A taxonomy of accelerator architectures and their programming models

C. Caşcaval
S. Chatterjee
H. Franke
K. J. Gildea
P. Pattnaik

As the clock frequency of silicon chips is leveling off, the computer architecture community is looking for different solutions to continue application performance scaling. One such solution is the multicore approach, i.e., using multiple simple cores that enable higher performance than wide superscalar processors, provided that the workload can exploit the parallelism. Another emerging alternative is the use of customized designs (accelerators) at different levels within the system. These are specialized functional units integrated with the core, specialized cores, attached processors, or attached appliances. The design tradeoff is quite compelling because current processor chips have billions of transistors, but they cannot all be activated or switched at the same time at high frequencies. Specialized designs provide increased power efficiency but cannot be used as general-purpose compute engines. Therefore, architects trade area for power efficiency by placing in the design additional units that are known to be active at different times. The resulting system is a heterogeneous architecture, with the potential of specialized execution that accelerates different workloads. While designing and building such hardware systems is attractive, writing and porting software to a heterogeneous platform is even more challenging than parallelism for homogeneous multicore systems. In this paper, we propose a taxonomy that allows us to define classes of accelerators, with the goal of focusing on a small set of programming models for accelerators. We discuss several types of currently popular accelerators and identify challenges to exploiting such accelerators in current software stacks. This paper serves as a guide for both hardware designers by providing them with a view on how software best exploits specialization and software programmers by focusing research efforts to address parallelism and heterogeneity.

Introduction

Heterogeneous multicore systems are emerging as one of the most attractive design points for next-generation systems. The reasons are multiple. First is the leveling of clock frequency increases due to power constraints. In 2001, Gelsinger [1] extrapolated that continuing on the frequency scaling at the time would result in chips running at 10–30 GHz, with a heat dissipation equivalent to that of nuclear reactors (proportional to size) by 2011. While researchers have developed chips that run at clock frequencies of tens of gigahertz in the laboratory, commercial microprocessors have embraced parallelism to continue performance scaling. All processor vendors (Advanced Micro Devices [2, 3], ARM [4], IBM [5–7], Intel [8], and Sun Microsystems [9]) have been offering multicore chips for quite some time. Rather than widening the superscalar issue of the cores, designers have focused on multiple simpler cores. For many throughput-oriented workloads, this approach has worked very well; the operating system (OS) and middleware exploit these cores transparently by dispatching independent tasks on separate processors.

Digital Object Identifier: 10.1147/JRD.2010.2059721

©**Copyright** 2010 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied by any means or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of this paper must be obtained from the Editor.

0018-8646/10/$5.00 © 2010 IBM

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

However, only increasing the number of cores is not sustainable [10]. Similar to exploiting instruction-level parallelism in wider superscalar processors [11], automatically exploiting multicore parallelism will produce diminishing returns. In the case of instruction-level parallelism, hardware requires ever-increasing resources (e.g., instruction window and reorder buffers) to overcome instruction dependences. In addition to dependences, as parallelism increases, application programmers are fighting against Amdahl's Law. Therefore, to utilize resources efficiently, all but a vanishingly small amount (> 99%) of the code must be parallelized. Furthermore, the increased number of cores puts additional pressure on the memory subsystem, in particular the memory bandwidth and the number of outstanding memory requests that must be sustained and managed.

In this context, the trend is moving toward specialization, with designs trading off chip area for power. Special-purpose units and processors provide increased power efficiency (operations per second per watt) by implementing particular functions in hardware [12]. These units cannot be used directly for general-purpose computing; thus, software has to be customized to take advantage of their computational power. In this paper, we attempt to classify the space of accelerators with the goal of identifying the most appropriate programming models that target special-purpose computation.

We start by defining an accelerator. An accelerator is a special-purpose hardware device that is optimized to perform a certain function or set of functions as part of a general-purpose computational system. As such, an accelerator is necessarily part of a larger system. It spans a scale, from being closely integrated with a general-purpose processor to being attached to a computing system, and potentially contains general-purpose hardware that is specialized for a particular function through software. In the next sections, we discuss trends in system architecture and workload characteristics that have led to the development of accelerators, present our taxonomy of accelerators, and conclude with an exposition of programming models for accelerators.

Systems characterization

In this section, we provide a broad characterization of the architectural and workload trends that have led to the development and recent popularity of accelerators.

System architecture

Accelerators have been deployed in many scenarios and solutions over the years. A few categories emerge from the multitude of designs.

- *Integrated specialized units*—These special-purpose fixed-function functional units perform operations on behalf of the CPU more efficiently. Examples include the floating-point unit (FPU) and vector units.
- *Specialized application-specific integrated circuits (ASICs)*—These accelerators provide special-purpose processing, typically targeted to a set of algorithms in a particular domain. Examples include digital signal processors and graphics processing units (GPUs). Sometimes, when the functionality provided by these accelerators becomes more pervasive, they are integrated with the CPU, for example, the math coprocessor on the Intel x86 architecture [13].
- *Specialized appliances*—These accelerators are used to offload entire applications to a specialized box that is attached to a general-purpose system. DataPower* [14] and Azul [15] are examples of such systems.
- *Generated accelerators*—These are reconfigurable devices, such as field-programmable gate arrays (FPGAs), which allow applications written in high-level languages to be directly compiled into hardware [16–18].

Today's systems are a hierarchical composition of compute cores organized as nodes that are connected over a coherent system bus to memory and I/O devices attached through I/O bridges. Conceptually, accelerators can be attached at all levels of this hierarchy. The mapping of system function to system level is quite fluid and depends on both system characteristics and offloaded functionality characteristics. Most often, the goal is improved performance, but other metrics, such as power and isolation, play a role in defining the granularity and integration of the accelerator.

The integration of accelerators into a general-purpose system is in fact one of the key aspects that leads to their popularity. At one end of the spectrum are appliances that are connected to the general-purpose system through a standard interconnect (Gigabit Ethernet or Infiniband**). These have been fairly successful because they provide a turnkey solution; entire workloads are offloaded and managed outside the general-purpose system. The existing software stack is not perturbed; thus, no additional testing and certification is needed. At an intermediate level are specialized cores (such as GPUs). Again, the emergence of standard protocols, such as PCI Express** (PCIe**), makes integration with the general-purpose system much easier. In addition, some companies are also opening and standardizing their bus protocols, such as AMD HyperTransport** [3], thus enabling even closer integration of accelerators. Finally, several companies have been opening up core designs. Examples include embedded PowerPC* [19] and OpenSparc** [20]. This allows designers to extend and integrate proprietary modules on the same die. Systems-on-a-chip are also encouraging the definition of macros that can be integrated within a chip or die. These efforts will certainly enable the proliferation and eventual standardization of accelerators.

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

Workloads

There are a number of workload characteristics that lead to ease of accelerator exploitation. Some of these characteristics, such as parallelism granularity and type (synchronous, asynchronous, or streaming), and modularity deeply influence our taxonomy. A programming model for hybrid computing must encourage a programming style that emphasizes some of these characteristics. In this section, we discuss these aspects in more detail.

One of the most important characteristics is *parallelism granularity*, which is determined by the size of the activities that are executed concurrently. Whether specialized functional units, specialized cores, or offloading appliances, accelerators perform most efficiently when the workload parallelism granularity matches the size of tasks that can be offloaded to the accelerator. Note that a workload may exhibit finer grain concurrency that can be exploited by a coarser grain accelerator by aggregating several concurrent activities into a single execution task. This aggregation may be performed by a runtime system either automatically, through user directives, or manually by the programmer by defining appropriately sized activities. In the latter case, the granularity must be exposed to the programming model in order for the programmer to control it directly. Manual aggregation also leads to codes that are typically not portable between different platforms. The parallelism granularity can be categorized as follows:

- *Instruction-level parallelism*—Best exploited by specialized functional units, such as FPUs or vector units.
- *Computational kernel-level parallelism*—Exemplified by small data-parallel kernels that run well on GPUs, encryption/decryption kernels (not to be confused with OS kernels).
- *Function level*—Typical granularity for exploitation on specialized cores on which certain functions or tasks can be performed more efficiently and in parallel with the main computation thread. These functions can be stateful or stateless.
- *Offload*—The application can be decomposed into loosely coupled modules, some of which are offloaded to an accelerator.

The parallelism granularity affects many dimensions of the accelerator design: coupling with the CPU, invocation overhead, memory coupling, and addressing. The larger the granularity, the more work can be done in parallel, and thus, less coupling with the main CPU is needed.

As parallelism granularity characterizes the size of the activities and, therefore, the amount of work that can be executed concurrently, synchronization between tasks determines how much of the existing concurrency can be exploited through parallel execution on a particular heterogeneous system. We identify three types of *parallelism synchronization* when considering how different activities overlap and synchronize. *Synchronous parallelism* implies that operations are spawned and the spawning task will wait for their completion before proceeding further. Fine-grain synchronous parallelism is best exploited at the instruction level or when multiple [single-instruction, multiple-data (SIMD)-like] tasks can be invoked with a single command. Coarse-grain synchronous parallelism works well when the fraction of parallel work is significantly higher than the work that could be done on the spawning thread. *Asynchronous parallelism* allows for tasks to be spawned and completed arbitrarily. Typically, spawned tasks will signal their completion, and the spawning task may choose whether it will wait for the completion of the spawned task (task completion may be deduced through other properties of the application, such as termination of other dependent tasks). Asynchronous parallelism does favor coarser parallelism, although accelerators can be designed to significantly reduce the overhead of invocation and completion to exploit fine-grain asynchronous parallelism. *Streaming parallelism* exploits parallelism that occurs from processing multiple streams of data. Streaming parallelism is seeing a resurgence with the appearance of sensor networks and devices. Typical streaming applications process streams of data through a pipeline, with stages that are either stateless or almost so. Stream processing has well-defined I/O specifications and, thus, is a good match for acceleration. Specialized accelerators can be defined for pipeline stages or for the entire stream processing task.

These two workload characteristics—parallelism granularity and parallelism synchronization—essentially determine the amount of available and exploitable parallelism in a workload. Of course, when mapping a workload to a particular system, there are numerous other factors that limit the exploitation of parallelism, such as communication overhead and load balancing. An expert programmer will tune the granularity and synchronization of her application to minimize the overhead and efficiently exploit the available parallelism.

Finally, we would like to mention the more traditional classification of parallelism into data, task, and pipeline parallelism. We consider that this classification can be mapped on the granularity and synchronization dimensions as follows:

- *Data parallelism* distributes data across multiple units and executes similar operations on the data operations. A typical example is SIMD, but more generally, we can consider data parallelism as synchronous parallelism at the instruction and kernel levels.
- *Task parallelism* distributes execution tasks to different processing units. Task parallelism is best represented by multiple-instructions, multiple-data (MIMD) processing and is mainly asynchronous kernel- and function-level parallelism.

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

- *Pipeline parallelism* is characteristic of the streaming parallelism at the function level.

We use these characteristics to establish a taxonomy of accelerators (see the next section) and also in our discussion of programming models in the section "Programming models for accelerators."

Taxonomy of accelerators

As special-purpose devices, accelerators can be used in many contexts. In fact, the main reason for their popularity is the ability to customize them to a particular function. The large diversity of such functions, coupled with the diversity of their origin and the emergence of open standards, has resulted in a huge design space of accelerators that are increasingly more difficult to integrate and program. In this section, we classify the space of accelerators in an attempt to help designers position their accelerators to take advantage of existing and emerging standardized interfaces, programming models, and programming environments.

There are three key dimensions along which we define our taxonomy: 1) architecture, 2) invocation, and 3) addressing. There are also additional secondary dimensions, such as management and programming level. In **Table 1**, we present a number of existing accelerators classified along these dimensions, and we discuss the characteristics of these dimensions.

The taxonomy dimensions are as follows:

1) *Architecture* defines the accelerator architecture.
   a) *Fixed architectures* are justified for functions that have a very high degree of reuse such as standardized operations that gain substantial performance and power benefits from being implemented in hardware. Examples of fixed architectures are FPUs and cryptographic accelerators that are typically integrated on-chip or on-board or externally as I/O attached devices using ASIC technology.
   b) *Programmable architectures* allow more flexibility while maintaining a customized function. Examples are systems such as programmable network interface controllers (NICs) and CPU- [21] or GPU-based systems [22].
   c) *Reconfigurable architectures* provide the most flexibility: Both the architecture and the function are programmable. FPGAs enable modification at runtime [16]. Such systems have been deployed as financial feed handlers [23]. Though flexible, they have some drawbacks, such as lower frequencies than ASICs and higher power.
2) *Invocation and completion* determines function invocation and completion semantics. There is a correlation between the invocation granularity and the coupling of the accelerator to the host system.
   a) *Instruction level*—Captures accelerators that have been integrated into the processor instruction set architecture (ISA). These are typically short latency operations, such as floating-point operations and vector operations. Floating-point instructions are part of all existing ISAs, whereas vector instructions are now part of many instruction sets (POWER* VMX, Intel SSE [streaming SIMD extensions]). Their execution is typically synchronous.
   b) *Command packet*—Accelerators that are attached as I/O devices in many cases are invoked through memory-mapped I/O registers. Their invocation is, in many cases, asynchronous, where the completion and results are polled for or are signaled through interrupt mechanisms. A generic service instruction that enables invocation of accelerators that are attached to the system bus is described in [24]. Examples are cryptographic engines and Transmission Control Protocol/Internet Protocol (TCP/IP) offload.
   c) *Task level*—Represents accelerators with a coarse-grain invocation. This basically implies that the accelerator is programmable and the function may take long to complete. To support concurrence, such accelerators are typically asynchronous.
   d) *Workload*—Offloads the entire application to a workload-specialized appliance. These are typically standalone systems that connect to the host system through the network. They perform entire tasks. The host system bootstraps the operation and often does not see the results.
3) *Memory addressing* characterizes the way accelerators integrate with the memory of the host system. This includes memory addressing, memory partitioning, and coherency.
   a) *Addressing*—The accelerator plugs into the application address space and uses the CPU address translation or has its own address space or address translation. Examples for shared addressing are the Synergistic Processor Units on the Cell Broadband Engine** (Cell/B.E.**) processor [25]. Examples of separate address spaces are I/O devices that have the host maintain address translations through explicit I/O memory management unit management.
   b) *Partitioning*—Independent of the address space, accelerator memory can be shared with the CPU, distributed (separate memory, accessible by the CPU through a bus or interconnect), or completely separate (local to the accelerator and the only access is through messages processed by the accelerator). Examples of shared memory include multicores and functional units. Distributed memory appears in GPUs, whereas appliances typically have separate memory.
   c) *Coherence*—Most shared-memory closely integrated (system bus) accelerators are coherent. Typically, the I/O bus and network-attached accelerators are noncoherent.

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

Table 1 Accelerator characteristic dimensions. (Sync: synchronous; async: asynchronous; cmd: command; mgmt: management; OS: operating system; RNIC: RDMA-enabled network interface controller; GPU: graphics processing unit; APGAS: asynchronous partitioned global address space; PGAS: partitioned global address space; MPI: Message Passing Interface; FPGA: field-programmable gate array.)

Accelerator	*Programming models*		*Architecture*	*Invocation and completion*			*Addressing memory partitioning*
	System	*Application*		*Granularity*	*Synchronization*	*Synchronization method*	
Functional unit: floating point, vector	Compiler supported	Data types and intrinsics	Fixed	Instruction	Sync	Implicit: instruction issue	Shared, coherent
Parallel Sysplex*	Firmware for queues and lock mgmt, resource mgmt and threading	OpenMP**, APGAS, MPI	Fixed	Instruction (lock, enqueue, dequeue)	Sync, async	Sync instruction issue, explicit async tasking	Distributed, PGAS
Checksum, compression, Unicode	Microcode, device drivers	Libraries	Programmable	Cmd packet	Sync, async	Sync: explicit instruction issue	Shared, coherent
Crypto			Fixed			Async: explicit wait on response (poll or interrupt)	
TCP/IP offload, RNIC	Kernel device driver	Libraries (BSD sockets)	Programmable	Task/cmd invocation	Async	Explicit	Shared
Multicore, multithreaded	Threading	OpenMP, APGAS, MPI	Programmable	Parallel task	Sync, async	Explicit	Shared, coherent
GPU, Cell	Device drivers and resource mgmt	OpenCL**, OpenMP, APGAS, CUDA, Cell SDK	Programmable	Parallel task	Async	Explicit	Distributed/PGAS
XML (DataPower)	OS	Libraries	Programmable	Parallel task, workload	Async, none	Explicit	Distributed
DWA compute offload	OS, SQL query engine	Libraries	Programmable	Workload	Async, none	Explicit	Distributed
FPGA	VHDL, Verilog	Libraries, data parallel (bit level)	Reconfigurable	Emulated accelerator, application	Async, none	Explicit	Emulated accelerator, distributed

We can draw the following conclusions with regard to programming models from Table 1.

- Libraries are a universal "programming model" for all kinds of accelerators, although they may not be the most elegant or most efficient (due to composability issues). Typically, they are the first programming model that is developed for any accelerator, and higher level programming models are often built (and depend internally) on the interfaces provided by libraries. Thoughtful design of the library application program interfaces (APIs) is therefore recommended.
- Several categories of accelerators require explicit programming models beyond libraries. Such accelerators include clustered or multithreaded processor complexes, and the kinds of SIMD processors exemplified by

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

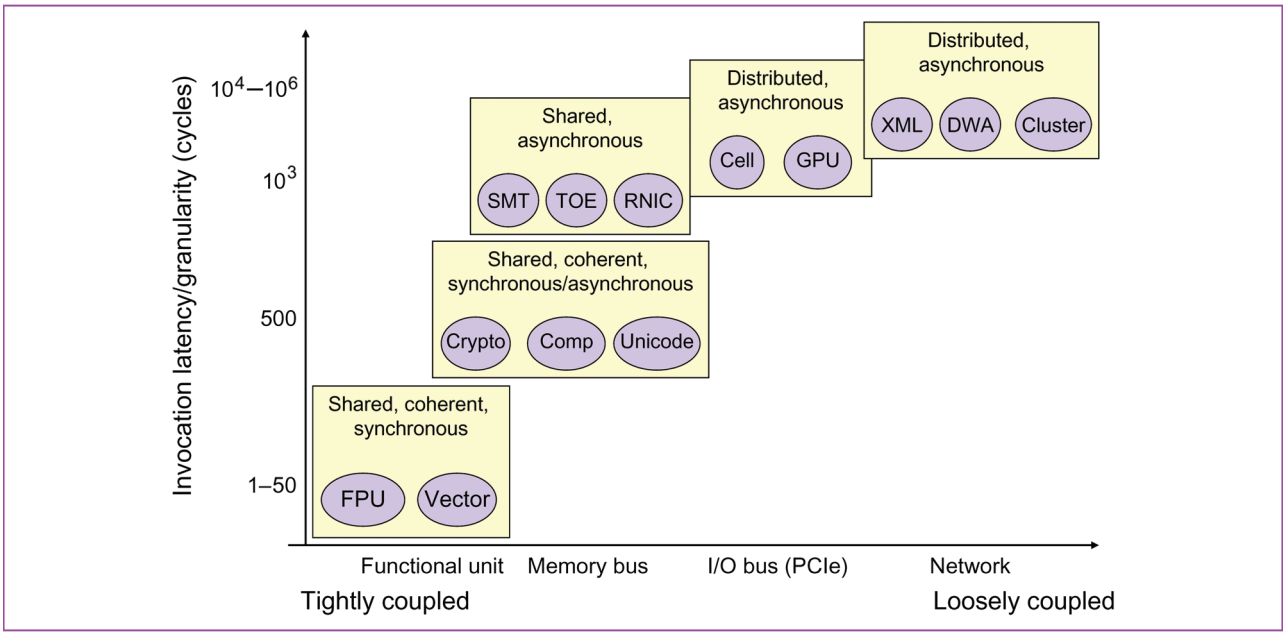

Figure 1.

Invocation granularity and latency versus CPU coupling. (GPU: graphics processing unit; RNIC: RDMA-enabled network interface controller; TOE: TCP offload engine; FPU: floating-point unit.)

Cell/B.E., GPUs, and the Intel Larrabee. We discuss classes of programming models later in this paper.

- Because of their inherent morphable nature, FPGA-based accelerators may require multiple programming models. Obviously, the programming model of the accelerator emulated through an FPGA would be a direct match. However, because of the FPGA characteristics, such models are not necessarily the most efficient. For example, FPGAs provide a high degree of SIMD parallelism. While they can be programmed like GPUs, it is more effective to program them at the level of bit-level parallelism.

Based on the preceding discussion of existing accelerators, **Figure 1** places them along two dimensions: 1) the degree of coupling between the accelerator and the main processor and 2) the invocation granularity in processor cycles. We discretize the notion of "degree of coupling" based on the system attach point: functional unit, memory bus, I/O bus, or network. The ranges along the "invocation granularity" axis are representative and should be considered as guidelines rather than hard numbers. Unsurprisingly, accelerators tend to fall along the diagonal of this graph.

Programming models for accelerators

A critical aspect of exploiting accelerators efficiently is programming. In general, the more details of the hardware architecture that are exposed, the greater the control programmers have over these devices. However, greater control means increased complexity in programming. The goal in this section is to provide an overview of existing and proposed techniques that are used for programming accelerators. Using our taxonomy, it becomes quite easy to reason which programming techniques are suited for different types of accelerators, as well as estimate the level of system support that has to be designed to enable these techniques. **Figure 2** depicts this important tradeoff.

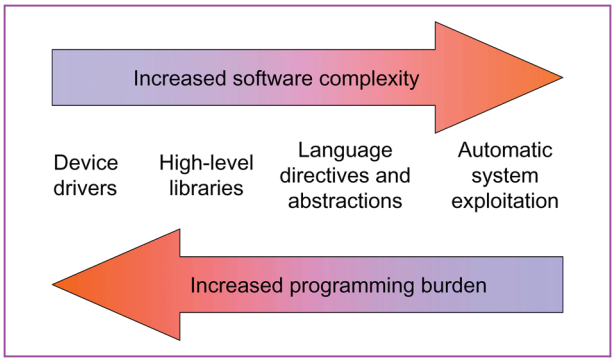

Figure 2.

Complexity tradeoff: The simpler the programming model, the higher the burden on the user to exploit accelerators.

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

Low-level libraries and device drivers

At the bottom of the stack are low-level libraries and device drivers. These libraries and drivers are an essential part of the system software and the OS and make the accelerator available to users. Drivers and initialization routines are needed even for tightly coupled accelerators that are part of the processor core, as the hardware is booted and made ready for execution.

There is not much that a programming model can do for the system programmers responsible for these; they will typically use assembly language or C and have intimate knowledge of the hardware operation, instruction sets, and device control functions. They must closely integrate with a particular OS, and thus, the code is not necessarily portable across platforms. However, an effort should be made to ensure that the exposed APIs from these libraries either follow existing standards or are designed in a generic reusable fashion such that higher level libraries can be developed in a platform-independent way.

One of the best known examples of a well-designed API is the Open Systems Interconnection model [26, 27]. In particular, the data link layer specifies a number of standard protocols such as Ethernet (IEEE 802.3) and wireless (IEEE 802.11) that provide a foundation to implement the higher levels in the model. NIC manufacturers adhere to these protocols. Higher layers build upon these APIs and provide higher levels of abstraction, for example, the IP protocol in the network layer or the TCP/IP protocol in the transport layer. TCP offload [28] uses the NIC to accelerate the processing of the TCP protocol. As general-purpose CPUs continue to enjoy performance increases, there is no compelling reason to move the processing outside of the CPU. However, as frequency increases taper off and new protocols such as Remote Direct Memory Address (RDMA) become more popular, offloading becomes attractive [28]. NICs can also do packet reassembly and, thus, accelerate data transfers. It is very likely that such accelerators will remain hidden from application programmers.

High-level libraries

As a programming paradigm, libraries are universal. The functionality of any accelerator can be encapsulated in a library with an API that can be called from a variety of programming languages and frameworks. For a number of application domains, libraries are both widely accepted and popular [29–32].

For example, a widely used high-level library is the cryptography suite OpenSSL** [32], which provides a security layer for Internet connections. The exposed API of the OpenSSL library defines a set of standard functions and algorithms. Internally, it defines *engines*—a set of functions specific for particular cryptographic accelerators. These functions can be supported in hardware or software. Functions that are not supported in hardware for a particular platform rely on software implementations. However, because of the engine abstraction, new accelerators can be easily integrated without the need to change (or, if dynamic libraries are supported, even recompile) the application.

Intel QuickAssist [33] also provides a high-level library that exposes a service-oriented architecture interface to accelerators. In its model, the accelerator abstraction layer is a bundle of software that virtualizes the accelerator by implementing an accelerator proxy that communicates with a set of device drivers. These device drivers may implement the acceleration in software or control a hardware device. Again, the advantage is encapsulation; once a set of services is defined, one can change the implementation of the library without the need to change the application using the services. QuickAssist further defines dynamic registration and discovery of services. This simplifies the management of different devices on a platform, but the greater flexibility impacts the programmability of applications that use these services; programmers now have to deal with metaprogramming. For particular services, some of the metaprogramming techniques may of course be encapsulated in libraries.

Although libraries are popular and convenient, they do have a number of shortcomings.

Composability—Library APIs impose a rigid boundary that must be obeyed. There are many situations in which removing the boundary enables optimizations. For example, consider an offload accelerator (such as a GPU) with separate memory and two operations, the second using the result produced by the first. Using a library API that defines the first operation as returning the result will force the data copy of the result from the accelerator memory to the host memory and then copy it back for the execution of the second operation, although the two could execute back to back without the extra copies.
Semantics—The semantics of library calls is defined outside of the API, usually documented in a manual. It relies on the library implemented to keep these consistent, and there is no easy way in which to check that they are consistent. In addition, it is not possible to check whether the use of the library is consistent with the defined semantics, unless the libraries are generated from a higher level description.
Explicit coding—Using library APIs requires explicitly coding the function calls in the application. As the library interfaces change with new releases, the applications need to change as well. This is not much of an issue with mature libraries, but a more productive solution is to use abstractions and generate the calls to the library.

As mentioned, even with these drawbacks, libraries are hugely popular, and they form the backbone of programming

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

at a higher level, since compilers also rely on libraries to provide the functionality of higher level abstractions.

Language extensions and language abstractions

A common paradigm to ease the programming burden is to provide high-level abstractions that are expressed through programming extensions. In this approach, it is the responsibility of the compiler and runtime system to ensure the efficient utilization of the accelerator. The classic example for language abstractions is the floating-point data type. Most modern languages provide a floating-point numeric type, for example, float and double in C and related languages, float in Python, and double in Java**. The compiler understands the semantics of these data types and directly generates code for the underlying accelerators.

There is a spectrum of language extensions.

- Extensions to existing languages to directly support abstractions. A number of intrinsic functions encapsulate functionality that the compiler knows about and can generate code directly. For example, vector intrinsics are popular today, and it is expected that they will be more closely integrated into the language in the future, such as the case with OpenCL [34]. OpenCL also defines the notion of a kernel and a new memory model that takes into consideration the fact that there are multiple address spaces (host and accelerator) and exposes these abstractions to the programmer.
- Extensions to existing languages to support libraries encapsulating accelerators. This approach conveys semantic information to the compiler so the compiler and runtime system can better optimize for the exploitation of the accelerator. Examples include Unicode [31] and Secure Socket Layer (SSL) [32].
- New domain-specific languages allow full exploitation of accelerators at the language level. The most popular in this category are graphics languages, such as Brook [35], Cg (C for graphics) [36], and StreamC [37]. In this case, the programmer has the means of expressing computation that is offloaded to a GPU, including data structures and predefined functions that are implemented directly to take advantage of the accelerator customization.

Automatic exploitation

Of course, the least intrusive technique of running with accelerators is through automatic exploitation. In this approach, the compiler and runtime system discover sections of code (instructions or entire procedures) that can be offloaded on an acceleration engine. This is quite challenging for compiler writers, and the results are typically restricted to small classes of accelerators and applications domains. In our taxonomy, one example of automatic exploitation is autoparallelization to take advantage of multicore acceleration. Automatic parallelization builds on 40 years of research in compiler technologies, such as data dependence, pointer analysis, and locality analysis, and leverages concurrence APIs and extensions, such as OpenMP.

Another example is autovectorization, the automatic exploitation of vector units. Surprisingly, it is a technology that is still evolving, although vector units appeared in Cray 1. However, with multiple processor vendors offering different vector architectures, we have seen a proliferation of vendor-specific low-level interfaces to program these units. Examples include ISA extensions such as MMX, SSE1-4, VMX, and the Cell/B.E. SPU. Significant research has been done to raise the level of abstraction and use compiler technologies to eliminate the need of manual programming [38, 39]. The results of automatic compiler-driven vectorization did not meet the level of expectations due to a combination of factors: limitations in the hardware architecture that requires that accesses are aligned, lack of instructions to cover a number of possible operations, and limitations in the compiler analysis related to static disambiguation of dependences and pointers. As a consequence, the community is now looking at other approaches, primarily programming language extensions, that explicitly provide higher level abstractions such as vectors, compute tasks, and kernels that can be mapped to many different architectures and, therefore, alleviate the burden on the compiler to automatically discover when, for instance, vector units can be utilized or how tasks need to be split up. Examples of this approach are OpenCL [34], Cg [36], and the Brook stream processing language [35].

Conclusions

We analyzed a set of existing accelerator-based systems with respect to their system characteristics. Based on the identified characteristics, we defined a taxonomy of accelerators and discussed the available and emerging programming models for these systems. We propose our classification as a design guideline for hardware architects to help with the convergence of specialized units into a small number of categories for which programming models and programming environments can be developed and reused.

Acknowledgments

The authors would like to thank Rodric Rabbah, Vijay Saraswat, and Craig Stunkel for their input on defining the taxonomy presented in this paper.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Infiniband Trade Association, PCI-SIG Corporation, Advanced Micro Devices, Inc., SPARC International, Sony Computer Entertainment Corporation, OpenMP Architecture Review Board, Apple, Inc., The OpenSSL Software Foundation, Sun Microsystems, Inc., Linus Torvalds, or The Open Group in the United States, other countries, or both.

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

References

1. P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *Proc. Int. Solid State Circuits Conf.*, 2001, pp. 22–25.
2. AMD Multicore, Advanced Micro Devices, Sunnyvale, CA, 2009. [Online]. Available: http://www.amd.com/us/products/technologies/multi-core-processing/Pages/multi-core-processing.aspx
3. AMD HyperTransport Technology, Advanced Micro Devices, Sunnyvale, CA, 2009. [Online]. Available: http://www.amd.com/us/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx
4. ARM Cortex-A9 MP Core, ARM, Cambridge, U.K., 2009. [Online]. Available: http://arm.com/products/CPUs/ARMCortex-A9_MPCore.html
5. IBM POWER4 System. *IBM J. Res. & Dev.*, vol. 46, no. 1, 2002, entire issue. [Online]. Available: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=5389038
6. POWER5 and Packaging. *IBM J. Res. & Dev.*, vol. 49, no. 4/5, 2005, entire issue. [Online]. Available: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=5388808
7. IBM POWER6 Microprocessor Technology. *IBM J. Res. & Dev.*, vol. 51, no. 6, 2007, entire issue. [Online]. Available: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=5388622
8. Intel Multicore Technology, Intel Corporation, Santa Clara, CA, 2009. [Online]. Available: http://www.intel.com/multi-core/
9. Sun Niagara Processors, 2009. [Online]. Available: http://www.sun.com/processors/niagara
10. M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
11. S. Palacharla, N. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. ISCA*, 1997, pp. 206–218.
12. G. Pfister, Why accelerators now? Jul. 2009. [Online]. Available: http://perilsofparallel.blogspot.com/2009/07/why-accelerators-now.html
13. Intel Math Co-Processors. [Online]. Available: http://www.cpu-info.com/index2.php?mainid=Copro
14. IBM WebSphere DataPower XML Accelerator, IBM Corporation, Armonk, NY, 2009. [Online]. Available: http://www-01.ibm.com/software/integration/datapower/xa35
15. Azul Syst., Mountain View, CA, 2009. [Online]. Available: http://www.azulsystems.com/
16. A. Hagiescu, W.-F. Wong, D. Bacon, and R. Rabbah, "A computing origami: Folding streams in FPGAs," in *Proc. 46th DAC*, San Francisco, CA, Jul. 2009, pp. 282–287.
17. S. Hauck and A. Dehon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*. San Mateo, CA: Morgan Kaufmann, 2007.
18. H. Ziegler and M. Hall, "Evaluating heuristics in automatically mapping multi-loop applications to FPGAs," in *Proc. ACM/SIGDA 13th Int. Symp. FPGA*, 2005, pp. 184–195.
19. Xilinx Embedded PowerPC Solution, 2005. [Online]. Available: http://www.xilinx.com/prs_rls/ip/0571ppc2mship.htm
20. OpenSparc, 2010. [Online]. Available: http://www.opensparc.net/
21. Intel IXP2800 Network Processor, Intel Corporation, Santa Clara, CA. [Online]. Available: http://download.intel.com/design/network/ProdBrf/27905403.pdf
22. J. Owens, "GPU architecture overview," in *Proc. ACM SIGGRAPH Courses*, 2007, p. 2.
23. Enabling Ultra Low-Latency Trading, 2009. [Online]. Available: http://www.celoxica.com/
24. H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson, "Introduction to the wire-speed processor and architecture," *IBM J. Res. & Dev.*, vol. 54, no. 1, pp. 3:1–3:11, Jan./Feb. 2010.
25. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. Res. & Dev.*, vol. 49, no. 4/5, pp. 589–604, Jul. 2005.
26. The Open System Interconnection Model, ISO/IEC Standard 7498, 1994. [Online]. Available: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip
27. H. Zimmermann, "OSI reference model: The ISO model of architecture for open systems interconnection," *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 425–432, Apr. 1980.
28. J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *Proc. HotOS IX*, May 2003, pp. 25–30.
29. BLAS: Basic Linear Algebra Subprograms. [Online]. Available: http://www.netlib.org/blas/
30. M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE—Special Issue* '*Program Generation, Optimization, and Platform Adaptation*', vol. 93, no. 2, pp. 216–231, Feb. 2005.
31. ICU: International Components for Unicode, 2009. [Online]. Available: http://site.icu-project.org/
32. OpenSSL: Cryptography and SSL/TLS Toolkit. [Online]. Available: http://www.openssl.org/
33. Intel QuickAssist Technology Accelerator Abstraction Layer. [Online]. Available: http://developer.intel.com/technology/platforms/quickassist/index.htm
34. OpenCL, 2009. [Online]. Available: http://www.khronos.org/opencl/
35. I. Buck, T. Foley, D. R. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004.
36. W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," in *Proc. Int. Conf. Comput. Graph. Interactive Tech.*, 2003, pp. 896–907.
37. A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proc. 15th PACT*, 2006, pp. 33–42.
38. A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD architectures with alignment constraints," in *Proc. PLDI*, 2004, pp. 82–93.
39. D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in *Proc. PLDI*, 2006, pp. 132–143.

Received October 24, 2009; accepted for publication March 29, 2010

Calin Caşcaval *Qualcomm Research, Santa Clara, CA 95051 USA (cascaval@acm.org).*

Dr. Caşcaval received an M.S. degree in computer engineering from Technical University, Cluj-Napoca, Romania, an M.S. degree in computer science from West Virginia University, and a Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign (2000). He is Director of Engineering at Qualcomm Research Silicon Valley, where he is leading projects in the area of parallel software for mobile computing. Previously, he worked at the IBM T. J. Watson Research Center, where he was a Research Staff Member and Manager of the Programming Models and Tools for Scalable Systems Group. His research interests are in parallel systems software, hardware-software co-design, parallel programming models, and compilers. He has more than 40 peer-reviewed publications and 25 patent disclosures. He is an ACM Senior Member.

Siddhartha Chatterjee *IBM Systems and Technology Group, Austin, TX USA (sc@us.ibm.com).*

Dr. Chatterjee received his B.Tech. degree in electronics and electrical communications engineering in 1985 from the Indian Institute of Technology, Kharagpur, and his Ph.D. degree in computer science in 1991 from Carnegie Mellon University. He is Vice President, Strategy and Partnerships, for Systems Software in the IBM Systems and Technology Group. Prior to this, he was director of software strategy and architecture in the IBM Systems and Technology Group, and director of the IBM Austin Research Laboratory, one of IBM's worldwide research laboratories. He has held technical, managerial, executive, strategy, and staff positions during his tenure at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, including technical leadership of the Blue Gene* performance team. Before joining IBM Research, he was a visiting scientist at the Research Institute for Advanced Computer Science (RIACS) in Mountain View, California, from 1991

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.

through 1994, and was assistant and associate professor of computer science at the University of North Carolina at Chapel Hill from 1994 through 2001. Dr. Chatterjee has performed research and published in the areas of compilers for parallel languages, computer architecture, and parallel algorithms. His research interests include the design and implementation of programming languages and systems for high-performance applications, memory hierarchy issues in high-performance systems, and software quality. He is an ACM Distinguished Scientist, an IEEE Senior Member, and a Sigma Xi member.

Hubertus Franke *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (frankeh@us.ibm.com).* Dr. Franke received a Diplom Informatik degree from the Technical University of Karlsruhe, Germany, in 1987, and M.S. and Ph.D. degrees in electrical engineering from Vanderbilt University in 1989 and 1992, respectively. He subsequently joined IBM Research, where he worked on the IBM SP1/2 MPI (message passing interface) subsystem, scalable operating systems, Linux** scalability, multicore architectures, scalable applications, and the Prism architecture and application space. He is currently a Research Staff Member and manager in the Scalable Systems department at the T. J. Watson Research Center and member of the IBM Academy of Technology. He received several IBM Outstanding Innovation Awards for his work. He is an author or coauthor of more than 20 patents and more than 90 technical papers.

Kevin J. Gildea *IBM Systems and Technology Group, Poughkeepsie, NY 12601 USA (gildeak@us.ibm.com).* Dr. Gildea received a B.S. degree in computer science from the University of Scranton and M.S. and Ph.D. degrees in computer science from Rensselaer Polytechnic Institute. He is an IBM Distinguished Engineer responsible for high-performance computing (HPC) software development and is leading IBM's Productive Easy-to-use Reliable Computing System (PERCS) project funded under DARPA's HPCS program. Dr. Gildea has 20 years of experience developing HPC software, including leading the development of IBM's MPI, Parallel Environment, and GPFS* products. His current interests are enablement of new parallel programming models, high-performance interconnect architecture, system performance, and mitigation strategies for jitter in general-purpose operating systems.

Pratap Pattnaik *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (pratap@us.ibm.com).* Dr. Pratap is an IBM Fellow and is currently the Senior Manager of the Scalable Systems Group in the IBM Research Division. Over the past ten years, he and his team have developed a number of key technologies for IBM's high-end servers. His current research work includes the development and design of computer systems, including system and processor hardware, operating systems and autonomic components for IBM's UNIX** and mainframe servers, and theory of computing. In the past, he has worked in the field of parallel algorithms for Molecular Dynamics, solutions of linear systems, quantum Monte Carlo and theory of high-temperature superconductivity, communication subsystems, and fault management subsystems. He also has more than ten years of research experience in various aspects of integrated circuit design and fabrication, silicon processing, and condensed matter theory.

Authorized licensed use limited to: SIMON FRASER UNIVERSITY. Downloaded on August 20,2021 at 18:51:46 UTC from IEEE Xplore. Restrictions apply.