Proteus: Portable Runtime Optimization of GPU Kernel Execution with Just-in-Time Compilation

Giorgis Georgakoudis
Lawrence Livermore National Laboratory
Livermore, USA
georgakoudis1@llnl.gov

Konstantinos Parasyris
Lawrence Livermore National Laboratory
Livermore, USA
parasyris1@llnl.gov

David Beckingsale
Lawrence Livermore National Laboratory
Livermore, USA
beckingsale1@llnl.gov

Abstract

In High-performance computing (HPC) fast application execution is the primary objective. HPC software is written in high-performance languages (C/C++, Fortran) and is statically compiled Ahead-of-Time (AOT) using optimizing compilers to generate fast code. AOT compilation optimizes source code with only limited information available at compile time, which precludes possible optimization leveraging runtime information.

We propose Proteus, an easy-to-use, portable, and lightweight Just-In-Time (JIT) compilation approach to optimize GPU kernels at runtime. Proteus dynamically extracts, compiles, and optimizes language-agnostic LLVM IR to reduce compilation overhead while enhancing portability and versatility compared to language-specific solutions. Using a minimally intrusive annotation-based interface, Proteus specializes GPU kernels for input arguments and launch parameters. Evaluation on a diverse set of programs on AMD and NVIDIA GPUs shows that Proteus achieves significant end-to-end speedup, up to 2.8× for AMD and 1.78× on NVIDIA, over AOT optimization, while outperforming CUDA-specific Jitify with an average 1.23× speedup, thanks to reduced overhead and faster binary code in certain cases.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Runtime environments**.

Keywords: Runtime optimization, GPU programming

ACM Reference Format:
Giorgis Georgakoudis, Konstantinos Parasyris, and David Beckingsale. 2025. Proteus: Portable Runtime Optimization of GPU Kernel Execution with Just-in-Time Compilation. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708939

CGO '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708939

1 Introduction

Heterogeneous computing systems comprised of CPUs and GPUs are the norm in HPC and datacenters. They leverage GPU execution to accelerate computation with power efficiency for a wide range of applications, from scientific simulations to machine learning. This trend is apparent in Supercomputing systems, where 9 out of the 10 fastest supercomputers in the world equip GPUs [2]. Nonetheless, those systems deploy diverse GPU architectures from different vendors with different software environments, which makes software portability and its optimization a first-order concern. For example, the Frontier supercomputer comes with AMD MI250X GPUs, the Aurora supercomputer features Intel Max GPUs, and Microsoft's Eagle cluster in the Azure cloud service utilizes NVIDIA H100 GPUs.

Software development in HPC is primarily done using statically compiled languages such as C/C++ and Fortran. Applications are typically compiled ahead-of-time (AOT) – prior to execution – with an optimizing compiler to generate fast machine code. Optimizing compilers, such as LLVM, implement an aggressive optimization pipeline that runs extensive static analysis to optimize generated code based on code structure and information available or inferred at compile time. However, AOT compiler analysis and optimization is limited by the information inferred at compile time, for example unknown values of program variables inhibit possible optimization.

Just-In-Time (JIT) compilation defers compilation until runtime, lifting such limitations to analysis and optimization by having access to runtime information, at the cost of runtime overhead. Interpreted languages, like Java, Python, and Javascript, use JIT optimization to mitigate the significant overhead of interpretation and achieve higher performance. Additionally, JIT compilation for these languages has been shown to optimize code further than simply removing interpreting overheads by *specializing* code. A central optimization is *runtime value specialization*, where a code region is specialized and optimized for a specific set of runtime values of program variables.

Applying similar JIT optimization on statically compiled languages is challenging because of their limited introspection and because it is hard to mitigate the overhead of dynamic compilation, even if dynamically optimized code is faster. Interestingly, recent work [7, 16, 25–27, 39] has shown promise for JIT optimization targeting C/C++ code, by extending compiler technology, as in Clang/LLVM [7, 16, 25–27], or providing vendor-specific runtime compilation interfaces [39]. While inspirational, those approaches fall short because of lacking GPU support [7] or being non-portable and language dependent [16, 39]. Furthermore, the interface they present to programmers for enabling JIT optimization is convoluted and hard-to-use. They require significant code refactoring to retrofit C++ functors/templates/lambdas [7, 16, 25–27] for runtime value specialization, or worse require source code as a C-string for runtime compilation [39].

In this paper, we propose a new approach, named Proteus[1] for easy-to-use, portable, optimizing JIT compilation targeting GPU kernels in statically compiled languages with minimal overhead. Proteus consists of: (1) minimally-intrusive source code annotations to specify JIT compiled regions and optimization possibilities, (2) extensions in AOT compilation to extract annotated code regions and runtime information, and (3) a JIT compilation runtime library leveraging LLVM [33] to dynamically generate highly-optimized, specialized code using extracted runtime information, crucially including specialization-based code caching to minimize overhead.

The design of Proteus is portable, implemented for both CUDA and HIP codes, targeting NVIDIA and AMD GPUs. Our evaluation on both these architectures shows significant end-to-end speedup over AOT compiled code, recuperating overheads, and also outperforming the similar but non-portable NVIDIA's Jitify solution. We summarize the paper contributions as following:

- We present Proteus, a novel, easy-to-use, portable, low-overhead JIT optimizing compilation approach for GPU kernels that targets CUDA and HIP programs, based on LLVM. Proteus includes (1) novel extensions in AOT compilation for code and runtime value extraction, (2) key techniques for runtime JIT compilation with specialization, and (3) specialization-based caching to minimize overhead.
- We present two specialization optimizations portably enabled by our JIT compiler, namely runtime constant folding of kernel arguments and dynamic setting of kernel launch bounds, to generate highly optimized code at runtime.
- We provide our implementation open-source [1] to the community to extend, experiment, and study further.

[1]In Greek mythology, Proteus is an elusive, shape-shifting sea deity with prophetic powers that answers only when captured. In analogy, our Just-In-Time optimization approach captures the elusive, changing runtime information during execution to dynamically optimize code.

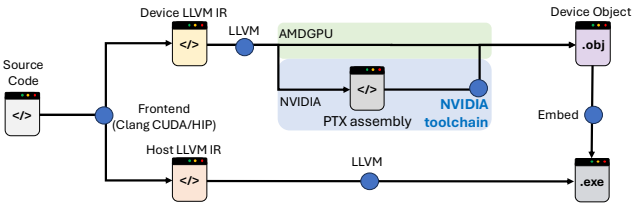


Figure 1. Split compilation for CUDA/HIP codes

- We perform an extensive evaluation on performance and overheads of Proteus, on several benchmark GPU programs from different application domains on an AMD MI250X and an NVIDIA V100 GPU. Results show that Proteus significantly speeds up end-to-end execution time compared to AOT compilation, by up to 2.8× for AMD and 1.78× for NVIDIA, fully mitigating JIT-induced overheads.
- We also compare Proteus with the non-portable NVIDIA Jitify solution on the NVIDIA GPU. Our approach consistently outperforms it by achieving 1.23× higher end-to-end speedup on average.

We structure the paper as follows. Section 2 provides background information. Section 3 discusses in detail the design and implementation of Proteus. Section 4 evaluates our approach. Section 5 reviews related literature, and finally Section 6 concludes the paper.

2 Background: GPU Programming and Compilation

CUDA and HIP extend C/C++ for GPU programming, allowing programmers to define GPU *kernels* – functions executed on the GPU device – within host-executed source code. Programmers *launch* device kernels using CUDA/HIP runtime interfaces or the *triple chevron* notation[2]. Along with input arguments, kernel launches require specifying the *launch configuration*, defining the number of thread blocks and threads per block to execute the kernel. Programmers can also define device functions and global variables, accessible only by GPU code. Kernel functions are denoted with the `__global__` qualifier, while other device functions and global variables are denoted using the `__device__` qualifier.

Compiling GPU-enabled code ahead-of-time *splits* compilation to two different branches: one compiling code targeting the host and another compiling code targeting the device. Figure 1 exemplifies split compilation, following the Clang/LLVM implementation.

Device compilation.

Clang directs GPU code (qualified functions and globals) to the device compilation path, emitting a separate *Device LLVM IR* module. LLVM optimizes this IR and passes it to the backend – AMDGPU for AMD or NVPTX for NVIDIA –

[2]The CUDA/HIP syntax for launching kernels, e.g., `kernel<<<grid_dim, block_dim>>>(args)`

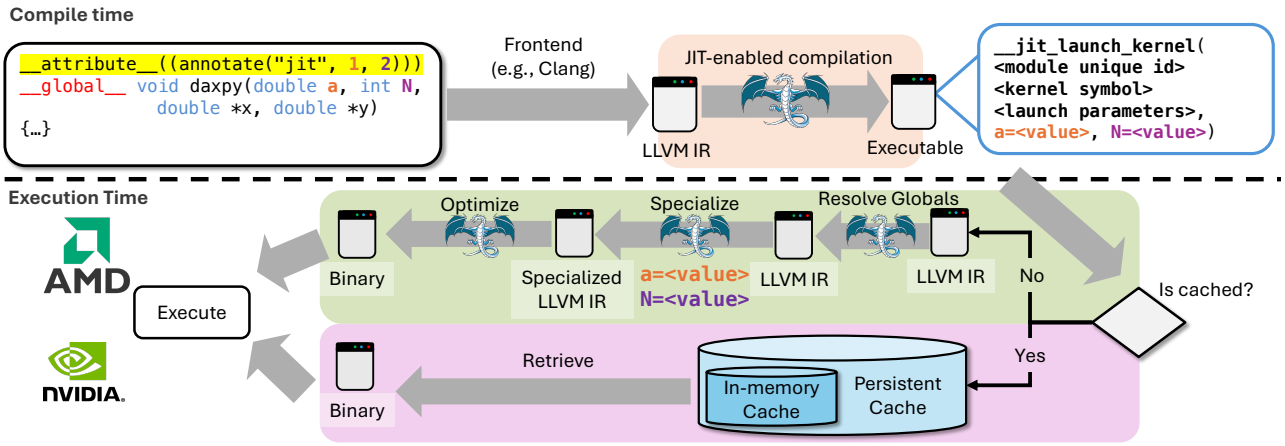


Figure 2. An overview of the Proteus JIT approach.

to generate machine code. AMDGPU produces binary code directly, while NVPTX generates PTX assembly (NVIDIA's re-targetable GPU assembly), requiring NVIDIA's assembler to produce the final binary. The device binary is then embedded in the host binary for runtime kernel execution.

A crucial GPU-specific optimization at compile time involves maximizing thread occupancy and register usage. GPU architectures have a fixed number of registers per processor[3], shared among threads in scheduled thread blocks. CUDA/HIP's `launch_bounds` qualifier allows specifying the *maximum* threads per block and *minimum* blocks per processor, enabling optimized register allocation for maximum occupancy. Without it, the compiler assumes multiple thread configurations, leading to conservative register allocation and performance degradation from register spilling. Since launch parameters depend on runtime inputs, we revisit launch bounds as a JIT optimization, leveraging the runtime-known kernel launch parameters for improved performance.

Host compilation. For host compilation, Clang emits a separate *Host LLVM IR* module lowering host code, including CUDA/HIP runtime API calls for device binary loading and kernel launching. Clang generates a *shadow host function stub* for each kernel and a *shadow host global variable* for each device global variable, associating them with their device counterparts via calls to the CUDA/HIP internal runtime API. The shadow host function launches the corresponding GPU device kernel using the CUDA/HIP runtime API. Clang replaces triple-chevron calls with calls to the shadow function stub and uses the stub's address to refer to the kernel in direct launch kernel calls of the CUDA/HIP runtime API.

Our JIT extensions in AOT compilation operate on both host and device compilation paths to extract code and runtime information, as detailed next in Section 3.

3 Proteus: Design and Implementation

3.1 Overview

Figure 2 shows an overview of the workflow of Proteus. In this example, the programmer annotates a kernel named `daxpy`, corresponding to scaled vector addition, for JIT compilation and optimization. The annotation consists of the generic *annotate* function attribute[4]. Its parameters are the string "`jit`", signifying JIT compilation, and a list of integers corresponding to kernel arguments (indexed from 1) to specialize for JIT optimization. Specifically, this annotation specifies to specialize for argument `a` (1), which is the scaling factor, and the argument `N` (2), which is the vector size.

Compile time. During AOT compilation, Proteus extensions in LLVM parse those annotations and extract the unoptimized LLVM IR of the associated kernels. Moreover, they modify AOT code generation to replace calls to kernels with calls to the JIT compilation runtime library through an entry point function. The entry point takes as input arguments the module identifier as well as the kernel symbol to extract its LLVM IR, and the runtime values of kernel arguments and launch parameters for specialization. Further, AOT code generation modifications include adding instrumentation functions to register device global variables with the JIT runtime library of Proteus. Those are needed to dynamically link device global variables to the JIT-generated binaries – we expand on this in the implementation discussion later. The end product of AOT compilation with Proteus extensions is an executable linked with the JIT runtime library for dynamic compilation, optimization, and execution.

Runtime. At execution time, this JIT-enabled executable calls into Proteus's runtime library whenever invoking an annotated kernel. The library first checks if there is a precompiled, cached instance of this kernel specialization, taking into account the runtime information to retrieve it from

[3]Streaming multiprocessor in NVIDIA or compute unit in AMD

[4]https://clang.llvm.org/docs/AttributeReference.html

cache. If the cache hits, the JIT runtime library retrieves the cached entry and executes it, saving the overhead of dynamic compilation and optimization. Otherwise, the runtime parses the LLVM IR module of the kernel, links any device global variables, specializes the IR on runtime values, and compiles it with an aggressive optimization pipeline to leverage specialization for generating highly-optimized binary code. Lastly, it executes the generated kernel binary and returns to its caller in the main program.

The design of Proteus is implemented in two components: an LLVM module pass plugin that extends AOT compilation for code and runtime information extraction, and a JIT runtime library for dynamic compilation and optimization. AOT extensions are purposefully implemented as an LLVM plugin to easily integrate with existing LLVM installations. The JIT runtime library itself utilizes LLVM as a library when parsing and optimizing the extracted LLVM IR. Additionally, it utilizes CUDA/HIP runtime APIs for loading JIT-compiled modules, launching kernels, and in the case of CUDA code, lowering the LLVM-generated PTX assembly to machine code. We expand on those components in the next sections.

3.2 AOT Compilation Extensions

Proteus includes an LLVM plugin pass that parses attribute annotations for specifying JIT kernels. Executing in the LLVM context, it automatically detects whether it is invoked during host or device compilation by inspecting the targeted architecture and adjusts its operation. For device compilation, it extracts the LLVM IR bitcode of JIT-annotated kernels and stores them as byte arrays to be read by the JIT runtime library when generating code. Whereas for host compilation, it replace calls in the host LLVM IR to launch annotated kernels with calls into the JIT runtime library, passing runtime information for optimized execution. Additionally, it inserts instrumentation routines to relay the addresses of device global variables to the JIT runtime library. We elaborate on those two modes next.

Device compilation. The LLVM IR represents attribute annotations as entries in a global array, named by convention as `llvm.global.annotations`. Each entry in this array includes the corresponding kernel function, the annotation identifier "jit" and the list of kernel argument indices to specialize at runtime. The plugin pass extracts the LLVM IR bitcode for each annotated kernel as a byte array and stores it in an emitted global variable, uniquely named following the pattern `__jit_bc_<kernel symbol>`. The goal is to expose extracted kernel's LLVM IR bitcode to the JIT runtime library by embedding it in the device binary (see Figure 1).

Implementation wise, this slightly differs depending on AMD or NVIDIA compilation. In particular, for AMD device compilation, the plugin allocates the global variables storing the LLVM IR of kernels to different, designated sections in the device binary, uniquely named as `.jit.<kernel symbol>`. Following, the JIT runtime library extracts the LLVM IR bitcode for a kernel using its symbol to retrieve it from the corresponding section.

The same method cannot be used for CUDA compilation because NVIDIA's proprietary binary tools discard such non-standard sections. To circumvent this, the plugin pass stores the global variable containing the byte array of the kernel's LLVM IR code to the standard data section of the device binary. Then, the JIT runtime library uses the CUDA runtime API (`cuModuleGetGlobal`) to retrieve the global variable's content from device memory using kernel's symbol, at the cost of an extra read operation from the device prior to dynamic compilation.

Host compilation. During host compilation, the plugin pass modifies host LLVM IR to redirect all calls to annotated device kernels to the JIT runtime library. Also, it injects calls to the JIT runtime library to register any device global variables. This is needed because the JIT runtime library must dynamically resolve the memory addresses of device global variable symbols and link them with JIT-generated code. This is to ensure that global variables are common between all device code, JIT-generated or AOT-generated.

In more details, JIT annotations of kernels are visible to host compilation as annotations to the host's shadow kernel stub function (see section 2). The LLVM plugin finds calls to `cudaLaunchKernel` (CUDA) or `hipLaunchKernel` (HIP) and inspects whether they launch kernels corresponding to the annotated stubs. For such calls, the plugin replaces them with a call to the JIT runtime library's entry point (`__jit_launch_kernel`), forwarding: (1) an LLVM-generated source module unique identifier bound to source code used for caching (more details in section 3.3), (2) the unique kernel symbol, (3) the kernel launch parameters including the grid/block size, shared memory, and stream number needed to mirror the call to `cudaLaunchKernel` or `hipLaunchKernel`, and (4) the runtime values of denoted kernel arguments for specialization.

The pass detects the existence of device global variables by tracking calls to registration API functions for vendor runtimes, namely `__cudaRegisterVar` and `__hipRegisterVar`. Typically, those functions are invoked during program initialization within a global constructor to link the device global variable symbol with its shadow host counterpart. The plugin adds a call to our own registration function in the JIT runtime library, named `__jit_register_var`, forwarding the global variable's symbol, which enables the runtime to query its device memory address. The JIT runtime uses this functionality to dynamically link references to device global variables in the JIT-compiled modules to their registered address in GPU global memory resolved at runtime.

3.3 JIT Compilation Runtime Library

Proteus implements a JIT runtime library, based on LLVM, to specialize, optimize and compile the extracted LLVM IR. The library invokes vendor-specific CUDA/HIP interfaces to

create and load object binaries of the JIT-compiled kernels, and to launch those kernels. It also implements specialization-aware code caching, both in-memory and persistent, to avoid re-compilation overhead for previously JIT compiled code.

Referring back to Figure 2, during execution, the host executable calls into the JIT runtime library through the `__jit_launch_kernel` interface when launching a kernel. If cache misses, the JIT runtime retrieves the LLVM IR bitcode for the invoked kernel to specialize, optimize, and launch. If it hits, the runtime retrieves the optimized kernel specialization and launches it without the dynamic compilation overhead. The following discussion goes into details on the different operations of the JIT runtime library.

Dynamic linking of device global variables. The first step in modifying the extracted LLVM IR of a kernel is to link any references to device global variables. Device global variables are assigned to memory locations at loading time of the device binary embedded in the host executable. As discussed in section 3.2, our JIT extensions in AOT compilation register device global variables with the JIT runtime. The runtime then queries the memory addresses of those symbols through CUDA/HIP runtime interfaces, namely `cudaGetSymbolAddress` and `hipGetSymbolAddress` respectively. Following, it replaces references to device global variables in the JIT module LLVM IR with the queried memory addresses resolved at runtime. That ensures that either AOT kernels or JIT-compiled kernels view and modify the same, global program state.

Dynamic specialization. After linking device global variables, the JIT runtime specializes the LLVM IR for runtime values of kernel arguments and launch parameters. In specifics, the runtime performs two specializations: it folds kernel arguments to constants, using their runtime values, and sets launch bounds for the specific threading configuration at invocation time, as communicated through `__jit_launch_kernel`, to optimize register allocation.

Runtime constant folding of kernel arguments has possibly cascading optimization effects by revealing opportunities to simplify control-flow, eliminate unused code, and enable more aggressive loop unrolling and vectorization. The JIT runtime specializes the LLVM IR bitcode by replacing uses of folded kernel arguments with their exact runtime value.

Further, explicitly setting kernel launch bounds helps register allocation to maximize register usage under expected thread occupancy, with possibly significant performance benefits [42]. To set launch bounds, the JIT runtime either adds metadata to LLVM IR for CUDA kernels, or adds a function attribute for HIP kernels, to encode the maximum number of threads per block at runtime, known through the `__jit_launch_kernel` interface. Note that setting the maximum number of threads per block is required, thus explicitly set by the JIT runtime using the exact launch value, whereas the minimum number of blocks per multiprocessor is optional, for which the runtime sets the default, minimum value of 1.

Code optimization. Next, the JIT runtime passes the specialized LLVM IR through the aggressive `O3` optimization pipeline by calling into the LLVM API. The expectation is that the output LLVM IR has additional code optimization thanks to specialization.

Machine code generation and execution. The last step is to generate machine code by invoking the LLVM backend on the specialized and optimized LLVM IR, and launch the JIT-optimized kernel through CUDA/HIP runtime interfaces. As discussed in section 2, the AMDGPU backend directly generates machine code, whereas LLVM's NVIDIA NVPTX backend generates PTX assembly, which requires NVIDIA's PTX compiler to generate the binary. Hence, for NVIDIA compilation, the JIT runtime invokes CUDA's PTX API, specifically the interface `nvPTXCompilerCompile`, as an extra step for generating machine code. Upon generating machine code, the JIT runtime inserts it in the code cache, we elaborate on it next, and launches the JIT-optimized kernel.

Code caching. For each compiler kernel specialization the JIT runtime library generates a uniquely identifying hash value. Hashing ensures uniqueness by jointly encoding: (1) a unique module identifier bound to source code, generated by LLVM, (2) the kernel's symbol, and (3) the runtime values of specialized kernel arguments and launch bound values. The code cache stores key-value pairs with the unique hash value being the key that retrieves the corresponding JIT-optimized machine code for a specific kernel specialization.

On a kernel invocation, the JIT runtime library computes the hash value encoding the above information that uniquely identifies the specialization. If the hash value matches an entry in the cache, the runtime retrieves the machine code obviating dynamic compilation overhead. If it does not, the runtime dynamically compiles the kernel specialization and inserts it in the cache.

As for the caching implementation, the JIT runtime library employs two-level caching. The first level is a fast, in-memory cache, populated afresh at runtime during program execution. The second level is a persistent storage cache, designed to retain objects across program runs, feeding the in-memory cache. The persistent storage cache is implemented using file storage, storing each JIT-optimized binary in a unique file, named `cache-jit-<hash>.o`.

In the current implementation, we do not limit the size of the persistent cache, thus it is monotonically increasing as more specializations accumulate across program runs. Nevertheless, users can clear the persistent cache as needed by deleting its files in storage, to re-populate it. One particular challenge with code caching is that source code changes must avoid stale entries in the cache from prior source code versions. Notably, our hashing method is responsive to source code changes, since the unique, LLVM-generated module identifier changes if source code changes. Due to that, stale

entries will not propagate from the persistent cache to the in-memory cache, triggering instead re-compilation at runtime. Clearing the persistent cache due to source code changes is easy to integrate in software build systems by deleting cache files when re-building.

3.4 Discussion

Our approach purposefully operates at the LLVM IR level for both code and runtime value extraction and optimization, to facilitate porting to other frontends besides Clang (e.g., Flang for Fortran, Rust) that target LLVM IR. Proteus differs from other state-of-the-art solutions [3, 16, 39, 41] which are bound to source code or Clang's AST, thus restricted to a single language. Besides porting advantages, JIT specialization and optimization using the LLVM IR avoids parsing and syntax/semantic analysis overheads at runtime, when this is required for language-bound solutions.

Regarding caching, our present implementation aggressively caches specializations without eviction, assuming programmers judiciously use JIT annotations for hotspots with a manageable number of runtime specializations. We are developing Least Recently Used (LRU) eviction to prevent cache overgrowth and plan to add environment variables for users to limit in-memory and persistent caches sizes. Additionally, we aim to explore runtime-informed eviction mechanisms that prioritize evicting less likely-to-execute specializations over LRU. As shown in the next section, the evaluated programs' code caching is in the KB range, which does not require such mechanisms.

4 Evaluation

We select a comprehensive set of programs from the HeCBench benchmark suite [29], as shown in Table 1, to evaluate Proteus. HeCBench is curated by the HPC community and includes benchmark implementations representative of large applications in HIP and CUDA. We annotate kernels in those programs for JIT compilation, specifying meaningful arguments for runtime specialization (arguments used in loop bounds, conditionals, or numeric computation).

We evaluate on both an NVIDIA V100 GPU and an AMD MI250X GPU, thanks to the portability of our approach. For our software setup, we use the Clang/LLVM compiler provided by AMD in ROCm version 5.7.1, and the Clang/LLVM version 17.0.5 for compiling CUDA codes using CUDA 12.2 libraries. Program implementations in HIP are executed on the AMD GPU, while CUDA ones on the NVIDIA GPU.

We contrast the Proteus JIT optimization approach with AOT compilation on both NVIDIA and AMD systems in terms of performance and overheads. Additionally, for NVIDIA, we contrast it with NVIDIA's Jitify [39] tool, which performs similar specialization, though limited to C++ templates and a cumbersome interface requiring to provide kernel code in a string representation. To ensure statistical stable results,

Table 1. Benchmark programs.

Benchmark	Domain	Input
ADAM	Machine Learning	160000 1600 1000
RSBENCH	Neutron Transport Algorithm	-m event -s large
WSM5	Weather Simulation	10
FEY-KAC	Monte Carlo PDEs	1
LULESH	Physics	-s 128
SW4CK	Earth Science	sw4ck.in 1000

Table 2. End-to-end execution time per program and method. Highlighted entries show the fastest execution time.

AMD	End-to-end Execution Time (s)					
	ADAM	RSBENCH	WSM5	FEY-KAC	LULESH	SW4CK
AOT	2.56±0.11%	3.78±0.15%	9.10±0.10%	10.66±0.07%	63.25±0.01%	120.75±0.03%
Proteus	2.03±0.06%	2.28±0.14%	3.62±0.02%	8.36±0.09%	62.05±0.04%	84.96±0.08%
Proteus+$	2.01±0.09%	1.91±0.10%	3.25±0.08%	8.27±0.06%	61.70±0.07%	83.11±0.07%
NVIDIA	**End-to-end Execution Time (s)**					
	ADAM	RSBENCH	WSM5	FEY-KAC	LULESH	SW4CK
AOT	2.87±0.23%	2.67±0.15%	3.69±0.23%	14.57±0.03%	24.39±1.64%	53.20±0.04%
Proteus	2.20±0.25%	2.38±0.43%	2.73±0.21%	13.17±0.09%	24.64±0.58%	57.22±0.02%
Proteus+$	2.11±0.34%	2.01±0.55%	2.08±0.21%	12.96±0.08%	24.32±0.26%	53.64±0.02%
Jitify	3.35±0.31%	3.34±0.37%	2.73±0.21%	15.04±0.06%	N/A	57.76±0.13%

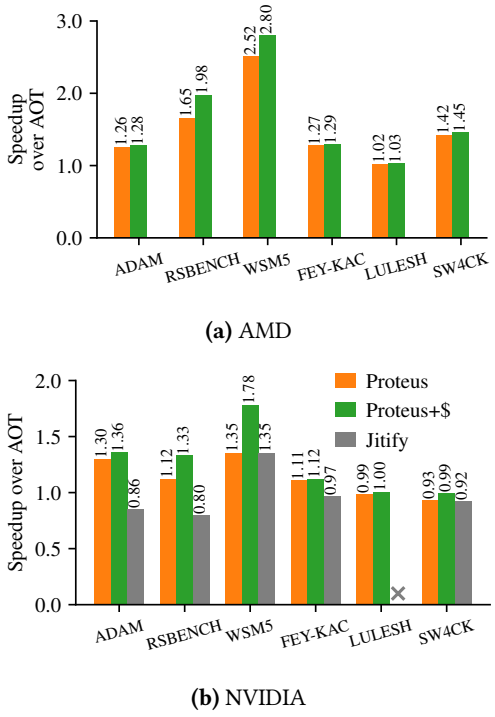
each program goes through a warm-up run discarding measurements, to avoid any static initialization overheads. Then, we perform three separate program runs and report the mean out those three runs. We found the relative standard error of the mean to be less than 1.64% in all cases (see Table 2), which is expected since GPU is an isolated accelerator platform, thus results are deemed statistically significant.

4.1 End-to-End Performance Comparisons

Figure 3 presents the end-to-end speedup for each program across both architectures. We contrast AOT compilation to Proteus JIT compilation with all runtime optimizations activated. To show the effects of caching on speedup, we show results as *Proteus* when execution starts with a "cold" persistent cache, hence incurring the full overhead of dynamic compilation, and as *Proteus+$*, when it starts with a "warm" cache of pre-compiled kernel specializations, hence overhead consists of only loading the kernel binary from persistent storage. Additionally, for the NVIDIA system, we compare against NVIDIA's *Jitify* tool for all programs except LULESH, as *Jitify* hangs during compilation.

Overall, results show that Proteus, owing to runtime optimization, demonstrates significant speedup across both architectures, even when the persistent cache is "cold", effectively recuperating dynamic compilation overheads.

For AMD, 5 out of the 6 evaluated programs significantly speed up between 1.26× to 2.52× with a "cold" cache and 1.28× to 2.8× with a "warm" cache. The one exception is LULESH, where Proteus marginally speeds up execution (1.02× without caching, 1.03× with), which shows that specialization optimizations have little benefit for this program.



(a) AMD

(b) NVIDIA

Figure 3. End-to-end speedup over AOT, incl. JIT overhead.

For NVIDIA, 4 out of 6 programs exhibit noticeable speedup, ranging from 1.11× to 1.35× for a "cold" cache and 1.12× to 1.78× for a "warm" cache. Consequently, a pre-populated cache is more important for achieving higher speedup on NVIDIA rather than AMD. This is expected since NVIDIA requires the extra runtime compilation overhead for pulling LLVM IR from device memory and the intermediate step of PTX compilation. LULESH on NVIDIA also does not benefit from JIT specialization, hence its performance is on a par with AOT. A difference between the two platforms is that SW4CK under Proteus significantly speeds up on AMD (up to 1.45×), whereas it slightly slows down for NVIDIA. Specifically, SW4CK with Proteus on NVIDIA is slightly slower (0.93×) with a "cold" persistent cache compared to AOT, or it has similar performance (0.99×) when the cache is "warm". The brief explanation is that JIT specialization benefits AMD, but not NVIDIA. Specifically, setting launch bounds optimizes register allocation on AMD but has no effect on NVIDIA, whereas runtime constant folding does not benefit either architecture. We provide more details in our in-depth analysis at section 4.5.

Interestingly, Proteus outperforms the NVIDIA-specific *Jitify* approach across all tested programs. *Jitify* results in slowdown across most programs and achieves end-to-end

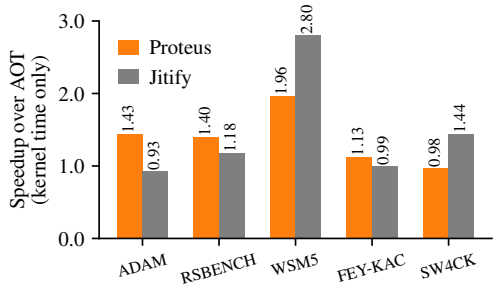


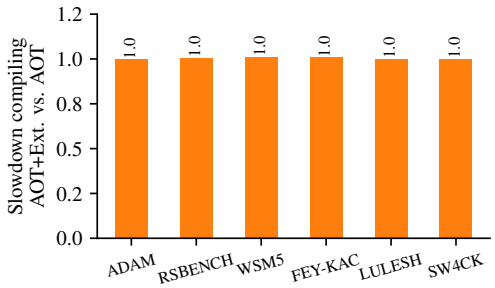
Figure 4. Speedup over AOT considering only kernel execution, excluding JIT overhead, for NVIDIA.

speedup only for WSM5. Figure 4 shows kernel-only execution times without including runtime compilation overheads, to understand whether this advantage of our JIT approach is due to faster kernel execution times or by minimizing runtime overhead, or both. Looking at results, *Jitify* generates slower kernel code than Proteus for some programs (ADAM, RSBENCH, FEY-KAC) while faster for others (WSM5, SW4CK). That suggests that our JIT approach achieves end-to-end speedup by always minimizing runtime compilation overheads, even with a "cold" persistent cache which is equivalent to *Jitify* execution, and this speedup compounds in certain cases with generating faster kernel code too. The higher dynamic compilation overhead of *Jitify* is explainable as it starts off with parsing, analyzing, and compiling stringified source code, whereas Proteus uses lower level LLVM IR for JIT compilation and optimization.

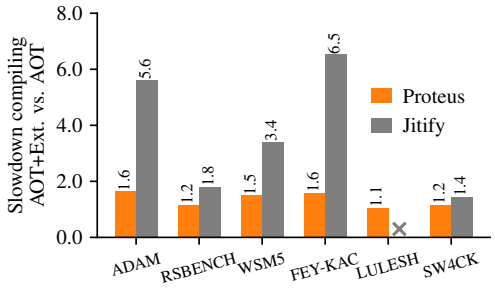
4.2 AOT Extensions Compilation Overhead

Since both Proteus and NVIDIA's *Jitify* extend AOT compilation for code and runtime value extraction, we measure the static, one-off cost when building a program including those extensions during AOT compilation. Figure 5 shows the slowdown of AOT compilation (single-threaded) when building each program with JIT extensions versus without them. For Proteus, extensions include the LLVM JIT plugin pass and linking with the runtime library. For *Jitify*, JIT extensions in compilation include usage of its header-only, templated C++ library.

Results show Proteus imposes minimal overhead across systems. For HIP/AMD, compilation overhead is negligible with no noticeable slowdown. For CUDA/NVIDIA, compilation time increases by 1.1× to 1.6× across benchmarks, primarily due to static linking of our JIT runtime library and NVIDIA's proprietary libraries as the Proteus LLVM JIT plugin itself has negligible overhead. In contrast, *Jitify* incurs greater slowdowns, ranging from 1.4× to 6.5×, due to its templated header library inflating compilation time.

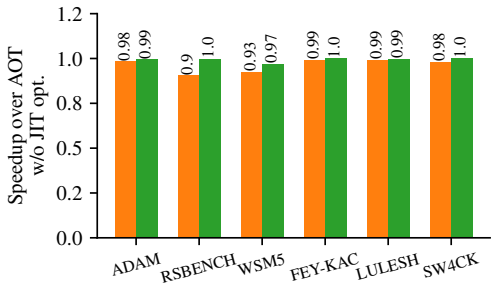


(a) AMD

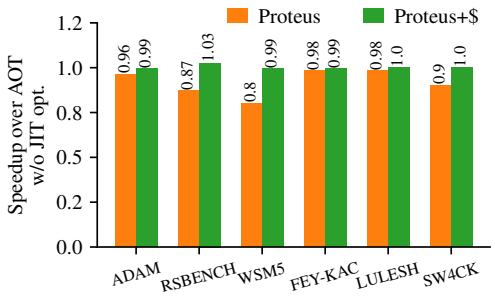


(b) NVIDIA

Figure 5. Slowdown AOT compilation with JIT extensions.



(a) AMD



(b) NVIDIA

Figure 6. Speedup (<1 is slowdown) over AOT when deliberately disabling runtime optimizations in JIT compilation.

Table 3. Maximal code cache size.

Machine	Program					
	ADAM	RSBENCH	WSM5	FEY-KAC	LULESH	SW4CK
NVIDIA	5.9KB	32KB	73KB	18KB	54KB	282KB
AMD	6.7KB	24KB	35KB	18KB	46KB	199KB

4.3 Runtime Overheads

We conduct experiments where we deliberately turn off any runtime specialization to measure the overhead of dynamic JIT compilation in Proteus. Instead, we use our kernel annotations and dynamically compile kernels only with the default O3 pipeline, akin to AOT compilation. This way JIT compilation will be invoked as many times as there are possible specializations, despite not performing specialization as it is explicitly turned off, to expose the overheads of dynamic compilation without the benefits of specialization.

Figure 6 shows the end-to-end speedup under this setup, showing results with and without a pre-populated persistent cache. Note that the pre-populated persistent cache contains kernel objects optimized just with O3, hence without runtime specialization. Results affirm that Proteus compilation overhead is indeed small and that caching significantly mitigates it. The best case under this setup is a speedup of 1, meaning negligible slowdown due to JIT compilation. For AMD, the slowdown of JIT without caching is small, between 0.9×–0.99×, whereas with caching it is effectively negligible, ranging between 0.97×–1.0×. For NVIDIA, overhead is greater than AMD, due to pulling LLVM IR from device memory and the need to compile PTX for machine code generation. However, that overhead is still manageable. Specifically, slowdown using Proteus without caching is between 0.8×–0.98×, while with caching it is negligible, between 0.99×–1.0×. Interestingly, RSBENCH is an outlier where we see marginal speedup of 1.03×, attributed to marginal differences in machine code generation when using the PTX compiler through our runtime library. Revisiting Figure 3 on end-to-end speedup, those small overheads are effectively recuperated when runtime optimization is enabled, hence the significant end-to-end speedup from Proteus.

4.4 Code Cache Size

Table 3 shows the *maximal* cache size for each program on both machines, including all specializations without eviction or cache size limits. Code caches are typically small (on the order of KB), resulting in minimal overhead. Nonetheless, we will include eviction mechanisms and size limits to manage cache scaling and aggressive specialization (see Section 3.4).

4.5 Detailed Performance Analysis

Further, We conduct an in-depth analysis of execution time and hardware performance counters under various specialization optimizations to identify factors driving the observed

```
template <typename T, typename G>
__global__
__attribute__((annotate("jit", 5, 6, 7, 8, 9, 10, 11, 13)))
void adam (
  T* __restrict__ p,
  T* __restrict__ m,
  T* __restrict__ v,
  const G* __restrict__ g,
  const float b1,
  const float b2,
  const float eps,
  const float grad_scale,
  const float step_size,
  const int time_step,
  const size_t vector_size,
  adamMode_t mode,
  const float decay)
```

Listing 1. The ADAM kernel prototype with JIT annotations. Highlighted arguments are designated as runtime constants.

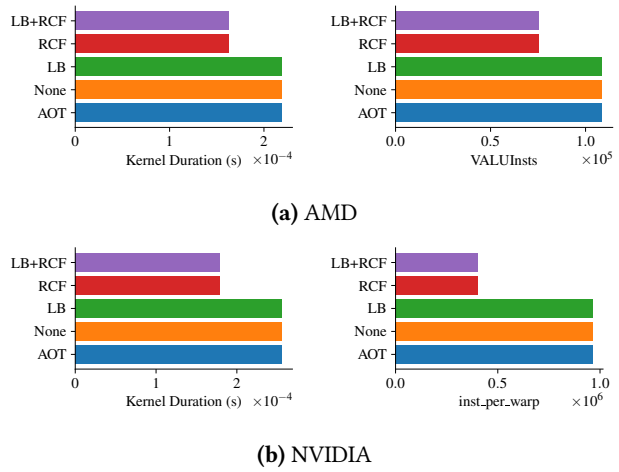
performance improvements. For that, we utilize the vendor profiling tools *nvprof* (NVIDIA) and *rocprof* (AMD):

- *AOT* denotes results when the kernel has been compiled and optimized using AOT compilation only with the default O3 optimization pipeline.
- *None* denotes Proteus JIT compilation, but without specialization, hence only O3 optimization, expecting similar performance with AOT compilation.
- *LB* denotes results when Proteus JIT compilation specializes the kernel only for launch bounds, and optimizing with the O3 pipeline, to show the impact of better register allocation through runtime threading configuration.
- *RCF* denotes results when Proteus specializes the kernel only with runtime constant folding of kernel arguments, optimizing with the O3 pipeline, to show the effects of runtime folding for enhanced optimization.
- Lastly, *LB+RCF* denotes results when Proteus specializes the kernel with both runtime constant folding and launch bounds, and optimizing with the O3 pipeline, to show the combined effects of our runtime optimizations.

Those different modes aim to isolate and understand the individual and combined effects of specializations, and the resulting kernel performance enabled by Proteus JIT compilation. Next, we discuss detailed performance results for each kernel.

ADAM. The Adam [31] optimizer is a popular algorithm in the field of machine learning for optimizing neural network parameters. Listing 1 shows the kernel function prototype from the ADAM implementation with JIT annotations, designating all scalar variables[5], as runtime constants – recall

[5] `b1, b2, eps, grad_scale, step_size, time_size, vector_size, decay`



(a) AMD

(b) NVIDIA

Figure 7. In-depth analysis of the ADAM benchmark.

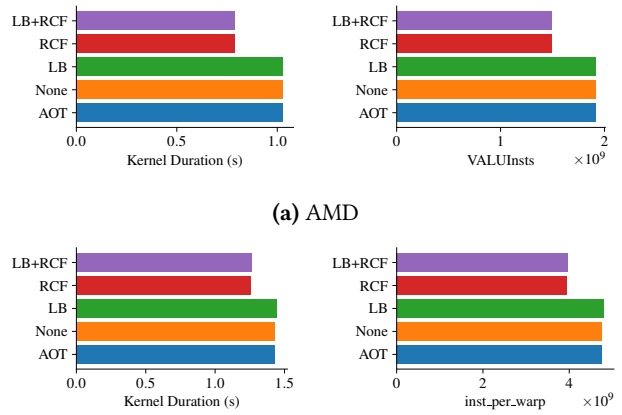
```
void potential(double a, double b,
               double x, double y) {
return (2.0 * ( pow ( x / a / a, 2.0 )
        + pow ( y / b / b, 2.0 ) )
        + 1.0 / a / a + 1.0 / b / b);
}
```

Listing 2. Function computing the potential of a point in the inner loop of FEY-KAC, called it in the innermost loop of the computational kernel function.

argument counting starts from 1 in our annotations. Figure 7 shows results on kernel duration and number of executed instructions. We consistently observe speedup across both systems. Notably, runtime constant folding is the most effective specialization technique for yielding higher performance. The primary factor contributing to this improvement is a significant reduction in the number of executed instructions.

Specifically, performance profiling on AMD reveals a substantial decrease in the average number of vector ALU instructions executed per work item (`VALUInsts`), dropping from 108,854 to 75,226. On NVIDIA, the total number of executed instructions (`inst_per_warp`) also decreases from approximately 9.6E5 to 4E05.

FEY-KAC. The Feynman-Kac formula [12] establishes a connection between the solutions of certain partial differential equations (PDEs) and expected values from stochastic processes, with applications in quantum mechanics and financial mathematics. Using Monte Carlo methods, individual computational threads execute stochastic trajectories, calculating the *potential* at discrete points on a two-dimensional lattice, as illustrated in Listing 2. Runtime constant folding specialization for variables `a` and `b` enables JIT compilation to eliminate redundant vector instructions, improving computational efficiency.



(a) AMD

(b) NVIDIA

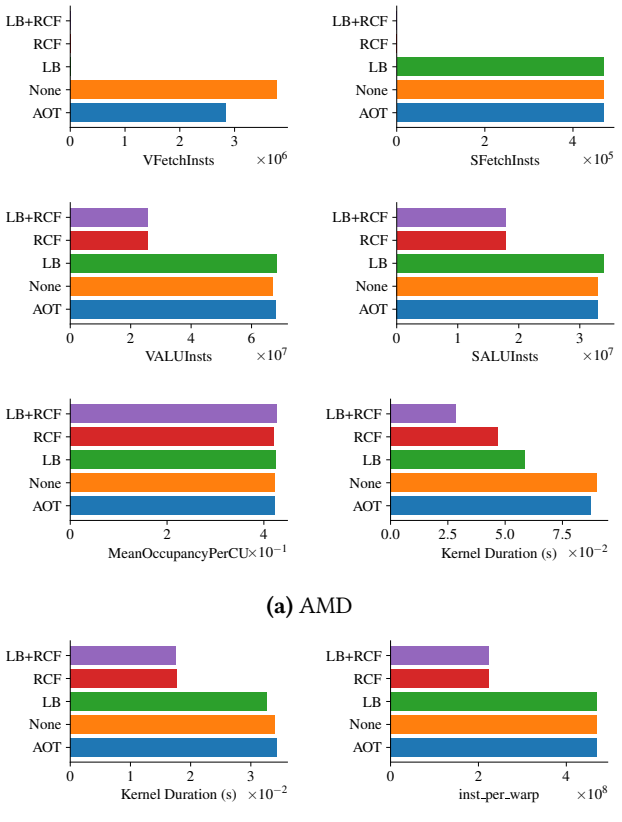
Figure 8. In-depth analysis for FEY-KAC.

Figure 8 shows results on kernel duration and instruction counters. Primarily RCF reduces vector instructions compared to AOT. On AMD, AOT has around 1.9E9 vector instructions, whereas JIT-optimized versions with RCF reduce them to 1.4E9 vector instructions, a reduction factor of 1.29×. Similarly on NVIDIA, Proteus reduces instructions from 4.7E9 to 3.9E9 instructions, achieving a 1.12× kernel speedup.

WSM5. WSM5 (Weather Research and Forecasting Single Moment 5-class) [35] is a weather simulation benchmark focusing on cloud microphysics for meteorological research. Figure 9 highlights differing optimization behaviors between AMD and NVIDIA GPUs. For NVIDIA, runtime constant folding (RCF) reduces instructions by 2.1× compared to AOT, achieving a 1.95× kernel speedup, although memory-bound limits lower the Instructions-Per-Cycle (IPC) for JIT (0.9) compared to AOT (1.4).

On AMD, combining RCF and LB yields the best results, reducing both vector and scalar ALU instructions (`VALUInsts`, `SALUInsts`) and eliminating costly memory spills for vector and scalar registers (`VFetchInsts`, `SFetchInsts`). In more detail, AOT allocates 128 vector and 102 scalar registers, leading to significant memory spills and a kernel duration of 0.088 seconds. RCF alone uses the same number of registers but eliminates spills and reduces vector/scalar instructions, achieving a 1.87× speedup with a duration of 0.047 seconds. LB alone aggressively allocates 212 vector and 98 scalar registers, eliminating vector spills but keeping scalar spills, nevertheless resulting in a 1.5× speedup (0.059 seconds).

The greatest improvement comes from combining RCF and LB, reducing vector register usage, eliminating register spills, and reducing instruction counts, thus achieving the shortest kernel duration of 0.029 seconds – a 3.03× speedup over AOT. These results demonstrate that while RCF and



(a) AMD

(b) NVIDIA

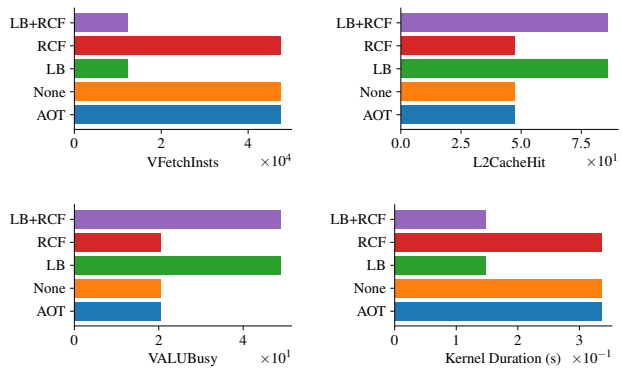
Figure 9. In-depth analysis for the WSM5 benchmark.

LB individually improve performance, their combination delivers the highest gains.

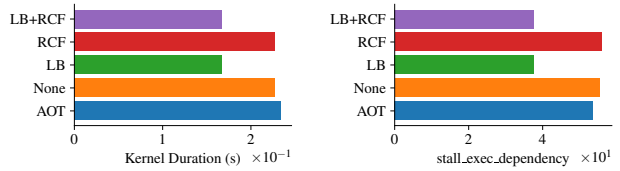
RSBench. Figure 10 presents the analysis of the RSBench HPC proxy application. On AMD, LB achieves a 2.45× speedup over AOT by optimizing register allocation, reducing memory spills, and increasing the L2 cache hit ratio. It also improves vector ALU utilization, as indicated by the higher `VALUBusy` counter, measuring the percentage of time GPU execution spent in computation.

For NVIDIA, LB again delivers a 1.4× speedup over AOT by minimizing memory spills and reducing pipeline stalls, as measured by the `stall_exec_dependency` counter.

SW4CK. Figure 11 shows the analysis of the five kernels in the SW4CK benchmark on AMD. LB specialization delivers significant improvements across all kernels, achieving an average 2.99× speedup due to better register utilization and a higher L2 cache hit ratio (from 50% to 78%, as shown in Figure 11b). In contrast, RCF does not provide speedup and degrades performance for `kernel4` due to suboptimal register usage, leading to more spills and a lower L2 cache hit ratio (49% vs. 56%). However, combining LB and RCF results in a net speedup, as LB compensates for RCF's inefficiencies.



(a) AMD



(b) NVIDIA

Figure 10. In-depth analysis for the RSBench benchmark.

Results for NVIDIA are omitted as neither LB nor RCF yield improvements over AOT. NVIDIA's proprietary register allocator already optimizes effectively, rendering LB unnecessary, while RCF fails to optimize due to limited opportunities for computational instruction improvements.

LULESH. Results are omitted for NVIDIA and AMD as execution times match the AOT baseline, showing no noticeable speedup. Nonetheless, this demonstrates that Proteus is lightweight and avoids slowdowns, even for programs less amenable to JIT optimization.

5 Related Work

The foundations of Just-In-Time optimization are based on partial evaluation [30], a longstanding technique for optimizing programs. Partial evaluation specializes a program modulo its inputs to generate a specialized version of the program, called the *residual*. Futamura [17–19] lays the theoretical foundation on partial evaluation and describes how partial evaluations applied progressively on an interpreter generate an optimized program executable (*first projection*), a compiler (*second projection*), or a compiler generator (*third projection*). Sullivan [44] discusses theoretical aspects of *dynamic partial evaluation* when specializing a program to its environment, i.e., arguments of functions, in a simple lambda calculus language. Partial evaluation via dynamic JIT compilation can be used to optimize both dynamic, interpreted languages and statically typed, compiled ones.

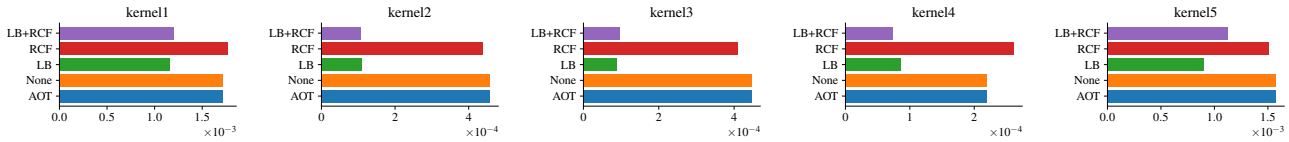
Interpreted languages, such as Java [4, 13, 48, 49], Julia [5], Python [6, 32], Ruby [9], Javascript [11], frequently rely on JIT compilation to accelerate their execution. JIT compilers for those languages alleviate the interpretation overhead by compiling hotspots of the program for native execution. JIT compilation for dynamically-typed languages often uses type-based specialization [8, 9, 20, 24, 43, 47] as a partial evaluation technique to reduce boxing/unboxing overhead and generate specialized computational instructions. Cost et al. [11] propose automatic value specialization for JavaScript functions in Mozilla's IonMonkey Javascript JIT, as those are called repeatedly with the same parameter values.

JIT compilation for GPUs in Python is an important area, due to its prevalence for AI/ML codes, with Numba [32], Triton [46] being two state-of-the-art solutions. They both work on a subset of Python and inspect the Python AST, which is available at runtime, having ample time for dynamic compilation due to the interpreted nature of the language. By contrast, Proteus works on statically compiled languages with stringent performance constraints given highly-optimizing AOT compilation. Additionally, those approaches use LLVM IR just for backend generation of binary code, whereas Proteus uses LLVM IR also to introspect and specialize the code before applying automated compiler-based optimization.

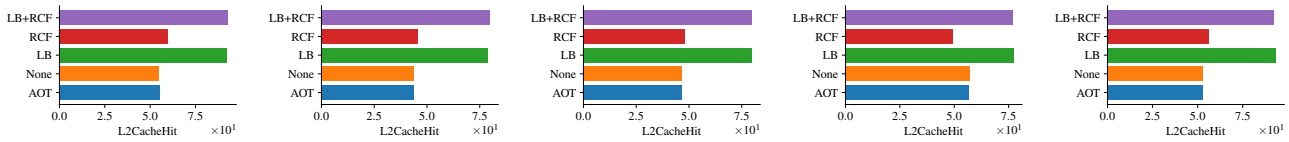
'C [14], Tempo [10, 37], DyC [22, 23] propose extensions to the C language to indicate possible variables as runtime constants for specializing at scope level. Tempo produces specialized code variants offline whereas 'C, DyC produce a customized IR fed to a code generator at execution time.

Few JIT solutions leverage Clang/LLVM but are limited to C++ with minimal GPU support. EasyJIT [7] offers a C++ API using *placeholders* for runtime function specialization via LLVM IR. atJIT [15] extends EasyJIT with tuning capabilities. ClangJIT [16] introduces a C++ attribute for annotating templates, including CUDA programs, deferring specialization to runtime to reduce AOT compilation time. It specializes functions for non-type template parameters at runtime by operating on the Clang AST. PACXX [25–27] extends Clang with a C++ API for GPU development using CUDA and OpenCL. It specializes C++ lambda GPU kernels by embedding user-selected captured variables as constants into the kernel for dynamic compilation. Similarly, LambdaJIT [34] specializes kernels for all captured variables of a lambda. An OpenMP-specific approach [45] uses JIT compilation to optimize GPU kernels by re-targeting them to the GPU architecture, removing redundant calls to OpenMP runtime functions, and folding scalar kernel arguments to constants.

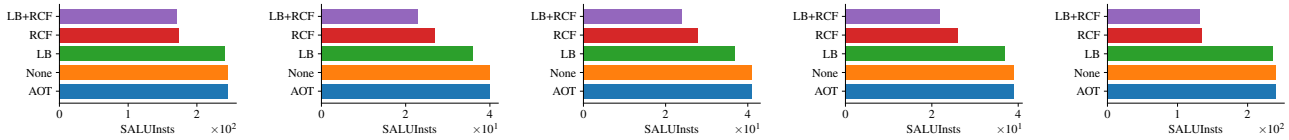
Further, CUDA/HIP Runtime Compilation (RTC) [3, 41] supports JIT compilation for C/C++ kernels, with NVIDIA Jitify [39] providing a high-level interface. Jitify is a single header C++ library that supports runtime constants through template specialization, and an experimental API for user-managed caching. Both CUDA/HIP and Jitify digest C/C++ source code as a string and invoke the full compilation toolchain, with significant associated overhead, to generate binary code. KART [36] shares ideas with CUDA/HIP



(a) Average execution time per kernel invocation.



(b) L2 Cache hit rate for all kernels included in sw4ck.



(c) Number of Scalar Instructions per kernel invocation.

Figure 11. In-depth analysis of the SW4CK benchmark on AMD.

RTC and NVIDIA's Jitify as it provides API abstractions to JIT compile source code in textual format, requiring a full compiler toolchain at the backend. All those approaches require significant code refactoring effort to define kernels as C/C++ strings, while the overhead of invoking the complete toolchain to parse, lower, and compile source code is non-trivial.

Regarding possible JIT optimizations, authors in [28] use profile-guided information to specialize the register allocation of functions by cloning specializations based on runtime values affecting control flow. Another profiled guided approach [38] proposes to use JIT compilation and optimize code layout, by re-compiling parts of the LLVM IR. DASS [21] proposes embedding the LLVM IR in the executable for a generic target architecture for portability, to later specialize through dynamic compilation for the specific sub-target architecture at deployment time. Those runtime specializations are complementary to our approach.

Table 4 summarizes the differences between our work and other GPU-targeting JIT approaches. In summary, Proteus presents an easy-to-use, portable, language agnostic, high-performance JIT compilation methodology by providing simple source annotations, operating in LLVM IR for portability, and implementing JIT specialization optimizations as well as specialization-based in-memory and persistent code caching.

6 Conclusion

We introduced Proteus, a portable, high-performance JIT compilation approach that optimizes GPU kernel execution with minimally intrusive developer annotations and runtime specialization. Proteus extends LLVM's AOT compilation to extract LLVM IR and runtime information for annotated kernels, incorporating a JIT runtime library for dynamic optimization and caching. Specialization optimizations include runtime constant folding for kernel arguments and dynamic launch bounds to improve register allocation. Evaluations on HPC benchmarks with NVIDIA and AMD GPUs show Proteus achieves 1.26× to 2.8× end-to-end speedup through effective specialization and minimal overhead.

Future work includes exploring runtime optimizations like kernel scheduling and auto-tuning, automating specialization decisions to balance performance and compilation overhead, and porting Proteus to additional language frontends and GPU architectures.

Table 4. Contrasting key attributes of GPU JIT approaches.

Method	JIT Code	Portability		Cache	
		NVIDIA	AMD	In-memory	Persistent
RTC/Jitify [39, 40]	C++	✓	✗	✓	✓*
PACXX[25–27]	C++	✓	✗	✓	✗
LambdaJIT [34]	C++	✓	✗	✗	✗
ClangJIT [16]	Clang AST	✓	✗	✓	✗
Proteus	LLVM IR	✓	✓	✓	✓

*user-managed (experimental)

Acknowledgments

This work was prepared by LLNL under Contract DE-AC52-07NA27344 and supported by the LLNL LDRD Program under Project No. 23-ERD-022 and Project No. 25-ERD-019. (LLNL-CONF-869329). The authors thank anonymous reviewers for their valuable feedback and LLNL colleagues Tal Ben-Nun, John Bowen, and Thomas Stitt for their support.

A Artifact Appendix

A.1 Abstract

The artifacts comprise of:

- the Proteus software stack including the LLVM plugin pass and the runtime library,
- the HeCBench benchmark programs modified for JIT compilation using Proteus annotations or NVIDIA Jitify, and
- the experimentation driver and data analysis scripts that compile and execute HeCBench programs to plot results

All artifacts are provided open-source, publicly available in Zenodo (https://doi.org/10.5281/zenodo.14087063) and GitHub (https://github.com/Olympus-HPC/proteus.git, branch: `cgo25-artifact`).

The hardware requirements are an NVIDIA V100 GPU hosted on an IBM Power9 machine and an AMD MI250X GPU hosted on an AMD EPYC 7A53 machine. For experimentation, the GPU systems are more important, the host system configurations are less essential.

For the NVIDIA GPU machine, software requirements are a Linux OS (RHEL 7.9), an NVIDIA CUDA 12.2 installation, Clang/LLVM 17.0.5, and Python 3.

For the AMD GPU machine, software requirements are a Linux OS (RHEL 8.10), AMD ROCm 5.7.1 (which includes AMD's Clang/LLVM installation), and Python 3.

The key results to reproduce are Proteus end-to-end speedup over AOT compilation (Figure 3) and the end-to-end execution times (Table 2). Results in the rest of figures are elaborations of the key result, showing overheads, GPU kernel-only execution times, or detailed performance counters.

The experimentation workflow is to compile benchmark programs with Proteus/Jitify and execute to collect timing/profiling measurements. The driver and data analysis scripts we provide automate the experimentation workflow and perform data collection and plotting of the results.

A.2 Artifact Check-list (Meta-information)

- **Algorithm:** JIT compilation with runtime optimizations.
- **Program:** Proteus software stack, NVIDIA Jitify, modified HeCBench programs.
- **Compilation:** Clang 17.0.5 and CUDA 12.2 for NVIDIA, ROCm 5.7.1 for AMD
- **Transformations:** Proteus LLVM plugin pass parses annotations and transforms program for JIT compilation, supported by Proteus's JIT runtime library.
- **Binary:** Driver script compiles different versions (Proteus configurations, Jitify for CUDA, or plain AOT compilation) of benchmark programs and generates binaries.
- **Data set:** HeCBench GPU benchmark programs, modified for Proteus/Jitify execution.
- **Run-time environment:** For NVIDIA, Linux OS (tested on RHEL 7.9), CUDA 12.2, Clang/LLVM 17.0.5, Python 3. For AMD, Linux OS (tested on RHEL 8.10), ROCm 5.7.1 including Clang/LLVM installation for AMD, Python 3. We provide an installation script that installs and uses conda to download software dependencies.
- **Hardware:** NVIDIA V100, AMD MI250X
- **Execution:** Driver script executes the experimentation workflow.
- **Metrics:** Key: end-to-end execution times and speedup of Proteus over AOT compilation. Derivative: compilation overhead with JIT extensions, runtime overhead of JIT compilation, GPU performance counters.
- **Output:** Key: Table of execution times and plots of speedup of Proteus over AOT compilation. Derivative: remaining plots on JIT compilation overheads, kernel-only execution times, and GPU performance counters.
- **Experiments:** Run HeCBench programs under different Proteus configuration, Jitify (CUDA), and AOT compilation.
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 6 hours (NVIDIA), 10 hours (AMD)
- **Publicly available?:** Yes, DOI: https://doi.org/10.5281/zenodo.14087063 (Zenodo), or https://github.com/Olympus-HPC/proteus.git branch `cgo25-artifact`
- **Code licenses (if publicly available)?:** Apache 2.0 w/ LLVM exceptions, BSD 3-Clause.

A.3 Description

A.3.1 How Delivered. All artifacts are available in DOI: https://doi.org/10.5281/zenodo.14087063 through Zenodo, or https://github.com/Olympus-HPC/proteus.git branch `cgo25-artifact`.

A.3.2 Hardware Dependencies. NVIDIA V100 GPU, AMD MI250X GPU

A.3.3 Software Dependencies. For NVIDIA, Linux OS (tested on RHEL 7.9), CUDA 12.2, Clang/LLVM 17.0.5, Jitify (included in the repo), Python 3. For AMD, Linux OS (tested on RHEL 8.10), ROCm 5.7.1, Python 3. We provide an installation script that uses conda to install dependencies.

A.3.4 Data Sets. Modified HeCBench suite, included in the repo.

A.4 Installation

Download and unzip from Zenodo: https://doi.org/10.5281/zenodo.14087063, or clone the repo and checkout the `cgo25-artifact` branch:

```
git clone --single-branch --depth 1 \
    --branch cgo25-artifact \
    https://github.com/Olympus-HPC/proteus.git
```

Installation assumes CUDA 12.2 is available and loaded in the environment for the NVIDIA GPU. Installation assumes ROCm 5.7.1 is available and loaded in the environment for the AMD GPU.

We provide a script that sets up the environment by installing conda in userspace to download and install dependencies (Python 3 and packages, Clang/LLVM for NVIDIA), and also to build Proteus for the GPU target machine (NVIDIA or AMD).

Change directory inside the artifact root path and source the script named `setup-proteus-env.sh` under the `buildscripts` directory:

```
# From git
cd proteus
# From Zenodo
cd Olympus-HPC-proteus-89a7563
source buildscripts/setup-proteus-env.sh
```

After the script executes, the user environment includes all software dependencies and a working installation of Proteus for the targeted GPU. The environment is ready to execute the experiment workflow as described in the next section.

A.5 Experiment Workflow

Change to the `benchmarks` directory inside the root path:

```
cd benchmarks
```

We provide separate scripts for the NVIDIA and AMD architectures that run all the respective experiments. The scripts invoke the experimentation driver (`driver.py`) to run the experiments.

For NVIDIA, run:

```
bash runscripts/run-all-nvidia.sh
```

For AMD, run:

```
bash runscripts/run-all-amd.sh
```

The measurements are collected in CSV files in the `benchmarks` directory, under a directory named `results`.

A.6 Evaluation and Expected Result

We provide a script that produces all the evaluation plots and tables in the paper. After collecting measurements for both NVIDIA and AMD run this script inside the `benchmarks` directory:

```
bash vis-scripts/plot-all.sh
```

The scripts will produce plots (in PDF format) and tables (in TEX format) under a directory named `plots`. The generated files start with the prefix of the corresponding figure or table (e.g., `figure-3-...`).

A.6.1 Key Results. End-to-end execution times of HeCBench benchmark programs for AOT and Proteus/Jitify JIT execution (Table 2) are expected to match, subject to measurement margins. Similarly, the end-to-end speedup graph (Figure 3) is expected to reproduce.

A.6.2 Derivative Results. Results on maximal cache size (Table 3) are expected to match. Subject to measurement margins, results on kernel-only execution times (Figure 4) are expected to match. Results on JIT compilation extensions and JIT compilation runtime overhead (Figures 5, 6) depend on the host system, nevertheless those trends should be observable. Results on GPU metrics (Figures 7, 8, 9, 10, 11) should reproduce on systems with the same GPU model.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://2025.cgo.org/track/cgo-2025-artifact-evaluation

References

[1] 2024. Proteus GitHub. https://github.com/Olympus-HPC/proteus.

[2] 2024. Top 500 List. https://www.top500.org/lists/top500/2024/06/.

[3] AMD. 2023. HIP RTC Programming Guide. https://rocm.docs.amd.com/projects/HIP/en/latest/user_guide/hip_rtc.html.

[4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2011. Adaptive optimization in the Jalapeno JVM. *SIGPLAN Not.* 46, 4 (may 2011), 65–83. doi:10.1145/1988042.1988048

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. doi:10.1137/141000671 arXiv:https://doi.org/10.1137/141000671

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. doi:10.1145/1565824.1565827

[7] Juan Manuel Martinez Caamaño and Serge Guelton. 2018. Easy::Jit: compiler assisted library to enable just-in-time compilation in C++ codes. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming* (Nice, France) *(Programming '18)*. Association for Computing Machinery, New York, NY, USA, 49–50. doi:10.1145/3191697.3191725

[8] Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael Franz. 2011. The impact of optional type information on jit compilation of dynamically typed languages. *SIGPLAN Not.* 47, 2 (oct 2011), 13–24. doi:10.1145/2168696.2047853

[9] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: a basic block versioning JIT compiler for CRuby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 25–32. doi:10.1145/3486606.3486781

[10] Charles Consel and François Noël. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 145–156. doi:10.1145/237721.237767

[11] Igor Costa, Péricles Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2013. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. doi:10.1109/CGO.2013.6495006

[12] Pierre Del Moral and Pierre Del Moral. 2004. *Feynman-kac formulae*. Springer.

[13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/2542142.2542143

[14] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 1996. 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 131–144. doi:10.1145/237721.237765

[15] Kavon Farvardin, H Finkel, M Kruse, and J Reppy. 2018. atJIT: A just-in-time autotuning compiler for C++. In *LLVM Developer's Meeting Technical Talk*.

[16] Hal Finkel, David Poliakoff, Jean-Sylvain Camier, and David F. Richards. 2019. ClangJIT: Enhancing C++ with Just-in-Time Compilation. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 82–95. doi:10.1109/P3HPC49587.2019.00013

[17] Yoshihiko Futamura. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–35.

[18] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler. *Higher Order Symbol. Comput.* 12, 4 (dec 1999), 381–391. doi:10.1023/A:1010095604496

[19] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. 1991. Essence of generalized partial computation. *Theoretical Computer Science* 90, 1 (1991), 61–79. doi:10.1016/0304-3975(91)90299-H

[20] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. doi:10.1145/1542476.1542528

[21] Tyler Gobran, João P. L. de Carvalho, and Christopher Barton. 2023. DASS: Dynamic Adaptive Sub-Target Specialization. In *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. 36–45. doi:10.1109/SBAC-PADW60351.2023.00016

[22] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. 2000. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science* 248, 1 (2000), 147–199. doi:10.1016/S0304-3975(00)00051-7 PEPM'97.

[23] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An evaluation of staged run-time optimizations in DyC. *SIGPLAN Not.* 34, 5 (may 1999), 293–304. doi:10.1145/301631.301683

[24] Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 239–250. doi:10.1145/2254064.2254094

[25] Michael Haidl and Sergei Gorlatch. 2014. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In *2014 LLVM Compiler Infrastructure in HPC*. 1–11. doi:10.1109/LLVM-HPC.2014.9

[26] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. PACXXv2 + RV: An LLVM-based Portable High-Performance Programming Model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (Denver, CO, USA) *(LLVM-HPC'17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. doi:10.1145/3148173.3148185

[27] Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. 2016. Multi-stage programming for GPUs in C++ using PACXX. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (Barcelona, Spain) *(GPGPU '16)*. Association for Computing Machinery, New York, NY, USA, 32–41. doi:10.1145/2884045.2884049

[28] Era Jain and Subhajit Roy. 2016. Phase Directed Compiler Optimizations. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 270–279. doi:10.1109/HiPC.2016.039

[29] Zheming Jin. [n. d.]. HeCBench. https://github.com/zjin-lcf/HeCBench/. Accessed: 2023-09-01.

[30] Neil D. Jones. 1996. An introduction to partial evaluation. *ACM Comput. Surv.* 28, 3 (sep 1996), 480–503. doi:10.1145/243439.243447

[31] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

[32] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. doi:10.1145/2833157.2833162

[33] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[34] Thibaut Lutz and Vinod Grover. 2014. LambdaJIT: a dynamic compiler for heterogeneous optimizations of STL algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Gothenburg, Sweden) *(FHPC '14)*. Association for Computing Machinery, New York, NY, USA, 99–108. doi:10.1145/2636228.2636233

[35] John Michalakes and Manish Vachharajani. 2008. GPU acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–7.

[36] Matthias Noack, Florian Wende, Georg Zitzlsberger, Michael Klemm, and Thomas Steinke. 2017. KART – A Runtime Compilation Library for Improving HPC Application Performance. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, Cham, 389–403.

[37] F. Noel, L. Hornof, C. Consel, and J.L. Lawall. 1998. Automatic, template-based run-time specialization: implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*. 132–142. doi:10.1109/ICCL.1998.674164

[38] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. 2013. JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim.* 10, 4, Article 59 (dec 2013), 25 pages. doi:10.1145/2541228.2555315

[39] NVIDIA. [n. d.]. Nvidia/Jitify: A single-header C++ library for simplifying the use of Cuda runtime compilation (NVRTC)., url=https://github.com/NVIDIA/jitify.

[40] NVIDIA. 2023. NVRTC. https://docs.nvidia.com/cuda/nvrtc/index.html

[41] NVIDIA. 2023. NVRTC Runtime Compilation Library. https://docs.nvidia.com/cuda/archive/12.2.0/.

[42] Konstantinos Parasyris, Giorgis Georgakoudis, Esteban Rangel, Ignacio Laguna, and Johannes Doerfert. 2023. Scalable Tuning of (OpenMP) GPU Applications via Kernel Record and Replay. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (, Denver, CO, USA,) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 28, 14 pages. doi:10.1145/3581784.3607098

[43] Armin Rigo. 2004. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Verona, Italy) *(PEPM '04)*. Association for Computing Machinery, New York, NY, USA, 15–26. doi:10.1145/1014007.1014010

[44] Gregory T. Sullivan. 2001. Dynamic Partial Evaluation. In *Programs as Data Objects*, Olivier Danvy and Andrzej Filinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–256.

[45] Shilei Tian, Joseph Huber, John Tramm, Barbara Chapman, and Johannes Doerfert. 2022. Just-in-Time Compilation and Link-Time Optimization for OpenMP Target Offloading/. In *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, Michael Klemm, Bronis R. de Supinski, Jannis Klinkenberg, and Brandon Neth (Eds.). Springer International Publishing, Cham, 145–158.

[46] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[47] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and evaluating transient gradual typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 28–41. doi:10.1145/3359619.3359742

[48] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.* 52, 6 (jun 2017), 662–676. doi:10.1145/3140587.3062381

[49] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. doi:10.1145/2509578.2509581

Received 2024-09-12; accepted 2024-11-04