INTEGRATING ACCELERATORS IN HETEROGENEOUS SYSTEMS

by

JÁN VESELÝ

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Abhishek Bhattacharjee

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

Jan, 2021

ABSTRACT OF THE DISSERTATION

Integrating Accelerators in Heterogeneous Systems

By Ján Veselý

Dissertation Director:

Abhishek Bhattacharjee

This work studies programmability enhancing abstractions in the context of accelerators and heterogeneous systems. Specifically, the focus is on adapting abstractions that have been successfully established to improve the programmability of CPUs.

Specialized accelerators including GPUs, TPUs, and FPGAs promise to deliver orders of magnitude improvements in performance and energy efficiency. However, to exploit these benefits programmers must port existing applications, or develop new ones, that target accelerator-specific programming environments. The availability of established programmability abstractions aids this process and extends the performance benefits to a wider range of applications.

This work presents three cases of known CPU abstractions and studies their suitability for accelerator programming; virtual memory, operating system services, and mapping of high-level languages. I study both suitability in terms of existing operational semantics, as well as design considerations necessary for efficient implementation.

First, I study the mapping of high-level dynamic languages to accelerators. High-level languages, like Python, are increasingly popular with designers of scientific applications with a large selection of support libraries. High-level languages are often used to bind together otherwise highly optimized components to form a complete program.

I use this observation to examine a specific case of cognitive modeling workloads written in Python and propose a path to efficient execution on accelerators. I demonstrate that it is often possible to extract and optimize core computational kernels using standard compiler techniques. Extracting such kernels offers multiple benefits; it improves performance, it eliminates dynamic language features for more efficient mapping to accelerators, and it offers opportunities for exploiting compiler-based analyses to provide direct user feedback.

The second major area of study is the access to system services from accelerator programs. While accelerators often work as memory-to-memory devices, there is an increasing amount of evidence in favour of providing them with direct access to network or permanent storage. This work discusses the suitability of existing operating system interfaces (POSIX) and their semantics for inclusion in GPU programs. This work considers the differences between CPU and GPU execution model and the suitability of CPU system calls from both semantics and performance point of view.

Finally, I examine challenges in implementing virtual memory for accelerators. To avoid expensive data marshalling overhead, accelerators often support unified virtual address space (also called unified virtual memory). This feature allows the operating system to synchronize CPU and accelerator address spaces. However, designing such a system needs to make several trade-offs to accommodate the complexities of maintaining the mirror layout and at the same time matching accelerator specific data access patterns. This work investigates integrated GPUs as a case study of accelerators and identifies several opportunities for improvement in designing device-side address translation hardware to provide unified virtual address space.

Overall, this thesis studies programmability enhancements known from the CPU world and their applications to accelerators. It demonstrates that these techniques adapt well and provide programmability and familiarity to application programmers. Such combination not only opens door to new applications but allows for straightforward acceleration of existing ones, delivering performance benefits of accelerators to a wide range of applications. Proposed extensions to accelerators were implemented and data collected on real systems without any use of system simulators or hardware emulation.

Acknowledgements

I'd like to thank my advisor Abhishek Bhattacharjee for his guidance during my studies. I enjoyed the freedom to study interesting problems of my choosing, combined with nuanced guidance to steer my stubbornness. I'd also like to thank my labmate Binh Pham, who included in his project when I joined the lab and allowed me to observe his research from up close. I'm thankful to my Rutgers lab mates; Guilherme Cox, Zi Yan, Karthik Sriram, as well as the new lab mates at Yale; Nick Lindsay, Ketaki Joshi, and Bowen Huang for lively discussions and challenging questions. Finally I'd like to thank my family and especially Parnian Mokri, for their support and believing in me even when I wouldn't.

Dedication

To my family and friends, both those who supported me from the start and those I found along the way.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Accelerators have become an increasingly popular solution to improve the performance and power efficiency of computer systems. With the slowdown of Dennard's scaling and diminishing benefits from Moore's Law [1], hardware specialization has emerged as a common way to satisfy the ever-growing need for more performance. Modern systems include many accelerators such as GPUs [2–5], TPUs [6–8], FPGAs [9–11], or even connection to quantum computers [12, 13] in order to accelerate computation.

To benefit from improved performance and efficiency of accelerators, applications need to be written to explicitly offload portions of their algorithms [14]. Moreover, operating systems often need to include a specialized device driver to expose accelerator hardware to these applications. Unfortunately, programmers thus face multiple challenges when accelerating their programs. A primary problem is that they often need to map their algorithms to a compute paradigm radically different from sequential CPU execution. Such mapping needs to consider not only algorithm suitability, but also the cost of data movement. For example, a parallel computation can be offloaded to a GPU or a tightly integrated vector unit depending on the vector width and the ratio between compute and data movement. The performance benefit of accelerated code is heavily dependent on the specific hardware configuration and the relative performance between the CPU and the accelerator. These and other software design decisions need to be considered even before any accelerated code is written.

Moreover, programmers need to learn new languages and adapt to a new set of constraints depending on the features of accelerator programming environments [15, 16]. For example, a parallel workload is more difficult to map to a GPU or FPGA if it needs frequent access to IO services. Following decades of programmability improvements of

CPUs (*e.g.* virtual memory, OS services, etc), programmers have become familiar with these abstractions. Abstractions allow software engineers to develop more generic and portable applications. An application can use the same file system services whether the data is located on an HDD, SSD, or a network-attached device. This generality of programming is in stark contrast to hardware-specific programming used by accelerators today.

As a consequence, research has begun exploring the future of OS and compiler driven abstractions to enhance accelerator programmability [17–23]. While they present a compelling start, these efforts are limited in one of two ways; either they provide a restricted set of accelerator extensions [20, 21], or they extend the accelerator programming environment with a specialized version of otherwise more generic abstraction [17, 18]. Both of these approaches still require programmers to understand new sets of semantics and restrictions before they can accelerate their programs. As such, they reinforce the view that the only way to encourage performance efficiency on accelerator code is via specialized and often constrained abstractions. In contrast, this thesis presents an alternative approach and vision:

> *Adapting existing abstractions from CPUs to accelerators is a good way to extend the performance benefits of accelerators to both new and legacy applications.*

The main focus of this work is thus on *integrating accelerators* by adapting known CPU abstractions to improve programmability without compromising performance. This work includes a compiler study that exploits domain-specific information to achieve compatibility of high-level programming language with accelerators, and studies on virtual memory and operating system abstractions to offer programmers the ability to map new applications onto accelerators like GPUs, and profiling of hardware for these abstractions. The included studies also propose enhancements to these abstractions when necessary to enable accelerator programmability, and evaluate the impact on application performance. I study programmability enhancing mechanisms in applications, operating

systems, and hardware. Specifically, I study abstractions and programmability enhancements that proved successful on CPUs, such as virtual memory and operating system services, and consider their adaptation to accelerators. Existing interfaces have the benefit of familiarity, which not only makes writing new applications easier, but it also reduces the programming effort needed to port existing applications to new accelerator platforms. The challenge is not only in making sure the existing semantics do not conflict with the accelerator compute environment, but also that an efficient implementation is possible that wouldn't constrain the performance benefits of using accelerators in the first place.

1.1 Overview of Integration Challenges

An ideal accelerator delivers performance and efficiency benefits with only minimal or no effort on the application side. This effort manifested in the current generation of "single-source" programming languages (*e.g.* C++AMP [20], SYCL [24], OpenMP [21]), that combine both CPU and accelerator code paths. However, these solutions work by extending the base host language with accelerator compatible constructs.

This thesis goes further and in Chapter 2 it presents a mapping of a high-level language to GPU. Python has become a popular language of choice for scientific applications, boasting a wide selection of highly optimized scientific libraries [25–28], many of which can be accelerated [28, 29]. This combination of convenience and programmability allows for rapid development and deployment of scientific applications. I observe that although Python is a highly dynamic language, applications in the scientific domain often only use its dynamism to assemble programs from a collection of optimized libraries.

A traditional approach would be to design a domain-specific language (DSL) that includes all the capabilities of required libraries, as well as enough features to construct final programs. However, developing DSLs is a long and demanding process, generic languages offer quick turn-around time and a faster pace of development, especially if there is a rich selection of specialized libraries available. A specialized library with

ample development resources might include a domain-specific language (*e.g* TorchScript in pytorch [28]), but such effort is out of reach for smaller teams and evolving workloads.

Chapter 2 studies an alternative approach of extracting computational kernels from high-level languages. In particular, I use the domain of cognitive neuroscientific modeling to show that most of Python's dynamism is unnecessary and can be removed using standard compiler techniques. Cognitive neuroscience relies on modeling to study the connection between biological processes in the brain to psychological processes in the human mind. Although the models use floating-point computation at their core, their construction can use multiple specialized libraries, such as numpy [25], scipy [26], and PyTorch [28]. This presents a highly heterogeneous environment that combines traditional high-precision floating-point computation with neural networks and stochastic sampling. The combination of heterogeneous compute elements connected via high-level language and the need for high-throughput computation to build the final distribution is an appealing target for accelerators and a challenge for the software stack. I show that expensive concepts like dynamic typing system and dynamic data structures can be replaced by their static counterparts if the compiler knows the data structures won't change during execution, lowering the demands for memory management operations. The resulting compiled code is not only amenable to hardware acceleration on GPUs, but offers orders of magnitude performance improvement on CPUs.

Python's growing popularity [30] is a testament to its power and ease of use. With a rich selection of libraries to support numerical computation the barrier to writing programs to support scientific effort across domains is getting lower. This work presents a guide on how domain-specific information about the nature of computation can be exploited to achieve both compatibility with current accelerators and a good performance on CPUs.

However, not all programs can be transformed into accelerator-amenable form by the compiler. Many programs require communication with the outside world, whether it is network, permanent storage, or communication with other processes via inter-process communication. As these interfaces continue to develop to include new high-throughput storage and network devices, the advances on the CPU side would transfer to accelerator

environments. This is especially important as high-throughput accelerators like GPUs, and TPUs are well-positioned to take advantage of high-throughput IO. An example in Chapter 3 shows how GPUs can already take advantage of high-throughput SSD devices.

Chapter 3 studies system services in Linux and their suitability for GPU execution environment. This work is not the first to notice potential benefits of enabling GPU access to system IO [17–19, 31]. However, it demonstrates that systems services beyond IO can be useful for GPU workloads. I find that almost 80% of the system services available in Linux map well to GPU execution environment. I propose and implement a highly efficient system call invocation mechanism that allows GPU kernels to directly invoke system services. Availability of system services allows not only a more straightforward path towards accelerating existing applications, but it also facilitates development of system call heavy applications that previously wouldn't be considered suitable for acceleration.

Chapter 2 and 3 aim to bridge the gap between programs in need of acceleration and capabilities available to them. The former removes unnecessary constructs that would be difficult to map and allows the extracted kernels to use accelerators that do not provide memory management capabilities. On the other hand, interactions with the outside world through operating system services cannot be removed without modifying program semantics. The latter thus focuses on providing accelerators with the necessary abstractions that expose operating system services. A combination of these two approaches opens opportunities to accelerate programs that would otherwise be restricted to running on CPUs. Expanding the amount of code that can be accelerated exacerbates the issue of data sharing. Not just between CPU and one accelerator, but also between multiple different accelerators.

Chapter 4 studies virtual memory abstraction for accelerators in heterogeneous systems. Virtual memory is a powerful abstraction that presents a consistent view of system memory to applications, irrespective of specific machine parameters. It is crucial for efficient and safe multiprogramming and data sharing between applications. As large parts of applications move to accelerators, preserving efficient virtual memory abstraction will

be crucial to maintaining the programmability benefits while enjoying the performance benefits of accelerators.

More specifically, Chapter 4 studies an existing implementation of unified virtual address space (UVA) in CPU-GPU heterogeneous systems. A unified virtual address space provides an accelerator with the same view of application memory as the CPU. Presenting the same memory layout as the CPU enables easier sharing of data and allows accelerators to directly operate on pointer-based data structures like lists and trees. Without UVA, programmers have to rely on expensive data *marshaling and demarshaling*, or cumbersome workarounds that replace pointers with array indices. Accelerators can also benefit from other features provided by virtual memory, such as demand paging to improve memory utilization and improve support for multi-programmed workloads. This work shows that designing an efficient address translation mechanism still remains a challenge, exacerbated by the throughput oriented nature of GPU computation. The evaluation shows that mismatch between expected data access patterns and those observed in real workloads can lead to severe overheads, and a robust accelerator specific address translation mechanism is necessary to deliver the benefits of shared virtual address spaces.

Overall, the studies presented in this thesis cover every level of system design relevant to accelerator integration. Specifically, Chapter 3 and Chapter 2 propose software-only solutions that improve both programmability and performance of accelerated applications, highlighting significant opportunities that exist in the software stack.

1.2 Thesis Organization

The overall structure of this thesis is presented in Table 1.1. A study on accelerating Python-based models in cognitive neuroscience, under submission to a computer architecture conference, is presented in Chapter 2. A novel implementation of generic system calls for GPUs, presented at ISCA'18, is included in Chapter 3. Finally, a study on the performance and overall system impacts of unified virtual addressing for GPUs, that was presented at ISPASS'16 is included in Chapter 4. A more detailed description of

the contributions follows.

Integration level	Mechanism	Proposal	Chapter
Application	Kernel Extraction and Compilation	PsyLang	2
OS	System Calls	GENESYS	3
HW	Unified Virtual Addressing	N/A	4

Table 1.1: Overview of thesis organization

1.2.1 Accelerating and Analysing Cognitive Models Using Compiler Techniques

The first study in this thesis presents a case study in accelerating applications written in a high-level language. It shows that a combination of domain-specific information and standard compiler techniques can be used to bridge this gap. I observe that in the field of cognitive modeling, the models are often composed of individual components with a high-level scripting language, like Python, directing the data-flow and control-flow decision. Python owes much of its popularity [30] to ease of use and programmability, however, it is also known for relatively low performance and multiple projects use compiler techniques to improve performance [32–36]. I further observe that cognitive models do not need all dynamic features of the Python language and efficient application of domain-specific knowledge can achieve both significantly higher performance on CPUs, as well as target accelerators such as GPUs.

I propose and implement *PsyLang*, a compiler frontend translating cognitive models to LLVM IR. *PsyLang* extracts computational kernels from Python constructed models and eliminates costly dynamic features. In doing so, it achieves up to four orders better performance on CPUs, and allows acceleration of cognitive models on GPUs. This effort presents an alternative to domain-specific languages. Unlike designing a DSL, *PsyLang* relies on programmers' tendency to prefer certain structures in their programs, rather than restricting the programming environment to said structures by design. A compiler optimization effort can be easily extended to support new constructs if needed. At the extreme end, this would lead to a complete compiler for Python. However, *PsyLang* demonstrates the significant performance opportunities in exploiting

restrictions naturally present in programs of a specific domain.

Beyond improved performance, I discuss language features that cannot be eliminated by compiler optimizations. Choices such as numeric precision, and algorithms used for random number generation cannot be changed without affecting program semantics, yet they have a significant impact on accelerator performance. These optimizations can still be exploited to achieve better performance, but they might require re-validation of model results and parameters, and thus need human intervention.

Transparent integration of accelerators in high-level languages extends the performance and efficiency benefits of accelerators to applications that would otherwise have to invest significant resources to achieve comparable performance improvements. At the same time, it extends programmability benefits of high-level languages to accelerators opening paths to higher performance to even more applications.

1.2.2 System Services for GPUs

The second study presents *GENESYS* – an efficient system call invocation framework for GPUs. *GENESYS* enables efficient communication between a GPU kernel requesting system services and the privileged OS code running on the CPU that handles these requests. It goes beyond previous work that focused on network and storage IO [17–19, 37], and enables the implementation of system calls in areas such as memory management and inter-process communication.

Acknowledging the parallel nature of GPU execution I study the impact of *invocation granularity*. When invoked on a GPU, a system call can represent service requested by a single GPU thread, a group of threads, or an entire GPU kernel. Smaller granularity preserves the independence of execution in individual GPU threads, however, it also suffers higher communication overhead. Overall, group-level invocation proves to be the most flexible but some situations benefit from either thread-level or kernel-level system calls.

To evaluate *GENESYS* I implement a suite of micro-benchmarks, and characterize the performance aspects of system call invocation and processing. Further, *GENESYS* demonstrates the benefits of improved GPU programmability by designing a set of novel

heterogeneous applications. The targeted workloads include well-known applications like *memcached*, *grep*. The former can benefit from the parallel nature of GPUs, but the acceleration efforts were constrained by its high network communication requirements. *GENSYS* allows integration of network communication as well as parallel lookup on a GPU, achieving $30 - 40\%$ improvements in both latency and throughput. The latter application, *grep*, has been considered IO-bound and not suitable for GPU acceleration. I demonstrate that not only is the acceleration of *grep* low effort, but the parallel nature of GPUs helps extract better throughput from modern SSD storage devices than even multicore CPUs can.

I complement the performance and programmability study by analyzing nearly 300 Linux system calls in their POSIX-like semantics, and assess their suitability for the GPU programming environment.

1.2.3 Challenges and Opportunities in Designing Virtual Memory for Accelerators

The final study evaluates a commercial heterogeneous CPU-GPU system, analyzing the impact of address translation on GPU performance. A combination of micro-benchmarks and real-world applications characterizes the address translation subsystem of GPUs. This characterization reveals significantly higher address translation latency compared to both CPU address translation and GPU data accesses and the throughput oriented design of the translation hardware. A set of GPU applications is classified based on the overheads of address translation and the impact of different page sizes on address translation efficiency. This classification highlights the role of GPU local memories in determining the applications' memory access patterns. Studying these applications further reveals that existing address translation optimizations target streaming and linear access patterns which exacerbates address translation overheads for applications with irregular memory access patterns. Moreover, profiling divergent workloads shows that data divergence has a significantly larger impact on the address translation performance than the rest of the memory hierarchy. Beyond address translation, this work characterizes the costs of TLB invalidation routines and inspect the added benefit of demand

paging. The study further evaluates the impact of GPU demand paging on CPU and the rest of the system, noticing its potential to reduce memory usage, but also increased costs caused by additional CPU-GPU communication overhead. Based on these observations this work proposes a set of recommendations to guide future research on address translation for GPUs and accelerators.

1.3 Published Work

Studies presented in this thesis focus on the integration of programmability enhancing techniques and accelerators. Observation and opportunities in architecting virtual memory and unified virtual address spaces to GPUs were presented at ISPASS'16. The work on providing GPU programming environment with access to generic system calls was presented at ISCA'18 and the study on accelerating models in high-level languages is under submission to an architecture conference.

Moreover, research on virtual memory abstractions contributed to other programmability studies. *Large pages and lightweight memory management in virtualized environments: can you have it both ways?*, presented at MICRO'15, proposed a mechanism to address the growing overhead of virtual memory abstractions in virtualized environments. Similarly, *Hardware translation coherence for virtualized systems*, presented at ISCA'17, studied the overheads of maintaining coherent address translation mappings in virtualized environments. Similarly, research on accelerators and benefits of specialization contributed to accelerator-centric design proposed in *Hardware-Software Co-Design for Brain-Computer Interfaces*, presented at ISCA'20.

Chapter 2

Accelerating and Analysing Cognitive Models Using Compiler Techniques

2.1 Introduction

Cognitive neuroscientific models seek to explain the neural mechanisms underlying the psychological processes responsible for human cognitive function, including the dynamics of perception, attention allocation, decision making, cognitive control, and learning [38–42]. In addition to gaining deeper insights into how the brain gives rise to the mind, a long-term goal of such modeling is to provide a better scientific grounding for the diagnosis and treatment of psychological disorders and trauma [43–45].

Models from cognitive neuroscience, and neuroscience more broadly, can also help design human-like artificial intelligence. Consider artificial neurons [46] and their learning algorithms (*e.g.*, Hebbian learning [47] and back propagation [48, 49]), and concepts like lateral inhibition [50, 51] that have made their way from neuroscience to artificial intelligence. More recently, Google's DeepMind has incorporated knowledge of complementary learning systems in the hippocampus and neocortex (reflecting the cognitive functions of episodic and semantic memory, respectively) in its design of AI for gameplay [52]. Looking ahead, cognitive models are expected to assist the development of meta-learning by offering guidance on learning algorithms that can induce learning algorithms themselves (*i.e.,* learning to learn) [53–57].

However, efficient execution of congitive models faces several key challenges – they can be computationally intensive, heavily composited, and highly heterogeneous in their computational requirements. *We show how compilation techniques can be used to accelerate and analyze cognitive neuroscience models.* In so doing, we address these key

challenges. Figure 2.1 shows an example cognitive neuroscience model[1], the "predator-prey" game, a virtual pursuit avoidance task that provides a simplified version of models that address the role of cognitive control in the deeply evolved skill of chase and escape behavior. This game is used to measure perception, attention allocation, and action selection, and has been used previously to compare intelligent agents against non-human primates [58]. A neural network processes visual data to extract the position of the player, predator, and prey avatars on a computer screen. These positions are perturbed with random number generators that are mitigated by the allocation of attention to an avatar; allocation of attention carries a cost that constraints the total amount of attention that can be paid to all avatars, requiring a control decision about how much allocation should be paid to each. A control mechanism implements optimization strategies for determining how much allocation to attend to each avatar to maximize the likelihood of success while minimizing attentional costs. Cognitive models like the predator-prey task face performance and analysis challenges.

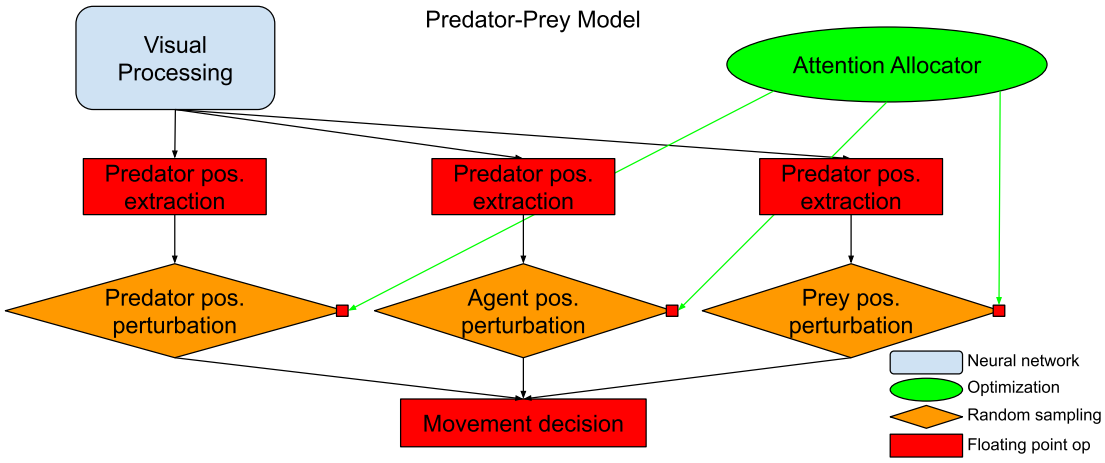


Figure 2.1: An example cognitive model for the predator-prey task, a virtual avoidance game for chase and escape behavior.

Performance: The greater the biological sophistication of cognitive models, the longer their execution time; *e.g.,* we have found variants of the predator-prey model to take one week of runtime. The problem is their reliance on high-level languages like Python. Python's flexibility makes it well-suited for modeling environments that need composited

[1] A demonstrator that combines several features based on real model presented later in this work

assembly of plug-and-play components, and it offers a rich library of highly optimized modules for scientific computation [25–28]. However, its interpreted nature and dynamic typing system incur high performance overhead, while its global interpreter lock precludes easy multi-threading. In §2.5, we show that existing JIT-based approaches to accelerate state of the art Python-based cognitive models [34, 35] achieve only 15% speedup in the best cases, and usually far less.

One might also consider using hardware accelerators to exploit the inherent heterogeneity of cognitive models, but again practical considerations stand in the way. Ideally, the neural network in Figure 2.1 would run on a GPU or neural network accelerator [6, 59], perturbation steps would use stochastic accelerators [60], attention allocation would run on a GPU or optimization accelerator (*e.g.,* a quantum annealer [61]), etc. In our work, we show that accelerating individual components in isolation with transitions from/to Python is insufficient, and we need holistic accelerator-level parallelism instead [62]. The challenge is that different accelerators generally use distinct specialized runtimes and suffer high overheads when passing data amongst one another [63].

Another impediment to accelerator-level parallelism is that neuroscientists are interested in questions like – How long did an agent take to make a correct versus incorrect decision? When was a particular component in the brain activated? These model dynamics result in the generalized execution algorithm in Listing 2.1, where control-flow is a fundamental part of the model, with control-flow decisions often reported as model outcomes. This makes it difficult to design straightforward data-flow acceleration solutions as the model's control and dataflow graph (CDFG) is similar to Figure 2.2. Execution of each model component updates global state and control always returns to the central scheduler, obfuscating opportunities for offloading and accelerating component sequences.

```
def wrap_exec(model,inputs,params,state):
    while not model.is_finished(state,params):
        for n in model.nodes:
            if n.can_execute(params,state):
                n.wrap_exec(inputs,params,state)
```

Listing 2.1: Simplified model execution algorithm that shows the impact of a centralized scheduler managing control flow.

Analysis: In addition to performance, the composited nature of cognitive models leads

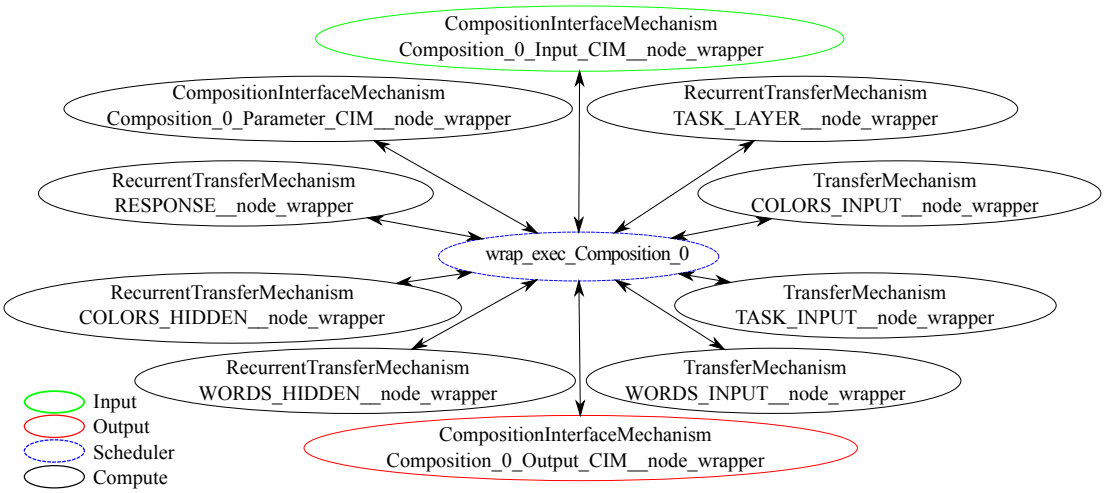


Figure 2.2: A CDFG of the model code that captures the impact of the centralized scheduler shown in Listing 2.1.

to ever-increasing growth in model complexity as well as parameter types and counts. This rapid scaling makes it difficult to reason about and build larger models even when performance may allow it. For example, when constructing the predator-prey task in Figure 2.1, modelers would benefit from knowing information like value ranges passed along edges or calculated between nodes. They would also benefit from automated approaches that highlight similarities between newly developed models and libraries of existing models. One might expect such relationships to be obvious to modelers on a small scale, but providing automated analyses of model structures can lead to better model development, more frequent model reuse and ultimately, a better understanding of the underlying mechanisms and their interactions, especially as models become more complex. Model analysis will become vital with the development of common model exchange formats that are enabling scientists to fuse models designed by distinct third-party researchers for their research [64].[2]

Research contributions. We realize *Psylang*, a compiler frontend to LLVM [66] that uses domain-specific information about cognitive models to extract and generate efficient portable code and enable automated analysis of brain models. Our key insight is that this is possible by reconciling the CDFG generated by the compiler front-end with

[2]This is akin to the use of common exchange formats in machine learning and deep learning, like ONNX [65].

the original model graph. *Psylang* encodes domain-specific information about cognitive modeling so that the model's scheduling rules can be inferred statically, thereby allowing conversion of the CDFG from Figure 2.2 into a graph that more closely resembles the model in Figure 2.1. While *Psylang* may not completely elide translations to/from the central model scheduler and reconstruct the model graph, particularly when aspects of the model control flow cannot be determined statically, it often comes close. This has several benefits, which we evaluate by integrating *Psylang* into *PsyNeuLink*, a state of art open-source cognitive modeling environment [67]:

- *Generation of highly efficient CPU code.* We achieve up to $20K\times$ speedup versus Python baselines of cognitive models as well as JIT-enabled Python implementations (PyPy [34] and Pyston [35]) on an Intel i7-8700 CPU core.

- *Accelerator offload.* We achieve another 3.3× speedup over our CPU-optimized code using LLVM's code generation backend for GPUs. Such accelerator-level parallelism was previously not exploitable in environments like *PsyNeuLink*.

- *Repurposing LLVM passes for model analysis.* We use a combination of automated code translation and manual implementation to efficiently represent the original cognitive model in a standard compiler IR. We then repurpose standard compiler analyses like value range propagation [68], scalar value evolution [69], and dead code elimination to generate useful user-oriented feedback to the modeler. We use monomorphic code generation and line debugging information to link generated code back to the original brain model to offer information on values that propagate along model edges or nodes or identify parts of models that are not executed under certain parameter ranges, and more.

- *Repurposing software engineering tools for model analysis.* Via efficient representation in compiler IR, we also reuse software engineering techniques like clone detection [70] to detect similarities between models. Detected similarities between generated functions can be presented in terms of model components or collections of components.

Central to *Psylang*'s design is the fact that cognitive models do not need Python's

dynamic typing system, that hot functions can be inferred before runtime, and that the compiler can leverage domain-specific information to determine scheduling rules that reconstruct the original data-flow of the cognitive model. These observations mean that the compiler can be freed of slow runtime features of Python and the presence of modeling necessities like the centralized scheduler, and can determine the original model structure at compile time. By leveraging these insights, we have integrated *Psylang* within *PsyNeuLink*. *Psylang* is implemented in Python and generates LLVM IR which is then passed to LLVM for compilation and execution. The same IR is also analyzed using combination of existing, extended, and completely new tools.

Broad research lessons: *Psylang* presents a case study of acceleration and analysis of domain specific workloads using generic software engineering tools. As systems embrace extreme heterogeneity, the development of languages and systems software that can productively use hardware enhancements becomes vital. Questions on whether to create new or change and reuse existing software abstractions are first-class research endeavors. We believe that the answer is a delicate balance to deliver on performance and productivity. While the specific decisions we made and detail in this work are tied to the nature of cognitive modeling, we believe that the lessons presented offer a template applicable to other domains. This includes, for example, the lesson that interpreted execution and highly dynamic nature of high-level languages can be barriers to good performance, but that domain-specific information can circumvent these problems. We also show the need for efficient integration of accelerator runtimes, an observation that is likely to be vital for future accelerator-rich systems [62]. Finally, we also show that compiler augmentations likely to be broadly beneficial for accelerator-centric tasks; *e.g.,* for model analysis, we had to extend passes to support floating point computation in a manner likely useful for emerging GPU compute tasks that often require special floating point handling.

2.2 Background

Cognitive neuroscience relies on a wide selection of high-level models that explain the psychological processes behind brain function. These models can include mathematical expressions that describe biologically-accurate modeling of groups of neurons [71, 72], higher-level behavioral models of brain activity [73].Notably, cognitive models differ from neural networks – even though they may include some nodes that implement neural networks – in that they describe our understanding of the human mind rather than an opaque solution to a specific problem. This "explainability" is of interest to not just understand brain function, but to also help build transparent AI decision-making based on an understanding of the human brain rather than a black-box trained on correlations hidden in neural network training data.

High-level languages in cognitive modeling: Cognitive modeling environments have evolved over time in their ability to support large-scale composited models with flexibility. Table 2.1 shows that while some early tools like NEURON, GENESIS, and ACT-R relied on C, C++, or even Lisp, increasingly higher-level language features have become non-negotiable. This is best understood by considering the flexibility and plug-and-play functionality desired by cognitive neuroscientists today. As an example, consider the addition of a memory component to the model in Figure 2.1. This component may be represented as either a biologically accurate model of the hippocampus, a Hopfield network, or a simple key-value store, depending on the desired level of fidelity. This is akin to architecture simulators, where a microarchitectural component can be represented with either a high-level functional simulation or detailed cycle accurate modeling. High-level languages are able to accommodate this level of flexibility naturally.

Consequently, modern cognitive modeling tools include support for scripting by exporting an interface to higher-level languages like Python or providing custom scripting solutions. We focus on *PsyNeuLink* as it is representative of sophisticated modeling environments, and it is the state of the art in single modeling environments that can scale to arbitrary levels of detail based on the modeler's needs. Moreover, *PsyNeuLink*

also supports compositions of sub-models built in some of the other environments; *e.g.,* portions of the models can consist of neural networks built in frameworks like Pytorch.

Tool	Language	Description
PsyNeuLink [67] *Focus of our work*	Python	High-level block modeling with support for sub-models built using frameworks like PyTorch.
ACT-R [73]	Lisp	High-level behavioural modeling that uses symbolic computation.
TensorFlow [74]	C++ Python bindings	Widely used highly optimized neural network framework.
PyTorch [28]	C++, CUDA, Python bindings	Widely used highly optimized neural network framework.
EMERGENT [75]	C++, C-Super Script	Biologically accurate and artificial neural networks.
Human Neurocortical Neurosolver [72]	Python	Neuronal circuit simulation to match EEG and MEG recordings.
NEURON [71]	C, Python bindings	Biologically accurate modeling of neurons, parts of neurons and neuronal circuits.
GENESIS [76]	C, Custom script	Sub-cellular components and biochemical reactions to complex models of single neurons, simulations of large networks, and system-level models.

Table 2.1: Modeling tools used in computational neuroscience.

Cognitive model structure: Each brain center or psychological process in *PsyNeuLink* is represented by a node or *component* in a graph. For example, in Figure 2.1, the visual processing, attention allocation, position, perturbation, and movement decision nodes are components. Components can be either high-level functions, like logistic functions, or entire nested models. Each component also includes input pre-processing, parameter processing, and output post-processing. These assemblies are dubbed *mechanisms*. Each node in Figure 2.1 other than 'Visual Processing', which is a neural network composition, is a mechanism. Edges between nodes are *projections* and reflect not only I/O relationships, but can also perform arbitrary transformations of the data along the way. For example, Figure 2.1 presents the 'Extraction' nodes for structural clarity, but the same computation and reduction in data dimension can be done in projections.

A critical part of modeling environments like *PsyNeuLink* – and the biggest challenge

in enabling model acceleration and automated analysis – is its scheduler. The scheduler allows modelers to specify when each component should run via a collection of scheduling rules. The rules can be as simple as "*run component X once after component Y has run*" to more complex ones like "*run component X once after component Y has settled and component Z has run at least N times*". Table 2.2 presents examples of scheduler rules in *PsyNeuLink*. The nature of scheduling rules and whether they depend on processed data, determine whether a static schedule exists. Static schedules are useful because they enable analyzing computational sequences rather than each component in isolation. They are also the targets that *Psylang* aims to eliminate in order to bring the compiled CDFG as close as possible to the model graph to facilitate performance and analysis.

Rule	Semantics
$afterNRuns(other\_node, n)$	A node can execute in every iteration after *other_node* executed at least n times
$everyNRuns(other\_node, n)$	A node can execute in this iteration if number of *other_node* executions is divisible by n
$isFinished(other\_node)$	A node can execute in this iteration if *other_node*'s $finished$ flag is set.
$and(r1, r2)$	A node can execute in this iteration if both $r1$ and $r2$ are satisfied.
$or(r1, r2)$	A node can execute in this iteration if either $r1$ or $r2$ is satisfied.
$not(r1)$	A node can execute in this iteration if $r1$ is not satisfied.

Table 2.2: The first two scheduling rules are static (*i.e.,* independent of processed data), the third is dynamic (*i.e.,* data-dependent). The last three are either static or dynamic, depending on the combined rules.

Evaluated models: Table 2.3 summarizes the set of cognitive models that we use to drive *Psylang*'s design and evaluation. The experiments used to construct these models are shown in Figure 2.3. Our choice of models captures several key characteristics of cognitive modeling tools. First, all of these models are composited and heterogeneous. There is therefore no single constituent kernel that can be accelerated to achieve good performance; instead, the entire model *is* the kernel. Second, these models include stochastic elements and need to be run thousands or millions of times to build sufficiently detailed distributions of results. This means that it is desirable that any single run

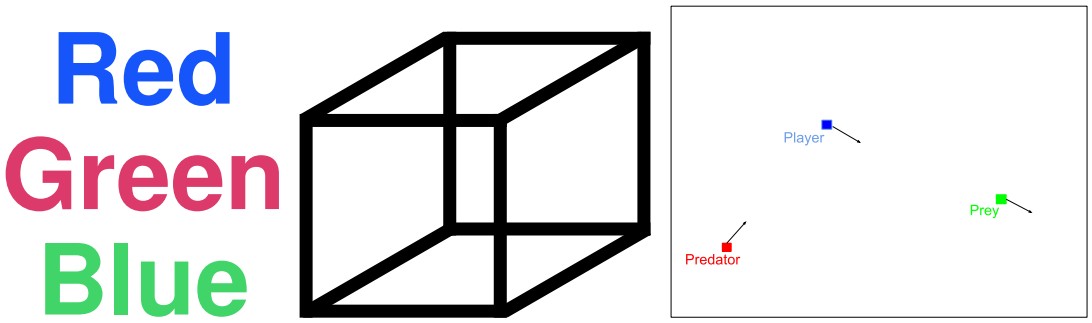
Model Name	Section	Modelled effect	Computation
Stroop model	2.2	Conflict in shared representation of colour	Time progression of accumulated stimuli (and thus time to make a decision) for aligned and conflicting conflicting inputs. The baseline model used in our work is configured to run 1500 time steps.
Necker cube	2.2	Conflict in perceived orientation of a geometric object	Repeated adjustment of perception stimuli strength in favour of one orientation or the other. For our studies, we use a full model with all eight vertices, a hand-optimized/vectorized version of the same model, and a simplified version of it that uses only three vertices.
Predator-Prey	2.2	Attention allocation and decision making in a simple game	Search for optimal parameters to allocate attention between three game avatars. In our work we consider variants with 2, 4, 6 and 100 levels of attention.
Multitasking	2.2	Conflict monitoring in task decisions	Input processing using neural network feeding into *LCA*

Table 2.3: Cognitive models used to drive *Psylang*'s design and evaluation.

completes in fractions of a second of wall clock time. Third, these models benefit from acceleration not only in reducing wall clock time but by being amenable to improvements in biological plausibility and sophistication in model. Fourth, these models are highly composited and interact with other frameworks like Pytorch. And finally, partly due to this composition, they are sufficiently complex that they are useful demonstrators of the types of automated analysis that would be essential on models of even large scale.

Botvinick Stroop test

Figure 2.3a shows an example of the Stroop effect test, which demonstrates conflict in the brain's representation of color emanating from visual and language stimuli [77]. We run a *PsyNeuLink* implementation of the Botvinick version of the Stroop test [78]. The model graph, shown in Figure 2.4a, uses the strength of font color and word meaning as inputs, and the underlying conflict in representation of color in human mind. The model output is 'decision energy' and the response time needed for the energy level to cross a



(a) Stroop test (b) Necker cube (c) Predator prey

Figure 2.3: Experiments used to construct the cognitive models used for evaluations in this work.

threshold and for subjects to perform the assigned task. The purpose of this model is to predict response/reaction times which can then be compared to experimental results. Because each time step of this model updates decision energy, improved performance permits simulation with smaller time-steps for higher fidelity results.

Necker cube

Figure 2.3b shows an optical illusion first described by L. Necker [79] and an example of the notion of bi-stable perception, where the observer experiences changes in perception of the cube's orientation. The model graph, shown in Figure 2.4b, represents each cube vertex and tracks its association with one of the two possible orientations. The model predicts the time taken for perception to switch from one orientation to another. Improved performance allows scientists to reduce the size of time step, and run models with higher fidelity, or consider objects with more vertices.

Predator-prey game

Figure 2.1 presented a sophisticated version of a predator-prey task, shown in Figure 2.3c, that we consider in our evaluation. A blank canvas includes avatars for the player, a predator, and a prey, with the goal of the game being to catch the prey while avoiding the predator. However, the input locations are perturbed before being processed and the amount of perturbation is inverse to the amount of attention paid

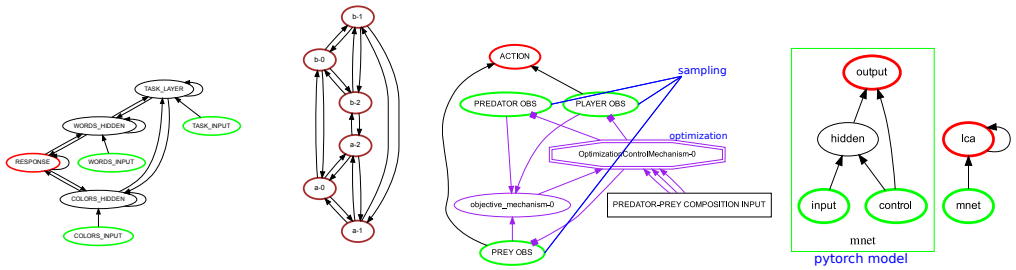
to each avatar. At each time step the model searches for the best attention allocation considering the location of next move, and the costs associated with maintaining or changing attention levels. The optimization step considers attention allocation and runs the rest of the model to evaluate fitness, as well as the cost functions. Accelerating this model permits scientists to significantly increase model sophistication by either considering more levels of attention, decreasing the time step size, or both.

Multi-tasking model

Like the *Stroop model*, the multi-tasking model represents conflict in representation. The model graph in Figure 2.4d shows that the inputs are colours and shapes associated with a specific task. A neural network is used as an input to identify color and shape, with this information passed to a *Leaky Competing Accumulator (LCA)* component [80]. The *LCA* returns the time taken to reach a decision, and the decision outcome. Repeated runs provide a distribution of times and correct versus incorrect decisions. This model differs from the other examined models by using a neural network implemented in *PyTorch* to process input colours and shapes. It thus represents a class of heterogeneous models that span multiple execution environments (*PyTorch* and LCA in this case). Morover, accelerating this model enables use of smaller time steps, more inputs, and the interference of earlier results on later decisions. It also allows designing models that more realistically represent human behavior, with more than two conflicting inputs and tasks.

2.3 Design & Integration with LLVM

Psylang exploits several observations in order to generate efficient code. The first observation is that the models don't need dynamic type system that Python offers. Thus the first step in compiling cognitive models is type inference to deduce static types used in computation. The second observation is that the 'hot' functions are known in advance and don't have to be determined at runtime. The third observation is that the compiler is able understand static scheduling rules and reconstruct the original data-flow. This



(a) Stroop test (b) 3-vertex Necker Cube (c) Predator-Prey (d) Multitasking

Figure 2.4: Example models, as visualized by *PsyNeuLink* (annotations are in blue). Green nodes denote input, red nodes denote outputs, and brown nodes are both input and output. Purple nodes determine control over other nodes.

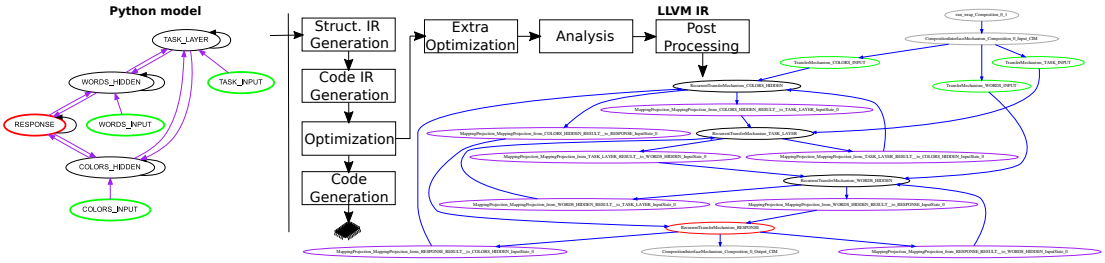


Figure 2.5: *PsyNeuLink* model (Stroop test model) visualization processed through *Psylang* pipeline. Green nodes denote data input, red nodes denote outputs, and purple nodes correspond to projections (edges) in the original graph. Because the Stroop test model uses only static scheduling rules, the original model can be recovered in the form of data-flow graph.

has advantages for analysis that can consider chains of executed components, as well as potential performance benefits.

We choose LLVM as our compiler backend, but GCC offers similar capabilities and would work with only minor modifications to *Psylang*. There are multiple benefits for targeting state-of-the-art production compiler. Beyond the obvious benefits in targeting multiple architectures (*Psylang* can run on x86,arm64,armv7,power, and even s390x), as well as GPUs (*Psylang* targets NVPTX to run on CUDA capable GPUs), we also exploit serialized for of the IR to pass generated programs and external analysis tools. It also enables us to use existing compiler APIs to process and analyze generated code.

The compiler pipeline is built around the above observations and it's outlined in figure 2.5. The compilation process takes few steps; ① data structures are analyzed and converted to static representation. Lists, sets, and other dynamic data types are

converted to structs and arrays. This conversion also determines data layout to be used by the compiled functions. Some model parameters are considered structural (e.g. 'metric' parameter of distance function) and integrated directly in step ② rather than converted to static representation. This allows *Psylang* to choose and decide which parameters are exposed to the compiler and which are not. ② LLVM IR is generated to match computational semantics, this is done via combination of automated translation and hand-written IR code. This step uses thin LLVM abstraction provided by the *llvmlite* [81] module. Because the data structures were determined in step ①, Python's polymorphic semantics are specialized as necessary. For example where Python uses simple addition, this step generates either vector addition, scalar to vector addition, or scalar addition based on the inferred types of operands. IR generation tries to match the original Python model semantics as closely as possible to avoid the need to re-validate compiled variants of the models. This often leads to code that is more demanding (*e.g.* uses higher precision data types) or less suitable for acceleration (*e.g.* using specific PRNG). These issues are discussed in more depth in § 2.6. ③ the generated IR can be passed through LLVM's builtin optimization and code generation passes. ④ the final results is exported as a Python *ctype* function that is invoked from Python and uses *ctype* structures generated in ① as arguments.

For analysis purposes we extract the LLVM IR generated in step ② and pass it to a collection of analysis tools. The analysis tools target generic IR form, and are not affected by domain specialization that was exploited to generate it.

There are several key differences between *Psylang* and traditional JIT compilation, based on the observations discussed earlier. *Psylang* avoids runtime analysis of hot paths because it can exploit domain specific knowledge of model semantics, i.e. *Psylang* knows which code to compile and which can be left in Python. *Psylang* doesn't need to support full semantics of Python language and dynamic data types. *Psylang* also uses monomoprhic code generation. Even if a Python function is reused by multiple components, *Psylang* generates separate implementation for each component. This is partly required by the polymorhic nature of Python's operations (e.g. vector addition and scalar addition can use the same Python function). It also enables us to track

generated code to each model component rather than shared source code function, and this can be exploited for more accurate analysis. The combination of the above allows us to not only generate much more efficient code than traditional JIT compilers, but it also significantly simplifies the design.

2.4 Model Analysis

Beyond achieving high-performance, *Psylang* provides modelers with insights gleaned from compiler analysis passes. Inspecting the LLVM pass library we found several passes that can provide beneficial information to the modeler. This is similar to providing compiler feedback to application programmers. In order to extract useful information we construct a separate 'analysis' pipeline. This pipeline extends passes beyond what is currently done for efficient code generation. An overview of these passes and our extensions is presented in table 2.4.

In addition to standard compiler passes, we use IR representation of cognitive models to apply clone detection techniques. This demonstrates the usefulness of common IR representation of models, and wider applicability of software engineering tools in cognitive modeling.

Both of these approaches aim to improve the modeling environment by providing modelers with rich information about models they are building. While a significant part of model analysis is domain specific and needs to be done on a higher level, we demonstrate several analyses that are currently used across programming languages and extend them to the domain of cognitive modeling.

Function Inlining and Loop Unrolling

Increasing the aggressiveness of function inlining and loop unrolling beyond their thresholds for efficient code generation benefits our analysis pipeline in several ways. First, it allows the compiler to analyze and potentially eliminate static scheduler rules. This in turn allows analyzing sequences instead of individual components. Second, it allows us to use existing intra-procedural passes for an entire model instead of extending the

Pass Name	Application	Modification
Function inlining	Enables application of other intra-procedural optimizations and analyses on the entire model.	Increase constraints.
Loop unrolling	Enables composition of static scheduling rules to reconstruct node execution order in the original model.	Increase constraints.
Dead Code Elimination (DCE)	Highlights redundant parts of the model, pointing to design inefficiencies or parameter special cases. Works in combination with constant propagation and value range propagation.	None. Benefits from extensions to VRP and SCEV.
Value Range Propagation (VRP)	Calculates result ranges when provided with parameter and input constraints. Ranges can be reported back to the modeler, or used in DCE.	Extended to support floating point types.
Scalar Evolution (SCEV)	Calculates variable ranges in relation to loop iterations, extends VRP to loops. Similarly to VRP the calculated values can be reported directly to the modeler, or used in DCE.	Extended to support floating point types.
Global Value Numbering (GVN)	Detects identical computation, variant of common subexpression elimination (CSE)	Analysis uses more robust clone detection instead (2.4)
Scalarization and Reassociation	Scalar operations are used for canonical representation of computation.	None

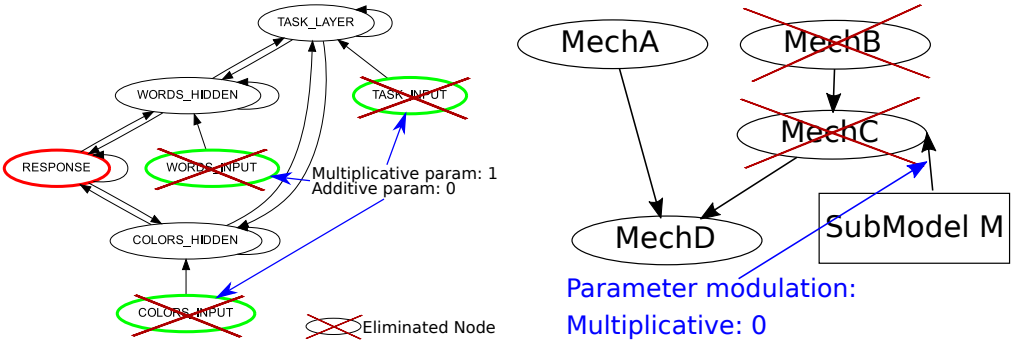
Table 2.4: Example of compiler passes used by *Psylang* for model analysis

amount of information passed between procedures.

Dead code elimination (DCE)

DCE is a rather straightforward optimization included in the LLVM suite that can be used to provide compiler feedback to the modeler. Code that can be safely eliminated does not impact the result or preserved state. This redundancy can be a consequence of specific selection of parameters, or a bug in the model design. Dead code can be also introduced intentionally to achieve a more readable model structure.

To demonstrate usefulness of this transformation, we looked at the Stroop model from [78], and discussed in section 2.2. The visualization of *PsyNeuLink* representation of the model is presented in figure 2.6a. After fixating the selected parameters,



(a) Stroop test (b) Nested model

Figure 2.6: Impact of DCE on Stroop test model after exposing configuration parameters to the compiler

DCE reveals that the operations in the input nodes can be safely eliminated without changing the result. For the Stroop test model this is expected, and the behaviour was implemented intentionally for better structural clarity. For larger composited models this analysis should uncover redundancies that do not manifest in individual submodels. However, such models are only currently being developed utilizing the performance benefits provided by *Psylang*.

Pinpointing unused parts of the model will become more important with future models composed of separately developed submodels. For example a situation in figure 2.6b shows how a nested submodel can modulate a parameter in parent model. It also enables modelers to use a library of generic modeling approaches and pinpoint redundancies that emerge from specializing generic models to specific use cases.

Value Range Propagation (VRP)

Next we investigate *Value Range Propagation*, a dataflow analysis that determines ranges of variables based on control flow, type restrictions, and used operations. For example $exp(x)$ can only ever by a positive number or NaN, and a commonly used *Logistic* function can be shown to always output values in range (0,1>. We have extended LLVM's implementation of this pass to include support for floating point types and common floating point operations.

We checked the impact of varying values of the back edge in the task selection node

of the Stroop model. The results are reported in table 2.5. The ability to examine parameter ranges allows modelers to reason about model sensitivity to specific parameters.

Back-edge matrix	Results (decision)
$\begin{bmatrix} 0 & <-2,-1) \\ <-2,-1) & 0 \end{bmatrix}$	$<0.990, 0.995>$
$\begin{bmatrix} 0 & <-5,-2) \\ <-5,-2) & 0 \end{bmatrix}$	$<0.976, 0.990>$

Table 2.5: Impact of back edge matrix values on results after 2 iterations of the Stroop test model.

Extending *value range propagation* to support floating point ranges is useful beyond analyzing cognitive models. Many floating point operations need special handling in the presence of special values like negative zero, not-a-number, or infinities. While the compiler can be instructed to optimize these using special fast-math optimization flags, these are currently set globally per compilation unit or per function, or tracked in a limited way. Floating point ranges can be used to determine the absence of special values for each operation and fast-math optimizations can be applied without breaking strict semantics. This is especially useful for GPU targets which often have specialized fast instructions that do not fully adhere to IEEE floating point semantics.

Scalar Evolution (SCEV)

extends variable value tracking to loops and tracks value ranges across loop iterations as well as calculating number of loop iterations if it can be inferred from the available information. Similarly to *Value Range Propagation* we extend LLVM's SCEV pass to support floating point type. We also extend the pass to calculate minimum number of loop iterations iterations. This allows us to provide modeler feedback even on models that predict response times based on accumulation of evidence, such as race models.

For example a simple linear ballistic accumulator (LBA) [82] race, presented in Listing 2.2, would need between 17 and 481 iterations to complete. Variable ranges at loop exit can then be used to continue range analysis beyond loops. Although these techniques can only use previously discovered analytical forms of recurrent expressions, exposing the information to the modeler can provide immediate feedback that could

indicate reduction of model strength based on used parameters.

```
#define assume(x, l, h) \
  if (!(x<=h && x>=l)) return; \
  else (void)0

void race_model(float starting_point,
                float threshold,
                float rate,
                float stimulus,
                float noise,
                float time_step,
                float *out)
{
  assume(starting_point, 0.1f, 0.1f);
  assume(threshold, 2.0f, 2.5f);
  assume(noise, 0.5f, 1.5f);
  assume(rate, 0.5f, 0.5f);
  assume(time_step, 0.1f, 0.1f);
  assume(stimulus, 0.2f, 1.5f);

  float acc = starting_point;
  float count = 0;
  while (acc < threshold) {
    count += 1;
    acc += (1.0f - rate) * stimulus
           * noise * time_step;
  }
  *out = count;
}
```

Listing 2.2: Race model example in C. Scalar evolution can calculate the expected number of iterations to be between 17 and 481

Adaptive Mesh Refining (AMR)

While value ranges can be reported directly to the modeler, range computation can be also exploited to improve performance of another critical task; parameter search. In this task the model is run repeatedly with associated cost function in order to find the best cost-benefit trade-off. With the ability to do ranged computation we can examine subspaces of global parameter space, replacing traditional optimization techniques with adaptive mesh refining approach.

AMR is a known technique used to dynamically increase or reduce resolution (and resource usage), based on how "interesting" a given region is. This can be co-opted to improve optimization functions in cognitive models. Utilizing compiler value range

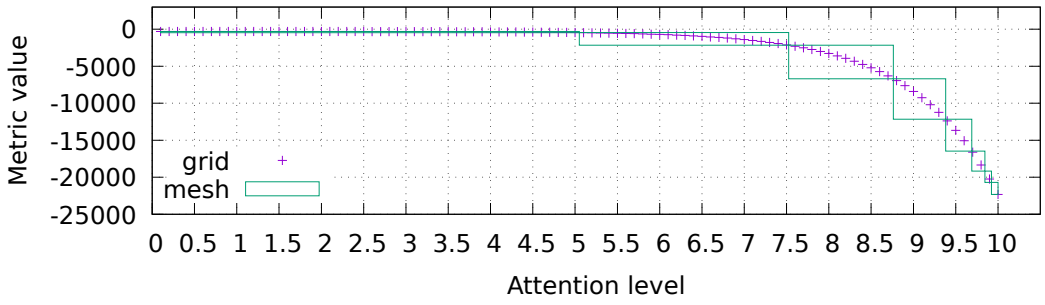


Figure 2.7: Search for minimum using mesh refining in the Predator-Prey model superimposed over sampled grid

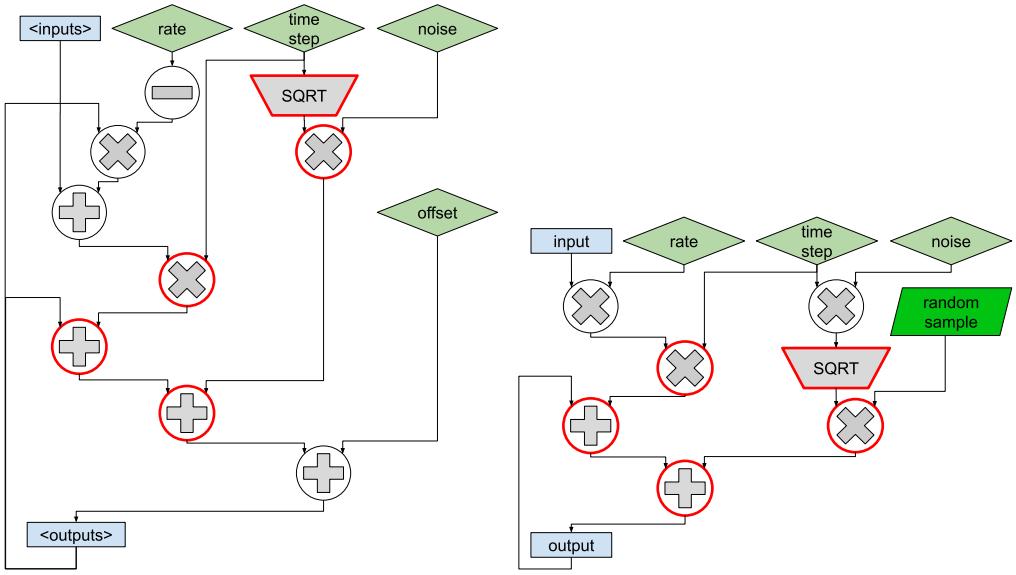
propagation can quickly provide information about large sections of the parameter space.

An example is shown in figure 2.7. There are multiple techniques that can be used to solve this optimization problems. However, gradient based techniques require analyzing the computational model and constructing a gradient function. We observe that the compiler already has similar information available in the form of value ranges. Although using compiler value ranges is not a complete substitute for gradient based techniques, the information is already generated during the compilation process and can be easily reused for optimization purpose.

Clone detection

detects computation clones between two models. Detecting clones or similarities in CDFG is an instance of subgraph isomorphism problem, which is known to be NP-complete [83]. Our tool leverages the limited size of analyzed models and uses straightforward brute-force approach to demonstrate the feasibility of clone detection in models at IR level. It demonstrates that given the IR for two models it is possible to find shared computations between them. To achieve this, we leverage LLVM's existing FunctionComparator framework to detect exactly equivalent functions. Typically this has been used to detect similar functions within the same module to merge them. We use this to detect similar functions across modules. The tool then traverses each function in CFG-order. Its output includes the binary information if two functions are exactly identical or one of them is subset of the other. We extend this approach to output the parts of code which were identified as different. The differences can be traced (using line

debugging information) back to the corresponding model element.



(a) Leaky Competing Integrator (LCI) (b) Drift Diffusion Integrator (DDI)

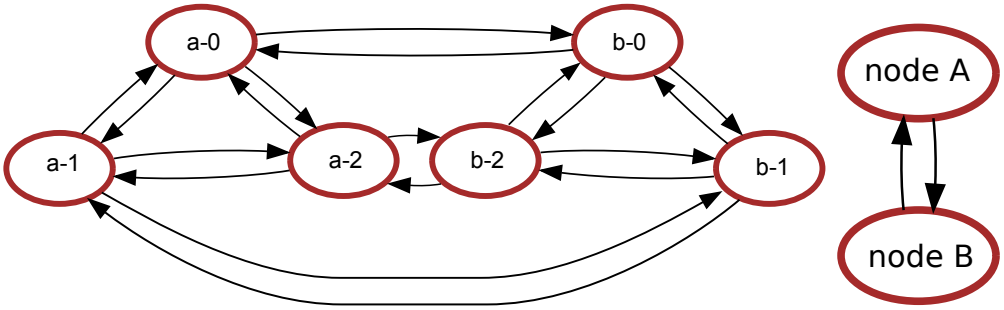
Figure 2.8: Identical computation highlighted in red. Setting $rate_{LCI} = 0$, $offset_{LCI} = 0$, $noise_{LCI} = N(0, 1)$, $rate_{DDI} = 1$, and $noise_{DDI} = 1$, configures both functions to perform identical computation.

To demonstrate usefulness of clone detection we first consider two similar functions; DriftDiffusionIntegrator and LeakyCompetingIntegrator. These functions underpin two most commonly used decision making models; Drift Diffusion Model (DDM) for two alternative force choice decision making, and Leaky Competing Integrator (LCA) multi-choice model. Both of them work in a similar fashion, they accumulate 'decision energy' until a threshold for decision is crossed, the results are not just the decision taken, but also the number of iterations (time-steps) it took to reach that decision. The models are run thousands of time to build histogram of response times. Parameters include strength of stimuli, noise, and any inhibitory influence between different competing decision. It's easy to see that a multi-choice model can be used in place of a two-choice model. Figure 2.8 shows that the underlying accumulation is very similar with identical sequence at the core of the computation.

The important difference is that there's a known analytical equivalent of DDI. The analytical function calculates the first three statistical moments (mean, variance, skew) of times to take both correct and incorrect decisions. Notifying the modeler that a

decision making iterator can be replaced with an analytical solution can save thousands of model executions and thus significantly improve performance.

Our first example found identical computation on the level of functions. However, aggressive inlineing allows our methodology to work across components. To demonstrate cross-component clone detection we selected two models of Necker cube, shown in figure 2.9. The first model is simplified to only three vertices per geometric shape (originally eight – cube). It uses two nodes per shape vertex, one for each way the orientation can be perceived, for six nodes total. The other model, is hand-tuned to operate on 16 element vectors and uses only two nodes total. One vector lane represents signal strength of particular combination of vertex and orientation.



(a) 3-vertex Necker cube model

(b) Vectorized Necker cube model

Figure 2.9: Comparison of *PsyNeuLink* Necker cube models. a) is a simplified version with only three vertices, using percepts per vertex (6 nodes) b) is a vectorized version, each vertex × percept is represented as lane in a vector rather than an individual model node

The models have completely different structure, and each node performs different amount of computation. Our tool correctly recognizes that the simplified Necker cube model is completely included in the full vectorized version. The only identified difference is in two IR instructions (out of 3000) corresponding to input handling.

Because our clone detection tool works on the IR level, it is independent of the original model structure. This allows us to analyze core computational differences of the models. We can check if two models or parts of the same model are similar, whether this is expected to confirm our understanding, or unexpected result of a bug, or a completely new insight into relationship between two cognitive models.

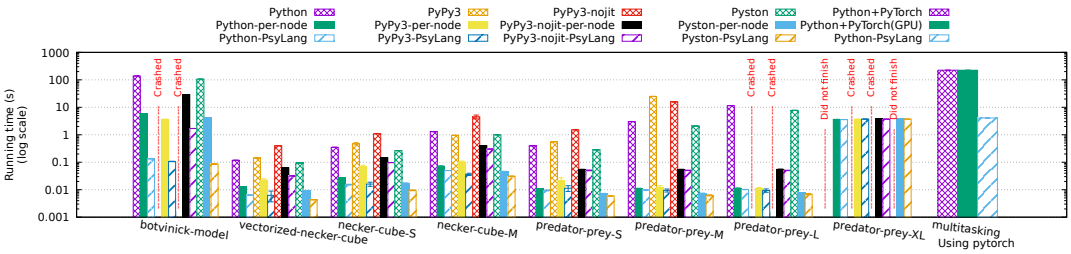


Figure 2.10: Running time comparison of models. Large models (*botvinick* and *predator-prey-large*) did not complete when run using *PyPy3*. *multitasking* uses *pytorch* which is not compatible with *PyPy3*.

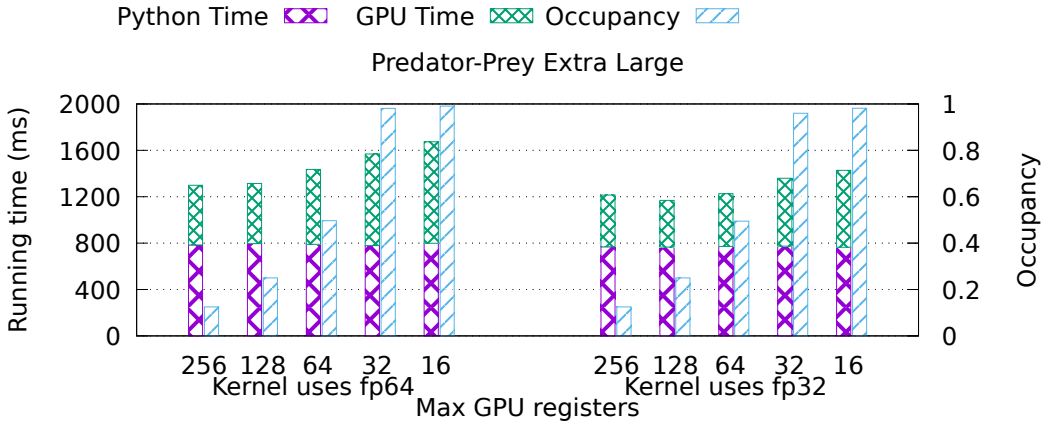


Figure 2.11: GPU running time using different restrictions on available register space. Total size of private (per-thread) data is 18.5kB for double precision variant, and 15.5kB for single precision.

Although our demonstrator uses straightforward brute-force comparison method, operating on standard IR makes approaches developed for detecting clones in application software [84, 85] directly applicable.

2.5 Performance Evaluation

We compare *Psylang* with the baseline *PsyNeuLink* implementation executed using Python-3.6.9 as well as pypy3-7.3.2 and pyston-2.0.0. All test were run on an off-the-shelf workstation using Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz and 16GB of DDR4@2666Mhz RAM. The execution times were collected using *pytest-benchmark* package and we report average running time and standard deviation error bars. *pytest-benchmark* was configured to include two warmup runs before collecting the runtime data.

We run a selection of models described in section 2.2 in different configurations.

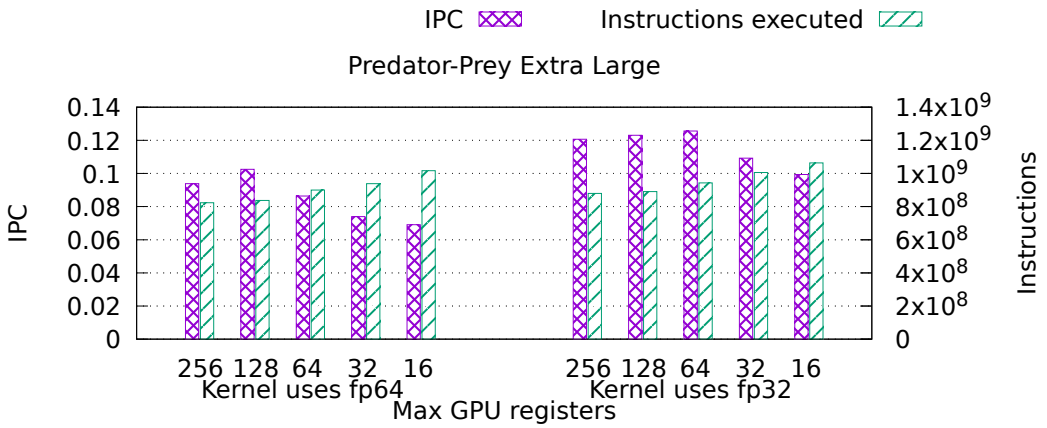


Figure 2.12: GPU IPC and instruction count using different restrictions on available register space.

"Botvinick" refers to the Stroop model described in section 2.2. The Necker cube model was run in three different configurations; manually vectorized version, simplified variant using three vertices (S), and a full eight vertex version (M). Predator-prey model was configured to use two, four, and six levels of attention per game avatar for **S**mall, **M**edium, and **L**arge configuration, respectively. We also ran an e**X**tra **L**arge configuration with 100 levels of attention. *Multitasking* model was run without special configuration.

We were surprised by *PyPy3*'s performance. Despite its stated goals of increased performance and lower memory usage. It performed $\sim 30\%$ slower than vanilla Python on average, and the two largest instances failed to complete after exhausting all 16GB of memory available on our test system.

While further development will undoubtedly solve some of these issues, it illustrates the complexity of designing a generic JIT compiler for Python. *PyPy3* needs to spend resources to gain information about the workload that is already known by its domain specific nature. Indeed we have observed that *PyPy3* in some models achieves better performance with its JIT engine turned off.

Moreover, *PyPy3* and *Pyston* do not deliver the expected benefits of scaling. Scaling the models is important, as it allows neuroscientists to collect high fidelity data. While compilation cannot change the properties of the implemented algorithms, *Psylang* lowers the base running time enough for large scale models to execute on standard machines.

We also evaluated the option of using CUDA GPUs for Predator-Prey's parallel search. To push scaling to its limits, we increased the number of attention levels to 100 (or 1M model evaluations). Using a GPU provided further 3.3× performance improvement over compiled CPU execution, completing the task in 1.25 and 4.18 seconds for GPU and CPU respectively. Both CPU and GPU significantly outperformed Python execution which was stopped after 24 hours, at which point the Python run was terminated, giving a speed up of more than 20000× for CPU and almost 70000× for GPU. More fundamental challenges in mapping cognitive models to GPUs and accelerators in general are discussed in § 2.6.

The *Multitasking* model demonstrates the need to accelerate models as a whole rather than focusing on individual parts. The baseline model invokes *PyTorch* for the neural net part of the model. However, repeated switches between Python and *PyTorch* execution slow down the performance. Even though *Psylang* only generates naive, straightforward, implementation of neural network inference, tighter integration with the rest of the model allows it to outperform the baseline more than three orders of magnitude.

In general our experiments show that available solutions are either too generic and miss available optimization opportunities (*PyPy3*, *Pyston*), or too specialized on a narrow subtask (*PyTorch*). *Psylang* strikes a balance between exploiting enough domain specific knowledge to achieve good performance while maintaining its generality to support wide range of modeling workloads expressed in *PsyNeuLink*. This allows *Psylang* to achieve between 10× and 70000× better performance than the Python baseline.

2.6 Discussion and Future Work

Psylang is the first step in bringing the benefits of compilers to the world of cognitive modeling. We have discussed both performance implications and how compiler analyses can be used to provide useful feedback to the modeler.

Heterogeneity

The biggest challenge is to provide provide high performance for model partitions that might have significantly different characteristics. Different parts can map to different accelerators, GPUs, TPUs [6], and number of other HW accelerators [86–88] have been proposed to speed up both learning and inference in ML workloads. Efficient use of these resources will be crucial to provide high performance to cognitive models as well. At the same time this presents an opportunity to leverage the existing work on accelerator integration done in systems community [63, 89, 90].

Compiler optimizations

Another challenge is to generate efficient code, and perform useful analysis across heterogeneous compute environments. Optimizations of neural networks have developed their own set of techniques (like quantization and synaptic pruning) that are different from traditional compiler optimizations. Some of them (like quantization) might be necessary to make efficient use of specialized HW. For example GPUs, and TPUs offer the highest throughput for low bit data types like 16-bit floating point or 8-bit integers.

Psylang sidesteps both of these issues by providing numerical equivalence between the original Python model and the compiled version. However, this is achieved by only using 64-bit floating point to match Python semantics.

Randomness

Many of cognitive models are stochastic or rely on stochastic processes. For example, input perturbation in the Predator-Prey model relies on Gaussian noise applied to accurate inputs. This stochasticity is modelled using software implementations of pseudo-random number generators (PRNGs). Previous work showed that using different PRNGs can lead to up to 30% difference in results of Monte-Carlo simulations [91]. We are not aware of any similar work in the field of cognitive modeling. Once more, *Psylang* sidesteps this problem by strictly following Python semantics and using Mersenne Twister PRNG, the same PRNG used by Python and numpy. However, this PRNG requires $\sim 2KB$

of state and therefore maps poorly to accelerators such as GPUs. Alternative, GPU friendly, PRNGs exist but the choice of PRNG will need to be closely coordinated with the cognitive modeling community to make sure the models are numerically validated with different PRNGs.

2.7 Related Work

2.7.1 Accelerating Python

Given Python's increasing popularity [30] there have been several attempts to increase it's performance and interoperability with other environments.

PyPy [34] is a reimplementation of the Python language using RPython translation framework with built-in JIT compiler. It's state goals are speed and compatibility. PyPy successfully runs most of the models and was included in our evaluation.

Pyston [35] is the latest addition to JIT enabled Python implementations. It's capable of running all models without modifications and it delivers on the advertised $\sim 15\%$ performance improvement.

Jython [32] is a community maintained Python implementation that aims to improve interoperability between Python and Java™. Any improved performance comes as a side effect of utilizing JVM environment to run merged code. Unfortunately, Jython only supports Python 2 language.

IronPython [33] is similar in its goal to improve interoperability between Python and .NET programs. Potential for improved performance comes from using .NET common language runtime (CLR). Similarly to Jython above, IronPython support for Python3 is not ready yet.

Numba [36] is a JIT compilation framework that uses function decorators to transparently compile python code and execute functions natively. However, its limited support for some of Python's features [92] would require significant refactoring of *PsyNeuLink* codebase.

2.7.2 Workload integration

We are not the first to notice that a good performance often depends on effectively integrating different parts rather than focusing on a single dominant kernel. CUDA Graphs [93] exposes an interface that coalesces multiple successive kernel invocation into a single combined invocation to amortize invocation overhead.

Similarly, Ravishankar and Grover [29] propose lazy evaluation of Numpy expressions until multiple operations can be combined. *Psylang* has the advantage of knowing the hot path ahead of time so it can optimize the calls, rather than tracking invoked operations for lazy acceleration.

2.7.3 Model Analysis

To the best of our knowledge, there has not been much work on analyzing compiled models. However, Alafi et al. [94] notice similarity between block level modeling and standard programming to deploy similarity detection techniques. Unlike Alafi et al., *Psylang* considers model execution in compiler IR form rather source code.

2.7.4 Clone detection

Previous work has proposed a variety of clone detection techniques targeting different languages. These techniques differ in both the way they search the code base and algorithms they use to select and compare code segments.

Some work, like NICAD [85] and previously mentioned SIMONE [94] focus on analyzing source code text. This has the advantage of applying the comparison techniques early and avoiding potentially costly source parsing. However, *PsyNeuLink* models are constructed dynamically at runtime so a source code representing the entire model is not available. Applying the techniques to serialized dump of the model is possible, but unlikely to match our results given the high level nature of functions in cognitive models.

Works targeting semantic clones on IR level are more directly applicable. Gabel et al. [84] uses program dependence graph (another name for CDFG). To avoid scalability problems of detecting sub-graph isomorphism Gabel first reconstructs normalized forms

of abstract syntax trees (ASTs). Similarly, [95, 96], and [97] detect similarities in Java bytecode. We believe these approaches to be applicable to cognitive models.

2.8 Conclusion

Psylang examines the role of compilers to support robust and high performance modeling environment for cognitive neuroscience. Beyond the critical performance improvements, necessary to support large, high fidelity models, we examine suitability of compiler analyses to cognitive modeling. We propose and implement modifications to production compiler suite to provide rich feedback to the modelers. Further we notice the heterogeneous nature of cognitive models, and discuss the role of compiler to bridge the gap between the expressed computation and different hardware execution platforms. All our contributions are part of open-sources projects and will be released for public use.

Chapter 3

Generic System Services for GPUs

3.1 Introduction

GPUs have evolved from fixed function 3D accelerators to fully programmable units [98–100] and are now widely used for high-performance computing (HPC), machine learning, and data-analytics. Increasing deployments of general-purpose GPUs (GPGPUs) have been partly enabled by programmability enhancing features like virtual memory [101–106] and cache coherence [107–111].

GPU programming models have evolved with these hardware changes. Early GPGPU programmers [112–116] adapted graphics-oriented programming models such as OpenGL [117] and DirectX [118] for general-purpose usage. Since then, GPGPU programming has become increasingly accessible to traditional CPU programmers, with graphics APIs giving way to computation-oriented languages with a familiar C/C++ heritage such as OpenCL [16], C++AMP [20], and CUDA [15]. However, access to privileged OS services via system calls is an important aspect of CPU programming that remains out of reach for GPU programs.

Designers have begun exploring ways to fill this research void. Studies on filesystem I/O (GPUfs [17]), networking I/O (GPUnet [18]), and GPU-to-CPU callbacks [31] established that direct GPU invocation of some specific system calls can improve GPU performance and programmability. These studies have two themes. First, they focus on specific system calls (i.e., for filesytems and networking). Second, they replace the traditional POSIX-based APIs of these system calls with custom APIs to drive performance. In this study, we ask – can we design an interface for invoking *any* system call from GPUs, and can we do so with standard POSIX semantics to enable seamless and wider adoption? To answer these questions, we design the first framework for **generic**

system call invocation on GPUs, or **GENESYS**. GENESYS offers several concrete benefits.

First, GENESYS can support implementation of most of Linux's 300+ system calls. As a proof of concept, we go beyond the specific set of system calls from prior work [17, 18, 31] and implement not only filesystem and networking system calls, but also those for asynchronous signals, memory management, resource querying, and device control.

Second, GENESYS's use of POSIX allows programmers to reap the benefits of standard APIs developed over decades of real-world usage. Recent work on SPIN [19] takes a step in this direction by considering how to modify the specific system calls in GPUfs to match traditional POSIX semantics. But GENESYS's generality in supporting all system calls means that it goes further, enabling, among other things, backwards compatibility – GENESYS makes it possible to deploy on GPUs the vast body of legacy software written to invoke OS-managed services.

Third, GENESYS's generality enables GPU acceleration of programs that were previously considered a poor match for GPUs. For example, it allows applications to directly manage their memory, query the system for resource information, employ signals, interface with the terminal, etc., in a manner that lowers programming effort for GPU deployment. These examples underscore GENESYS's ability to support new programming strategies and even legacy applications (e.g., using terminal/signals).

Finally, GENESYS can leverage the benefits of support for important OS features that prior work cannot. For example, GPUnet's use of custom APIs precludes the use traffic shaping and firewalls that are already built into the OS.

When designing GENESYS, we ran into several design questions. For example, what system calls make sense for GPUs? System calls such as *pread/pwrite* to file(s) or *send/recv* to and from the network stack are useful because they underpin many I/O activities required to complete a task. But system calls such as *fork* and *execv* do not, for now, seem necessary for GPU threads. In the middle are many system calls that need adaptation for GPU execution and are heavily dependent on architectural features that could be supported by future GPUs. For example, *getrusage* can be adapted to

return information about GPU resource usage. We summarize the conclusions from this qualitative study.

We then perform a detailed design space study on the performance benefits of GENESYS. Key questions are:

How does the GPU's hardware execution model impact system call invocation strategies? To manage parallelism, the GPU's underlying hardware architecture decomposes work into a hierarchy of execution groups. The granularity of these groups ranges from work-items (or GPU threads) to work-groups (composed of hundreds of work-items) to kernels (composed of hundreds of work-groups)[1]. This naturally presents the following research question – at which of these granularities should GPU system calls be invoked?

How should GPU system calls be ordered? CPU system calls are implemented such that instructions prior to the system call have completed execution, while code following the system call remains unexecuted. This model is a good fit for CPUs, which generally target single-threaded execution. But such "strong ordering" semantics may be overly conservative for GPUs. It acts as implicit synchronization barriers across thousands of work-items, compromising performance. Similar questions arise as to whether GPU system calls should be "blocking" or "non-blocking."

Where and how should GPU system calls be processed? Like all prior work, we assume that system calls invoked by GPU programs need to ultimately be serviced by the CPU. This makes efficient GPU-CPU communication and CPU-side processing fundamental to GPU system calls. We find that careful use of modern GPU features like shared virtual addressing [101] and page fault support [105, 106], coupled with traditional interrupt mechanisms, can enable efficient CPU-GPU communication of system call requests and responses.

---

[1]Without loss of generality, we use AMD's terminology of work-items, work-groups, kernels, and compute unit (CU), although our work applies equally to the NVIDIA threads, thread-blocks, kernels, and streaming multiprocessors (SMs), respectively.

To explore these questions, we study GENESYS with microbenchmarks and end-to-end applications. Overall, our contributions are:

①: We take a step toward realizing truly heterogeneous programming by enabling GPUs to directly invoke OS-managed services, just like CPUs. This builds on the promise of recent work [17–19, 31] but goes further by enabling direct invocation of *any* system call through standard POSIX APIs. This permits GPUs to use the entire ecosystem of OS-managed system services developed over decades of research.

② As a proof-of-concept, we use GENESYS to realize system calls previously unavailable on GPUs to directly invoke OS services for memory management, signals, and specialized file-system use. Additionally, we continue supporting all the system services made available by prior work (i.e., GPUs, GPUnet, SPIN), but do so with standard POSIX APIs.

③ We shed light on several novel OS and architectural design issues in supporting GPU system calls. We also offer the first set of design guidelines for practitioners on how to directly invoke system calls in a manner that meshes with the execution hierarchy of GPUs to maximize performance. While we use Linux as a testbed to evaluate our concepts, our design choices are applicable more generally across OSes.

④ We publicly release GENESYS hosted under the Radeon Open Compute stack [119–123], offering its benefits broadly.

3.2 Motivation

A reasonable question to ponder is, why equip GPUs with system call invocation capabilities at all? Conceptually, OSes have traditionally provided a standardized abstract machine in the form of a process to user programs executing on the CPU. Parts of this process abstraction, such as memory layout, the location of program arguments, and ISA, have benefited GPU programs. Other aspects, however, such as standardized and direct protected access to the filesystem, network, and memory allocation, are extremely important for processes but are yet lacking for GPU code. Allowing GPU code to invoke

Table 3.1: GENESYS enables new classes of applications and supports all prior work.

	Type	Application	Syscalls	Description
Previously Unrealizable	Memory Management	miniAMR	madvise, getrusage	Uses madvise to return unused memory to the OS (Sec 3.8.1).
	Signals	signal-search	rt_sigqueueinfo	Uses signals to notify the host about partial work completion (Sec 3.8.2).
	Filesystem	grep	read, open, close	Work-item invocations not supported by prior work, prints to terminal (Sec 3.8.3).
	Device Control (ioctl)	bmp-display	ioctl, mmap	Kernel granularity invocation to query and setup framebuffer properties (Sec 3.8.5)
Previously Realizable	Filesystem	wordsearch	pread, read	Supports the same workloads as prior work (GPUfs) (Sec 3.8.3).
	Network	memcached	sendto, recfrom	Possible with GPUnet but we do not need RDMA for performance (Sec 3.8.4).

system calls is a further step to providing a more complete process abstraction to GPU code.

Unfortunately, GPU programs can currently only invoke system calls indirectly, and thereby suffer from performance challenges. Consider the diagram on the left in Figure 3.1. Programmers are currently forced to delay system call requests until the end of the GPU kernel invocation. This is not ideal because developers have to take what was a single conceptual GPU kernel and partition it into two – one before the system call and one after it. This model, which is akin to continuations, is notoriously difficult to program [124]. Compiler technologies can assist the process [125], but the effect of ending the GPU kernel, and restarting another is the same as a barrier synchronization across all GPU threads and adds unnecessary round-trips between the CPU and the GPU, both of which incur significant overhead.

In response, recent studies invoke OS services from GPUs [17–19, 31], as shown on the right in Figure 3.1. This approach eliminates the need for repeated kernel launches, enabling better GPU efficiency. System calls (e.g., *request_ data*) still require CPU processing, as they often require access to hardware resources that only the CPU interacts

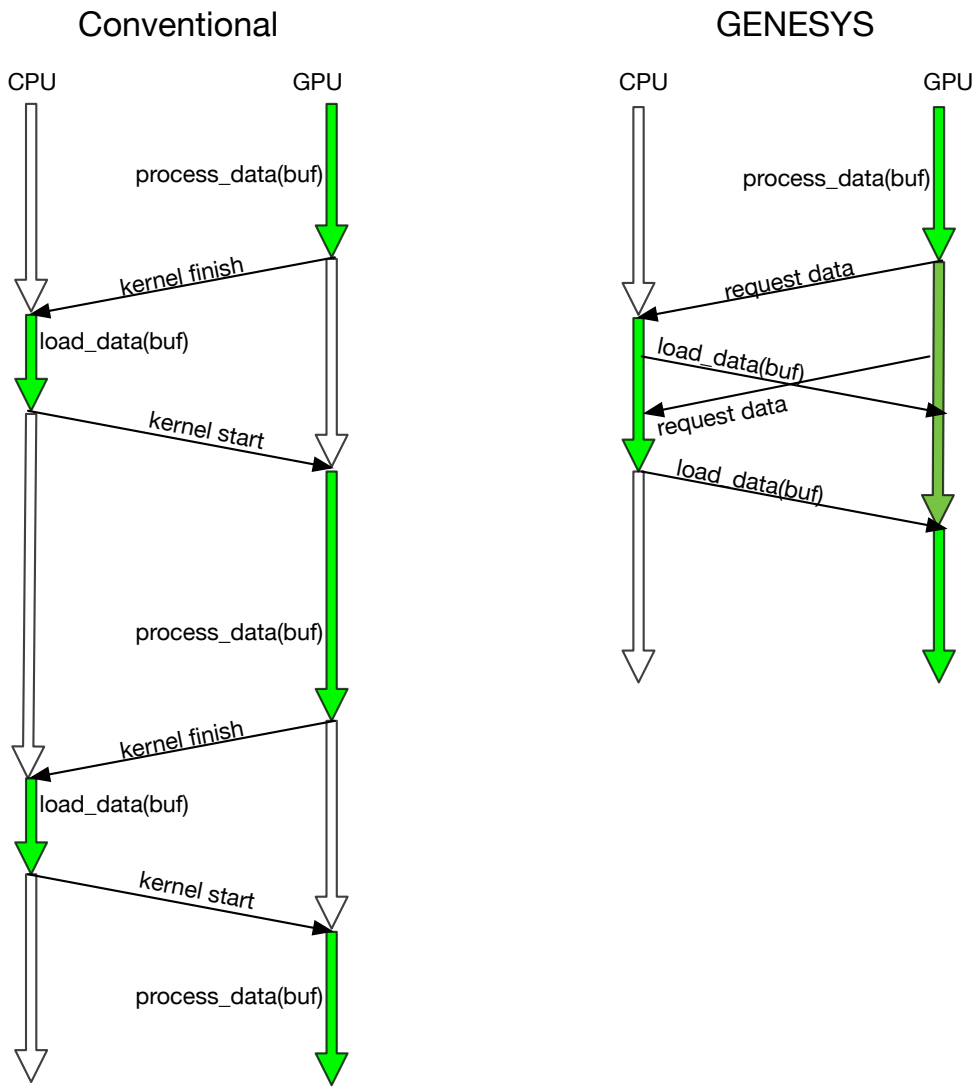


Figure 3.1: (Left) Timeline of events when the GPU has to rely on a CPU to handle system services; and (right) when the GPU has system call invocation capabilities.

with. However, CPUs only need to schedule tasks in response to GPU system calls as and when needed. CPU system call processing also overlaps with the execution of other GPU threads. Studies on GPUfs, GPUnet, GPU-to-CPU callbacks, and SPIN are seminal in demonstrating such direct invocation of OS-managed services but suffer from key drawbacks:

Lack of generality: They target specific OS-managed services, and therefore, realize only specific APIs for filesystem or networking services. These interfaces are not readily extensible to other system calls/OS services.

Lack of flexibility: They focus on specific system call invocation strategies. Consequently, there has been no design space exploration on the general GPU system call interface. Questions such as the best invocation granularity (i.e., whether system calls should be invoked per work-item, work-group, or kernel) or ordering remain unexplored, and as we show, can affect performance in subtle and important ways.

Reliance on non-standard APIs: Their use of custom APIs precludes the use of many OS-managed services (e.g., memory management, signals, process/thread management, scheduler). Further, custom APIs do not readily take advantage of existing OS code-paths that enable a richer set of system features. Recent work on SPIN points this out for filesystems, where using custom APIs causes issues with page caches, filesystem consistency, and incompatibility with virtual block devices such as software RAID.

While these past efforts broke new ground and demonstrated the value of OS services for GPU programs, they did not explore the question we pose – why not simply provide generic access from the GPU to all POSIX system calls?

3.3 High-Level Design

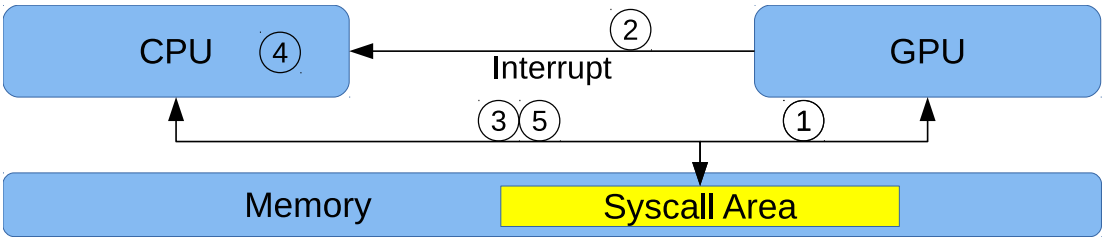


Figure 3.2: High-level overview of how GPU system calls are invoked and processed on CPUs.

Figure 3.2 outlines the steps used by Genesys. When the GPU invokes a system call, it has to rely on the CPU to process system calls on its behalf. Therefore, in step ①, the GPU places system call arguments and information in a portion of system memory also visible to the CPU. We designate a portion of memory as the *syscall area* to store this information. In step ②, the GPU interrupts the CPU, conveying the need for system call processing. Within the interrupt message, the GPU also sends the ID number of the wavefront issuing the system call. This triggers the execution of an

Table 3.2: Examples of system calls that require hardware changes to be implementable on GPUs. In total, this group consists of 13% of all Linux system calls. In contrast, we believe that 79% of Linux system calls are readily-implementable.

Type	**Examples**	**Reason that it is not currently implementable**
capabilities	capget, capset	Needs GPU thread representation in the kernel
namespace	setns	Needs GPU thread representation in the kernel
policies	set_mempolicy	Needs GPU thread representation in the kernel
thread scheduling	sched_yield set_cpu_affinity	Needs better control over GPU scheduler
signals	sigaction suspend sigreturn sigprocmask	Signals require the target thread to be paused and then resumed after signal action is completed. GPU threads cannot be targeted. It is currently not possible to independently set program counters of individual threads. Executing signal actions in newly spawned threads might require freeing of GPU resources.
architecture specific	ioperm	Not accessible from GPU

interrupt handler on the CPU. The CPU uses the wavefront ID to read the system call arguments from the *syscall area* in step ③. Subsequently, in step ④, the CPU processes the interrupt and writes the results back into the *syscall area*. Finally, in step ⑤, the CPU notifies the GPU wavefront that its system call has been processed.

We rely on the ability of the GPU to interrupt the CPU and use readily-available hardware [126–128] for this. However, this is not a fundamental design requirement; in fact, prior work [17, 31] uses a CPU polling thread to service a limited set of GPU system service requests instead. Further, while increasingly widespread features such as shared virtual memory and CPU-GPU cache coherence [101, 105, 106, 110] are beneficial to our design, they are not necessary. CPU-GPU communication can also be achieved via atomic reads/writes in system memory or GPU device memory [129].

3.4 Analyzing System Calls

While designing Genesys, we classified all of Linux's over 300 system calls and assessed which ones to support. Some of the classifications were subjective and were debated even among ourselves. Many of the classification issues relate to the unique nature of the GPU's execution model.

Recall that GPUs use SIMD execution on thousands of concurrent threads. To

keep such massive parallelism tractable, GPGPU programming languages like OpenCL [16] and CUDA [15] expose hierarchical groups of concurrently executing threads to programmers. The smallest granularity of execution is the GPU work-item (akin to a CPU thread). Several work-items (e.g., 32-64) operate in lockstep in the unit of wavefronts, the smallest hardware-scheduled unit of execution. Many wavefronts (e.g., 16) constitute programmer visible work-groups and execute on a single GPU compute unit (CU). Work-items in a work-group can communicate among themselves using local CU caches and/or scratchpads. Hundreds of work-groups comprise a GPU kernel. The CPU dispatches work to a GPU at the granularity of a kernel. Each work-group in a kernel can execute independently. Further, it is possible to synchronize just the work-items within a single work-group [16, 109]. This avoids the cost of globally synchronizing across thousands of work-items in a kernel, which is often unnecessary in a GPU program and might not be possible under all circumstances[2].

The bottom-line is that GPUs rely on far greater forms of parallelism than CPUs. This implies the following OS/architectural considerations in designing system calls:

Level of OS visibility into the GPU: When a CPU thread invokes the OS, that thread has a *representation* within the kernel. The vast majority of modern OS kernels maintain a data-structure for each thread for several common tasks (e.g., kernel resource use, permission checks, auditing). GPU tasks, however, have traditionally not been represented in the OS kernel. *We believe this should not change.* As discussed above, GPU threads are numerous and short lived. Creating and managing a kernel structure for thousands of individual GPU threads would vastly slow down the system. These structures are also only useful for GPU threads that invoke the OS and represent wasted overhead for the rest. Hence, we process system calls in OS worker threads and switch CPU contexts if necessary (see Section 3.6). As more GPUs support system calls, this is an area that will require careful consideration by kernel developers.

[2]Although there is no single formally-specified barrier to synchronize across work-groups today, recent work shows how to achieve the same effect by extending existing non-portable GPU inter-work-group barriers to use OpenCL 2.0 atomic operations [130].

Evolution of GPU hardware: Many system calls are heavily dependent on architectural features that could be supported by future GPUs. For example, consider that modern GPUs do not expose their thread scheduler to software. This means that system calls to manage thread affinity (e.g., *sched_setaffinity*) are not implementable on GPUs today. However, a wealth of research has shown the benefits of GPU warp scheduling [131–135], so should GPUs require more sophisticated thread scheduling support appropriate for implementation in software, such system calls may ultimately become valuable.

With these design themes in mind, we discuss our classification of Linux's system calls.

① **Readily-implementable:** Examples include *pread, pwrite, mmap, munmap*, etc. This group is also the largest subset, comprising nearly 79% of Linux's system calls. In Genesys, we implemented 14 such system calls for filesystems (*read, write, pread, pwrite, open, close, lseek*), networking (*sendto, recvfrom*), memory management (*mmap, munmap, madvise*), system calls to query resource usage (*getrusage*), and signal invocation (*rt_sigqueueinfo*). Furthermore, we also implement device control *ioctls*. Some of these system calls, like *read, write, lseek*, are stateful. Thus, GPU programmers must use them carefully; the current value of the file pointer determines what value is read or written by the *read* or *write* system call. This can be arbitrary if invoked at work-item or work-group granularity for the same file descriptor because many work-items/work-groups can execute concurrently.

An important design issue is that Genesys's support for standard POSIX APIs allows GPUs to *read*, *write*, and *mmap* any file descriptor Linux provides. This is particularly beneficial because of Linux's "everything is a file" philosophy – Genesys readily supports features like terminal for user I/O, pipes (including redirection of *stdin*, *stdout*, and *stderr*), files in */proc* to query process environments, files in */sys* to query and manipulate kernel parameters, etc. Although our studies focus on Linux, the broader domains of OS-services represented by the specific system calls generalize to other OSes like FreeBSD and Solaris.

At the application level, implementing this array of system calls opens new domains of OS managed services for GPUs. In Table 3.1, we summarize previously unimplementable system calls realized and studied in this work. These include applications that use *madvise* for memory management and *rt_sigqueueinfo* for signals. We also go beyond prior work on GPUfs by supporting filesystem services that require more flexible APIs with work-item invocation capabilities for good performance. Finally, we continue to support previously implementable system calls.

② Useful but implementable only with changes to GPU hardware: Several system calls (13% of the total) seem useful for GPU code, but are not easily implementable because of Linux's existing design. Consider *sigsuspend*/*sigaction* – there is no kernel representation of a GPU work-item to manage and dispatch a signal to. Additionally, there is no lightweight method to alter the GPU program counter of a work-item from the CPU kernel. One approach is for signal masks to be associated with the GPU context and for signals to be delivered as additional work-items. This works around the absence of GPU work-item representation in the kernel. However, POSIX requires threads that process signals to pause execution and resume only after the signal has been processed. Targeting the entire GPU context would mean that all GPU execution needs to halt while the work-item processing the signal executes, which goes against the parallel nature of GPU execution. Recent work has, however, shown the benefits of hardware support for dynamic kernel launch that allows on-demand spawning of kernels on the GPU without any CPU intervention [136]. Should such approaches be extended to support thread recombination assembling multiple signal handlers into a single warp (akin to prior approaches on control flow divergence [132]), *sigsuspend* or *sigaction* may become implementable. Table 3.2 presents more examples of currently not implementable system calls (in their original semantics).

③ Requires extensive modification to be supported: This group (8% of the total) contains perhaps the most controversial set of system calls. At this time, we do not believe that it is worth the implementation effort to support these system calls. For example, *fork* necessitates cloning a copy of the executing caller's GPU state. Technically,

this can be done (e.g., it is how GPGPU code is context switched with the graphics shaders) but it seems unnecessary at this time.

3.5 Design Space Exploration

3.5.1 GPU-Side Design Considerations

Invocation granularity: In assessing how best to use GPU system calls, several questions arise. The first and most important question is – how should system calls be aligned with the hierarchy of GPU execution groups? Should a GPU system call be invoked separately for each work-item, once for every work-group, or once for the entire GPU kernel?

Consider a GPU program that writes sorted integers to a single output file. One might, at first blush, invoke the *write* system call at each work-item. This can present correctness issues, however, because *write* is position-relative and requires access to a global file pointer. Without synchronizing the execution of *write* across work-items, the output file will be written in a non-deterministic unsorted order.

Using different system call invocation granularities can fix this issue. One could, for example, use a memory location to temporarily buffer the sorted output. Once all work-items have updated this buffer, a single *write* system call at the end of the kernel can be used to write the contents of the buffer to the output file. This approach loses some benefits of the GPU's parallel resources, because the entire system call latency is exposed on the program's critical path and might not be overlapped with the execution of other work-items. Alternatively, one could use *pwrite* system calls instead of *write* system calls. Because *pwrite* allows programmers to specify the absolute file position where the output is to be written, per-work-item invocations present no correctness issue. However, per-work-item *pwrites* result in a flood of system calls, potentially harming performance.

Overly coarse kernel-grained invocations also restrict performance by reducing the possibility of overlapping system call processing with GPU execution. A compromise may be to invoke a *pwrite* system call per work-group, buffering the sorted output

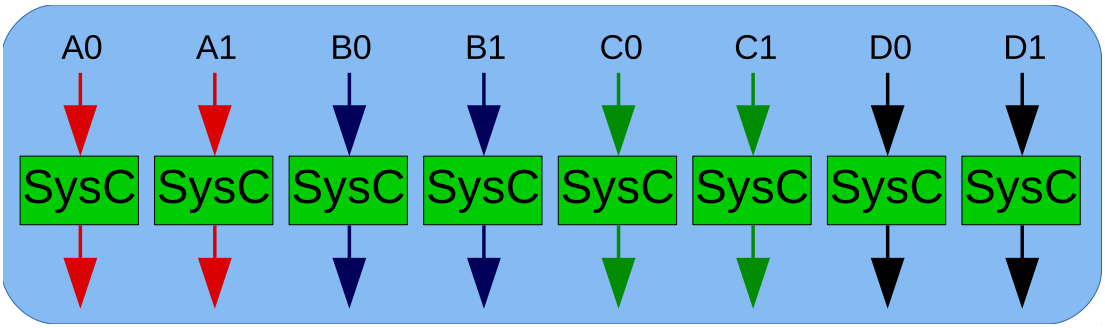


Figure 3.3: Work-items in a work-group (shown as a blue box) execute strongly ordered system calls.

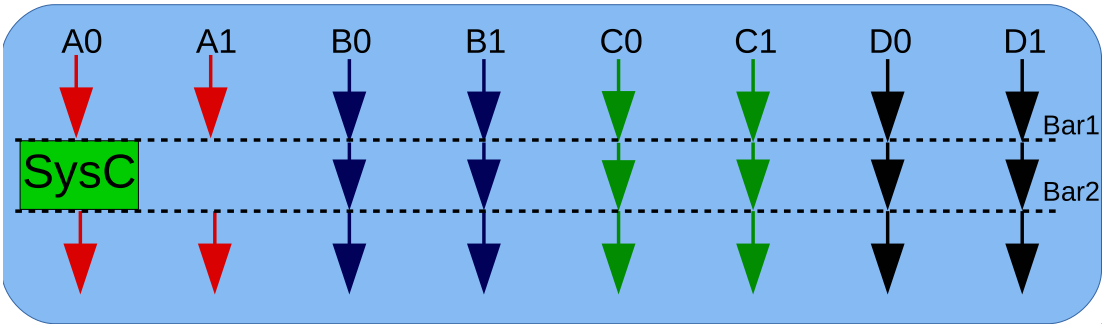


Figure 3.4: Work-group invocations can be relax-ordered by removing one of the two barriers.

of the work-items until the per-work-group buffers are fully populated. Section 3.7 demonstrates that these decisions can lead to a 1.75× performance difference.

System call ordering semantics: When programmers invoke system calls on CPUs, they expect that all program instructions before the system call will complete execution before the system call executes. They also expect that instructions after the system call will only commence once the system call returns. We call this "strong ordering." For GPUs however, we introduce "relaxed ordering" semantics. The notion of relaxed ordering is tied to the hierarchical execution scopes of the GPU and is needed for both correctness and performance.

Figure 3.3 shows a programmer-visible work-group (in blue), consisting of four wavefronts A, B, C and D. Each wavefront has two work-items (e.g., A0 and A1). If system calls (SysCs) are invoked per work-item, they are strongly ordered. Another approach is depicted in Figure 3.4, where one work-item, A0, invokes a system call, *on behalf of* the entire work-group. Strong ordering is achieved by placing work-group barriers (Bar1,

Bar2) before and after system call execution. One can remove these barriers with relaxed ordering, allowing threads in wavefronts B, C, and D to execute overlapping with the CPU's processing of A's system call.

For correctness, we need to allow programmers to use relaxed ordering when system calls are invoked at kernel granularity (across work-groups). This is because kernels can consist of more work-items than can concurrently execute on the GPU. For strongly ordered system calls at the kernel-level, all kernel work-items must finish executing pre-invocation instructions prior to invoking the system call. But all work-items cannot execute concurrently because GPU runtimes do not preemptively context switch work-items of the same kernel. It is not always possible for all work-items to execute all instructions prior to the system call. Strong ordering at kernel granularity risks deadlocking the GPU.

At the work-group invocation granularity, relaxed ordering can improve performance by avoiding synchronization overheads and overlapping CPU-side system call processing with the execution of other work-items. The key is to remove the barriers Bar1/Bar2 in Figure 3.4. To do this, consider that from the application point of view, system calls are usually producers or consumers of data. Take a consumer call like *write*, invoked at the work-group level. Real-world GPU programs may use multiple work-items to generate the data for the write, but instructions after the *write* call typically do not depend on the *write* outcome. Therefore, other work-items in the group need not wait for the completion of the system call, meaning that we can remove Bar2, improving performance. Similarly, producer system calls like *read* typically require system calls to return before other work-items can start executing program instructions post-*read*, but do not necessarily require other work-items in the work-group to finish executing all instructions before the *read* invocation. Bar1 in Figure 3.4 becomes unnecessary in these cases. The same observations apply to per-kernel system call invocations that need relaxed ordering for correctness anyway.

In summary, work-item invocations imply strong ordering. Programmers balance performance/programmability for work-group invocations by choosing strong or relaxed ordering. Finally, programmers must use relaxed ordering for kernel invocations so that

the GPU does not deadlock.

Blocking versus non-blocking approaches: Most traditional CPU system calls – barring those for asynchronous I/O (e.g., *aio_read*, *aio_write*) – return only after the system call completes execution. Blocking system calls have a disproportionate impact on performance because GPUs use SIMD execution. In particular, if even a single work-item is blocked on a system call, no other work-item in the wavefront can make progress either. We find that GPUs can often benefit from *non-blocking* system calls that can immediately return before system call processing completes. Non-blocking system calls can therefore overlap system call processing on the CPU with useful GPU work, improving performance by as much as 30% in some of our studies (see Section 3.7).

The concepts of blocking/non-blocking and strong/relaxed ordering are related but orthogonal. The strictness of ordering refers to the question of when a system call can be invoked with respect to the progress of work-items within its granularity of invocation. System call blocking refers to how the return from a system call relates to the completion of its processing. Relaxed ordering and non-blocking can be combined in several useful ways. For example, consider a case where a GPU program writes to a file at work-group granularity. The execution may not depend upon the output of the write, but the programmer may want to ensure that the write successfully completes. In such a scenario, blocking *writes* may be invoked with weak ordering. Weak ordering permits all but one wavefront in the work-group to proceed without waiting for completion of the *write* (see Section 3.6). Blocking invocation, however, ensures that one wavefront waits for the *write* to complete and can raise an alarm if the write fails. Consider another scenario, where a programmer wishes to prefetch data using *read* system calls but may not use the results immediately. Here, weak ordering with non-blocking invocation is likely to provide the best performance without breaking the program's semantics. In short, different combinations of blocking and ordering enable programmers to fine-tune performance and programmability tradeoffs.

3.5.2 CPU Hardware

GPUs rely on extreme parallelism for performance. This means there may be bursts of system calls that CPUs need to process. System call coalescing is one way to increase the throughput of system call processing. The idea is to collect several GPU system call requests and batch their processing on the CPU. The benefit is that CPUs can manage multiple system calls as a single unit, scheduling a single software task to process them. This reduction in task management, scheduling, and processing overheads can often boost performance (see Section 3.7). Coalescing must be performed judiciously as it improves system call processing throughput at the expense of latency. It also implicitly serializes the processing of system calls within a coalesced bundle.

To allow the GPGPU programmer to balance the benefits of coalescing with its potential challenges, GENESYS accepts two parameters – a time window length within which the CPU coalesces system calls, and the maximum number of system call invocations that can be coalesced within the time window. Section 3.7 shows that system call coalescing can improve performance by as much as 10-15%.

3.5.3 CPU-GPU Communication Hardware

Prior work implemented system calls using polling, where GPU wavefronts monitored predesignated memory locations populated by CPUs upon system call completion. But recent advances in GPU hardware enable alternate modes of CPU-GPU communication. For example, AMD GPUs now allow wavefronts to relay interrupts to CPUs and then halt execution, relinquishing SIMD hardware resources [126]. CPUs can in turn message the GPU to wake up halted wavefronts.

We have implemented polling and halt-resume approaches in GENESYS. With polling, if the number of memory locations that needs to be polled by the GPU exceeds its cache size, frequent cache misses lower performance. On the other hand, halt-resume has its own overheads, namely the latency to resume a halted wavefront.

We have found that polling yields better performance when system calls are invoked

Table 3.3: System configuration used for our studies.

SoC	Mobile AMD FX-9800P™
CPU	4× 2.7GHz AMD Family 21h Model 101h
Integrated-GPU	758 MHz AMD GCN 3 GPU
Memory	16 GB Dual-Channel DDR4-1066MHz
Operating system	Fedora 26 using ROCm stack 1.6 (based on Linux 4.11)
Compiler	HCC-0.10.17166 + LLVM 5.0 C++AMP with HC extensions

at coarser work-group granularities since fewer memory locations are needed to convey information at the work-group versus work-item level. Consequently, the memory locations easily fit in the GPU's L2 data cache. When system calls are invoked per work-item however, the sheer number of such memory locations becomes so high that cache thrashing becomes an issue. In these situations, halt-resume approaches outperform polling. We quantify the impact of this in the following sections.

3.6 Implementing GENESYS

We implemented Genesys on the system in Table 3.3. We used an AMD FX-9800P processor with an integrated GPU and ran the open-source Radeon Open Compute (ROCm) software stack [137]. Although we use a system with integrated GPU, Genesys is not specific to integrated GPUs, and generalizes to discrete GPUs. We modified the GPU driver and Linux kernel to enable GPU system calls. We also modified the HCC compiler to permit GPU system call invocations in C++AMP.

Invoking GPU system calls: Genesys permits work-item, work-group, and kernel-level invocation. At work-group or kernel-level invocations, a single work-item is designated as the caller. For strongly ordered work-group invocations, we use work-group scope barriers before and after system call invocations. For relaxed ordering, a barrier is placed either before (for consumer system calls) or after (for producer calls) system call invocation.

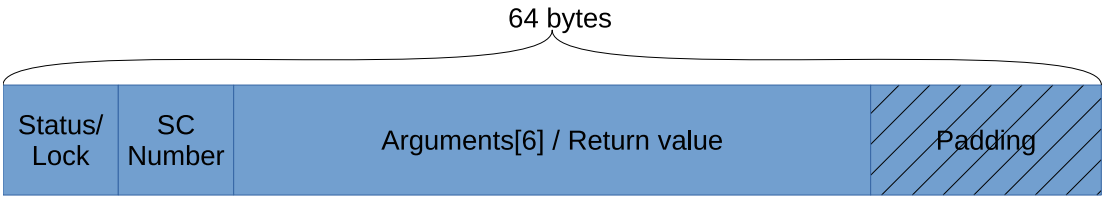


Figure 3.5: Content of each slot in syscall area.

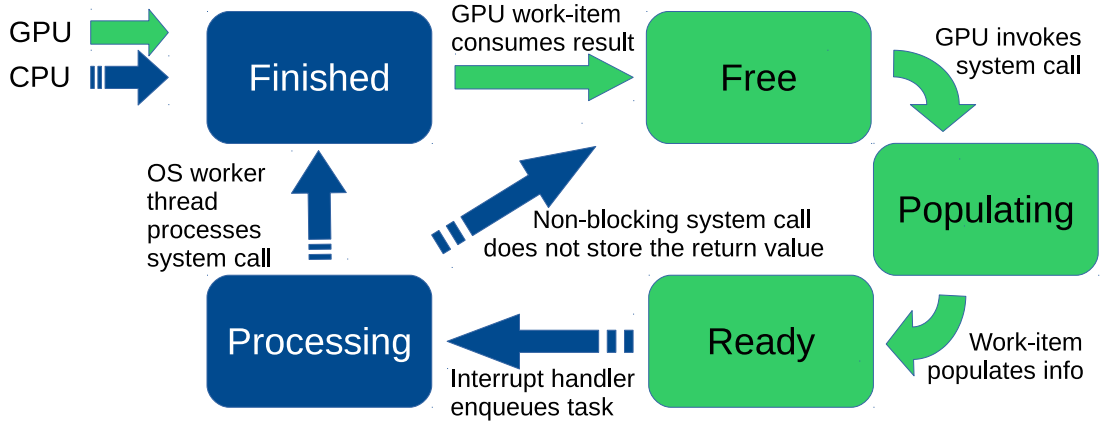


Figure 3.6: State transition diagram for a slot in syscall area. Green shows GPU state/actions, blue shows that of the CPU.

GPU to CPU communication: GENESYS facilitates efficient GPU to CPU communication of system call requests. GENESYS uses a preallocated shared memory *syscall area* to allow the GPU to convey parameters to the CPU (see Section 3.3). The *syscall area* maintains one *slot* for each active GPU work-item. The OS and driver code can identify the desired slot by using the hardware ID of the active work-item, which is available to GPU runtimes. This hardware ID is distinct from the programmer-visible work-item ID. Each of the work-items has a unique work-item ID visible to the application programmer. At any one point in time, however, only a subset of these work-items executes (as permitted by the GPU's CU count, supported work-groups per CU, and SIMD width). The hardware ID distinguishes among these active work-items. Overall, our system uses 64 bytes per slot, totaling 1.25 MBs of *syscall area*.

Figure 3.5 shows the contents in a slot – the requested system call number, the request state, system call arguments (as many as 6, in Linux), and padding to avoid false sharing. The field for arguments is also re-purposed for the return value of the system call. When a GPU program's work-item invokes a system call, it atomically updates the state of the corresponding slot from *free* to *populating* (Figure 3.6). If

the slot is not *free*, system call invocation is delayed. Once the state is *populating*, the invoking work-item populates the slot with system call information and changes the state to *ready*. The work-item also adds one bit of information about whether the invocation is blocking or non-blocking. The work-item then interrupts the CPU using a *scalar wavefront GPU instruction* [3] (*s_sendmsg* on AMD GPUs). For blocking invocation, the work-item either waits and polls the state of the slot or suspends itself using halt-resume.

CPU-side system call processing: Once the GPU interrupts the CPU, system call processing commences. The interrupt handler creates a new kernel task and adds it to Linux's work-queue. This task is also passed the hardware ID of the requesting wavefront. At an expedient future point in time an OS worker thread executes this task. The task scans the 64 *syscall slots* of the given hardware ID and atomically switches any *ready* system call requests to the *processing* state. The task then carries out the system call work.

A challenge is that Linux's traditional system call routines implicitly assume that they are to be executed within the context of the original process invoking the system call. Consequently, they use context specific variables (e.g., the *current* variable used to identify the process context). This presents a challenge for GPU system calls, which are instead serviced purely in the context of the OS' worker thread. GENESYS overcomes this challenge in two ways – it either switches to the context of the original CPU program that invoked the GPU kernel, or it provides additional context information in the code performing system call processing. The exact strategy is determined on a case-by-case basis.

GENESYS implements coalescing by waiting for a predetermined amount of time in the interrupt handler before enqueueing a task to process a system call. If multiple requests are made to the CPU during this time window, they are coalesced with such that they can be handled as a single unit of work by the OS worker-thread. GENESYS uses Linux's *sysfs* interface to communicate coalescing parameters.

---

[3] Scalar wavefront instructions are part of the Graphics Core Next ISA and are executed once for the entire wavefront, rather than for each active work-item. See Chapter 4.1 in the manual [126].

Communicating results from the CPU to the GPU: Once the CPU worker-thread finishes processing the system call, the results are put in the field for arguments in the slot for blocking requests. Further, the thread also changes the state of the slot to *finished* for blocking system calls. For non-blocking invocations, the state is changed to *free*. The invoking GPU work-item is then re-scheduled (on hardware that supports wavefront suspension) or automatically restarted because it was polling on this state. The work-item can consume the result and continue execution.

Other architectural design considerations: GENESYS requires two key data structures to be exchanged between GPUs and CPUs – the *syscall area* and for some system calls, *syscall buffers* that maintain data required for system call processing. We discovered that carefully leveraging architectural support for CPU-GPU cache coherence in the context of these data structures was vital to overall performance.

Consider, for example, the *syscall area*. As described in Section 3.3, every GPU work-item invoking a system call is allocated space in the *syscall area*. Like many GPUs, the one used as our experimental platform supports L2 data caches that are coherent with CPU caches/memory but integrates non-coherent L1 data caches. At first blush, one may decide to manually invalidate L1 data cache lines. However, we sidestepped this issue by restricting per-work-item slots in the *syscall area* to individual cache lines. This design permits us to use atomic instructions to access memory – these atomic instructions, by design, force lookups of the L2 data cache and guarantee visibility of the entire cacheline, sidestepping the coherence challenges of L1 GPU data caches. We quantify the measured overheads of the atomic operations we use for GENESYS in Table 3.4, comparing them to the latency of a standard load operation. Through experimentation, we achieved good performance using *cmp-swap* atomics to claim a slot in the *syscall area* when GPU work-items invoked system calls, *atomic-swaps* to change state, and *atomic-loads* to poll for completion.

Unfortunately, the same approach of using atomics does not yield good performance for accesses to *syscall buffers*. The key issue is that *syscall buffers* can be large and span multiple cache lines. Using atomics here meant that we suffered the latency of

Table 3.4: Profiled performance of GPU atomic operations.

Op	cmp-swap	swap	atomic-load	load
Time(us)	1.352	0.782	0.360	0.243

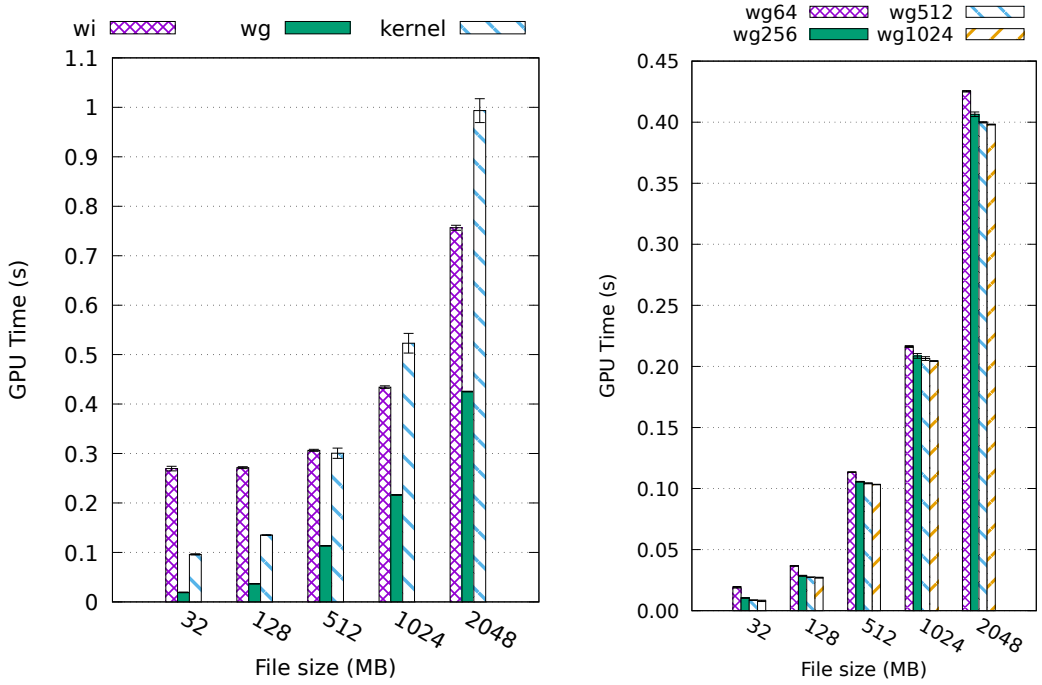


Figure 3.7: Impact of system call invocation granularity. The graphs show average and standard deviation of 10 runs.

several L2 data cache accesses to *syscall buffers*. We found that a better approach was to eschew atomics in favor of manual software L1 data cache coherence. This meant, for example, that we preceded *sys_ write* system calls with L1 data cache flush.

3.7 Microbenchmark Evaluations

Invocation granularity: Figure 3.7 quantifies performance for a microbenchmark that uses *pread* on files in *tmpfs*[4]. The x-axis plots the file size, and y-axis shows read time, with lower values being better. Within each cluster, we separate runtimes for different *pread* invocation granularities.

Figure 3.7(left) shows that work-item invocation granularities tend to perform worst. This is not surprising as it is the finest granularity of system call invocation and leads

[4] *Tmpfs* is a filesystem without permanent backing storage. In other words, all structures and data are memory-resident.

to a flood of individual system calls that overwhelm the CPU. On the other end of the granularity spectrum, kernel-level invocation is also challenging as it generates a single system call and fails to leverage any potential parallelism in processing of system call requests. This is particularly pronounced at large file sizes (e.g., 2GB). A good compromise is to use work-group invocation granularities. It does not overwhelm the CPU with system call requests while still being able to exploit parallelism in servicing system calls.

When using work-group invocation, an important question is how many work-items should constitute a work-group. While Figure 3.7(left) uses 64 work-items in a work-group, Figure 3.7(right) quantifies the performance impact of *pread* system calls as we vary work-group sizes from 64 (wg64) to 1024 (wg1024) work-items. In general, larger work-group sizes enable better performance, as there are fewer unique system calls necessary to handle the same amount of work.

Blocking and ordering strategies: To quantify the impact of blocking/non-blocking with strong/relaxed ordering, we designed a GPU microbenchmark that performs block permutation on an array, similar to the permutation steps performed in DES encryption. The input data array is preloaded with random values and divided into 8KB blocks. Work-groups each execute 1024 work-items independently permute blocks. The results are written to a file using *pwrite* at work-group granularity. The *pwrite* system calls for one block of data are overlapped with permutations on other blocks of data. To vary the amount of computation per system call, we permute multiple times before writing the result.

Figure 3.8 quantifies the impact of using blocking versus non-blocking system calls with strong and weak ordering. The x-axis plots the number of permutation iterations performed on each block by each work-group before writing the results. The y-axis plots execution time for one permutation (lower is better). Figure 3.8 shows that strongly ordered blocking invocations (*strong-block*) hurt performance. This is expected as they require work-group scoped barriers to be placed before and after *pwrite* invocations. The GPU's hardware resources for work-groups are stalled waiting for the *pwrite* to complete.

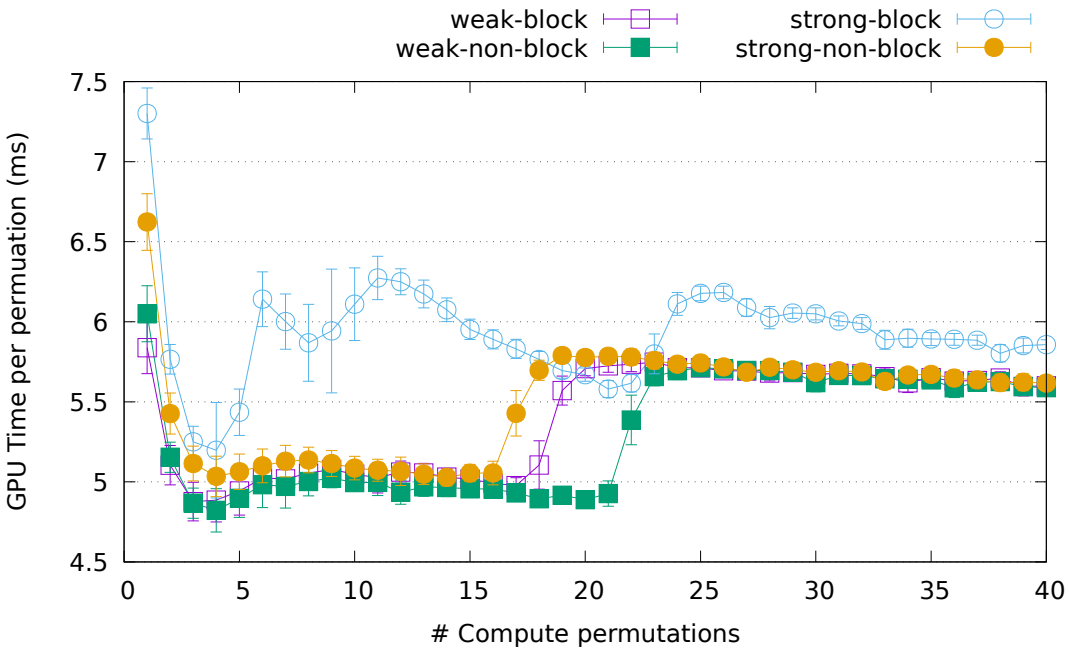


Figure 3.8: Performance implications of system call blocking and ordering semantics. The graph shows average and standard deviation of 80 runs.

Not only does the inability to overlap GPU parallelism hurt strongly ordered blocking performance, it also means that GPU performance is heavily influenced by CPU-side performance vagaries like synchronization overheads, servicing other processes, etc. This is particularly true at iteration counts where system call overheads dominate GPU-side computation – below 15 compute iterations. Even when the application becomes GPU-compute bound, performance remains non-ideal.

Figure 3.8 shows that when *pwrite* is invoked in a non-blocking manner (with strong ordering), performance improves. This is because non-blocking *pwrites* permit the GPU to end the invoking work-group, freeing GPU resources it was occupying. CPU-side *pwrite* processing can overlap with the execution of another work-group permuting on a different block of data. Figure 3.8 shows that latencies generally drop by 30% compared to blocking calls at low iteration counts. At higher iteration counts (beyond 16), these benefits diminish because the latency to perform repeated permutations dominates any system call processing times.

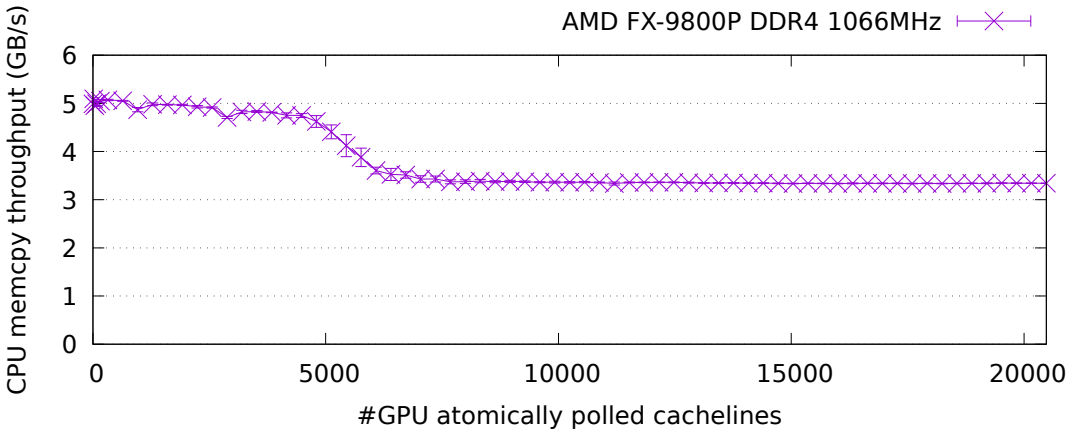


Figure 3.9: Impact of polling on memory contention.

For relaxed ordering with blocking (*weak-block*), the post-system-call work-group-scope barrier is eliminated. One out of every 16 wavefronts[5] in the work-group waits for the blocking system call, while others exit, freeing GPU resources. The GPU can use these freed resources to run other work-items to hide CPU-side system call latency. Consequently, the performance trends follow those of *strong-non-block*, with minor differences in performance arising from differences GPU scheduling of the work-groups for execution. Finally, Figure 3.8 shows system call latency is best hidden using relaxed and non-blocking approaches (*weak-non-block*).

Polling/halt-resume and memory contention: As previously discussed, polling at the work-item invocation granularity leads to memory reads of several thousands of per-work-item memory locations. We quantify the resulting memory contention in Figure 3.9, which showcases how the throughput of CPU accesses decreases as the number of polled GPU cache lines increases. Once the number of polled memory locations outstrips the GPU's L2 cache capacity (roughly 4096 cache lines in our platform), the GPU polls locations spilled to the DRAM. This contention on the memory controllers shared between GPUs and CPUs. We advocate using GENESYS with halt-resume approaches at such granularities of system call invocation.

Interrupt coalescing: Figure 3.10 shows the performance impact of coalescing system calls. We use a microbenchmark that invokes *pread*. We read data from files of different

[5]Each wavefront has 64 work-items. Thus, a 1024 work-item work-group has 16 wavefronts.

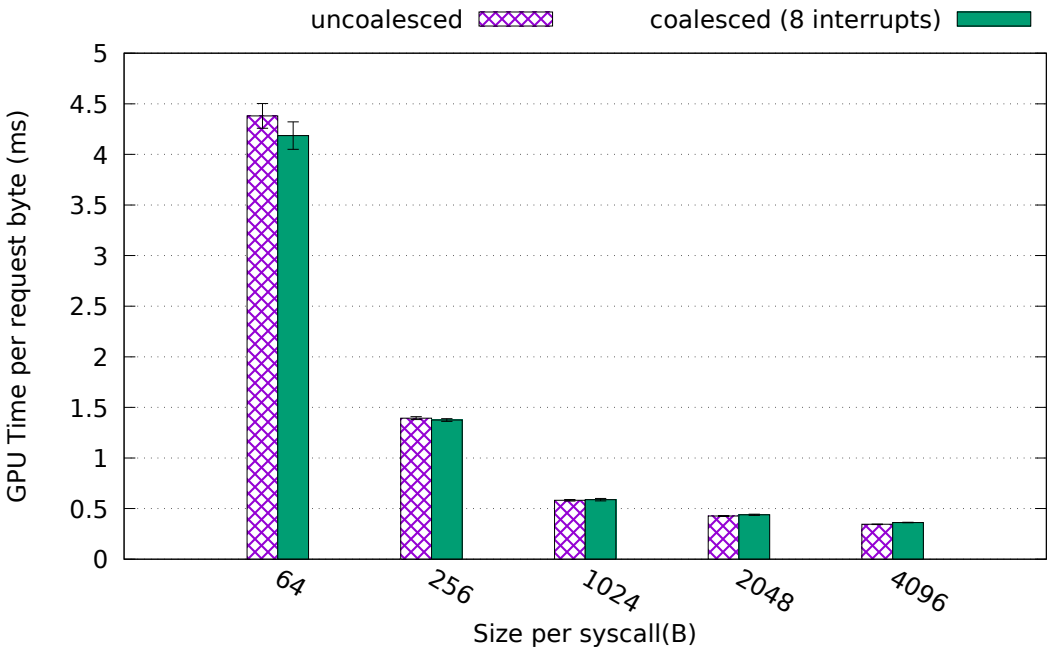


Figure 3.10: Implications of system call coalescing. The graph shows average and standard deviation of 20 runs.

sizes using a constant number of work-items. More data is read per *pread* system call from the larger files. The x-axis shows the amounts of data read and quantifies the latency per requested byte. We present two bars for each point on the x-axis, illustrating the average time needed to read one byte with the system call in the absence of coalescing and when up to eight system calls are coalesced. Coalescing is most beneficial when small amounts of data are read. Reading more data takes longer; the overhead reduction is negligible compared to the significantly longer time to process the system call.

3.8 Case Studies

3.8.1 Memory Workload

We now assess the end-to-end performance of workloads that use Genesys. Our first application requires memory management system calls. We studied miniAMR [138] and used the *madvise* system call directly from the GPU to better manage memory. MiniAMR performs 3D stencil calculations using adaptive mesh refinement and is a candidate for memory management because it varies its memory needs in a data-model-dependent manner. For example, when simulating regions experiencing turbulence,

miniAMR needs to model with higher resolution. However, if lower resolution is possible without compromising simulation accuracy, miniAMR reduces memory and computation usage, making it possible to free memory. While relinquishing excess memory in this way is not possible in traditional GPU programs without explicitly dividing the offload into multiple kernels interspersed with CPU code (see Figure 3.1), GENESYS permits this with direct GPU *madvise* invocations. We invoke *madvise* using work-group granularities with non-blocking and weak ordering. We leverage GENESYS' generality by also using *getrusage* to read the application resident set size (RSS). When the RSS exceeds a watermark, *madvise* relinquishes memory.

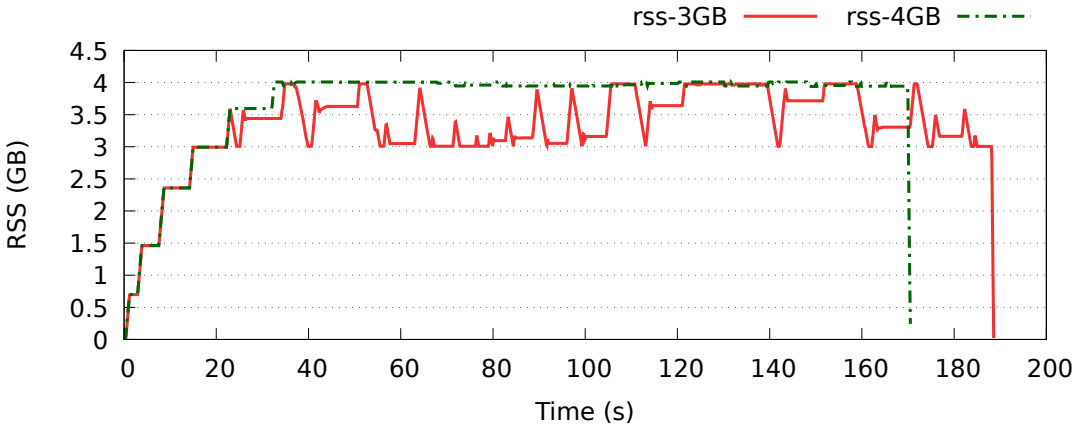


Figure 3.11: Memory footprint of miniAMR using *getrusage* and *madvise* to hint at unused memory.

We execute miniAMR with a dataset of 4.1GB – just beyond the hard limit we put on physical memory available to GPU. Without using *madvise*, memory swapping increases so dramatically that it triggers GPU timeouts, causing the existing GPU device driver to terminate the application. Because of this, there is no baseline to compare to as the baseline simply does not complete.

Figure 3.11 shows two results: one for a 3GB RSS watermark, and one for 4GB. Not only does GENESYS enable miniAMR to complete, it also permits the programmer to balance memory usage and performance. While *rss-3GB* lowers memory utilization, it also worsens runtime compared to *rss-4GB*. This performance gap is expected; the more memory is released, the greater the likelihood that the GPU program suffers from page faults when the memory is touched again in the future, and the more frequent the

madvise system call is invoked. Overall, Figure 3.11 illustrates that GENESYS allows programmers to perform memory allocation to trade memory usage and performance on GPUs analogous to CPUs.

3.8.2 Workload Using Signals

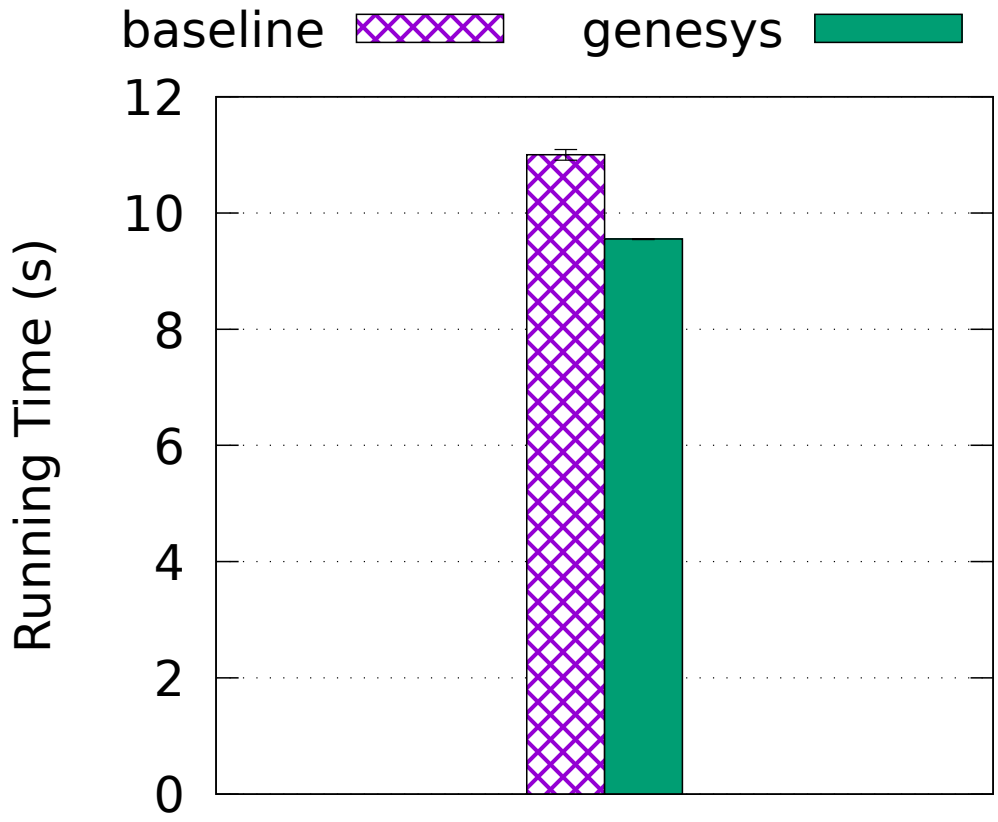


Figure 3.12: Runtime of CPU- GPU map reduce workload.

GENESYS also enables system calls that permit the GPU to send asynchronous notifications to the CPU. This is useful in many scenarios. We study one such scenario and implement a map-reduce application called signal-search. The application runs in two phases. The first phase performs parallel lookup in a data array, while the second phase processes blocks of retrieved data and computes sha512 checksums on them. The first phase is a good fit for GPUs since the highly parallel lookup fits its execution model, while the second phase is more appropriate for CPUs, many of which support performance acceleration of sha checksums via dedicated instructions.

Without support for signal invocation on GPUs, programmers would launch a kernel with the entire parallel lookup on the GPU and wait for it to conclude before proceeding with the sha512 checksums. GENESYS overcomes this restriction and permits a heterogeneous version of this code, where GPU work-groups can emit signals using *rt_sigqueueinfo* to the CPU, indicating per-block completions of the parallel search. As a result, the CPU can begin processing these blocks immediately, permitting overlap with GPU execution.

Operationally, *rt_sigqueueinfo* is a generic signal system call that allows the caller to fill a *siginfo* structure that is passed along with the signal. In our workload, we find that work-group level invocations perform best, so the GPU passes the identifier of this work-group through the *siginfo* structure. Figure 3.12 shows that using work-group invocation granularity and non-blocking invocation results in roughly 14% performance speedup over the baseline.

3.8.3 Storage Workloads

We have also studied how to use GENESYS to support storage workloads. In some cases, GENESYS permits the implementation of workloads supported by GPUfs, but in more efficient ways.

Supporting storage workloads more efficiently than GPUfs: We implement a workload that takes as input a list of words and a list of files. It then performs *grep -F -l* on the GPU, identifying which files contain any of the words on the list. As soon as these files are found, they are printed to the console. This workload cannot be supported by GPUfs without significant code refactoring because of its use of custom APIs. Instead, since GENESYS naturally adheres to the UNIX "everything is a file" philosophy, porting *grep* to GPUs requires only a few hours of programming.

Figure 3.13 shows the results of our GPU *grep* experiments. We compare a standard CPU implementation, a parallelized OpenMP CPU implementation, and two GPU implementations with GENESYS, with non-blocking system calls invoked at work-item (WI) and work-group (WG) granularities. Furthermore, since work-item invocations can

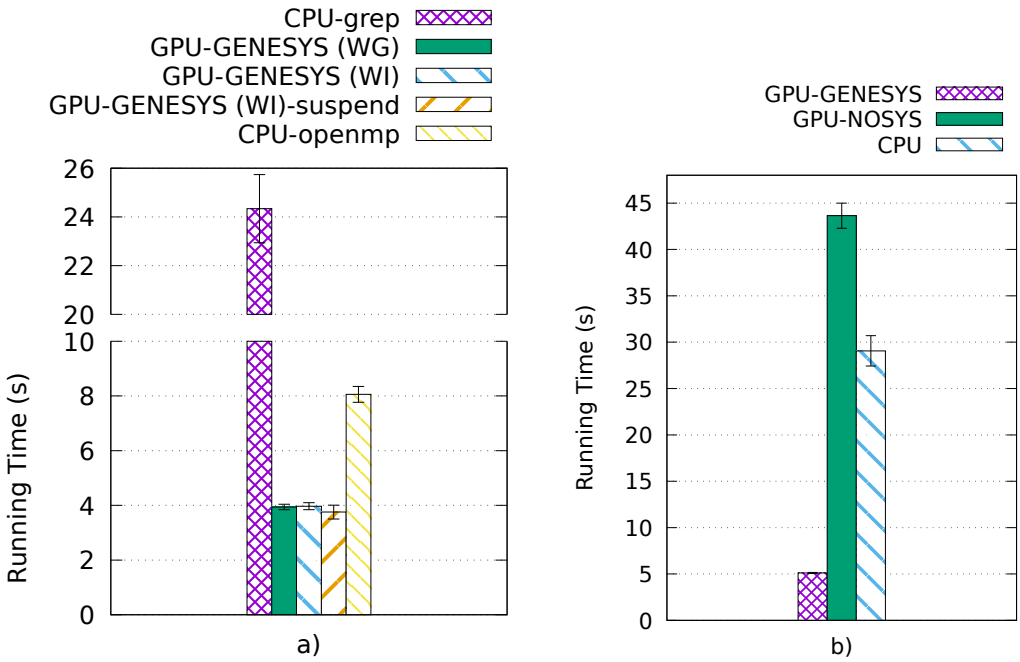


Figure 3.13: a) Standard *grep*, OpenMP *grep*, versus work-item/work-group invocations with GENESYS. b) Comparing the performance of CPU, GPU with no system call, and GENESYS implementations of wordcount. Graphs show average and standard deviation of 10 runs.

sometimes achieve better performance using halt-resume (versus work-group and kernel invocations, which always achieve better performance with polling), we separate results for WI-polling and WI-halt-resume. GENESYS enables our GPU *grep* implementation to print output lines to console or files using standard OS output. GENESYS achieves 2.0-2.3× speedups over an OpenMP version of *grep*.

Figure 3.13 also shows that GENESYS' flexibility with invocation granularity, blocking/non-blocking, and strong/relaxed ordering can boost performance (see Section 3.5). For our *grep* example, because a file only needs to be searched until the first instance of a matching word, a work-item can immediately invoke a *write* of the filename, rather than waiting for all matching files. We find that WI-halt-resume outperforms both WG and WI-polling by roughly 3-4%.

Workload from prior work: GENESYS also supports a version of the same workload that is evaluated in the original GPUfs work; traditional word count where using *open*, *read*, and *close* system calls. Figure 3.13 shows our results. We compare the performance of a parallelized CPU version of the workload with OpenMP, a GPU version of the

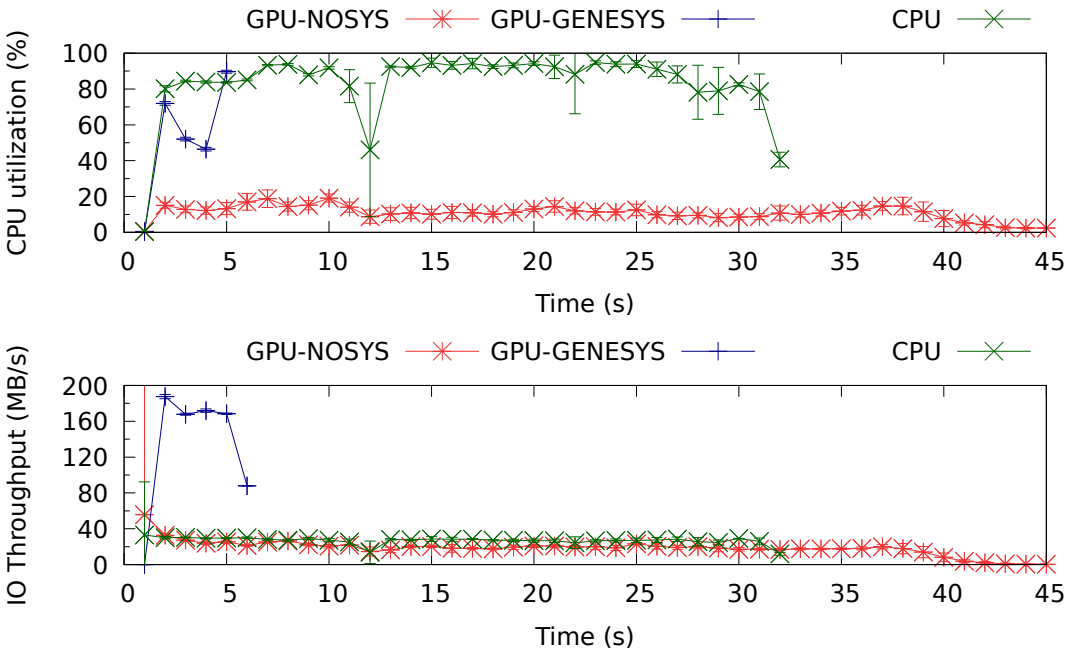


Figure 3.14: Wordcount I/O and CPU utilization reading from SSD. Graphs show average and standard deviation of 10 runs.

workload with no system calls, and a GPU version with GENESYS. Both CPU and GPU workloads are configured to search for occurrences of 64 strings. We found that with GENESYS, these system calls were best invoked at work-group granularity with blocking and weak-ordering semantics. All results are collected on a system with an SSD.

We found that GENESYS achieves nearly 6× performance improvement over the CPU version. Without system call support, the GPU version is far worse than the CPU version. Figure 3.14 sheds light on these benefits. We plot traces for CPU and GPU utilization and disk I/O throughput. GENESYS extracts much higher throughput from the underlying storage device (up to 170MB/s compared to the CPU's 30MB/s). Offloading search string processing to the GPU frees up the CPU to process system calls effectively. The change in CPU utilization between the GPU workload and CPU workload reveals this trend. In addition, we found that the GPU's ability to launch more concurrent I/O requests enabled the I/O scheduler to make better scheduling decisions.

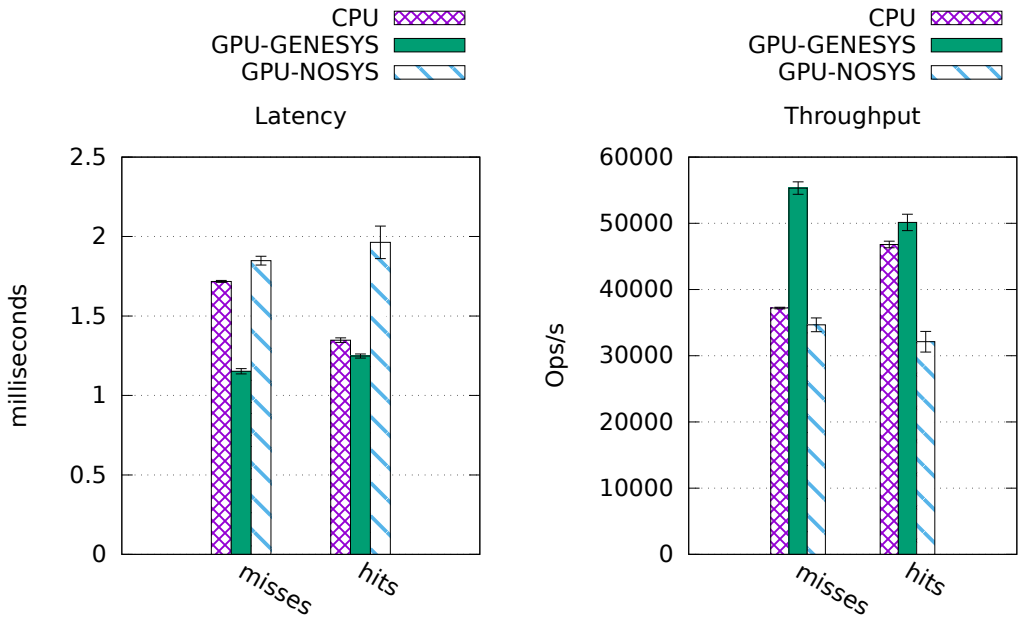


Figure 3.15: Latency and throughput of memcached. Graphs show average and standard deviation of 20 runs.

3.8.4 Network Workloads

We have studied the benefits of GENESYS for network I/O in the context of *memcached*. While this can technically be implemented using GPUnet, the performance implications are unclear because the original GPUnet paper used APIs that extracted performance using dedicated RDMA hardware. We make no such assumptions and focus on the two core commands – *SET* and *GET*. *SET* stores a key-value pair, and *GET* retrieves a value associated with a key if it is present. Our implementation supports a binary UDP *memcached* version with a fixed-size hash table as a back-end storage. The hash table is shared between CPU and GPU, and its bucket size and count are configurable. Further, this *memcached* implementation enables concurrent operations. CPUs can handle *SETs* and *GETs*, while the GPU supports only *GETs*. Our GPU implementation parallelizes the hash computation, bucket lookup, and data copy. We use *sendto* and *recvfrom* system calls for UDP network access. These system calls are invoked at work-group granularity with blocking and weak ordering as this performs best.

Figure 3.15 compares the performance of a CPU version of this workload with GPU versions using GENESYS. GPUs accelerate *memcached* by parallelizing lookups on buckets with more elements. For example, Figure 3.15 shows speedups when there are 1024

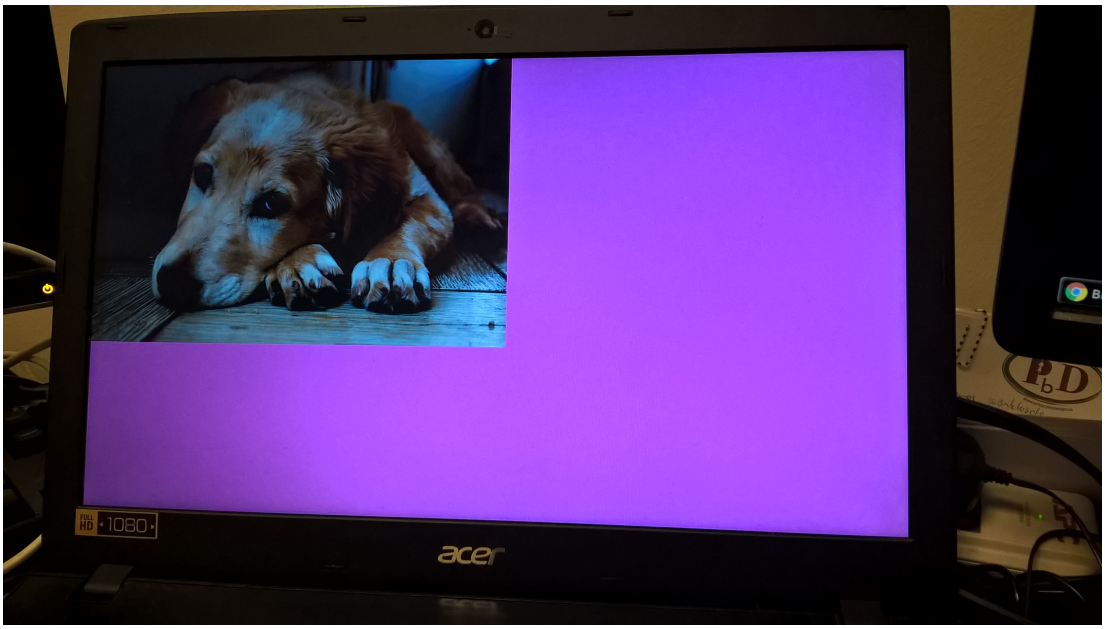


Figure 3.16: Raster image copied to the framebuffer by the GPU.

elements per bucket (with 1KB data size). Without system calls, GPU performance lags behind CPU performance. However, GENESYS achieves 30-40% latency and throughput benefits over not just CPU versions, but also GPU versions without direct system calls.

3.8.5 Device Control

Finally, we also used GENESYS to implement *ioctl* system calls. As an example, we used *ioctl* to query and control framebuffer settings. The implementation is straightforward; the GPU opens */dev/fb0*, and issues a series of *ioctl* commands to query and set settings of the active frame buffer. It then proceeds to *mmap* the framebuffer memory and fill it with data from a previously *mmaped* raster image. This results in the image displayed on computer screen. While not a critical GPGPU application, this *ioctl* example demonstrates the generality and flexibility of OS interfaces implemented by GENESYS.

3.9 Discussion

Asynchronous system call handling: GENESYS enqueues the GPU system call's kernel task and processes it outside of the interrupt context. We do this because Linux is designed such that few operations can be processed in an interrupt handler. A potential concern with this design, however, is it defers the system call processing to potentially

past the end of the life-time of the GPU thread and potentially the process that created the GPU thread itself! It is an example of a more general problem with asynchronous system calls [139]. Our solution is to provide a new function call, invoked by the CPU, that ensures all GPU system calls have completed before the termination of the process.

Related work: Beyond work already discussed [17–19], the latest generation of C++AMP [20] and CUDA [15] provide access to the memory allocator. These efforts use a user-mode service thread on the CPU to proxy requests from the GPU [31]. Like GENESYS, system call requests are placed in a shared queue by the GPU. From there, however, the designs are different. Their user-mode thread polls this shared queue and "thunks" the request to the libc runtime or OS. This incurs added overhead for entering and leaving the OS kernel.

Some studies provide network access to GPU code [18, 37, 140, 141]. NVidia provides GPUDirect [37], used by several MPI libraries [142–144], that allows the NIC to bypass main memory and communicate directly to memory on the GPU itself. GPUDirect does not provide a programming interface for GPU-side code. The CPU must initiate communication with the NIC. Oden exposed the memory-mapped control interface of the NIC to the GPU and thereby allowed the GPU to directly communicate with the NIC [141]. This low-level interface, however, lacks the benefits of a traditional OS interface (e.g., protection, sockets, TCP).

3.10 Conclusions

We shed light on research questions fundamental to the idea of accessing OS services from accelerators by realizing an interface for generic POSIX system call support on GPUs. Enabling such support requires subtle changes of existing kernels. In particular, traditional OSes assume that system call processing occurs in the same context as the invoking thread, and this needs to change for accelerators. We have released GENESYS to make these benefits accessible for broader research on GPUs.

3.11 Acknowledgments

We thank the National Science Foundation, which partially supported this work through grants 1253700 and 1337147. We thank Guilherme Cox and Mark Silberstein for their feedback.

Chapter 4

Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems

4.1 Introduction

Computing has witnessed a proliferation of accelerators serving a diverse set of needs like cryptography, graphics, databases, regular expressions [145–147]. Accelerators often enable significantly superior performance and power efficiency for specific tasks, compared to general-purpose CPUs. Consequently, accelerators are increasingly being integrated with the CPU on the same die. Such tight integration allows efficient offloading of computation to accelerators. Commercial manufacturers like AMD, Apple, Intel, and Qualcomm, today ship millions of processors with CPUs and GPUs tightly coupled together on a single die. IBM's CAPI [148] and ARM®'s ACP [149] extend this concept to a plethora of third-party accelerators.

A key challenge in harnessing the full potential of accelerators in these heterogeneous processors is to make them easily programmable. A crucial first step in addressing this challenge is to enable shared (unified) coherent virtual memory across CPUs and accelerators. Indeed, industry-promoted models such as the Heterogeneous System Architecture (HSA) [150] mandate shared virtual memory in an effort to make accelerators a first-class computing element. This allows a pointer on the CPU to be equally valid on an accelerator; thus avoiding manual data copies. Importantly, it helps provide a programming environment familiar to common programmers.

In this work, we present a detailed characterization and analysis of a shared virtual memory system across the CPU and the integrated GPU in a commercially available heterogeneous processor. To the best of our knowledge, this is the first such study on a commercial platform. We study how the CPU's virtual memory is extended to the

integrated-GPU in three key aspects: (1) virtual-to-physical address translation; (2) creation of new virtual-to-physical address mappings through page faults; and (3) invalidation of stale address mappings through TLB (Translation Lookaside Buffer) shootdowns. We use six applications that use the integrated GPU to perform computation. Although we focus on the GPU, our key results are applicable to other throughput-orientated accelerators.

Our measurements highlight several new research opportunities in this space. (1) A TLB miss on the GPU can be 25× slower than that on the CPU. Research into enabling greater concurrency in servicing TLB misses from throughput-oriented accelerators is critical to hide this latency; (2) Poor locality in memory accesses, particularly from divergent accesses within the same wavefront or warp, impacts the GPU's address translation hardware more than the rest of the memory hierarchy. Research into divergence tolerant address translation mechanisms is important; (3) The prefetching of address translations built into the industry standard PCIe® specification may hurt performance under certain circumstances. More research into effective translation prefetching for accelerators is necessary; (4) Servicing GPU page faults can be significantly slower than that for CPU page faults. Most of this added time is spent in the operating system (OS) kernel, and likely can be designed out; (5) The latency of a TLB shootdown on the GPU is comparable to that in the CPU. Research into reducing TLB shootdown latency in heterogeneous systems needs to optimize both the CPU and GPU shootdown. Several other such observations and research opportunities are detailed throughout this work. Section 4.2 provides background on how integrated-GPUs perform basic virtual-memory operations. Section 4.3 describes our methodology and applications used. Section 4.4 describes our analysis of address translation while Section 4.5 and Section 4.6 explore page fault and shootdown costs.

4.2 Background: Shared Virtual Memory

Figure 4.1 depicts the system that we analyze in this work. A hardware block called the IO Memory Management Unit (IOMMU) services the address translation requests

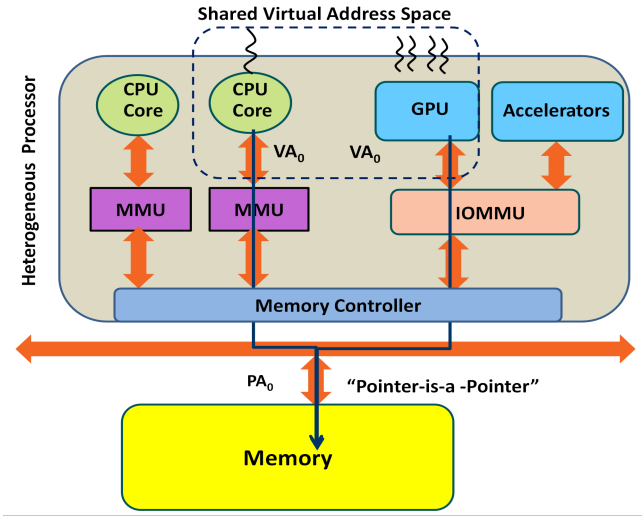


Figure 4.1: Heterogeneous system enabling shared virtual memory

of the GPU and other accelerators. The IOMMU resides in the processor's north-bridge complex. The IOMMU can access the same x86-64 page table structures used by processes running on the CPU. This enables the accelerator to share the same set of page tables (and thus the same virtual address space) as the processes running on the CPU via the IOMMU. A software driver in the OS executing on the CPU manages the IOMMU. The runtime software, in coordination with the OS driver, sets up the IOMMU to enable accelerators access to the same virtual address spaces of the CPU. In the following subsections we describe how basic virtual-memory operations are performed by the integrated-GPU in this environment. Although the following operations are explained for the GPU, similar mechanisms are applicable to other accelerators and IO devices.

4.2.1 GPU address translation

The GPU has its own TLB hierarchy that caches recently used address translations. On a GPU TLB miss, a translation request is sent as an ATS (Address Translation Service [22]) request packet over the PCIe®-based [27] internal interconnect to the IOMMU. This interconnect carries PCIe® packets but latency and bandwidth are not necessarily constrained by PCIe®'s electrical specifications. The IOMMU has its own TLB hierarchy which is checked first; on a miss there, a hardware page table walker

in the IOMMU checks the page table. ATS requests are tagged with a process address space identifier (PASID) and the IOMMU maintains a table that matches PASIDs to page table base physical addresses. Once the address is successfully translated, the IOMMU sends an ATS response to the GPU. The protocol and packet formats for ATS requests and responses are part of the PCIe® standard specification and are the same across all accelerators.

The PCIe®'s ATS protocol enables devices (and accelerators) to prefetch translation requests for up to eight contiguous virtual address pages in a single ATS response from the IOMMU. By default, the GPU in our system allows the prefetch value to the maximum setting of eight.

Comparison with CPUs: In the CPU, per-core Memory management Units (MMUs) are responsible for address translations. In contrast, the IOMMU services requests from all accelerators. Unlike the CPU's MMU, the IOMMU is not tightly integrated with CPU's data cache hierarchy. The data caches may contain the most up-to-date translations but the cached copies cannot be directly accessed by accelerators.

4.2.2 GPU page faults

If the IOMMU's page table walker fails to find the desired translation in the page table, it sends an ATS response to the GPU notifying it of this failure. This in turn corresponds to a page fault. In response, the GPU sends another request to the IOMMU called a Peripheral Page Request (PPR). The IOMMU places this request in a memory-mapped queue and raises an interrupt on the CPU. Multiple PPR requests can be queued before the CPU is interrupted. The OS must have a suitable IOMMU driver to process this interrupt and the queued PPR requests. In Linux, while in an interrupt context, the driver pulls PPR requests from the queue and places them in a work-queue for later processing. Presumably this design decision was made to minimize the time spent executing in an interrupt context, where lower priority interrupts would be disabled. At a later time, an OS worker-thread calls back into the driver to process page fault requests in the work-queue. Once the requests are serviced, the driver notifies the IOMMU. In turn, the IOMMU notifies the GPU. The GPU then sends another ATS request to retry

the translation for the original faulting address.

Comparison with CPU: On the CPU, a hardware exception is raised on a page fault, which immediately switches to the OS. In most cases in Linux, this routine services the page fault directly, instead of queuing it for later processing. Contrast this with a page fault from an accelerator, where the IOMMU has to interrupt the CPU to request service on its behalf, and also note the several back-and-forth messages between the accelerator, the IOMMU, and the CPU. Furthermore, page faults on the CPU are generally handled one at a time on the CPU, while for the GPU they are batched by the IOMMU and OS work-queue mechanism

4.2.3 GPU TLB shootdowns

The IOMMU plays a pivotal role in extending the TLB shootdown process to GPUs. An OS driver monitors any changes to virtual address mappings for address spaces shared with the GPU and triggers a TLB shootdown when necessary. The driver first sends a command to the IOMMU via a memory-resident command queue to invalidate the stale mapping in the IOMMU's TLB hierarchy. The driver then waits for the IOMMU to confirm successful invalidation from the IOMMU. Next, the driver commands the GPU (via the IOMMU) to invalidate the stale mapping from its TLB. The IOMMU hardware collects the completion notification of the invalidation in the GPU's TLB and forwards this information to the OS. Note there can be two types of invalidation requests: (1) requests to invalidate a given address mapping in the TLBs, and (2) requests to flush all entries for a given address space.

Comparison with CPU: Historically, CPUs have used a variety of mechanisms to perform remote TLB shootdowns. For example, x86 processors use inter-processor interrupts (IPIs) to keep per-core TLBs coherent. CPUs initiating TLB shootdowns send IPIs to other CPUs in the system that may have a stale entry. The IPI invokes the operating system on those CPUs, which executes a handler to invalidate the local per-CPU TLB. Just as the operating system can choose not to send IPIs to processors that provably cannot have a stale mapping (e.g., they never executed the process), an efficiently constructed driver will selectively send shootdowns to the IOMMU and the accelerators

only if the address space was shared.

In contrast, processors like ARM® and PowerPC® have also employed dedicated TLB shootdown instructions in their ISAs. In these cases, software on the shootdown initiator core executes a TLB invalidation instruction, which is then broadcast to the other cores. Although the OS is typically not invoked in this approach, the downside is that broadcast signals are often conservatively relayed to all system cores, constraining the scalability of this approach

4.3 Methodology and Workloads

We run our experiments on a system with an AMD A10-7850K APU (previously code-named "Kaveri") as described in Table 4.1. AMD A10-7850K APU is one of the first heterogeneous processor to support shared virtual memory across the CPU and GPU. We measured TLB events and page table walks using hardware performance counters. We designed a software profiler to access these counters. We also instrumented the Linux IOMMU driver to measure the latency for software events like page faults and TLB shootdowns.

CPU	AMD A10-7850K APU, maximum core frequency 3.7GHz.
GPU	8 compute units (CUs), maximum core frequency 720 MHz.
Memory	DDR-3, 32GB (4×8GB), 1600 MHz.
Software	Linux 4.0 with Kernel Fusion Driver (KFD). Heterogeneous System Architecture (HSA) [151] runtime, C++AMP compiler and OpenCL stack on HSA

Table 4.1: Description of experimental system

We use six applications (described in Table 4.2) and two targeted micro-benchmarks for this study. All the applications and micro-benchmarks use the on-die integrated-GPU to perform their primary compute. All the data for the applications reside in the system memory (DRAM) and uses shared virtual addressing between the CPU and the GPU.

B+Tree Search(BPT) [152]	Searches 15M keys in a B+tree concurrently on the GPU. The B+tree is pre-generated.
CoMD [153]	Molecular dynamics simulation that evaluates the force acting on an atom due to other atoms in the system. Force potential computation is evaluated on the GPU.
miniAMR [138]	Applies a stencil calculation on the GPU to a dense 3D array.
miniFE [154]	Assembles and solves a sparse linear-system from the steady-state conduction equation[155].
graph500 [156]	BFS traversal on a Kronecker generated graph. Bottom-up traversal uses the GPU.
XSBench [157]	Monte Carlo neutron transport across macroscopic neutron cross sections.

Table 4.2: Description of applications used

4.4 Address Translation in Shared Virtual Memory

We first analyze intricacies of the GPU address translation under a controlled execution environment with custom microbenchmarks and then present performance measurements of the applications.

4.4.1 Analyzing GPU's address translation using microbenchmarks

We sought to answer two questions in the analysis using a carefully designed micro-benchmark: (1) what is a typical GPU TLB miss latency, and (2) how much concurrency is supported by the hardware in servicing GPU TLB misses? Answers to these questions reveal potential performance bottlenecks, particularly as shared virtual memory is scaled in future heterogeneous systems.

Our micro-benchmark runs a kernel (GPU program) on the integrated-GPU with a varying number of workitems. Each workitem accesses 10,000 different memory locations in a loop and performs a simple computation (XOR) on the data. A stride (parameter) determines the distance (in bytes) between memory locations accessed by two consecutive accesses in each workitem. We execute the micro-benchmark with one workitem with a stride of 64 bytes (cache line size) and again with a stride of 4KB (page size). We use the hardware performance counters to count the number of TLB

Stride (in Bytes)	GPU		CPU	
	Number of TLB misses	Running time (in $\mu secs$)	Number of TLB misses	Running time (in $\mu secs$)
64 (cache line size)	166	12,707	1,412	134
4096 (page size)	10,010	18,444	15,938	477
TLB miss latency (calculated)	(18,444 - 12,707) / (10,010 - 166) **0.58**μs		(477 - 134) / (15,938 - 1,412) **0.023**μs	

Table 4.3: Measurements of TLB miss latencies

and cache misses. Table 4.3 lists the measurements. When executed with a stride of 64 bytes (second row) each memory access incurs a cache miss while every 64th access (4096/64) incurs a TLB miss. With a stride of 4KB, every access incurs a cache miss and a TLB miss (third row). Thus the difference between these two executions is the number of TLB misses. Therefore, we attribute the difference in the runtime between the two runs to the additional TLB misses. We calculate that the latency of servicing a GPU TLB miss that incurs a page walk by the IOMMU to be 582 nanoseconds (last row). We further ran the same micro-benchmark on the CPU with a single thread for comparative analysis. Measurement on the CPU is presented in the last column and we similarly calculate that latency of the CPU's TLB miss to be 23 nanoseconds. Thus, a TLB miss from the GPU is about 25% slower than on a CPU. Several reasons contribute to the longer latency of GPU TLB misses: (1) TLB miss request and responses travel as PCIe® packets to the IOMMU; (2) more levels (up to four here) of TLBs to check; (3) the IOMMU's page table walker does not have fast access to CPU caches that might have the latest page table entries; and (4) the resulting wavefront must be rescheduled for execution on the GPU.

We then executed the micro-benchmark on the GPU with an increasing number of workitems from 1 to 64 to estimate the concurrency available in servicing TLB misses. Each workitem accesses a distinct set of memory locations and thus offers no opportunity to coalesce the accesses. The maximum number of outstanding TLB misses (and the corresponding page walk requests) at any given time is thus bounded by the number of workitems. Figure 4.2 shows how the latency experienced by a wavefront on a GPU

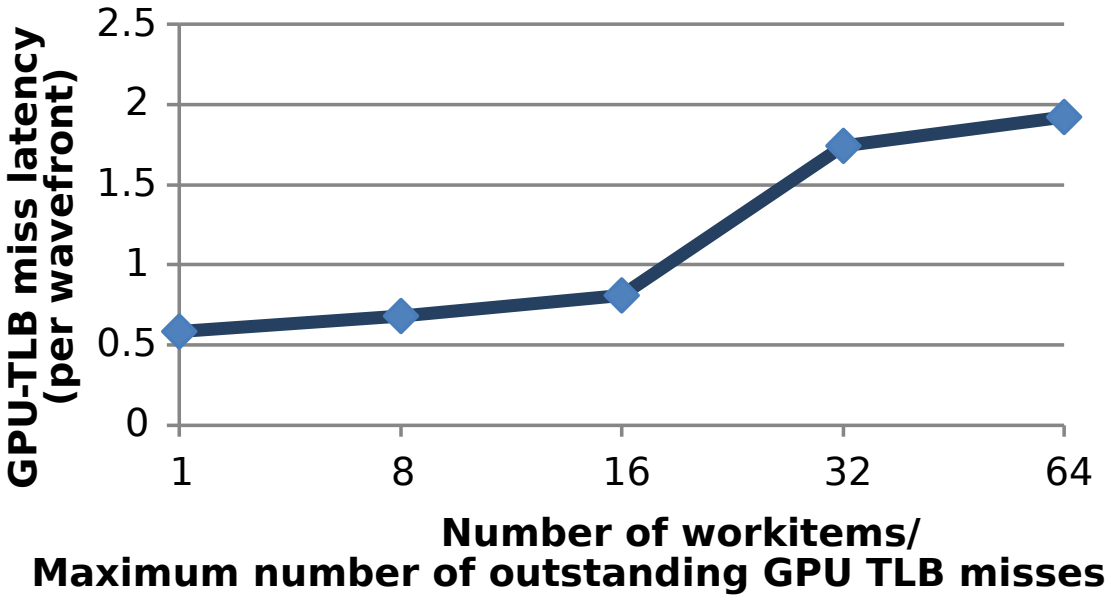


Figure 4.2: Scaling of GPU TLB miss latency

TLB miss scales with an increasing number of workitems. Note that there is a significant jump in the page walk latency beyond 16 workitems (and thus, 16 outstanding page table walks). This suggests that in our test hardware, the IOMMU allows up to 16 concurrent page table walks, beyond which the latency of page walks increases sharply due to queuing.

4.4.2 Measuring GPU's address translation overhead

Next, we run six applications (Table 4.2) to measure and analyze the overheads of the GPU's address translation on real workloads. We run each application with increasing memory footprints to understand the scalability of the GPU's address translation overheads. We use hardware performance counters to measure the number of GPU TLB misses and the number of accesses to the page table by the IOMMU. We perform the measurements of each application using 4KB (default) and 2MB pages. Larger pages reduce the number of TLB misses and enable us to better understand the performance benefits of reducing TLB misses.

Figure 4.3 depicts a summary of our measurements for each application. The x-axis is the approximate memory footprint. The plot's right y-axis represents the GPU runtime, and the left y-axis is the number of GPU TLB misses and the number of accesses to

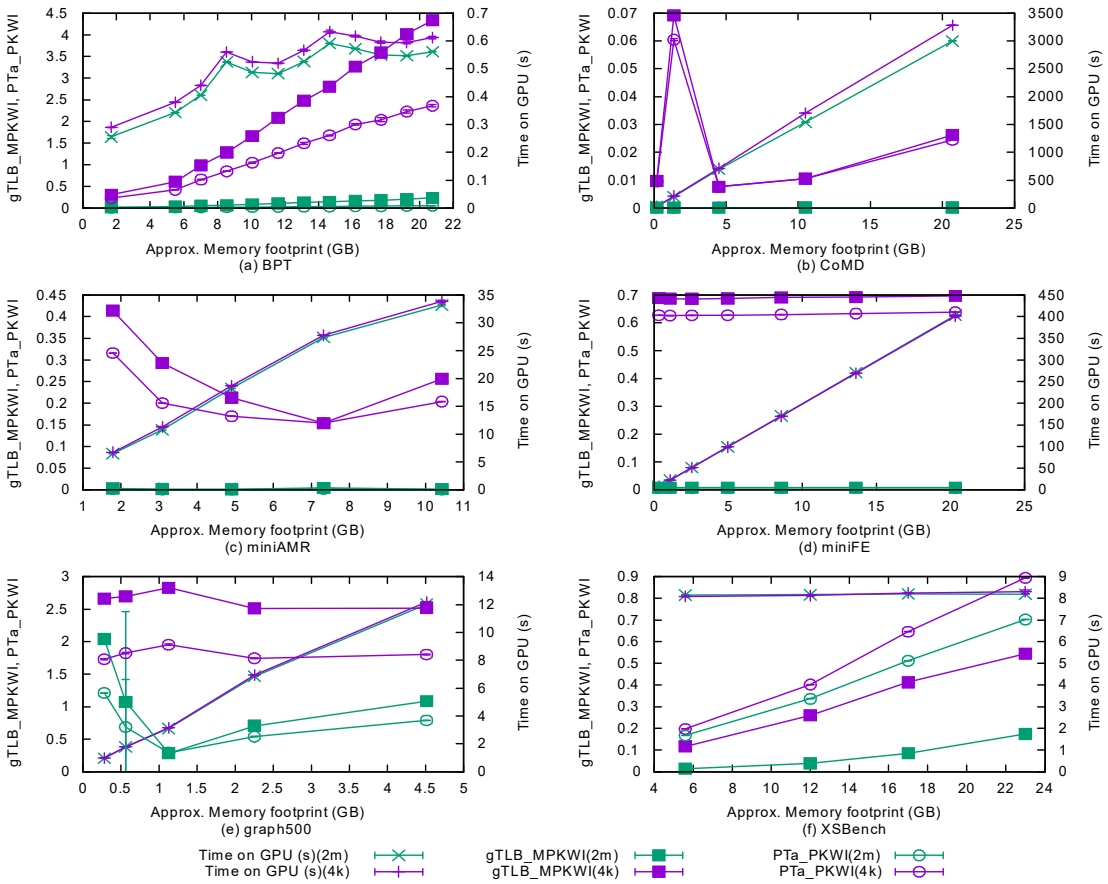


Figure 4.3: GPU TLB miss rates and impact of larger page size

the page table by the IOMMU per kilo wavefront instructions (PKWI) executed on the GPU. A wavefront instruction is a single-instruction-multiple-data (SIMD) instruction executed by workitems (a.k.a GPU threads) in a given wavefront (warp) in a lock-step fashion. In one set of runs, the applications make use of 4KB pages (4k), and in another set of runs the applications make use of larger 2MB pages (2m).

For example, we scale the memory footprint of the workload BPT using different input sets, from 2 to 21 GB. Observe that with 4KB pages, the GPU TLB miss rate goes up from 0.5 to 4.4 misses per KWI [gTLB_MPKWI (4k)]. The number of accesses to the page table goes up from 0.4 to 2 accesses per KWI [PTa_PKWI(4k)]. The GPU TLB misses and the number of accesses to the IOMMU becomes negligible if larger 2MB pages are used. Correspondingly, there is up to an 11% reduction in execution time. We note that PTa_PKWI counts the number of accesses to the in-memory page table and not the number of page table walks. In x86-64, a page table walk can incur up to

four accesses to the page table.

On the opposite end of the spectrum is graph500 (Figure 4.3(e)). There are at most 2.5 GPU TLB misses per KWI. A larger page size (2MB) eliminates almost all TLB misses for graph500, but there is no observable change in execution time. This suggests that the GPU's address translation overhead is not a factor in graph500's performance. We find that graph500 loads data from the memory to the GPU's scratchpad or local data store (LDS) in contiguous chunks, and thus amortizes TLB misses well. Similar behavior is observed for miniFE which also makes use of LDS. In contrast, BPT accesses data more randomly and has less opportunity to amortize this cost. Other applications (Figure 4.3(b)–(e)) show varying degrees of sensitivity to the GPU's address translation overheads.

In summary, BPT, CoMD, and miniAMR are sensitive to TLB miss rates. BPT, CoMD, and miniAMR, respectively, achieve up to 11.9%, 9.7% to 4.6% improvement in runtimes when TLB misses are significantly reduced. On the other hand, graph500, miniFE, and XSBench are insensitive to translation overheads.

Furthermore, preliminary experiments with a recently released second generation APU suggests that while the overall runtimes of applications have significantly improved, the contribution of overheads due to address translation has doubled. This suggests that with future improvement in the rest of the coherent memory hierarchy the address translation is likely to become the bottleneck, unless paid attention to.

4.4.3 Effect of locality on GPU's address translation

The effect of memory-access locality on shared-memory heterogeneous applications is subtle. With sufficient application concurrency and locality, a small number of TLB misses do not affect performance. On a TLB miss, the GPU can switch to another wavefront (warp) to hide the latency of the resulting page walk. On the other hand, if every wavefront incurs a TLB miss (poor locality), or worse, if many workitems (GPU thread) within a wavefront incur a TLB miss (called "data divergence" in GPU terminology), then programs can be highly sensitive to address-translation overhead.

To analyze how poor locality in memory accesses may affect the GPU's address

translation, we used alternative versions of XSBench and BPT. Both XSBench and BPT perform concurrent searches on the GPU over a large data structure (e.g., nuclear energy grid and B+ tree, respectively). In the original versions, the search keys are sorted (on the CPU) before performing the searches on the GPU. This significantly increases the locality in the resulting execution because adjacent workitems (GPU threads) in the same wavefront, and wavefronts scheduled nearby in time access the same memory pages. The alternative versions used in this experiment (called XSBench-unsorted and BPT-unsorted) perform the same work but without the pre-sorting step. Thus, XSBench-unsorted and BPT-unsorted demonstrate considerably less locality in their memory accesses.

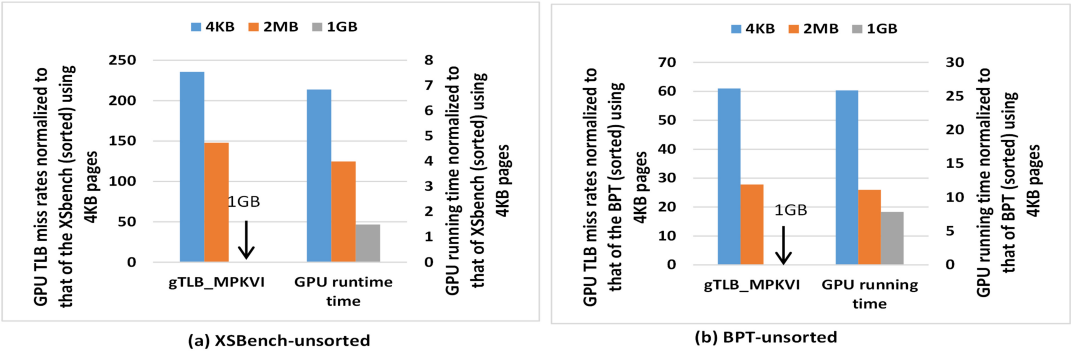


Figure 4.4: GPU TLB miss rates and running time of unsorted version of BPT and XSBench

Figure 4.4 shows the runtime and the TLB miss rates of XSBench-unsorted and BPT-unsorted normalized to their corresponding original (sorted) versions. In Figure 4.4(a), the right cluster of bars (using the right y-axis) shows the GPU running time of XSBench-unsorted using different page sizes (4KB, 2MB, 1GB) normalized to the runtime of the original XSBench (sorted) version using only 4KB pages. The left cluster of bars (using the left y-axis) shows the corresponding number for the GPU TLB misses per KWI. Figure 4.4(b) shows the same data for BPT. Both workloads use the largest dataset for this experiment.

We observe that there is a significant slowdown ($6 - 25\%$) due to poor locality. This slowdown is due to poor locality in address translation, caching, and DRAM. To isolate the impact on the address translation we use large and huge pages to alleviate

TLB misses. For example, observe that with 1GB pages, TLB misses are nearly non-existent (invisible in the graphs) even for the unsorted versions, and correspondingly the slowdown reduces from 6% to 1.5%, and from 26% to 7% for XSBench-unsorted and BPT-unsorted, respectively. This suggests the residual slowdowns (1.5% and 7%) are due to the effect of poor locality in the rest of the memory hierarchy (e.g., caches, memory bandwidth). This strongly indicates that poor locality affects the GPU's address translation far more than the rest of the memory hierarchy. Research into divergence tolerant address translation mechanisms for throughput-oriented accelerators is important.

4.4.4 Effect of address translation prefetching

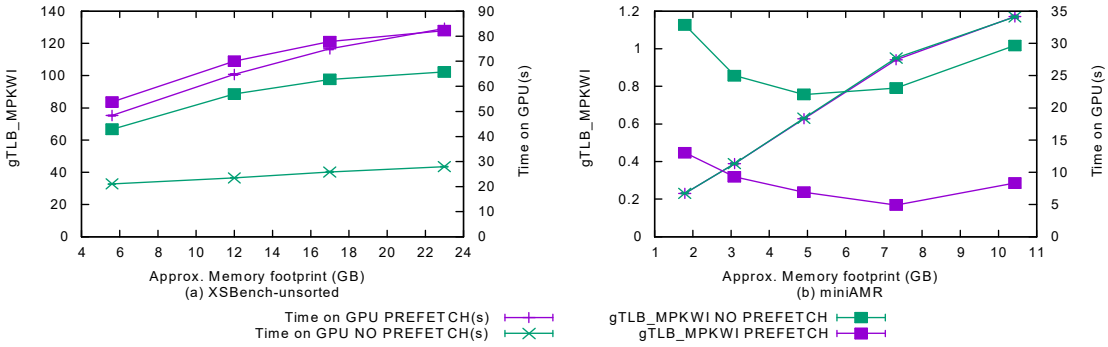


Figure 4.5: Effect of address translation prefetching on GPU's address translation mechanism

As described in Section 4.2.1, the GPU by default prefetches translations for up to eight contiguous (4KB) pages in each ATS request sent to the IOMMU. Figure 4.5 depicts the runtime and the number of GPU TLB misses per KWI with and without translation prefetching for two of the workloads: XSBench-unsorted and miniAMR. In Figure 4.5(a), we find that XSBench-unsorted incurs up to 24% more GPU TLB misses, and consequently, nearly 3% performance degradation due to translation prefetching. The contiguous prefetches are useless for XSBench-unsorted's nearly random accesses to large amounts of data. Furthermore, the useless prefetches evict useful translations from the GPU TLB and ultimately hurt performance by increasing the number of page table walks. Conversely, we find that translation prefetching reduces the number of

TLB misses for the majority of workloads studied. For example, Figure 4.5(b), presents measurements for miniAMR. We observe that prefetching translations aids miniAMR by reducing the number of TLB misses by half. These measurements suggest that the effectiveness of translation prefetching is highly application-dependent and providing application-aware or programmable prefetching would be prudent.

Observations and opportunities:

1. Latency of servicing a TLB miss is significantly higher on a GPU than on a CPU (25×).
2. Increasing the number of concurrent page table walks supported by the hardware is key to supporting diverse heterogeneous applications.
3. Half of the programs we studied suffer performance degradation from GPU address translation overheads.
4. Larger pages are effective in reducing TLB misses. Heterogeneous software and hardware should enhance support for larger page sizes.
5. Divergence in memory accesses impacts address translation overhead more than cache and DRAM latency. Research into divergence-tolerant address translation mechanisms for throughput-oriented accelerators is important.
6. Prefetching address translations can degrade performance for programs with poor locality. Application dependent translation prefetching is desirable.

4.5 Extending page faults to GPUs

A key feature of virtual memory is the ability to establish mappings between virtual and physical addresses on demand. This defers committing physical memory until (and if) needed. Page faults allow the operating system to consolidate physical memory across multiple concurrently running processes and enable memory over-commitment. This

is particularly useful, for example, when an application maps a large input dataset to memory, but only ends up using a small portion of it. Further, page faulting is key in enforcing page permission changes and many advanced memory management techniques like garbage collection and page frame reclamation.

While the ability to perform page faults has been an integral part of a CPU's virtual memory for decades, accelerators like GPUs traditionally lacked such capability. AMD's A10 APUs released in 2014 are one of the first commercial heterogeneous processors to support page faulting on the shared memory from accelerators. However, today's heterogeneous applications are not written to utilize this new capability and instead follow an OpenCL-like programming model. In time this will change, but for now, we modified applications to utilize this functionality to study its impact. We focus on softpage faults, which do not incur accesses to secondary storage.

4.5.1 Analyzing GPU page fault latency and throughput

We analyze the latency and throughput of GPU page faults using a micro-benchmark. We instrumented the IOMMU driver to perform the measurements. The micro-benchmark generates a constant number (512,000) of soft-page faults from the GPU. The soft page faults do not access storage and are generated by the first access to a page in memory. The micro-benchmark is designed to generate faults in controlled bursts by varying the number of workitems. When the microbenchmark is executed with 'n' workitems, it generates 'n' concurrent page fault requests from the GPU. Thus increasing the value of 'n' generates larger bursts of concurrent page faults. Figure 4.6 shows the measured latency and throughput of servicing these GPU page faults. The total height of each bar represents the average latency to service a page fault. The left y-axis represents the page fault latency in microseconds. The right y-axis represents throughput of servicing GPU page faults. For example, with 64 workitems, the average latency to service a page fault is around 100 microseconds and 260 page faults are serviced per millisecond (throughput).

We make two key observations: (1) the average latency to service a page fault increases from 5 microseconds to 140 microseconds with increasing number of concurrent

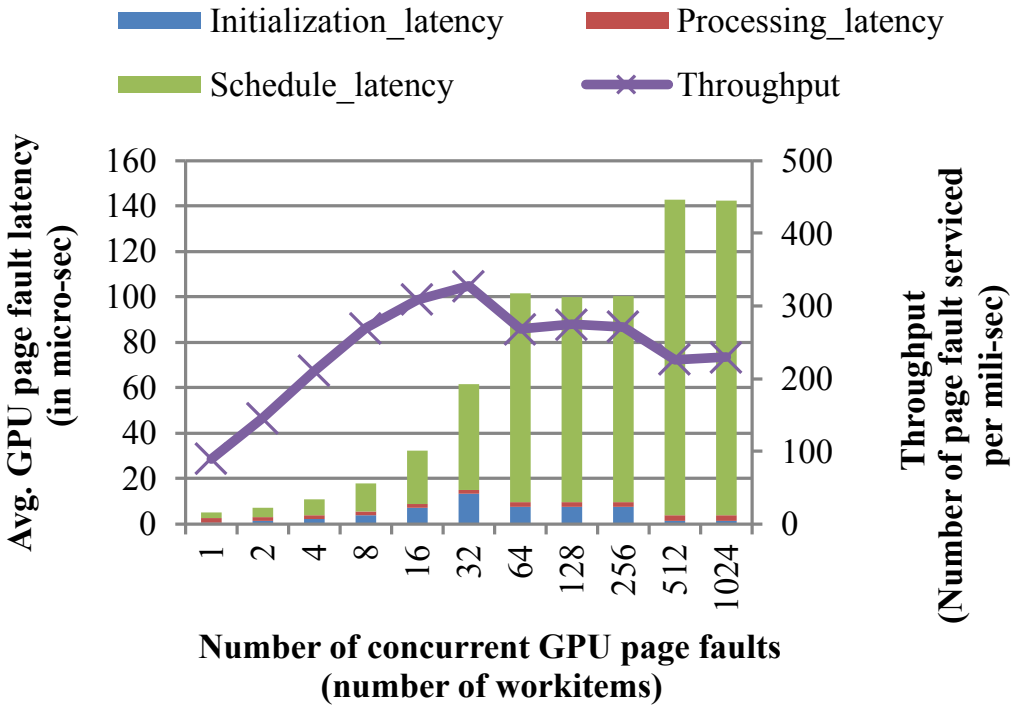


Figure 4.6: Scaling of GPU page faults.

page faults from the GPU; and (2) the throughput of servicing GPU page faults does not scale beyond 32 concurrent page faults. To put these numbers in perspective, we executed the experiment on the CPU (single-threaded) and found the typical page fault latency on the CPU is around 1.7 microseconds. GPU pagefaults are $3 - 80\times$ slower. Larger concurrency in servicing page faults from the GPU can help amortize this high latency.

We breakdown the (software) latency to service a GPU page fault in Figure 4.6. We divide the time to handle a GPU page fault into three major parts: (1) "initialization", the latency for the OS driver to read the fault requests from the PPR queue and pre-process it; (2) "processing", the latency to find a physical page and update the page table; (3) "schedule", the time between initialization and processing of a page fault request. We observe that only a small fraction of the time is spent in actually processing the work to service a page fault. The OS's scheduling delay introduced by the asynchronous handling of GPU page faults is the primary contributor to the latency. This suggests

that page faults from the GPU can be handled more efficiently by modifying the OS driver to handle the faults synchronously whenever possible.

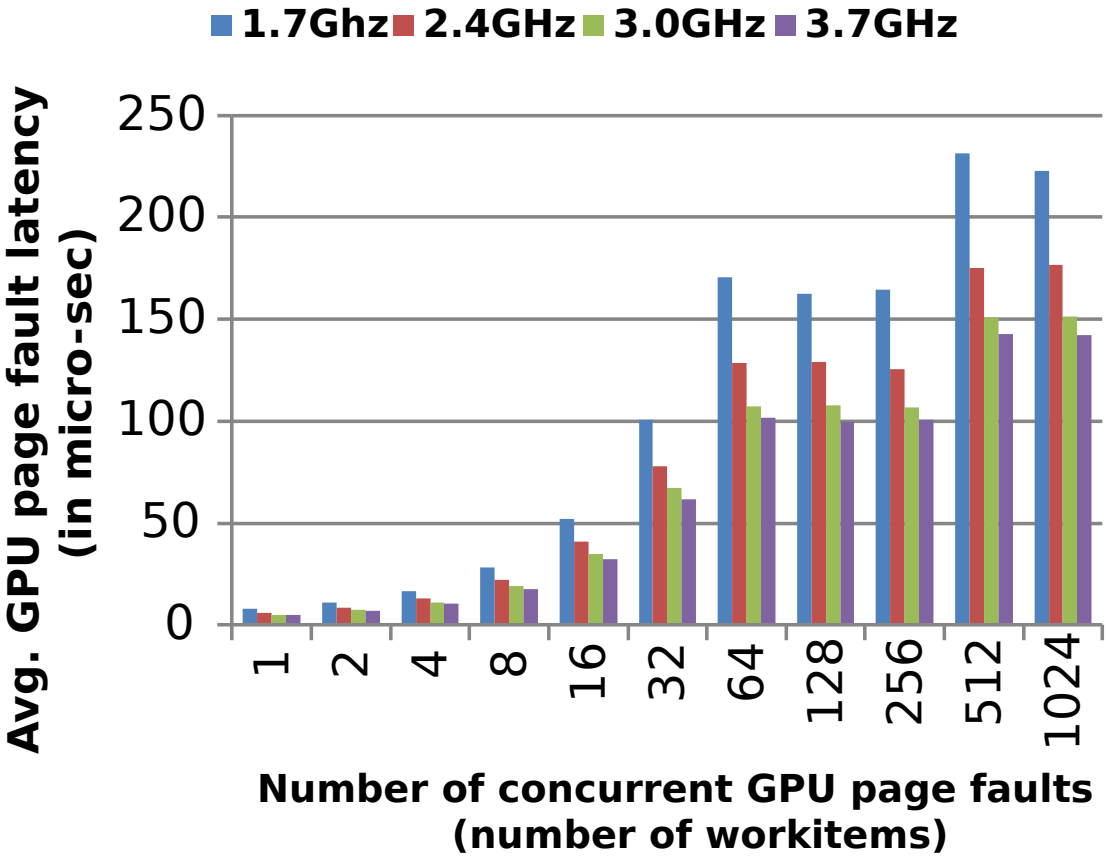


Figure 4.7: Scaling of GPU page faults with CPU frequency.

Figure 4.7 shows the relationship between the CPU's core frequency and the latency to service a GPU page fault. The height of each bar in the clusters represents average GPU-page fault latency with the CPU running at the given frequency. We observe that the latency to service a GPU page fault nearly doubles when the CPU core frequency is reduced from 3.7GHz to 1.7GHz. In general, the graph shows that the GPU page fault latency inversely scales linearly with the CPU's core frequency. This suggests that the CPU's core frequency needs to scale up for faster servicing of GPU page faults and the CPU power setting may affect GPU page fault behavior.

4.5.2 Physical memory consolidation through GPU page faults

We measured the reduction in the physical memory footprint through the use of on-demand page faults from the GPU and measure its performance overhead.

We modified four applications (BPT, XSBench, CoMD and Graph500) to utilize demand faulting of memory from the GPU. These modifications include using memory-mapped files, changes to the data structures, and memory allocation. We did not modify miniFE and miniAMR as it was not practical to dynamically fault in data from the GPU to achieve memory consolidation without major alterations to their code bases.

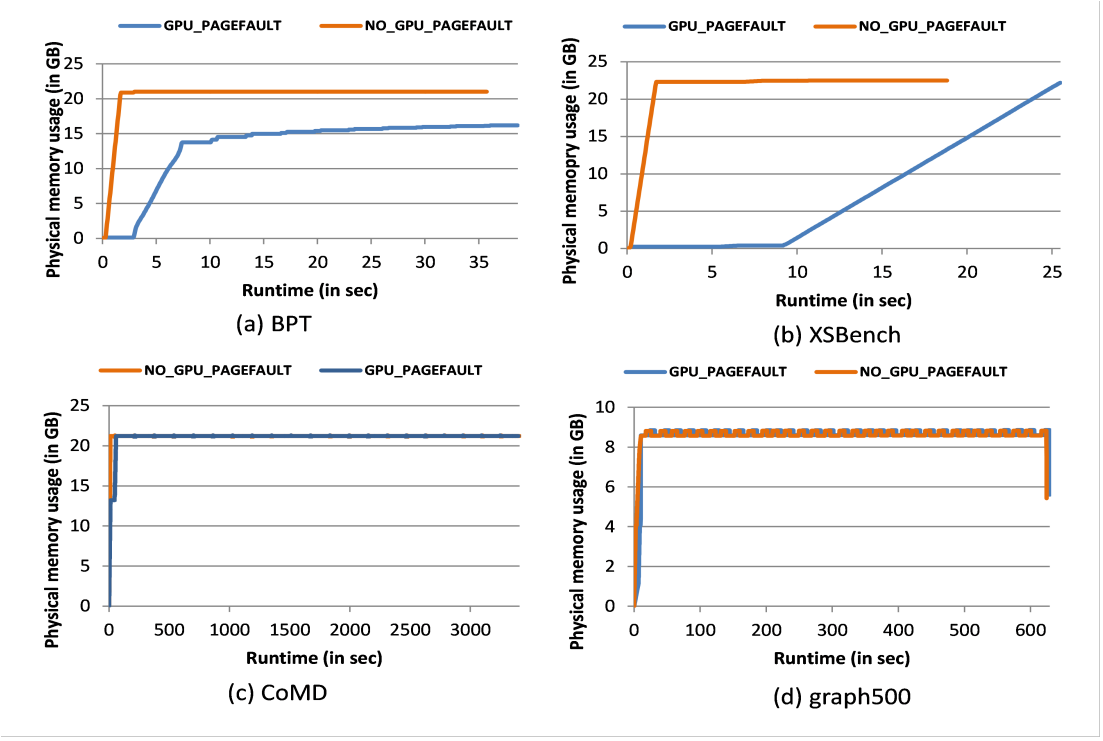


Figure 4.8: Physical memory usage with and without page faults from GPU.

Figure 4.8 depicts the reduction in physical memory footprint through the use of page faults from the GPU. The x-axis of each graph represents a given workload's running time on the GPU while the y-axis represents the physical memory allocated to the workload at a given time. Each graph has two lines representing two versions of each workload. "GPU_PAGEFAULT", is the modified version of a workload that dynamically faults memory from the GPU. "NO_GPU_PAGEFAULT", represents the original version. The difference between these two lines signifies opportunity to save

physical memory. In Figure 4.8(a), we observe that the physical memory footprint of BPT reduces significantly with GPU page faults. BPT performs concurrent searches on the GPU over a pre-generated B+tree (size 20GB). It is not necessary to access the entire tree for finding the keys. Thus the use of page faults from the GPU avoids unnecessarily allocating physical memory for the entire tree. In Figure 4.8(b),we observe that XSBench allows savings in physical memory footprint during initialization, but memory footprint grows quickly as the entire allocated memory is gradually accessed over time. Deeper inspection into XSBench reveals that the biggest contributor to its memory footprint is the data structure for the nuclear energy grid. This energy grid is accessed to perform concurrent cross-sectional lookups on the GPU. We find that if a large number of lookups are performed (a parameter, default 15M) at random cross-sections, then eventually the entire energy grid is accessed. Thus the physical memory usage with and without GPU page faults converges. However, XSBench's physical memory footprint reduces substantially when a smaller number of lookups (e.g., 500K) are performed. We find that there is very little scope for consolidating physical memory for workloads like graph500 and CoMD. In graph500, the entire graph data structure is traversed and thus all of allocated memory is accessed. Similarly, for CoMD the entire allocated memory is needed by the GPU. Hence, the potential for reducing memory footprint varies across workloads and can be dependent upon the input.

Figure 4.9 shows the runtimes normalized to the runtimes with no page faults from the GPU. Each bar also shows a breakdown of runtime spent on the CPU and GPU. We observe that XSBench and BPT can incur significant performance degradations due to page faults from the GPU. We note that a larger fraction of the time is spent on the GPU if page faulting is used. This is expected as GPU page faults hinder concurrency in the GPU. Other workloads show little or no performance impact. In summary, we find significant scope for research into heterogeneous software and hardware to reduce the page fault latency and enabling more concurrency in servicing them. These research efforts, however, should focus on enabling new capabilities in the runtime rather than improving the performance of legacy applications.

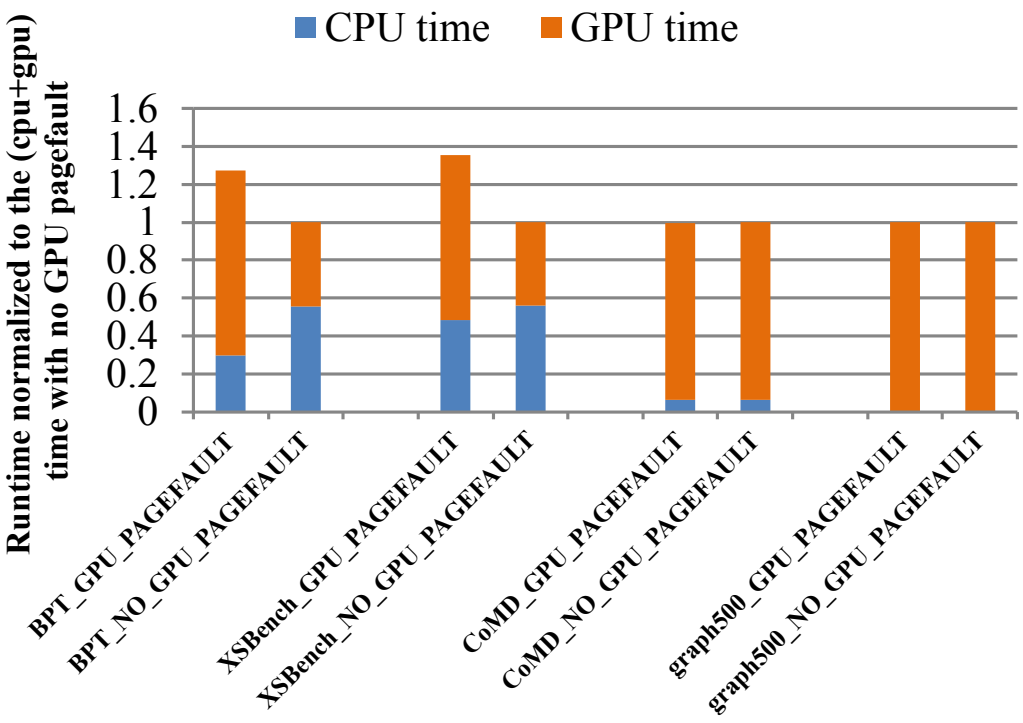


Figure 4.9: Performance overhead of GPU page faults.

Observations and opportunities:

1. The latency to service a page fault from the GPU can be significantly higher than from the CPU.
2. Enhancements into system software to handle page faults synchronously can reduce this latency.
3. Software-hardware co-design is needed service a large number of concurrent faults from the GPU/accelerators.
4. It is imperative to scale CPU performance and resources to scale the GPU page fault servicing.
5. Future heterogeneous applications can reduce their physical memory footprints through the use of on-demand page faults from the GPU, although current applications may need to be re-written.

4.6 Extending TLB shootdowns to GPUs

	Avg. GPU TLB shootdown latency (in nanoseconds)	Breakdown of GPU TLB shootdown latency (in nanoseconds)			
		Initi-liza-tion	IOMMU TLB inval	GPU TLB inval	Fina-liza-tion
Single-entry shootdown	4234	79	1970	2021	162
All-entry shootdown	4409	122	2052	2035	299

Table 4.4: GPU TLB shootdown analysis

We wrote a simple micro-benchmark that generates a large number of GPU TLB shootdowns. It generates both single-entry TLB invalidations and entire TLB flushes. Table 4.4 presents measurements and the breakdown of latency of a GPU TLB shootdown. The first column depicts the average latency of a GPU TLB shootdown. The first row shows the latency of invalidating a single entry, and the second row shows the latency of flushing the entire TLB. The average latency of a TLB shootdown is around 4.2-4.4 microseconds. This latency is comparable to typical times required to perform a TLB shootdown across 4 to 8 CPU cores [158]. We note that nearly an equal amount of time is spent in invalidating the IOMMU's and the GPU's TLB entries. We also note that there is no significant difference between the latency to invalidate a single entry or flushing the entire TLB. In the Linux operating system, TLB shootdowns often take long time to complete for large systems with many nodes. This may potentially be a scaling bottleneck in the future. For example, future systems with multi-level memory [159] that migrate pages between different levels of memory will critically depend on TLB shootdown performance.

4.7 Related Work

The advent of big-data workloads, often with poor access locality (e.g., graph processing algorithms, massive key value stores), have recently led to a surge of research on virtual

memory for big-memory servers [160–169]. A number of research proposals have considered hardware and software mechanisms to improve the effective capacity of TLBs without additional area costs [161, 162, 165, 167–169] with speculation [160, 170], and with approaches that better managed large pages [170–172]. In parallel, the emergence of the unified address space paradigm for APUs (and more broadly, heterogeneous systems) has prompted the first set of studies on GPU address translation and memory management units [23, 173]. For example Pichai, Hsu, and Bhattacharjee show that intelligent hardware page table walkers are crucial to the performance of throughput-oriented accelerators and are deeply tied to the operation of the wavefront scheduler for both round-robin (the default) and advanced dynamic wavefront formation strategies designed to improve cache locality and mitigate control-flow divergence overheads [23]. Similarly, Power, Hill, and Wood show that throughput-oriented, multi-threaded page table walkers are critical for GPU performance, especially in conjunction with intelligently-designed translation caches [173]. Beyond these works on GPUs, researchers have begun considering address translation for fixed-function and programmable accelerators [174]. Unlike past works on GPU address translation, we are the first to characterize shared virtual memory behavior on a real heterogeneous system with realistic workloads. Our work shows the benefits, challenges, and potentially interesting research avenues in this space by collecting results on the first generations of hardware and software that actually implement CPU-GPU shared address spaces. Our work sheds light on the detailed interactions in the address translation microarchitecture beyond the scope of prior works. As such, we believe that our study provides a foundation for guiding the research community on some of the most-pressing problems in the shared virtual memory paradigm.

4.8 Summary: Observations and Opportunities

We summarize the lessons learned analyzing shared virtual memory in one of the first commercially available heterogenous processors. We discuss possible research opportunities in the space for future-generation heterogeneous processors.

Address translation: TLB misses in GPUs are currently an order of magnitude slower than that in CPUs. A GPU program's large memory-level parallelism, along

with concurrent servicing of GPU TLB misses can potentially help to hide this latency. Research on techniques that increases TLB miss handling concurrency are crucial, particularly for throughput-oriented accelerators like GPUs. We observe that divergence in memory accesses impacts address translation more than the rest of the memory hierarchy. Although there has been a significant body of research in managing divergence in the cache hierarchy, there is a dearth of work that studies its impact on address translation. We find that prefetching translations usually aids performance, but under certain circumstances it can degrade performance. Research into software-hardware co-design for application-aware prefetchers for address translation will be useful. Finally, we find that large pages universally help in reducing address translation overheads. Hardware designers should build enhanced support for large pages and programmers should make use them. Furthermore, our preliminary experiments with second generation APU suggests that address translation is likely to become a bigger contributor to the performance overhead as the rest of the memory hierarchy is improved in future generation heterogenous processors.

Page fault: Dynamically allocating physical memory via page faults from the GPU could potentially enable significant memory consolidation for future applications with large footprints. However, we observed that servicing a GPU page fault on current systems can take an order of magnitude longer ($3 - 82\times$) than that for CPU page faults. Addressing this challenge requires changes to both the hardware and software. Enhancements to both the system software and hardware to service a larger number of concurrent page faults could help mitigate the overheads in the page fault process.

TLB Shootdown: Our workloads encountered only a few instances of TLB shootdowns. TLB shootdown latencies for current heterogeneous systems are comparable to those in the CPU and are hence expensive. However, because TLB shootdowns are serialized, they could be a potential performance bottleneck in future systems, particularly for heterogeneous memory systems with frequent physical page migration. This is not an intrinsic limitation of the hardware, but architecting the OS to support concurrent shootdowns will be required as systems scale upward.

4.9 Acknowledgements

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. The format PCIe® is a registered trademark of PCI-SIG Corporation. The format ARM® is a registered trademark of ARM Limited. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. This work was supported in part by the National Science Foundation, under grant number 1337147.

Chapter 5

Conclusion

5.1 Summary

For accelerators to realize their promise of increased performance and efficiency they need to be accessible to programmers. However, their compute environment and programming model is often wildly different from that of CPUs – a situation that restricts the adoption of accelerators for applications that cannot invest significant developer resources to optimize their codebase. This work studies accelerator programmability to extend the set of applications that can benefit from improved performance and efficiency.

First, Chapter 2 studied the problem of accelerating programs written in a high-level language. I noticed that high-level languages are often used to assemble composited programs from highly optimized components. Many of these components can take advantage of accelerators, even if the language that ties them together can not. An example of cognitive models written in Python demonstrated that compiler techniques can be used to bridge this gap. The resulting code is not only amenable for acceleration, but also provides orders of magnitude improvements when run on CPU. I discussed how certain language features – such as the choice of algorithm for generating random numbers, or the choice of high-precision arithmetic – become crucial for efficient mapping of computation to accelerators.

Although some language features can be safely eliminated during the compilation process, outside communication in the form of system services can not. Chapter 3 examined extending GPU programming environment with the same level of access to operating system services as is common on CPUs. I proposed and implemented an efficient system call invocation routine for GPUs in CPU-GPU heterogeneous systems.

I studied the semantics of POSIX interfaces and assessed their suitability for highly-parallel GPU environment. I evaluated the performance of a selection of POSIX system calls, and demonstrated the significant benefits of GPU system services in designing heterogeneous applications even on scales that wouldn't traditionally be considered for GPU acceleration (*e.g.* grep).

Finally, Chapter 4 studied the impact of extending virtual memory – an important abstraction that improves CPU programmability [175] – to GPUs. It noticed the disproportionate impact of address translation on performance, compared to the rest of the memory hierarchy. This chapter also includes a study on the impact of GPU page faults on the operating system and heterogeneous CPU-GPU system as a whole. Designing efficient address translation mechanisms for specialized accelerator hardware, and GPUs in particular has now become a major area of study [176–180].

All studies in this work were done on off-the-shelf hardware and none of the proposed solutions require hardware modifications. This thesis demonstrates that extending known programmability features from CPUs to accelerators provides significant benefits; It enables new applications that previously wouldn't be considered for acceleration, it improves the performance of existing applications by better utilization of system resources, and it extends the benefits of accelerators to high-level languages that are otherwise difficult to map to accelerators. This highlights significant opportunities for improvement of both performance and programmability of accelerators that exist across application runtimes and software stack.

5.2 Towards Ideal Heterogeneous Systems

An ideal heterogeneous system would allow programmers to express their algorithms in any language that suits the domain, and execute each part of the code on hardware that gives the best performance or efficiency. This work takes crucial steps towards enabling efficient use of accelerator heterogeneity on the application level. This exposes further opportunities in adapting operating systems and other parts of the software hierarchy to bridge the gaps between heterogeneous software and hardware.

Improving the performance of code generated from program sources is a never-ending research endeavor. Chapter 2 shows that manual embedding of domain-specific information can improve this process. Ideally, a compiler would be able to deduce most or all of the information automatically and future research should aim to automate much of the process. A compiler should be able to make do more radical data structure conversions, and make decisions on which parts of the code are best suitable for acceleration. Dynamic languages with interpreted or just-in-time compiled execution can make these decisions based at execution time rather than build time, which makes them better positioned to exploit future accelerator rich systems.

Operating systems assume that every program has at least one CPU thread, and this work does not challenge this assumption. One can, however, imagine a system with tasks that only execute on accelerators. Such a system will challenge the traditional representation of tasks using process and thread abstractions. Throughput oriented accelerators, like GPUs, use many short-lived threads. Representing all of them in the operating system would add significant overhead, yet using coarser representation may impact the semantics of available system services. On the other hand, systems may include accelerators that work in event-driven fashion (*e.g.* machine learning inference tasks) and don't use any threads at all, execution on such hardware will also need to be represented within operating systems if such accelerators should work as independent units rather than just CPU co-processors.

Similarly, this work does not challenge the fundamental design decision that all privileged code in an operating system runs on a CPU. Reusing the same operating system code-paths allowed the implementation in Chapter 3 to provide the same semantics for both CPU and GPU programs. However, this centralized design will be stressed as more accelerators in the system allow programs to request system services. A more decentralized, distributed approach would alleviate some of the communication overheads and reduce the pressure on the central CPU. To achieve this vision, accelerators will need to gain the ability to execute privileged code and efficiently share system-wide data structures. Alternatively, systems can opt for even more specialization introducing dedicated, OS-only, cores to handle the increased demand for system services. Which one of

these directions becomes prevalent depends on a lot of factors including demands of heterogeneous applications, and availability of fast communication hardware to implement system interconnects and coherence protocols.

This work studies existing CPU programmability abstractions and their suitability for accelerators. The demonstrated balance of convenience, familiarity, and performance necessary makes accelerators accessible with little effort and enables a new class of heterogeneous applications. It will be the demands of these new applications that will determine how heterogeneous systems develop in the future.

Bibliography

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual international symposium on computer architecture (ISCA)*, pp. 365–376, IEEE, 2011.

[2] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient computation of sum-products on gpus through software-managed cache," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 309–318, 2008.

[3] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010.

[4] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 88–98, IEEE, 2012.

[5] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong, "A case for hardware-based demand paging," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1103–1116, IEEE, 2020.

[6] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of

a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[8] P. Pandey, P. Basu, K. Chakraborty, and S. Roy, "Greentpu: Improving timing error resilience of a near-threshold tensor processing unit," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2019.

[9] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 240–249, 2015.

[10] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 57–60, 2013.

[11] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, IEEE, 2018.

[12] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.

[13] D. D. Thaker, T. S. Metodi, A. W. Cross, I. L. Chuang, and F. T. Chong, "Quantum memory hierarchies: Efficient designs to match available parallelism in quantum computing," in *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 378–390, IEEE, 2006.

[14] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[15] NVidia, "CUDA Toolkit Documentation," 2016. `"http://docs.nvidia.com/cuda"`.

[16] K. Group, "OpenCL," 2009. `https://www.khronos.org/opencl/`.

[17] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: integrating file systems with GPUs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[18] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[19] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs," in *Usenix Technical Conference (ATC)*, 2017.

[20] Microsoft, "C++ AMP (C++ Accelerated Massive Parallelism) [Online]," 2016. `https://msdn.microsoft.com/en-us/library/hh265137.aspx`.

[21] OpenMP ARB, "OpenMP," 1997. `https://www.openmp.org/`.

[22] P. Express, "Address translation services."

[23] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 743–758, 2014.

[24] K. Group, "SYCL," 1992. `https://www.khronos.org/sycl/`.

[25] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[29] M. Ravishankar and V. Grover, "Automatic acceleration of numpy applications on gpus and multicore cpus," *arXiv preprint arXiv:1901.03771*, 2019.

[30] S. Cass, "The 2015 top ten programming languages," *IEEE Spectrum, July*, vol. 20, 2015.

[31] J. Owens, J. Stuart, and M. Cox, "GPU-to-CPU Callbacks," in *Workshop on Unconventional High Performance Computing*, 2010.

[32] "Jython." `https://www.jython.org/`. Accessed on 05/24/2020.

[33] "Ironpython." `https://ironpython.net/`. Accessed on 05/24/2020.

[34] T. P. Team, "Pypy." `https://www.pypy.org/`, Dec 2019. Accessed on 05/24/2020.

[35] "Pyston." `https://github.com/pyston/pyston/releases/tag/v2.0`.

[36] "A high performance python compiler." `http://numba.pydata.org/`. Accessed on 05/24/2020.

[37] NVidia, "NVIDIA GPUDirect [Online]," 2013. `https://developer.nvidia.com/gpudirect`.

[38] J. L. McClelland, D. E. Rumelhart, P. R. Group, *et al.*, "Parallel distributed processing," *Explorations in the Microstructure of Cognition*, vol. 2, pp. 216–271, 1986.

[39] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly, "Why there are complementary learning systems in the hippocampus and neocortex: insights from the

successes and failures of connectionist models of learning and memory.," *Psychological review*, vol. 102, no. 3, p. 419, 1995.

[40] P. R. Montague, P. Dayan, and T. J. Sejnowski, "A framework for mesencephalic dopamine systems based on predictive hebbian learning," *Journal of neuroscience*, vol. 16, no. 5, pp. 1936–1947, 1996.

[41] E. K. Miller and J. D. Cohen, "An integrative theory of prefrontal cortex function," *Annual review of neuroscience*, vol. 24, no. 1, pp. 167–202, 2001.

[42] A. Shenhav, M. M. Botvinick, and J. D. Cohen, "The expected value of control: an integrative theory of anterior cingulate cortex function," *Neuron*, vol. 79, no. 2, pp. 217–240, 2013.

[43] J. D. Cohen and T. R. Insel, "Cognitive neuroscience and schizophrenia: translational research in need of a translator," *Biological psychiatry*, vol. 64, no. 1, pp. 2–3, 2008.

[44] P. R. Montague, R. J. Dolan, K. J. Friston, and P. Dayan, "Computational psychiatry," *Trends in cognitive sciences*, vol. 16, no. 1, pp. 72–80, 2012.

[45] T. V. Maia, Q. J. Huys, and M. J. Frank, "Theory-based computational psychiatry," *Biological psychiatry*, vol. 82, no. 6, pp. 382–384, 2017.

[46] A. G. Ivakhnenko and V. G. Lapa, "Cybernetic predicting devices," tech. rep., PURDUE UNIV LAFAYETTE IND SCHOOL OF ELECTRICAL ENGINEERING, 1966.

[47] W. Gerstner and W. M. Kistler, "Mathematical formulations of hebbian learning," *Biological cybernetics*, vol. 87, no. 5-6, pp. 404–415, 2002.

[48] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, "A theoretical framework for back-propagation," in *Proceedings of the 1988 connectionist models summer school*, vol. 1, pp. 21–28, CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

[49] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representation by error propagation, parallel distributed processing," *MIT Press, Cambridge*, 1986.

[50] A. Bouzerdoum and R. B. Pinter, "Biophysical basis, stability, and directional response characteristics of multiplicative lateral inhibitory neural networks," in *International 1989 Joint Conference on Neural Networks*, pp. 617 vol.2–, 1989.

[51] S.-i. Amari, "Dynamics of pattern formation in lateral-inhibition type neural fields," *Biological cybernetics*, vol. 27, no. 2, pp. 77–87, 1977.

[52] D. Kumaran, D. Hassabis, and J. L. McClelland, "What learning systems do intelligent agents need? complementary learning systems theory updated," *Trends in cognitive sciences*, vol. 20, no. 7, pp. 512–534, 2016.

[53] Y. Bengio, S. Bengio, and J. Cloutier, "Learning to Learn a Synaptic Rule," in *International Joint Conference on Neural Networks*, 2013.

[54] C. Lemke, M. Budka, and B. Gabrys, "Metalearning: A Survey of Trends and Technologies," in *Artificial Intelligence Review*, 2013.

[55] P. Brazdil, C. G. Carrier, C. Soares, and R. Vilalta, "Metalearning," in *Springer - Cognitive Technologies*, 2009.

[56] Y. Sagiv, S. Musslick, Y. Niv, and J. D. Cohen, "Efficiency of learning vs. processing: Towards a normative theory of multitasking," *arXiv preprint arXiv:2007.03124*, 2020.

[57] S. Ravi, S. Musslick, M. Hamin, T. L. Willke, and J. D. Cohen, "Navigating the trade-off between multi-task learning and learning to multitask in deep neural networks," *arXiv preprint arXiv:2007.10527*, 2020.

[58] T. Willke, S. Yoo, M. Capotă, S. Musslick, B. Hayden, and J. Cohen, "A comparison of non-human primate and deep reinforcement learning agent performance in a virtual pursuit-avoidance task," 2019.

[59] Nvidia, "V100 gpu architecture." `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2018.

[60] X. Zhang, R. Bashizade, C. LaBoda, C. Dwyer, and A. R. Lebeck, "Architecting a stochastic computing unit with molecular optical devices," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 301–314, 2018.

[61] S. Boixo, T. F. Rønnow, S. V. Isakov, Z. Wang, D. Wecker, D. A. Lidar, J. M. Martinis, and M. Troyer, "Evidence for quantum annealing with more than one hundred qubits," *Nature physics*, vol. 10, no. 3, pp. 218–224, 2014.

[62] M. Hill and V. J. Reddi, "Accelerator level parallelism," in *arxiv*, 2020.

[63] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365, IEEE, 2013.

[64] J. Cohen, A. Bhattacharjee, and T. Wilke, "A standardized model description format for accelerating convergence in neuroscience, cognitive science, machine learning and beyond," *NSF Awards*, 2020.

[65] J. Bai, F. Lu, K. Zhang, *et al.*, "Onnx: Open neural network exchange," *GitHub repository*, 2019.

[66] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[67] "Psyneulink." `https://princetonuniversity.github.io/PsyNeuLink/`. Accessed on 05/24/2020.

[68] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 3, pp. 243–250, 1977.

[69] R. A. Van Engelen, "Efficient symbolic analysis for optimizing compilers," in *International Conference on Compiler Construction*, pp. 118–132, Springer, 2001.

[70] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[71] M. L. Hines and N. T. Carnevale, "The neuron simulation environment," *Neural Computation*, vol. 9, no. 6, pp. 1179–1209, 1997.

[72] S. A. Neymotin, D. S. Daniels, B. Caldwell, R. A. McDougal, N. T. Carnevale, M. Jas, C. I. Moore, M. L. Hines, M. Hämäläinen, and S. R. Jones, "Human neocortical neurosolver (hnn), a new software tool for interpreting the cellular and network origin of human meg/eeg data," *Elife*, vol. 9, p. e51214, 2020.

[73] J. R. Anderson, M. Matessa, and C. Lebiere, "Act-r: A theory of higher level cognition and its relation to visual attention," *Human–Computer Interaction*, vol. 12, no. 4, pp. 439–462, 1997.

[74] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[75] B. Aisa, B. Mingus, and R. O'Reilly, "The emergent neural modeling system," *Neural Networks*, vol. 21, no. 8, pp. 1146 – 1152, 2008. Neuroinformatics.

[76] J. M. Bower, H. Cornelis, and D. Beeman, *GENESIS, The GEneral NEural SImulation System*, pp. 1–8. New York, NY: Springer New York, 2013.

[77] J. R. Stroop, "Studies of interference in serial verbal reactions.," *Journal of experimental psychology*, vol. 18, no. 6, p. 643, 1935.

[78] M. M. Botvinick, T. S. Braver, D. M. Barch, C. S. Carter, and J. D. Cohen, "Conflict monitoring and cognitive control.," *Psychological review*, vol. 108, no. 3, p. 624, 2001.

[79] L. A. Necker, "LXI. Observations on some remarkable optical phænomena seen in Switzerland; and on an optical phænomenon which occurs on viewing a figure of a crystal or geometrical solid," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 1, no. 5, pp. 329–337, 1832.

[80] R. Bogacz, M. Usher, J. Zhang, and J. L. McClelland, "Extending a biologically inspired model of choice: multi-alternatives, nonlinearity and value-based multidimensional choice," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 362, no. 1485, pp. 1655–1670, 2007.

[81] "A lightweight llvm python binding for writing jit compilers." `https://github.com/numba/llvmlite`.

[82] S. D. Brown and A. Heathcote, "The simplest complete model of choice response time: Linear ballistic accumulation," *Cognitive Psychology*, vol. 57, no. 3, pp. 153 – 178, 2008.

[83] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[84] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008.

[85] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*, pp. 172–181, IEEE, 2008.

[86] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "Genesys: Enabling continuous learning through neural network evolution in hardware," in *2018 51st Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 855–866, IEEE, 2018.

[87] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, IEEE, 2020.

[88] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 940–953, IEEE, 2020.

[89] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 161–171, IEEE, 2016.

[90] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 37–48, IEEE, 2017.

[91] T. H. Click, A. Liu, and G. A. Kaminski, "Quality of random number generators significantly affects results of monte carlo simulations for organic and biological systems," *Journal of computational chemistry*, vol. 32, no. 3, pp. 513–524, 2011.

[92] "Numba supported python features." `https://numba.pydata.org/numba-doc/dev/reference/pysupported.html`. Accessed on 08/14/2020.

[93] nVidia, "Cuda graphs." `https://devblogs.nvidia.com/cuda-graphs/`. Accessed on 05/24/2020.

[94] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[95] I. Keivanloo, C. K. Roy, and J. Rilling, "Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning," in *2012 6th International Workshop on Software Clones (IWSC)*, 2012.

[96] I. Keivanloo, C. K. Roy, and J. Rilling, "SeByte: A semantic clone detection tool for intermediate languages," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012.

[97] P. Schugerl, J. Rilling, and P. Charland, "Reasoning about global clones: Scalable semantic clone detection," in *2011 IEEE 35th Annual Computer Software and Applications Conference*, 2011.

[98] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," in *ACM Computing Surveys*, 2015.

[99] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Computer Graphics Forum, Vol. 26, Num. 1, pp 80-113*, 2007.

[100] D. Tarditi, S. Puri, and J. Oglesby, "Acelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[101] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee, "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[102] J. Obert, J. V. Waveran, and G. Sellers, "Virtual Texturing in Software and Hardware," in *SIGGRAPH Courses*, 2012.

[103] G. Cox and A. Bhattacharjee, "Efficient Address Translation with Multiple Page Sizes," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[104] L. Olson, J. Power, M. Hill, and D. Wood, "Border Control: Sandboxing Accelerators," in *International Symposium on Microarchitecture (MICRO)*, 2007.

[105] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[106] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[107] N. Agarwal, D. Nellans, E. Ebrahimi, T. Wenisch, J. Danskin, and S. Keckler, "Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[108] H. Foundation, "HSA Programmer's Reference Manual [Online]," 2015. `http://www.hsafoundation.com/?ddownload=4945`.

[109] G. Kyriazis, "Heterogeneous system architecture: A technical review," *AMD Fusion Developer Summit*, p. 21, 2012.

[110] J. Power, A. Basu, J. Gu, S. Puthoor, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[111] S. Sahar, S. Bergman, and M. Silberstein, "Active Pointers: A Case for Software Address Translation on GPUs," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[112] C. J. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," in *International Symposium on Microarchitecture (MICRO)*, 2002.

[113] J. Owens, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, and W. Dally, "Media Processing Applications on the Imagine Stream Processor," in *International Conference on Computer Design (ICCD)*, 2002.

[114] J. Owens, B. Khailany, B. Towles, and W. Dally, "Comparing Reyes and OpenGL on a Stream Architecture," in *SIGGRAPH, EUROGRAPHICS Conference on Graphics Hardware*, 2002.

[115] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens, "A Bandwidth-Efficient Architecture for Media Processing," in *International Symposium on Microarchitecture (MICRO)*, 1998.

[116] B. Serebrin, J. Owens, B. Khailany, P. Mattson, U. Kapasi, C. Chen, J. Namkoong, S. Crago, S. Rixner, and W. Dally, "A Stream Processor Development Platform," in *International Conference on Computer Design (ICCD)*, 2002.

[117] K. Group, "OpenGL," 1992. `https://www.khronos.org/opengl/`.

[118] Microsoft, "Programming Guide for DirectX [Online]," 2016. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476455(v=vs.85).aspx`.

[119] AMD, "Radeon Open Compute [Online]," 2017. `https://rocm.github.io/`.

[120] AMD, "ROCK_syscall," 2017. `https://github.com/RadeonOpenCompute/ROCK_syscall`.

[121] AMD, "ROCT_syscall," 2017. `https://github.com/RadeonOpenCompute/ROCT_syscall`.

[122] AMD, "HCC_syscall," 2017. `https://github.com/RadeonOpenCompute/HCC_syscall`.

[123] AMD, "Genesys_syscall_test," 2017. `https://github.com/RadeonOpenCompute/Genesys_syscall_test`.

[124] R. Draves, B. Bershad, R. Rashid, and R. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," in *Symposium on Operating Systems Principles (SOSP)*, 1991.

[125] O. Shivers and M. Might, "Continuations and Transducer Composition," in *International Symposium on Programming Languages Design and Implementation (PLDI)*, 2006.

[126] AMD, "Graphics Core Next 3.0 ISA [Online]," 2013. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/07/AMD_GCN3_Instruction_Set_Architecture.pdf`.

[127] Intel, "Intel Open Source HD Graphics Programmers Reference Manual [Online]," 2014. `https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-bdw-vol02b-commandreference-instructions_0_0.pdf`.

[128] NVidia, "NVIDIA Compute PTX: Parallel Thread Execution," 2009. `https://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf`.

[129] PCI-SIG, "Atomic Operations [Online]," 2008. `https://pcisig.com/sites/default/files/specification_documents/ECN_Atomic_Ops_080417.pdf`.

[130] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 39–58, ACM, 2016.

[131] T. Rogers, M. O'Connor, and T. Aamodt, "Cache-Conscious Wavefront Scheduling," in *International Symposium on Microarchitecture (MICRO)*, 2012.

[132] T. Rogers, M. O'Connor, and T. Aamodt, "Divergence-Aware Warp Scheduling," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[133] A. ElTantawy and T. Aamodt, "Warp Scheduling for Fine-Grained Synchronization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[134] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and Fair Multiprogramming in GPUs via Pattern-Based Effictive Bandwidth Management," in

International Symposium on High Performance Computer Architecture (HPCA), 2018.

[135] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[136] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[137] AMD, "ROCm: a New Era in Open GPU Computing [Online]," 2016. `https://radeonopencompute.github.io/index.html`.

[138] S. N. Lab, "miniamr." `https://proxyapps.exascaleproject.org/app/miniamr/`.

[139] Z. Brown, "Asynchronous System Calls," in *Ottawa Linux Symposium*, 2007.

[140] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending mpi to accelerators," *PACT 2011 Workshop Series: Architectures and Systems for Big Data*, Oct. 2011.

[141] L. Oden, "Direct Communication Methods for Distributed GPUs," in *Dissertation Thesis, Ruprecht-Karls-Universitat Heidelberg*, 2015.

[142] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 2595–2605, Oct 2014.

[143] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

[144] IBM, "IBM Spectrum MPI," 2017. http://www-03.ibm.com/systems/spectrum-computing/products/mpi/.

[145] K. Atasu, F. Doerfler, J. van Lunteren, and C. Hagleitner, "Hardware-accelerated regular expression matching with overlap handling on ibm poweren processor," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 1254–1265, IEEE, 2013.

[146] I. Q. A. Technology, "Integrated cryptographic and compression accelerators on intel® architecture platforms." http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/integrated-cryptographic-compression-accelerators-brief.pdf.

[147] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 468–479, IEEE, 2013.

[148] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[149] J. Goodacre and A. Cambridge, "The evolution of the arm architecture towards big data and the data-centre," in *VHPC@ SC*, pp. 4–1, 2013.

[150] "Hsa foundation." http://hsafoundation.com.

[151] HSAFoundation, "Hsaruntimeamd." https://github.com/HSAFoundation/HSA-Runtime-AMD.

[152] M. Daga and M. Nutter, "Exploiting coarse-grained parallelism in b+ tree searches on an apu," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 240–247, IEEE, 2012.

[153] "Comd." https://proxyapps.exascaleproject.org/app/comd/.

[154] "minife." https://proxyapps.exascaleproject.org/app/minife/.

[155] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[156] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[157] A. N. Laboratory, "Xsbench." https://proxyapps.exascaleproject.org/app/xsbench/.

[158] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 340–349, IEEE, 2011.

[159] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365, IEEE, 2015.

[160] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: a mechanism for speculative address translation," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 307–318, 2011.

[161] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, 2010.

[162] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 237–248, 2013.

[163] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pp. 26–35, 2008.

[164] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 383–394, 2013.

[165] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 62–63, IEEE, 2011.

[166] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 178–189, IEEE, 2014.

[167] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 66–78, 2015.

[168] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 558–567, IEEE, 2014.

[169] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 258–269, IEEE, 2012.

[170] B. Pham, J. Veselỳ, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 1–12, 2015.

[171] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 223–234, IEEE, 2015.

[172] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly tlb designs," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–222, IEEE, 2015.

[173] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 568–578, IEEE, 2014.

[174] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward cache-friendly hardware accelerators," in *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pp. 1–6, 2015.

[175] G. Albuquerque Cox, *Improving and complementing virtual memory using hardware techniques*. PhD thesis, Rutgers University-School of Graduate Studies, 2018.

[176] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "riommu: Efficient iommu for i/o devices that employ ring buffers," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 355–368, 2015.

[177] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular gpu applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 180–192, IEEE, 2018.

[178] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 503–518, 2018.

[179] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a gpu memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 136–150, 2017.

[180] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 49–63, 2019.