, *allocator* and *executor* within the `hpx.compute` subproject. The backend of the computing platform includes CUDA, HCC and SYCL [73, 116].

Thrust is a C++ standard parallel template library for NVIDIA GPUs. It implements four basic core algorithms for *each*, *reduce*, *scan* and *sort* [58]. By doing so, users need not know how to map the calculation to the computing resources (e.g., thread block size, buffer management, algorithm variant selection, etc.), but only pay attention to the calculation itself. This approach can greatly improve productivity. Thrust's backend is based on CUDA, TBB, and OpenMP. Kokkos allows programmers to write modern C++ applications in a hardware-agnostic manner [87]. It is a programming model for parallel algorithms that use many-core chips and share memory among those cores. This programming model includes computation abstractions for frequently used parallel computing patterns, policies that provide details for how those computing patterns are applied, and execution spaces that denote on which cores the parallel computation is performed.

Other similar STL-based programming models include Microsoft's PPL (Parallel Patterns Library) [9], RAJA [57], etc. This kind of programming model implements the functions and their extensions in the standard C++ template library, and provides concurrent data structures. Thus, this programming model can unbind the parallel programming itself with the underlying hardware resources. Programmers do not need to care about the details of the underlying architectures, which effectively lowers the programming barrier.

2.2.4 Directive-based Programming Models

Another high-level programming models are based on directive annotations, including both industry standardards (OpenMP [41], OpenACC [39], Intel LEO [26]) and academia-maintained efforts (OmpSs [86], XcalableMP [155], Mint [186], OpenMDSP [113]). These programming models only have to add directive constructs before the target code region, while the tasks of offloading, parallelization and optimization are delegated to compilers and runtime systems. On one hand, programmers do not have to master a large number of architectural details, thus leading to an improved productivity. On the other hand, programmers can annotate their codes incrementally, which also lowers the barrier of debugging. Therefore, this programming model enables the non-experts to enjoy the performance benefits of heterogeneous many-cores without being entangled in the architecture details.

Multiple directive-based programming models can be mapped onto a single many-core architecture. As we have mentioned in Section 2.1.2, programming the Cell/B.E. processor is challenging. There is a significant amount of research in programming models that attempts to make it easy to exploit the computation power of the CEBA architecture. Bellens *et al.* present Cell superscalar (CellSs) which addresses the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the CEBA architecture [59]. Based on annotating the source code, a source-to-source compiler generates the necessary code and a runtime library exploits the existing parallelism by building a task dependency graph. The runtime takes care of task scheduling and data movements between the different processors of this architecture. O'Brien *et al.* explore supporting OpenMP on the Cell processor [157] based on IBM's XL compiler,

so that programmers can continue using their familiar programming model and existing code can be re-used.

2.2.5 Domain-Specific Programming Models

To achieve an even better performance and programmability, the domain-specific programming models are preferred on heterogeneous many-cores.

Mudalige *et al.* propose a high-level programming framework, `OPS`, for multi-block structural grid applications [5]. The frontend of `OPS` leverages a common set of APIs for users, and the backend generates highly optimized device code for target platforms. The `OP2` framework is built upon `OPS`. The difference is that `OPS` is suitable for dealing with multi-block structured grids, while `OP2` is targeted for unstructured grid applications [97]. `AMGCL` is a header-only C++ library for solving large sparse linear systems with algebraic multigrid (AMG) method [81]. This library has a minimal dependency, and provides both shared-memory and distributed memory versions of the algorithms. It allows for transparent acceleration of the solution phase with OpenCL, CUDA, or OpenMP [83].

Dubach *et al.* propose a new Java compatible object-oriented programming language (`Lime`) [85]. It uses high-level abstractions to explicitly represent parallelism and computation. The backend compiler and runtime can automatically manage the data mapping and generate OpenCL/CUDA code for GPUs. `Halide` is a new language for generating efficient image processing code on heterogeneous platforms and simplifying programming [164,165]. Its frontend is embedded in C++, while its backend includes x86, ARMv7, CUDA and OpenCL. Equally used in image processing, `KernelGenius` is a high-level programming tool for EMM (explicitly managed memory many cores) [136]. `Membarth` has implemented a source-to-source compiler, which translates a high-level description into low-level GPU codes (OpenCL, CUDA or renderscript) [147, 148]. Sidelnik *et al.* have proposed to implement a high-level programming language, `Chapel`, for controlling task allocation, communication and data locality structure on multi-core CPUs and GPUs. A program in `Chapel` can run on multiple platforms, and achieve the same performance as CUDA programs [175]. Hong *et al.* describe `Green-Marl`, a domain-specific language, whose high-level language constructs allow developers to describe their graph analysis algorithms intuitively, but expose the data-level parallelism inherent in the algorithms [118].

The deep learning frameworks (e.g., `Tensorflow` [46], `PyTorch` [161], and `MXNet` [70]) provide users with script or functional languages (e.g., *Python*, *R*, *Scala*, *Julia*) in the frontend. These script languages are used to describe the workflow of training or inference. At the backend, the frameworks dispatch tasks to the underlying heterogeneous systems (GPUs or FPGAs) via low-level or other high-level programming models such as OpenCL, CUDA or SYCL. This whole process of mapping tasks is transparent to users.

To sum up, the domain-specific programming models have the potential to improve programmer productivity, to support domain-specific forms of modularity, and to use domain-specific information to support optimizations [144]. Most of these advantages are obtained by raising the language's abstraction level with domain-specic data types, operators, and control constructs. Although the domain-specific programming models can generate efficient kernel code, they are limited to specific application domains.

3 Compiling Techniques for Improved Programmability and Portability

Translating code in one programming model to another enables code reuse and reduce the learning curve of a new computing language. Ultimately, code translation can improve programmability, portability, and performance. In this section, we review the code translation techniques between parallel programming models on heterogeneous many-core architectures.

3.1 C-to-CUDA

CUDA provides a multi-threaded parallel programming model, facilitating high performance implementations of general-purpose computations on GPUs. However, manual development of high-performance CUDA code still requires a large amount of effort from programmers. Programmers have to explicitly manage the memory hierarchy and multi-level parallel view. Hence the automatic transformation of sequential input programs into parallel CUDA programs is of significant interest.

Baskaran *et al.* describe an automatic code transformation system that generates parallel CUDA code from input sequential C code, for affine programs [56]. Using publicly available tools that have made polyhedral compiler optimization practically effective, the authors develop a C-to-CUDA transformation system that generates two-level parallel CUDA code that is optimized for efficient data access. The performance of the automatically generated CUDA code is close to hand-optimized versions and considerably better than their performance on multi-core CPUs. Building on Baskaran's experience, Reservoir Labs developed its own compiler based on R-Stream [137], which introduces a more advanced algorithm to exploit the memory hierarchy.

PPCG Verdoolaege *et al.* address the compilation of a sequential program for parallel execution on a modern GPU [187]. They present a source-to-source compiler (`PPCG`), which singles out for its ability to accelerate computations from any static control loop nest, generating multiple CUDA kernels when necessary. The authors introduce a multilevel tiling strategy and a code generation scheme for the parallelization and locality optimization of imperfectly nested loops, managing memory and exposing concurrency according to the constraints of modern GPUs.

Bones Nugteren *et al.* evaluate a number of C-to-CUDA transformation tools targeting GPUs, and identify their strengths and weaknesses [156]. Then they address the weaknesses by presenting a new classification of algorithms. This classification is used in a source-to-source compiler (`Bones`) based on the algorithmic skeletons technique. The compiler generates target code based on skeletons of parallel structures, which can be seen as parameterisable library implementations for a set of algorithm classes. This compiler still requires some modifications to the original sequential source code, but can generate readable code for further fine-tuning.

PIPS Non-polyhedral tools have also seen major developments. PIPS is an open-source initiative developed by the HPC Project to unify efforts concerning compilers for parallel architectures [32, 48]. It supports the automatic integrated compilation of applications for heterogeneous architectures including GPUs. The compiler uses abstract interpretation for array regions based on polyhedra, which allows PIPS to perform powerful interprocedural analysis on the input code.

DawnCC Mendonca *et al.* argue that inserting pragmas into production code is a difficult and error-prone task, often requiring familiarity with the target program [149,152]. This difficulty restricts developers from annotating code that they have not written themselves. Therefore, they provide a suite of compiler-based methods and a tool, `DawnCC`, to mitigate the issue. The tool relies on symbolic range analysis to achieve two purposes: populate source code with data transfer primitives and to disambiguate pointers that could hinder automatic parallelization due to aliasing.

3.2 CUDA-to-OpenCL

Restricted to NVIDIA GPUs, CUDA has the largest code base and high-performance libraries. On the other hand, OpenCL is an open standard supported on a large number of mainstream devices. With the great interest in OpenCL comes a challenge: manufacturers have a large investment in CUDA codes and yet would like to take advantage of wider deployment opportunities afforded by OpenCL. Therefore, an automated tool for CUDA-to-OpenCL translation is required.

SnuCL-Tr Kim *et al.* present similarities and differences between CUDA and OpenCL, and develop an automatic translation framework between them [124]. `SnuCL-Tr` can achieve comparable performance between the original and target applications in both directions. Given that each programming model has a large user-base and code-base, this translator is useful to extend the code-base for each programming model and unifies the efforts to develop applications.

CU2CL Gardner *et al.* summarize the challenges of translating CUDA code to its OpenCL equivalence [96]. They develop an automated CUDA-to-OpenCL source-to-source translator (`CU2CL`), to automatically translate three medium-to-large, CUDA-optimized codes to OpenCL, thus enabling the codes to run on other GPU-accelerated systems [146,172]. `Swan` is tool used to ease the transition between OpenCL and CUDA [112]. Different from CU2CL, Swan provides a higher-level library that abstracts both CUDA and OpenCL, such that an application makes calls to Swan and Swan takes care of mapping the calls to CUDA or OpenCL.

NMT Kim *et al.* present source-to-source translation between CUDA to OpenCL using neural machine translation (`NMT`). To generate a training dataset, they extract CUDA API usages from CUDA examples and write corresponding OpenCL API usages. With a pair of API usages acquired, they construct API usage trees that help users find unseen usages from new samples and easily add them to a training input [127].

O2render With a similar goal, Yang *et al.* introduces `O2render`, an OpenCL-to-Renderscript translator that enables the porting of an OpenCL application to a Renderscript application [202]. O2render automatically translates OpenCL kernels to a Renderscript kernel.

3.3 Directive-to-CUDA/OpenCL

Translating OpenMP-like codes into CUDA/OpenCL codes will not only reuse the large OpenMP code base, but also lower their programming barrier.

OpenMP-to-CUDA Lee and Eigenmann present a framework for automatic source-to-source translation of standard OpenMP applications into CUDA applications [133]. This translator aims to further improve programmability and make existing OpenMP applications amenable to execution on GPGPUs. Later, they propose a new programming interface, `OpenMPC`, which builds on OpenMP to provide an abstraction of CUDA

and offers high-level controls of the involved parameters and optimizations [134].

OpenMP-to-OpenCL Kim *et al.* propose a framework that translates OpenMP 4.0 accelerator directives to OpenCL [125]. They leverage a run-time optimization to automatically eliminates unnecessary data transfers between the host and the accelerator.

OpenMP-to-LEO Managing data transfers between the CPU and XeonPhi and optimizing applications for performance requires some amount of effort and experimentation. Ravi *et al.* present `Apricot`, an optimizing compiler and productivity tool for Intel XeonPhi that minimizes developer effort by automatically inserting LEO clauses [166]. This optimizing compiler aims to assist programmers in porting existing multi-core applications and writing new ones to take full advantage of the many-core accelerator, while maximizing overall performance.

CUDA-lite CUDA programmers shoulder the responsibility of managing the code to produce the desirable access patterns. Experiences show that such responsibility presents a major burden on the programmer, and this task can be better performed by automated tools. Ueng *et al.* present `CUDA-lite`, an enhancement to CUDA, as one such tool [185]. This tool leverages programmers' knowledge via annotations to perform transformations and show preliminary results that indicate auto-generated code can have performance comparable to hand-crafted codes.

hiCUDA Han *et al.* have designed `hiCUDA` [109,110], a high-level directive-based language for CUDA programming. They develop a prototype compiler to facilitate the translation of a hiCUDA program to a CUDA program. The compiler is able to support real-world applications that span multiple procedures and use dynamically allocated arrays.

CUDA-CHiLL The CHiLL developers extended their compiler to generate GPU code with `CUDA-CHiLL` [170], which does not perform an automatic parallelization and mapping to CUDA but instead offers high-level constructs that allow a user or search engine to perform such a transformation.

3.4 Adapting CUDA/OpenCL to Multi-core CPUs

MCUDA is a source-to-source translator that translates CUDA to multi-threaded CPU code [179]. This translator is built on Cetus [53], a source-to-source translator framework for C and other C-based languages.

CUDA-x86 by PGI allows developers using CUDA to compile and optimize their CUDA applications to run on x86-based multi-core architectures [31]. `CUDA-x86` includes full support for NVIDIA's CUDA C/C++ language for GPUs. When running on x86-based systems without a GPU, CUDA applications can use multiple cores and the streaming SIMD capabilities of Intel and AMD CPUs for parallel execution.

Ocelot [84] is primarily a PTX-to-LLVM translator and run-time system that can decide whether to run the PTX on a GPU device or on a CPU with just-in-time (JIT) compilation. `Ocelot` is similar to `MCUDA` in that it allows for CUDA kernels to be run on CPUs, but it takes the approach of performing translations on lower-level bytecodes.

3.5 Supporting Multiple Devices

The interest in using multiple accelerating devices to speed up applications has increased in recent years. However, the existing heterogeneous programming models (e.g., OpenCL) abstract details of devices at the per-device level and require programmers to explicitly schedule their kernel tasks on a system equipped with multiple devices. This subsection examines the software techniques of extending parallel programming models to support multiple devices.

GPUSs The GPU Superscalar (`GPUSs`) is an extension of the Star Superscalar (`StarSs`) programming model that targets application parallelization on platforms with multiple GPUs [51]. This framework deals with architecture heterogeneity and separate memory spaces, while preserving simplicity and portability.

VirtCL You *et al.* propose a framework (`VirtCL`) that reduces the programming burden by acting as a layer between the programmer and the native OpenCL runtime system. `VirtCL` abstracts multiple devices into a single virtual device [203]. This framework comprises two main software components: a front-end library, which exposes primary OpenCL APIs and the virtual device, and a back-end run-time system (CLDaemon) for scheduling and dispatching kernel tasks based on a history-based scheduler.

OpenMP extension Yan *et al.* explore support of multiple accelerators in high-level programming models by extending OpenMP to support offloading data and computation regions to multiple accelerators [201]. These extensions allow for distributing data and computation among a list of devices via easy-to-use interfaces, including specifying the distribution of multi-dimensional arrays and declaring shared data regions.

OpenACC-to-CUDA Komoda *et al.* present an OpenACC compiler to run single OpenACC programs on multiple GPUs [129]. By orchestrating the compiler and the runtime system, the proposed system can efficiently manage the necessary data movements among multiple GPUs memories. The authors extend a set of directives based on the standard OpenACC API to facilitate communication optimizations. The directives allow programmers to express the patterns of memory accesses in the parallel loops to be offloaded. Inserting a few directives into an OpenACC program can reduce a large amount of unnecessary data movements.

4 Optimization Techniques for Minimizing the Host-Accelerator Communication

While the heterogeneous many-core design offers the potential for energy-efficient, high-performance computing, software developers are finding it increasingly hard to deal with the complexity of these systems [91, 151]. In particular, programmers need to effectively manage the host-device communication, because the communication overhead can completely eclipse the benefit of computation offloading if not careful [61, 67, 92, 101]. Gregg and Hazelwood have shown that, when memory transfer times are included, it can take 2x–50x longer to run a kernel than the GPU processing time alone [101].

Various parallel programming models have introduced the streaming mechanism to amortize the host-device communication cost [154]. It works by partitioning the processor cores to allow independent communication and computation tasks (i.e., streams) to run concurrently on different hardware resources, which effectively overlaps the kernel execution with data movements. Representative heterogeneous streaming implementations include CUDA Streams [29], OpenCL Command Queues [30], and Intel's hStreams [93, 119, 138, 139, 154, 206]. These implementations allow the program to spawn more than one stream/pipeline so that the data movement stage of one pipeline overlaps the kernel execution stage of another.

4.1 The Streaming Mechanism

The idea of heterogeneous streams is to exploit temporal and spatial sharing of the computing resources.

Temporal Sharing. Code written for heterogeneous computing devices typically consists of several stages such as host-device communication and computation. Using temporal sharing, one can overlap some of these stages to exploit pipeline parallelism to improve performance. This paradigm is illustrated in Figure 3. In this example, we can exploit temporal sharing to overlap the host-device communication and computation stages to achieve better runtime when compared to execute every stage sequentially. One way of exploiting temporal sharing is to divide an application into independent tasks so that they can run in a pipeline fashion.

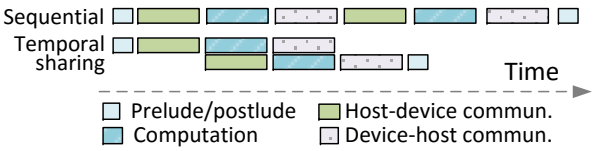

Fig. 3 Exploit pipeline parallelism by temporal sharing. Reproduced from [206].

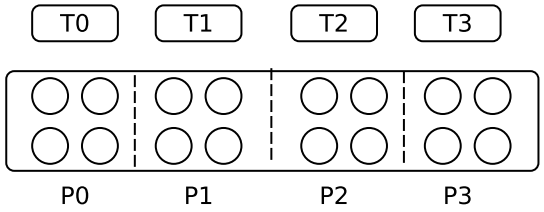

Fig. 4 Spatial sharing. The circles represent processing cores, `Tx` represents a task, and `Px` represents a partition.

Note that the PPE and SPEs of Cell/B.E. share the same memory space, and thus there is no need of host-accelerator communication [69]. But each SPE has a channel/DMA transport for controlling input and output. To prevent memory stalls, we can take advantage of the DMA engines by adopting a double-buffer approach to overlap computations on the previously fetched datablocks with transfers of subsequent data blocks to and from memory. We regard that this is a special case of *temporal sharing*.

Spatial Sharing. Using multiple streams also enjoys the idea of resource partitioning. That is, to partition the resource (e.g., processing cores) into multiple groups and map each stream onto a partition. Therefore, different streams can run on different partitions concurrently, i.e., resource *spatial sharing*. Nowadays accelerators have a large number of processing units that some applications cannot efficiently exploit them for a given task. Typically, we offload a task and let it occupy all the processing cores. When using multiple streams, we divide the processing cores into multiple groups (each group is named as a *partition*). Figure 4 shows that a device has 16 cores and is logically divided into four partitions (`P0`, `P1`, `P2`, `P3`). Then different tasks are offloaded onto different partitions, e.g., `T0`, `T1`, `T2`, `T3` runs on `P0`, `P1`, `P2`, `P3`, respectively. In this way, we aim to improve the device utilization.

```
1  //setting the partition-size and task
       ↪ granularity
   hStreams_app_init(partition_size,
       ↪ streams_p_part);
3
   //stream queue id
5  stream_id = 0;
   for(...){
7    //enquque host-device transfer to current
         ↪ stream
     hStreams_app_xfer_memory(,,, stream_id,
         ↪ HSTR_SRC_TO_SINK,...);
9    ...
     //enqueue computation to the current stream
11   hStreams_EnqueueCompute(stream_id, "kernel1"
         ↪ , ...);
     ...
13   //move to the next stream
     stream_id = (stream_id++) % MAX_STR;
15 }
   //transfer data back to host
17 hStreams_app_xfer_memory(,,, HSTR_SINK_TO_SRC
       ↪ ,...);
```

Fig. 5 Example `hStreams` code.

4.2 Intel hStreams

Intel `hStreams` is an open-source streaming implementation built on the COI programming model [154]. At its core is the resource partitioning mechanism [119]. At the physical level, the whole device is partitioned into multiple groups and thus each group has several processing cores. At the logical level, a device can be seen as one or more *domains*. Each domain contains multiple *places*, each of which then has multiples *streams*. The logical concepts are visible to programmers, while the physical ones are transparent to them and the mapping between them are automatically handled by the runtime.

`hStreams` is implemented as a library and provides users with APIs to access coprocessors/accelerators efficiently. Programming with `hStreams` resembles that in `CUDA` or `OpenCL`. Programmers have to create the streaming context, move data between host and device, and invoke kernel execution. And they also have to split tasks to use multiple streams. Figure 5 gives a simplified code example written with Intel's `hStreams` APIs. At line 2, we initialize the stream execution by setting the number of partitions and tasks/streams per partition. This initialization process essentially creates multiple processor domains and determines how many logical streams can run on a partition. In the *for* loop (lines 7-14) we enqueue the communication and computation tasks to a number of streams identified by the `stream_id` variable. In this way, communication and computation of different streams can be overlapped during execution (temporal sharing); and streams on different processor domains (or partitions) can run concurrently (spatial sharing).

4.3 Performance Modelling for Streaming Programs

The previous work have demonstrated that choosing a right stream configuration has a great impact on the resultant performance [93, 206]. And the best configuration must be determined on a per-program and per-dataset basis. Attempting to find the optimal configuration through an exhaustive search would be ineffective, and the overhead involved would be far bigger than the potential benefits. Therefore, building models for heterogeneous streaming programs is of great significance.

4.3.1 Hand-Crafted Analytical Models

Gomez-Luna *et al.* present performance models for asynchronous data transfers on GPU architectures [98]. The models permit programmers to estimate the optimal number of streams in which the computation on the GPU should be broken up. Werkhoven *et al.* present an analytical performance model to indicate when to apply which overlapping method on GPUs [198]. The evaluation results show that the performance model is capable of correctly classifying the relative performance of the different implementations. Liu *et al.* carry out a systematic investigation into task partitioning to achieve maximum performance gain for AMD and NVIDIA GPUs [142]. This approach is not ideal, as it is not only complex to develop the analytical models, but is likely to fail due to the variety of programs and the ever-changing hardware architecture. That is, these hand-crafted models have the drawback of being not portable across architectures as the model is tightly coupled to a specific many-core architecture.

4.3.2 Machine-Learning based Models

Researchers have also exploited the machine learning techniques to automatically construct a predictive model to directly predict the best configuration [206,208]. This approach provides minimal runtime, and has little development overhead when targeting a new many-core architecture. This is achieved by employing machine learning techniques to automatically construct a predictive model to decide at runtime the optimal stream configuration for any streamed application. The predictor is first trained *off-line*. Then, using code and dynamic runtime features of the program, the model predicts the best configuration for a *new*, *unseen* program. This approach can avoid the pitfalls of using a hard-wired heuristic that requires human modification every time when the architecture evolves, where the number and the type of cores are likely to change from one generation to the next. Experimental results XeonPhi and GPGPUs have shown that this approach can achieve over 93% of the Oracle performance [208].

5 A Vision for the Next Decade

Given the massive performance potential of heterogeneous many-core hardware design, it is clear that future computing hardware will be increasingly specialized and diverse. As the hardware design keeps evolving, so does the software development systems and the programming model. In this section, we discuss some challenges and open research directions for future parallel programming models.

A holistic solution. China, US, Japan and EUROPE are currently working towards the exascale computing. The design and construction of an exascale machine will be built based on heterogeneous many-core architectures of various forms [140]. This achievement will require significant advances in the software paradigm and require that parallelism in control and data be exploited at all possible levels. Therefore, the dominant design parameter will shift from hardware to system software and in particular, parallel programming systems [79]. We envision that a hierarchy of programming models have to be implemented as well as the equipment of expert optimized libraries. Low-level programming models should be implemented, but are not suggested to be exposed to programmers. Instead, the high-level programming models and highly optimized domain libraries are exposed as the programming interface. Intel's OneAPI is one of such examples [27].

Pattern-aware parallel programming. Parallel programming based on patterns (such as *map/reduce* and *pipeline*) or algorithmic skeletons (Section 2.2.2), where programmers write algorithmic intents that abstract away parallelization, heterogeneity, and reliability concerns, offer a partial solution for programming heterogeneous parallel systems [80]. As can been from Section 2.2, this is an essential feature of high-level parallel programming models. The key to the success of pattern-based parallel programming is to have a fully-supported development toolchain for code optimization, debugging and performance diagnosis. The current compiler implementation is oblivious to the high-level algorithmic intents expressed in the patterns, leading to disappointing performance, discouraging the adoption of pattern-based parallel programming. For example, a sequential loop that adds one to an integer literal one million times will be optimized away at compile time. However, if we implement it as a parallel pipeline pattern, existing compilers, including leading industrial parallel compilers, Intel TBB on ICC and Go, would fail to merge the trivial pipeline elements. As a result, the pattern-based implementation takes minutes to run, not nanoseconds[1], leading to a massive slowdown over the sequential version. The issue arises from the fact that current compilers are oblivious to parallel patterns. A parallel construct encodes the sequential semantics of the program, but this is lost to the compiler. If the compiler knew the sequential semantics, it can then dynamically merge small pipeline elements. If we can do these, the primary barrier to adopting pattern-based programming would be torn down.

Machine learning based program tuning. Programmers are faced with many decisions when writing heterogeneous programs, such as selecting an optimal thread configuration [77] and/or selecting a right code variant [153]. This is due to the profound differences in many-core architectures, programming models and input applications [54]. By default, the runtime system of high-level programming models has to assist users in automating these online decisions. If a wrong configuration is selected, the overall performance will be significantly decreased. Therefore, it is significant to design a model to help programmers to automatically choose a reasonable configuration, i.e., *automated performance tuning*, which is regarded to have the potential to dramatically improve the performance portability of petascale and exascale applications [54]. A key enabling technology for optimizing parallel programs is *machine learning*. Rather than hand-craft a set of optimization heuristics based on compiler expert insight, learning techniques automatically determine how to apply optimizations based on statistical modelling and learning [190,191]. This provides a rigorous methodology to search and extract structure that can be transferred and reused in unseen settings. Its great advantage is that it can adapt to changing platforms as it has no a priori assumption about their behaviour. There are many studies showing it outperforms human-based approaches. Recent work shows that it is effective in performing parallel code optimization [68, 75, 76, 104, 158, 194, 196], performance predicting [193, 209], parallelism mapping [103, 181, 183, 192, 195–197, 208], and task scheduling [88,143,167–169,171,204]. As the many-core design becomes increasingly diverse, we believe that the machine-learning techniques provide a rigorous, automatic way for constructing optimization heuristics, which is more scalable and sustainable, compared to manually-crafted solutions.

6 Conclusions

This article has introduced programming models for heterogeneous many-core architectures. Power, energy

[1] Code is available at: `https://goo.gl/y7bBdN`.

and thermal limits have forced the hardware industry to introduce heterogeneous many-cores built around specialized processors. However, developers are struggling to manage the complexity of heterogeneous many-core hardware. A crisis is looming as these systems become pervasive. As such, how to enable developers to write and optimize programs for the emerging heterogeneous many-core architectures has generated a large amount of academic interest and papers. While it is impossible to provide a definitive cataloger of all research, we have tried to provide a comprehensive and accessible survey of the main research areas and future directions. As we have discussed in the article, this is a trustworthy and exciting direction for systems researchers.

Acknowledgements This work was partially funded by the National Key Research and Development Program of China under Grant No. 2018YFB0204301, the National Natural Science Foundation of China under Grant agreements 61972408, 61602501 and 61872294, and a UK Royal Society International Collaboration Grant.

References

1. Amd brook+ programming. Tech. rep., AMD Corporation (2007)
2. Nvidia's next generation cuda compute architecture: Fermi. Tech. rep., NVIDIA Corporation (2009)
3. Amd cal programming guide v2.0. Tech. rep., AMD Corporation (2010)
4. Directcompute programming guide. Tech. rep., NVIDIA Corporation (2010)
5. High-level abstractions for performance, portability and continuity of scientific software on future computing systems. Tech. rep., University of Oxford (2014)
6. Nvidia geforce gtx 980. Tech. rep., NVIDIA Corporation (2014)
7. Nvidia's next generation cuda compute architecture: Kepler tm gk110/210. Tech. rep., NVIDIA Corporation (2014)
8. Nvidia tesla p100. Tech. rep., NVIDIA Corporation (2016)
9. Parallel Patterns Library. https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=vs-2019 (2016)
10. Nvidia tesla v100 gpu architecture. Tech. rep., NVIDIA Corporation (2017)
11. Qualcomm snapdragon mobile platform opencl general programming and optimization. Tech. rep., Qualcomm Corporation (2017)
12. "vega" instruction set architecture. Tech. rep., AMD Corporation (2017)
13. Common-Shader Core. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-common-core?redirectedfrom=MSDN (2018)
14. HLSL: the High Level Shading Language for DirectX. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl (2018)
15. Nvidia turing gpu architecture. Tech. rep., NVIDIA Corporation (2018)
16. Introducing rdna architecture. Tech. rep., AMD Corporation (2019)
17. Sycl integrates opencl devices with modern c++. Tech. Rep. version 1.2.1 revison 6, The Khronos Group (2019)
18. AMD's OpenCL Implementation. https://github.com/RadeonOpenCompute/ROCm-OpenCL-Runtime (2020)
19. Beignet OpenCL. https://www.freedesktop.org/wiki/Software/Beignet/ (2020)
20. FreeOCL. http://www.zuzuf.net/FreeOCL/ (2020)
21. GalliumCompute. https://dri.freedesktop.org/wiki/GalliumCompute/ (2020)
22. GalliumCompute. https://github.com/intel/compute-runtime (2020)
23. Green500 Supercomputers. https://www.top500.org/green500/ (2020)
24. HCC: Heterogeneous Compute Compiler. https://gpuopen.com/compute-product/hcc-heterogeneous-compute-compiler/ (2020)
25. HIP: Heterogeneous-Compute Interface for Portability. https://github.com/RadeonOpenCompute/hcc (2020)
26. Intel Manycore Platform Software Stack. https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss (2020)
27. Intel's OneAPI. https://software.intel.com/en-us/oneapi (2020)
28. MPI: Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/ (2020)
29. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit (2020)
30. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/ (2020)
31. PGI CUDA C/C++ for x86. https://developer.nvidia.com/pgi-cuda-cc-x86 (2020)
32. PIPS: Automatic Parallelizer and Code Transformation Framework. https://pips4u.org/ (2020)
33. Renderscript Compute. http://developer.android.com/guide/topics/renderscript/compute.html (2020)
34. ROCm: a New Era in Open GPU Computing. https://www.amd.com/en/graphics/servers-solutions-rocm-hpc (2020)
35. ROCm Runtime. https://github.com/RadeonOpenCompute/ROCR-Runtime (2020)
36. The Aurora Supercomputer. https://aurora.alcf.anl.gov/ (2020)
37. The El Capitan Supercomputer. https://www.cray.com/company/customers/lawrence-livermore-national-lab (2020)
38. The Frontier Supercomputer. https://www.olcf.ornl.gov/frontier/ (2020)
39. The OpenACC API specification for parallel programming. https://www.openacc.org/ (2020)
40. The OpenCL Conformance Tests. https://github.com/KhronosGroup/OpenCL-CTS (2020)
41. The OpenMP API specification for parallel programming. https://www.openmp.org/ (2020)
42. The Tianhe-2 Supercomputer. https://top500.org/system/177999 (2020)
43. TI's OpenCL Implementation. https://git.ti.com/cgit/opencl (2020)
44. Top500 Supercomputers. https://www.top500.org/ (2020)
45. ZiiLABS OpenCL. http://www.ziilabs.com/products/software/opencl.php (2020)

46. Abadi, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. CoRR (2016)
47. Alfieri, R.A.: An efficient kernel-based implementation of POSIX threads. In: USENIX Summer 1994 Technical Conference. USENIX Association (1994)
48. Amini, M., et al.: Static compilation analysis for host-accelerator communication optimization. In: Languages and Compilers for Parallel Computing, 24th International Workshop, LCPC (2011)
49. Andrade, G., et al.: Parallelme: A parallel mobile engine to explore heterogeneity in mobile computing architectures. In: Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing (2016)
50. Arevalo, A., et al.: Programming the Cell Broadband Engine: Examples and Best Practices (2007)
51. Ayguadé, E., et al.: An extension of the starss programming model for platforms with multiple gpus. In: Euro-Par 2009 Parallel Processing (2009)
52. Bader, D.A., Agarwal, V.: FFTC: fastest fourier transform for the IBM cell broadband engine. In: High Performance Computing, HiPC (2007)
53. Bae, H., et al.: The cetus source-to-source compiler infrastructure: Overview and evaluation. International Journal of Parallel Programming (2013)
54. Balaprakash, P., et al.: Autotuning in high-performance computing applications. Proceedings of the IEEE (2018)
55. Barker, K.J., et al.: Entering the petaflop era: the architecture and performance of roadrunner. In: Proceedings of the ACM/IEEE Conference on High Performance Computing, SC (2008)
56. Baskaran, M.M., et al.: Automatic c-to-cuda code generation for affine programs. In: R. Gupta (ed.) 19th International Conference on Compiler Construction (CC) (2010)
57. Beckingsale, D., et al.: Performance portable C++ programming with RAJA. In: Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP (2019)
58. Bell, N., Hoberock, J.: Chapter 26 - thrust: A productivity-oriented library for cuda. In: W. mei W. Hwu (ed.) GPU Computing Gems Jade Edition, Applications of GPU Computing Series, pp. 359 – 371. Morgan Kaufmann (2012)
59. Bellens, P., et al.: Cellss: a programming model for the cell BE architecture. In: Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (2006)
60. Bodin, F., Romain, D., Colin De Verdiere, G.: One OpenCL to Rule Them All? In: International Workshop on Multi-/Many-core Computing Systems, MuCoCoS (2013)
61. Boyer, M., et al.: Improving GPU performance prediction with data transfer modeling. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (2013)
62. Breitbart, J., Fohry, C.: Opencl - an effective programming model for data parallel computations at the cell broadband engine. In: 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS (2010)
63. Brodtkorb, A.R., et al.: State-of-the-art in heterogeneous computing. Scientific Programming (2010)
64. Buck, I., et al.: Brook for gpus: stream computing on graphics hardware. ACM Trans. Graph. (2004)
65. de Carvalho Moreira, W., et al.: Exploring heterogeneous mobile architectures with a high-level programming model. In: 29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD (2017)
66. Chandrasekhar, A., et al.: IGC: the open source intel graphics compiler. In: IEEE/ACM International Symposium on Code Generation and Optimization, CGO (2019)
67. Che, S., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization. IEEE Computer Society (2009)
68. Chen, D., et al.: Characterizing scalability of sparse matrix–vector multiplications on phytium ft-2000+. International Journal of Parallel Programming (2020)
69. Chen, T., et al.: Cell broadband engine architecture and its first implementation - A performance view. IBM Journal of Research and Development (2007)
70. Chen, T., et al.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. CoRR **abs/1512.01274** (2015)
71. Ciechanowicz, P., et al.: The münster skeleton library muesli: A comprehensive overview. Working Papers, ERCIS - European Research Center for Information Systems, No. 7 (2009)
72. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation (1989)
73. Copik, M., Kaiser, H.: Using SYCL as an implementation framework for hpx.compute. In: Proceedings of the 5th International Workshop on OpenCL, IWOCL (2017)
74. Crawford, C.H., et al.: Accelerating computing with the cell broadband engine processor. In: Proceedings of the 5th Conference on Computing Frontiers (2008)
75. Cummins, C., et al.: End-to-end deep learning of optimization heuristics. In: PACT (2017)
76. Cummins, C., et al.: Synthesizing benchmarks for predictive modeling. In: CGO (2017)
77. Dao, T.T., Lee, J.: An auto-tuner for opencl work-group size on gpus. IEEE Trans. Parallel Distrib. Syst. (2018)
78. Dastgeer, U., et al.: Adaptive implementation selection in the skepu skeleton programming library. In: Advanced Parallel Processing Technologies - 10th International Symposium, APPT (2013)
79. Davis, N.E., et al.: Paradigmatic shifts for exascale supercomputing. The Journal of Supercomputing (2012)
80. De Sensi, D., et al.: Bringing parallel patterns out of the corner: the p3 arsec benchmark suite. ACM Transactions on Architecture and Code Optimization (TACO) (2017)
81. Demidov, D.: Amgcl: An efficient, flexible, and extensible algebraic multigrid implementation. Lobachevskii Journal of Mathematics (2019)
82. Demidov, D., et al.: ddemidov/vexcl: 1.4.1 (2017). URL `https://doi.org/10.5281/zenodo.571466`
83. Demidov, D., et al.: ddemidov/amgcl: 1.2.0 (2018). URL `https://doi.org/10.5281/zenodo.1244532`
84. Diamos, G.F., et al.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: 19th International Conference on Parallel Architectures and Compilation Techniques, PACT (2010)
85. Dubach, C., et al.: Compiling a high-level language for gpus: (via language support for architectures and compilers). In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2012)

86. Duran, A., et al.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters (2011)
87. Edwards, H.C., et al.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. (2014)
88. Emani, M.K., et al.: Smart, adaptive mapping of parallelism in the presence of external workload. In: CGO (2013)
89. Ernstsson, A., et al.: Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming (2018)
90. Fang, J.: Towards a systematic exploration of the optimization space for many-core processors. Ph.D. thesis, Delft University of Technology, Netherlands (2014)
91. Fang, J., et al.: A comprehensive performance comparison of CUDA and opencl. In: ICPP (2011)
92. Fang, J., et al.: Test-driving intel xeon phi. In: ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 137–148 (2014)
93. Fang, J., et al.: Evaluating multiple streams on heterogeneous platforms. Parallel Process. Lett. **26**(4) (2016)
94. Fang, J., et al.: Implementing and evaluating opencl on an armv8 multi-core CPU. In: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC) (2017)
95. de Fine Licht, J., Hoefler, T.: hlslib: Software engineering for hardware design. CoRR (2019)
96. Gardner, M.K., et al.: Characterizing the challenges and evaluating the efficacy of a cuda-to-opencl translator. Parallel Computing (2013)
97. Giles, M.B., et al.: Performance analysis of the OP2 framework on many-core architectures. SIGMETRICS Performance Evaluation Review (2011)
98. Gómez-Luna, J., et al.: Performance models for asynchronous data transfers on consumer graphics processing units. J. Parallel Distrib. Comput. (2012)
99. Govindaraju, N.K., et al.: High performance discrete fourier transforms on graphics processors. In: Proceedings of the ACM/IEEE Conference on High Performance Computing, SC (2008)
100. Grasso, I., et al.: Energy efficient HPC on embedded socs: Optimization techniques for mali GPU. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS (2014)
101. Gregg, C., et al.: Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS (2011)
102. Gregory, K., Miller, A.: C++ AMP: Accelerated Massive Parallelism With Microsoft Visual C++ (2012)
103. Grewe, D., et al.: Opencl task partitioning in the presence of gpu contention. In: LCPC (2013)
104. Grewe, D., et al.: Portable mapping of data parallel programs to opencl for heterogeneous systems. In: CGO (2013)
105. Gschwind, M., et al.: Synergistic processing in cell's multicore architecture. IEEE Micro (2006)
106. Haidl, M., Gorlatch, S.: PACXX: towards a unified programming model for programming accelerators using C++14. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM (2014)
107. Haidl, M., Gorlatch, S.: High-level programming for many-cores using C++14 and the STL. International Journal of Parallel Programming (2018)
108. Haidl, M., et al.: Pacxxv2 + RV: an llvm-based portable high-performance programming model. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC (2017)
109. Han, T.D., et al.: *hi*cuda: a high-level directive-based language for GPU programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU, ACM International Conference Proceeding Series (2009)
110. Han, T.D., et al.: hicuda: High-level GPGPU programming. IEEE Trans. Parallel Distrib. Syst. (2011)
111. Harris, M.J., et al.: Simulation of cloud dynamics on graphics hardware. In: Proceedings of the 2003 ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (2003)
112. Harvey, M.J., et al.: Swan: A tool for porting CUDA programs to opencl. Computer Physics Communications (2011)
113. He, J., et al.: Openmdsp: Extending openmp to program multi-core DSP. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT (2011)
114. Heler, T., et al.: Hpx - an open source c++ standard library for parallelism and concurrency. In: OpenSuCo (2017)
115. Heller, T., et al.: Using HPX and libgeodecomp for scaling HPC applications on heterogeneous supercomputers. In: Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA (2013)
116. Heller, T., et al.: Closing the performance gap with modern C++. In: High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS (2016)
117. Hong, S., et al.: Accelerating CUDA graph algorithms at maximum warp. In: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP (2011)
118. Hong, S., et al.: Green-marl: a DSL for easy and efficient graph analysis. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (2012)
119. Intel Inc.: hStreams Architecture for MPSS 3.5 (2015)
120. Jääskeläinen, P., et al.: pocl: A performance-portable opencl implementation. International Journal of Parallel Programming (2015)
121. Kahle, J.A., et al.: Introduction to the cell multiprocessor. IBM Journal of Research and Development (2005)
122. Karp, R.M., et al.: The organization of computations for uniform recurrence equations. Journal of the ACM (JACM) (1967)
123. Kim, J., et al.: Snucl: an opencl framework for heterogeneous CPU/GPU clusters. In: International Conference on Supercomputing, ICS (2012)
124. Kim, J., et al.: Bridging opencl and CUDA: a comparative analysis and translation. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC (2015)
125. Kim, J., et al.: Translating openmp device constructs to opencl using unnecessary data transfer elimination. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC (2016)

126. Kim, W., Voss, M.: Multicore desktop programming with intel threading building blocks. IEEE Software (2011)
127. Kim, Y., et al.: Translating CUDA to opencl for hardware generation using neural machine translation. In: IEEE/ACM International Symposium on Code Generation and Optimization, CGO (2019)
128. Kistler, M., et al.: Petascale computing with accelerators. In: D.A. Reed, V. Sarkar (eds.) Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP (2009)
129. Komoda, T., et al.: Integrating multi-gpu execution in an openacc compiler. In: 42nd International Conference on Parallel Processing, ICPP (2013)
130. Komornicki, A., et al.: Roadrunner: Hardware and Software Overview (2009)
131. Krüger, J.H., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graph. (2003)
132. Kudlur, M., et al.: Orchestrating the execution of stream programs on multicore platforms. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI (2008)
133. Lee, S., Eigenmann, R.: Openmp to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP (2009)
134. Lee, S., Eigenmann, R.: Openmpc: Extended openmp programming and tuning for gpus. In: Conference on High Performance Computing Networking, Storage and Analysis, SC (2010)
135. Lee, V.W., et al.: Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: 37th International Symposium on Computer Architecture, ISCA (2010)
136. Lepley, T., et al.: A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory. In: International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES (2013)
137. Leung, A., et al.: A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In: Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU, ACM International Conference Proceeding Series (2010)
138. Li, Z., et al.: Evaluating the performance impact of multiple streams on the mic-based heterogeneous platform. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops (2016)
139. Li, Z., et al.: Streaming applications on heterogeneous platforms. In: Network and Parallel Computing - 13th IFIP WG 10.3 International Conference, NPC (2016)
140. Liao, X., et al.: Moving from exascale to zettascale computing: challenges and techniques. Frontiers of IT & EE (2018)
141. Lindholm, E., et al.: NVIDIA tesla: A unified graphics and computing architecture. IEEE Micro (2008)
142. Liu, B., et al.: Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity. IJHPCA (2016)
143. Marco, V.S., et al.: Improving spark application throughput via memory aware task co-location: A mixture of experts approach. In: Middleware (2017)
144. Mark, W.R., et al.: Cg: a system for programming graphics hardware in a c-like language. ACM Trans. Graph. (2003)
145. Marqués, R., et al.: Algorithmic skeleton framework for the orchestration of GPU computations. In: Euro-Par 2013 Parallel Processing, Lecture Notes in Computer Science (2013)
146. Martinez, G., et al.: CU2CL: A cuda-to-opencl translator for multi- and many-core architectures. In: 17th IEEE International Conference on Parallel and Distributed Systems, ICPADS (2011)
147. Membarth, R., et al.: Generating device-specific GPU code for local operators in medical imaging. In: 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS (2012)
148. Membarth, R., et al.: Hipa^{cc}: A domain-specific language and compiler for image processing. IEEE Trans. Parallel Distrib. Syst. (2016)
149. Mendonca, G.S.D., et al.: Dawncc: Automatic annotation for data parallelism and offloading. TACO (2017)
150. Merrill, D., et al.: Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP (2012)
151. Meswani, M.R., et al.: Modeling and predicting performance of high performance computing applications on hardware accelerators. IJHPCA (2013)
152. Mishra, A., et al.: Kernel fusion/decomposition for automatic gpu-offloading. In: IEEE/ACM International Symposium on Code Generation and Optimization, CGO (2019)
153. Muralidharan, S., et al.: Architecture-adaptive code variant tuning. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (2016)
154. Newburn, C.J., et al.: Heterogeneous streaming. In: IPDPSW (2016)
155. Nomizu, T., et al.: Implementation of xcalablemp device acceleration extention with opencl. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSWP (2012)
156. Nugteren, C., Corporaal, H.: Introducing 'bones': a parallelizing source-to-source compiler based on algorithmic skeletons. In: The 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU (2012)
157. O'Brien, K., et al.: Supporting openmp on cell. International Journal of Parallel Programming (2008)
158. Ogilvie, W.F., et al.: Fast automatic heuristic construction using active learning. In: LCPC (2014)
159. Owens, J.D., et al.: A survey of general-purpose computation on graphics hardware. In: Eurographics, pp. 21–51 (2005)
160. Owens, J.D., et al.: GPU computing. Proceedings of the IEEE (2008)
161. Paszke, A., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS, pp. 8024–8035 (2019)
162. Patterson, D.A.: 50 years of computer architecture: From the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set. In: 2018 IEEE International Solid-State Circuits Conference, ISSCC (2018)

163. Pham, D., et al.: The design methodology and implementation of a first-generation CELL processor: a multi-core soc. In: Proceedings of the IEEE 2005 Custom Integrated Circuits Conference, CICC (2005)
164. Ragan-Kelley, J., et al.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. (2012)
165. Ragan-Kelley, J., et al.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2013)
166. Ravi, N., et al.: Apricot: an optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In: International Conference on Supercomputing, ICS (2012)
167. Ren, J., et al.: Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In: INFOCOM (2017)
168. Ren, J., et al.: Proteus: Network-aware web browsing on heterogeneous mobile systems. In: CoNEXT '18 (2018)
169. Ren, J., et al.: Camel: Smart, adaptive energy optimization for mobile web interactions. In: IEEE Conference on Computer Communications (INFOCOM) (2020)
170. Rudy, G., et al.: A programming language interface to describe transformations and code generation. In: Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC (2010)
171. Sanz Marco, V., et al.: Optimizing deep learning inference on embedded systems through adaptive model selection. ACM Transactions on Embedded Computing (2019)
172. Sathre, P., et al.: On the portability of cpu-accelerated applications via automated source-to-source translation. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia (2019)
173. Scarpazza, D.P., et al.: Efficient breadth-first search on the cell/be processor. IEEE Trans. Parallel Distrib. Syst. (2008)
174. Seiler, L., et al.: Larrabee: A many-core x86 architecture for visual computing. IEEE Micro (2009)
175. Sidelnik, A., et al.: Performance portability with the chapel language. In: 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS, pp. 582–594 (2012)
176. Steinkrau, D., et al.: Using gpus for machine learning algorithms. In: Eighth International Conference on Document Analysis and Recognition (ICDAR. IEEE Computer Society (2005)
177. Steuwer, M., Gorlatch, S.: Skelcl: a high-level extension of opencl for multi-gpu systems. The Journal of Supercomputing (2014)
178. Steuwer, M., et al.: Skelcl - A portable skeleton library for high-level GPU programming. In: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS (2011)
179. Stratton, J.A., et al.: MCUDA: an efficient implementation of CUDA kernels for multi-core cpus. In: Languages and Compilers for Parallel Computing, 21th International Workshop, LCPC (2008)
180. Szuppe, J.: Boost.compute: A parallel computing library for C++ based on opencl. In: Proceedings of the 4th International Workshop on OpenCL, IWOCL (2016)
181. Taylor, B., et al.: Adaptive optimization for opencl programs on embedded heterogeneous systems. In: LCTES (2017)
182. Tomov, S., et al.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing (2010)
183. Tournavitis, G., et al.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. ACM Sigplan Notices (2009)
184. Trevett, N.: Opencl, sycl and spir - the next steps. Tech. rep., OpenCL Working Group (2019)
185. Ueng, S., et al.: Cuda-lite: Reducing GPU programming complexity. In: J.N. Amaral (ed.) Languages and Compilers for Parallel Computing, 21th International Workshop, LCPC (2008)
186. Unat, D., et al.: Mint: realizing CUDA performance in 3d stencil methods with annotated C. In: Proceedings of the 25th International Conference on Supercomputing, (2011)
187. Verdoolaege, S., et al.: Polyhedral parallel code generation for CUDA. ACM TACO (2013)
188. Viñas, M., et al.: Exploiting heterogeneous parallelism with the heterogeneous programming library. J. Parallel Distrib. Comput. (2013)
189. Viñas, M., et al.: Heterogeneous distributed computing based on high-level abstractions. Concurrency and Computation: Practice and Experience (2018)
190. Wang, Z.: Machine learning based mapping of data and streaming parallelism to multi-cores. Ph.D. thesis, University of Edinburgh (2011)
191. Wang, Z., O'Boyle, M.: Machine learning in compiler optimisation. Proc. IEEE (2018)
192. Wang, Z., O'Boyle, M.F.: Partitioning streaming parallelism for multi-cores: a machine learning based approach. In: PACT (2010)
193. Wang, Z., O'boyle, M.F.: Using machine learning to partition streaming programs. ACM TACO (2013)
194. Wang, Z., et al.: Exploitation of gpus for the parallelisation of probably parallel legacy code. In: CC '14 (2014)
195. Wang, Z., et al.: Integrating profile-driven parallelism detection and machine-learning-based mapping. ACM TACO (2014)
196. Wang, Z., et al.: Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. ACM TACO (2015)
197. Wen, Y., et al.: Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In: HiPC (2014)
198. van Werkhoven, B., et al.: Performance models for CPU-GPU data transfers. In: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (2014)
199. Williams, S., et al.: The potential of the cell processor for scientific computing. In: Proceedings of the Third Conference on Computing Frontiers (2006)
200. Wong, H., et al.: Demystifying GPU microarchitecture through microbenchmarking. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS (2010)
201. Yan, Y., et al.: Supporting multiple accelerators in high-level programming models. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP (2015)
202. Yang, C., et al.: O2render: An opencl-to-renderscript translator for porting across various gpus or cpus. In: IEEE 10th Symposium on Embedded Systems for Real-time Multimedia, ESTIMedia (2012)

203. You, Y., et al.: Virtcl: a framework for opencl device abstraction and management. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP (2015)
204. Yuan, L., et al.: Using machine learning to optimize web interactions on heterogeneous mobile systems. IEEE Access (2019)
205. Zenker, E., et al.: Alpaka - an abstraction library for parallel kernel acceleration. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops (2016)
206. Zhang, P., et al.: Auto-tuning streamed applications on intel xeon phi. In: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS (2018)
207. Zhang, P., et al.: MOCL: an efficient opencl implementation for the matrix-2000 architecture. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, CF (2018)
208. Zhang, P., et al.: Optimizing streaming parallelism on heterogeneous many-core architectures. IEEE TPDS (2020)
209. Zhao, J., et al.: Predicting cross-core performance interference on multicore processors with regression analysis. IEEE TPDS (2016)