LEAP: Lifting Legacy x86-64 Binaries to Recompilable IR for Automatic Parallelization

Vahram Martirosyan*
Department of CIS
Higher Colleges of Technology
Abu Dhabi, UAE
Email: vmartirosyan@hct.ac.ae
*Corresponding author

***Abstract*—Binary lifting enables analysis of software whose source is unavailable, but existing lifters target decompilation rather than *recompilation*. We present LEAP, a toolchain that lifts an x86-64 ELF binary to LLVM IR, recovers parallelizable loop patterns from the type-erased IR, transforms them into parallel constructs, and recompiles to a working parallel binary. LEAP supports 89 opcode classes covering the full instruction stream of all evaluated benchmarks, including SSE/AVX, bit manipulation, string operations, indirect control flow, and exception handling. The central challenge is that lifting erases source-level structure: pointers become `i64` values, reduction accumulators live on an emulated stack, and loop bounds are hidden behind store/load pairs. We describe three recovery techniques—pointer-argument recovery, backward data-flow tracing through stack memory, and exhaustive return-value load collection—that make reduction detection robust on lifted IR. On a 20-benchmark suite and the NAS Parallel Benchmarks, source-IR parallelization achieves $1.9\times$–$8.3\times$ (geometric mean $3.7\times$) over a `-O2` baseline on an 8-core CPU. The uplifted binary path, with no source access, achieves $3.8\times$–$27.2\times$ (geometric mean $10.4\times$) over its `-O0` input and remains $2.2\times$ faster (geometric mean) than the same program compiled at `-O2`, outperforming `-O2` on five of seven applications; GPU offload reaches $958\times$ for compute-bound kernels. Lifting is validated by round-trip tests that recompile and compare program output.**

***Index Terms*—binary lifting, automatic parallelization, LLVM, legacy binaries, program transformation**

I. Introduction

Billions of lines of legacy C/C++ were compiled before multi-core processors became ubiquitous, and much of this code runs sequentially, leaving hardware parallelism untapped. Recompiling with a modern parallelizing compiler is the preferred remedy, but source is often unavailable, proprietary, or bound to obsolete toolchains.

Binary lifting—translating machine code to a higher-level intermediate representation (IR)—enables analysis and transformation without source access. Combined with automatic parallelization, lifted binaries could be retargeted to multi-core hardware. However, the two research threads have largely remained separate: recent binary lifters and decompilers target analysis, patching, and reverse engineering [8], [10], [11] rather than performance-oriented recompilation, while automatic parallelizers operate on source-level IR and cannot ingest binaries. Systems that transform binaries dynamically incur runtime instrumentation overhead and require a heavy-weight framework [6], [7].

We bridge this gap with LEAP, a toolchain that lifts an x86-64 ELF binary to LLVM IR while preserving execution semantics, detects parallelizable loop patterns, transforms sequential loops into parallel constructs targeting a thread-pool or GPU runtime, and recompiles the result to a working binary. The problem is one of *retargeting*: an already-deployed program must be re-engineered to exploit hardware it was never compiled for, using only its binary as input.

The central difficulty is that lifting *erases* the structure that makes source-level parallelization tractable. In compiler IR a reduction accumulator is an SSA ϕ node, pointers carry types, and loop bounds are direct comparisons; in lifted IR every pointer is a flattened `i64` reached through `inttoptr`, accumulators live on an explicitly modeled software stack, and bounds must be recovered from store/load pairs. Naïve matching therefore misses most realistic loops. Our core contribution is a set of *semantic recovery* techniques that reconstruct these facts, enabling pattern-driven parallelization on binaries.

Our contributions are: (i) a static binary lifter producing *recompilable* LLVM IR from x86-64 ELF binaries, supporting 89 opcode classes that cover the complete static instruction stream of every benchmark evaluated (unsupported instructions cause a fail-safe abort rather than silent miscompilation); (ii) a pattern-driven parallelizer recognizing element-wise, reduction, nested-reduction, matrix-multiply, stencil, histogram, search, and sort patterns, with conservative rejection of loops whose cross-iteration dependencies cannot be disproven; (iii) three lifted-IR recovery techniques—*pointer-argument recovery*, *backward data-flow tracing* through the emulated stack, and *exhaustive return-value load collection*—that recover reduction semantics from flattened, stack-indirected code (Section IV); (iv) a GPU offload path that synthesizes OpenCL kernels (including integer reductions) from lifted IR and selects the CPU or GPU with a *statically estimated, intensity-aware* policy (Section IV-A); and (v) an end-to-end evaluation on 20 benchmarks plus the NAS Parallel Benchmarks, against three independent baselines (Clang, Intel `-xhost`, GCC auto-parallelization), together with an ablation, a failure-mode analysis, and a threats-to-validity discussion.

II. Background and Related Work

Binary lifting translates machine code to a higher-level IR for analysis and transformation. Recent lifters recover LLVM IR from GPU assembly [2] and 64-bit ARM machine code [3], reconstruct source-level type information from debug metadata [4], and formally verify the lifting of programs to a re-targetable IR [5]. Binary rewriting and recompilation systems patch or reassemble deployed binaries for security and correctness [1], [6], [7], and interprocedural and symbolic analysis platforms build on lifted IR [8], [9]. A recurring limitation is that these systems prioritize *analysis*, verification, or patching over *recompiling for performance*: the produced IR is hard to compile back to optimized machine code because of unstructured control flow, indirect jumps, and imprecise memory modeling. LEAP's lifter is designed from the outset for recompilability—it models CPU state explicitly through LLVM allocas, preserves the original stack layout, resolves position-independent references through section globals, and lowers every construct to instructions accepted by the standard LLVM backend [25]. The LLVM MC disassembler [25] provides instruction decoding.

Decompilation has been reshaped by machine learning, with neural and large-language-model decompilers recovering C-like code [10]–[12]. Automatic parallelization has advanced along several axes: MLIR-based polyhedral frameworks [14], automatic loop-nest normalization and scheduling [15], reinforcement-learning loop transformations [16], transformer-guided OpenMP parallelization [18], and barrier-free pipelining of irregular nested loops [17]. All of these operate on source or compiler-generated IR. Proving independence remains the crux, and alias and pointer analyses for LLVM and for binaries continue to improve [13], [20], [21]. LEAP operates on *lifted binary* IR, where the type and control-flow structure these analyses rely upon has been erased; its pattern-driven approach [26] is deliberately more conservative than dependence-based analysis for such irregular inputs, at the cost of being less sophisticated on affine nests. GPU and heterogeneous compilation has also matured, with OpenCL/SYCL tooling and ML-guided kernel compilation [22]–[24]. To our knowledge, no prior system combines static binary lifting with automatic loop parallelization to produce a standalone, recompiled, parallel binary.

III. System Architecture

LEAP consists of four stages: (1) static binary lifting to LLVM IR, (2) IR normalization, (3) pattern-driven parallelization, and (4) recompilation (Figure III). The same parallelizer also accepts compiler-generated IR directly (the *source-IR* path), which isolates the effect of lifting from that of parallelization.

Binary lifter. The lifter decodes each executable section with the LLVM MC disassembler, recovering function boundaries from ELF symbols and EH_FRAME data and reconstructing a control-flow graph including indirect jump tables. Each function is lowered to a flat alloca-based CPU-state model (16 general-purpose registers as `i64`, 16 XMM registers as `<2 x double>`, condition flags, and a software stack), one LLVM instruction sequence per machine instruction. Data sections become LLVM globals so position-independent references resolve through section globals rather than raw addresses; indirect calls and jumps become `indirectbr` or computed calls with targets recovered from ELF relocations; and exceptions are preserved by parsing the `.gcc_except_table` LSDA and emitting `invoke`/`landingpad`. The 89 supported opcode classes (∼130 mnemonics)—data movement, integer/floating-point arithmetic, logic and shifts, SSE/AVX, bit manipulation, string operations, indirect control flow, and exceptions—are not full x86-64 coverage (the ISA defines over a thousand mnemonics), but rather the subset emitted by unoptimized compiler output, the input mode we target. Coverage is operational: a binary lifts only if every instruction translates; all 20 benchmarks lift, and unsupported instructions (AES-NI, SHA, gather/scatter) abort rather than miscompile.

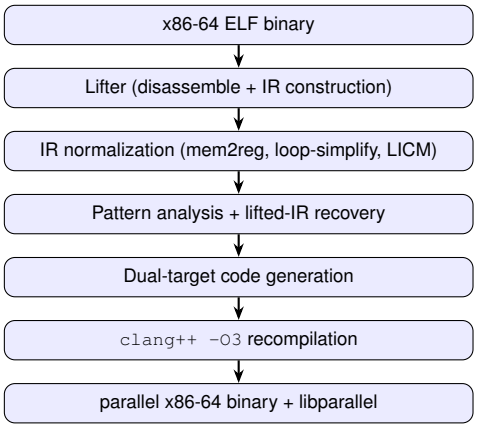


Fig. 1. End-to-end pipeline. The optional source-IR path enters at the analysis stage.

Parallelizer. The parallelizer takes normalized IR—from the lifter or the compiler front end—and identifies parallelizable loops by structural pattern matching rather than dependence analysis, which is more robust when types are flattened. It recognizes eight patterns: *element-wise* (disjoint indexed accesses verified linear in the induction variable), *reduction* (the characteristic ϕ–binop–ϕ cycle over an associative operation), *nested reduction* (as in FFT stages and matrix inner products), *matrix multiply* ($C[i][j]$ += $A[i][k]B[k][j]$), *stencil* (affine neighbor accesses), *histogram* (scatter-update with atomic bins), *search* (partitioned space with early termination), and *sort* (bitonic merge networks). Loops that match none, or whose dependencies cannot be disproven, remain sequential. When a lifted function is parallelized, the original body stays in the binary and a symbol-weakening pass (`objcopy --redefine-syms`) redirects calls to the parallel wrapper, allowing selective parallelization without disturbing the rest of the binary.

GPU offload and runtime. The GPU path translates parallelizable loop bodies to OpenCL C [24] and compiles them to SPIR-V. Its `OpenCLKernelGenerator` performs

AST reconstruction: it walks the LLVM use-def chain of the loop body recursively and re-synthesizes structured expressions (e.g., reconstructing `out[i]=a*in[i]+b` from its `fmul`/`fadd`/`getelementptr` instructions), which is robust for non-affine math chains but expects a pure data-flow DAG and rejects bodies with arbitrary control flow. Floating-point intrinsics map to OpenCL built-ins, and multi-dimensional accesses are reconstructed from flat `i64` pointers and strides; the kernel is compiled to SPIR-V at parallelization time and embedded in the output module, so runtime loading incurs no compilation cost. The runtime (`libparallel.a`) provides a work-stealing thread pool, `parallel_for_range`, specialized reductions with tree-based merging, and an optional OpenCL backend with buffer pooling and kernel caching.

IV. Design: Recovering Parallelism from Lifted IR

A source-level reduction such as `for (i=0;i<n;++i) s += a[i];` appears in compiler-generated IR as typed pointers, an SSA ϕ accumulator, and a direct `icmp slt i32 %i, %n`. Lifting erases all three facts: the array and the bound n become `i64` arguments, the bound is first stored to a software-stack slot and later reloaded to form the `icmp`, the accumulator is a stack slot updated by a `load`→`fadd`→`store` chain rather than a ϕ, and the function returns a load from that slot, often masked by intervening arithmetic. A naïve source-IR matcher therefore misses the majority of realistic loops. We reconstruct the three missing facts as follows.

Pointer-argument recovery. Once lifting flattens pointers and integers to `i64`, they are type-indistinguishable, so we recover pointer-ness structurally: an `i64` parameter is *pointer-like* if it can reach a memory address—i.e., it feeds an `inttoptr`, directly or through one level of integer arithmetic. The rule is a *may* analysis and over-approximates: arithmetic values that merely happen to feed an `inttoptr`, and parameters used both as scalars and as base addresses (dual use), are treated as pointer-like. This is safe because the classification is only an *enabler*—it lets the dependency check reason about base addresses and strides, as binary-level pointer analyses do [13], [20]—but the check still refuses to parallelize unless it can *prove* disjointness. A false positive only makes the analysis stricter (a loop may stay sequential—a performance, never a correctness, loss); a false negative is caught by the separate cross-iteration check. The one-level depth bound (widenable in future work) may miss deep chains and cause a safe rejection; type-recovery work for LLVM IR [19] is complementary.

Backward data-flow tracing. To confirm that a loop bound is parameter-derived (and not, say, pointer-dependent), we walk backward from each `icmp` operand to an argument. The subtlety is that the argument is stored to a stack slot and later reloaded, breaking the chain with a memory round-trip; when the walk encounters a load from a constant stack offset it locates the store to the same offset and continues from the stored value. The traversal follows casts, ϕ nodes, and arithmetic operators, with a visited set and a depth bound guaranteeing termination, and accepts a bound only if at least one `icmp` operand reaches an argument.

Return-value load collection. The most impactful pass recovers the accumulator. In lifted IR the function returns the accumulator by loading it from a fixed stack offset relative to the emulated stack pointer `rsp.raw`. We first collect *all* loads reachable from the return value through casts, vector inserts/extracts, and arithmetic chains—the arithmetic traversal is essential because compilers frequently mask an accumulator behind a constant multiply or divide (e.g., returning `sum/n` or `sum*scale`). For each load we extract the constant offset o of the form `inttoptr(add(rsp.raw,o))`, and accept it as a true accumulator only if the candidate loop contains a store to o whose value is an `fadd`/`add` of a load from o with a non-constant operand—excluding counter increments and unrelated values that happen to share the return offset. A companion rewriter replaces post-loop loads from the ephemeral accumulator slot with loads from the thread-shared alloca, so the merged total is returned rather than the last chunk's partial sum.

Dependency rejection. Safety is preserved by a conservative check: for each candidate loop we require that stored addresses derive solely from the induction variable, traced through recovered pointer arguments; if a location may be reached by more than one iteration and aliasing cannot be ruled out, the loop is marked NOT PARALLELIZED. This fail-safe orientation is validated by negative tests—a synthetic lifted function `step(i64 ptr,...)` and the prefix sum are correctly rejected—while time-stepping stencils that call such a function internally (`heat3d_step`) are still parallelized, so LEAP never trades correctness for speed. A complementary *memory-model guard* refuses to parallelize any function containing an atomic read-modify-write, compare-exchange, fence, or atomic load/store: lifting preserves instruction-level semantics but not the C++ memory model, so lock-free code is left sequential rather than miscompiled.

A. Backend Selection

The runtime selects between the CPU thread pool and the GPU using both problem size and a *static arithmetic-intensity* estimate. During analysis we count floating-point operations and memory traffic in each loop body to obtain an intensity in FLOPs per byte; the value is propagated into the generated GPU descriptor, registered with the runtime, and consulted by the back-end selector, which keeps any operation below a break-even intensity (4 FLOPs/byte) on the CPU regardless of problem size. A per-task minimum of 10 elements amortizes thread wake-up ($\sim$5 µs). The policy is grounded in a sensitivity study on a memory-bound vector kernel (1K–1M elements, CPU-only, GPU-always, and adaptive policies): GPU offload is never profitable for it—at 5K elements GPU-always is $1.9\times$ *slower*—and size alone is insufficient, because even at 1M elements the GPU loses: the arithmetic intensity is too low to amortize transfer and launch. The intensity-aware selector classifies this kernel as memory-bound and never offloads it, while compute-heavy kernels (thousands of FLOPs per byte)

are routed to the GPU and achieve three-order-of-magnitude gains.

V. Experimental Methodology

We evaluate on 20 benchmarks in three groups. **Scientific applications (7):** integer sum reduction (20M elements); Monte Carlo π (150M samples); 2D reservoir simulation (2000^2 grid, 50 steps); K-means (1M points, 16 clusters, 8 dimensions, 100 iterations); Monte Carlo Asian option pricing (150M paths); FFT (16M complex samples); and 3D heat diffusion (256^3, 10 steps). **PolyBench kernels (6):** GEMM, SYRK, 2MM, 3MM, SYR2K, and TRMM at LARGE_DATASET ($N = 2000$); a full 15-kernel PolyBench sweep supports the failure analysis. **Heavy synthetic workloads (7):** four CPU kernels (2M–25M elements, 100–200 iterations per element) and three GPU kernels (1M elements, thousands of FLOPs per element).

All experiments run on an Intel Core i7-13700H (8 P-cores, 16 threads), 32 GB DDR5 RAM, Ubuntu 24.04, GCC 13.3, Clang 18.1, and Intel oneAPI DPC++/C++ 2026.0.0. Each benchmark runs at least 20 times; we report median wall-clock runtime, excluding process startup. The longer-running NPB kernels (Table VI-B) are an exception: a single run takes seconds to minutes, so we report the best of three runs to suppress occasional system interference, whereas the shorter suites use the median of $\geq$20 runs because repetition is cheap. Both are robust central estimates measured on the same machine and toolchain, and every reported speedup is accompanied by the absolute runtimes from which it is derived, so any residual run-to-run variance is transparent. We compare against `clang++ -O2` (Sequential), Intel `icpx -O3 -xhost` (AVX2/FMA and aggressive vectorization), and GCC auto-parallelization (`g++ -O2 -ftree-parallelize-loops=4 -fopenmp`), all on the same hardware. Because aggressive optimization destroys the canonical loop structure the matcher relies on, the uplifted path necessarily begins from an unoptimized (`-O0`) binary. This yields two comparisons, both reported: speedup *relative to the `-O0` input* (isolating the value added to a given artifact) and speedup *relative to the `-O2` source baseline* (the fair comparison to the source-IR path). The `-O0` requirement applies only to IR generation; the transformed IR is compiled at `-O3 -march=native`, so the emitted binary is fully optimized and the only cost is the optimization debt the lifter must recover.

On the `-O0` input. LEAP targets binaries with canonical loop structure, and unoptimized output is the canonical case. Optimization destroys the affordances the matcher uses: unrolling and software pipelining remove the induction-variable form, strength reduction rewrites array indexing, inlining fuses callee bodies, vectorization packs scalar bodies, and scheduling scrambles the stack-slot storage our tracing depends on. This matches deployed legacy code, which is often built without optimization—default `-O0` toolchains, debug builds, or build systems that never enable the optimizing flags—and whose source or toolchain is frequently unavailable, so the binary is the only artifact. The restriction is analysis-only: the emitted binary is `-O3 -march=native`, and the parallelizer also accepts optimized IR on its source-IR path. Quantifying the boundary, the seven scientific applications are parallelized from `-O0`/`-O1`/`-O2` binaries six/five/four times (and one `-O2` binary fails to lift), confirming degradation as optimization erases loop cues (Table V).

TABLE I
Detection rate vs. input optimization level for the seven scientific applications.

Input	Parallelized	Sequential	Lift fail
`-O0`	6	1	0
`-O1`	5	2	0
`-O2`	4	2	1

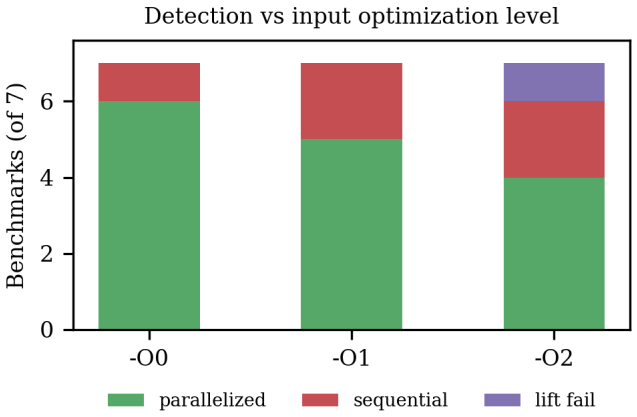


Fig. 2. Parallelized benchmarks vs. input optimization level.

VI. Results

A. Lab and PolyBench Benchmarks

Table VI-A reports the scientific applications. **Source-IR** parallelization achieves consistent speedups on all seven, from 1.9× (heat3d) to 8.3× (fft), geometric mean 3.7×; the largest gains come from compute-heavy benchmarks with ample parallelism, while bandwidth-bound benchmarks (simple, heat3d) gain less. **Uplifted vs. input** achieves 3.8×–27.2× over `-O0` (geometric mean 10.4×), with reservoir_sim (27.2×) and heat3d (18.0×) benefiting most because their stack-based accumulator patterns—previously opaque—are now recovered by the backward-tracing and return-load passes. **Uplifted vs. `-O2`** is faster on five of seven applications (geometric mean 2.2×, up to 6.4× on fft and 5.8× on option_pricing); it loses on simple (0.80×), a bandwidth-bound sum whose `-O2` baseline is already vectorized and near memory-bandwidth-bound, and on kmeans (0.69×). Figure VI-A visualizes the per-application comparison relative to the `-O2` baseline.

On PolyBench at $N = 2000$ (Table VI-A), LEAP achieves 1.51×–7.38× (geometric mean 2.66×), winning four of six kernels. Intel's `-xhost` is strong on GEMM (5.37×) and

TABLE II
RUNTIME (S) AND SPEEDUP FOR SOURCE-IR AND UPLIFTED-BINARY PARALLELIZATION. **UPLIFT/O2** IS THE FAIR COMPARISON TO THE SOURCE-IR PATH.

Benchmark	**Seq-O2 (s)**	**Seq-O0 (s)**	**Src-IR Par (s)**	**Uplift Par (s)**	**Src-IR /O2**	**Uplift /O0**	**Uplift /O2**
simple	0.004	0.019	0.002	0.005	2.0×	3.8×	0.80×
pi	0.167	0.442	0.026	0.050	6.4×	8.8×	3.3×
reservoir_sim	0.219	2.394	0.081	0.088	2.7×	27.2×	2.5×
kmeans	3.228	32.429	0.793	4.698	4.1×	6.9×	0.69×
option_pricing	2.185	4.015	0.549	0.374	4.0×	10.7×	5.8×
fft	14.497	24.764	1.753	2.280	8.3×	10.9×	6.4×
heat3d	0.187	2.082	0.099	0.116	1.9×	18.0×	1.6×
Geometric mean:	Src-IR/O2: **3.7**×		Uplift/O0: **10.4**×		Uplift/O2: **2.2**×		

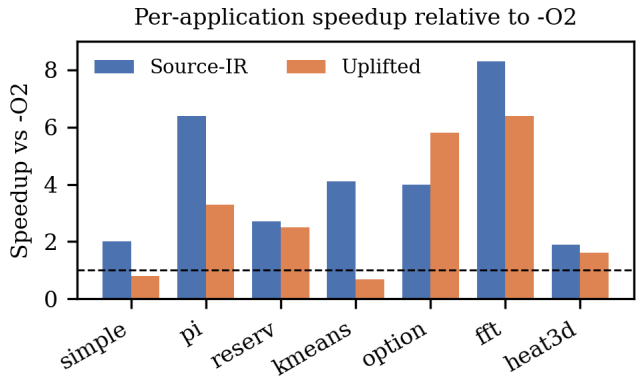


Fig. 3. Per-application speedup relative to the `-O2` baseline: source-IR and uplifted paths.

TABLE III
POLYBENCH AT LARGE_DATASET ($N = 2000$): SPEEDUP RELATIVE TO CLANG.

Kernel	**Intel**	**GCC-par**	**LEAP**
GEMM	5.37×	0.97×	**7.38**×
SYRK	1.00×	1.02×	**3.04**×
2MM	**5.86**×	1.03×	1.89×
3MM	**3.87**×	1.07×	1.52×
SYR2K	0.97×	1.04×	**2.58**×
TRMM	1.00×	1.04×	**2.14**×
Geo. mean	2.21×	1.03×	**2.66**×

2MM (5.86×) through AVX2/FMA and BLAS substitution but negligible on SYRK, SYR2K, and TRMM (0.96×–1.00×); GCC's auto-parallelizer exploits little meaningful parallelism at this size (0.97×–1.07×). LEAP's geometric mean (2.66×) exceeds Intel's (2.21×) and GCC's (1.03×).

Heavy workloads and GPU offload. The heavy suite exercises the system at scale. The four CPU kernels average 8.4× across 2M–25M element workloads, and the three compute-bound GPU kernels average 735× (geometric mean) with a peak of 958× over the single-threaded `-O2` CPU baseline. The GPU is the integrated Intel Iris Xe graphics (device `0x7d51`, 128 execution units) sharing unified memory with the Intel Core i7-13700H host; because host and device share physical memory and the runtime uses pinned, zero-copy buffers, data movement is negligible. Table VI-A reports the per-kernel breakdown: input setup and device-to-host readback together account for 0.12% of runtime and kernel compute for 99.87–99.88%. This is why compute-bound kernels scale almost perfectly, and why the GPU path is selected only for arithmetic-heavy loops (Section IV-A); on a discrete GPU with PCIe, transfer would dominate memory-bound kernels, so the size threshold is necessary but not sufficient.

TABLE IV
GPU TRANSFER VS. COMPUTE BREAKDOWN (INTEL IRIS XE IGPU, 128 EUS, UNIFIED MEMORY; 1M ELEMENTS, AVERAGED OVER 5 RUNS). COMPUTE/TIME IN MS UNLESS NOTED.

Kernel	**H2D (µs)**	**Compute**	**D2H (µs)**	**Total**
GPUHeavy	0	15.91	20	15.93
GPUMath	1	14.51	16	14.53
GPUVector	1	5.76	6	5.76
Compute share: 99.87–99.88%				

Failure modes. A parallelizer is defined as much by what it rejects. Across a full 15-kernel PolyBench sweep, ten kernels were parallelized profitably and five were not. The scatter-reduction kernels (bicg, atax) and prefix sum are *correctness successes*: the conservative dependency check refused them, preventing races (scatter stores indexed only by the inner IV race across outer iterations; prefix sum is loop-carried). The 2mm/3mm (region overhead of many small parallel regions) and doitgen (temp-array reuse without privatization) cases are *performance* failures exposing missing optimizations—parallel-region coalescing and array privatization—that bound the profitable regime.

Correctness. The parallelizer is validated by 273 unit tests (lifter, patterns, runtime, type/name analysis) and 48 end-to-end round-trip tests (lift, parallelize, recompile, compare output), plus pattern/speedup, heavy, and regression tests; all pass. GPU floating-point results are verified below 10^{-3} relative error, and ASAN/UBSAN report no violations. Two negative tests assert that prefix sum and a synthetic lifted cross-

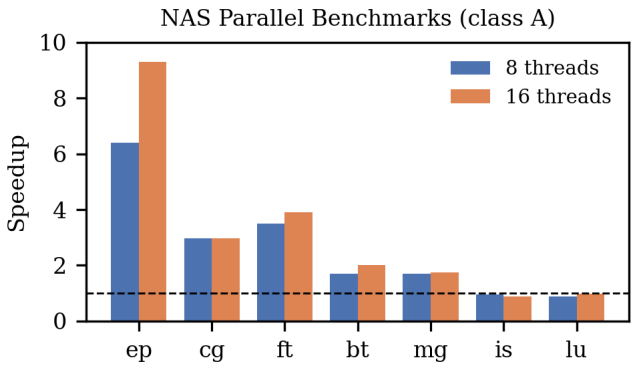


Fig. 4. NPB class A speedup at 8 and 16 threads.

TABLE V
NPB 3.0 CLASS A. BOTH BUILDS ARE LIFTED AND COMPILED AT `-O2`; SPEEDUPS ARE PARALLEL OVER THE SAME SEQUENTIAL RUN.

Kernel	Seq (s)	Par 8t (s)	Sp. 8t	Par 16t (s)	Sp. 16t
ep	9.47	1.48	6.4×	1.02	9.3×
cg	1.57	0.53	2.96×	0.53	2.96×
ft	3.92	1.11	3.5×	1.00	3.9×
bt	4.49	2.60	1.7×	2.25	2.0×
mg	0.58	0.34	1.70×	0.33	1.75×
is	5.13	5.34	0.96×	5.76	0.89×
lu	3.61	4.10	0.88×	3.65	0.99×

iteration dependency are *not* parallelized, guarding against unsafe over-parallelization.

B. Additional Evaluation: NAS Parallel Benchmarks

To test LEAP on a widely used, independently authored suite, we evaluate the NAS Parallel Benchmarks (NPB) 3.0 at class A. Each kernel is compiled at `-O0` and lifted; the sequential and parallel builds are then recompiled at `-O2` and timed on 8 and 16 threads, so the speedup isolates parallelization exactly as in Table VI-A. Table VI-B reports the results (best of three runs). EP (embarrassingly parallel) scales to 9.3×; CG (conjugate gradient) reaches 2.96×; FT and BT/MG show 2–4×; IS and LU remain near-sequential (0.89×, 0.99×) because IS is dominated by an inherently serial random-number generator (`create_seq`/`randlc`, ∼77% of runtime) and LU by triangular solves. Figure VI-B shows the per-kernel scaling at 8 and 16 threads. All kernels verify.

VII. Discussion

Strengths and limitations. LEAP's primary strength is parallelizing pre-compiled binaries without source access or annotations. It is most effective for programs with large, compute-intensive loops (seconds to minutes sequential) where dispatch overhead is amortized; for irregular patterns (indirect indexing, stack-based induction variables, nested reductions) that confound dependence-based parallelizers; and for legacy binaries whose toolchains are obsolete. Limitations include: (i) **instruction coverage**—89 opcode classes are supported and unsupported instructions (AES-NI, SHA, FMA, gather/scatter) abort lifting; (ii) **small problems**—benchmarks under ∼100 ms show limited or negative speedup due to runtime initialization and dispatch; (iii) **dynamic libraries**—the lifter processes a single ELF executable and does not intercept `dlopen`; (iv) **memory model**—instruction-level semantics are preserved but the C++ memory model (atomics, acquire/release) is not; a guard refuses to parallelize functions that contain atomic operations or fences, so lock-free code is left sequential rather than miscompiled; (v) **GPU coverage**—the offload path targets simple, single-input reductions and element-wise kernels, and GPU memory capacity bounds problem size; (vi) **portability**—only x86-64 ELF is supported; (vii) **optimization debt**—starting from `-O0` means the lifter must recover optimizations the source path never lost, which is why uplifted binaries trail source-IR parallel builds in absolute terms even when they beat `-O2`.

Threats to validity. The dominant internal threat is measurement methodology; we fix the baseline at `-O2` for both paths, report the uplifted-vs-`-O2` column, take medians of at least 20 runs, and exclude process startup. The `-O0` input is a deliberate design choice, and we report the resulting debt explicitly. Externally, all benchmarks are custom scientific applications or PolyBench/NPB kernels (no SPEC CPU, Parboil, or Rodinia), and all measurements come from one machine (one Intel CPU with integrated GPU), so results characterize the approach on realistic but not standardized workloads. Speedup is the sequential/parallel wall-clock ratio, which for memory-bound kernels can understate cache benefits and overstate reduced work; we therefore report absolute runtimes alongside every ratio.

Comparison with prior work. Compared to modern polyhedral and MLIR-based parallelizers [14], LEAP's pattern-driven approach is less sophisticated on affine nests but more robust on irregular patterns; polyhedral representations are hard to recover from lifted IR, whereas structural matching works directly on the flattened IR. Compared to compiler auto-parallelization [18], LEAP is more conservative on regular loops but covers irregular patterns and operates on binaries. Compared to Intel `-O3 -xhost`, LEAP demonstrates the value of thread-level parallelism where vectorization cannot help (irregular access, nested reductions, branchy bodies), while Intel wins on regular affine nests. Compared to dynamic binary-rewriting systems [6], [7], LEAP incurs no instrumentation overhead but cannot exploit dynamic information or handle self-modifying code.

VIII. Conclusion

We presented LEAP, a toolchain that combines static binary lifting with automatic loop parallelization to accelerate legacy x86-64 binaries without source access. The central technical result is that pattern-driven parallelization can be made to work on *lifted* IR—where types, loop bounds, and accumulators have been erased—by explicitly recovering pointer arguments,

tracing data flow backward through the emulated stack, and exhaustively collecting return-value loads. With these techniques, uplifted binaries achieve a geometric mean of $10.4\times$ over their `-O0` input and remain $2.2\times$ faster than the same program at `-O2`, outperforming `-O2` on five of seven applications; source-IR parallelization achieves $3.7\times$ over `-O2`; heavy CPU workloads average $8.4\times$; NPB EP scales to $9.3\times$; and GPU offload reaches $958\times$ for compute-bound kernels.

The results also delineate the approach's boundaries. Uplifted binaries still trail source-IR parallel builds in absolute terms because they must recover optimization debt; memory-bound kernels are correctly kept off the GPU but still see limited gain; and support for array privatization, parallel-region coalescing, and scatter-reductions is needed to extend coverage to further PolyBench kernels. Future work includes a JIT mode that avoids recompilation, polyhedral tiling of lifted IR, and a lightweight loader that intercepts `dlopen`.

Declaration of AI and AI-assisted technologies in the writing process

During the preparation of this work the author used a generative AI assistant (DeepSeek, a large language model) in order to improve the readability and language of the manuscript. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

References

[1] H. Dai *et al.*, "Scribe: Practical static binary patching via binary-aware recompilation of decompiled code," in *Proc. IEEE EuroS&P*, 2026. doi: 10.1109/EuroSP68448.2026.00079.

[2] J. Wan, L. Tan, and D. Tian, "NVLift: Lifting NVIDIA GPU assembly to LLVM IR," in *Proc. Workshop on Binary Analysis Research (BAR)*, 2026. doi: 10.14722/bar.2026.23028.

[3] S. Nagao, K. Ootsu, and T. Yokota, "A translation method from 64-bit ARM machine code to LLVM IR," in *Proc. CANDARW*, 2024. doi: 10.1109/CANDARW64572.2024.00064.

[4] A. Hück, S. Kreutzer, and C. Bischof, "llvm-dimeta: Extracting source-level type information in LLVM IR," in *Proc. IEEE SCAM*, 2025. doi: 10.1109/SCAM67354.2025.00019.

[5] N. Naus *et al.*, "Formally verified binary lifting to P-Code," in *Proc. ACM CCS*, 2024. doi: 10.1145/3658644.3691386.

[6] X. Meng and W. Liu, "Incremental CFG patching for binary rewriting," in *Proc. ACM ASPLOS*, 2021. doi: 10.1145/3445814.3446765.

[7] K. Lei *et al.*, "BiRFIA: Selective binary rewriting for function interception on ARM," in *Proc. IEEE/ACM IWQoS*, 2023. doi: 10.1145/3577193.3593701.

[8] M. Revelle, M. Parker, and K. Orr, "Blaze: A framework for interprocedural binary analysis," in *Proc. BAR*, 2023. doi: 10.14722/bar.2023.23009.

[9] D. de Ruck *et al.*, "dAngr: Lifting software debugging to a symbolic level," in *Proc. BAR*, 2025. doi: 10.14722/bar.2025.23014.

[10] H. Tan *et al.*, "LLM4Decompile: Decompiling binary code with large language models," in *Proc. EMNLP*, 2024. doi: 10.18653/v1/2024.emnlp-main.203.

[11] I. Hosseini and B. Dolan-Gavitt, "Beyond the C: Retargetable decompilation using neural machine translation," in *Proc. BAR*, 2022. doi: 10.14722/bar.2022.23009.

[12] D. Cotroneo *et al.*, "Can neural decompilation assist vulnerability prediction on binary code?," in *Proc. ACM SAC*, 2025. doi: 10.1145/3722041.3723097.

[13] S. Kim *et al.*, "BinPointer: Precise, sound, and scalable binary-level pointer analysis," in *Proc. Int. Conf. Compiler Construction (CC)*, 2022. doi: 10.1145/3497776.3517776.

[14] J. Zhao *et al.*, "PolyMorphous: An MLIR-based polyhedral compiler with loop transformation primitives," in *Proc. IEEE IPDPS*, 2025. doi: 10.1109/IPDPS64566.2025.00041.

[15] L. Trümper *et al.*, "A priori loop nest normalization: Automatic loop scheduling," in *Proc. PACT*, 2025. doi: 10.1145/3696443.3708951.

[16] S. Jain *et al.*, "Reinforcement learning assisted loop distribution for locality and vectorization," in *Proc. LLVM-HPC*, 2022. doi: 10.1109/LLVM-HPC56686.2022.00006.

[17] S. Yun *et al.*, "Selene: Barrier-free pipelining for irregular nested loops," in *Proc. IEEE/ACM CGO*, 2026. doi: 10.1109/CGO68049.2026.11395206.

[18] T. Kadosh *et al.*, "Advising OpenMP parallelization via a graph-based approach with transformers," in *Proc. IWOMP*, 2023. doi: 10.1007/978-3-031-40744-4_1.

[19] N. Nicolosi *et al.*, "Type deduction analysis: Reconstructing transparent pointer types in LLVM-IR," in *Proc. ACM SPLASH*, 2026. doi: 10.1145/3771775.3786268.

[20] K. Chitre, P. Kedia, and R. Purandare, "HORIZON: Estimating alias analysis precision bounds," in *Proc. ACM SPLASH*, 2026. doi: 10.1145/3771775.3786270.

[21] K. Chitre, P. Kedia, and R. Purandare, "The road not taken: Alias-analysis-based optimizations missed by the compiler," *Proc. ACM Program. Lang. (OOPSLA)*, 2022. doi: 10.1145/3563316.

[22] W. He *et al.*, "Profile guided optimization transfer-learning for OpenCL/SYCL kernels," in *Proc. IWOCL*, 2023. doi: 10.1145/3585341.3585359.

[23] W. He, Y. Zou, and F. Zou, "Machine learning for vectorization decision in OpenCL/SYCL kernels," in *Proc. IWOCL*, 2023. doi: 10.1145/3585341.3585364.

[24] P. Thoman *et al.*, "On the compilation performance of current SYCL implementations," in *Proc. IWOCL*, 2022. doi: 10.1145/3529538.3529548.

[25] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. IEEE/ACM CGO*, 2004. doi: 10.1109/CGO.2004.1281665.

[26] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001. https://openlibrary.org/isbn/9781558602861.