VectorVisor: A Binary Translation Scheme for Throughput-Oriented GPU Acceleration

Samuel Ginzburg, *Princeton University;* Mohammad Shahrad, *University of British Columbia;* Michael J. Freedman, *Princeton University*

https://www.usenix.org/conference/atc23/presentation/ginzburg

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

VectorVisor: A Binary Translation Scheme for Throughput-Oriented GPU Acceleration

Samuel Ginzburg Mohammad Shahrad[†] Michael J. Freedman

Princeton University [†]*University of British Columbia*

Abstract

Beyond conventional graphics applications, general-purpose GPU acceleration has had significant impact on machine learning and scientific computing workloads. Yet, it has failed to see widespread use for server-side applications, which we argue is because GPU programming models offer a level of abstraction that is either too low-level (e.g., OpenCL, CUDA) or too high-level (e.g., TensorFlow, Halide), depending on the language. Not all applications fit into either category, resulting in lost opportunities for GPU acceleration.

We introduce VectorVisor, a vectorized binary translator that enables new opportunities for GPU acceleration by introducing a novel programming model for GPUs. With VectorVisor, many copies of the same server-side application are run concurrently on the GPU, where VectorVisor mimics the abstractions provided by CPU threads. To achieve this goal, we demonstrate how to (i) provide cross-platform support for system calls and recursion using continuations and (ii) make full use of the excess register file capacity and high memory bandwidth of GPUs. We then demonstrate that our binary translator is able to transparently accelerate certain classes of compute-bound workloads, gaining significant improvements in throughput-per-dollar of up to 2.9× compared to Intel x86-64 VMs in the cloud, and in some cases match the throughput-per-dollar of native CUDA baselines.

1 Introduction

Server-side GPU acceleration has become ubiquitous, with all major cloud providers offering virtual machine instances with attached GPUs. GPU workloads such as graphics and machine learning have found widespread adoption due to the superior throughput-per-dollar that GPUs offer.

Typical approaches to accelerating these workloads on GPUs use domain-specific programming languages (DSLs). DSLs for GPUs heavily restrict which abstractions can be used by developers to write applications, and in particular forces them to use parallel abstractions. For example, machine learning programming systems such as TensorFlow [24] require users to specify programs as a series of operations performed on n-dimensional arrays. Approaches to extracting parallelism on GPUs for graphical workloads such as Halide [73] enforce more extreme restrictions such as requiring developers to express image operations as pure mathematical functions, defining the value of each function at each point. Other DSLs targeting batch dataflow workloads require developers to express their program using built-in parallel functions, which impose additional restrictions on application logic [75].

Developers who cannot express their application logic using these restricted abstractions are stuck manually rewriting applications in OpenCL or CUDA, which expose a low-level programming interface. Complex programs that use large pre-existing libraries, or where extracting parallelism is difficult, can be time-consuming to write and require drastic modifications to run using GPUs.

In this paper, we explore the feasibility of an alternative programming model for GPUs—where we take existing single-threaded programs and execute many copies of them using GPU threads. Each GPU thread corresponds to an emulated CPU thread, running a single instance of the program. Unlike prior approaches [37] which utilize interpretation, we translate the input program to native GPU code, substantially boosting performance while enabling a wider variety of target languages and runtimes. Unlike OpenCL or CUDA programs, we provide support for system calls and a CPU-like flat memory model. While less efficient than manual translation, this approach substantially reduces the barrier to accelerating throughput-oriented workloads using GPUs, ultimately improving the throughput and cost efficiency of applications that would otherwise run on CPUs.

Many applications written to run on CPUs are single-threaded programs, often implemented using high-level programming languages with large imported libraries. Without modification, these applications do not map cleanly to existing GPU programming models (e.g., those using language-level parallel functions such as in TensorFlow or Halide). Instead, these workloads process requests independently, with

no inter-request synchronization or communication. Examples of these workloads include cryptographic operations, image manipulation, and compression. These workloads are generally amenable to GPU acceleration [60, 67, 73], but are frequently run on CPUs instead.

Effectively enabling this programming model requires one to overcome several technical challenges in dealing with the substantial differences between GPUs and CPUs. These differences include how programs are executed—e.g., in which programs are run to completion without preemption—as well as a lack of support in GPUs for system calls. Further, failing to take differences in GPU memory hierarchies into account can result in an order of magnitude decrease in read and write performance. Prior approaches to running unmodified programs on GPUs suffer from poor performance due to the overheads of interpretation [37] as well as compatibility issues such as the lack of support for system calls [45].

To explore this unique programming model for GPUs, we built VectorVisor—a system which utilizes a vectorizing binary translator for GPUs. VectorVisor is designed to accelerate existing and unmodified programs that are designed to run on CPUs but can benefit from GPU acceleration. Target programs are automatically translated to run on GPUs efficiently, eliminating the need for complex manual translation. In particular, VectorVisor uses WebAssembly [54] as the intermediate binary format, which enables secure, fast, and efficient compilation for a wide range of applications.

We overcome the differences in program execution and memory hierarchy by translating WebAssembly programs to run directly on the GPU as opposed to using interpretation. We show that the remaining differences between CPUs and GPUs can be bridged with a combination of three techniques:

Continuations: CUDA and OpenCL do not provide support for preempting running applications in addition to lacking support for system calls. Without preemption, we cannot dispatch system calls, making it impossible to run complex and unmodified programs. To bypass this issue, we implement *continuations* for OpenCL C. Continuations are language-level primitives that allow us to save the program state at arbitrary locations, and then resume execution at a later time. Doing so allows us to pause and resume running GPU kernels, and to provide support for system calls. We also benefit from the portability of our approach—enabling VectorVisor to be run with multiple GPU vendors (e.g., NVIDIA, AMD).

WebAssembly: WebAssembly (WASM) binaries are designed with performance, portability, and security in mind. Many popular languages can compile to WASM (e.g., Rust, Go, C, C++, AssemblyScript, and more), making it an ideal intermediate format. WASM binaries are designed with runtime JIT compilation in mind, persisting vital information not present in x86 binaries. WASM semantics provide VectorVisor with memory alignment information, register allocation hints, type-checks on operations, and language-enforced structured control-flow. We heavily utilize this information to deal with challenges such as efficiently making use of the substantially larger per-thread [14] register space on GPUs—which is crucial for maximizing performance. Other important performance optimizations are also enabled through this compile-time information.

Memory Interleaving: GPUs organize threads in *warps*, or groups of threads. Each thread in a warp has a numerical index, and threads with adjacent indices must access adjacent bytes for optimal performance—so that memory accesses can be coalesced together. Coalesced memory accesses enable GPUs to maximize memory bandwidth usage at the cost of a more complex programming model. To bridge the differences between the GPU and CPU memory hierarchies, we automatically interleave the memory of each virtual machine running on the GPU to transparently coalesce all memory accesses.

We demonstrate VectorVisor's capabilities to accelerate several unmodified, third-party applications which use popular open-source libraries. We then evaluate VectorVisor's efficacy using nine benchmarks with throughput-per-dollar as our primary metric. Selected benchmarks include multiple classes of workloads, some of which reflect ideal applications of VectorVisor, with others reflecting the limitations of our programming model. Comparisons against native x86-64 and WebAssembly versions of each benchmark are provided, showing that VectorVisor can achieve superior throughput-per-dollar. We also provide native CUDA versions of two benchmarks to evaluate the efficacy of our translation. Our paper makes the following contributions:

1. We introduce a novel cross-platform approach to running lightweight virtual machines using GPUs, where VMs securely execute native code to maximize performance and support multiple high-level languages.

2. We show that support for system calls can be efficiently provided using continuations in addition to supporting recursion and indirect calls in OpenCL.

3. We demonstrate that we can emulate a flat memory model using an efficient memory interleave, enabling existing programs to leverage the high memory bandwidth of GPUs.

4. We explore the implications of batch size, latency, and throughput on VectorVisor's programming model and discuss which categories of workloads are optimal for it.

5. We discuss the limitations of our system and optimal GPU configurations for it.

2 Motivation and Challenges

The past several years have shown a large increase in the availability of cloud accessible GPUs. GPUs that cost thousands of dollars are now available at affordable prices per hour. Developers can quickly test if accelerating their program using

a GPU is cost-effective without large up-front investments in GPU hardware. However, despite having strong parallel processing power and cloud availability, GPUs are not often used for running high concurrency server-side applications.

Translating programs originally intended to execute on CPUs to run on GPUs is difficult due to the substantial differences between the execution models and memory hierarchies. Today's approaches to tackling these issues either require strong language-level restrictions with unintuitive stumbling blocks for developers, or slower automated approaches such as interpretation [37].

2.1 Execution Model Differences

Taking advantage of the throughput that GPUs offer requires using a different execution model than CPUs offer. GPUs feature restrictions on both the application runtimes and control flow that limit the set of possible workloads.

Runtime Limitations: CUDA and OpenCL are the two most popular compute APIs available for GPUs, and they share a near identical programming model. Programs that run on GPUs (GPU kernels) are submitted and execute until completion without preemption. High-level languages targeting general-purpose GPU programming such as CUDA C++ and OpenCL C feature restrictions on the usage of standard libraries, recursion, indirect function calls, variable length arrays, virtual functions, and templates [12, 16]. Support for other common features such as system calls and preemption are absent, further restricting the set of programs that can run.

Divergence: Unlike CPUs, which allow for different hardware threads to execute different instructions, GPUs organize threads into groups of threads (warps). Each warp shares a program counter, so all threads execute the same instruction on each clock cycle. Support for conditional branching is provided by executing no-ops for threads that have diverged, while the remaining threads block on threads executing the conditional branch. This results in the serialized execution of branches. Programs with substantial divergence are not able to efficiently use GPU resources as a result [33, 43, 51].

2.2 Memory Hierarchy Differences

GPUs and CPUs handle memory accesses differently due to the different design constraints imposed upon the hardware. CPUs optimize for reduced memory latency for all threads of execution, so they feature large cache sizes to minimize accesses to main memory. In contrast, GPUs seek to maximize memory bandwidth. GPUs can achieve $3\times$ the memory bandwidth of a comparable CPU [5, 8]. While GPUs can achieve higher memory bandwidth, memory latency on a given GPU can be up to $2.75\times$ worse than a comparable CPU [61, 66]. These differences in the memory hierarchy between GPUs and CPUs have two key implications for developers:

Register Space: At a high level, the memory hierarchies of CPUs and GPUs are similar, with both devices featuring registers, data caches, and byte-addressable memory. However, GPUs feature substantially larger register files. Each thread on recent NVIDIA GPUs can have a maximum of 255 32-bit register values [12] (just under 1 KiB of storage). In contrast, the x86-64 instruction set architecture has 16 64-bit general purpose registers (128 bytes of storage). Additionally, GPUs typically feature far more threads of execution than CPUs, further magnifying the difference. *Making use of this extra space is critical to maximizing performance on GPUs* [36, 70].

Memory Accesses: Rules regarding efficient memory access patterns are different for GPUs. GPUs require that programs perform *coalesced memory accesses*. Similar to CPUs, locality of memory accesses allows reads and writes to be cached, and is required to get optimal performance. However, GPU kernels must also ensure the locality of memory accesses across threads. Threads in a GPU warp are numerically indexed, and adjacent threads must access adjacent bytes of memory as a general rule. Otherwise, memory accesses within a warp can be serialized. Without proper memory coalescing GPU memory bandwidth can be cut by up to $32\times$ [12].

In addition to performance drops, GPUs can have stricter memory access policies than CPUs. Ideally, memory accesses are naturally aligned—meaning N byte accesses must be N-bytes aligned. Unaligned accesses cause running GPU kernels to fault on NVIDIA GPUs [12], causing programs that would have run correctly on a CPU to crash on a GPU.

3 System Design

In this section we introduce VectorVisor [1], a vectorizing binary translator for GPUs designed to leverage the implicit parallelism provided by our programming model. Our approach enables us to run many instances of unmodified programs on the GPU concurrently, making it far easier to utilize the substantial parallelism that GPUs offer.

We first describe our programming model in depth, where we explain how developers can leverage VectorVisor to accelerate programs. We characterize a set of ideal workloads for VectorVisor and explore the limitations of our programming model. Following this, we provide a high-level overview of VectorVisor's primary system components. To enable our simple and easy to use programming model, VectorVisor automates data transfer to and from the GPU. We illustrate this by showing the life cycle of a request processed by VectorVisor. Lastly, we provide an example of a short program, and how we transform it to run on the GPU.

3.1 Programming Model, Target Workloads

In contrast to OpenCL or CUDA, VectorVisor mimics the abstractions provided by CPU threads, treating each individual GPU thread as a small virtual machine. Each VM operates on a statically allocated chunk of memory, fully isolated from

[1] https://github.com/SamGinzburg/VectorVisor

other VMs. This memory model does not support inter-VM communication, and thus prevents deadlocking. Many copies of the same program are mapped to GPU threads, which then operate on distinct inputs.

This approach to parallelism for GPUs enables developers to run complex and *unmodified* single-threaded programs originally written for the CPU. Developers can leverage GPU acceleration without learning complex programming models or rethinking the logical structure of their programs.

VectorVisor is designed to function with unmodified workloads, but not all programs are equally amenable to acceleration using our programming model. Data parallel, latency-insensitive, and compute-bound 'serverless-like' workloads are ideal targets for GPU acceleration using VectorVisor. For a subset of these workloads, correct manual translation can be difficult without domain-specific knowledge—and those workloads represent an ideal use-case for VectorVisor. Suitable workloads share a number of characteristics:

High Execution Volume: VectorVisor relies on running many instances of the same program concurrently, instead of accelerating a single execution. Therefore, the more instances packed on the GPU, the higher the cost efficiency. Naturally, the latency QoS of the application should be able to afford the added batching latency prior to execution.

Application Limitations: VectorVisor runs unmodified programs where possible, but some abstractions are expensive to emulate on GPUs. Recursion and indirect calls reduce application performance due to how we implement them (explained further in Section 3.3).

Navigating tradeoffs between application concurrency and heap size are key to maximizing performance when using VectorVisor. We experimentally found in our evaluation that running 4096-6144 VMs with a heap size of 3-4 MiB proved optimal for our selected workloads. However, it is possible to run VectorVisor with varying degrees of concurrency—adjusting for different heap sizes.

Lastly, floating point differences between CPUs and GPUs can result in different outputs for applications [11, 37, 65], depending on the specific application and compiler.

Low Divergence: Ideal workloads should minimize program divergence in order to fully utilize the superior throughput and memory bandwidth that GPUs can offer.

Data Transfer Overheads: VectorVisor automates data transfer to and from the GPU; however, the overhead involved can be a substantial fraction of end-to-end request time. Maintaining a high ratio of GPU compute to input and output size is ideal for maximizing VM throughput.

3.2 Design Overview

VectorVisor consists of two key components, the binary translator (compiler) and the vectorized virtual machine monitor (VMM). We show an overview of VectorVisor in Figure 1, showing the role of each component as well as the life cycle of an incoming request to the system. Requests are first queued externally to VectorVisor, before being batched by the VMM, and submitted to the VMs running on the GPU—which are blocked on a system call awaiting input. We provide a pre-configured web server that automatically handles all data transfer to the VMM. After executing a batch of requests, responses are returned via another system call, and then back to the web server. This approach enables VectorVisor to be used as a drop-in replacement for existing systems, without the need for developers to manually batch incoming requests.

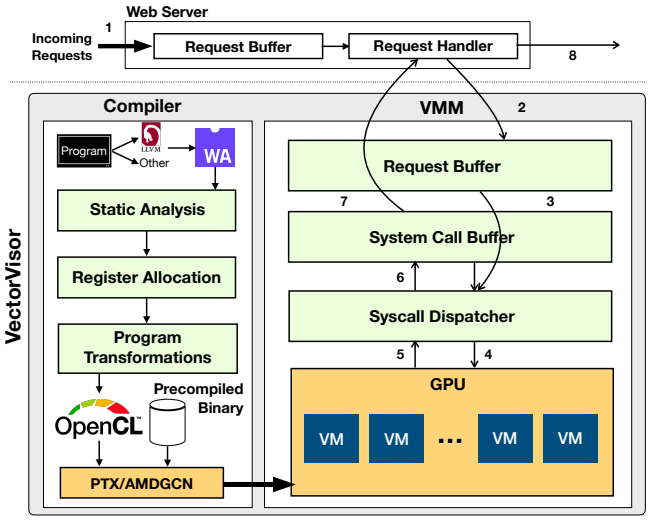

Figure 1: *System Overview*.

Binary translation is a separate process, occurring before applications run. Programs are compiled from any language targeting LLVM [64] (e.g., Rust, C, C++) into WebAssembly, our intermediate binary format. We then compile WebAssembly to OpenCL C. Targeting OpenCL C enables VectorVisor to support multiple GPU vendors. This approach allows us to run existing programs without the need to worry about complex language semantics—we only need to concern ourselves with WebAssembly semantics which are far simpler than alternatives such as LLVM IR and directly compiling high-level languages. LLVM IR places minimal restrictions on control flow structures, and can represent programs that are impossible for any GPU to run. WebAssembly only provides structured control flow by design, ensuring that programs can always be translated to run on the GPU [54]. Alternative approaches that directly compile high-level languages to run using GPUs require substantial engineering effort, and can run into compatibility issues [37].

Our system design features a number of novel contributions that we employ to bridge the substantial differences that exist between GPUs and CPUs described in Section 2. Our contributions succeed in bridging most of the gaps in capabilities between CPU and GPU runtimes. Recursive and indirect functions limit performance for some workloads but do not limit our functionality or correctness.

```
;; // Pseudocode in C:
;; int main(void) {
;;   // fd 1 == stdout
;;   char text[] = "ABCD\n";
;;   return write(1, text, strlen(text));
;; }
(module
  (import "wasi_unstable" "fd_write"
  (func $fd_write (param i32 i32 i32 i32) (result i32
      )))
  (memory (;0;) 1)
  (export "memory" (memory 0))
  (export "_start" (func $_start))
  (func $_start (result i32)
      i32.const 1  ;; stdout
      i32.const 0  ;; iovec ptr
      i32.const 2  ;; entries
      i32.const 24 ;; out bytes
      call $fd_write
  )
  (data (i32.const 0) "\10\00\00\00\02\00\00\00")
  (data (i32.const 8) "\12\00\00\00\02\00\00\00ABCD")
)
```

Figure 2: *WebAssembly Example*. We show an example of a simple program which makes a single system call.

3.3 Compiler

VectorVisor uses a binary translator (compiler) to translate input programs such that they can run on the GPU. The role of the compiler is to automate away the difficulties involved in writing programs for the GPU that are outlined in Section 2. We explore a set of techniques for enabling the execution of unmodified programs, which we demonstrate using a simple example of an input program.

3.3.1 Compiling WebAssembly

WebAssembly (WASM) [54], is a low-level language designed for performance, size, portability, and security. WASM binaries differ significantly from x86-64 binaries, as they are designed to be recompiled before runtime, retaining significant compilation information that can be used. Using WASM as an intermediate format simultaneously allows us to avoid dealing with the complex semantics of higher-level languages (e.g., Rust, C, C++) while also improving the performance of VectorVisor. We make use of this information in three places within our compiler:

1. Register Allocation: Recent NVIDIA GPUs have up to 255 32-bit registers per thread [12], providing roughly 8× the amount of storage per CPU thread ignoring vector registers. Traditional x86-64 binaries target CPUs with only 16 64-bit general purpose registers. Static analysis could conceivably be used to place stack allocations in x86-64 binaries into GPU registers, but WebAssembly provides a more convenient solution. In contrast to x86-64 binaries, WASM is a stack-based virtual machine and does not explicitly allocate registers [54]. Instead, values are placed either onto the stack or into local variables. Figure 2 shows an example WebAssembly program, which places four integers onto the stack. During compilation, we are able to store these values directly into variables which the backend GPU compiler (OpenCL C compiler) can then place into GPU registers. This approach allows the OpenCL C compiler to place values into registers that an x86-64 compiler would have placed on the stack.

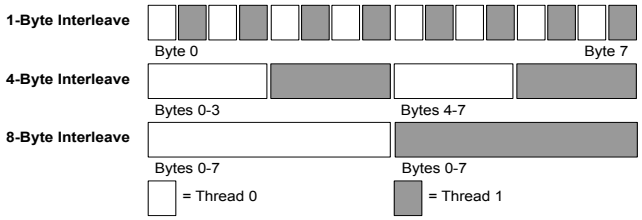

Figure 3: *Memory Interleaving* Examples of 1, 4, and 8-byte interleavings for a system with 2 threads are shown.

2. Runtime support: Most programs require some degree of modification to run on GPUs. Memory allocation, locking primitives, and threading primitives make assumptions about the underlying system that are false on GPUs. However, many such modifications are already performed by WebAssembly compilers. WASM binaries not only provide substantial compilation information, but also a "batteries-included" set of runtime modifications. Compilers targeting WebAssembly typically compile programs with a modified standard library with the necessary modifications already made.

3. Memory Alignment: Misaligned accesses cause running programs to crash when run using NVIDIA GPUs [12]. Handling misaligned accesses can be done at runtime by performing multiple aligned reads, but doing so introduces runtime overhead. Emitting optimized code for aligned accesses substantially boosts application performance. WASM binaries contain alignment information (e.g., the `align` attribute) that we can use to optimize reads and writes. However, the align attribute is only a hint, and as per the WASM specification, programs are expected to run correctly even with incorrectly specified alignments [54]. In practice, WASM binaries compiled by LLVM always contain the correct alignment information. By restricting the set of programs that we run to those compiled by LLVM, we can leverage these compilation hints safely to improve the performance of VectorVisor. VectorVisor supports running programs with and without this optimization using compiler-flags.

3.3.2 Memory Interleaving

GPUs have strict memory access rules to obtain optimal performance. As described in Section 2.2, GPU kernels must coalesce memory accesses to maximize memory bandwidth. Doing so requires developers to interleave objects in memory, such that adjacent threads access adjacent bytes, breaking the abstraction of a flat memory model. Other aspects of the flat memory model, such as process (or VM) memory isolation are also absent on GPUs by design.

VectorVisor provides the abstraction of a flat memory model to developers, automatically interleaving the address space of underlying virtual machines (threads) on the GPU

```
__kernel  void wasm_entry(...) {
  // Set up the stack, heap, buffers, ...
  do {
    /* call the next func/continuation */
    switch (*entry_point) {
      case 0:
        ___start(...);
        break;
      default:
        return;
    }
  // Check if we are done executing
  } while(*sfp!=0 && *syscall_number==-2);
}
```

Figure 4: The *trampoline* function serves as the entry point to each GPU kernel.

to provide both performance and security. This approach allows existing programs to run, while also extracting the full performance benefits of a GPU—assuming that running VMs exhibit similar memory access patterns. Randomized memory access patterns or significant program divergence can reduce memory bandwidth. Figure 3 shows how memory is interleaved across VMs in VectorVisor. Memory is organized into *cells* of contiguous bytes. Cell addresses are computed using the following pointer arithmetic (C operator precedence):

$$cell\_addr = (\frac{offset}{ileave}) \times (num\_vms \times ileave) + (vm\_idx \times ileave) + mem\_base$$

Where the interleave (*ileave*) represents the byte-width of the interleaving (e.g., 1, 4 or 8), the *offset* is the zero-indexed WebAssembly address, and *mem_base* is the base address of the allocated chunk of memory. Memory accesses are rewritten to operate on cells, with misaligned and larger (e.g., 8, 16-byte value) accesses requiring multiple operations. Our approach enables us to support 1, 4, and 8-byte interleavings, with larger interleavings typically achieving superior memory bandwidth.

WASM memory is represented as a zero-indexed linear array of bytes with pointers in the range of 0–2^{32}-1 and does not expose virtual addresses to running VMs. The relative addressing model WASM uses enables the compiler to control the virtual addresses of all memory reads and writes. Our cell address computation prevents VMs from computing cell adresses which belong to other VMs— preventing out-of-bounds accesses from corrupting or leaking data and providing memory isolation by construction.

3.3.3 GPU Preemption

Section 2.1 described the limitations of GPU programming models such as OpenCL and CUDA. Common features of programs such as system calls, recursion, and indirect calls vary in support—with system calls being absent from both OpenCL and CUDA. To fully mimic the execution environment provided by a CPU in VectorVisor, we support all three features. Implementing these features within OpenCL C requires us to provide support for preempting running programs. We provide support for preemption in VectorVisor by extending OpenCL C with support for continuations. Continuations provide the abstraction of being able to pause and resume programs at arbitrary points. To maximize the performance of VectorVisor, we leverage several compiler optimizations to reduce the overhead they introduce.

```
# 'c' is the called continuation
def example_fn(c, ...):
# context restore handler
switch (context):
  case 0:
    goto resume0;
  case n:
    goto resume_N;
  default:
    # Direct call, start from the top
# Indirect function call
switch(func_ptr):
  case 0:
    return c(example_fn, 0, resume0, ...)
    resume0:
  case n:
    return c(example_fn, 1, resume1, ...)
    resume1:
  default:
    trap;
# Optimized calls can be issued directly
call_example_func(...)
# Standard function call
return c(example_fn, 2, resume2, ...)
resume2:
# Recursive function call]
# In this case example_fn == 3
return c(example_fn, 3, resume3, ...)
resume3:
# System call
# These functions return control to the VMM
return c(example_fn, 4, resume4, ...)
resume4:
```

Figure 5: *Transformed Program* Pseudocode example of how different types function calls are implemented.

Continuations. Continuation-Passing Style [81] (CPS) is a relatively uncommon programming style where functions take in an additional parameter (the continuation), and instead of returning a value call the provided continuation with the return value. CPS with trampolining [27] is similar to standard CPS, with the difference being that function calls return continuations instead of just calling the provided continuation. A control operator (trampoline function), is used to repeatedly call the returned continuations. Figure 4 shows the trampoline function used in VectorVisor, which is the main entry point to each running GPU kernel. Implementing CPS with trampolines in this manner enables VectorVisor to preempt running GPU kernels at arbitrary locations—although we only return control to the CPU when either every VM is finished executing or when every VM is blocked on a system call. In Figure 5 we see that the only difference between recursive, indirect, and standard calls is the returned continuation (which encapsulates the program control state). This approach makes it easy to bypass OpenCL C language-level restrictions and provide support for recursive and indirect calls.

Compiler Optimizations. Naively implementing CPS with trampolines enables support for system calls and recursion

with large runtime overheads. To obtain better performance, VectorVisor performs static analysis to minimize the size of saved program contexts. We apply liveness analysis in addition to leveraging WASM type and control flow information to enable (1) incremental context saving, (2) loop-invariant code motion, and (3) WebAssembly-specific optimizations.

Liveness is associated with local usage inside WASM stack frames, and we insert all context save and restore operations around control flow instructions (e.g., `block`, `loop`, `br`, `br_if`, and `end`) and function (or system) calls. Runtime taint tracking is used to further enhance our liveness estimates.

Stack frame contexts are saved incrementally—only saving values written to since the previous context save operation. Liveness estimates are used to minimize context sizes in addition to only restoring live values when resuming continuations or unwinding stack frames. Loops without recursive or indirect calls can be further optimized—with context saving and restoring operations hoisted out of the loop. WASM function type signatures are used to translate amenable indirect calls into direct calls by filtering possible indirect call targets.

3.3.4 Profile-Guided Optimization

Minimizing the overhead of translating recursive and indirect calls is key to running complex applications. Compiler optimizations eliminate much of the overhead in the common-case. Edge cases, such as heavy usage of indirect and recursive calls in a tight loop remain a challenge. While recursion often cannot be eliminated without restructuring programs, indirect calls are easier to remove [25, 34]. Most indirect calls in high-level languages have only one target—with on average 73.5% of indirect call sites in Java programs being monomorphic [59]. Despite aggressive monomorphization in the Rust compiler [26], up to 37% of the most popular Rust libraries reduce code size by not removing optimizable indirect calls where possible [85]. Up to 98% of indirect calls in Java programs can be optimized out entirely [59].

We package a separate tool for instrumenting binaries, to implement profile-guided optimization for VectorVisor. Each program is instrumented and run using sample inputs representative of the overall workload. Using profiler data, we replace all indirect calls with less than 15 seen call targets with direct calls. To avoid emitting indirect calls to handle unseen targets, we instead emit panic handlers which check for valid targets.

3.3.5 Soundness

VectorVisor performs a 1-to-1 translation for all operations in input WebAssembly programs (e.g., stack operations, memory access, arithmetic, control flow). Limitations on the soundness of our approach come from (1) Compilation to WebAssembly and (2) Optimizations.

Most common workloads can be recompiled to WebAssembly without problems, but programs which rely on specific x86 instructions (e.g., 80-bit floats), language implementation details (e.g., undefined and implementation defined behavior), and complex language runtimes with unimplemented features (e.g., Go) can experience correctness issues.

Compiler-flags and tools (e.g., `wasm-snip`) are used to replace panic-related functions with `unreachable` statements. Unrecoverable errors can be expensive to handle, and in most cases replacing them with program aborts has no impact on correctness. Profile-guided optimization (PGO) can reduce indirect call counts, significantly improving performance in some cases. Our implementation of PGO only includes function calls we observe as potential targets at indirect call sites, aborting on unseen call targets. In practice, the indirect call targets we observed did not vary significantly with user-input beyond what we observed during profiling.

3.4 VMM

VectorVisor's VMM handles all data transfer between the running VMs on the GPU and CPU, as well as executing all system calls. The VMM greatly simplifies the use of VectorVisor by developers, avoiding the need to manage data transfer manually or to batch incoming requests.

Support for dispatching system calls is provided through the WebAssembly System Interface (WASI). We implement two custom WASI system calls—which are used to create a serverless-like event handler API for running VMs. Other implemented calls are primarily used to initialize language runtimes (e.g., reading environmental variables), support random number generation, serve as synchronization barriers (e.g., block on a subset of VMs), and perform simple IO (e.g., error logging).

Incoming requests are buffered using the request buffer, while system calls use an alternate buffer, as shown in Figure 1. Double buffering adds some overhead, but enables VectorVisor to overlap expensive network IO with on-GPU execution time. Sufficiently compute-intensive workloads prevent workloads from bottlenecking on the VMM, which can process thousands of VMs per-CPU. VectorVisor supports using pinned memory transfers with multiple GPU vendors (e.g., NVIDIA, AMD) to further optimize data transfer speeds—with vendor-specific optimizations [1, 4].

4 Evaluation

In this section, we present an evaluation of VectorVisor. First, we discuss the efficiency of (1) our memory interleaving and (2) system call implementation. Second, we explore a variety of modified and unmodified workloads to better understand the tradeoff space of our novel approach to accelerating programs. In several cases we show that we obtain superior throughput-per-dollar against x86 CPUs. Breakdowns of the end-to-end latencies of each benchmark are provided as well to explain our results. Finally, we evaluate the efficiency of our translation against handwritten CUDA baselines.

Instance Name	CPU	GPU	Cost/Hr
g4ad.xlarge	Intel Cascade Lake	AMD Radeon Pro V520	$0.3785
g4dn.xlarge	Intel Cascade Lake	NVIDIA T4	$0.526
g4dn.2xlarge	Intel Cascade Lake	NVIDIA T4	$0.752
g5.xlarge	AMD EPYC 7002	NVIDIA A10G	$1.006
g5.2xlarge	AMD EPYC 7002	NVIDIA A10G	$1.212
c5.xlarge	Intel Cascade Lake	N/A	$0.17
c5a.xlarge	AMD EPYC 7002	N/A	$0.154

Table 1: *Hardware Configurations.* Prices as of 1/5/2023.

4.1 Methodology

Testbed. We evaluated VectorVisor using Amazon Web Services (AWS). Five different VM types were used to compare against x86-64 baselines, and an additional two larger instances types were used to compare VectorVisor against CUDA baselines. We provisioned three VMs with attached GPUs (g4ad.xlarge, g4dn.xlarge, g5.xlarge), each with 4 vCPUs and 16 GiB of memory. Two additional compute-optimized VMs were used for evaluating CPU performance (c5.xlarge, c5a.xlarge), each with 8 GiB of memory and 4 vCPUs. Lastly, we used a single invoker VM (c5.8xlarge) for sending requests. These instances were used to obtain the results in Figures 8 and 9. Double extra large (2xl) instances have 2× the memory and CPU of smaller (xlarge) instances. These instances were used (in addition to xlarge instances) to evaluate handwritten CUDA programs. CUDA results which use 2xl instances can be found in Table 4. All VMs are allocated in `us-east-1`, in the same availability zone. Benchmarks are evaluated end-to-end over the network with IO and system overheads included in all measurements.

Some hardware configurations could not be evaluated due to AMD-specific bugs. AMD v520 GPUs, which are the only cloud-available AMD GPU on AWS, are unsupported in ROCm [21, 23] resulting in runtime crashes. Two benchmarks (Strings-Go and Strings-AScript) are built with WebAssembly-focused runtimes and do not have evaluated x86-64 configurations. Detailed results for all system configurations can be found in Tables 5 and 6 (Appendix).

Table 1 shows the hardware each VM has attached. NVIDIA configurations use the latest CUDA 12 backend, while AMD GPUs use ROCm 5.4.0 with the latest AMDGPU-Pro driver. For our GPU instances, we do not run any fraction of our workload on the available CPU cores to focus on evaluating GPU performance using VectorVisor. Undoubtedly, a hybrid CPU/GPU deployment would be more cost-efficient. We leave the exploration of heterogeneous deployments to future work. VectorVisor runs each benchmark using a 4- or 8-byte memory interleaving.

Workloads. Workloads are run using a cross-platform 'serverless-like' event-loop. Figures 11, 12, and 13 (Appendix) demonstrate how programs are written to run using our event-loop with x86-64, WASM, and VectorVisor. Our benchmarks evaluate VectorVisor using 'as-is' open-source library code in addition to optimized code written to run more efficiently. We explore the translation of lightweight (e.g., Rust) and complex language runtimes featuring garbage collection such as Go using TinyGo [22], and AssemblyScript [19]—a TypeScript-like language. Our x86-64 baselines are compiled at -O3 with SIMD enabled. Two benchmarks (Blur-Bmp, PHash-Modified) were manually rewritten to run natively using NVIDIA CUDA APIs to explore the efficiency of our translation. Table 2 shows each benchmark, its category, whether it features code that runs inefficiently in our system, whether we had to modify the imported library, batch sizes, and the total number of downloads on crates.io, a public repository for Rust libraries [10]. Benchmarks are run for 10 minutes to compute the average throughput (RPS).

Example Functions. Perceptual hashing is widely used in industry, such as by Facebook [6, 7], to cross-reference a given image against a database of images. We evaluate an open-source implementation of Blockhash—a variation on existing perceptual hashing algorithms [9, 90]. To further evaluate the efficiency of our translation, we also evaluate a modified blockhash library that we optimized to run more efficiently using VectorVisor. Additionally, we evaluate a bill generator which generates PDFs containing a set of purchased items formatted with a default template. Both benchmarks use mock data to simulate realistic workloads, with the hashing benchmark using 200×200 randomly generated images and the bill generation benchmark using 25 randomly generated item names and prices with an attached image.

Microbenchmarks. We evaluate a set of common microbenchmarks, including image processing workloads (e.g., Gaussian image blur), cryptography (e.g., password-based key derivation functions such as Scrypt and Pbkdf2), string compression (LZ4), histogram computation (Histogram), and string processing (e.g., stop word filtering and hashtag extraction). Our image processing, histogram, and cryptographic benchmarks operate on realistic, synthetic, and random inputs. We use the same parameters as Cisco type 8 passwords [20] for Pbkdf2 and Litecoin parameters for Scrypt [15]. To simulate realistic workloads, the compression and string benchmarks use as input a public dataset of tweets [41].

Baseline comparison. For our evaluation, we use two different baselines as points of comparison. First, for each of our benchmarks we compile them to WebAssembly (WASM), optimize them using `wasm-snip` and `wasm-opt` [17, 91], and execute them using Wasmtime [18] (a popular WASM JIT compiler). VectorVisor takes in the same WASM binary as an input. Second, for each of our benchmarks we compile and run them natively on an x86-64 CPU. This is the default choice for many developers who choose to run applications in the cloud, as most programs target x86-64. Each CPU benchmark is evaluated with multiple threads executing in parallel—proportional to the number of cores available.

Features	Scrypt	Pbkdf2	Blur Jpeg	Blur Bmp	PHash	PHash Modified	Bill PDF	Histogram	LZ4	Strings
Category	Crypto.	Crypto.	Image Proc.	Image Proc.	Image Proc.	Image Proc.	Misc.	Misc.	Misc.	Misc.
Reason for Inclusion	M	A	M & A	Alg.	Uses Indirect Calls	Alg.	D	A	A	D
Recursive or Indirect Code	×	×	✓	×	✓	×	✓	×	×	×
Unmodified Library Code	✓	✓	✓	✓	✓	×	✓*	✓	✓	✓
Batch Size (V520, T4, A10G)	2048, 4096, 6144	2048, 4096, 6144	1536, 3072, 4096	1536, 3072, 4096	1536, 3072, 4096	1536, 3072, 4096	1536, 3072, 4096	2048, 4096, 5120	1536, 3072, 4096	2048, 4096, 6144
Downloads	1.9M	12.4M	10.5M	10.5M	80k	0	10K	4.8M	298K	16K

Table 2: Details of evaluated benchmarks. We count benchmarks as containing recursive or indirect calls only if they execute those calls in the critical path of the application. All-Time crates.io download counts are as of 1/5/2023. 'M' and 'A' represent memory and arithmetically intensive benchmarks respectively. 'D' represents benchmarks with substantial divergence. 'Alg' represents benchmarks with significant algorithmic differences. *Bill-PDF uses a no-op system call as a barrier to mitigate heavy program divergence, but does not modify imported libraries.

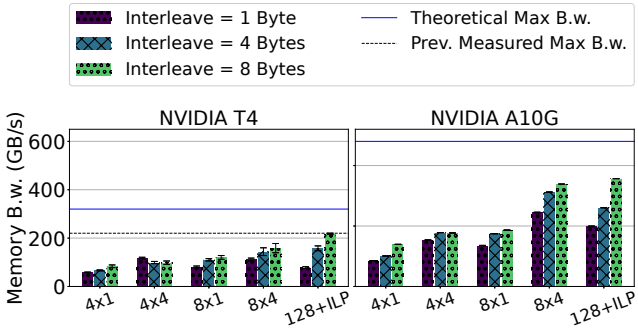

Figure 6: GPU `memcpy` bandwidth (Bytes copied×unroll #)

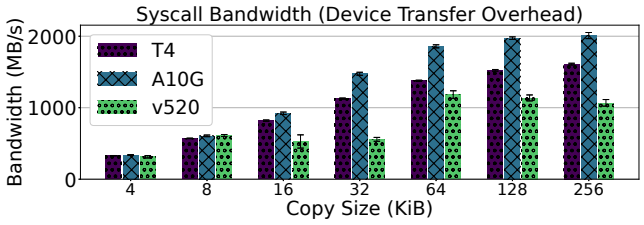

Figure 7: System call host-to-GPU PCIe transfer bandwidth.

4.2 System Performance

4.2.1 Copy Efficiency

Memory Bandwidth. To demonstrate that our memory interleaving can efficiently utilize the high memory bandwidth of GPUs, we evaluate five different `memcpy` implementations which vary copy size (bytes copied per-loop iteration) and loop unroll count. For each configuration we copy 1 MiB of data (using volatile memory accesses to bypass caching effects) from one array in memory to another non-aliased array. Each benchmark is run 50 times, with a heap size of 3 MiB with 4096 (on T4) or 6144 (on A10G) VMs running concurrently. Figure 6 shows that VectorVisor can achieve close to 100% of the experimentally derived maximum memory bandwidth of the T4 [61] and 74% of the theoretical memory bandwidth of the A10G [5]. We can see that larger interleaves, loop unrolling, and instruction level parallelism (ILP) [88] all have substantial impacts on memory bandwidth. VectorVisor leverages the `memory.copy` and `memory.fill` WASM intrinsics to insert optimized copy and fill functions into programs.

Syscall Performance. System calls provide a simple, familiar abstraction for developers to transfer inputs to and from a GPU. However, performing per-VM system calls incurs high data transfer overheads for smaller inputs. To evaluate our system call implementation, we copy inputs to and from the GPU, using batch sizes of 2048 (v520), 4096 (T4), and 6144 (A10G). Figure 7 shows the bandwidth for our VMM excluding network IO. Native CUDA transfer speeds peak at 6.3 GB/s for the T4 and 12.9 GB/s for the A10G—for a single large transfer. Despite high batching overheads, VectorVisor obtains ∼25% of the max possible bandwidth for fine-grained transfers of 256 KiB per-request using the T4. VectorVisor additionally supports overlapping data transfers with running GPU programs to avoid bottlenecks on VMM overhead.

4.2.2 Throughput

We evaluate VectorVisor's throughput against native x86-64 and WebAssembly baselines—with x86-64 being our primary baseline. WebAssembly numbers show the potential for heterogeneous deployments of VectorVisor in addition to showing the overhead of using WebAssembly. Figure 8 shows the best throughput for each configuration that we evaluated in terms of requests per second (RPS). AMD-specific issues, detailed in Section 4.1, prevent some configurations from being evaluated. Detailed throughput results for all system configurations can be found in Table 5 (Appendix).

VectorVisor outperforms x86 and WebAssembly for all but one benchmark (LZ4). We see high variation in throughput for each benchmark across our configurations. Device architecture (e.g., Turing vs. Ampere vs. RDNA 1), memory interleave size, backend compiler optimizations, and program characteristics have outsized impacts on performance. To evaluate the impact of program characteristics such as recursive and indirect function calls, we make use of profile-guided optimization (PGO) to remove indirect calls. Table 3 shows that we can remove all indirect calls from our benchmarks, reducing unoptimized function calls by up to 3430×. Removing all indirect calls and inlining functions where possible obtains mixed results. Bill-PDF improves in throughput by 1.3×, while other benchmarks are 10-20× *slower*. Function inlining caused by removing indirect calls can result in expensive register spilling. Counterintuitively, not removing indirect calls can improve throughput despite the context saving overhead.

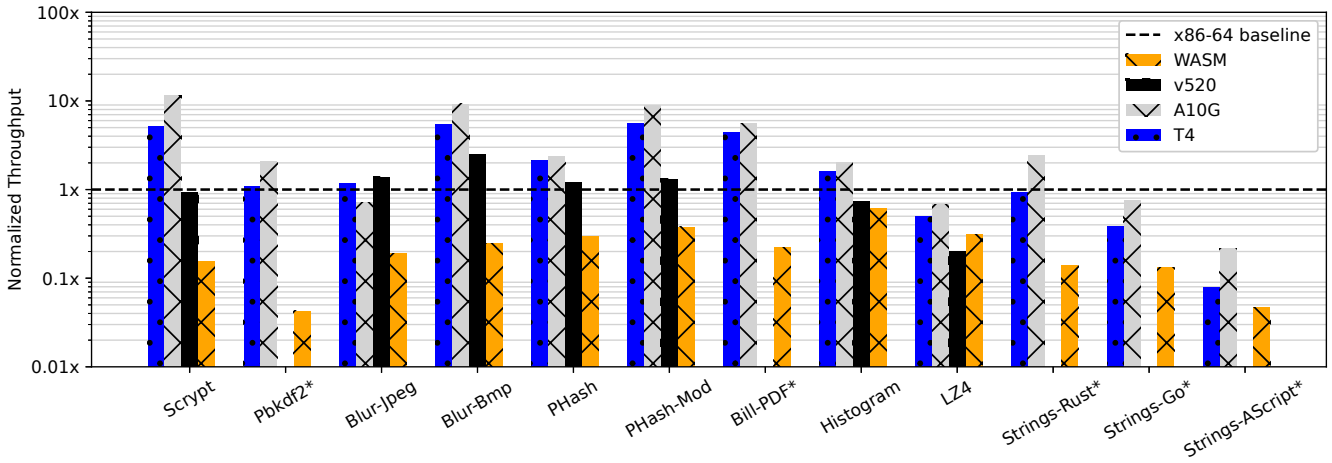

Figure 8: Normalized throughput (average RPS of each benchmark). Results are normalized to the x86-64 baseline for each benchmark except for Strings-Go and Strings-AScript, which are normalized to the x86-64 baseline of Strings-Rust instead. *Benchmarks without an AMD v520 result.

	Scrypt	Pbkdf2	Blur-Jpeg	Blur-Bmp	PHash	PHash-Modified	Bill-PDF	Histogram	LZ4	Strings	Strings-Go	Strings-AssemblyScript
# Total Slowcalls	52062	4828	4023	1416	11460	1410	285632	4117086	804807	43941	2172120	137212574
# Total Slowcalls w/PGO	206	1211	206	213	7844	206	2023	206	206	43744	2990690	143397289
# Indirect Calls	1	5228	162007	2	207031	1	211656	1	1	2	2869093	820142
# Indirect Calls w/PGO	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: *Profile-Guided Optimization Results.* Cumulative indirect and unoptimized call counts for 200 invocations of each instrumented WASM function. These benchmarks were run locally using a 16-core, 64 GB RAM machine running Ubuntu 18.04.

Complex runtimes such as Go and AssemblyScript have significantly higher overhead than Rust on x86-64 WASM baselines (on average 0.41× the throughput of the Rust baseline for Strings-Go using the T4). Runtime support for garbage collection, reflection, and compiler design choices in TinyGo/AssemblyScript all contribute to the observed overheads.

4.2.3 Throughput-per-dollar

Throughput as a metric is insufficient to evaluate VectorVisor. Improving throughput for data parallel workloads by allocating more resources (VMs) represents the status quo. Instead, we show that VectorVisor can achieve greater *efficiency*—improving throughput using fewer resources. Measuring efficiency requires normalizing performance across both CPUs and GPUs, which we accomplish using throughput-per-dollar. It is computed by dividing the requests-per-second (RPS) by the cost of each respective instance per-hour using on-demand pricing in `us-east-1`. On-demand prices are used as a conservative measure of the cost benefits of VectorVisor. Spot instance pricing can be cheaper, further improving the throughput-per-dollar of GPU (T4) instances vs CPU (Intel) instances by 1.49× (reported as of 1/5/2023).

Figure 9 shows the best throughput-per-dollar results for each configuration. Detailed throughput-per-dollar results for all system configurations can be found in Table 6 (Appendix). VectorVisor outperforms x86 instances for four benchmarks (Scrypt, Blur-Bmp, PHash-Modified, Bill-PDF), and on all but two benchmarks versus WebAssembly. Throughput-per-dollar results are overall lower than our throughput results in Section 4.2.2. Leveraging GPU acceleration requires substantial throughput improvements to offset the high cost of GPU hardware (e.g., 3.42× for the T4 vs. AMD x86-64 CPUs). Bottlenecks on application-level divergence (e.g., Strings) and data transfer overheads (e.g., LZ4 and Histogram) result in lower throughput and throughput-per-dollar results.

In three out of the four benchmarks where VectorVisor surpasses our x86-64 baselines, the T4 outperforms the A10G, even though it belongs to an earlier generation of GPUs (e.g., Turing vs. Ampere). Despite differences in GPU hardware, the best predictor of superior throughput-per-dollar with VectorVisor is the ratio of the global memory size (e.g., the number of VMs that can fit) to cost. Compared to the A10G and v520, the T4 packs 27.5% and 43% more VMs-per-dollar respectively. Workloads such as Scrypt, which leverage hardware differences like the larger memory bandwidth of the A10G, can break this trend.

4.2.4 Latency

VectorVisor runs many instances of a program in parallel, improving total throughput, but not latency. Batches of requests have higher on-device execution times than x86-64, ranging from 84-1040× longer using the T4—limiting usage to non-latency sensitive applications.

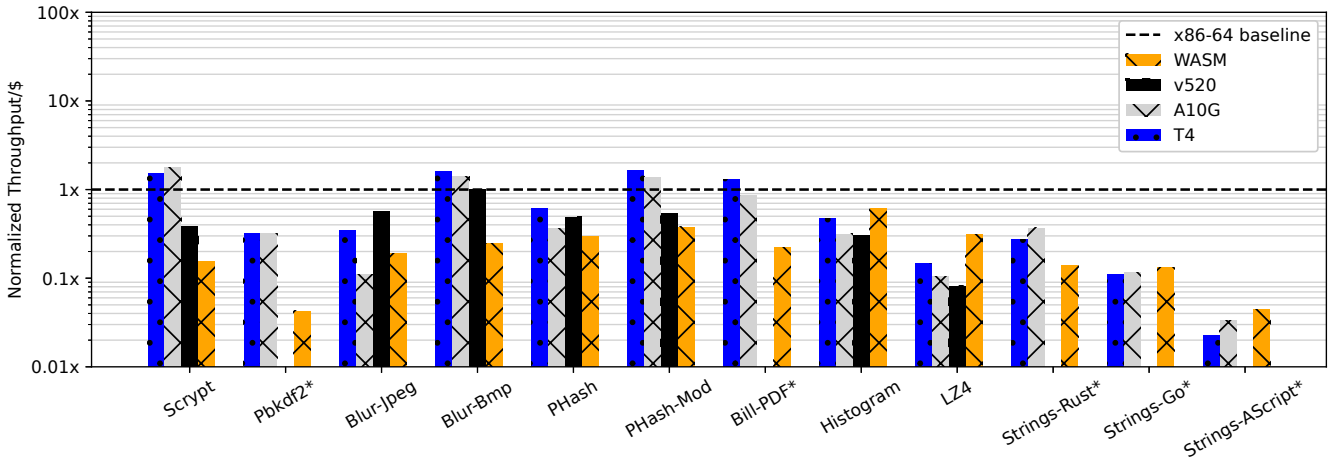

Figure 9: *Benchmark Throughput-per-Dollar.* Results are normalized to the x86-64 baseline for each benchmark except for Strings-Go and Strings-AScript, which are normalized to the x86-64 baseline of Strings-Rust instead. *Benchmarks without an AMD v520 result.

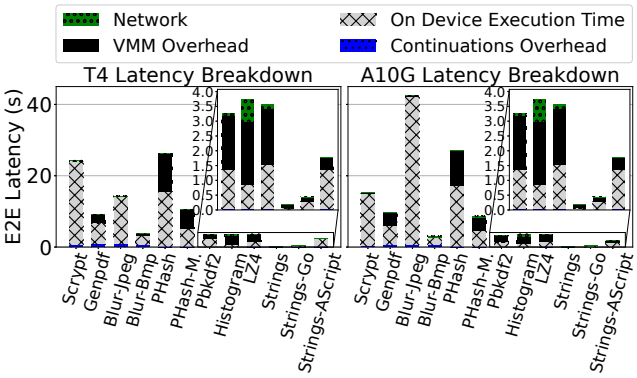

Figure 10: Per-benchmark latency breakdown of execution time, VMM overhead (e.g., syscall overhead), continuations overhead (e.g., context saving/restoring), and network IO. Breakdowns correspond to the best performing configurations with PGO disabled from Table 5.

4.3 Latency breakdown

Figure 10 shows the end-to-end (E2E) latency breakdown for each benchmark. Batch sizes, which impact request latency, can be found in Table 2. On-device execution time dominates the E2E latency for most benchmarks, with the histogram benchmark being the exception. We see that supporting preemption using continuations has low overhead, varying between <1% (PHash-Modified) and 19% (Blur-Bmp) of the on-device execution time. Similarly, by overlapping compute with VMM and network IO VectorVisor significantly reduces related overheads. Benchmarks with a low operational intensity (Ops/Byte) (e.g., Histogram, LZ4) which cannot overlap on-device execution time with batch formation as efficiently are more likely to bottleneck on VMM or network IO.

GPU	Platform	Instance Name	Benchmark	Throughput	Throughput/$
NVIDIA T4	VectorVisor	g4dn.xlarge	Blur-Bmp	804.83	1530.10
NVIDIA A10G	VectorVisor	g5.xlarge	Blur-Bmp	1365.84	1357.69
NVIDIA T4	VectorVisor	g4dn.xlarge	PHash-Modified	384.32	730.65
NVIDIA A10G	VectorVisor	g5.xlarge	PHash-Modified	608.02	604.40
NVIDIA T4	CUDA	g4dn.xlarge	Blur-Bmp	576.28	1095.59
NVIDIA T4	CUDA	g4dn.2xlarge	Blur-Bmp	1118.95	1487.96
NVIDIA A10G	CUDA	g5.xlarge	Blur-Bmp	652.96	649.06
NVIDIA A10G	CUDA	g5.2xlarge	Blur-Bmp	1250.33	1031.62
NVIDIA T4	CUDA	g4dn.xlarge	PHash-Modified	408.85	777.27
NVIDIA T4	CUDA	g4dn.2xlarge	PHash-Modified	821.15	1091.95
NVIDIA A10G	CUDA	g5.xlarge	PHash-Modified	462.27	459.52
NVIDIA A10G	CUDA	g5.2xlarge	PHash-Modified	896.35	739.56

Table 4: Performance of handwritten CUDA benchmarks.

4.3.1 CUDA Comparison

Leveraging GPU acceleration typically involves manually breaking down a program into fine-grained tasks which can be parallelized—speeding up individual invocations of a function. In contrast, VectorVisor runs many instances of the same program in parallel, improving throughput but not latency. To evaluate the efficiency of our translation, we manually rewrote two benchmarks (Blur-Bmp and PHash-Modified) using CUDA. CUDA baselines incur additional CPU overhead from increased kernel launch overheads and running a fraction of the workload on the CPU. To fairly evaluate these baselines, we benchmark them using both xlarge and 2xlarge instances with additional CPUs (Table 1).

We see in Table 4 that VectorVisor slightly outperforms a handwritten CUDA Gaussian blur function, and obtains 67% of the throughput-per-dollar of our CUDA PHash function. PHash-Modified has higher VMM overhead than Blur-Bmp (35% vs 12% of the E2E latency) which affects overall efficiency.

5 Discussion

Workload Characterization. Identifying ideal workloads for VectorVisor is key to improving the cost efficiency of real

applications. Ideal workloads minimize divergent execution, recursion, indirect calls, and are compute-bound. Future work can incorporate model-based approaches [79] to identifying acceleration opportunities for VectorVisor.

Evaluation Limitations. We use both throughput and throughput-per-dollar as evaluation metrics. Throughput-per-dollar is a powerful metric that enables us to compare the end-to-end *efficiency* of VectorVisor, which considers system complexities as well as the capital and operational cost implications of running throughput-oriented workloads. Cloud providers allow customers to insure themselves against high variation in hardware pricing [62, 63], providing a steady baseline cost (at a premium). On-premises hardware configurations can be less expensive over long periods of time, for those willing to pay higher up-front costs. Despite shortcomings, cost-based efficiency metrics provide tangible baselines.

System Call Implementations. Providing support for system calls using continuations was key to running realistic workloads using VectorVisor. Systems such as GPUfs, GPUnet, and Berkeley Borph [76–78, 82, 86] instead provide support using a more performant RPC-like interface using vendor-specific APIs or custom drivers. RPC-style interfaces rely on the ability to perform concurrent and consistent CPU–GPU memory accesses. OpenCL 2.0 *in theory* enables this with fine-grained buffer SVM [3, 53]. In practice, support for fine-grained SVM is mixed—with NVIDIA OpenCL 3.0 not supporting the API and AMD providing partial support [2]. Continuations provide a cross-platform and reasonably performant approach to supporting system call support for GPUs.

6 Related Work

Continuations. Continuations are often used by compilers to support complex control flow operations such as exceptions and preemption [27, 28, 30, 71, 84]. VectorVisor uses continuations to efficiently provide support for preemption and complex control flow on GPUs.

GPU Preemption. GPU kernel preemption can be supported through compiler-based approaches that partition (or slice) programs into chunks [29, 35, 39, 89, 93, 94], or with hardware/driver support [13, 68, 83, 86].

High-Level GPU Languages. CUDA or OpenCL require developers to write programs using low level abstractions. High level language approaches [2, 31, 44, 48, 52, 55, 57, 72, 92] make it easier to accelerate existing programs by reusing existing codebases. Common language features such as dynamic memory allocation, garbage collection, reflection, and recursion are often absent. Unlike VectorVisor, code often must be rewritten to explicitly leverage parallel APIs.

Domain-Specific GPU Systems and Languages. Programming languages designed for domain-specific workloads (DSLs) [24, 38, 40, 46, 58, 73–75, 80] can offer substantially improved performance over general-purpose programming languages. DSLs obtain superior performance through language restrictions, forcing developers to express programs using specific syntax or function calls. While DSLs can efficiently accelerate specific workloads, they trade off performance for programmability—e.g., many workloads cannot be expressed using restrictive DSLs. Similar to DSLs, domain-specific systems can significantly improve performance for throughput-oriented workloads [32, 42, 56, 87]. Domain-specific systems vectorize common workloads (e.g., image processing, machine learning, database operations) using handwritten GPU kernels. Other systems manually vectorize functions from (non GPU-specific) DSLs (i.e. SQL) [50, 51, 69].

Vectorized Program Translation. Systems that abstract a SIMT or SIMD lane as a VM often target restricted use-cases (e.g., fuzz testing) [37, 45, 47, 49]. VectorVisor's design and implementation notably differ from prior work, offering superior GPU language, runtime, and hardware support.

7 Conclusion

VectorVisor is a research prototype which demonstrates that applications originally written for CPUs can be directly run on GPUs without significant modifications. Not only is such GPU execution possible, but it can in fact yield superior throughput-per-dollar versus compute-optimized x86-64 CPUs in the cloud.

Binary translation for GPUs is an exciting and predominantly unexplored area of research, with many potential applications. VectorVisor shows the viability of our new approach to parallelism, opening up the area to future research.

8 Acknowledgements

We would like to thank our shepherd, Redha Gouicem, and the anonymous reviewers for helping us improve this paper. We also thank Rachit Nigam, Mieszko Lis, and Devon Loehr for their valuable comments on earlier versions of it.

References

[1] NVIDIA OpenCL Best Practices Guide. https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, 2009.

[2] Aparapi. https://code.google.com/archive/p/aparapi/, 2011.

[3] OpenCL™ 2.0 Shared Virtual Memory Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html, 2014.

[4] AMD OpenCL Programming Optimization Guide. https://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf, 2015.

[2] Fine-grained SVM support is present, but not SVM Atomics.

[5] NVIDIA A10G Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a10/pdf/a10-datasheet.pdf, 2019.

[6] Open Sourcing Photo and Video-Matching Technology to Make the Internet Safer. https://about.fb.com/news/2019/08/open-source-photo-video-matching/, 2019.

[7] Threatexchange. https://github.com/facebook/ThreatExchange, 2019.

[8] AMD EPYC 7002 Processor Datasheet. https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf, 2021.

[9] Blockhash. https://web.archive.org/web/20210827144701/http://blockhash.io/, 2021.

[10] crates.io. https://crates.io/, 2021.

[11] CUDA. https://developer.nvidia.com/cuda-toolkit, 2021.

[12] CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2021.

[13] cuda-gdb. https://docs.nvidia.com/cuda/pdf/cuda-gdb.pdf, 2021.

[14] CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html, 2021.

[15] Scrypt. https://litecoin.info/index.php/Scrypt, 2021.

[16] The OpenCL C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html, 2021.

[17] wasm-snip. https://github.com/rustwasm/wasm-snip, 2021.

[18] Wasmtime. https://github.com/bytecodealliance/wasmtime, 2021.

[19] AssemblyScript. https://www.assemblyscript.org/, 2022.

[20] Cisco Password Types: Best Practices. https://media.defense.gov/2022/Feb/17/2002940795/-1/-1/1/CSI_CISCO_PASSWORD_TYPES_BEST_PRACTICES_20220217.PDF, 2022.

[21] Radeon Pro v520 ROCm Support. https://github.com/RadeonOpenCompute/ROCm/issues/1706, 2022.

[22] TinyGo. https://github.com/tinygo-org/tinygo, 2022.

[23] ROCm Installation Guide v5.0. https://docs.amd.com/bundle/ROCm_Installation_Guidev5.0/page/Prerequisite_Actions.html, 2023.

[24] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[25] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 1996.

[26] Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. Tighten Rust's Belt: Shrinking Embedded Rust Binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2022.

[27] Henry G Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM Sigplan Notices*, 1995.

[28] Joel F Bartlett. SCHEME-> C a portable Scheme-to-C compiler. In *WRL Research Report 89/1*, 1989.

[29] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[30] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in all the stops: Execution control for JavaScript. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[31] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2018.

[32] Nils Boeschen and Carsten Binnig. GaccO-A GPU-accelerated OLTP DBMS. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, 2022.

[33] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2012.

[34] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 1994.

[35] Jon Calhoun and Hai Jiang. Preemption of a CUDA Kernel Function. In *13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2012.

[36] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[37] Ahmet Celik, Pengyu Nie, Christopher J Rossbach, and Milos Gligoric. Design, Implementation, and Application of GPU-Based Java Bytecode Interpreters. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2019.

[38] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP)*, 2011.

[39] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

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

[41] Zhiyuan Cheng, James Caverlee, and Kyumin Lee. You are where you tweet: a content-based approach to geo-locating Twitter users. In *19th ACM International Conference on Information and Knowledge Management (CIKM)*, 2010.

[42] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[43] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[44] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F Bacon, and Stephen J Fink. Compiling a High-Level Language for GPUs: (via language support for architectures and compilers). In *33th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[45] Ryan Eberhardt, Artem Dinaburg, and Peter Goodman. Let's build a high-performance fuzzer with GPUs! https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/, 2020.

[46] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: DSL for linear algebra and neural net computations on GPUs. In *2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2018.

[47] Brandon Falk. Vectorized Emulation: Hardware accelerated taint tracking at 2 trillion instructions per second. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html, 2018.

[48] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time GPU compilation for interpreted languages with partial evaluation. In *13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2017.

[49] Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. Running parallel bytecode interpreters on heterogeneous hardware. In *4th International Conference on Art, Science, and Engineering of Programming (Programming)*, 2020.

[50] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, 2018.

[51] Henning Funke and Jens Teubner. Data-parallel query processing on non-uniform data. *Proceedings of the VLDB Endowment*, 2020.

[52] Kate Gregory and Ade Miller. *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press, 2012.

[53] Khronos OpenCL Working Group. The OpenCL Specification Version: 2.1 Document Revision: 24. https://registry.khronos.org/OpenCL/specs/opencl-2.1.pdf#page=174, 2018.

[54] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up

to speed with WebAssembly. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[55] Michael Haidl and Sergei Gorlatch. PACXX: Towards a unified programming model for programming accelerators using C++ 14. In *LLVM Compiler Infrastructure in HPC*, 2014.

[56] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. MemcachedGPU: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[57] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D Matsakis. GPU programming in Rust: Implementing high-level abstractions in a systems-level language. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, 2013.

[58] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[59] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2000.

[60] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. Accelerating SSL with GPUs. *ACM SIGCOMM Computer Communication Review*, 2010.

[61] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[62] Michael Kan. Read it and weep: Here's how bad Nvidia GPU prices got in a single year. https://www.pcmag.com/news/read-it-and-weep-heres-how-bad-nvidia-gpu-prices-got-in-a-single-year, 2021.

[63] Michael Kan. Corsair: Expect some GPU prices to dip 'below msrp' soon as prices normalize. https://www.pcmag.com/news/corsair-expect-some-gpu-prices-to-dip-below-msrp-soon-as-prices-normalize, 2022.

[64] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *In International Symposium on Code Generation and Optimization (CGO)*, 2004.

[65] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2008.

[66] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the AMD EPYC™ and Ryzen™ processor families : Industrial product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[67] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Pipelined parallel LZSS for streaming data compression on GPGPUs. In *IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[68] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015.

[69] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proceedings of the VLDB Endowment*, 2020.

[70] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle inventor: data movement synthesis for gpu kernels. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[71] Donald Pinckney, Arjun Guha, and Yuriy Brun. Wasm/k: delimited continuations for WebAssembly. In *16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*, 2020.

[72] Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. Rootbeer: Seamlessly using GPUs from Java. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[73] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image

processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[74] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *23th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2011.

[75] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2013.

[76] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[77] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)*, 2016.

[78] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[79] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[80] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013.

[81] Gerald Jay Sussman and Guy L Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 1998.

[82] Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. GLoop: an event-driven runtime for consolidating GPGPU applications. In *8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[83] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2014.

[84] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1992.

[85] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in Rust. *Proceedings of the ICSE-SEIP*, 2022.

[86] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. Generic system calls for GPUs. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[87] Matthias Vogelgesang, Suren Chilingaryan, Tomy dos Santos Rolo, and Andreas Kopmann. UFO: A scalable GPU-based image processing framework for on-line monitoring. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*, 2012.

[88] Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference, GTC*, 2010.

[89] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. FLEP: Enabling flexible and efficient preemption on GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017.

[90] Bian Yang, Fan Gu, and Xiamu Niu. Block mean value based image perceptual hashing. In *IEEE International Conference on Intelligent Information Hiding and Multimedia*, 2006.

[91] Alon Zakai. Binaryen. https://github.com/WebAssembly/binaryen, 2022.

[92] Wojciech Zaremba, Yuan Lin, and Vinod Grover. JaBEE: framework for object-oriented Java bytecode compilation and execution on graphics processor units. In *5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, 2012.

[93] Lior Zeno, Avi Mendelson, and Mark Silberstein. GPUpIO: The case for I/O-driven preemption on GPUs. In *9th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, 2016.

[94] Husheng Zhou, Guangmo Tong, and Cong Liu. GPES: A preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

A Appendix

A.1 Tables

System	Platform	PGO	Interleave	Scrypt	Pbkdf2	Blur Jpeg	Blur Bmp	PHash	PHash Mod.	Bill PDF	Histogram	LZ4	Strings (Rust / Go / AScript)
VectorVisor	AMD v520	Y	4	32.27	N/A	202.24	368.64	70.28	77.85	N/A	832.43	449.89	N/A / N/A / N/A
VectorVisor	AMD v520	Y	8	29.29	N/A	**245.76**	N/A	79.67	89.86	N/A	848.31	N/A	N/A / N/A / N/A
VectorVisor	NVIDIA T4	N	4	109.32	71.09	209.94	726.59	129.18	341.73	339.06	1570.90	818.90	9535.47 / 4112.16 / 861.24
VectorVisor	NVIDIA T4	N	8	160.25	1355.05	177.07	804.83	140.44	367.31	380.55	1730.05	1114.30	10242.08 / 3735.93 / 826.29
VectorVisor	NVIDIA T4	Y	4	108.51	63.68	50.66	720.79	10.44	349.98	398.26	1827.91	371.75	9077.32 / 4204.24 / 845.07
VectorVisor	NVIDIA T4	Y	8	179.54	59.49	60.59	728.86	10.90	384.32	497.42	1676.06	486.94	10332.82 / 3842.04 / 841.04
VectorVisor	NVIDIA A10G	N	4	297.29	**2596.52**	93.30	1237.87	153.99	546.35	480.29	886.25	1527.21	24543.24 / **8438.01** / 1781.76
VectorVisor	NVIDIA A10G	N	8	389.08	168.56	69.52	**1365.84**	**158.99**	592.87	509.71	484.22	1490.67	25981.84 / 8030.55 / 1965.93
VectorVisor	NVIDIA A10G	Y	4	297.08	145.66	71.73	1166.44	14.81	533.85	308.62	**2356.72**	722.01	24109.98 / 8377.06 / **2398.01**
VectorVisor	NVIDIA A10G	Y	8	**397.10**	143.55	127.47	1163.57	20.87	**608.02**	**619.04**	517.97	945.06	**26598.05** / 8302.31 / 1977.61
CPU (x86-64)	AMD	N/A	N/A	33.89	1233.44	176.63	147.14	66.05	68.02	111.18	1140.20	**2235.93**	11002.53 / N/A / N/A
CPU (x86-64)	Intel	N/A	N/A	34.27	149.33	148.53	135.43	55.83	58.00	85.95	1144.96	1987.63	10182.98 / N/A / N/A
CPU (WASM)	AMD	N/A	N/A	5.33	52.67	33.81	36.13	19.86	25.78	24.97	697.94	700.77	1536.22 / 1450.18 / 485.62
CPU (WASM)	Intel	N/A	N/A	4.35	46.49	25.83	17.33	12.13	13.70	21.44	659.40	596.05	1431.18 / 1375.47 / 514.84

Table 5: Average requests per second (RPS) of each benchmark. Bold values correspond to the best throughput.

System	Platform	PGO	Interleave	Scrypt	Pbkdf2	Blur Jpeg	Blur Bmp	PHash	PHash Mod.	Bill PDF	Histogram	LZ4	Strings (Rust / Go / AScript)
VectorVisor	AMD v520	Y	4	85.27	N/A	534.32	973.95	185.68	205.69	N/A	2199.28	1188.61	N/A / N/A / N/A
VectorVisor	AMD v520	Y	8	77.38	N/A	649.30	N/A	210.49	237.41	N/A	2241.24	N/A	N/A / N/A / N/A
VectorVisor	NVIDIA T4	N	4	207.84	135.16	399.13	1381.34	245.58	649.67	644.60	2986.51	1556.85	18128.26 / 7817.80 / 1637.33
VectorVisor	NVIDIA T4	N	8	304.67	2576.15	336.63	**1530.10**	266.99	698.31	723.48	3289.07	2118.43	19471.63 / 7102.54 / 1570.90
VectorVisor	NVIDIA T4	Y	4	206.30	121.06	96.31	1370.32	19.85	665.35	757.15	3475.12	706.74	17257.26 / 7992.86 / 1606.60
VectorVisor	NVIDIA T4	Y	8	341.32	113.11	115.20	1385.66	20.73	**730.65**	**945.67**	3186.43	925.75	19644.15 / 7304.26 / 1598.94
VectorVisor	NVIDIA A10G	N	4	295.52	2581.03	92.74	1230.48	153.07	543.09	477.42	880.96	1518.10	24396.86 / 8387.68 / 1771.13
VectorVisor	NVIDIA A10G	N	8	386.76	167.56	69.11	1357.69	158.04	589.33	506.67	481.33	1481.78	25826.88 / 7982.65 / 1954.21
VectorVisor	NVIDIA A10G	Y	4	295.31	144.79	71.30	1159.48	14.72	530.66	306.78	2342.66	717.70	23966.18 / 8327.10 / 2383.71
VectorVisor	NVIDIA A10G	Y	8	**394.73**	142.69	126.71	1156.63	20.74	604.40	615.35	514.88	939.43	26439.41 / 8252.79 / 1965.81
CPU (x86-64)	AMD	N/A	N/A	220.08	**8009.37**	**1146.95**	955.45	**428.87**	441.67	721.94	**7403.91**	**14519.04**	**71444.98** / N/A / N/A
CPU (x86-64)	Intel	N/A	N/A	201.60	878.40	873.70	796.63	328.39	341.18	505.61	6735.05	11691.97	59899.89 / N/A / N/A
CPU (WASM)	AMD	N/A	N/A	34.60	342.03	219.52	234.62	128.97	167.39	162.16	4532.09	4550.45	9975.48 / **9416.76** / **3153.38**
CPU (WASM)	Intel	N/A	N/A	25.58	273.47	151.95	101.95	71.34	80.58	126.13	3878.82	3506.18	8418.72 / 8091.03 / 3028.45

Table 6: *Benchmark Throughput-per-Dollar.* Values correspond to the average RPS of each benchmark normalized to instance cost. Bold values correspond to the best throughput-per-dollar.

A.2 Rust Example

```rust
#[macro_use]
extern crate lazy_static;
// Import existing open-source libraries!
use pdf_writer::*;
use pdf_writer::types::{ActionType, AnnotationType, BorderType};
use std::fs::File;
use std::io::Write;
use std::time::Instant;
// Import our custom 'serverless' runtime. We use this in our x86 and WASM benchmarks as well.
// WASM benchmarks run the same binary that VectorVisor does!
use wasm_serverless_invoke::wasm_handler::*;
use wasm_serverless_invoke::wasm_handler::WasmHandler;
use wasm_serverless_invoke::wasm_handler::SerializationFormat::MsgPack;
use serde::Deserialize;
use serde::Serialize;
// Image and compression libraries
use image::{ColorType, GenericImageView, ImageFormat};
use miniz_oxide::deflate::{compress_to_vec_zlib, CompressionLevel};
// Include a sample template image for our PDF footer
lazy_static! {
    static ref EMBED_IMAGE: &'static [u8] = include_bytes!("test.png");
}
// Syntactic sugar for (de)serializing JSON/MsgPack inputs
#[derive(Debug, Deserialize)]
struct FuncInput {
    name: String,
    purchases: Vec<String>,
    price: Vec<f64>, // Typically prices should not be encoded as floats, we do this for simplicity.
}
#[derive(Debug, Deserialize)]
struct BatchInput {
    inputs: Vec<FuncInput>
}
#[derive(Debug, Serialize)]
struct FuncResponse {
    resp: Vec<u8>
}
#[derive(Debug, Serialize)]
struct BatchFuncResponse {
    resp: Vec<FuncResponse>
}
#[inline(never)]
fn makePdf(event: FuncInput) -> Vec<u8> {
    // Perform PDF formatting, image manipulation, and compression to generate a valid PDF
}
fn batch_genpdf(inputs: BatchInput) -> BatchFuncResponse {
    let mut results = vec![];
    for input in inputs.inputs {
        results.push(FuncResponse { resp: makePdf(input) });
        // Bill-PDF is the only benchmark to use system calls as synchronization barriers
        unsafe { vectorvisor_barrier() }; // We can wait on arbitrary subsets of VMs (unlike OpenCL barrier(...))
    }
    return BatchFuncResponse{ resp: results };
}
fn main() {
    // Specify input format type and buffer sizes
    let handler = WasmHandler::new(&batch_genpdf);
    // Starts the event-loop and encapsulates serverless_invoke/serverless_response
    handler.run_with_format(1024*512, MsgPack);
}
```

Figure 11: *Bill-PDF*. This benchmark performs PDF processing, image manipulation, and compression.

A.3 Golang Example

```go
package main;

// define our system call interface
// #include "serverless.c"
import "C"
import (
    // Import JSON + string manipulation libraries
    "github.com/json-iterator/tinygo"
    "unsafe"
    "strings"
)

//go:generate go run github.com/json-iterator/tinygo/gen
type Payload struct {
    Tweets  []string `json:"tweets"`
}
//go:generate go run github.com/json-iterator/tinygo/gen
type Response struct {
    Tokenized  [][]string
    Hashtags   [][]string
}
// Go doesn't provide Map/Filter for us, so we use our own implementation
func Map[T, U any](ts []T, f func(T) U) []U {
    ...
}
func Filter(vs []string, f func(string) bool) []string {
    ...
}
func main() {
    json := jsoniter.CreateJsonAdapter(Payload_json{}, Response_json{})
    // Use this as a set, track all stopwords
    stopwordSet := make(map[string]bool)
    for _, word := range stopWords {
        stopwordSet[word] = true
    }
    input_buf := make([]byte, 1024*450) // buffer for raw inputs from VectorVisor
    for { // serverless_invoke is the system call used for transferring inputs from the host (CPU) to the GPU
        in_size := C.serverless_invoke((*C.char)(unsafe.Pointer(&input_buf[0])), 1024*450)
        if in_size == 0 { // if in_size == 0, then this VM is blocked off and has no input for this batch
            fakeaddr := uintptr(0x0) // serverless_response copies inputs from the GPU back to the CPU.
            C.serverless_response((*C.char)(unsafe.Pointer(fakeaddr)), 0)
            continue
        }
        var input Payload;
        json.Unmarshal(input_buf[0:in_size], &input);
        // First tokenize each tweet []string --> [][]string
        ...
        // Now process each tweet, filtering out stop words
        ...
        // Get the hashtags, we will add them as we see them
        var tags = make([][]string, 0)
        ...
        var response Response;  // create a JSON response and return it!
        response.Tokenized = tokenized;
        response.Hashtags = tags;
        bytes, _ := json.Marshal(response);
        C.serverless_response((*C.char)(unsafe.Pointer(&bytes[0])), (C.uint)(len(bytes)))
    }
}
```

Figure 12: *Strings-Go*. Tokenize some input tweets and return the hashtags. TinyGo (https://tinygo.org/docs/reference/lang-support/) provides us with a conservative mark and sweep garbage collector, limited runtime reflection and goroutine support.

A.4 AssemblyScript Example

```
import { Console, FileSystem, Descriptor } from "as-wasi/assembly"; // Import needed syscalls
import { JSON, JSONEncoder } from "assemblyscript-json/assembly"; // Import a JSON encoder/decoder
import { listen } from "./env"; // Import our event-driven runtime
import { stopWords, initSet, getSet } from "./stop"; // Import a dataset of stopwords
function abort(message: usize, fileName: usize, line: u32, column: u32): void {
    unreachable() // needed for the AssemblyScript runtime
}
initSet(); // init our set of "stop words"
let set: Set<string> = getSet();

// TypeScript-like syntax for GPU programing!
function process_tweets(input: JSON.Obj): Uint8Array | null {
    let tweets: JSON.Arr | null = input.getArr("tweets");
    if (tweets != null) {
        let strTweets: string[] = tweets._arr.map<string>((val: JSON.Value): string => val.toString());
        // Split each tweet (tokenize)
        let tokenize: string[][] = strTweets.map<string[]>((val: string): string[] => val.split("␣"));
        // Remove empty values and stop words
        let filtered: string[][] = tokenize.map<string[]>((arr: string[]): string[] =>
                                    arr.filter((word: string): bool => {
                                            if (set.has(word)) {
                                                return false;
                                            } else {
                                                return true;
                                            }
                                        }));
        // Get the array of hashtags for each tweet
        let hashtags: string[][] = filtered.map<string[]>((tweet: string[]): string[] =>
                                    tweet.filter((word: string): bool => {
                                        if (word.charAt(0) == '#' && word.charAt(1) != "") {
                                            return true;
                                        } else {
                                            return false;
                                        }
                                    }));
        let encoder = new JSONEncoder(); // encode a JSON response
        encoder.pushArray("tokenized");
        for (let tweet_idx = 0; tweet_idx < filtered.length; tweet_idx++) {
            ...
        }
        encoder.popArray();
        encoder.pushArray("hashtags");
        for (let tweet_idx = 0; tweet_idx < hashtags.length; tweet_idx++) {
            ...
        }
        encoder.popArray();
        let json: Uint8Array = encoder.serialize();
        return json;
    }
    // else we failed somehow...
    return null;
}
listen(1024*512, process_tweets); // Starts the event-loop and encapsulates serverless_invoke/serverless_response
```

Figure 13: *Strings-AssemblyScript*. Same as Strings+Strings-Go, but with different syntax. Support for incremental garbage collection is provided.