Operating Systems Should Manage Accelerators

Sankaralingam Panneerselvam and Michael M. Swift
Computer Sciences Department
University of Wisconsin, Madison, WI
{sankarp,swift}@cs.wisc.edu

Abstract

The inexorable demand for computing power has lead to increasing interest in accelerator-based designs. An accelerator is specialized hardware unit that can perform a set of tasks with much higher performance or power efficiency than a general-purpose CPU. They may be embedded in the pipeline as a functional unit, as in SIMD instructions, or attached to the system as a separate device, as in a cryptographic co-processor.

Current operating systems provide little support for accelerators: whether integrated into a processor or attached as a device, they are treated as CPU or a device and given no additional consideration. However, future processors may have designs that require more management by the operating system. For example, heterogeneous processors may only provision some cores with accelerators, and IBM's wire-speed processor allows user-mode code to launch computations on a shared accelerator without kernel involvement. In such systems, the OS can improve performance by allocating accelerator resources and scheduling access to the accelerator as it does for memory and CPU time.

In this paper, we discuss the challenges presented by adopting accelerators as an execution resource managed by the operating system. We also present the initial design of our system, which provides flexible control over where and when code executes and can apply power and performance policies. It presents a simple software interface that can leverage new hardware interfaces as well as sharing of specialized units in a heterogeneous system.

1 Introduction

For many years, processor performance improved as predicted by Moore's law [17]. The recent decline in Dennard's scaling [6] motivated the creation of many-core processors to reduce power consumption. However, the continued rise in transistor density has provided processors with more transistors than they can use simultaneously.

This situation motivates the use of *accelerators* to further improve performance. Accelerators are fixed- or programmable-function hardware that improves power or performance significantly for a small set of codes. While common for widely used functions, such as floating-point computation and video decoding, accelerators are receiving more interest because they promise an efficient use for the transistors becoming available on future processors.

Accelerators can take on many forms. At the finest granularity, specialized instructions, such as SIMD or CRC32 support in Intel x86, provide accelerated computation over small data items at low latency. In contrast, coarse-grained accelerators may be accessed through a kernel-mode device driver, as in cryptography accelerators in Sun/Oracle Niagara processors and H.264 video encoders for mobile devices [22]. Recent years have seen a flurry of accelerator architectures, from GPUs for offloading data-parallel computations [15] to specialized units like c-cores [21] and DySER [10] to shared accelerators in IBM's wire-speed processor [9].

Currently, accelerators are ignored by the operating system: the OS is unaware that a computation can execute either on a general purpose core or an accelerator, and provides no assistance in finding the "best" place for a computation. For example, a program that can either use a shared accelerator or execute on a CPU may choose to execute on the CPU for lower latency rather than wait for more efficient execution on the accelerator. In addition, processors that provide direct access to accelerators from user mode, such as wire-speed, may suffer from contention without OS involvement.

We propose that operating systems should abstract and manage accelerators, rather than leaving it up to compilers, runtimes, and drivers. First, many proposed accelerator systems are inherently asymmetric in that not every core is provisioned with identical accelerators. In a general-purpose system, the OS must manage contention for limited accelerator resources. Second, it may be useful to choose at runtime between execution on an accelerator and on a CPU. Supporting this capability requires the OS make accelerator usage information available. Finally, OS abstractions designed for accelerators can also be applied to heterogeneous systems by treating a CPU as an accelerator and scheduling or allocating its use. This provides a unified framework for all forms of hardware acceleration.

In the remainder of this paper, we first discuss motivating hardware features and analyze the different classes of accelerators and the system challenges posed by accelerators. We then discuss the design of our proposed system.

2 Motivation

Our work is motivated by the increasing interest in accelerators from the hardware community and the lack of support for accelerators in the systems community.

2.1 Rise of Accelerators

The choice of providing accelerators in the package is attractive for two reasons (i) they are power efficient (ii) with the increasing transistor count, all of them cannot powered on simultaneously [3, 5].

Accelerators provide dramatically better efficiency than processors. Compared to custom-designed ASICs, a CPU may be 500 times less energy efficient due to costs such as instruction fetch and programmable data paths [12]. Thus, specialized units within processors can be used to execute specific functions efficiently.

In addition, Moore's law continues to provide additional transistors, even though processors lack the power to use them all concurrently. As a result, general-purpose multicore architectures and even GPUs cannot continue to scale performance because of the limited power available to processors [8]. Thus, there may be ample transistors available to provide accelerators even for uncommon workloads.

2.2 Accelerator Types

There are currently several types of accelerators that require different software interfaces. Table 1 gives the characteristics of three types of accelerators.

Acceleration devices. Shared accelerators are commonly implemented as devices and accessed through memory-mapped or port I/O instructions from a kernel device driver. For example, the Oracle/Sun Niagara cryptography accelerator requires a kernel device driver for access, as do GPUs for accelerating data-parallel computations. Such accelerators promise the greatest power, as they are freed from the design constraints of executing in the processor pipeline. However, access through the kernel and I/O instructions increases the latency of access and limits these accelerators to coarse-grained computations.

Furthermore, as acceleration devices execute outside the processor, they may not have access to virtual addressing. Thus, invoking the accelerator may require pinning data in memory and translating virtual addresses in advance of launching the accelerated computation.

Co-processors. Several accelerator designs augment the processor pipeline with acceleration logic to offload program logic. A simple example is vector SIMD instructions, which provide data-parallel execution for greater performance and efficiency. More recently, c-cores executes specific application logic with greater efficiency [21], and DySER provides a specialized data path [10].

These accelerators provide low-latency access directly from registers or virtual memory. However, co-processor designs may still contend for power if not all co-processors or cores can be active simultaneously. In addition, heterogeneous system with a variety of accelerators attached to different cores may also experience contention. Finally, programmable accelerators, such as DySER, require a configuration step that may limit its ability to accelerate short code fragments.

Independent cores. Finally, asymmetric or heterogeneous processors can also be considered accelerator-based systems. Rather than specializing hardware to a specific computation, such a system provides a variety of general-purpose cores with different performance and power characteristics. For example, NVidia's Kal-El processor provides a low-power companion core [18]. On such a system, a program may execute faster or with lower power if it switches to a specific core for phases of its execution. Similar to other designs, these systems can experience contention if many processes desire a limited set of cores.

2.3 Lack of OS Support

We examine how the following activities are handled in todays operating systems: (i) task invocation, (ii) virtualization, and (iii) scheduling.

Task Invocation. We refer to the function block or code region that executes on an accelerator, like encrypting a chunk of data as a task. Currently, each accelerator type presents a different interface to the programmer. For example, functional units are accessed via new instructions, while devices require a system call into the kernel. Thus, it is difficult to write programs that can use a variety of accelerators.

Furthermore, it may be desirable to decide at runtime whether to execute a computation on an accelerator or a CPU. If there is contention for an accelerator, it reduce latency to execute code on a general-purpose CPU rather than to wait for the accelerator. In addition, a process may desire to use both the accelerator and idle CPUs simultaneously to further speed execution.

Operating systems do not currently provide runtime support to allow programs to decide whether to use acceleration. Instead, a program may be written or compiled with calls to specific accelerators, as in a GPU.

Properties	Acceleration Devices	Co-Processor	Independent Core
Systems	- Accelerated Processing Unit like AMD Llano processors, GPU on a separate die from the processor cores like Fermi GPUs - Crypto accelerators in Sun Niagara chips	- Consevation Cores - DySER - SIMD units like SSE	- NVIDIA's Kal-El processor - Over Provisioned Multicore System - Scalable Cores like WiDGET, ForwardFlow
Granularity	Suitable for coarse-grained acceleration	Fine-grained acceleration is possible since the instructions are avaiable in ISA to access the special units	Thread is migrated to the special core for performance or to conserve energy
Resource Contention	Multiple applications might want to make use of the devices at any instant	Per-core resource will not incur contention whereas sharing of resource like DySER block, c-cores by multiple cores can incur contention	Multiple threads might want to access the powerful core
Accessibility	Device driver is used to communicate with the device and the required data is transferred to the device's memory through techniques like DMA or zero copy	Special functional units are integrated with the core's pipeline and can be accessed through instructions in ISA. So, no special support is needed from OS	Thread needs to be migrated to the special core and all states are readily available due to cache coherency
Usage Latency	Overhead of transferring data to device memory	Latency of reconfiguring the hardware logic to fit the computational loop. But this can be avoided if reconfiguration is done during compile time	Time taken for migrating thread onto the newer core and additional latency caused by cache misses due to cold cache effect
Capability	They are similar to other external devices accepting instructions from OS and returning back the result. Most of the current devices are not able to run OS code on them	Special functional unit that makes sense in context of some core. It cannot function independently	Regular core that can execute threads, handle interrupts, access system memory

Table 1: **Types of Accelerators**

Virtualization. Several processors provide user-mode access to accelerator devices [20, 9, 4]. These systems provide new instructions that enqueue requests to a shared accelerator. This reduces the communication latency to a great extent since the path through driver/system is completely avoided.

However, direct access from user mode raises several issues with virtualization. First, a shared accelerator must be able to translate virtual addresses from multiple processes, and the OS must provide those translations. Second, a process may be preempted after launching an accelerated computation, and may not be able to receive the output. Thus, the OS must be aware when the computation completes so it can reclaim the process's resources.

Scheduling. Accelerator-based systems raise several new scheduling problems. First, operating systems work best with uniform resources, such as identical cores or identical memory performance. When systems are heterogeneous, as in NUMA designs, the OS must predict future execution patterns in order to optimize performance, and works less well. Accelerator-based systems are fundamentally heterogeneous, as they require mapping a mix of workloads to a mix of accelerators. Scheduler work on ACMP systems has approached this problem where they dynamically decide on the type of core to execute on [13]. But, generalizing this issue to all types of accelerators is difficult because predicting performance on a variety of accelerators may be more difficult.

Second, an accelerator-based systems requires policies to provide fairness and performance isolation for accelerator access in addition to CPU and memory. The OS must decide which processes deserve access to the accelerator and for how long. User-mode access to shared accelerators complicates such scheduling decisions, because the OS cannot interpose on every request.

Third, the OS must provide usage information to inform application of what accelerators are available and for how long. This allows applications to make informed decisions about whether to use an accelerator or rely on direct execution on the CPU instead for low-latency operation.

In addition, device scheduling and sharing is implemented by device drivers. In a contended system where multiple processes desire acceleration, this prevents the OS from imposing a scheduling policy on accelerator access. Recent work on abstractions for GPUs have addressed this problem [11, 19] with mechanisms that apply to other device, that works well for acceleration devices accessed through drivers.

3 Design

We propose a simple accelerator programming model with operating system support. The model treats the use of an accelerator as a function call that can be dynamically dispatched to an accelerator or executed in-line on the CPU. Based on this model, we discuss kernel and runtime support mechanisms for flexible use of accelerators. Figure 1 shows the different components in our system and how they interact.

3.1 Programming Model

The goal of our model is to allow accelerators to be integrated into common programming paradigms with little effort yet provide flexibility on how accelerators are accessed and when they are used. Thus, we treat accelerator invocation as a non-blocking procedure call, similar to a parallel function invocation in Cilk [2]. The calling function may block, if the accelerated function can execute on the local core, or may return immediately. The calling function can then wait for the computation to complete, similar to handling work units in an event driven model or futures in asynchronous systems [16]. The task to be accelerated is scheduled on an appropriate available accelerator unit and the main thread can continue with its execution or wait for the results.

This model provides great flexibility in how accelerators are invoked, as the mechanism is hidden behind a function call. Furthermore, an accelerated procedure may have multiple implementations depending on the accelerators available, which can be selected dynamically. Furthermore, the abstraction is simple enough that it fits many uses of accelerators, such as cryptography libraries invoking a kernel-mode driver. Finally, it can leverage existing parallel runtimes, to provide synchronization and scheduling.

The PTask dataflow model assumes coarse-grained tasks that must be executed completely by accelerators [19]. In contrast, our model identifies the right execution resource to choose based on the system condition and also the task properties. Thus, both these models can co-exist to provide better benefits to the system.

3.2 Accelerator Stub

The procedure call a program makes to invoke an accelerated function does not directly execute the function. To provide a dynamic choice of how to execute the task, our system interposes *accelerator stubs* on every invocation. The responsibility of the stub is to (i) select the best implementation of the task, either with an accelerator or natively, (ii) pass data to the implementation, and (iii) implement synchronization mechanisms if necessary to block the caller until the result is available. An accelerator stub is functionally similar to an RPC stub [1], which similarly dispatches a function to execute elsewhere.

Stubs abstract the presence of different execution resources present in the system and instead expose a single procedural interface to applications. Furthermore, the stub enables the system to make online decisions of which implementation to choose based on power and performance considerations. For example, a simple policy would be to allow a web server with high priority to make complete use of a cryptography accelerator, but allow its use by other tasks when idle.

The mechanism for selecting the implementation of a task is called *binding*, and may occur early, when the program loads, or late, when the task is invoked (or later). For example, a program may link its stubs against implementations that invoke an accelerator when the program loads, and all subsequent invocations of those tasks will use the accelerator. If, however, binding is deferred until call time, then the program can choose on every call whether to use an accelerator or execute natively. This decision can even be based on the parameters to the function. Binding can be deferred further if tasks enqueued, because the choice of implementation can be made when executing a task rather than submitting it. The stub can also send tasks directly to the accelerator if it supports direct user-mode access from applications.

3.3 Accelerator Agent

Every accelerator has an *agent* that manages the accelerator. In the case of an acceleration device, it may be a driver that communicates with the device. For a co-processor, it may be a thread scheduled on the core with the accelerator. The role of the agent is to (i) provide mechanisms to bind programs to the accelerator and create communication channels, (ii) expose accelerator usage to the OS, to guide policy decisions, and (iii) implement scheduling decisions for the accelerator based on OS policy.

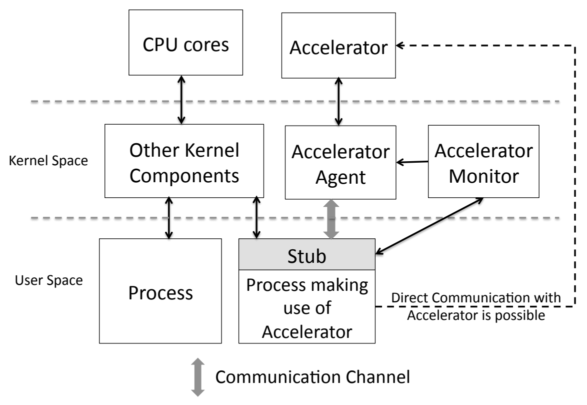

Figure 1: **System Design**

Before using an accelerator, a program must establish a communication channel to the accelerator. For example, many drivers establish a ring buffer of requests in a shared memory region to communicate with a device. A program establishes a connection with the agent in advance of using an accelerator, and the agent performs access-control checks and notifies the OS that the program is interested in using the accelerator. To ensure security and that one process does not interrupt another process's communication channel, an agent creates a separate channel for each requesting process.

Once bound, the stubs for an accelerator task can use the communication channel to invoke the accelerator. For a device accelerator, it may make a system call to the agent. For a co-processor or independent core, it may send data to the agent thread running on the accelerator's core or it may ask the agent to migrate the thread to the accelerator core for execution.

The agent provides policies to schedule the accelerator between processes. For a device with a kernel-mode driver, it may decide which queued requests to send to the device, while for a co-processor it may decide which tasks to execute on the desired core. If an accelerator supports direct communication from user-mode, then the agent may need additional hardware support to schedule its use. One possibility is to virtualize communication channels: when the accelerator is in use by one process, the channels of all other processes are disconnected from the accelerator. Alternatively, the agent can act as a proxy for the accelerator. In this scenario, the agent receives requests via shared memory and decides when to pass them to the accelerator. As communication can be overlapped with the accelerator's work, this does not increase latency. In this case, the agent can also decide to assign the accelerator to a single application if there are no competing requests from other applications.

Agents aid in virtualization by providing the mechanisms to map virtual memory for accelerator devices and to monitor communication channels. Thus, if a process is preempted while using an accelerator, the agent can take responsibility for eventually delivering the result.

3.4 Accelerator Monitor

The *accelerator monitor* is a centralized kernel service responsible for monitoring and scheduling access to multiple accelerators. Thus, it provides the global policy that decides which process should have access to an accelerator and when. It notifies agents of which processes should receive access, and it is up to the agent to put the policy into practice.

The monitor is responsible for system-wide energy and performance goals. It tracks accelerator usage by interrogating agents about their recent use. This provides information about the utilization of accelerators, and can be exposed to applications to help them choose whether to use an accelerator. Thus, the monitor acts as an online modeling tool that can return dynamic information such as queuing delay for the accelerators. This helps the system to decide on a better execution resource on which to schedule the task at that point of time.

4 Related Work

Past work on OS support for accelerators has targeted a narrow set of accelerators. PTasks [19] and Pegasus [11] treat the GPU as first class resource and provide scheduling policies to ensure fair sharing among applications. However, these systems assume that GPUs must be used for the desired computation, and do not provide support for dynamically choosing between GPU or CPU implementations of a computation.

Frameworks like Merge [14] and Harmony [7] provide runtime support to incorporate different task implementations on different accelerators. However, they make decision based on the application parameters, such as task granularity. Thus, they do not consider other users of acceleration hardware that might be present in multiprogrammed systems.

Recently, there have been growing interests in designing hardware interfaces to target the segment between fine-grained and coarse-grained acceleration like wire-speed processor [9]. This architecture is a good fit for our design, as it provides shared accelerators with user-mode access, and thus requires additional support from the OS to manage contention from multiple clients.

5 Conclusion

Accelerators provide opportunities to achieve power efficiency without hurting performance. While different solutions exist for different type of accelerators, a common interface to leverage multiple accelerators is not available. Moreover, current systems cannot ensure performance isolation for accelerators. We propose a simple procedural interface to accelerators that can dynamically select an implementation at runtime. An accelerator agent abstracts the accelerator to the operating system, allowing it to participate in scheduling and resource allocation decisions. Finally, the accelerator monitor enforces global properties such as fairness, performance isolation, and power efficiency. With these mechanisms, the difference between regular cores and accelerators will be blurred, and new accelerators can be gracefully integrated into existing code.

Acknowledgements

We thank the anonymous reviewers for their feedback and also like to thank Mohit Saxena and Vijay Chidambaram for their comments on the earlier drafts of the paper. This work was supported by NSF Award CNS-0834473. Swift has a financial interest in Microsoft Corp.

References

[1] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2:39–59, February 1984.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 207–216, Aug. 1995.

[3] S. Borkar and A. A. Chien. The future of microprocessors. *CACM*, 54:67–77, May 2011.

[4] Cavium Networks. OCTEON II CN68XX Multi-Core MIPS64 Processors. http://www.cavium.com/OCTEON-II_CN68XX.html.

[5] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A General-purpose Architectural Approach to Heterogeneity and Energy Efficiency. In *Proceedings of the International Conference on Computational Science (ICCS)*, 2011.

[6] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256 – 268, oct 1974.

[7] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High performance distributed computing*, pages 197–200, 2008.

[8] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, June 2011.

[9] 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 Journal of Research and Development*, 54(1):3:1 –3:11, january-february 2010.

[10] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 503 –514, feb. 2011.

[11] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 3–3, 2011.

[12] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, pages 37–47, 2010.

[13] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. of SC2007*, Nov. 2007.

[14] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, 2008.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39 –55, march-april 2008.

[16] E. Lippert. Easier Asynchronous Programming with the New Visual Studio Async CTP. *MSDN Magazine*, Oct. 2011.

[17] G. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan 1998.

[18] NVIDIA Corporation. Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. `http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf`.

[19] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[20] Solarflare Communications. Introduction to OpenOnloadBuilding Application Transparency and Protocol Conformance into Application Acceleration Middleware. `http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_OpenOnload_IntroPaper.pdf`.

[21] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 205–218, 2010.

[22] Z. Wei, K. L. Tang, and K. Ngan. Implementation of H.264 on Mobile Device. *Consumer Electronics, IEEE Transactions on*, 53(3):1109 –1116, aug. 2007.