Evolution of Programming Approaches for High-Performance Heterogeneous Systems

Jacob Lambert
University of Oregon
jlambert@cs.uoregon.edu

Advisor: Allen D. Malony
University of Oregon
malony@cs.uoregon.edu

External Advisor: Seyong Lee
Oak Ridge National Lab
lees2@ornl.gov
Area Exam Report

Committee Members: Allen Malony, Boyana Norris, Hank Childs

Computer Science
University of Oregon
United States
December 14, 2020

Evolution of Programming Approaches for High-Performance Heterogeneous Systems

ABSTRACT

Nearly all contemporary high-performance systems rely on heterogeneous computation. As a result, scientific application developers are increasingly pushed to explore heterogeneous programming approaches. In this project, we discuss the long history of heterogeneous computing and analyze the evolution of heterogeneous programming approaches, from distributed systems to grid computing to accelerator-based supercomputers.

1 INTRODUCTION

Heterogeneous computing is paramount to today's high-performance systems. The top and next generation of supercomputers all employ heterogeneity, and even desktop workstations can be configured to utilize heterogeneous execution. The explosion of activity and interest in heterogeneous computing, as well as the exploration and development of heterogeneous programming approaches, may seem like a recent trend. However, heterogeneous programming has been a topic of research and discussion for nearly four decades. Many of the issues faced by contemporary heterogeneous programming approach designers have long histories, and have many connections with now antiquated projects.

In this project, we explore the evolution and history of heterogeneous computing, with a focus on the development of heterogeneous programming approaches. In Section 2, we do a deep dive into the field of distributed heterogeneous programming, the first application of hardware heterogeneity in computing. In Section 3, we briefly explore the resolutions of distributed heterogeneous systems and approaches, and discuss the transitional period for the field of heterogeneous computing. In Section 4, we provide a broad exploration into contemporary accelerator-based heterogeneous computing, specifically analyzing the different programming approaches developed and employed across different accelerator architectures. Finally, in Section 5, we take a zoomed-out look at the development of heterogeneous programming approaches, introspect on some important takeaways and topics, and speculate about the future of next-generation heterogeneous systems.

2 DISTRIBUTED HETEROGENEOUS SYSTEMS 1980 - 1995

Even 40 years ago, computer scientists realized heterogeneity was needed due to diminishing returns in homogeneous systems. In the literature, the first references to the term "heterogeneous computing" revolved around the distinction between single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD) machines in a distributed computing environment.

Several machines dating back to the 1980s were created and advertised as heterogeneous computers. Although these machines were conceptually different than today's heterogeneous machines, they still were created to address the same challenges: using optimized hardware to execute specific algorithmic patterns.

The Partitionable SIMD/MIMD System (PASM) [270] machine developed at Purdue University in 1981 was initially developed for image processing and pattern recognition application. PASM was unique in that it could be dynamically reconfigured into either a SIMD or MIMD machine, or a combination thereof. The goal was to create a machine that could be optimized for different image processing and pattern recognition tasks, configuring either more SIMD or MIMD capabilities depending on the requirements of the application.

However, like many early heterogeneous computing systems, programmability was not the primary concern. The programming environment for PASM required the design of a new procedure-based structured language similar to TRANQUIL [2], the development of a custom compiler, and even the development of a custom operating system.

Another early heterogeneous system was TRAC, the Texas Reconfigurable Array Computer [264], built in 1980. Like PASM, TRAC could weave between SIMD and MIMD execution modes. But also like PASM, programmability was not a primary or common concern with the TRAC machine, as it relied on now-arcane Job Control Languages and APL source code [197].

The lack of focus on programming approaches for early heterogeneous systems is evident in some ways by the difficulty in finding information on how the machines were typically programmed. However, as the availability of heterogeneous computing environments increased throughout the 1990s, so did the research and development of programming environments.

Throughout the 80s and early 90s, this environment expanded to include vector processors, scalar processors, graphics machines, etc. To this end, in this first major section we explore distributed heterogeneous computing.

Although the first heterogeneous machines consisted of mixed-mode machines like PASM and TRAC, mixed-machine heterogeneous systems became the more popular and accessible option throughout the 1990s. Instead of a single machine with the ability to switch between a synchronous SIMD mode and an asynchronous MIMD mode, mixed-machine systems contained a variety of different processing machines connected by a high-speed interconnect.

Examples of machines used in mixed-machine systems include graphics and rendering-specific machines like the Pixel Planes 5, Silicon Graphics 340 VGX, SIMD and vector machines like the MasPar MP-series and the CM 200/2000, and coarse grained MIMD machines like the CM-5, Vista, and Sequent machines.

It was well understood that different classes of machines (SIMD, MIMD, vector, graphics, sequential) excelled at different tasks (parallel computation, statistical analysis, rendering, display), and that these machines could be networked together in a single system. However, coordinating these distributed systems to execute a single

application presented significant challenges, which many of the projects in the next section began to address.

In this section, we explore different programming frameworks developed to utilize these distributed heterogeneous systems. In Section 2.1, we review several surveys to gain a contextualized insight into the research consensus during the time period. Then in Section 2.2, we review the most prominent and impactful programming systems introduced during this time. Finally in Section 2.3 we discuss the evolution of distributed heterogeneous computing, and how it relates to the subsequent sections.

2.1 Distributed Heterogeneous Architectures, Concepts, and Themes

For insight into high-level perspectives, opinions, and the general state of the area of early distributed heterogeneous computing, we include discussions from several survey works published during the targeted time period. We aim to extract general trends and overarching concepts that drove the development of early systems and early heterogeneous programming approaches.

The work by Ercegovac [106], *Heterogeneity in Supercomputer Architectures*, represents one of the first published works specifically surveying the state of high performance heterogeneous computing. They define heterogeneity as the combination of different architectures and system design styles into one system or machine, and their motivation for heterogeneous systems is summed up well by the following direct quote:

> Heterogeneity in the design (of supercomputers) needs to be considered when a point of diminishing returns in a homogeneous architecture is reached.

As we see throughout this work, this drive for specialization to counter diminishing returns from existing hardware repeatedly resurfaces, and this motivation for heterogeneous systems is very much relevant today.

Ercegovac's work defines four distinct avenues for heterogeneity:

1. *System Level* - The combination of a CPU and an I/O channel processor, or a host and special processor, or a master/slave multiprocessor system.
2. *Operating System Level* - The operating system in a distributed architecture, and how it handles functionality and performance for a diverse set of nodes.
3. *Program Level* - Within a program, tasks need to be defined as concurrent, either by a programmer or compiler, and then those tasks are allocated and executed on different processors.
4. *Instruction Level* - Specialized units, like an arithmetic vector pipelines, are used to provide optimal cost/performance ratios. These units execute specialized instructions to achieve higher performance than possible with a generalized unit, at an extra cost.

At the time of Ercegovac's work, there existed three primary homogeneous processing approaches in high-performance computing: (1) vector pipeline and array processors, (2) multiprocessors and multi-computers following the MIMD model, and (3) attached SIMD processors. These approaches were ubiquitous across all the early surveyed works related to distributed heterogeneous computing, and they heavily influenced the heterogeneous systems created and heterogeneous software and programming approaches used. Ercegovac [106] lists how, at the time, the three different approaches were combined in different ways to form the five following heterogeneous approaches:

1. Mainframes with integrated vector units, programmed using a single instruction set augmented by vector instructions.
2. Vector processors having two distinct types of instructions and processors, scalar and vector. An early example includes the SAXPY system, which could be classified as a Matrix Processing Unit.
3. Specialized processors attached to the host machine (AP). This approach closely resembles accelerator-based heterogeneous computing, the subject of Section 4. The ST-100 and ST-50 are early examples of this approach.
4. Multiprocessor Systems with vector processors as nodes, or scalar processors augmented with vector units as nodes. For example in PASM, mentioned earlier in this Section, the operating system supported multi-tasking at the FORTRAN level, and the programmer could use the fork/join API calls to exploit MIMD-level parallelism.
   CEDAR [172] represented another example of a multiprocessor cluster with eight processors, each processor modified with an Alliant FX/8 mini-supercomputer. This allowed heterogeneity within clusters, and among clusters, and at the level of instructions, supporting vector processing, multiprocessing, and parallel processing.
5. Special-purpose architectures that could contain heterogeneity at both the implementation and function levels. The Navier-Stokes computer (NSC) [262] is an example. The nodes could be personalized via firmware to respond to interior or boundary nodes.

Five years later, another relevant survey, *Heterogeneous Computing: Challenges and Opportunities* was published by Khokhar et al [166]. Where the previous survey focused on heterogeneous computing as a means to improve performance over homogeneous systems, this work offers an additional motivation; instead of replacing existing costly multiprocessor systems, they propose to leverage heterogeneous computing to use existing systems in an integrated environment. Conceptually, this motivation aligns closely with the goals of grid and metacomputing, discussed in Section 3.

The authors present ten primary issues facing the developing heterogeneous computing systems, which also serve as a high-level road map of the required facilities of a mature heterogeneous programming environment:

1. *Algorithm Design* - Should existing algorithms be manually refactored to exploit heterogeneity, or automatically profiled to determine types of heterogeneous parallelism?
2. *Code-type Profiling* - The process of determining code properties (vectorizable, SIMD/MIMD parallel, scalar, special purpose)
3. *Analytical Benchmarking* - A quantitative method for determining which code patterns and properties most appropriately map to which heterogeneous components in a heterogeneous system

(4) *Partitioning* - The process of dividing up an assigning an application to heterogeneous system, informed by the code-type profiling and analytical benchmarking steps.
(5) *Machine Selection* - Given an array of available heterogeneous machines, what is the process for selecting the most appropriate machine for a given application. Typically, the goal of machine selection methods and algorithms, for example the Heterogeneous Optimal Selection Theory (HOST) algorithm [65] was to select the least expensive machine while respecting a maximal execution time.
(6) *Scheduling* - A heterogeneous system-level scheduler needs to be aware of the different heterogeneous components and schedule accordingly.
(7) *Synchronization* - Communication between senders and receivers, shared data structures, and collectives operations presented novel challenges in heterogeneous systems.
(8) *Network* - The interconnection network itself between heterogeneous machines presented challenges.
(9) *Programming Environments* - Unlike today, where programmibility and productivity lie at the forefront of heterogeneous system discussions, in this work the discussion of programming environments almost seems like an afterthought. This is not unusual in works exploring early heterogeneous systems however, as hardware system-level issues were typically the primary focus. However, they do mention that a programming language would need to be independent, portable, and include cross-parallel compilers and debuggers.
(10) *Performance Evaluation* - Finally, they discuss the need for development of novel performance evaluation tools specifically designed for heterogeneous systems.

In summary, the authors call for a need for better tools to identify parallelism, improved high-speed networking and communication protocols, standards for interfaces between machines, efficient partitioning and mapping strategies, and user-friendly interfaces and programming environments. Many of these issues are addressed by the programming approaches and implementations discussed throughout this work. However, as more heterogeneous and specialized processors emerge (Sections 4.8 and 4.9), many of these issues resurface and remain as outstanding issues and challenges with today's high-performance heterogeneous computing.

In the guest editor's introduction of the 1993 ACM *Computer* journal, a special edition on *Heterogeneous Processing*, Freund and Siegel offer a high-level perspective on the then-current state of high-performance heterogeneous computing [117].

They offer several motivations for heterogeneous processing. Different types of tasks inherently contain different computational characteristics requiring different types of processors, and forcing all problem sets to map to the same fixed processor is unnatural. They also consider the notion that the primary goal of heterogeneous computing should be to maximize usable performance as opposed to peak performance, by means of using all available hardware in a heterogeneous way instead of maximizing performance on a specific processor.

Freund and Siegel also offer two potential programming paradigms: (1) the adaptation of existing languages for heterogeneous environments and (2) explicitly designed languages with heterogeneity in mind. They discuss advantages and disadvantages of both paradigms. This discussion of balance between specificity and generality in heterogeneous program paradigms continues today, with contention between specific approaches like CUDA and general approaches like OneAPI. Additionally, the authors depart from the opinion that there would be one true compiler, architecture, operating system, and tool set to handle all heterogeneous tasks well, insisting that a variety of options will likely be beneficial depending on the application and context.

In the conclusion, the authors predict that heterogeneity will always be necessary for wide classes of HPC problems; computational demands will always exceed capacity and grow faster than hardware capabilities. This has certainly proven to be true, as heterogeneous computing is a staple in today's high-performance computing.

The 1994 work by Weems et al., *Linguistic Support for Heterogeneous Parallel Processing: A Survey and an Approach* [292], is particularly interesting in the context of this project. As previously mentioned, programming approaches and methodologies are typically a minor consideration in many early heterogeneous computing works. However, this work explored the existing options for heterogeneous programming and the challenges and requirements for heterogeneous languages.

The authors define three essential criteria for evaluating the suitability of languages for heterogeneous computing: (1) efficiency and performance, (2) ease of implementation, and (3) portability. They discuss how languages would need to support an orthogonal combination of different programming models, including sequential, control (task) parallelism, coarse and fine-grained data parallelism, and shared and distributed memory. They stress that heterogeneous programming languages must be extendable to avoid limitations on their adaptability, and that abstractions over trivialities must be provided in order to not overwhelm programmers, while still providing access details needed by system software. Furthermore, they discuss the need for an appropriate model of parallelism at different levels, i.e., control parallelism at a high level, and data parallelism at a lower level. These kinds of considerations and concerns are still relevant today. For example, the ubiquitous MPI+X approach has long been the de facto solution for this kind of tiered parallelism, but requires interfacing with two standards and two implementations.

Weems et al. then survey the existing languages, and discuss their limitations with respect to their vision of a truly heterogeneous language. They include Cluster-M [107], HPF [170], Delirium [203], Linda [62], PCN [115], PVM [278], p4 [60], C**, PC++, ABCL/1 [302], Jade [254], Ada9x [276], Charm++ [161], and Mentat [126] in the discussion, some of which are explored in this project in Section 2.2. They further detail six features of an ideal heterogeneous programming language:

(1) supports any mode of parallelism
(2) supports any grain size
(3) supports both implicit and explicit communication
(4) users can define and abstract synchronizations

(5) users can define new first-class types
(6) users can specify patterns to aid data distribution

So far, a quarter-century later, no single unified heterogeneous computing solution has evolved to meet all of these requirements. Many of the developed solutions adapt a hybrid-approach, adopting multiple languages to deal with different modes of parallelism, different grain sizes, and communication layers. As discussed in Section 3.2, the popularity of these hybrid approaches can partially be attributed to the widespread adoption of and reliance on MPI. However, this does not nullify these properties of an ideal language, and many contemporary solutions are exploring more unified approaches, such as PGAS languages and OneAPI.

The authors then explain the three primary deficiencies of the then-current programming approaches: (1) dependence on external languages, (2) lack of support for extension to new models of parallelism, and (3) lack of support for transporting code between targets.

Finally, the authors outline two important features that they feel novel heterogeneous programming approaches should include, coined (1) metamorphism and (2) pseudomorphism.

To produce high-performance device code from high-level abstracted source code, compilers need to extract a sufficient amount of semantic information from the original application. If this semantic information is obscured by programming abstractions, the compiler may fail to apply appropriate optimizations. Thus the authors coin metamorphism as a term to address this issue. Metamorphism enables a language to incorporate new first-class constructs, in which new modes of parallelism can be expressed with appropriate semantic information, allowing optimizing compilers to generate efficient code. Essentially, programmers define first-class types, effectively changing the language, that the compiler can understand well, instead of user-defined types and operator overloads that may obscure semantics.

Different devices may prefer different variants of an algorithm for the most efficient execution. Pseudomorphism as a language feature, as defined by the authors, enables the management of alternate implementations of an operation for mapping algorithms to different target architectures.

We next explore the 1995 survey *Goals of and Open Problems in High-Performance Heterogeneous Computing* by Siegel et al. [268]. Siegel was very involved in the early development of distributed heterogeneous computing, and many of his publications are included in this project, including the outline of the PASM system in this section's introduction. The authors present the following goal of heterogeneous computing:

> To support computationally intensive applications with diverse computing requirements. Ideally presented to the user in an invisible way.

They continue by outlining the then-current hurdles for achieving this goal. For the existing heterogeneous solutions, the user would need to manually decompose applications and assign sub-tasks to machines. This significantly hindered the potential benefit of heterogeneous systems. However, they do also offer two examples of success stories using heterogeneous computing:

- Example 1: Turbulent Convection Simulation. Essentially, the simulation data was forwarded from device to device, with each device performing a task well suited to its architecture. First, a CM-5 device was used for a conjugate gradient method. Then a Cray 2 was used to perform a vectorized Lagrangian approach. Next, a CM-200 was used to calculate particle distribution statistics. Last, a Silicon Graphics VGX simulation was used to visualize the simulation using an interactive volume renderer.
- Example 2: Cancer Treatment Application. Like the simulation, this application pipelines the data through three different devices. First, a CRAY-MP is used for radiation dose calculations and model interpolations. Then, a Silicon Graphics 340 VGX was used directly by physicians to interact with the model. When the model was adjusted by the physician, the new viewpoint information was sent to a Pixel Planes 5 for rendering. If a treatment plan was modified, the changes were sent back to the CRAY-MP, and the pipeline restarted.

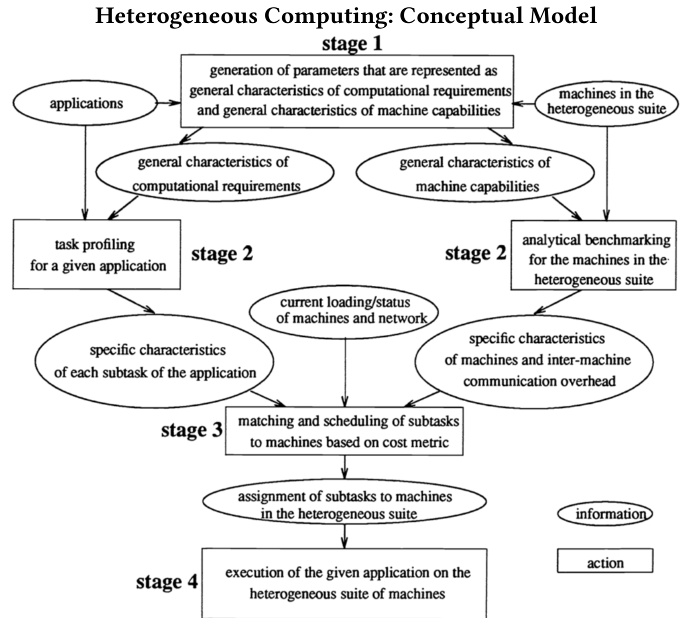

Figure 1: Siegel et al. [268]

Looking to the future, this survey introduced a conceptual model for an end-to-end heterogeneous programming and computing approach, shown in Figure 1. Many of the components in the model addressed the issues and challenges outlined by Khokhar et al [166], including task profiling, analytical benchmarking, and partitioning and machine selection. Although the model is conceptual, as no complete implementation existed at the time, the model and derivations of it appear frequently in the subsequent heterogeneous computing literature. The stages are summarized as follows:

- Stage 1: Generate a set of parameters of interest for codes and devices. Essentially, this consists of general properties of machines and algorithms that might be used to determine suitability of algorithms for machines.
- Stage 2: Independently analyze the application (task profiling), and the heterogeneous machine (analytical benchmarking). In the application, extract the types of tasks and

operations performed. For the heterogeneous machine, perform benchmarking to see which types of tasks map well to each different device.
- Stage 3: Use the information from Stage 2 to match application tasks to devices, and schedule the application across the heterogeneous machine.
- Stage 4: Execute the application on the heterogeneous machine.

The survey concludes with several open questions with respect to heterogeneous computing. Many of these questions are still relevant today, and important when considering contemporary heterogeneous programming approaches.

(1) What information should the user provide, and what should be automatically determined by the system?
(2) How do you strike a balance between the amount of information provided by the user and the expected performance of the system?
(3) Can you develop a machine-independent language to allow users to augment code with compiler directives, facilitate compilation of the program into efficient device code for any device in the heterogeneous machine, decompose the tasks into homogeneous sub-tasks, and use machine-dependent subroutine libraries?
(4) How do you develop debugging, performance, and visualization tools for heterogeneous systems?
(5) How can we address policy issues requiring system support?

The first two questions are especially relevant in the context of this project, and explored further in later sections.

The same authors, Siegel et al., published another survey *Software Support for Heterogeneous Computing* [269] in 1996. This survey addressed many of the same points as their previous survey. They similarly conclude that, with the then-current heterogeneous systems users were required to decompose applications into subtasks and manually map partitions to target machines. They stress that a long-term goal of heterogeneous computing should be to automate this decomposition, partitioning, and scheduling, but that research tools could be developed to aid programmers until full-automation is realized. They conclude by stating that portable heterogeneous programming languages at each stage in the conceptual model need significantly more research until they could be implemented in a practical way.

2.2 Distributed Heterogeneous Programming Languages and Environments

We now explore the different languages and programming approaches developed for distributed heterogeneous computing during this time period, approximately 1985-1995.

2.2.1 PVM: Parallel Virtual Machine. While most parallel computing research at the time focused on computation models, algorithms, or machine architectures, the PVM project [278], started at Oak Ridge National Laboratory, was an early attempt to provide a unified programming model for both homogeneous and heterogeneous distributed environments. PVM primarily consisted of a set of user interface primitives that could supplement a language of choice, typically C or FORTRAN. The overarching goal of PVM was to allow a diverse and scalable set of heterogeneous computer systems to be programmed as a single parallel virtual machine. Essentially, PVM was designed as a programming environment for interacting but independent components.

Prior to PVM, applications were typically written with machine-specific function calls for inter-process communication, shared memory operations, and thread spawning. PVM also included primitives for generalized locking and barriers, although shared-memory loop-level locking still required internal locking constructs. Where most machines at the time required manual coordination between diverse computation models and architectures, PVM primitives provided a portable alternative to these machine-specific constructs.

A PVM application would consist of components, which are separate stages of the application, not just subroutines. Users can the specify which components should execute on which machines, and coordinate communication between the components via message passing. PVM also provided shared-memory emulation on distributed machines, although at the cost of significant performance degradation.

This application view allowed a PVM programmer to execute specific algorithms within an application on the most appropriate hardware available, one of the fundamental goals of heterogeneous computing even today. It also allowed programmers to utilize hardware resources that may have existed but would have been otherwise wasted.

The PVM project allowed for experimentation with trade-offs between versatility and efficiency, specifically comparing the abstraction provided by a shared memory view, and the performance degradation of emulation. PVM was created to meet the existing and expected need for cheap, flexible, and portable parallel programming environments.

Several application examples are presented in the initial PVM publication [278]. These examples, including a Global Environment Simulation (GES) and Block Matrix-Multiplication (BMM), highlight both the capabilities of PVM, and the general nature of problems targeted by heterogeneous computing at the time. In the GES application, a vector processing unit is used to compute fluid flows, a distributed multiprocessing system is used to model contaminant transport, a high-speed scalar machine is used to simulate temperature effects, and a real-time graphics system is used to support user interaction. The separate components shared data using both shared memory and message passing. GES represents an application well-suited to a heterogeneous environment due to the diverse requirements across the different components of the applications.

BMM represents an alternate use case within the PVM environment. For BMM, each computing system performs the same task, but using different algorithms depending on the machine type: square sub-block for hypercube architectures, a static pre-schedule approach on the Sequent machine, and sequential execution on the Sun workstation. This application demonstrates how PVM can be used to execute an inherently homogeneous application across a diverse computing system using a unified programming model.

Figure 2 demonstrates an example of a matrix multiplication decomposition, performed using PVM. The decomposition assigns partitions to an array of different devices in a heterogeneous system, with each device receiving a block size proportionate to its computational abilities, at least for matrix multiplication.

Matrix Multiplication Decomposition

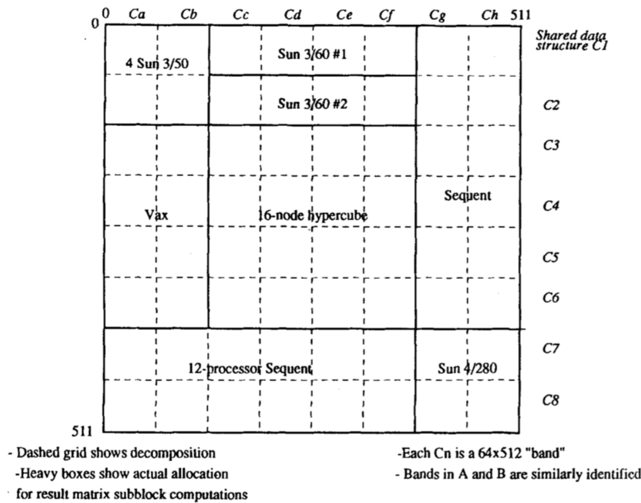

Figure 2: asymmetric decomposition for matrix multiplication[278]

PVM was also used as the parallelization strategy in many other works. A widely cited (over 19,000) Crystallography and NMR system [55] relied on PVM to provide parallelism and portability across computing platforms. In fact, prior to the ubiquitous rise of MPI, PVM existed as the de facto standard for distributed computing [122]. PVM also inspired the HeNCE project, discussed in the next section [39].

2.2.2 HeNCE. A suggested future work in the PVM publication was a GUI for component order and interactions. The HeNCE project [39–41] tackles this objective. HeNCE, a Heterogeneous Network Computing Environment, implements a graphical interface on top of the PVM framework.

One of the goals of HeNCE was to provide supercomputer performance from inexpensive local-area-network systems. HeNCE provides programmers a way to easily map conceptual software-design graphs to a graphical interface, where nodes represent FORTRAN or C subroutines, and arcs represent data control flow.

In the HeNCE system, the programmer was responsible for specifying parallelism, but the environment includes support for compilation, execution, debugging, and performance analysis. The application was represented as a control flow diagram within the GUI, and the user can even display an animated view during the application execution.

Some works related to HeNCE include Paralex and Isis [33], Network Linda [62], Piranha [61] and Condor, and Pthread. The designers of HeNCE admitted performance and efficiency was an area needing improvement and future work.

HeNCE represented an early attempt to create a very high-level interface for heterogeneous computing, opposed to interfacing directly with low-level device-specific code. Although the user was still required to write code for each specific device once, they provided an automated way to switch between device codes.

2.2.3 Linda. The Linda programming model [62, 63], a product of Yale University, was frequently discussed in the literature as an attractive option for heterogeneous computing. Linda diverged from the mainstream models of distributed computing, or coordination systems, such as message passing, concurrent logic languages, and parallel functional programming systems. Instead, Linda adopted a tuple-space model of parallel programming, where instead of multiple processes exchanging messages, they pushed data to and pulled data from an uncoupled "tuple-space". The tuple space existed separately from the black-box computation of programs.

Although Linda was not originally designed for heterogeneous programming systems, it became an attractive option for heterogeneous computing for several reasons. First, Linda implementations already existed for several machines. Also, there were no restrictions in the Linda programming model that prohibited heterogeneity. Finally, Linda tuples used in communication were uncoupled from Linda processes, and language and machine independent, making them ideal for heterogeneous systems.

For example, with the right implementation support, processes on different machines written in different languages can access the same tuple space without explicit programmer coordination between the different programming environment.

Although conceptually the Linda programming model was well suited for heterogeneous computing, the initial Linda implementations lacked support for distributed heterogeneous computing. Most extensions by the Linda authors, such as Piranha [123], targeted objectives other than heterogeneous processing. However, one project developed a backend to Linda based on the p4 system [59], which enabled users to write Linda applications for heterogeneous systems.

2.2.4 p4. The p4 project [60], developed at Argonne National Laboratory, was a C and FORTRAN library originally stemming from a number of previous Argonne projects, including the Monomacs and Parmacs libraries. Like many of the other early projects, the main goals of p4 were portability, efficiency, and simplicity. The p4 project juggled these goals by providing multiple ways to do things, absorbing complexity into the p4 library, and making judgments to sacrifice one of the primary goals in certain situations.

The p4 library primarily consisted of subroutine calls, macros, and type-def data types. At the start of a p4 application, a user would use a process group routine, which would read a supplied process group file, and then create and assign processes, either on a single machine or across several possibly heterogeneous machines. Each process would be assigned a process ID (analogous to an MPI rank).

The project supported shared-memory parallelism using a monitor paradigm (shared-memory MIMD). This included abstractions for vendor-specific locks, semaphores and other shared-memory synchronization operations. The p4 project also supported distributed-memory parallelism (distributed-memory MIMD) via message passing, and provided abstractions such as send, receive, and collective operations (min, max, barrier, sum, product, etc.)

Several successful projects were built using p4, typically projects within the Argonne National Laboratory umbrella. Argonne's theorem-prover [204], a maximum-likelihood method for phylogenetic trees,

several computational chemistry applications, a method for computing resonances in piezoelectric crystals, and the SPLASH benchmarks all used p4, either directly or internally, to support either shared- or distributed-memory parallelism.

As previously mentioned, p4 was also used as a backend to enable heterogeneous computing for other programming environments. The p4-Linda project [59], developed by the p4 project team, developed a p4 backed for the Linda programming language.

Additionally, as part of the Chameleon project [143], p4 was indirectly used in the reference MPI implementation, which was co-developed with the MPI standard [129]. Interestingly, the p4 authors had aspirations that vendor-specific MPI could provide performance improvements to p4. Directly quoted, "We hope that vendor implementations of MPI will replace p4's portability layer with corresponding improvements in efficiency" [60]. However, as we discuss later, this never became the reality, as MPI became a standard for frontend programming and replaced many of the other distributed computing frameworks.

2.2.5 Mentat. Mentat [126] and associated extensions [127], introduced and researched in the early 90s at the University of Virginia, represent another early attempt at providing a comprehensive programming environment for HPC systems. Mentat initially started as a homogeneous parallel processing system, with three primary goals: to provide easy-to-use parallelism, to provide high performance via parallel execution, and to provide application portability across different platforms.

The Mentat architects acknowledged that the then-current state of parallel programming demanded too much from the programmer, regarding manual management of communication, synchronization, scheduling, etc., and often overwhelmed the programmer. Their proposed solution allowed a programmer to simply describe the parallelism within the program, and deferred management of parallelism to the compiler. This approach is still desirable today, albeit at vastly different abstraction levels than Mentat.

The Mentat Programming Language extended C++ with specialized Mentat classes, or classes intended to be operated on and executed in parallel. Because the underlying parallelism in Mentat was based on a macro dataflow model (MIMD) computation, Mentat classes typically included computationally expensive routines or member functions with high latency. These classes could then be launched or executed asynchronously, allowing for parallelism via overlapping computation. Classes with objects like shared data structures or queues would also be implemented as Mentat classes.

From a programming perspective, Mentat provided several advantages. There existed no shared or distributed memory model, as only Mentat objects could communicate directly, and this communication was managed by the underlying compiler. Similarly, although Mentat objects were launched asynchronously, the Mentat runtime would block if any function-call arguments were returned by another Mentat objects, or even automatically forward data between Mentat objects.

These abstractions allowed Mentat code to be completely source-portable. The authors mention developing Mentat-based applications on a Sun workstation, and then running the application on an Intel iPSC/2, needing only to modify the grain-size selection [126]. Although Mentat did provide source portability, the initial implementation still assumed a homogeneous system at execution time. However, as we see in Section 3, the Mentat development team quickly retargeted Mentat to focus on distributed heterogeneous systems, specifically "Metasystems".

2.3 Distributed Heterogeneous Systems Discussion

Thirty years later, many of the visions of the developers of early distributed heterogeneous systems are still just that—visions. As we see in the later sections of this project, most modern heterogeneous programming approaches still require some manual management of data transfers, communication, and synchronization, although typically with more user-friendly programming approaches than those of early systems like Mentat. Much of the research and discussion today in heterogeneous computing revolves around finding the appropriate abstraction level.

The diversity of processors in these early heterogeneous distributed systems seems small relative to today's array of co-processors (GPUs, FPGAs, TPUs, etc.). These early processors would all typically fall into the "traditional CPU" in today's categorization.

However, the diversity in supporting hardware and software was far greater in the early heterogeneous than today's typical cluster and supercomputer environments. Because the sub-components were typically completely separate machines, they experienced heterogeneity in the network architecture, connection latencies, different communication bandwidths for different machines. On the software side, different machines had different operating systems, different process support and inter-process communications, varied compiler and language support, and multiple file systems. Unlike today's cluster and supercomputing environments with mostly homogeneous software environments, early distributed heterogeneous system approaches required masking these network and software diversities.

3 METASYSTEMS, GRID COMPUTING, AND CLUSTERS 1995 - 2005

Around the turn of the century, the keywords and terminology surrounding heterogeneous distributed systems research began to shift. The next realization of heterogeneous computing systems began to be referred to as Metasystems, or referenced in the context of Metacomputing, and Grid Computing. This shift in perspective reflected a more universal or global outlook on heterogeneous computing. This section serves as a brief intermission between the start of heterogeneous computing with distributed heterogeneous, and the current realization related to accelerators. We explore and discuss the ways in which distributed heterogeneous computing, coincident with the rapid and impressive growth of the internet and web-based computing, expanded into Metacomputing, Grid Computing, and eventually cloud-based computing. We also explore the impact of MPI on the development of heterogeneous programming approaches and systems.

3.1 Meta, Grid, and Cluster Architectures

3.1.1 MetaSystems and MetaComputing. Previously, distributed heterogeneous systems consisted of a few different types of processors, with a goal of partitioning a single application across a small number of diverse machine-types. As we see in this section, the metasystems approach aimed to significantly extend the scope of distributed heterogeneous computing, creating massive virtualized environments to support extremely distributed applications and users.

In their 1998 survey-style publication, Grimshaw et. al discuss the high-level goals of metacomputing [125]. In essence, a metasystem should "create the illusion of a giant desktop computational environment" [125]. By connecting a huge amount of distributed resources in a virtualized way, metasystems can present advantages across several different areas: (1) they can allow more effective collaboration between remote developers, (2) the increase in overall resources can improve overall application performance, (3) they can facilitate access to remote data and compute resources, and (4) the simplified programming environment can improve productivity for the development of meta-applications.

They argued that a metasystems approach solves a lot of the problems of traditional approaches like HPF, MPI, or PVM, where separate components of an application are maintained by separate groups, with potentially separate licenses and separate software stacks [125].

They discussed several promising projects attempting to tackle metacomputing, including AppLes [44], Globus [114], Legion [128], and Network Weather Service [295], several of which are discussed below in the context of Grid Computing. They projected that metasystems would have a significant impact on scientific productivity.

In a complementary 1998 survey [196], the same team expresses many of the same sentiments. They discuss the goal of a "transparent, distributed, shared, secure, fault-tolerant computational environment." They discuss means to achieving this goal, and the technical challenges of transparent remote execution and a shared persistent object file space across a distributed metasystem. Overall, this survey primarily focuses on the hopes and necessities for future systems.

3.1.2 Grid and Cloud Computing. Grid computing emerged as the next key term related to heterogeneous computing, following and closely related to metasystems. Many of the projects originally coined as metasystems are also discussed in the context of grid computing, including Mentat and Legion.

In a 2000 Survey, *The Grid: A survey on Global Efforts in Grid Computing* [34], Baker et al. present three general principles for Grid Computing.

- Heterogeneity: resources are heterogeneous in nature across numerous administrative domains and geographical distances
- Scalability: latency tolerant applications, no degradation of performance as grid grows
- Dynamicity or Adaptability: resource failure is the rule, not the exception. Must dynamically extract performance form available resources

Baker et al. also describe the components required to build and coordinate a Grid Computer.

- Grid Fabric - the physical resources distributed across the globe
- Grid Middleware - core services, remote process management, allocation of resources, security
- Grid Development Environments and Tools - high-level services that allow programmers to develop applications
- Grid Applications and Portals - applications developed using things like HPC++ and MPI. Portals allow users to submit jobs via web-enabled applications services

Baker et al. continue by listing several different programming frameworks either directly or indirectly related to Grid Computing, including Globus [114], Legion [128], WebFLow [47], NetSolve [21], NASA IPG [156], Condor [282], Harness [37], Hotpage [283], Ninf [259], Nimrod [3], DISCWorld [142], and MetaMPI [100]. We briefly explore some of these approaches in Section 3.2.

In their outro, Baker et al. explain the rising role of Java in grid and network computing, and they project that Grid Computing as a field would have a revolutionary effect, similar to that of the invention of the railroad. As we see later in this project, in some ways it has, and in other ways it has not.

In his 2003 survey *The Grid: Computing without Bounds*[113], Ian Foster, one of the developers of Globus, provides a high-level view of the role of Grid Computing.

> We would not accept a situation in which every home and business had to operate its own power plant, printing press, and water reservoir. Why should we do so for computers?

Foster, a major figure in the foundation of grid computing, viewed virtualized computation as a natural extension to the internet. The rapid increases in internet speeds infrastructure at the time led to a high level of optimism in the possibilities of remote computers. The goal was to integrate software into existing systems to enable sharing information, data, and software to create a virtual organization structure, instead of replacing systems at participating sites.

Foster describes several examples of existing Grid Computing network systems, including:

- European Data-Grid [263]
- US Grid Physics Network
- Particle Physics Data Grid
- Sloan Digital Sky Survey [118]
- National Digital Mammography Survey
- US Biomedical Informatics Research Network [163]
- US Network for Earthquake Engineering Simulation [56]
- 2002 Launch of Open Grid Services Architecture [112]

In a parting note, Foster notes that for broad adoption of Grid Computing, access to core technologies should be free and open source.

In 2008, five years later, Foster et al. published another survey, *Cloud Computing and Grid Computing 360-Degree Compared* [116]. In this work the authors compared the decade-strong grid computing concepts with the newly emerged cloud computing paradigm. They posed the following question,

> Is cloud computing a new name for Grid?

and answered as follows:

- **Yes**: the vision is essentially the same, to reduce cost, increase reliability, and increase flexibility of distributed heterogeneous computing. Cloud simply shifts from user operation to third-party operation, and cloud computers can be considered viable commercial grid computing providers.
- **No**: The scale for cloud computing is much larger. Instead of linking up commodity clusters, cloud systems contain over 100 thousand computers and are owned by large corporations like Amazon, Google, and Microsoft. This introduces an entirely new set of problems.
- **But Yes**: The technological challenges in cloud and grid computing mostly overlap, in that they both manage large facilities, and define methods for users to discover, request, and consume provided resources. And intrinsically, both cloud and grid rely on highly parallel computations.

The authors describe how cloud computing is driven by economies of scale, and that their viability is driven by rapid decreases in hardware costs, increases in storage capacities, exponential growth in data demands, and the adoption of services computing and web 2.0. Cloud computing evolved from grid computing and relies on grid computing technologies as its backbone, but operates on a very different model. With grid computing, the model is based on large, well-planned and well-funded projects with a significant scientific impact. With cloud computing, thousands of processors can be accessed with only a credit card.

The authors further describe the different enabling technologies and software for grid and cloud computing. Grid applications are usually HPC oriented, and rely on software like MPI [129], MapReduce [85], Hadoop [267], and coordination languages like Linda [62]. In contrast, Cloud applications can be large-scale, but Cloud infrastructures do not typically support fast low-latency network interconnects needed in HPC. Cloud programming models are typically a mashup of scripting, Javascript, PHP, Python, HTTP, and web services.

3.1.3 Clusters. While the metasystems approach aimed to connect together existing machines using a software overlay, other alternatives to the classic supercomputer were being developed during this time period. The notion of clusters, or networks of workstations (NOWs), sought to blend the lines between personal machines and supercomputers, delivering supercomputer-like performance with personal workstation components.

The Berkeley NOW architecture [19], first introduced in 1995, aimed to utilize the strengths of both personal machines and supercomputers, while eliminating their weaknesses. They proposed that NOW architectures would outlast and outperform both small computers for personal interactive use and traditional large supercomputing machines for several reasons. At that time, the performance improvements for personal workstation machines were significantly outpacing the improvements for large supercomputers. The larger volume of workstations being produced allowed manufactures to amortize the development costs and push the rate of technological innovation. More simply put, the smaller machines could be offered at a higher cost/performance ratio than supercomputers. Also, the development cycle for supercomputer architectures lagged significantly behind the workstation counterparts. Another weakness of traditional supercomputers was the high cost of changing the operating systems and other commodity software.

A NOW system approach circumvents these issues by merging local area networks and massively parallel processors. A NOW system contained an array of commodity workstation components, Ethernet, a system like PVM to communicate between the components, and a sequential file system. Although the original conceptual presentation of NOW systems [19] referenced PVM for the software communication layer, the first actual implementation of the Berkeley NOW architecture [80] in 1997 employed MPI, foreshadowing the shift from PVM to MPI for cluster-based computing.

While the shift to MPI for the Berkeley NOW architecture disqualified it for more heterogeneous computing, other NOW architectures were more heterogeneity-oriented. For example, the heterogeneous NOW architecture developed in 1995 at the University of Texas at San Antonio [304] relied on the PVM programming model for the communication layer. Their system contained SPARC 10-30 workstations, SPARC 5-70 workstations, and classic SPARC workstations, all connected by Ethernet.

3.2 Meta, Grid, and Cluster Languages and Environments

Although the various computing paradigms covered in this section all stemmed from distributed heterogeneous programming, and maintain heterogeneity as a core component, the different paradigms employ vastly different programming environments and languages. In this section, we briefly discuss a few of the most widely used programming approaches in each area.

3.2.1 Meta and Grid Computing Languages. Many of the programming approaches for meta and grid computing developed as extensions to their distributed heterogeneous computing counterparts. However, several approaches revolutionized the ideas from distributed computing, aiming to create more global solutions.

Mentat Extended: In a work extending Mentat [127], discussed in Section 2.2.5, Andrew Grimshaw and the team at the University of Virginia discuss Mentat's shortcomings when retargeted for a heterogeneous environment, steps to be taken to overcome these deficiencies, and experiments with the application to metasystems.

The Metasystems approach for Mentat addressed three primary issues: hand-writing parallel applications is difficult, codes are not portable across MIMD architectures, and the wide variety of available systems cannot be simultaneously utilized for a single application. While the original Mentat implementation focused on the first two issues, the extension primarily focused on the third issue.

Their approach involved pulling from two separate but related communities: the parallel processing community, and the heterogeneous distributed computing community. However, the primary focus for the Metasystem approach was performance, as the authors argued that significant performance degradation would render the implementation useless. The motivation for using a heterogeneous system is lost if performance is lost. Therefore, they started with a parallel performance system, Mentat, and extended by including heterogeneous distributed computing concepts and features as needed, as long as they were not accompanied by significant performance sacrifices. This is a different approach than other systems

that were built to target heterogeneity, and tackled performance issues as they arose.

In order to retarget Mentat for metasystems, the authors needed to address two primary issues: data format and alignment issues between systems, and scheduling across diverse processors. The data format issues were relatively straightforward, and had previously been solved by related works, but the scheduling issues were still novel and largely unexplored at the time.

The main contributions of the Mentat extensions involved automatic methods for domain partitioning and decomposition. These methods relied on programmer call-backs and run-time heuristics. They took a two-faceted approach to scheduling, first focusing on partitioning (driven by granularity and load balancing), then on placement, which involved selecting the best available processor for a certain task. In order to implement an automated scheduler, the authors developed heuristics, a simplified machine model, and programmer-provided call-backs.

Globus: The Globus Project [114], pioneered by Ian Foster and Carl Kesselman, dominates the Meta and Grid computing literature, and Foster and Kesselman were heavily involved in both areas. Globus was funded as a multi-institutional DARPA project, and the initial version of the toolkit was first released in 1997.

The goal of Globus was to abstract a collection of distributed heterogeneous resources as a single virtual machine. Globus approaches this challenge using a layered architecture, presenting a Globus Metacomputing Toolkit containing a bag of high-level global services and low-level core services that developers can selectively apply depending on their needs. As we see in subsequent sections, these Globus services were also used as the backbone of other grid computing systems.

- *GRAM* - resource allocation and process management
- *Nexus* - unicast and multicast communication services
- *GSI* - authentication and related security services
- *MDS* - distributed access to structure and state information
- *HBM* - monitoring of health and status of system components
- *GASS* - remote access to data via sequential and parallel interfaces
- *GEM* - construction, caching , and location of executables
- *GARA* - advanced resource reservation and allocations

Webflow: Webflow [47] was a web browser interface built on top of several Globus tools, including MDS, GRAM, and GASS. However, the actual Webflow interface was conceptually separated from Globus, and thus could have been implemented on the backend using other systems like Legion or COBRA.

Webflow users could build and compose applications using a visual web-based interface and visual module icons. In many ways, this conceptually maps well to the massively distributed goals of Grid Computing.

Legion: Grimshaw et. al, the team responsible for Mentat and several of the surveys reviewed so far in this project, presented *Legion* [128] as their Grid Computing solution.

The Legion approach was based on and written in the *Mentat Programming Language* (MPL). In Legion, everything is an object, and the approach is organized by classes and metaclasses. Like Mentat, classes are defined as serial, or parallelizable, instructing the underlying compiler on how to approach an application. Users can define new classes, and classes manage their own instances. However, many classes related to parallelism are already defined in the Legion framework: host objects, persistent storage objects, binding agents mapping objects IDs to physical addresses, context objects mapping context names to Legion object IDs, etc.

Legion presented a very object-oriented approach, in contrast to the more tool and layer-based approaches of Globus and Webflow. We do note that this realization of Legion is not directly related to the contemporary realization discussed in Section 4.4, and shares only a name and the conceptual idea of the definition of "legion".

3.2.2 PVM and MPI. For a period of time after its initial development and release, PVM [278], discussed in Section 2.2, was a popular option for both homogeneous and heterogeneous distributed computing. PVM was a crucial technology for advancing distributed computing as a whole, but its longevity was overshadowed by MPI.

The Message Passing Interface, MPI [129], had a very significant effect on the then-current distributed heterogeneous programming systems. MPI worked to standardize many of the behaviors and functionalities of distributed computing implementations like PVM, p4, and others, and was largely successful to that end. In fact, MPI was so successful that it quickly became the de facto standard for distributed computing.

P4 and PVM were actually used internally for reference implementations for MPI as it was developed, a promising sign for the future of distributed heterogeneous computing. However, the MPI standard itself detailed few facilities for machine interoperability, rendering a standard-adhering implementation unsuitable for heterogeneous computing. Vendor-supported and large open source implementations like MPICH [130], relied on a more homogeneous approach, targeted large homogeneous clusters, and required multiple installations for a heterogeneous systems that may not be interoperable. So although MPI standardized and simplified distributed computing compared to previous approaches, the distributed heterogeneous computing facilities were inhibited.

Although the core MPI standard and implementations were not ideal for heterogeneous computing, several research projects extended MPI to enrich support for heterogeneous computing, though none of these implementations have experienced the longevity of the more popular MPI implementations, which are still developed and maintained today, nearly 20 years later.

- **MPICH-G** [162] A grid-enabled MPI implementation
- **PACX-MPI** [119] A metacomputing-oriented MPI implementation
- **PVMPI** [109] An Integration of the PVM and MPI Systems

3.3 Metasystems, Grid Computing, and Clusters Discussion

In this section, we briefly explore the transition from distributed heterogeneous computing into metacomputing and grid computing, and how these set up the backbone for the monolith that is today's cloud computing.

The goals of meta and grid computing were to create infinitely scaling systems by harnessing the power of remotely connected heterogeneous systems. While some projects tackled this, these ideas

were ultimately re-purposed for commercial success under the umbrella of cloud computing. Additionally, with respect to scientific endeavors, the construction of large-scale homogeneous clusters and supercomputers beckoned a shift from distributed heterogeneous machines. At the same time, the growth of MPI, without a major focus on heterogeneous interoperability, overshadowed projects like PVM and p4 that targeted heterogeneous systems.

Finally, the very things that made early machines heterogeneous began to be integrated into single homogeneous processors. Unlike mixed-mode machines like PASM with distinct SIMD and MIMD processing, many new multi-core vectorizing processors seamlessly integrate both SIMD and MIMD capabilities, which forgoes the need for a heterogeneous programming environment. Similarly, as we previously discussed, early distributed heterogeneous systems contained separate processors for visualization, statistics, and data processing. However, with the expansion of x86 and inclusion of specialized and vector instructions on general purpose CPU processors, the problems these early heterogeneous systems tackled can again be solved by homogeneous systems.

The shift into cloud computing, the ubiquity of MPI, and the continuous consolidation into x86 CPUs in many ways signaled the end of heterogeneous computing as it was originally imagined. However, as we see in Section 4, the rebirth of heterogeneous computing, and reinvention of many of the ideas previously mentioned, was sparked by the introduction of accelerator-based heterogeneous systems.

4 MULTICORE, MANYCORE, AND ACCELERATOR-BASED HETEROGENEOUS SYSTEMS 2010 - 2020

In this section, we follow the evolution of chips from a single core, to multi-core and manycore chips, which eventually developed into hardware accelerators. These developments eventually revolutionized the architectures of nearly all high-performance machines, and effectively rebirthed the field of heterogeneous computing.

The construction of large homogeneous machines marked the end of the 2000s decade and the end of heterogeneous distributed systems like we saw in the 1980s and 1990s. Jaguar [48], built around 2009 at Oak Ridge National Laboratory, was a Cray XT5 system, consisting of 224,256 x86-based AMD CPU cores, and was listed as the world's fastest machine in 2009 and 2019. Kraken [79], another Cray Xt5 system built in 2009, was listed as the world's fastest academic machine at the time. These homogeneous machines dominated the domain of HPC for several years. Likewise, HPC software support, programming approaches, and compiler infrastructure developed during this time was also largely homogeneous. However, at the same time, scientific programmers began experimenting with programming using Graphics Processing Units, or GPUs, a trend that would eventually revolutionize the HPC field.

In 2000, Toshiba, Sony, and IBM collaborated on the Cell Project [134]. This project culminated in the release of the Cell Processor in 2006. While not strictly a GPU, the Cell Processor was one of the first architectures to employ accelerator-based heterogeneity to multimedia and general purpose applications. The Cell Processor's first major commercial application was inside the Sony PlayStation 3 gaming console. In 2008, IBM and Los Alamos National Laboratory (LANL) released the Roadrunner supercomputer, which consisted of a hybrid design with 12,960 IBM PowerXCell and 6,480 AMD Opteron dual-core processors [35]. The IBM PowerXCell processors integrated the original Cell processor design.

While the Cell processor generated excitement and a new interest in a different type of heterogeneous computing, the Cell processor was only efficient for certain computations, and the overhead of manually transferring memory to and from the device was difficult due to the small memory size of the Cell architecture. Although GPUs and other heterogeneous accelerators suffer from these same issues, they evolved and developed to meet the demand of scientific computing.

The scientific community began evaluating GPUs for general purpose processing well before their use became mainstream. In 2001, researchers evaluated general purpose matrix multiplication, and in 2005 LU decomposition on a GPU was shown to outperform a CPU implementation [91]. Interest in utilizing GPUs in scientific computing continued to grow, but was inhibited by the complex programming approaches to GPUs, which typically required a low-level graphics interface and dealing with shaders and graphics-related APIs data structures. However, a major development for GPU programming, and the whole field of scientific heterogeneous programming, came with the development and release of of Nvidia's CUDA toolkit.

4.1 CUDA and GPGPUs

Nvidia was formed in 1993, but first gained major traction by winning the contract to develop the graphics hardware for the Microsoft Xbox gaming console in 2000. Nvidia continued to grow and increase its claim in the GPU market with the release of the GeForce line, in direct competition with AMD's Radeon line. However, these devices were still targeted toward graphics processing.

As the interest in scientific computing using GPUs continued to grow, Nvidia first recognized the potential financial advantages of supporting this community. In 2007, Nvidia launched the Tesla GPU, aimed at supporting general purpose computing, and the CUDA (Compute Unified Device Architecture) API and programming platform [77].

The CUDA programming platform abstracted programming GPU hardware into an API that was more consumable by scientific programmers and other programmers without extensive graphics programming experience. The CUDA programming model essentially presents a hierarchical multi-threading layout, where threads are executed as a 32 or 64-thread warp, warps are mapped onto thread-blocks, and thread-blocks are mapped onto a grid and grid blocks (Figure 3). These abstractions fit quite naturally with the nested loop structure of most scientific software. Listing 1 shows an example CUDA application, sourced from Nvidia's website (https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/).

As the popularity of CUDA and GPGPU programming grew, several large supercomputers began including both host CPUs and GPU accelerators. In 2010, China's Tianhe-1A machine launched, containing 14,336 Xeon X5670 processors and 7,168 Nvidia Tesla M2050 general purpose GPUs [301]. This heterogeneous machine overtook the previously mentioned Jaguar machine from Oak Ridge National Laboratory (ORNL) as the "world's fastest supercomputer".

Listing 1: Example CUDA C Application

```
1  #include <stdio.h>
2
3  __global__
4  void saxpy(int n, float a, float *x, float *y)
5  {
6    int i = blockIdx.x*blockDim.x + threadIdx.x;
7    if (i < n) y[i] = a*x[i] + y[i];
8  }
9
10 int main(void)
11 {
12   int N = 1<<20;
13   float *x, *y, *d_x, *d_y;
14   x = (float*)malloc(N*sizeof(float));
15   y = (float*)malloc(N*sizeof(float));
16
17   cudaMalloc(&d_x, N*sizeof(float));
18   cudaMalloc(&d_y, N*sizeof(float));
19
20   for (int i = 0; i < N; i++) {
21     x[i] = 1.0f;
22     y[i] = 2.0f;
23   }
24
25   cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
26   cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
27
28   // Perform SAXPY on 1M elements
29   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
30
31   cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
32
33   cudaFree(d_x);
34   cudaFree(d_y);
35   free(x);
36   free(y);
37 }
```

CUDA Threading Model

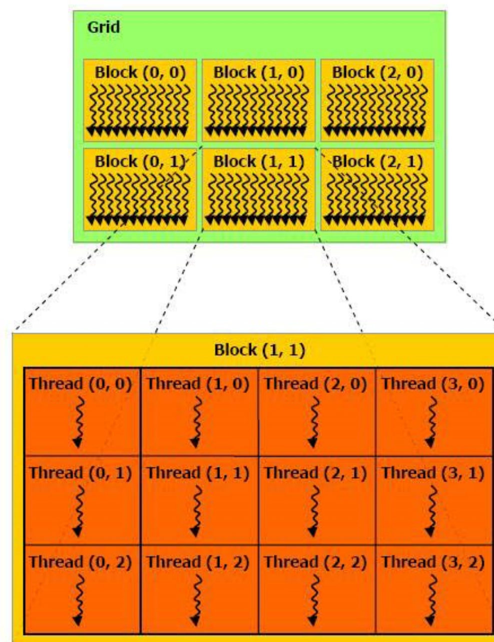

Figure 3: CUDA programming abstraction for GPGPUs "Thread Hierarchy in CUDA Programming". Retrieved 2016-09-21

ORNL's Titan supercomputer, a successor Jaguar, launched in 2013 and consisted of 18,688 AMD Opteron CPUs, each with an attached Nvidia Tesla (K20x) GPU [49]. This machine also secured the top spot as the world's fastest machine.

More recently, Nvidia GPUs and CUDA programming were employed in ORNL's Summit supercomputer [235], another machine that briefly held the title as the world's fastest. Summit was launched in 2018 and contains 4,608 nodes each with 6 Nvidia Tesla V100 GPUs. Similarly, Lawrence Livermore National Laboratory (LLNL) launched the Sierra supercomputer [202] in 2018, containing 4,320 nodes each with 4 Nvidia Tesla V100 GPUs.

Much of CUDA's success in scientific programming can be attributed to Nvidia's continued investment in and focus on CUDA training. Online and in-person training workshops, and a surplus of available training materials, made Nvidia and CUDA an attractive GPGPU option compared to other vendors. This focus on training and CUDA's success should provide a model for future heterogeneous programming approaches. Some newer approaches like OpenACC (also supported by Nvidia) have also adopted this strategy, frequently hosting learning-focused hackathons and generating significant training materials [229].

4.2 GPGPUs Beyond CUDA

Although CUDA has survived uncontested the most successful low-level GPGPU programming approach, several other low-level approaches have been developed and see a significant amount of use.

Microsoft's DirectCompute API, released in 2008 as part of DirectX 11, provided a GPGPU programming framework, specifically focused on Windows platforms [218]. The abstraction level presented by DirectCompute was very similar to CUDA, with a parallel workload being broken down into groups of thread that were then dispatched as "Compute Shaders". However, DirectCompute was quickly overshadowed by CUDA and other approaches, possibly due to its limitation to Windows-based systems. Little information on the original DirectCompute implementation can be found today.

CUDA's dependence on Nvidia devices spawned efforts to create an open source alternative. OpenCL was developed as a result of these efforts [131]. As we see in the remainder of this work, OpenCL has become a staple of accelerator-based heterogeneous programming approaches, both as a stand-alone approach and as an intermediate representation or backend for higher-level approaches.

OpenCL (Open Computing Language) was originally developed by Apple as a GPGPU option under the OSX umbrella. In early 2008, Apple submitted a proposal to the Khronos Group for creation and management of an OpenCL language [131]. On November 18, 2008 the OpenCL 1.0 technical specification was released. By the end of 2008, AMD, Nvidia, and IBM had all incorporated OpenCL support into their vendor tool chains.

Like CUDA, the OpenCL programming approach separates an application into host code and device code. The abstraction level for the OpenCL device code is very similar to CUDA, while the

Table 1: Comparison of CUDA and OpenCL GPGPU abstractions

CUDA	OpenCL
Grid	NDRange
Thread Block	Work group
Thread	Work item
Thread ID	Global ID
Block index	Block ID
Thread index	Local ID

host code abstractions are more verbose. Like CUDA, GPU cores are abstracted into a tiered parallelism. In OpenCL, work-items are executed as part of a work-group, and work-groups are organized inside an N-D range (Table 1). Listing 2 demonstrates and example vector addition application in OpenCL. From the line count alone, we can see that OpenCL requires a significant amount of low-level and boilerplate code, although this functionality is typically encapsulated in routines and libraries by frequent OpenCL programmers. However, each programmer creating a personalized set of routines to abstract OpenCL API calls creates issues with code portability and interpretability.

Although OpenCL does provide an open-source alternative to CUDA that is supported across several different device vendors (Nvidia, Intel, IBM, AMD), it has not become the de facto standard for heterogeneous GPGPU computing. First, the widespread success of CUDA and Nvidia's dominance in the GPGPU market has allowed scientific programmers to more safely choose a non-portable option. Second, the abstraction level, especially the verbosity of the host code, has led many GPGPU developers to seek higher-level abstractions, as we see in the following section. However, as we discuss later, although OpenCL has not seen widespread adoption as a programming approach, many frameworks and compilers target OpenCL as a backend API (OpenARC [187], TVM [68], etc.)

Although OpenCL and CUDA have been the most popular low-level GPGPU programming approaches over the past decade, some application developers have explored other approaches. Vulkan [265], a successor to OpenGL intended primarily for graphics computing, can outperform both OpenCL and CUDA for certain scientific applications [211].

Analogous to Nvidia's CUDA, the GPU manufacturer AMD has also released a vendor-specific API for GPGPU programming. AMD's HIP API and ROCm platform [17] provide a similar abstraction level to CUDA. The platform also provides ways to translate CUDA into HIP, allowing developers to execute CUDA applications on AMD hardware. Although the current generation of top supercomputers like Sierra and Summit employ Nvidia GPUs, future systems like ORNL's Frontier, expected to launch in 2021, will employ AMD GPUs. This transition could herald a shift away from CUDA, and increase the use of ROCm and HIP across all of scientific computing.

As discussed, there exist several low-level alternatives to CUDA for GPGPU programming. However, most scientific application developers turn to a high-level programming approach.

Listing 2: Example OpenCL C Application

```
#include <stdlib.h>
#include <CL/cl.h>

const char* programSource =
"__kernel                    \n"
"void vecadd(__global int *A,   __global int *B,   __global int *C) \n"
"{                                              \n"
"   int idx = get_global_id(0);                 \n"
"   C[idx] = A[idx] + B[idx];                   \n"
"}                                              \n"
;

int main() {
  int *A = NULL; int *B = NULL; int *C = NULL;

  const int elements = 2048;
  size_t datasize = sizeof(int)*elements;
  A = (int*)malloc(datasize); B = (int*)malloc(datasize); C = (int*)malloc(datasize);
  B = (int*)malloc(datasize);
  C = (int*)malloc(datasize);
  for(int i = 0; i < elements; i++) {
    A[i] = i; B[i] = i;
  }

  cl_uint numPlatforms = 0;
  cl_int status = clGetPlatformIDs(0, NULL, &numPlatforms);
  cl_platform_id *platforms =
    (cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
  status = clGetPlatformIDs(numPlatforms, platforms, NULL);

  cl_uint numDevices = 0;
  cl_device_id *devices = NULL;
  status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);
  devices = (cl_device_id*)malloc(numDevices*sizeof(cl_device_id));
  status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);

  cl_context context = clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);
  cl_command_queue cmdQueue = clCreateCommandQueue(context, devices[0], 0, &status);

  cl_mem bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
  cl_mem bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
  cl_mem bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
  status = clEnqueueWriteBuffer(cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
  status = clEnqueueWriteBuffer(cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);

  cl_program program = clCreateProgramWithSource(context, 1, (const char**)&programSource, NULL, &status);
  status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
  cl_kernel kernel = NULL;
  status  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
  status |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferB);
  status |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufferC);

  size_t globalWorkSize[1];
  globalWorkSize[0] = elements;
  status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize, NULL, 0, NULL, NULL);
  clEnqueueReadBuffer(cmdQueue, bufferC, CL_TRUE, 0, datasize, C, 0, NULL, NULL);

  clReleaseKernel(kernel);
  clReleaseProgram(program);
  clReleaseCommandQueue(cmdQueue);
  clReleaseMemObject(bufferA);
  clReleaseMemObject(bufferB);
  clReleaseMemObject(bufferC);
  clReleaseContext(context);

  free(A); free(B); free(C); free(platforms);free(devices);
}
```

4.3 GPGPU Research Project Languages

While CUDA has been a successful GPGPU programming approach, many developers and scientific programmers still find the abstraction level too low for daily use. Almost immediately after CUDA

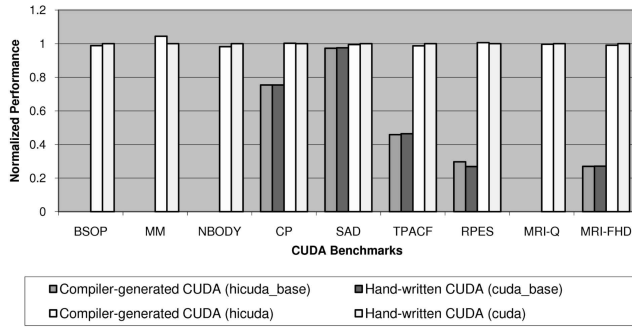

Figure 4: CUDA and hiCuda performance comparison [141]

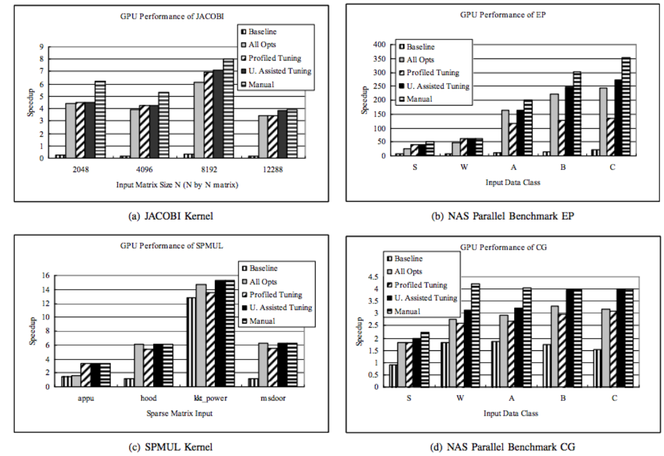

Figure 5: Performance of OpenMPC [182, 186]

was launched, research projects began to emerge offering high-level abstractions. These projects are discussed in this section. As time passed, robust production-level, high-level alternatives were developed. These languages and frameworks are discussed in Section 4.4.

4.3.1 hiCuda. In 2009, two years after CUDA first launched, Han et. al from the University of Toronto developed the hiCuda API [140, 141]. As the name suggests, hiCuda was created to abstract some of the complexities of CUDA, specifically memory management and allocation. hiCuda replaces these operations with compiler directives. Behind the scenes, a source-to-source compiler generates CUDA code from hiCuda which is then compiled with NVCC, the standard CUDA compiler. Source-to-source translation is a common tool used by academic compilers, as the burden of implementing a full-scale compiler is often beyond the scope of research projects [141, 187, 193, 252].

As an example, where a CUDA program would require the following:

```
size = 64 * 128 * sizeof(float);
cudaMalloc((void**)&d_A, size);
cudaMemcpy(d_A, A, size,
  cudaMemcpyHostToDevice);
```

hiCuda can accomplish the same with:

```
#pragma hicuda global alloc A[*][*]
```

4.3.2 CUDA-lite. A similar project, CUDA-lite [289], was developed in 2008 at the University of Illinois at Urbana-Champaign. Unlike hiCuda, which was intended to transition a sequential application into a GPU accelerated application using abstract directives, CUDA-lite was meant to be applied to existing CUDA applications, and was especially focused on abstracting memory bandwidth and coalescing optimizations. CUDA-lite was built on top of the now-discontinued Phoenix source-to-source compiler from Microsoft.

4.3.3 OpenMPC. Instead of adapting new abstractions over CUDA, some methods attempted to recycle older programming abstractions. For example, the OpenMP to GPGPU [186] developed by Lee et. al at the University of Purdue in 2009, and its extension OpenMPC [182] developed by the same team, worked to adapt the existing OpenMP programming approach for GPGPU applications.

By 2009, OpenMP [81] had already become the de facto standard for homogeneous CPU-based parallel computing. As a result, many high-performance scientific codes and applications at the time were programmed using OpenMP. Rewriting using CUDA would have been a major shift. Additionally, many programmers were already proficient in the use of OpenMP pragmas and library calls.

The goal of OpenMPC was to use existing OpenMP directives, along with minimal additional information in the form of OpenMPC pragmas, to generate CUDA GPU codes. Like the other research-project based approaches in this section, OpenMP to GPGPU and OpenMPC were built using a source-to-source compiler, in this case Cetus [83].

OpenMPC was shown to perform well on many scientific benchmarks compared to hand-written CUDA codes (Figure 5). However, because of OpenMPC's abstraction of CUDA memory management, OpenMPC did not perform as well on programs with complex memory transfer patterns between the GPU and CPU. Additionally, OpenMPC faced performance issues due to incompatibilities between OpenMP semantics and GPU programming and memory models. These incompatible regions were then computed on the host CPU instead of the GPU device.

4.3.4 StarPU. The StarPU runtime system [32, 273, 274] was developed at the University of Bordeaux in France, and first published in 2009. StarPU is an original runtime system with the goal of utilizing heterogeneous hardware without causing significant changes in programmer habits. Like many of the frameworks covered in this section, StarPU aims to allow programmers to focus on high-level algorithm issues without being restrained by low-level scheduling issues. StarPU presents a unified programming approach for CPU and GPU computing, and consists of a data-management facility and task execution engine. In the StarPU system, a central process maintains a queue of tasks, and a scheduler assigns these tasks to processing units, either a CPU or GPU.

StarPU is designed to dynamically schedule applications across a heterogeneous system, using one of several strategies:

- *greedy* - each time a CPU or GPU devices becomes available, StarPU greedily selects the task with the highest programmer-defined priority from the task queue, and begins execution,

- *no-prio* - similar to the greedy approach, but without the programmer-defined priorities,
- *work stealing* - if a processing unit has completed its task and no tasks are available in the queue, the processor attempts to steal work from other processes,
- *random weighted* - each processor is given a speed, or acceleration factor. The acceleration can be set by the programmer or measured using benchmarks. Then, tasks are scheduled according to these acceleration factors. For example, if a GPU has an acceleration factor four times higher than a CPU, the GPU should be scheduled approximately four times as much work.
- *heterogeneous earliest finish time* - performs performance or cost modeling for each task across each available processing unit. Based on these performance models, the task can be assigned to the most appropriate processing unit, either a GPU or CPU. Ideally, tasks well-suited for CPUs will be assigned to CPUs, likewise for GPU-suited tasks.

StarPU is still under active development, with a release as recent as 2020 and activity on the StarPU Gitlab page [274]. Listing 3 shows a "Hello World" code example pulled from the StarPU Gitlab site.

4.3.5 PGI Accelerator. In 2009 and 2010, another directive-based alternative to CUDA was developed the Portland Group, the PGI Accelerator [293, 294]. Although not strictly a research compiler, we include it in this section because of its scope and similarity to related research projects. The PGI Accelerator programming model aimed to target both C and FORTRAN codes, and generate executables to be run on Nvidia GPUs. The PGI Accelerator directives annotated regions of codes that appropriate for GPU computation, and were divided into two types, data directives for data regions and parallel management directives for GPU-compute regions. Using data directives, users directly controlled data transfers between the host and accelerators, even across multiple parallel or GPU compute regions. In this way, parallel regions could make use of data already transferred to GPU devices without re-transfer.

The PGI Accelerator did not provide support for mapping to architecture-specific features like CUDA shared, constant, and texture memories, or detect complex reduction patterns. Although the PGI Accelerator approach was short-lived, it did contribute significantly to the development of the OpenACC GPGPU programming approach, which is covered in depth in Section 4.4. PGI's proprietary compiler for the PGI Accelerator approach also formed the basis of PGI's successful OpenACC compiler, discussed in Section 4.6.

Certainly other research-based high-level alternatives to CUDA emerged during the time between CUDA's 2007 release, and the release of the first major production-level, high-level approaches, around 2012 (i.e., R-stream [192], HMPP [90]). Together, all of these research-based approaches laid the foundation of the more widely used and production-style, high-level approaches mentioned later in Section 4.4 and beyond.

4.4 GPGPU High-Level Programming

Over time, interest in GPU-based heterogeneous computation continually increased in the scientific community, which in turn drove the demand for more standardized and long-term high-level GPUPU programming approaches. By 2015, the research-style approaches developed between 2006 and 2011 evolved into several production-level approaches with large user bases and significant development support. Some approaches exist as large, well-documented standards supported by several device vendors (OpenMP [233], OpenACC [230]). Others were major initiatives supported by major national laboratories in the US (Kokkos [99], Raja [146]). Finally, many other approaches were implemented as libraries or wrappers inside two major languages used in scientific computing, C++ and

Listing 3: Example StarPU C Application

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <starpu.h>
4
5  #define FPRINTF(ofile, fmt, ...) do { if
6  (!getenv("STARPU_SSILENT")) {fprintf(ofile, fmt, ## __VA_ARGS__); }} while(0)
7
8  void callback_func(void *callback_arg) {
9        FPRINTF(stdout, "Callback function got argument %p\n", callback_arg);
10 }
11
12 struct params {
13   int i;
14   float f;
15 };
16
17 void cpu_func(void *buffers[], void *cl_arg) {
18   (void)buffers;
19   struct params *params = (struct params *) cl_arg;
20   FPRINTF(stdout, "Hello world (params = {%i, %f} )\n", params->i, params->f);
21 }
22
23 int main(void) {
24   struct starpu_codelet cl;
25   struct starpu_task *task;
26   struct params params = {1, 2.0f};
27   int ret;
28
29   ret = starpu_init(NULL);
30   if (ret == -ENODEV)
31     return 77;
32   STARPU_CHECK_RETURN_VALUE(ret, "starpu_init");
33
34   task = starpu_task_create();
35
36   starpu_codelet_init(&cl);
37   cl.cpu_funcs[0] = cpu_func;
38   cl.cpu_funcs_name[0] = "cpu_func";
39   cl.nbuffers = 0;
40   cl.name="hello";
41
42   task->cl = &cl;
43   task->cl_arg = &params;
44   task->cl_arg_size = sizeof(params);
45
46   task->callback_func = callback_func;
47   task->callback_arg = (void*) (uintptr_t) 0x42;
48   task->synchronous = 1;
49
50   ret = starpu_task_submit(task);
51   if (ret == -ENODEV) goto enodev;
52   STARPU_CHECK_RETURN_VALUE(ret, "starpu_task_submit");
53
54   starpu_shutdown();
55
56   return 0;
57
58 enodev:
59   starpu_shutdown();
60   return 77;
61 }
```

Python. We explore several of these programming approaches in this section.

4.4.1 OpenACC. OpenACC (originally short for Open Accelerators) is one of this first high-level GPGPU programming approaches that still supports a significant user base today (as of 2020). OpenACC was first released in 2012 as a collaboration between Cray, NVIDIA, and the Portland Group in order to support the users of ORNL's Titan, one of the first large heterogeneous supercomputers. As previously mentioned, Titan was a Cray machine with Nvidia devices. The Portland Group was involved because, as mentioned in the previous section, OpenACC was inspired by the high-level directive approach used in in the PGI-Accelerator model, and the first OpenACC compiler provided by PGI was developed as an extension to the PGI-Accelerator compiler.

The dream of OpenACC was to create an open, directive-based standard for GPU-computing as an analog and counterpart to the then de facto standard for parallel processing on multi-core CPUs, OpenMP. In the same way that a small number of OpenMP pragmas can be used to parallelize an existing application, OpenACC intended to provide a minimal set of directives that application developers could apply to accelerate an existing CPU-based scientific application on a GPU. This contrasted with the existing lower-level programming approaches like CUDA and OpenCL, which required a significant amount of code restructuring and rewriting for GPU acceleration.

The ideology of OpenACC is to allow users to expose and identify parallelism in an application using descriptive directives, and to leave the more complicated task of mapping parallelism to GPU devices in the hands of the OpenACC compiler. This deviates from the OpenMP model, which traditionally employed a very moderated and prescriptive application of directives.

This high burden of effort tasked to OpenACC compilers in some ways has prevented OpenACC from reaching the popularity and monopoly status of its OpenMP analog. Although OpenACC is intended for general-purpose GPU computing across different vendors, for most of its history, the PGI OpenACC compiler has been the only available production-level option, and was restricted to Nvidia devices. Now, nearly a decade later, other vendors have more fully adopted the OpenACC standard and implemented more functional support. We discuss these compilers in more detail in Section 4.6.

An OpenACC annotated application typically contains a combination of data and compute directives centered around a computationally intense region of code or loop nest. In Listing 4, we see a small C program annotated with two OpenACC directives, a data directive (line 16) and a compute directive (line 19). Replicating this high-level programming approach in a low-level approach like CUDA or OpenCL could require significantly more code, several source files, and multiple compilations.

Listing 4: Example OpenACC C Application

```
1  int main() {
2
3    int SIZE = 1024;
4
5    float *a, *b;
6    a = malloc(sizeof(float) * SIZE);
7    b = malloc(sizeof(float) * SIZE);
8
9    for (int i = 0; i = < SIZE; ++i) {
10     a[i] = 0;
11     b[i] = // some initial value
12   }
13
14   // Data Directives
15   #pragma acc data copyin(b[0:SIZE]) copyout(a[0:SIZE])
16
17   // Compute Directive
18   #pragma acc parallel loop collapse(2)
19   for (int i = 1; i <= SIZE; i++)
20     for (int j = 1; j <= SIZE; j++)
21       a[i][j] = (b[i − 1][j] + b[i + 1][j] + b[i][j − 1] + b[i][j + 1]) / 4.0f;
22 }
```

4.4.2 OpenMP. OpenMP reigned as the de facto standard for directive-based homogeneous multi-core CPU computing through the early 2000s, at least in the scientific computing domain. As the demand for high-level programming approaches for GPGPU computing increased in the early 2010s, there was a push for OpenMP to support accelerator-based heterogeneous computing in addition to the homogeneous multi-core computing. Although the previously-mentioned OpenACC was developed to address this demand, motivation for OpenMP prevailed for several reasons:

(1) OpenACC and OpenACC compilers have been too-tightly bundled to Nvidia devices, especially since PGI (the primary OpenACC compiler) was acquired by Nvidia in 2013.
(2) Most high-performance-oriented scientific programmers were already familiar with basic OpenMP directives and OpenMP programming styles.
(3) Many scientific applications already employed OpenMP for homogeneous CPU-based computing, lightening the burden of developing a new accelerator-based implementation.

As a result, in 2013, a year after the launch of OpenACC, the OpenMP standards committee released OpenMP 4.0, which included new directives for offloading to GPU accelerators. In 2018, the standards committee released OpenMP 5.0, which expanded support for accelerators and included additional directives for tasking and auto-parallelism. Even before the official inclusion of offloading directives in OpenMP, several research-oriented compilers had been prototyping support for GPU offloading for OpenMP, which we discuss further in Section 4.6.

Initially in their development, OpenACC and OpenMP contrasted in the programming approach philosophy. As mentioned, OpenACC employed a more descriptive approach, where users expose parallelism and compilers map that parallelism to devices. In OpenMP, the directives supplied by users are taken more literally and prescriptively, in that the user directly controls how the parallelism is mapped to a device. However, the two standards have recently become more aligned due to the *loop* directive introduced in OpenMP 5.0, which mimics the behavior of the descriptive OpenACC directives. The relationship between OpenMP and OpenACC has been somewhat contentious at times. However, both standards are still currently being maintained as a high-level programming approach for heterogeneous computing.

Although OpenACC has been limited due to its ties to Nvidia devices, the availability of the production-level PGI OpenACC compiler throughout its history has certainly been an advantage. In

Listing 5: Example OpenMPAMP Accelerated Massive Parallelism 2012 in MicrosoftâĂŹs Visual Studio Not tied to Nvidia GPUs C Application

```
1
2  int main() {
3
4    int SIZE = 1024;
5
6    float *a, *b;
7    a = malloc(sizeof(float) * SIZE);
8    b = malloc(sizeof(float) * SIZE);
9
10   for (int i = 0; i = < SIZE; ++i) {
11     a[i] = 0;
12     b[i] = // some initial value
13   }
14
15   // Data Directives
16   #pragma omp target data map(to:b[0:SIZE], from:a[0:SIZE])
17
18   // Compute Directive
19   #pragma omp teams parallel for collapse(2)
20   for (int i = 1; i <= SIZE; i++)
21     for (int j = 1; j <= SIZE; j++)
22       a[i][j] = (b[i − 1][j] + b[i + 1][j] + b[i][j − 1] + b[i][j + 1]) / 4.0f;
23 }
```

contrast, although OpenMP 4.0 originally was approved in 2013, compilers fully supporting the standard have been slow in coming. Only very recently have mature compilers successfully supported the entire standard, and many mainstream compilers are still under development for the OpenMP 4.0 standard and especially the OpenMP 5.0 updates. We discuss this further in Section 4.6.

In Listing 5, we show the same application as the previous listing, now annotated with OpenMP directives. Although this short example trivially highlights the use of OpenMP, this approach still greatly simplifies heterogeneous computing compared to CUDA and OpenCL.

4.4.3 Kokkos. In 2012, around the same time as the release of OpenACC and OpenMP 4.0, H.C. Edwards and a team at Sandia National Laboratory developed the Kokkos portability layer [97–99].

Kokkos is implemented as a performance portability layer. Unlike OpenACC and OpenMP that rely on directives, Kokkos is implemented as a C++ template library on top of OpenMP, CUDA, HPX [160] (discussed below), or Pthreads [214]. Essentially, the goal is to allow programmers to implement the Kokkos abstraction layer once in their application, which can then be executed across a diversity of hardware architectures. The C++ templating abstraction is an attractive model for heterogeneous programming, as it allows the same API calls to have multiple backend implementations.

Kokkos has been a popular option within the scientific community, and is supported by several national labs, including Sandia and Argonne National Laboratories. Listing 6 demonstrates an example Kokkos application. The Kokkos abstractions do require more in-depth knowledge of C++ including concepts like templates and functors, compared to the directive-based approaches.

4.4.4 Raja. Like Kokkos, Raja is a C++-based GPGPU programming approached developed by a major US National Laboratory, Lawrence Livermore National Lab (LLNL) [38, 146]. Raja was first released in 2014, shortly after Kokkos, OpenACC, and OpenMP 4.0. Raja is essentially another collection of C++ abstractions intended to provide architecture portability for HPC systems, specifically those with GPGPU architectures.

Listing 6: Example Kokkos C++ Application

```
1
2  #include <Kokkos_Core.hpp>
3  #include <cstdio>
4  #include <typeinfo>
5
6  struct hello_world {
7    KOKKOS_INLINE_FUNCTION
8    void operator()(const int i) const { printf("Hello␣from␣i␣=␣%i\n", i); }
9  };
10
11 int main(int argc, char* argv[]) {
12   Kokkos::initialize(argc, argv);
13
14   printf("Hello␣World␣on␣Kokkos␣execution␣space
15 ␣␣%s\n", typeid(Kokkos::DefaultExecutionSpace).name());
16
17   Kokkos::parallel_for("HelloWorld", 15, hello_world());
18
19   Kokkos::finalize();
20 }
```

A 2015 Supercomputing poster compared Raja and Kokkos using the TeaLeaf application [209]. While Kokkos relied on the C++ template metaprogramming approach, Raja instead relies on the C++11 lambda features. They also found that porting an application to Raja was relatively intuitive, on a similar level to an OpenMP port. Conversely, porting the application to Kokkos required extensive architectural changes. Like Kokkos, Raja relies on OpenMP and CUDA internally to target CPUs and GPUs, respectively.

Listing 7 highlights a vector addition example in Raja, pulled from the Raja github (https://github.com/LLNL/RAJA/). Although Raja does require manual memory management with specialized allocators and deallocators, the actual loop parallelization requires very little code modification.

4.4.5 Alpaka. A newer project, Alpaka [210, 296, 303], was first released in 2015 and first published in 2016. Like Kokkos, Alpaka aims to provide a performance portability layer that allows programmers to write a single application for accelerators from several different vendors with minimal code changes. However, unlike Kokkos, Alpaka also abstracts the data structures used in kernel arguments, allowing for further optimizability. Internally, Alpaka relies on several back-end languages and libraries, including OpenMP 4+, OpenACC (experimental), TBB, CUDA, and HIP. Alpaka also relies on gcc and clang for back-end compilation. Although Alpaka is a newer project with less adoption that Kokkos or Raja, Alpaka is very well documented, both on GitHub [8] and their web reference [7], which are strong indicators of continued success, higher adoption, and project longevity.

4.4.6 C++ Libraries and Extensions. While Raja and Kokkos are two of the most popular C++-based high-level GPGPU programming approaches, especially in scientific computing, several other C++ libraries and extensions have been developed to support heterogeneous computation. We briefly discuss several of these in this section.

Listing 7: Example Raja C++ Application

```
#include <cstdlib>
#include <cstring>
#include <iostream>

#include "memoryManager.hpp"
#include "RAJA/RAJA.hpp"

const int CUDA_BLOCK_SIZE = 256;

int main(int RAJA_UNUSED_ARG(argc), char **RAJA_UNUSED_ARG(argv[]))
{
  const int N = 1000000;

  int *a = memoryManager::allocate<int>(N);
  int *b = memoryManager::allocate<int>(N);
  int *c = memoryManager::allocate<int>(N);

  for (int i = 0; i < N; ++i) {
    a[i] = -i;
    b[i] = i;
  }

  // OpenMP CPU target
  RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
    c[i] = a[i] + b[i];
  });
  memoryManager::deallocate(a);
  memoryManager::deallocate(b);
  memoryManager::deallocate(c);

  // CUDA GPU target
  RAJA::forall<RAJA::cuda_exec<CUDA_BLOCK_SIZE>>(RAJA::RangeSegment(0, N),
    [=] RAJA_DEVICE (int i) {
    c[i] = a[i] + b[i];
  });

  memoryManager::deallocate_gpu(d_a);
  memoryManager::deallocate_gpu(d_b);
  memoryManager::deallocate_gpu(d_c);

  return 0;
}
```

- **AMP** C++ AMP (Accelerated Massive Parallelism) was originally released in 2012 as part of Microsoft's Visual Studio 12 [124]. Interestingly, AMP was one of the few early GPGPU programming approaches not tied to Nvidia devices. Although C++ AMP is now considered obsolete and is no longer supported by Microsoft, the ideas from the project inspired other more modern approaches like AMD's ROCm suite.
- **Boost.Compute** C++ Boost is a collection of C++ libraries with various functionalities, typically related to performance improvements [261]. Although Boost libraries are licensed, maintained, and well documented, they aren't part of the official standard, though many Boost libraries are eventually absorbed into the official C++ standard. The Boost.Compute library was accepted as a Boost library in January 2015 and was available in Boost v1.61 released April 2016 [279]. Boost.compute is essentially a wrapper over OpenCL, with a high-level API created to resemble typical C++ STL library calls, and a low-level API that more closely resembles simple abstractions over OpenCL boiler-plate code, effectively merging OpenCL into C++. Although the Boost.compute github, mailing list, and issue tracker saw a flurry of activity in 2015 and 2016, there does not seem to be a significant amount of activity and development currently as of 2020.
- **Thrust** In the same way that Boost.compute could be considered a C++ OpenCL wrapper, C++ Thrust was developed as a C++ CUDA wrapper [42]. First published in 2012, Thrust is now an important part of the CUDA C++ toolkit and actively under development by Nvidia.
- **Bolt** C++ Bolt is yet another C++ template library with support for OpenCL [51]. Unlike the other libraries mentioned, Bolt specifically was developed to target AMD GPU devices. Bolt was developed and released by AMD in 2014, shortly after Nvidia's Thrust and Microsoft's AMP. Bolt optimizes code for AMD devices by generating AMD-specific API calls at the OpenCL level.
- **VexCL** While AMP, Boost.compute, Thrust, and Bolt attempt to abstract a lower-level backend (OpenCL, CUDA, AMD OpenCL), VexCL is a higher-level vector expression template library, although these distinctions may be lost on most scientific programmers [86]. VexCL creates and compiles a distinct OpenCL application, each with a single execution kernel, for each vector expression encountered in the original application.
- **C++** All of the other programming approaches in this section refer to libraries and extensions not incorporated in the C++ standard. However, newer versions of C++ have begun to incorporate different types of CPU parallelism directly into the standard. For example, C++17 has increased SIMD support for parallel loops. Furthermore, there is a push with the C++ community to add support for heterogeneous computing in future releases. The major drawback is the slow timeline for C++ releases and the significant burden of defending inclusions into the already massive C++ standard.

In addition to the libraries mentioned, several other C++ heterogeneous programming libraries have been developed, although few have a significant user base today. Some examples include CUB (CUDA Unbound) [212], SkelCL [275], and ArrayFire [208].

4.4.7 Distributed- and Accelerator-based Approaches. **Legion** The Legion Project [36, 189, 190] originates from Stanford University, and was first published in 2012. Legion, a portmanteau of logical regions, is unique from many of the other high-level approaches in this section in that it aims to support both distributed and accelerator-based heterogeneous computing.

Like many of the other frameworks, a main goal of Legion is to abstract or decouple the algorithm design from the mapping or execution on heterogeneous architectures. For Legion, this concept extends to distributed heterogeneous machines. Legion specifically focuses on data movement and management abstractions, primarily by introducing the abstraction of logical regions. By partitioning data into logical regions and sub-regions, programmers can indicate data locality and independence, which can be used by the underlying framework components to facilitate communication and parallelism.

Legion remains relevant today, and regular software releases address bugs, performance issues, features and extensions, and additional system support. Furthermore, the Legion project is supported and funded by the DOE Exascale Computing project [188].

Listing 8: Example Legion C++ Application

```
#include <cstdio>
#include "legion.h"

using namespace Legion;

enum TaskID {
  HELLO_WORLD_ID,
};

void hello_world_task(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx, Runtime *runtime) {
  printf("Hello World!\n");
}

int main(int argc, char **argv)
{
  Runtime::set_top_level_task_id(HELLO_WORLD_ID);

  {
    TaskVariantRegistrar registrar(HELLO_WORLD_ID, "hello_world_variant");
    registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
    Runtime::preregister_task_variant<hello_world_task>(registrar, "hello_world_task");
  }

  return Runtime::start(argc, argv);
}
```

In Listing 8 we see a small "Hello World" example programmed using Legion, sourced from the Legion github site [189]. Although this example hardly captures the essence of Legion, lacking even invocations of logical regions, it does give a small insight into the Legion API.

HPX HPX, short for High Performance ParalleX, is another distributed computing focused framework, developed by Louisiana State University and first published in 2014 [144, 160, 237, 238]. Like Legion, HPX aims to provide a unified programming approach, allowing both single-node and distributed parallelism from a single API. HPX is strongly connected to C++, and depends heavily on the Boost C++ libraries. Although HPX has traditionally focused on CPU-based distributed and single-node parallelization, more recently, efforts have been made to support heterogeneous computation with HPX, either through integration with OpenCL (HPXCL [89]), development of a SYCL backend [76], or other approaches.

Listing 9 shows an example "Hello World" application with HPX.

Listing 9: Example HPX C++ Application

```
hello_world_component.hpp
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include "hello_world_component.hpp"
#include <hpx/iostream.hpp>

#include <iostream>

namespace examples { namespace server {
    void hello_world::invoke()
        hpx::cout << "Hello HPX World!" << std::endl;
}}
HPX_REGISTER_COMPONENT_MODULE();
typedef hpx::components::component<
    examples::server::hello_world
> hello_world_type;
HPX_REGISTER_COMPONENT(hello_world_type, hello_world);
HPX_REGISTER_ACTION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action);
#endif

// hello_world_component.cpp
#pragma once
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include <hpx/hpx.hpp>
#include <hpx/include/actions.hpp>
#include <hpx/include/lcos.hpp>
#include <hpx/include/components.hpp>
#include <hpx/serialization.hpp>

#include <utility>
namespace examples { namespace server {
    struct HPX_COMPONENT_EXPORT hello_world
        : hpx::components::component_base<hello_world>
    { void invoke(); HPX_DEFINE_COMPONENT_ACTION(hello_world, invoke);};
}}

HPX_REGISTER_ACTION_DECLARATION(
    examples::server::hello_world::invoke_action, hello_world_invoke_action);
namespace examples {
    struct hello_world
      : hpx::components::client_base<hello_world, server::hello_world> {
        typedef hpx::components::client_base<hello_world, server::hello_world>
            base_type;

        hello_world(hpx::future<hpx::naming::id_type> && f)
          : base_type(std::move(f))  {}
        hello_world(hpx::naming::id_type && f)
          : base_type(std::move(f))  {}
        void invoke()
        { hpx::async<server::hello_world::invoke_action>(this->get_id()).get(); }
    }
}
#endif

// hello_world_client.cpp
#include <hpx/config.hpp>
#if !defined(HPX_COMPUTE_DEVICE_CODE)
#include "hello_world_component.hpp"
#include <hpx/hpx_main.hpp>

int main(int argc, char* argv[]) {
    // Create a single instance of the component on this locality.
    examples::hello_world client =
        hpx::new_<examples::hello_world>(hpx::find_here());
    // Invoke the component's action, which will print "Hello World!".
    client.invoke();
    return 0;
}
#endif
```

4.4.8 *Python and Java.*

Traditionally C, C++, and FORTRAN have dominated the high-performance computing field. However, more recently newer languages have experienced a rise in popularity in the HPC field, especially those that are heavily used in domain sciences (i.e., python and R). While most high-level heterogeneous programming approaches, especially within the scientific community, still target C and C++, there are some approaches that have been developed to target python and Java.

PyCUDA [167, 169] was first published in 2009, several years earlier than many of the other high-level approaches discussed in this section. However, PyCUDA is less of a high-level language, and more of a simple set of CUDA-wrappers in python. For example, in Listing 10 sourced from the PyCUDA author's web page (https://documen.tician.de/pycuda/), we see that the computation kernel still very much uses CUDA syntax, but the host code uses a simplified python API. Judging from the documentation, version updates, and paper citations, PyCUDA still has a significant user-base today.

Listing 10: Example PyCUDA Python Application

```
1  import pycuda.autoinit
2  import pycuda.driver as drv
3  import numpy
4
5  from pycuda.compiler import SourceModule
6  mod = SourceModule("""
7  __global__ void multiply_them(float *dest, float *a, float *b)
8  {
9    const int i = threadIdx.x;
10   dest[i] = a[i] * b[i];
11 }
12 """)
13
14 multiply_them = mod.get_function("multiply_them")
15
16 a = numpy.random.randn(400).astype(numpy.float32)
17 b = numpy.random.randn(400).astype(numpy.float32)
18
19 dest = numpy.zeros_like(a)
20 multiply_them(
21       drv.Out(dest), drv.In(a), drv.In(b),
22       block=(400,1,1), grid=(1,1))
23
24 print(dest−a*b)
```

Listing 11: Example PyOpenCL Python Application

```
1  import numpy as np
2  import pyopencl as cl
3
4  a_np = np.random.rand(50000).astype(np.float32)
5  b_np = np.random.rand(50000).astype(np.float32)
6
7  ctx = cl.create_some_context()
8  queue = cl.CommandQueue(ctx)
9
10 mf = cl.mem_flags
11 a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
12 b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)
13
14 prg = cl.Program(ctx, """
15 __kernel void sum(
16      __global const float *a_g, __global const float *b_g, __global float *res_g)
17 {
18   int gid = get_global_id(0);
19   res_g[gid] = a_g[gid] + b_g[gid];
20 }
21 """).build()
22
23 res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
24 prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)
25
26 res_np = np.empty_like(a_np)
27 cl.enqueue_copy(queue, res_np, res_g)
```

The OpenCL analog to PyCUDA, PyOpenCL, was developed by the same research team [168] in 2012, three years after the release of PyCUDA. In Listing 11, we see that PyOpenCL provides a similar wrapper-style abstraction over OpenCL, although with slightly more OpenCL-oriented python API calls. The Clyther project [255] was another interesting but short-lived OpenCL-based python interface for heterogeneous computing.

Similar to how PyCUDA provided a way to develop GPU-accelerated applications in python, JCUDA [300] provided CUDA wrappers for Java. Also developed in 2009, JCUDA is still frequently updated and supports a significant user base. Because the abstraction level of Java is more similar to C, JCUDA's abstraction level is very similar to that of C-based CUDA. As a result, this approach does not strictly qualify as a "high-level" approach in the same manner as the other approaches in this section.

The options for performing general purpose accelerated computing in languages like python are still technically limited. However, as we see in the following section, python has a rich environment and a multitude of programming approaches for heterogeneous domain specific computing that more appropriately map to the abstraction level and programming style of python.

4.4.9 SYCL, DPC++, and OneAPI. The SYCL standard is yet another C++-based heterogeneous programming approach [164]. First released in 2014, SYCL originally aimed to be a programmer-productivity oriented abstraction layer on top of OpenCL. However, later implementations targeted other intermediate representations, like AMD HIP and CUDA. We discuss this further in Section 4.6. Although SYCL is several years old, it has seen limited uptake in the scientific community, until its recent involvement with DPC++ and Intel's OneAPI initiative.

DPC++ [29], launched in 2019, is a SYCL implementation developed and managed by Intel, that integrates the SYCL and OpenCL standards with additional extensions. These extensions are often championed for inclusion in the SYCL standard itself, analogous to how several of the heterogeneous and parallelism features of SYCL are then pushed for inclusion into the C++ standard. Examples of features in SYCL that originated in DPC++ include unified shared memory, group algorithms, and sub-groups.

Intel's OneAPI Library [151] attempts to encapsulate several of the technologies and programming approaches discussed in the section under a single umbrella. OneAPI consists of several APIs based on DPC++, SYCL, C++ Parallel STL, and Boost.Compute, including:

- oneAPI DPC++ Library
- oneAPI Math Kernel Library
- oneAPI Data Analytics Library
- oneAPI Threading Building Blocks
- oneAPI Video Processing Library
- Collective Communications Library
- oneAPI DNN Library
- Integrated Performance Primitives

It remains to be seen if OneAPI can truly deliver a unified programming approach for a diverse set of heterogeneous systems.

4.5 Domain Specific GPU Programming Approaches

Both the high-level and low-level general-purpose GPU programming approaches allowed developers to create heterogeneous applications for a huge diversity of application domains. However, many domain and computational scientists spend the entirety of their programming efforts within a very specific field or area. To combat the issues with the general purpose approaches, such as the complexity of the low-level approaches and inconsistency and performance issues with the high-level approaches, a multitude of domain-specific GPU programming approaches were developed.

More specifically, libraries or domain-specific languages (DSLs) targeting a single application space or area were developed to meet

the very specific needs of a smaller user-base. We discuss several of these approaches across a number of domains in this section.

4.5.1 Linear Algebra and Math Libraries. Linear and matrix algebra algorithms have consistently been some of the most important but also computationally demanding components of scientific computing. It is no surprise then, that several heterogeneous libraries and frameworks have been developed specifically for this domain.

cuBLAS: Possibly the most universal heterogeneous programming library, cuBLAS is a linear and matrix algebra library developed and maintained by Nvidia for CUDA-enabled GPUs [222]. Although cuBLAS greatly simplifies development of CUDA kernels containing BLAS-like algorithms, because of its closed-source and proprietary nature, the internal functionality can be difficult to debug. Still, cuBLAS has hundreds of citations just in 2020, over a decade after its initial release. An OpenCL alternative, clBLAS [16, 231], was developed in 2013 and is maintained by AMD, although its adoption has been much less widespread than cuBLAS.

cuSparse A separate library for sparse matrix computations, cuSparse [216], was introduced around the same time as cuBLAS. Developed and maintained by Nvidia, cuSparse is specialized for sparse matrix computations and sparse matrix storage and memory formats on the GPU. Like cuBLAS, cuSparse is limited to CUDA GPUs, but is still commonly used as of 2020.

MAGMA A major open-source alternative to cuBLAS, MAGMA [285, 286] (Matrix Algebra for GPU and Multicore Architectures), was developed at the University of Tennessee in 2009. MAGMA was designed to be similar to LAPACK [18] (also developed at the University of Tennessee), and MAGMA integrated with LAPACK APIs and data structures. Although MAGMA has been less popular overall than cuBLAS, even as of 2020 MAGMA has been shown to outperform cuBLAS on certain applications [120]. Also, current releases of MAGMA support both dense and sparse matrix operations.

ViennaCL Another open-source alternative, ViennaCL [256, 257], was first released in 2010. ViennaCL primarily focused on common sparse and dense linear algebra operations, and initially only had support for an OpenCL backend, although OpenMP and CUDA backends were later added. Like the most of the other frameworks in this section, ViennaCL could be used to target CPUs, GPUs, and MIC devices. Overall ViennaCL was not hugely popular compared to cuBLAS and MAGMA, but is still referenced in 2020.

Eigen The Eigen C++ library [101] specialized in matrices, vectors, numerical solvers, and related algorithms. Although Eigen was originally designed for CPU-based linear algebra, GPU-support was later added, making it an attractive option for heterogeneous computing as well.

ARPACK The ARPACK library [23, 191] is a Fortran77 based collection of subroutines, targeted toward large-scale eigenvalue problems. The ARPACK developers also released C++ versions of the routines, ARPACK++ [24]. In the context of this project, some effort has been made to extend ARPACK to support GPUs [22], although ARPACK is less oriented toward heterogeneity than other works in this project, and primarily included for completeness. Recently ARPACK development has shifted to an open-source project, ARPACK-NG [25]. Although not as popular as many of the other math libraries, ARPACK is still used in 2020, and is an underling numerical computing tool for several frontends such as SciPy [26] and Mathematica [27].

cuFFT Besides BLAS routines, another common but computationally demanding operation in scientific routines is Fast Fourier Transforms (FFTs). In conjunction with the cuBLAS and cuSparse routines, Nvidia also released an FFT library, cuFFT [227]. For completeness, the other math libraries released by Nvidia include cuParse, cuRand, cuSolver, and cuTensor [220].

Odient Differential equations occur frequently in algorithms with a time dimension, for example development of ecological systems and population modeling. Odient [4, 195] was developed in 2011 as a C++-based GPU-enabled library for solving the initial value problems (IVPs) of ordinary differential equations (ODEs). Interestingly, Odient was used as recently as 2020 to model the spread of the Covid-19 virus [93, 138, 205, 258].

SPIRAL The SPIRAL project [250, 272] is a DSL and programming framework focused on the development of digital signal processing (DSP) algorithms. SPIRAL aims to simplify the programming approach for domain scientists, offering a syntax similar to typical formula and mathematical expressions. SPIRAL then internally applies performance expert knowledge and optimizations to generate high-performance code for specific architectures.

4.5.2 Image Processing and Graph Algorithms. The Halide programming language was developed in 2013 as a collaboration between MIT's CSAIL laboratory and Adobe [136, 137, 253]. At its core, Halide is a DSL targeted for image processing and graph algorithms.

Like many of the other programming approaches in this section, Halide is embedded in C++, with a dedicated Halide C++ API. More recently, Halide has also developed python bindings. Halide supports a wide array of architectures, including x86, ARM, PowerPC, and other CPU architectures and CUDA, OpenCL, OpenGL, and DirectX enabled GPUs. Halide is used internally in Adobe Photoshop, and in projects related to Google's Tensorflow.

4.5.3 Machine Learning. The explosion of machine learning, undoubtedly the fastest-growing field in computer science, has led to the development of several heterogeneous programming approaches targeted specifically toward the machine learning domain.

Again, in a trend we commonly see in accelerator-based heterogeneous programming approaches, Nvidia has significantly contributed to the machine learning domain with the development of their cuDNN library [71], first released in 2014. Although the cuDNN (CUDA Deep Neural Network) library can be programmed directly, similarly to cuBLAS, more typically cuDNN is used as a backend to one of the widely used deep learning front end frameworks. More recently, these frameworks are the primary tool for machine-learning science, as manually coding neural network algorithms can be difficult. The specific frameworks that support cuDNN include MxNet [67], Tensorflow [1], Keras [135], Pytorch [239], Chainer [284], and Caffe [155].

Analogous to clBLAS, AMD has also developed an OpenCL-based analog to cuDNN, named MIOpen [14, 165]. Recently released in 2019, MIOpen is provided as part of the ROCm suite, and based on a software stack including both OpenCL and HIP. Although MIOpen is currently not as popular as cuDNN, and lacks integration into the major front-end frameworks and tools, it could become popular

in the near future with new AMD systems like ORNL's Frontier supercomputer [234], projected release in 2021 with four AMD GPUs on each node.

TVM [68], a community driven project developed at the University of Washington, offers an open source option for end-to-end deep neural network computation. Now managed by the Apache Software Foundation (ASF), TVM has several advantages over the other major machine-learning frameworks in that it is not tied or restricted to any vendor or device family. TVM supports multiple front-end approaches, including Tensorflow, Keras, and MxNet, and is able to target a diverse set intermediate representations, including CUDA, OpenCL, Rocm, and others. Additionally, TVM has limited support for FPGA devices.

4.5.4 Scientific Visualization. A very natural domain for heterogeneous programming is scientific visualization. Visualization applications typically already heavily rely on GPU architectures for image and video rendering and display, typically through low-level APIs like OpenGL or OpenCV. Development of domain-specific heterogeneous programming approaches for scientific computing is a natural extension. One approach involves in-situ visualization, where the computation and visualization are tightly coupled, without requiring offloading to the host device.

VTK-m [213] is an example of a heterogeneous scientific visualization approach. Like many other approaches, VTK-m relies on C++ template metaprogramming. The VTK-m programming abstraction is based on "data-parallel primitives", high-level algorithmic API calls that are then executed on the accelerator device.

Another example is the Alpine framework [177], which builds on the VTK-m framework and ideas. Alpine is focused on supporting modern supercomputing architectures, a flyweight infrastructure, and interoperability with software like R and VTK-m. Alpine was designed to accelerate scientific visualization codes using Nvidia GPUs and Intel Xeon Phis.

The Ebb framework [45, 96] is a DSL designed specifically for collision detection, a common algorithmic operation in scientific visualization and computer graphics.

Finally, the Listz framework [88, 198], developed at Stanford University, is a DSL created specifically to construct 3D mesh PDE solvers, critical for many visualization algorithms and graphics applications.

4.5.5 Climate and Weather. Due to the high number of computational resources required to model climate and weather at scale, climate and weather simulations represent a large fraction of most HPC system workloads. Unsurprisingly, DSLs have also been created to ease the creation of climate-based HPC applications. One example, the CLAW project [72, 243] developed in 2018 at ETH Zurich, is a FOTRAN-based DSL that aims to provide performance portability for column- and point-wise weather and climate computations.

4.6 Heterogeneous Compilers

Development of new heterogeneous programming approaches, APIs, libraries, and frameworks is important for advancing the field of heterogeneous computing. However, even the world's best-designed programming approach is rendered useless without an effective implementation, typically in the form of a compiler. Much of the success of different programming approaches hinges on the availability and usability of compilers for said approaches. In this section, we discuss different tiers of compilers, from vendor supported production-level to academic, each of which plays a crucial role in the life cycle of heterogeneous programming approaches.

4.6.1 Vendor Compilers. We first discuss vendor compilers. These typically refer to a language implementation, in the form of a compiler, developed by a major accelerator manufacturer, such as Nvidia, AMD, IBM, Intel, etc. We briefly highlight some major advantages and disadvantages of the vendor compiler model for heterogeneous programming approaches, and then discuss several vendor compilers in detail.

Advantages: Compilers developed and maintained by hardware accelerator vendors are typically very consistent and reliable for a small set of supported devices. The documentation and user guides are often detailed, thorough, and updated. These companies are financially motivated for success with their devices, which results in many of the advantages listed. These compilers also have somewhat of a guarantee of longevity, at least compared to the independent and open source projects.

Disadvantages: Vendor compilers, for obvious reasons, are limited to only compile code for devices produced by the vendor. This leads to replication of efforts for each different manufacturer. Furthermore, vendor compilers often introduce extensions to otherwise portable programming approaches that optimize the performance for their specific devices. These extensions break the original language intentions, and result in code that is no longer portable across an array of different accelerators. The vendor compilers also typically have a slower release cycle, are slower to incorporate updates to programming approaches, and are more conservative for the implementation of new language feature release of updated language versions.

NVCC Arguably the most popular, and dominant, vendor compiler in all of heterogeneous computing is *nvcc*, Nvidia's core CUDA compiler [219]. Released in 2006 along with Nvidia's CUDA toolkit, *nvcc* is based on the LLVM compiler toolchain [179], which is discussed later in this section. The *nvcc* compiler is implemented as a compiler driver; *nvcc* invokes the needed tools to perform a given compilation. Typically in a C CUDA application, the host code is compiled with *gcc*, and the device code is compiled using *cudacc*. In this case, *nvcc* would invoke *gcc* and *cudacc*, generating a C-code host binary and PTX device code respectively. PTX, or NVPTX, is a low level instruction set architecture used by CUDA-enabled GPUs.

Although *nvcc* is strictly for CUDA applications and Nvidia devices, *nvcc* is used extensively as an internal mechanism in many of the higher level source-to-source compilers, open source frameworks, and heterogeneous programming toolkits.

PGI The PGI OpenACC compilers, *pgcc* and *pgft*, have been the de facto standard for OpenACC compilation since its inception in 2012 [293, 294]. The PGI (Portland Group Inc.) company was founded in 1989, and originally developed parallel computing compilers for x86 architectures. PGI especially specialized in high-performance FORTRAN compilers. Because of this specialization, in 2009 PGI was contracted by Nvidia for the development of the first FORTRAN-based CUDA compiler.

PGI also worked with Nvidia to develop the PGI-Accelerator programming model, which we discussed in Section 4.3. As mentioned, the PGI-Accelerator compiler was eventually extended to develop the first OpenACC compiler. According to the definitions and classifications used in this section, PGI's role in heterogeneous programming and computing up to that point would have placed it exactly in the following "independent compilers" section. However, in 2013 PGI was acquired by Nvidia, redefining it as a "vendor compiler", at least for the purposes of this project. Interestingly, in 2013 PGI also developed an OpenCL compiler for ARM cores [133], but this was removed after the Nvidia acquisition.

Since then, PGI has continued to develop compilers for Nvidia devices for OpenACC C and OpenACC FORTRAN, and has been very involved in the promotion and development of OpenACC itself. Although PGI compilers have existed independently from the CUDA toolkit in the past, as of August 2020 *pgcc* and *pgft* have now been fully absorbed into Nvidia, and are now re-branded as part of the Nvidia HPC SDK [132, 224].

AMD The other major GPU manufacturer after Nvidia, at least in the context of scientific computing, is AMD. Unlike Nvidia, AMD has not developed a proprietary programming approach and vendor compiler for heterogeneous computing. AMD has developed a C/C++ optimizing vendor compiler, *aocc*, for its CPU Ryzen devices [11], but for their Radeon GPU devices, AMD has opted for an open-source solution.

In order support its GPU architectures, AMD has developed the open-source ROCm (Radeon Open Compute) suite [12]. ROCm is a collection of APIs, drivers, and development tools that support heterogeneous execution on both AMD GPUs, but also other architectures like CUDA GPUs. ROCm supports the AMD HIP representation, but can also process OpenMP and OpenCL applications. The compilers, libraries, and debuggers for ROCm are available from the open source github [13].

Intel Intel has long been at the frontier of high-performance compilers for their optimizing and parallelizing CPU compilers, enabling SIMD and multi-threaded parallelism for their homogeneous Intel Xeon CPU devices. Intel's first foray into heterogeneous compilation came in 2010 with the introduction of the Intel Xeon Phi coprocessor chip [217]. These chips followed a similar offload model and architecture as the contemporary GPU models.

Intel's acquisition of the FPGA-manufacturer Altera has also resulted in the release of a vendor-specific Intel-based OpenCL compiler for FPGAs [150]. However, this compiler framework suffers from many of the vendor-specific extensions and optimizations mentioned in the above "disadvantages" discussion, rendering the resulting OpenCL not portable to other devices. We discuss this further in Section 4.8.

Finally, with the release of the Xe GPGPU, Intel is also expected to release an Intel-based GPU-specific vendor compiler [152]. Few details have been released on the heterogeneous programming approach for the X^e accelerator, but it is expected to involve Intel's OneAPI and DPC++ frameworks, discussed in more detail in Section 4.4.

Cray Although Cray does not manufacture heterogeneous accelerators, Cray has been responsible for building several of the world's fastest heterogeneous supercomputers, including ORNL's Jaguar, Titan, and Frontier. For these machines, Cray has developed the "Cray Compiling Environment" [78]. For GPGPU offloading, *cce* supports both OpenACC and OpenMP 4.5 directives.

IBM Like Cray, IBM does not directly develop heterogeneous accelerators, although IBM does develop multicore chips, the Power processors (Power 1, ..., Power8, Power9, Power10, etc.) [148]. However, like Cray, IBM has also built top heterogeneous supercomputers, including ORNL's Summit and LLNL's Sierra machines. In turn, IBM's *xlc* compiler toolchain has limited support for heterogeneous programming approaches [147]. Specifically, *xlc* supports the OpenMP offloading directives in their C and C++ applications, though no support for OpenACC is available.

4.6.2 Independent Compilers. Several independent companies develop and maintain proprietary compiler technologies, a few of those specifically targeting heterogeneous computing. Before its acquisition by Nvidia, *pgcc/pgc++* from The Portland Group represented a prime example of an independent heterogeneous compiler.

Another example is the ComputeCPP compiler from Codeplay Software [74]. The ComputeCPP compiler is an implementation of the SYCL standard, discussed in Section 4.4, and currently represents one of the few production-level SYCL implementations, although Intel is currently developing a SYCL implementation as part of its OneAPI framework. Several other independent compiler development companies exist, and this list is by no means exhaustive.

4.6.3 Open Source Compilers. The main alternative to heterogeneous proprietary vendor compilers are production-level open source heterogeneous compilers. These compilers and compiler toolchains are typically maintained by steering committees, which can consist of representatives from accelerator vendors, scientific institutions, and independent companies. We discuss the advantages, disadvantages, and some examples of open source heterogeneous compilers.

Advantages Unlike the vendor compilers, open source compilers are often more community driven. That is, the direction and implementation of the compiler is not completely motivated and driven by device manufactures, although device manufactures are often involved. Also, most of the open-source compiler frameworks support a variety of accelerators and architectures. More generally, open source compilers benefit from all of the same advantages of open-source software as a whole, including transparency, flexibility, and independence. Specific to heterogeneous programming approaches, open source compilers can more quickly adapt new standards and features and the rapidly evolving array of architectures. Also, because the same compiler can be used across several architectures, the input programming approach used is inherently more portable. Most open source projects are managed through git or subversion, and hosted on a popular git repository hosting site like Github.

Disadvantages Open source compiler projects, especially the smaller ones, may not have the financial security of the vendor compilers. They also may not have the secured longevity. For example, if the main contributors to an open source compiler projects change positions or careers, continued maintenance on the project may terminate. Also, the open source compilers may not have access to low level architecture details that the vendor compilers use to

get increased performance on their specific devices. However, the large open source compiler projects, like LLVM and GCC, typically have no issues with longevity and closely tail vendor compilers in terms of performance.

LLVM, Clang, and MLIR LLVM, originally an abbreviation for Low Level Virtual Machine, has become one of the most important compiler toolchains, not just in heterogeneous compilation, but in all of computing [178, 179, 248]. As previously mentioned, the LLVM backend intermediate representation and compilation tools form the backbone of many of the other compilers, including the vendor compilers like nvcc.

First developed in 2000 by Chris Latner at the University of Illinois at Urbana Champaign, LLVM has grow significantly from its initial role as a virtual machine processor. Originally designed for C/C++, LLVM now provides an internal representation and compile time, runtime, and idle time optimization for a multitude other languages. In 2005 Apple began to manage and maintain LLVM for use in their internal projects, but LLVM was later re-licensed under Apache.

LLVM exists as a main project, LLVM-core, and a number of sub-projects, including three specifically relevant to heterogeneous programming approaches, Clang, OpenMP, and MLIR.

First released in 2008, clang is LLVM's own front end compiler for C and C++ [178, 247]. The clang compiler processes C and C++ code and generates LLVM IR, which is then optimized and processed by LLVM. LLVM's OpenMP sub-project implements OpenMP functionality into the LLVM clang compiler. Through the clang and OpenMP sub-projects, LLVM supports heterogeneous computing by compiling C and C++ applications with OpenMP offloading directives.

Though not yet an official LLVM sub-project, OpenACC support is also being developed for LLVM as part of the Clacc (Clang OpenACC) project [87]. Clacc builds on the LLVM OpenMP infrastructure. Clacc accepts C-based OpenACC as input, internally translates to OpenMP, and then generates LLVM intermediate representation using the existing LLVM OpenMP infrastructure.

MLIR (multi-level intermediate representation) is another LLVM project with significant implications for heterogeneous programming [180, 249]. The MLIR project adopts a layered compilation and optimization model, with different MLIR layers, or dialects, that have distinct abstraction levels and areas of focused. These layers can be combined and lowered, from higher abstraction dialects to lower abstraction dialects. Essentially, MLIR offers a reusable abstraction toolbox. A main goal of MLIR is to prevent software fragmentation and improve support for heterogeneous hardware, as the concept of dialects maps well to the ideas of different accelerators. MLIR also aims to provide support for the development of domain-specific programming approaches, which has a straightforward mapping to MLIR dialects and the progressive conversion and lowering structure of MLIR. The previously discussed Tensorflow framework relies on MLIR, and has been a major motivation for the development of the project [281]. Additionally, the Flang project (a FORTRAN-based front-end for LLVM) and Flang's OpenACC support rely on MLIR [236].

GNU C/C++ The GNU Compiler Collection, commonly referred to as just GCC, is undoubtedly the longest-living and most widespread open source compiler framework [244] (although Perl is a close second on longevity). It is no surprise then that GCC also plays a role in heterogeneous compilation.

GCC was first released in 1987 as the GNU C Compiler, but has since expanded to incorporate other languages such as C++ and FORTRAN. More recently GCC has worked to develop support for OpenACC [245] and OpenMP offloading models [246]. However, GCC's implementations are not as mature as PGI's OpenACC implementation and LLVM's OpenMP implementation.

pocl The pocl project (Portable Compute Language) is an open source implementation of the OpenCL standard [154, 232], first released in 2015. The pocl project relies on Clang and LLVM internally, and is able to target most CPUs, Nvidia GPUs, and other HSA-supported GPUs.

TriSYCL The TriSYCL project is an open source implementation of the SYCL standard [288]. TriSYCL was originally managed by AMD, but is now managed by Xilinx, who have implemented several SYCL extensions and additions in order to more efficiently support SYCL on their FPGA architectures. Although this likens TriSYCL to the vendor compilers, TriSYCL has also been used to evaluate, verify, and provide feedback for the SYCL standard itself, which more closely aligns with the goals of open source projects.

4.6.4 Academic Compilers. The last category of heterogeneous compilers we cover are academic project compilers. These projects are typically source-to-source translation compilers, or pre-compilers, that build on or extend existing production-level compiler projects. However, they play a crucial role in the development cycle of heterogeneous programming approaches. We briefly discuss the advantages and disadvantages of research-based compilers, and list a few notable examples.

Advantages Academic compilers are great for prototyping and experimentation of new language features. A production level compiler, either vendor or open source, may take months to push through new features and require several stages of approval. Conversely, an academic compiler is usually owned by a small group of researchers, and new features can be implemented and launched in a few days. Often, new language features are first evaluated in academic compiler settings, and only later re-implemented, or trickled down, into more production-setting compilers. Most academic compilers also host open source code on major code repositories.

Disadvantages Academic compilers often struggle with adoption and longevity. Because the projects are owned by a small number of people, small shifts in personnel can have disastrous effects on maintenance of a framework. Also, the compiler frameworks are typically funded by larger projects and grants, and therefore may be dependent on renewal of funding. Finally, because these compilers may be targeting a specific problem area for the research group, they often implement only a subset of the target programming language or approach.

ROSE The ROSE compiler framework is an open source, research based, source-to-source transformation compiler developed at LNLL [200, 252]. First published in 1999, ROSE has not suffered from longevity issues, and is still cited frequently in 2020. In 2013, ROSE was used in one of the first initial implementations and evaluations of the OpenMP offloading model, OpenMP specification 4.0 [194].

OpenUH The OpenUH project was managed by the HPCTools group at the University of Houston [30, 193]. OpenUH was based on the Open64 compiler framework [228], and was originally developed as an OpenMP and FORTRAN Coarray compiler. OpenUH did begin support for OpenMP offloading directives for heterogeneous programming, and experimental support for OpenACC on Nvidia and AMD GPUs, but as of 2020 the compiler framework does not seem to be under active development.

Omni The Omni compiler project is maintained and developed by researches at the University of Tsukuba and the RIKEN Center for Computational Science, both in Japan [75, 260]. First released in 1999, the Omni OpenMP compiler represented one of the first research-oriented implementations of the OpenMP standard. Over time, Omni has shifted to focus on cluster-based OpenMP computing. In 2010, an extension to the Omni project, XacalabeMP [181] integrated a PGAS-model distributed memory approach to OpenMP compilation. Also in 2010, the OMPCUDA project extended the Omni compiler to support compilation of OpenMP code for CUDA GPUs. Later in 2013, initial OpenACC support was added, shortly after the release of the OpenACC standard [280]. The next year, 2014, the XaclableMP and OpenACC extensions were combined to create the XalableACC extension [215], a PGAS-based heterogeneous distributed framework based on OpenACC.

The Omni compiler and its extensions are still under active development. In 2019 and 2020, extensions were made to include FPGA support [50, 291], although this support is still a work in progress.

OmpSs The OmpSs project, first published in 2011, aimed to support CUDA- and OpenCL-enabled GPUs with OpenMP input[94, 242]. OmpSs is developed and maintained by the Barcelona Supercomputing Center, BCS.

Because OmpSs pre-dated the OpenMP offloading directives, the developers created custom extensions to OpenMP for handling data, based on the StarSs framework [241]. OmpSs was evaluated and extended by a multitude of other works and projects [57, 58, 102], including one comparing OmpSs, OpenMPC, OpenACC, and OpenMP. OmpSs has also been explored for FPGA-based heterogeneous computing [52, 53, 240], which we discuss further in 4.8.

As is obvious from the numerous publications, OmpSs is still undergoing active development and still being used as part of the toolchain for a number of other projects.

OpenARC The OpenARC compiler framework, first published in 2014, is maintained and developed by Oak Ridge National Laboratory [187]. OpenARC is an extension of the OpenMPC framework [182], and like OpenMPC, is built on the Cetus compiler toolchain [83]. OpenARC was originally designed to be the first open source option for OpenACC compilation, acting as an source-to-source translator that consumes OpenACC C input and generates C and CUDA output. More recently, OpenARC has evolved to accept OpenMP offloading directives as additional inputs, and can generate OpenCL and AMD HIP as output sources, in addition to CUDA.

OpenARC also acts as the core framework for other heterogeneous programming projects. The Compass framework [185] relies on OpenARC to generate ASPEN performance models [271] of heterogeneous applications driven by user annotations and directives. The CCAMP project aims to create an inter-operable OpenMP and OpenACC framework [174, 175]. The Iris runtime library (citation pending), also integrated into OpenARC, is a work in progress that aims to allow multiple accelerators, even with different architectures, to collaborate together to execute a single application.

The OpenARC developers have also integrated support for FPGAs [173, 176, 183]. This is discussed in more detail in Section 4.8.

HipSYCL Like TriSYCL, the hipSYCL project is another open source implementation of the SYCL standard, although hipSYCL lacks the backing of a major vendor and is instead managed by Heidelberg University [9, 145]. HipSYCL implements a subset of the SYCL standard, and targets AMD GPUs, Nvidia GPUs, and OpenMP-enabled CPUs. Copied from the HipSYCL repository, in Figure 6, we see a breakdown of the different SYCL implementations, several of which were mentioned in this and previous sections.

HPVM HPVM (Heterogeneous Parallel Virtual Machine) [171, 206, 207] is a research project first published in 2018 originating from the University of Illinois at Urbana-Champaign. On the surface, HPVM is an extension to LLVM with direct support for heterogeneous computation, simplifying the intermediate representation that many of the LLVM-dependent heterogeneous programming approaches rely on.

The HPVM project aims to develop a uniform representation that can capture an array of different heterogeneous architectures, including GPUs, multi-core CPUs, FPGAs, and more. The main components of HPVM include: (1) a dataflow graph-based parallel program representation to capture task and data parallelism, (2) a heterogeneous compiler intermediate representation that supports optimizations commonly employed on GPU devices, like tiling and loop fusion, and (3) a heterogeneous virtual ISA supporting GPUs, SIMD vectorization, and multicore CPUs.

HPVM is implemented on top of the LLVM project, and aims to provide a valuable new asset, a heterogeneous-focused extension, to the LLVM community.

4.7 Heterogeneous Benchmark Suites

When evaluating heterogeneous programming approaches, typically performance is king. However, measurements of performance are relative, and difficult to compare across different projects, frameworks, or standards. The one control that makes performance comparisons possible are standard benchmarks. In this section, we review several different benchmark suites designed specifically for heterogeneous programming approaches.

First released in 2009, the Rodinia benchmark suite [64] is the oldest among the benchmark sets discussed in this section. Rodinia first released with CUDA and OpenMP versions of computational kernels from several different scientific domains. OpenCL kernels were added next, and after the release of the OpenACC standard, OpenACC versions of several of the kernels were included. The OpenMP kernels were updated to use some of the offloading directives, although they only annotated using directives specific to the Intel Xeon Phi devices, not general GPUs.

In 2010, ORNL released the SHOC (Salable Heterogeneous Computing) benchmark suite [82]. The SHOC benchmarks released with both CUDA and OpenCL versions of several kernels. Unlike Rodinia, SHOC was designed to test applications at scale, not just on a single node.

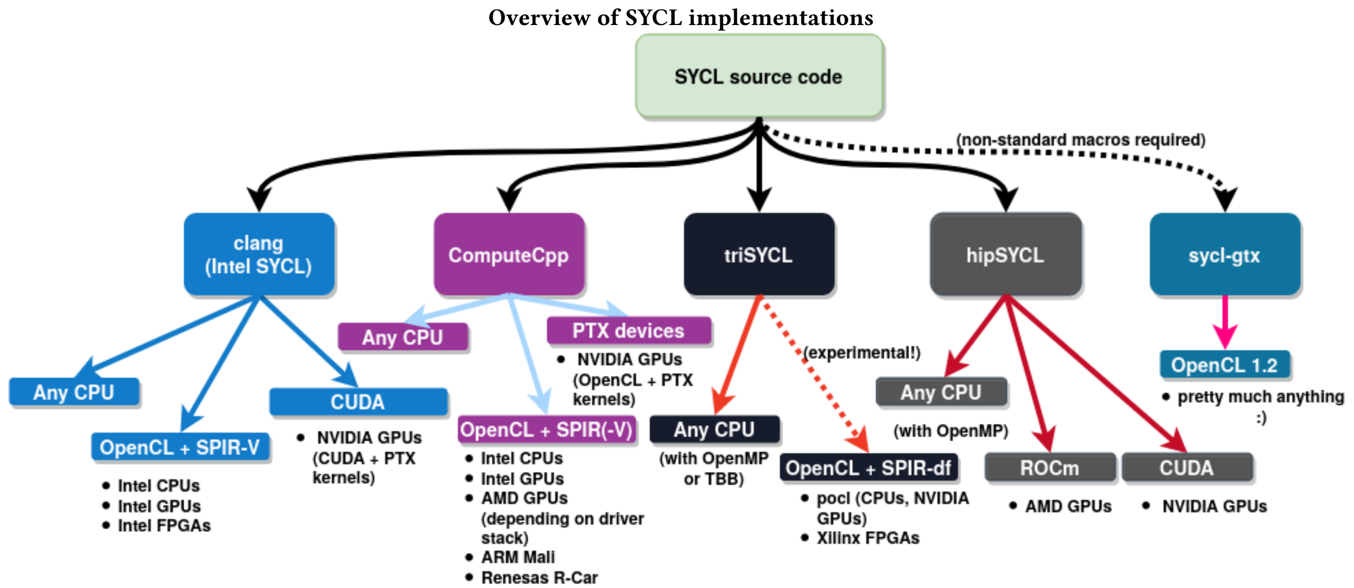

Figure 6: hipSYCL repository README [145]

The Parboil benchmarks [277] were developed by the University of Illinois at Urbana-Champaign and released in 2012. Like the other benchmark suites, Pairboil contains both CUDA and OpenCL code versions. One unique aspect with Parboil is that several different versions of each application are provided with different levels of optimizations. These versions can be used to measure the effectiveness of an automated optimizing compiler.

Also released in 2012, the OpenCL 13 Dwarfs benchmark suite [111] is a realization of Berkely's 13 computational dwarfs in OpenCL [28], where a dwarf is essentially a core computational or communication method or action.

In 2013, EPCC, the Edinburgh Parallel Computing Center, a supercomputing center associated with the University of Edinburgh, released a suite of OpenACC benchmarks [104, 105]. The suite contains low-level operations intended to test and measure the performance of hardware and compilers. The suite also contains a set of software kernels intended to replicate operations most commonly seen in scientific applications. Although the EPCC Benchmarks also contain OpenMP implementations, these versions are based on non-offloading OpenMP standards, 3.0 and earlier.

Finally, the SPEC Accel [158] benchmark suite was released in 2014. SPEC (Standard Performance Evaluation Corporation) is a non-profit specifically focused on developing and maintaining high-quality benchmarks. As a result, the SPEC Accel benchmarks are very well organized and documented, and have a robust set of scripts for executing and recording application information. However, the SPEC benchmarks are not open source, and require either a paid commercial license or a free academic licence.

Interestingly, the oldest benchmark suite, Rodinia, seems to be the most popular, with nearly an order of magnitude more citations than any of the other benchmark suites. This could be just an artifact of being released first, or from the Rodinia kernels more closely resembling desired scientific applications. However, the Rodinia benchmarks themselves are infrequently updated and fail to capture many of the new language features. This requires each research project that requires a benchmark set to develop their own updates to the benchmarks. The other benchmark suites face a similar challenge. Several newer benchmark suites have been presented, but all have faced issues with adoption. Moving forward, development, adoption, and maintenance of high-quality benchmark suites could significantly improve the productivity of heterogeneous programming approach developers.

4.8 FPGA Accelerators

Since the initial release of CUDA in 2006, GPGPUs have been the dominant driving force for accelerator-based heterogeneous computing. The concept of offloading computationally intense regions of code to a heterogeneous hardware accelerator has become commonplace in scientific computing, and for the past decade, heterogeneous computing has almost exclusively referred to GPGPU offloading. However, FPGAs have recently emerged as a major competitor to GPU accelerators, both in terms of computing power and power efficiency.

Field Programmable Gate Arrays (FPGAs) have been produced and developed for nearly 40 years. Altera, a major FPGA manufacturer, was founded in 1983, and released the first FPGA in 1984. Xilinx, the main competitor to Altera for several decades, was founded in 1984 and released their first FPGA in 1985. These devices have been pushed and promoted as potential architectures for high performance computing for decades, but until very recently, have not seen much adoption. The real revolution for FPGAs, and their adoption as a heterogeneous accelerator, has stemmed from the introduction of new FPGA programming approaches.

Traditionally, FPGAs were programmed using low-level Hardware Definition Languages (HDLs). These programming approaches

required detailed knowledge of FPGA hardware, manual management of timing and placing of wires on the FPGA board, and tedious implementation of flip flops, arithmetic logic, and other similar computer architecture-type features. Several high-level options, or High Level Synthesis (HLS) tools were developed, for example Handle-C, which was released in 1996 and generated HDL from C input [31]. However, many of these early HLS projects were more research oriented, and never had a wide adoption in the scientific community.

The first release of a viable HLS option for scientific programmers came with Altera's release of their OpenCL SDK, in 2013, around four years after the release of OpenCL itself [10]. Due to the widespread adoption of GPU offloading, by 2013 the idea of offloading to other accelerators such as FPGAs had become more palatable to the scientific community. The Altera OpenCL SDK was well documented, backed by a major vendor, and contained a suite of compilation, debugging, and performance measurement tools. Sensing the opportunity, Intel acquired Altera in 2015, and rebranded the SDK as the Intel FPGA SDK for OpenCL [150], which is still very active in 2020. Around the same time as Altera's developments, Xilinx also developed OpenCL SDKs and HLS tools, with their Vivado [299]and SDAccel [297]tools. These tools have been repackaged and recently released as the Vitis Software Platform [298].

Several research projects have sought to make the HLS programming approaches even more accessible for scientific programmers, typically by building software layers on top of the vendor-supported HLS backends. One example is the OpenARC 2016 extension, the OpenACC to FPGA framework [173, 176, 183, 184]. The OpenACC to FPGA framework accepts OpenACC C programs as input, and generates Intel-based OpenCL as output, creating optimized OpenCL from standard OpenACC directives. In similar a 2017 project, the team at Tskuba University extended the OmpSs framework to support FPGAs [52]. Like the OpenARC extension, the OmpSs extension relies on backend HLS tools, in this case the Xilinx HLS tools. In a later extension, OmpSs was also extended to support OpenACC [291].

In a slightly different approach, the ETH Zurich DaCe (Data-Centric) framework also supports FPGA deployment with a high-level programming approach [43]. DaCe is unique in that it employs a control-flow graph and GUI based interface as a frontend programming model, which is different than the current programming approach for nearly all scientific applications. However, this approach may map to FPGA architectures more appropriately than OpenACC or OpenMP, as it is reminiscent of the GUI-based HDL placing and routing programming style. Like OpenARC and OmpSs, Dace internally relies on the Xilinx HLS tools, with some support for the Intel tools through the hlslib extension [84].

4.9 Next Generation Accelerators

For upcoming and future systems, other types of accelerators besides GPUs and FPGAs are being explored as hardware accelerators. More and more exotic, customized, and specialized hardware accelerators are being explored as viable options in heterogeneous systems. In this section, we discuss some of these more recent accelerators, and the heterogeneous programming approaches associated with them.

4.9.1 Machine Learning Accelerators. Perhaps the most relevant and motivated new accelerator devices are machine learning accelerators. The interest in these types of devices closely followed the huge increase in activity in the area of machine learning. Due to the vast number of newly developed machine learning accelerators, this discussion is far from comprehensive, but we do discuss some of the most promising and widely available chips.

One of the first widely discussed chips specialized for machine learning was the Google Tensor Processing Unit (TPU) [73, 157]. Although Google began using the TPU devices internally in 2015, they were not publicly announced until 2017, and not publicly released until 2018. Google subsequently released the TPU1, which specialized on deep neural network inference, and the TPU2 and TPU3, both updates to the original TPU chip. Google also released a low-power alternative, the TPUEdge chip [287]. Interestingly, the TPUEdge device is shipped as part of a USB accelerator, a device not much larger than a typical flash drive. Machine learning models for the Google TPUs are built and programmed using Tensorflow, or in the case of TPUEdge, Tensorflow Lite, and compiled by Google's XLA compiler [5]. However, other XLA frontends are also supported on TPUs, including Julia [46], Jax, and Pytorch.

Nvidia has also invested heavily in machine learning-focused acceleration. The Nvidia Volta devices, the primary accelerator in ORNL's Summit supercomputer, contain 640 "Tensor Cores", specialized for machine learning workloads [226]. The Nvidia Ampere devices, the successor to the Volta, significantly improves on the Volta architecture and introduces double-precision tensor cores[221]. Nvidia has also released a packaged full system containing its core chips, the line of DGX workstations [223]. These workstations, including the DGX-1, DGX-2, DGX-V100, and DGX-A100, provide out-of-the box solutions for high performance and accelerated machine learning, for a significant cost. The initial price for the newest DGX A100 was $199,000. As a cheaper alternative, Nvidia has released a low-power, low-cost device for small scale and edge machine learning, the Nvidia Jetson and its successor, the Nvidia Xavier [225]. Not surprisingly, the Nvidia machine learning devices are still based on CUDA, and can be programmed with heterogeneous approaches that support CUDA backends, including Tensorflow and several of the other frameworks mentioned in this project.

Although Google and Nvidia are the most recognizable names in machine learning accelerators, many other major companies have also developed custom machine learning hardware. Amazon has developed a custom inferencing chip for its Amazon Web Server (AWS) EC2 instances, the AWS Inferentia chip [266]. Inferentia is programmed using the AWS Neuro SDK, which contains a compiler, a custom run-time, and profiling tools. The SDK supports many of the leading ML frameworks like Tensorflow, PyTorch, and MXNet. AMD has released a Radeon Instinct MI8 and a double precision update, the MI60 [15]. These devices are specifically built for deep learning and can be programmed using the AMD MIOpen, ROCm, and HIP toolchains (discussed in Section 4.4), which support many

of the popular ML frontends like Theano, Caffe, and Chainer. Interestingly, AMD has partnered with LLNL to deliver the El Capitan supercomputer in 2023, which will be deployed with Radeon Instinct devices [201]. Finally, ARM also has a hat in the ring with the ARM Ethos line of machine learning processors [20].

Besides the machine-learning oriented accelerators developed by major processor manufactures, there are also several chips developed by smaller, more research-based entities. Like the distinction between research-oriented and vendor compilers, research-oriented machine learning accelerators have more freedom to experiment and explore, both with the hardware and the programming approach. Some examples include the AI "IPU" (Inference Processing Chips) from Graphcore, which is a smaller company backed by larger entities like Dell, Samsung, and others [159]. The NeuFlow chip, developed as a collaboration between IBM, New York University, and Yale University is an older AI-oriented chip, first published in 2011 [110]. Stanford University has been responsible for developing two AI-oriented chips, the Energy Efficient Inference Engine (EIE) [139] and the TETRIS accelerator [121], focused on 3D stacking memory. Researchers at MIT CSAIL lab have developed the Eyeriss chip, an energy efficient inference processor [70]. Finally, scientists at the Institute of Computing Technology of the Chinese Academy of Sciences have developed the DianNao dataflow research chip [66], and several extensions, including DaDianNao [69], ShiDianNao [92], and PuDianNao [199]. These research processors all aim to allow developers to utilize the popular machine learning frontends, and really demonstrate the success of well-designed DSLs as a heterogeneous programming approach.

4.9.2 Neuromorphic Accelerators. Neuromorphic accelerators are being seriously explored by the scientific community. Starting in 2018, Oak Ridge National Lab has hosted an international conference, ICONS, on neuromorphic computing [149]. An early examples of prototype neuromorphic-style chips is the IBM TrueNorth System, first released in 2016 [6, 108]. As of 2019, Intel is also experimenting with neuromorphic chips, with its research chip "Loihi", and accompanying "Pohoiki" system [153]. Numerous other smaller companies are developing and researching neuromorphic chips, including BrainChip with the Akida processor [54] and General Vision with the NM500 chip [290].

4.9.3 Quantum Accelerators. Quantum computing, and quantum accelerators, have generated a significant amount of media attention and focus due to the revolutionary potential of a fully fledged quantum computer. Recently, quantum computing has been added as a core initiative for the US Department of Energy, with the funding of new centers to support the national quantum initiative. Although quantum computing as a viable accelerator in heterogeneous systems has yet to be realized, several companies are working to close this gap, including D-Wave [95] and IBM with the Q system [251].

4.9.4 Other Next Generation Accelerators. Finally, other even more novel accelerators are also being explored. The EMU Chick system from Emu Technologies implements a thread-migration based system with a novel architecture [103]. Xilinx and other FPGA vendors are exploring coarse-grained reconfigurable architectures (CGRAs), which can be thought of as a GPU-FPGA hybrid accelerator. Other novel acceleration ideas and chips are explored and published every year.

The next generation of accelerators will enable extreme heterogeneity for high-performance computing, but it will also create extreme challenges, both with the programming approaches and software stacks.

5 CONCLUSION

Since its first conceptualization with the PASM and TRAC machines in the early 80s, heterogeneous computing and heterogeneous programming approaches have shifted in and out of vogue. In Sections 2 and 3 we explored how distributed heterogeneous computing rose with the promise of robust diverse and distributed systems, supported by heterogeneous programming approaches and software stacks. We also saw how these systems were eventually eclipsed by homogeneous MPI-based supercomputers, homogeneous cloud servers, and CPU-chip advancements. In Section 4, we explored the rebirth of heterogeneous computing through accelerator-based computing, and the explosion of GPU-based computing and innovation with other more novel accelerators.

However, the challenges faced by distributed heterogeneous systems and contemporary accelerator-based systems are not so different. Many of the challenges early heterogeneous system developers faced are being constantly revived and re-imagined, especially in the face of the extreme heterogeneity of next generation systems. Many of the conceptual models, theoretical road maps, programming approaches, technical requirements and restrictions, and strategies for success from distributed heterogeneous research apply directly to accelerator-based systems. Figure 1, first published in 1995, would look right at home in a 2020 publication exploring extreme heterogeneity, albeit with improved graphics. Simply replace SIMD, MIMD, and vector processors, with CPUs, GPUs, FPGAs, ML accelerators, and neuromorphic chips.

One important balance in terms of heterogeneous programming approaches is the push and pull of generalized frameworks and specialized DSL frameworks. Successful high-level DSLs like Tensorflow have had incredible success, and made heterogeneous programming accessible to the large but specialized field of machine learning. Conversely, the generalized low-level CUDA and OpenCL APIs remain two of the most popular programming approaches for general accelerator programming, even 14 years after their initial release. This is especially impressive considering the huge number of research-based, open source, and even vendor-supported, high-level alternatives that have been developed over the past decade. However, the development of these high-level frameworks need to be funded at a level to readily intercept the developments of platforms and applications, otherwise implementations, and subsequently adoption, may lag significantly behind the development of standards. A prime example is OpenMP offloading, in which the standard was first released in 2013 but the functioning implementations were very recently released.

One immediate conclusion that could be drawn is that DSLs may provide an optimal high-level approach, and that these DSLs can be built using generalized low level approaches. However, this leaves stranded the programmer looking for a high-level heterogeneous

programming approach for an application outside the popular DSL frameworks.

Another significant takeaway is the huge success experienced by Nvidia in high performance computing for the scientific community. Developing a great framework is a challenge, but maintaining, updating, documenting, and marketing a great framework is what leads to widespread adoption and longevity. Also, the investments in training, library support, integration with other tools, and, for better or worse, aggressive acquisition of competition has solidified Nvidia's position in the heterogeneous computing landscape.

However, capitalistic development is not the only path to success. LLVM represents an open source project that nearly parallels Nvidia in terms of adoption and project integration. Part of LLVM's success can be attributed to the integrity and ingenuity of its initial design, and its early support and development from Apple. But LLVM has also been very successful at creating a community-driven environment, with very experienced contributors and a rigorous approval process for extensions and modifications. Both the successes of LLVM and Nvidia can be inspirations for creation of next-generation heterogeneous software stacks and programming approaches.

Finally, for all the computational heterogeneity discussed in this project, several other forms of heterogeneity are being explored in contemporary systems. A major example is heterogeneity not in the processor, but in the memory, with developments like non-volatile memory. These other types of heterogeneity, along with the constant development of novel accelerators, will continue to create challenges for the development of heterogeneous programming approaches.

Twenty-five years have passed since Siegel et al first laid their goals and challenges for heterogeneous computing. Despite the immense amount of progress in general and heterogeneous computing, most of these issues are still goals and challenges for today's accelerator-based systems. Twenty-five years from now, will we have a new landscape for heterogeneous computing and programming with these problems behind us, or will we still strive toward the same goals and face the same challenges?

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Norma E Abel, Paul P Budnik, David J Kuck, Yoichi Muraoka, Robert S Northcote, and Robert B Wilhelmson. 1969. TRANQUIL: a language for an array processing computer. In *Proceedings of the May 14-16, 1969, spring joint computer conference*. ACM, 57–73.

[3] David Abramson, Rok Sosic, Jonathan Giddy, and B Hall. 1995. Nimrod: a tool for performing parametrised simulations using distributed workstations. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*. IEEE, 112–121.

[4] Karsten Ahnert and Mario Mulansky. 2011. Odeint–solving ordinary differential equations in C++. In *AIP Conference Proceedings*, Vol. 1389. American Institute of Physics, 1586–1589.

[5] Google AI. 2020. Tensorflow XLA compiler. (2020). https://www.tensorflow.org/xla

[6] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. 2015. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems* 34, 10 (2015), 1537–1557.

[7] Alpaka. 2020. Alpaka Online Portal. (2020). https://alpaka.readthedocs.io/en/latest/index.html

[8] Alpaka. 2020. Alpaka Online Repostiory. (2020). https://github.com/alpaka-group/alpaka

[9] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*. 1–1.

[10] Altera. 2020. Altera OpenCL SDK. (2020). https://web.archive.org/web/20140109220211/http://newsroom.altera.com/press-releases/altera-opens-the-world-of-fpgas-to-software-programmers-with-broad-availability-of-sdk-and-off-the-shelf-boards-for-opencl.htm

[11] AMD. 2020. AMD ROCm AOCC Compiler. (2020). https://developer.amd.com/amd-aocc

[12] AMD. 2020. AMD ROCm Online Portal. (2020). https://www.amd.com/en/graphics/servers-solutions-rocm

[13] AMD. 2020. AMD ROCm Online Repository. (2020). https://github.com/RadeonOpenCompute/ROCm

[14] AMD. 2020. MIOpen - AMDâĂŹs Machine Intelligence Library. (2020). https://github.com/ROCmSoftwarePlatform/MIOpen

[15] AMD. 2020. Radeon Instinct Servers. (2020). https://www.amd.com/en/graphics/servers-radeon-instinct-mi

[16] AMD. 2020. ROCm Blas Library. (2020). https://rocmdocs.amd.com/en/latest/ROCm$_T$ools/clBLA.html

[17] AMD. 2020. ROCm Documentation Online. (2020). https://rocmdocs.amd.com/en/latest/

[18] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide*. SIAM.

[19] Thomas E Anderson, David E Culler, and David Patterson. 1995. A case for NOW (networks of workstations). *IEEE micro* 15, 1 (1995), 54–64.

[20] ARM. 2020. Arm Ethos-N series processors. (2020). https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-n

[21] Dorian Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Michelle Miller, and Sathish Vadhiyar. 2001. Users' guide to NetSolve V1. 4. (2001).

[22] ARPACK. 2020. ARPACK GPU Wrappers Online Repository. (2020). https://github.com/elezar/gpu-arpack

[23] ARPACK. 2020. ARPACK Online Portal. (2020). https://www.caam.rice.edu/software/ARPACK/

[24] ARPACK. 2020. ARPACK++ Online Portal. (2020). https://www.caam.rice.edu/software/ARPACK/arpack++.html

[25] ARPACK. 2020. ARPACK Online Repository. (2020). https://github.com/opencollab/arpack-ng

[26] ARPACK. 2020. ARPACK Scipy Web Reference. (2020). https://docs.scipy.org/doc/scipy/reference/tutorial/arpack.html

[27] ARPACK. 2020. ARPACK Wolfram Web Reference. (2020). https://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html

[28] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. The landscape of parallel computing research: A view from berkeley. (2006).

[29] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. 2020. Data Parallel C++ Enhancing SYCL Through Extensions for Productivity and Performance. In *Proceedings of the International Workshop on OpenCL*. 1–2.

[30] HPCTools at UH. 2020. OpenUH Compiler Online Repository. (2020). https://github.com/uhhpctools/openuh

[31] Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. 1996. Handel-C language reference guide. *Computing Laboratory. Oxford University, UK* (1996).

[32] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[33] Özalp Babaoğlu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. 1992. Paralex: An environment for parallel programming in distributed systems. In *Proceedings of the 6th international conference on Supercomputing*. 178–187.

[34] Mark Baker, Rajkumar Buyya, and Domenico Laforenza. 2000. The Grid: A Survey on Global Efforts in Grid Computing. *ACM Journal of Computing Surveys* (2000).

[35] Kevin J Barker, Kei Davis, Adolfy Hoisie, Darren J Kerbyson, Mike Lang, Scott Pakin, and Jose C Sancho. 2008. Entering the petaflop era: the architecture and performance of Roadrunner. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–11.

[36] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings*

of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 1–11.

[37] Micah Beck, Jack J Dongarra, Graham E Fagg, G Al Geist, Paul Gray, James Kohl, Mauro Migliardi, Keith Moore, Terry Moore, Philip Papadopoulous, et al. 1999. HARNESS: A next generation distributed virtual machine. *Future Generation Computer Systems* 15, 5-6 (1999), 571–582.

[38] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 71–81.

[39] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Keith Moore, and Vaidy Sunderam. 1993. PVM and HeNCE: Tools for heterogeneous network computing. In *Software for Parallel Computation*. Springer, 91–99.

[40] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. 1993. Visualization and debugging in a heterogeneous environment. *Computer* 26, 6 (1993), 88–95.

[41] Adam Beguelin, Jack J Dongarra, George Al Geist, Robert Manchek, and Keith Moore. 1994. HeNCE: A heterogeneous network computing environment. *Scientific Programming* 3, 1 (1994), 49–60.

[42] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[43] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[44] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, et al. 2003. Adaptive computing on the grid using AppLeS. *IEEE transactions on Parallel and Distributed Systems* 14, 4 (2003), 369–382.

[45] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–12.

[46] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

[47] Dimple Bhatia, Vanco Burzevski, Maja Camuseva, Geoffrey Fox, Wojtek Furmanski, and Girish Premchandran. 1997. WebFlow–a visual programming paradigm for Web/Java based coarse grain distributed computing. *Concurrency: Practice and Experience* 9, 6 (1997), 555–577.

[48] Buddy Bland. 2009. Jaguar: Powering and cooling the beast. (2009).

[49] Buddy Bland. 2012. Titan-early experience with the titan system at oak ridge national laboratory. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2189–2211.

[50] Taisuke Boku, Toshihiro Hanawa, Hitoshi Murai, Masahiro Nakao, Yohei Miki, Hideharu Amano, and Masayuki Umemura. 2019. GPU-accelerated language and communication support by FPGA. In *Advanced Software Technologies for Post-Peta Scale Computing*. Springer, 301–317.

[51] C++ Bolt. 2014. Template Library. *C++ template library for heterogeneous compute. AMD* (2014), 206.

[52] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. 2017. Exploiting parallelism on GPUs and FPGAs with OmpSs. In *Proceedings of ANDARE - the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. 1–5.

[53] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesus Labarta. 2018. Application acceleration on fpgas with ompss@ fpga. In *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 70–77.

[54] BrainChip. 2020. Akida Neural Processor System-on-Chip. (2020). https://brainchipinc.com/akida-neuromorphic-system-on-chip/

[55] Axel T Brünger, Paul D Adams, G Marius Clore, Warren L DeLano, Piet Gros, Ralf W Grosse-Kunstleve, J-S Jiang, John Kuszewski, Michael Nilges, Navraj S Pannu, et al. 1998. Crystallography & NMR system: A new software suite for macromolecular structure determination. *Acta Crystallographica Section D: Biological Crystallography* 54, 5 (1998), 905–921.

[56] Ian Buckle and Robert Reitherman. 2004. The consortium for the George E. Brown J. network for earthquake engineering simulation. In *13th World Conference on Earthquake Engineering*, Vol. 16. Vancouver: Venue West Conference Services Ltd.

[57] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. 2011. Productive cluster programming with OmpSs. In *European Conference on Parallel Processing*. Springer, 555–566.

[58] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. 2012. Productive programming of GPU clusters with OmpSs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 557–568.

[59] Ralph M Butler, Alan L Leveton, and Ewing L Lusk. 1993. p4-Linda: A portable implementation of Linda. In *[1993] Proceedings The 2nd International Symposium on High Performance Distributed Computing*. IEEE, 50–58.

[60] Ralph M Butler and Ewing L Lusk. 1994. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Comput.* 20, 4 (1994), 547–564.

[61] Nicholas Carriero, Eric Freeman, David Gelernter, and David Kaminsky. 1995. Adaptive parallelism and Piranha. *Computer* 28, 1 (1995), 40–49.

[62] Nicholas Carriero and David Gelernter. 1989. Linda in context. *Commun. ACM* 32, 4 (1989), 444–458.

[63] Nicholas Carriero, David Gelernter, and Timothy G Mattson. 1992. Linda in heterogeneous computing environments. In *Proceedings. Workshop on Heterogeneous Processing*. IEEE, 43–46.

[64] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.

[65] Song Chen, Mary Eshaghian, and Ashfaq Khokhar. 1993. A selection theory and methodology for heterogeneous supercomputing. (1993).

[66] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[67] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[68] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[69] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 609–622.

[70] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[71] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[72] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E Osuna, Robert Pincus, Jon Rood, and William Sawyer. 2018. The CLAW DSL: Abstractions for performance portable weather and climate models. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–10.

[73] Google Cloud. 2020. Google Cloud TPUs. (2020). https://cloud.google.com/tpu

[74] CodePlay. 2020. ComputeCPP Compiler Framework Online Portal. (2020). https://developer.codeplay.com/products/computecpp/ce/home/

[75] Omni Compiler. 2020. Omni Compiler Online Portal. (2020). https://omni-compiler.org/

[76] Marcin Copik and Hartmut Kaiser. 2017. Using sycl as an implementation framework for hpx. compute. In *Proceedings of the 5th International Workshop on OpenCL*. 1–7.

[77] Nvidia Corporation. 2007. NVIDIA CUDA compute unified device architecture programming guide. (2007).

[78] Cray. 2020. Cray Compiling Environment. (2020). https://pubs.cray.com/bundle/Cray-C-and-Cplusplus-Reference-Manual-86-S-2179-C-CPlusPlus-ditaval.xml/page/The-Cray-Compiling-Environment.html

[79] Yifeng Cui, Kim B Olsen, Thomas H Jordan, Kwangyoon Lee, Jun Zhou, Patrick Small, Daniel Roten, Geoffrey Ely, Dhabaleswar K Panda, Amit Chourasia, et al. 2010. Scalable earthquake simulation on petascale supercomputers. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–20.

[80] David E Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. 1997. Parallel computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing*.

[81] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[82] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 63–74.

[83] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for

Multicores. *IEEE Computer* 42, 12 (2009), 36–42. http://www.ecn.purdue.edu/ParaMount/publications/ieeecomputer-Cetus-09.pdf
[84] Johannes de Fine Licht and Torsten Hoefler. 2019. hlslib: Software engineering for hardware design. *arXiv preprint arXiv:1910.04436* (2019).
[85] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).
[86] Denis Demidov. 2012. VexCL: Vector expression template library for OpenCL. (2012).
[87] Joel E Denny, f Seyong Lee, and Jeffrey S Vetter. 2018. CLACC: Translating OpenACC to OpenMP in clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 18–29.
[88] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[89] Patrick Diehl, Madhavan Seshadri, Thomas Heller, and Hartmut Kaiser. 2018. Integration of CUDA Processing within the C++ Library for Parallelism and Concurrency (HPX). In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 19–28.
[90] Romain Dolbeau, Stéphane Bihan, and François Bodin. 2007. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, Vol. 28.
[91] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38, 8 (2012), 391–407.
[92] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104.
[93] Daniel J Duffy. [n. d.]. Analysis of Covid-19 Mathematical and Software Models. ([n. d.]).
[94] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 02 (2011), 173–193.
[95] Dwave. 2020. Dwave Online Information. (2020). https://www.dwavesys.com/quantum-computing
[96] Ebb. 2020. Ebb Online Portal. (2020). http://ebblang.org/
[97] H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. 2012. Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming* 20, 2 (2012), 89–114.
[98] H Carter Edwards and Christian R Trott. 2013. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 18–24.
[99] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.
[100] Thomas Eickermann, Wolfgang Frings, Stefan Posse, Gernot Goebbels, and Roland Volpel. 1999. Distributed applications in a german gigabit WAN. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*. IEEE, 143–148.
[101] Eigen. 2020. Eigen C++ Library Online Reference. (2020). http://eigen.tuxfamily.org/index.php
[102] Vinoth Krishnan Elangovan, Rosa M Badia, and Eduard Ayguade Parra. 2012. OmpSs-OpenCL programming model for heterogeneous systems. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 96–111.
[103] Emu. 2020. Emu Chick Product Page. (2020). https://www.emutechnology.com/products/
[104] EPCC. 2020. EPCC Online Repository. (2020). https://github.com/EPCCed/epcc-openacc-benchmarks
[105] EPCC. 2020. EPCC OpenACC Benchmarks. (2020). https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite
[106] MilošD Ercegovac. 1988. Heterogeneity in supercomputer architectures. *Parallel Comput.* 7, 3 (1988), 367–372.
[107] Mary M Eshaghian and Muhammad E Shaaban. 1994. Cluster-M parallel programming paradigm. *International Journal of High Speed Computing* 6, 02 (1994), 287–309.
[108] SK Esser, PA Merolla, JV Arthur, AS Cassidy, R Appuswamy, A Andreopoulos, DJ Berg, JL McKinstry, T Melano, DR Barch, et al. 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. 2016. *Preprint on ArXiv. http://arxiv. org/abs/1603.08270. Accessed* 27 (2016).
[109] Graham E Fagg and Jack J Dongarra. 1996. PVMPI: An Integration of the PVM and MPI Systems. *Calculateurs Paralleles* 2 (1996).
[110] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Cvpr 2011 Workshops*. IEEE, 109–116.
[111] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. 2012. OpenCL and the 13 dwarfs: a work in progress. In *Proceedings of the 3rd acm/spec international conference on performance engineering*. 291–294.
[112] Ian Foster. 2002. The physiology of the grid: An open grid services architecture for distributed systems integration. (2002).
[113] Ian Foster. 2003. The grid: Computing without bounds. *Scientific American* 288, 4 (2003), 78–85.
[114] Ian Foster and Carl Kesselman. 1997. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing* 11, 2 (1997), 115–128.
[115] Ian Foster, Robert Olson, and Steven Tuecke. 1992. Productive parallel programming: The PCN approach. *Scientific Programming* 1, 1 (1992), 51–66.
[116] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. 2008. Cloud computing and grid computing 360-degree compared. In *2008 grid computing environments workshop*. Ieee, 1–10.
[117] Richard F. Freund and Howard Jay Siegel. 1993. Guest Editor's Introduction: Heterogeneous Processing. *Computer* 26, 6 (June 1993), 13âĂŞ17.
[118] M Fukugita, K Shimasaku, T Ichikawa, JE Gunn, et al. 1996. *The Sloan digital sky survey photometric system.* Technical Report. SCAN-9601313.
[119] Edgar Gabriel, Michael Resch, and Roland Ruehle. 1999. Implementing MPI with optimized algorithms for metacomputing. (1999).
[120] Srinidhi Ganeshan, Naveen Kumar Elumalai, and Ramachandra Achar. 2020. A Comparative Study of MAGMA and cuBLAS Libraries for GPU based Vector Fitting. In *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 1–4.
[121] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 751–764.
[122] GA Geist, James A Kohl, and Phil M Papadopoulos. 1996. PVM and MPI: A comparison of features. *Calculateurs Paralleles* 8, 2 (1996), 137–150.
[123] David Gelernter and David Kaminsky. 1992. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 6th international conference on Supercomputing*. ACM, 417–427.
[124] Kate Gregory and Ade Miller. 2012. *C++ AMP*. Microsoft Press.
[125] Andrew Grimshaw, Adam Ferrari, Greg Lindahl, and Katherine Holcomb. 1998. Metasystems. *Commun. ACM* 41, 11 (1998), 46–55.
[126] Andrew S Grimshaw. 1993. Easy-to-use object-oriented parallel processing with Mentat. *Computer* 5 (1993), 39–51.
[127] Andrew S Grimshaw, Jon B Weissman, Emily A West, and Ed C Loyot. 1994. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *J. Parallel and Distrib. Comput.* 21, 3 (1994), 257–270.
[128] Andrew S Grimshaw, Wm A Wulf, and CORPORATE The Legion Team. 1997. The Legion vision of a worldwide virtual computer. *Commun. ACM* 40, 1 (1997), 39–45.
[129] William Gropp and Ewing Lusk. 1993. The MPI communication library: its design and a portable implementation. In *Proceedings of Scalable Parallel Libraries Conference*. IEEE, 160–165.
[130] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.
[131] Khronos Group. 2020. Open Compute Language Online Portal. (2020). http://www.khronos.org/opencl/
[132] The Portland Group. [n. d.]. PGI Compiler Web Reference, year = 2020, url = https://www.pgroup.com/index.htm,. ([n. d.]).
[133] The Portland Group. [n. d.]. PGI OpenCL Press Release, year = 2020, url = https://www.khronos.org/news/permalink/pgi-opencl-compiler-for-arm,. ([n. d.]).
[134] Michael Gschwind, H Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. 2006. Synergistic processing in cell's multicore architecture. *IEEE micro* 26, 2 (2006), 10–24.
[135] Antonio Gulli and Sujit Pal. 2017. *Deep learning with Keras*. Packt Publishing Ltd.
[136] Halide. [n. d.]. Halide Online Portal, year = 2020, url = https://halide-lang.org/,. ([n. d.]).
[137] Halide. [n. d.]. Halide Online Repository, year = 2020, url = https://github.com/halide/Halide,. ([n. d.]).
[138] FA Binti Hamzah, C Lau, H Nazri, DV Ligot, G Lee, CL Tan, et al. 2020. CoronaTracker: worldwide COVID-19 outbreak data analysis and prediction. *Bull World Health Organ* 1 (2020), 32.
[139] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep

neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.
[140] Tianyi David Han and Tarek S Abdelrahman. 2009. hi CUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. 52–61.
[141] Tianyi David Han and Tarek S Abdelrahman. 2010. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed systems* 22, 1 (2010), 78–90.
[142] Kenneth A Hawick, Heath A James, AJ Silis, Duncan A Grove, Craig J Patten, JA Mathew, Paul D Coddington, KE Kerry, JF Hercus, and FA Vaughan. 1999. DISCWorld: an environment for service-based matacomputing. *Future Generation Computer Systems* 15, 5-6 (1999), 623–635.
[143] Beat Heeb and Cuno Pfister. 1992. Chameleon: A workstation of a different colour. In *International Workshop on Field Programmable Logic and Applications*. Springer, 152–161.
[144] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. 2017. Hpx–an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo* (2017), 5.
[145] hipSYCL. 2020. hipSYCL Compiler Online Repository. (2020). https://github.com/illuhad/hipSYCL
[146] Richard D Hornung and Jeffrey A Keasler. 2014. *The RAJA portability layer: overview and status*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).
[147] IBM. 2020. IBM OpenMP Offloading Web Reference. (2020). https://www.ibm.com/support/knowledgecenter/en/SSXVZZ$_1$6.1.0/com.ibm.xlcpp161.lelinux.doc/proguide/offloading.html
[148] IBM. 2020. IBM Power9 Chip. (2020). https://www.ibm.com/it-infrastructure/power/power9
[149] ORNL ICONS. 2020. International Conference on Neuromorphic Systems. (2020). https://icons.ornl.gov/
[150] Intel. 2020. Intel FPGA SDK for OpenCL. (2020). https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
[151] Intel. 2020. Intel OneAPI Online Portal. (2020). https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-library.html
[152] Intel. 2020. Intel OneAPI Press Release. (2020). https://newsroom.intel.com/news-releases/intel-unveils-new-gpu-architecture-optimized-for-hpc-ai-oneapi/
[153] Intel. 2020. Loihi Chip. (2020). https://www.intel.com/content/www/us/en/research/neuromorphic-computing.html
[154] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* 43, 5 (2015), 752–785.
[155] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.
[156] William E Johnston, Dennis Gannon, and Bill Nitzberg. 1999. Grids as production computing environments: The engineering aspects of NASA's Information Power Grid. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*. IEEE, 197–204.
[157] Norman P Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. A domain-specific architecture for deep neural networks. *Commun. ACM* 61, 9 (2018), 50–59.
[158] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W Hwu, et al. 2014. Spec accel: A standard application suite for measuring hardware accelerator performance. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 46–67.
[159] Ilyes Kacher, Maxime Portaz, Hicham Randrianarivo, and Sylvain Peyronnet. 2020. Graphcore C2 Card performance for image-based deep learning application: A Report. *arXiv preprint arXiv:2002.11670* (2020).
[160] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 1–11.
[161] Laxmikant V Kale and Sanjeev Krishnan. 1993. Charm++ A portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 91–108.
[162] Nicholas T Karonis, Brian Toonen, and Ian Foster. 2003. MPICH-G2: A grid-enabled implementation of the message passing interface. *J. Parallel and Distrib. Comput.* 63, 5 (2003), 551–563.
[163] David B Keator, Jeffrey S Grethe, Daniel Marcus, B Ozyurt, Syam Gadde, Sean Murphy, Steve Pieper, D Greve, R Notestine, H Jeremy Bockholt, et al. 2008. A national human neuroimaging collaboratory enabled by the Biomedical Informatics Research Network (BIRN). *IEEE Transactions on Information Technology in Biomedicine* 12, 2 (2008), 162–172.
[164] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*. 1–1.
[165] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, et al. 2019. MIOpen: An Open Source Library For Deep Learning Primitives. *arXiv preprint arXiv:1910.00078* (2019).
[166] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and C-L Wang. 1993. Heterogeneous computing: Challenges and opportunities. *Computer* 26, 6 (1993), 18–27.
[167] Andreas Klöckner. 2010. Pycuda: Even simpler gpu programming with python. *Published on: Sep* 22 (2010), 65.
[168] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38, 3 (2012), 157–174.
[169] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, AD Sarma, D Nanongkai, G Pandurangan, P Tetali, et al. 2009. PyCUDA: GPU run-time code generation for high-performance computing. *Arxiv preprint arXiv* 911 (2009).
[170] Charles H Koelbel, David B Loveman, Robert S Schreiber, Guy Lewis Steele Jr, and Mary Zosel. 1994. *The high performance Fortran handbook*. MIT press.
[171] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80.
[172] D Kuck, E Davidson, D Lawrie, A Sameh, Chuan-Qi Zhu, A Veidenbaum, Jeff Konicek, P Yew, Kyle Gallivan, William Jalby, et al. 1993. The Cedar system and an initial performance study. In *Proceedings of the 20th annual international symposium on Computer architecture*. 213–223.
[173] Jacob Lambert, Seyong Lee, Jungwon Kim, Jeffrey S Vetter, and Allen D Malony. 2018. Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In *Proceedings of the 2018 International Conference on Supercomputing*. 160–171.
[174] Jacob Lambert, Seyong Lee, Allen Malony, and Jeffrey S Vetter. 2019. CCAMP: OpenMP and OpenACC Interoperable Framework. In *European Conference on Parallel Processing*. Springer, 357–369.
[175] Jacob Lambert, Seyong Lee, Allen Malony, and Jeffrey S Vetter. 2020. CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP. In *SuperComputing*.
[176] Jacob Lambert, Seyong Lee, Jeffrey S Vetter, and Allen Malony. 2020. In-Depth Optimization with the OpenACC-to-FPGA Framework on an Arria 10 FPGA. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 460–470.
[177] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The alpine in situ infrastructure: Ascending from the ashes of strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*. 42–46.
[178] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.
[179] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.
[180] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of MooreâĂŹs Law. *arXiv:2002.11054 [cs]* (Feb 2020). http://arxiv.org/abs/2002.11054 arXiv: 2002.11054.
[181] Jinpil Lee and Mitsuhisa Sato. 2010. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 413–420.
[182] Seyong Lee and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–11.
[183] Seyong Lee, Jungwon Kim, and Jeffrey S Vetter. 2016. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 544–554.
[184] Seyong Lee, Jacob Lambert, Jungwon Kim, Jeffrey S Vetter, and Allen D Malony. 2018. OpenACC to FPGA: A Directive-Based High-Level Programming Framework for High-Performance Reconfigurable Computing. (2018).
[185] Seyong Lee, Jeremy S Meredith, and Jeffrey S Vetter. 2015. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 405–414.
[186] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan*

Notices 44, 4 (2009), 101–110.

[187] Seyong Lee and Jeffrey S Vetter. 2014. OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 115–120.

[188] Stanford Legion. 2020. The Legion ECP Project. (2020). https://www.exascaleproject.org/research-group/programming-models-runtimes/

[189] Stanford Legion. 2020. The Legion Online Repository. (2020). https://github.com/StanfordLegion/legion

[190] Stanford Legion. 2020. The Legion Parallel Programming System. (2020). https://legion.stanford.edu/

[191] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. 1998. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM.

[192] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. 2010. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 51–61.

[193] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. 2007. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience* 19, 18 (2007), 2317–2332.

[194] Chunhua Liao, Yonghong Yan, Bronis R De Supinski, Daniel J Quinlan, and Barbara Chapman. 2013. Early experiences with the OpenMP accelerator model. In *International Workshop on OpenMP*. Springer, 84–98.

[195] Odient Library. [n. d.]. Odient Online Repository. ([n. d.]).

[196] Greg Lindahl, Andrew Grimshaw, Adam Ferrari, and Katherine Holcomb. 1998. Metacomputing–WhatâĂŹs in it for me? *White paper. http://legion.virginia.edu/papers.html* (1998).

[197] G Jack Lipovski. 1988. SIMD and MIMD processing in the Texas Reconfigurable Array Computer. In *Proceedings COMPSAC 88: The Twelfth Annual International Computer Software & Applications Conference*. IEEE, 268–269.

[198] Liszt. 2020. Liszt Online Portal. (2020). http://graphics.stanford.edu/hackliszt/

[199] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 369–381.

[200] LLNL. 2020. Rose Compiler Online Repository. (2020). https://github.com/rose-compiler/rose

[201] DOE LLNL. 2020. El Capitan Supercomputer. (2020). https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

[202] DOE LLNL. 2020. Sierra Supercomputer. (2020). https://computing.llnl.gov/computers/sierra

[203] Steven Lucco and Oliver Sharp. 1990. Delirium: an embedding coordination language. In *Supercomputing'90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. IEEE, 515–524.

[204] Ewing L Lusk, William W McCune, and John Slaney. 1992. Roo: A parallel theorem prover. In *International Conference on Automated Deduction*. Springer, 731–734.

[205] José Carlos Machicao. [n. d.]. Covid-19 infection speed and acceleration as better tools for monitoring with uncertain data in Peru. ([n. d.]).

[206] Heterogeneous Parallel Virtual Machine. 2020. HPVM Online Portal. (2020). https://publish.illinois.edu/hpvm-project/

[207] Heterogeneous Parallel Virtual Machine. 2020. HPVM Online Repository. (2020). https://gitlab.engr.illinois.edu/llvm/hpvm-release

[208] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. 2012. ArrayFire: a GPU acceleration platform. In *Modeling and simulation for defense systems and applications VII*, Vol. 8403. International Society for Optics and Photonics, 84030A.

[209] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, Wayne Gaudin, and David Beckingsale. 2015. A performance evaluation of Kokkos & RAJA using the TeaLeaf mini-app. In *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC15*.

[210] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, and M. Bussmann. 2017. Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library. arXiv:1706.10086 https://arxiv.org/abs/1706.10086

[211] Arya Mazaheri, Johannes Schulte, Matthew W Moskewicz, Felix Wolf, and Ali Jannesari. 2019. Enhancing the programmability and performance portability of GPU tensor operations. In *European Conference on Parallel Processing*. Springer, 213–226.

[212] D Merrill. 2015. CUB v1. 5.3: CUDA Unbound, a library of warp-wide, blockwide, and device-wide GPU parallel primitives. *NVIDIA Research* (2015).

[213] Kenneth Moreland, Christopher Sewell, William Usher, Li-ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, et al. 2016. Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE computer graphics and applications* 36, 3 (2016), 48–58.

[214] Frank Mueller. 1993. Pthreads library interface. *Florida State University* (1993).

[215] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. 2014. XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In *2014 First Workshop on Accelerator Programming using Directives*. IEEE, 27–36.

[216] M Naumov, LS Chien, P Vandermersch, and U Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[217] Chris J Newburn, Serguei Dmitriev, Ravi Narayanaswamy, John Wiegert, Ravi Murty, Francisco Chinchilla, Rajiv Deodhar, and Russ McGuire. 2013. Offload compiler runtime for the Intel® Xeon Phi coprocessor. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1213–1225.

[218] Tianyun Ni. 2009. Direct Compute: Bring GPU computing to the mainstream. In *GPU technology conference*. sn, 23.

[219] NVIDIA. [n. d.]. NVCC CUDA Compiler Online Reference, year = 2020, url = https://developer.nvidia.com/cuda-llvm-compiler,. ([n. d.]).

[220] NVIDIA. [n. d.]. Nvidia Math Libraries. ([n. d.]).

[221] Nvidia. 2020. NVIDIA A100. (2020). https://www.nvidia.com/en-us/data-center/a100/

[222] NVIDIA. 2020. NVIDIA cuBLAS Library. (2020). https://developer.nvidia.com/cublas

[223] Nvidia. 2020. NVIDIA DGX. (2020). https://www.nvidia.com/en-us/data-center/dgx-systems/

[224] Nvidia. 2020. Nvidia HPC SDK. (2020). https://developer.nvidia.com/hpc-sdk

[225] Nvidia. 2020. NVIDIA Jetson AGX Xavier. (2020). https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/

[226] Nvidia. 2020. NVIDIA V100. (2020). https://www.nvidia.com/en-us/data-center/v100/

[227] CUDA NVIDIA. 2012. Toolkit 4.1 CUFFT Library. (2012).

[228] Open64. 2020. Open64 Compiler Online Repository. (2020). https://github.com/open64-compiler/open64

[229] OpenACC. 2020. OpenACC Hackathon Online Reference. (2020). https://www.openacc.org/hackathons

[230] OpenACC. 2020. OpenACC Online Portal. (2020). https://www.openacc.org/

[231] OpenCL. [n. d.]. OpenCL clBlas Library. ([n. d.]).

[232] Pocl Portable OpenCL. 2020. pocl Compiler Online Repository. (2020). https://github.com/pocl/pocl

[233] OpenMP. 2020. OpenMP Online Portal. (2020). https://www.openmp.org/

[234] DOE ORNL. [n. d.]. Frontier Supercomputer, year = 2020, url = https://www.olcf.ornl.gov/frontier/,. ([n. d.]).

[235] DOE ORNL. 2020. Summit Supercomputer. (2020). https://www.olcf.ornl.gov/summit/

[236] Güray Özen, Simone Atzeni, Michael Wolfe, Annemarie Southwell, and Gary Klimowicz. 2018. OpenMP GPU offload in Flang and LLVM. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 1–9.

[237] High Performance ParalleX. 2020. HPX Online Portal. (2020). http://stellar.cct.lsu.edu/projects/hpx/

[238] High Performance ParalleX. 2020. HPX Online Repository. (2020). https://github.com/STEllAR-GROUP/hpx

[239] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[240] Marc Perelló Bacardit. 2019. *Porting Rodinia Applications to OmpSs@ FPGA*. B.S. thesis. Universitat Politècnica de Catalunya.

[241] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical task-based programming with StarSs. *The International Journal of High Performance Computing Applications* 23, 3 (2009), 284–299.

[242] Judit Planas, Rosa M Badia, Eduard Ayguade, and Jesus Labarta. 2013. Self-adaptive OmpSs tasks in heterogeneous environments. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 138–149.

[243] ETH CLAW Project. 2020. Documentation related to the CLAW Project. (2020). https://claw-project.github.io/

[244] GNU Project. 2020. GCC Compiler. (2020). https://gcc.gnu.org/

[245] GNU Project. 2020. GNU OpenACC Status Wiki. (2020). https://gcc.gnu.org/wiki/OpenACC

[246] GNU Project. 2020. GNU OpenMP Status Wiki. (2020). https://gcc.gnu.org/wiki/openmp

[247] LLVM Project. 2020. Clang Compiler Framework Online Portal. (2020). https://clang.llvm.org/

[248] LLVM Project. 2020. LLVM Compiler Framework Online Portal. (2020). https://llvm.org/

[249] LLVM Project. 2020. Multi-Level IR Compiler Framework Online Portal. (2020). https://mlir.llvm.org/

[250] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.
[251] IBM Quantum. 2020. IBM Quantum Press Release. (2020). https://newsroom.ibm.com/2019-01-08-IBM-Unveils-Worlds-First-Integrated-Quantum-Computing-System-for-Commercial-Use
[252] Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters* 10, 02n03 (2000), 215–226.
[253] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
[254] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. 1993. Jade: A high-level, machine-independent language for parallel programming. *Computer* 26, 6 (1993), 28–38.
[255] S Ross-Ross. 2011. Clyther opencl interface for python (website). (2011).
[256] Karl Rupp, Florian Rudolf, and Josef Weinbub. 2010. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*. 51–56.
[257] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jungel, and Siegfried Selberherr. 2016. ViennaCL—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing* 38, 5 (2016), S412–S439.
[258] Jurriaan Rutten. 2020. The spread of COVID-19. (2020).
[259] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. 1997. Ninf: A network based information library for global world-wide computing infrastructure. In *International Conference on High-Performance Computing and Networking*. Springer, 491–502.
[260] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. 1999. Design of OpenMP compiler for an SMP cluster. In *Proc. of the 1st European Workshop on OpenMP*. 32–39.
[261] Boris Schäling. 2011. *The boost C++ libraries*. Boris Schäling.
[262] Paul B. Schneck, D Austin, Stephen L. Squires, J Lehmann, D Mizeil, and Kenneth Wallgren. 1985. Parallel processor programs in the federal government. *Computer* 6 (1985), 43–56.
[263] Ben Segal, F Gagliardi, L Robertson, and Federico Carminati. 2000. Grid computing: the european data grid project. (2000).
[264] Matthew C Sejnowski, Edwin T Upchurch, Rajan N Kapur, Daniel PS Charlu, and G Jack Lipovski. 1980. An overview of the Texas reconfigurable array computer. In *afips*. IEEE, 631.
[265] Graham Sellers and John Kessenich. 2016. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional.
[266] Amazon Web Services. 2020. Inferentia Learning Chip. (2020). https://aws.amazon.com/machine-learning/inferentia/
[267] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.
[268] Howard Jay Siegel, John K Antonio, Richard C Metzger, Min Tan, and Yan Alexander Li. 1995. Goals of and open problems in high-performance heterogeneous computing. In *23rd AIPR Workshop: Image and Information Systems: Applications and Opportunities*, Vol. 2368. International Society for Optics and Photonics, 206–217.
[269] Howard Jay Siegel, Henry G Dietz, and John K Antonio. 1996. Software support for heterogeneous computing. *Acm computing surveys (csur)* 28, 1 (1996), 237–239.
[270] Howard Jay Siegel, Leah J Siegel, Frederick C Kemmerer, Philip T Mueller, Harold E Smalley, and S Diane Smith. 1981. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on computers* 100, 12 (1981), 934–947.
[271] Kyle L Spafford and Jeffrey S Vetter. 2012. Aspen: A domain specific language for performance modeling. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
[272] Spiral. 2020. Spiral Online Portal. (2020). https://www.spiral.net/
[273] StarPU. 2020. StarPU Online Repostiory. (2020). https://gitlab.inria.fr/starpu/starpu
[274] StarPU. 2020. StarPU Online Repostiory. (2020). https://gitlab.inria.fr/starpu/starpu
[275] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. Skelcl-a portable skeleton library for high-level gpu programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 1176–1182.
[276] Alexander D Stoyenko and Theodore P Baker. 1994. Real-time schedulability-analyzable mechanisms in Ada9X. *Proc. IEEE* 82, 1 (1994), 95–107.
[277] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).
[278] Vaidy S. Sunderam. 1990. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience* 2, 4 (1990), 315–339.
[279] Jakub Szuppe. 2016. Boost. Compute: A parallel computing library for C++ based on OpenCL. In *Proceedings of the 4th International Workshop on OpenCL*. 1–39.
[280] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. 2013. A source-to-source OpenACC compiler for CUDA. In *European Conference on Parallel Processing*. Springer, 178–187.
[281] Tensorflow. 2020. Tensorflow MLIR Project Page. (2020). https://www.tensorflow.org/mlir
[282] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience* 17, 2-4 (2005), 323–356.
[283] Mary Thomas, Steve Mock, and Jay Boisseau. 2000. Development of Web toolkits for computational science portals: The NPACI HotPage. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 308–309.
[284] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, Vol. 5. 1–6.
[285] S Tomov, J Dongarra, V Volkov, and J Demmel. 2009. Magma library. *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA* (2009).
[286] Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. 2011. MAGMA UsersâĂŹ Guide. *ICL, UTK (November 2009)* (2011).
[287] Google TPU. 2020. Edge TPU AI Platform. (2020). https://cloud.google.com/edge-tpu
[288] triSYCL. 2020. triSYCL Compiler Online Repository. (2020). https://github.com/triSYCL/triSYCL
[289] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. 2008. CUDA-lite: Reducing GPU programming complexity. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1–15.
[290] General Vision. 2020. NM500 Infomration. (2020). https://www.general-vision.com/nm500/
[291] Yutaka Watanabe, Jinpil Lee, Kentaro Sano, Taisuke Boku, and Mitsuhisa Sato. 2020. Design and Preliminary Evaluation of OpenACC Compiler for FPGA with OpenCL and Stream Processing DSL. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. 10–16.
[292] Charles C Weems, Glen E Weaver, and Steven G Dropsho. 1994. Linguistic support for heterogeneous parallel processing: A survey and an approach. In *Proceedings Heterogeneous Computing Workshop*. IEEE, 81–88.
[293] Michael Wolfe. 2010. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 43–50.
[294] Michael Wolfe and PGI Compiler Engineer. 2009. The PGI Accelerator Programming Model on NVIDIA GPUs Part 1. *PGI Group* (2009).
[295] Rich Wolski, Neil T Spring, and Jim Hayes. 1999. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 15, 5-6 (1999), 757–768.
[296] Benjamin Worpitz. 2015. Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures. (Sep 2015). https://doi.org/10.5281/zenodo.49768
[297] Xilinx. 2020. The SDAccel software development platform. (2020). https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[298] Xilinx. 2020. The Vitis software development platform. (2020). https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html
[299] Xilinx. 2020. The Vivado Design Suite HLx Editions. (2020). https://www.xilinx.com/products/design-tools/vivado.html
[300] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *European Conference on Parallel Processing*. Springer, 887–899.
[301] Xue-Jun Yang, Xiang-Ke Liao, Kai Lu, Qing-Feng Hu, Jun-Qiang Song, and Jin-Shu Su. 2011. The TianHe-1A supercomputer: its hardware and software. *Journal of computer science and technology* 26, 3 (2011), 344–351.
[302] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. 1986. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices* 21, 11 (1986), 258–268.
[303] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E. Nagel, and Michael Bussmann. 2016. Alpaka - An Abstraction Library for Parallel Kernel Acceleration. IEEE Computer Society. arXiv:1602.08477 http://arxiv.org/abs/1602.08477
[304] Xiaodong Zhang and Yong Yan. 1995. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. In *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*. IEEE, 25–34.