FarSlayer: Turnkey Acceleration of Legacy Software on Commodity FPGA Cards

Esmerald Aliaj
Department of Computer Science
University of California, Irvine
Irvine, USA
Email: ealiaj@uci.edu

Alberto Krone-Martins
Department of Informatics
University of California, Irvine
Irvine, USA
Email: algol@uci.edu

Joshua Garcia
Department of Informatics
University of California, Irvine
Irvine, USA
Email: joshug4@uci.edu

Sang-Woo Jun
Department of Computer Science
University of California, Irvine
Irvine, USA
Email: swjun@ics.uci.edu

***Abstract*—Application-specific hardware acceleration of computation-intensive kernels can often provide significant performance and power efficiency improvements over general-purpose software, but it is difficult and costly to incorporate them into existing software systems. Designing hardware accelerators and modifying legacy software to incorporate them are already complex tasks. Furthermore, identifying a suitable kernel for acceleration is complicated by the PCIe-attached architecture of commodity FPGA cards, meaning bandwidth and latency overhead must be considered for kernel selection. For example, small blocking kernels may not benefit from acceleration due to PCIe latency.**

As a remedy, we present FarSlayer, a high-level source-to-source compiler for end-to-end acceleration of legacy software. FarSlayer analyzes existing software code and emits an accelerated version of it, where the kernel is automatically selected considering data movement over PCIe. Specifically, FarSlayer identifies kernels which can be called asynchronously to hide the PCIe latency, while also having a high operational intensity for low bandwidth requirements. The entire process is automatic, meaning the programmer does not necessarily need to understand the existing code, or reason about hardware development. We demonstrate FarSlayer on multiple existing scientific computing software systems, and demonstrate it can automatically achieve significant performance improvements.

***Index Terms*—high-level synthesis, FPGA acceleration, PCIe**

I. Introduction

Application-specific hardware accelerators implemented on reconfigurable acceleration platforms such as Field-Programmable Gate Arrays (FPGAs) can often achieve significantly higher performance and power efficiency compared to conventional software implementations executing on general-purpose microprocessors. Performance and power efficiency benefits have been demonstrated for a very wide array of important applications, including scientific computing [1], bioinformatics [2], high-frequency trading [3], databases [4]–[6], and artificial intelligence [7]. As a result, efficient use of FPGA accelerators is being actively researched as one of the most prominent means of continued performance improvements in the face of the looming end of general-purpose performance scaling [8], [9], and FPGA-accelerated machines are becoming more available in cluster systems [10], [11].

However, development of efficient hardware accelerators, especially for augmenting complex legacy software, is often a prohibitively costly task leading to underutilization of available FPGA resources. We focus on two prominent reasons which make FPGA acceleration of legacy code challenging. First, the programming model of hardware accelerators is drastically different from software. Not only does hardware development introduce unfamiliar concepts such as clock timing closure and fine-grained parallelism, some natural tools of software development such as recursion and dynamic memory management are very inconvenient to implement in hardware [12], [13]. Second, identifying a worthwhile kernel to offload to the accelerator is difficult, due to the latency and bandwidth limitation of PCIe communication between the accelerator and software. For example, blocking invocation of small kernels are bottlenecked by high PCIe latency, and kernels with large data movement are bottlenecked by PCIe bandwidth [14], [15].

High-Level Synthesis (HLS), which translates high-level software code to hardware design, helps the programming model issue greatly once a kernel has been selected. But kernel selection and its integration into existing software is still mostly left to the programmer. The few existing exceptions either depend on the low latency of specific hardware platforms [16] or require the programmer to rewrite code in a domain-specific programming model [17]. Furthermore, even HLS tools require the programmer to provide guidance on hardware optimization via pragmas to achieve efficient implementation.

FarSlayer Solution: To address both issues, we present FarSlayer, a high-level source-to-source compiler which performs partial acceleration of existing software, while optimizing for the accelerator communications overhead. FarSlayer

DOI 10.1109/ASAP57973.2023.00037

receives as input a C/C++ program along with parameters directing it to the compute-intensive region of code, and emits a functionally identical C/C++ program augmented with glue logic for invoking an FPGA accelerator, as well as emit the design for the accelerator itself in a Register-Transfer Level (RTL) hardware-description language (HDL).

FarSlayer differs from conventional HLS tools in three ways: **First**, FarSlayer performs static analysis to automatically identify communication-effective kernels in the context of slow, high-latency PCIe communications. The programmer's task is simply to guide FarSlayer to the computationally intensive portion of existing software, based on profiling tools such as perf, and the tool will automatically explore surrounding code regions to determine the best kernel to accelerate. It checks for data movement per kernel invocation, as well as checks for inter-kernel dependencies. The best kernels will have low bandwidth requirements per unit computation, and lack of inter-kernel dependencies allow overlapping kernels to hide communication latency. **Second**, FarSlayer generates very conservatively pipelined accelerator designs, trading off chip space efficiency for reliable performance without human intervention, where the performance can be predicted offline without using a physical FPGA. **Third**, the automatically generated kernel is also automatically integrated into the existing software. In the future, we plan to follow a principled approach to improving chip space utilization while maintaining programmability and performance.

Primary Deployment Scenario: The primary beneficiaries of FarSlayer would be users of custom software such as scientific computing and bioinformatics software, running on a shared cluster computing environment with pre-installed FPGA accelerators. Admittedly, software accelerated by the current version of FarSlayer will likely be slower than a hand-optimized version by a costly human expert. However, the main benefits of FarSlayer is that it requires almost no hardware expertise or deep understanding of the legacy code. For users of custom software, it would be too expensive to pay the development cost of developing a hardware accelerator, leaving the FPGA resources underutilized. In such situations, FarSlayer can bring *free* performance improvements without additional effort.

Evaluation Results: We use important software examples to demonstrate the benefits of FarSlayer: Three complex, existing scientific computing software systems GIZMO [18], Gyoto [19] and Nemo [20], as well as two small-scale examples for illustrative purposes, multiplication and Cholesky factorization of small matrices. We show that FarSlayer can often accelerate computation-intensive kernels of existing software by an order of magnitude while considering the PCIe bottleneck, without manual development efforts, resulting in significant performance and power-efficiency improvement of the end-to-end system. The bandwidth-efficient kernels discovered managed to fit inside our target FPGA, Xilinx Alveo U50, despite the conservative pipelining approach of FarSlayer.

The rest of this paper is organized as follows: We present existing work and background in Section II. In Sections III, IV, and V, we present how FarSlayer performs static analysis to identify asynchronous kernels and generate efficient hardware kernels for them without human intervention, as well as present how the system can be used. We present performance and usability evaluations in Section VI, and then conclude with future work in Section VII.

II. Background and Related Work

Multiple HLS tools have addressed the programmability issues in different ways. The most popular approach is to simply compile a kernel written in a high-level language such as C/C++ to a hardware implementation. Multiple commercial and open-source offerings exist which take this approach, including Xilinx HLS [12], Dynamatic [21], GAUT [22], and ROCCC [23]. While such tools greatly help in reducing the hurdle of FPGA programming, they still leave a great amount of work to the developer, including managing timing closure and pipelining [12], [13], as well as integration of the generated hardware kernel into the existing software.

Other existing projects such as LegUp [24], PandA/Bambu [25], and others [26] have similar goals to our FarSlayer, where the tool aims to handle existing software without human expert intervention, as well as automatically integrate into existing software. But to the best of our knowledge, no existing tool has successfully targeted the slow, high-latency PCIe communication links used by typical, commercial FPGA accelerator cards such as the Xilinx Alveo or Intel Stratix offerings. Existing tools which automatically integrate into software have typically targeted a tightly integrated accelerator environment where the host processor and accelerator fabric is located on the same package [24], [26] and has memory-level low level access to the FPGA fabric.

III. FarSlayer System and Usage Overview

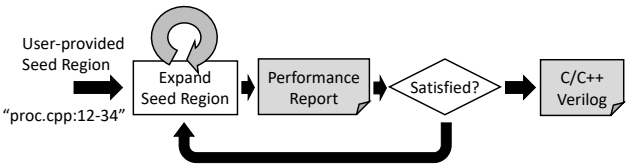

Fig. 1: FarSlayer usage overview.

Figure 1 illustrates the overall workflow of using FarSlayer on existing software. The human user's task is to provide the FarSlayer program with one or more *seed regions*, which is a rough, educated guess on the location of the computation intensive kernel worth accelerating. The direction does not need to be very exact, and is sufficient to point to a short snippet of code determined by profiling tools (such as `perf`) to be spending most of execution time. FarSlayer starts from the user-provided seed region, and automatically tries expanding the offload region to discover a good balance between computational throughput and host-side bandwidth. The details of this operation is described in Section IV. Once a good offload region has been discovered, the tool generates a projected

performance report, which is used by the user to determine the performance benefits of accelerating the kernel of interest. If the user is satisfied, the generated code and report are sent to the typical C/C++ and Verilog compilation/synthesis. If not, the user can choose to try providing a different seed region for exploration.

IV. Compiler-Supported Kernel Selection

As described in Section III, the FarSlayer compiler iteratively explores expanding a user-provided seed region to determine the most efficient region of code to offload to the accelerator. While expanding the seed region, the compiler performs static analysis of the high-level code to determine the suitability of each region for high-performance acceleration over PCIe.

A. Static Analysis Approach

When the compiler is invoked, FarSlayer parses the user-directed seed region and all other code which interacts with it, to understand and analyze how it interacts with the rest of the software. The compiler can handle multiple files of code, but because the current version of the compiler needs access to source code to analyze dependencies, it currently cannot handle pre-compiled libraries. This is currently one of the main limiting factor for universal application of FarSlayer, and we are working to address this issue using more generic binary analysis of pre-compiled libraries, and other approaches.

Internally, FarSlayer represents a program as a graph similar to the Hierarchical Task Graph (HTG) [27]. This approach organizes basic blocks (contiguous groups of instructions without control flow instructions) into a dependency graph. HTG nodes are hierarchical, where each HTG node can consist of another HTG. FarSlayer uses this representation to explore seed region expansion by climbing up the hierarchy. Figure 3 shows a simple example of an HTG graph for a very simple `for` loop depicted in Figure 2.

```
1 for (i=1; i<N; i=i+1) {
2     Loop Body;
3 }
```

Fig. 2: Simple loop example

For example, if the user pointed the FarSlayer compiler to a region of code within the loop body (HTG 0), FarSlayer will first perform static analysis of the user-provided code region, then the entirety of HTG 0, and then start moving up the graph hierarchy, i.e., HTG 1, HTG 2, and so on. The compiler will continue moving up the hierarchy until the number of operations in the target region is deemed too many to program onto the FPGA. FarSlayer uses a generous threshold heuristic value to determine this early stopping condition instead of actually attempting the synthesize the accelerator at every HTG level, because the hours-long synthesis time requirement will make the region expansion process take too long.

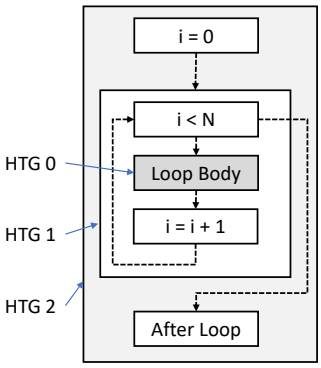

Fig. 3: Small kernels need to be overlapped to achieve high performance.

On top of the usual information stored in HTG nodes, FarSlayer also performs some simple reasoning about the number of bits actually used by each variable. This is either done via static analysis of all live variables for the region of code under compilation or by inferring the type based on the highest attainable positive value. This information allows FarSlayer to trim some bits of hardware registers or FIFOs if not all of the bits are used.

B. Kernel Suitability Metrics

Based on the HTG graph generated, FarSlayer determines the suitability of each HTG level as offloaded kernels, based on three metrics.

a) Inter-Kernel Dependencies: One of the most important metric for kernel suitability with FarSlayer is whether dependencies exist between kernel invocations. This is because kernel invocations can be overlapped if dependencies do not exist, which can effectively hide the PCIe communication latency.

Figure 4 illustrates the accelerator utilization efficiency of kernels with and without inter-kernel dependencies. For small blocking kernels, the majority of execution time is spent communicating data between the host and the accelerator. Things are a bit more favorable if the kernel is large, but in our experience most existing software systems we tested did not have a conveniently large code region which also had low data movement requirements that could be automatically extracted by the compiler. Instead, we target kernels whose executions can be overlapped in an asynchronous fashion, as seen in Figure 4(c).

b) Streaming Data Access: Another important metric for efficient kernels is whether the data access is streaming, and lacks dynamic pointer accesses. This is because dynamic pointer accesses within the offloaded kernel will either require high-latency access into host memory over PCIe, or copy the entire array into FPGA on-board memory. High-latency memory access over PCIe is a bad idea for obvious reasons. It was also difficult to reason about whether suffering the overhead of copying the entire array will result in net performance improvement.

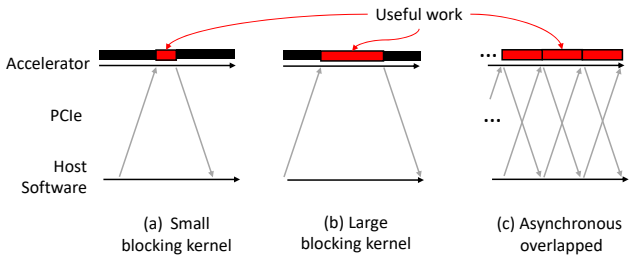

Fig. 4: Small kernels need to be overlapped to achieve high performance.

To achieve reliably good performance, FarSlayer targets kernels which are streaming, where all I/O to and from the kernel are simple scalar values. FarSlayer simply re-organizes the code so that pointer accesses are moved outside the offloaded region, along with all software code which the pointer accesses depend on. In unfavorable cases, this may result in most of the computation-intensive operations to move completely out of the offloaded region, resulting in no acceleration. When this happens, it will be plainly recognizable from the compiler-generated projected performance report, because it will include a very small number of accelerated operations. However, in our experience many publicly available scientific computing software systems, such extremely unfavorable situations did not really happen.

```
for ( i = 0 to N ) begin
    x = A[i];
    y = B[hash(i)];
    z = complex_proc(x,y);
    C[hash2(i)] = z;
end
```

Original

```
for ( i = 0 to N ) begin
    x = A[i];
    y = B[hash(i)];
    invoke_kernel(x,y,i);
    while(res = peek_kernel()) {
        C[res.id] = res.z;
    }
end
```

Asynchronous call

Accelerated

Fig. 5: Dynamic pointer accesses are moved out of the kernel region.

Figure 5 shows a simplified example code region within a loop being translated to invoke an accelerator. The compiler can decide the loop does not have inter-loop dependencies since reads and writes during the loop happen in non-overlapping locations. Reads from arrays A and B, as well as updates to array C are excluded from the offloaded kernel and left in the software region. Function `hash` is also left in the software region because access into B depends on it, but `hash2` can still be migrated to the accelerator because access into C happens after the accelerator kernel returns. Not depicted in the example is another block of glue code added after then end of the loop to wait until all hardware kernel results are returned and applied.

For extremely unfriendly examples, moving dynamic pointer accesses outside of the accelerated region may result in not much operations remaining to accelerate at all. In such situations the tool reports that the FarSlayer approach to acceleration may not be a good fit for this particular seed region.

c) Bandwidth Requirements: Once the lack of inter-kernel dependencies and dynamic pointer accesses have been assured, the expected throughput of the accelerator can be estimated via the rate of data movement necessary for each accelerator call. Since there is a strict limitation of communication bandwidth over PCIe, the effectiveness of the accelerator depends on how much computation can be done (e.g., FLOPS) per GB/s of PCIe bandwidth. For example, if the *operation intensity* is low, i.e., if a lot of data needs to be transported to the accelerator only to perform a small amount of computation on it, it may not be worthwhile to use the accelerator for that task. The compiler reports for each seed region analys includes this information to aid the user decision, including the number of bytes transported for each accelerated kernel call, and the number of operations performed per kernel call.

d) Generated Performance Projection Report: In order to roughly predict the effectiveness of the selected kernel region without going through the long synthesis process every time, the compiler generates a performance projection report for every iteration of seed region expansion. The current version of the report includes many loose pieces of information, but the important metrics for deciding kernel suitability are the following:

1) Existence of inter-kernel dependencies (Y/N).
2) The number of operations (floating-point, integer, and control flow) within the offloaded kernel.
3) The number of bytes of I/O per kernel call, as well as the expected calls/s supported by the PCIe communication bandwidth.
4) The calls/s performance of the software version of the kernel, measured by copying only the software kernel into a separate binary and executing it over an input trace.

Based on items 3 and 4, the user is able to reason about the expected performance benefits of the accelerated kernel, and reason about the reason for the performance difference using items 1 and 2.

V. Pipelined Hardware Accelerator Generation

The hardware generation functionality of FarSlayer primarily focuses on two goals: Highly efficient communication over PCIe, and not requiring user expertise in hardware design for an effective kernel generation. The design must take into account that multiple kernels must be instantiated to for the kernel performance matching with PCIe bandwidth, and that data transfer must be overlapped.

A. Automatic Kernel Hardware Generation

One of the major hurdles of accelerating existing software with hardware accelerators is the difficulty of hardware design, which involves many topics very foreign to programmers with an exclusively software background. Such topics include managing timing closure for fast clock speeds, and efficiently pipelining loops for short Initiation Intervals (II). While HLS tools greatly help with both topics, even they still require heavy expert intervention to achieve high performance and efficiency [12], and in our experience, this is enough of a hurdle to scare away hardware acceleration attempts.

To remedy this issue, the primary goal of hardware generation of FarSlayer is to first remove human intervention altogether, and attempt to achieve performance within this restriction. In order to remove clock timing concerns, FarSlayer conservatively pipelines operations into a deeper pipeline and fully unrolls loops, with software fallback. FarSlayer also employs conventional optimizations such as balancing the dependency graph to make the height shorter, lowering the latency whenever possible, and constants and bit width calculations are done whenever they can be determined at compile time.

a) Conservatively Pipelined Design: Timing violations happen typically when the depth of logic per pipeline stage is too deep to fit into the clock cycle speed, and FarSlayer tries to remove this issue by making each pipeline conservatively small. This results in a deeper pipeline with less logic per pipeline stage, which will reduce the likelihood of timing violations, but will also increase resource utilization, as well as the end-to-end latency of the pipelined kernel. This approach is feasible with FarSlayer because the compiler has already determined high latency can be hidden via overlapping independent kernel calls.

Figure 6 describes a conservatively pipelined kernel implementation of a simple mathematical formula, $x = e?(a + b + 5) : (c \times d)$. Data arrives from the host software via the on-board device memory buffers in a packed, fixed-width format. In the current version of the implementation, the input and output data width is always packed to 128-bits. Each pipeline stage only has a depth of a single arithmetic operation, minimizing the chance of long critical paths causing timing violations. The compiler also does not try to re-use operators (e.g., '$+$' for '$+5$'), instead focusing on invariably ingesting 128 bits of input data every cycle. Latencies between computation paths are matched via FIFOs, and the compiler is intelligent enough to conserve FIFO sizes according to when the data becomes available. e.g., 'e' arrives one cycle after a and b, meaning its pipeline bypass FIFO can be one slot smaller then the difference in pipeline depth. This is a somewhat extreme illustration, and the compiler actually does have a configurable threshold for how much operations it will try to pack into a pipeline stage, but by default it does try to keep each stage very simple to avoid timing issues.

b) Variable-length loops and recursion: To remove loop pipelining concerns such as II, FarSlayer also chooses to fully unroll loops (within chip resource boundaries) and fall back to software when the loop bounds exceed hardware capacity. For static loops, FarSlayer simply unrolls them fully regardless of length. For dynamic loops, it decides on an arbitrary maximum loop size and then fully unrolls the loop to that level. Then, software-side glue logic is generated to dynamically check if the variable loop variables are within the arbitrary loop bounds. If the variable exceeds the loop bounds, the glue logic simply rolls back to the original software implementation without calling the hardware kernel for that instance.

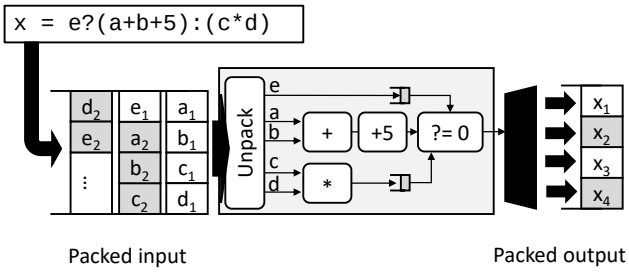

Fig. 6: FarSlayer conservatively pipelines the kernel operations to achieve wire-speed without requiring hardware expertise.

The maximum loop bounds for variable loops can also be iteratively tuned depending on resource availability and loop statistics. If the generated hardware fails to fit on the target FPGA chip, the maximum loop bounds must be adjusted. The currently implementation simply halves all loop bounds in the system uniformly until the design fits on-chip, and then increments all loop bounds by scalar amounts until it does not fit. It would be desirable to adjust these loop bounds separately for most effective resource allocation still without user intervention, and this is one of the most urgent future goals of this project.

B. Efficient Data Transfer over PCIe

Before hardware generation begins, the FarSlayer compiler should have determined that kernel call latencies can be hidden by asynchronously overlapping kernel calls. Based on the assurance, the FarSlayer hardware generator aims to achieve maximum *bandwidth*, while relaxing the latency requirements. For this, the software interface collects multiple kernel call parameters into large chunks which are efficiently transported over DMA.

Figure 7 illustrates the throughput-oriented data transfer over PCIe for multiple parallel accelerators. Every time the kernel is called, the kernel parameters are simply bit-packed and appended to the host-side buffers. Every time an 8-KB buffer is filled, the whole buffer is flushed to a on-board device memory buffer and a kernel is invoked to process it. The kernel operation will result in a stream of result values which are also stored in a packed format on the device memory buffer. Every time an input buffer chunk is finished processing, the output buffer is transported over DMA to the host to be made available to the rest of the system.

The design can have multiple kernel modules, which are invoked in a round-robin fashion. The number of kernel modules

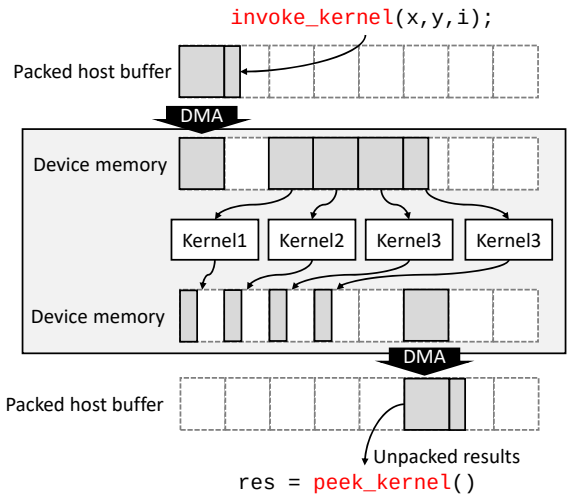

Fig. 7: Data transfer between the accelerator and the host over DMA

to instantiate depend on the width of the I/O parameters, kernel performance, and the PCIe bandwidth, so that enough kernels exist to always keep the PCIe bandwidth busy. For example, the PCIe Gen3 x16 interface of the Alveo U50 board supports a theoretical maximum of 16 GB/s of PCIe bandwidth. A 128-bit I/O port running at 500 MHz can support 8 GB/s, which would require two kernels to fully saturate the PCIe bandwidth. However, if the number of I/O parameters are smaller than 128 bits or if the kernel clock speed is slower than 500 MHz, the number of kernels must also increase to continue saturating PCIe performance.

VI. Evaluation

We implemented and evaluated our FarSlayer compiler using the Alveo U50 accelerator card with a PCIe Gen3 x16 interface. The U50 board is a relatively affordable offering in the modern Alveo line of FPGA accelerators from Xilinx. The PCIe interface has a theoretical peak bandwidth limitation of 16 GB/s. Our hardware/software platform for kernel and glue logic implementation was able to sustain 9.1 GB/s of throughput, with a kernel call latency of 8 us. The hardware platform also includes 8 GB of HBM memory. The software comparisons are done on a server machine with a 12-core Intel i7-8700K CPU running at 3.70GHz, equipped with 32 GB of DDR4 memory. When comparing kernel performance, the software kernel regions were compiled as a separate binary and executed using an execution trace.

A. Benchmark Software

We evaluated FarSlayer using five benchmark software, including three existing scientific computing software projects (Gizmo, Gyoto, and Nemo) and two scientific computing algorithms implemented in-house (Cholesky factorization, 3D distance calculation). More details about the individual benchmarked systems are decribed below:

- **Gizmo**: A multi-physics simulation platform originating from GADGET, in itself a legacy physics simulation software, used for massively parallel calculations on a wide range of topics like hydrodynamics or gravity and magnetic simulations that is able to solve the hydrodynamic equations using a variety of methods. It has been shown to compile and run on machines spanning from simple commercial-grade machines, to super-computers.
- **Gyoto**: General relativitY Orbit Tracer of Observatoire de Paris is a numerical general relativity ray tracer, mainly used for rendering stellar objects such as black holes. Gyoto seems to be less parallelizable by nature at least at this point, and due to it being quite heavy in computations, makes it a prime target for pipelining its hot-kernels.
- **Cholesky factorization:** A numerical optimization technique that is widely used in linear algebra for decomposing an Hermitian, positive definite matrix into a lower triangular and its conjugate component. These can later be used for performing various algebraic operations and solve linear systems.
- **3D Distance Calculation:** Computes the distance between two points in a 3D space, used as a component in many simulation systems such as N-Body simulators.
- **Nemo:** Nemo is a collection of programs dealing with topics on the area of stellar dynamics. These are things like various N-body systems, computing gravitational fields, and calculating orbits in various scales. Due to the interdependence of the algorithms used for e.g. orbit calculations, where the previous position, speed and acceleration states will define the next, it is hard to parallelize them. According to the output of FarSlayer, there was no loop without inter-kernel dependencies worthwhile for high-performance offloading. We present Nemo results as a negative example.

Table I shows the kernel offload regions the compiler ended up with, as well as runtime information about the kernel regions. The hot kernels were selected simply by running each software with Linux kernel `perf` and selecting the top function in terms of runtime percentage. As stated, FarSlayer was able to extract dependence-free kernels from simply this information for the two software systems GIZMO and Gyoto. It was not able to find a dependence-free kernel from Nemo which had worthwhile performance benefits. For the two in-house benchmarks the entire main loop was simply offloaded as they did not have any inter-loop dependencies by default. The percentage of time occupied by each kernel on a typical use-case is displayed together with the run time in seconds for 1 $million$ executions of that kernel.

TABLE I: Hot kernels of existing software benchmarks

	Hot Kernel	*Runtime %*	*Exec. time*
GIZMO	gravity::force_treeevaluate	82.92%	0.035s
Gyoto	generic::christoffel	24.5%	1.225s
Nemo	integrate_rk4	43.3%	0.001s

Table II shows the chip resource usage details of the five

kernels emitted automatically by FarSlayer. Despite the compiler's decision to prioritize conservative and deep pipelines, the kernels of interest managed to comfortably fit in the affordable U50 FPGA. One point of note is Gyoto, where a higher HTG node actually had superior operational intensity but the hardware kernel generated for the upper-level HTG node failed to fit on-chip, with over 1000% utilization. The Gyoto accelerator would have resulted in much higher performance (over 10x further improvement) if the whole upper-level kernel could fit on-chip, but it would require a significant amount of engineering effort to close the 1000% gap.

TABLE II: Hardware details of the compiled kernels

	Input (Bit)	*Output (Bit)*	*LUT*	*DSP*
GIZMO	832	576	16.49%	13.07%
Gyoto	64	4096	29.38%	37.71%
Cholesky	1600	1728	10.89%	2.14%
Distance 3D	384	64	15.06%	11.43%
Nemo	3232	2784	30.06%	21.04%

B. Automatic Acceleration Performance Evaluation

Figure 8 presents the performance of the hardware accelerated kernel as generated automatically by FarSlayer, and compares it against a single-thread software running on a general-purpose software, in terms of millions of kernel calls per second. As mentioned earlier, the software kernel performance was measured by compiling just the kernel region as a separate software binary and executing it with an execution trace, to measure its best-case performance without secondary effects such as cache evictions and function call overhead. We emphasize that the hardware kernel performance is measured considering the PCIe bottleneck, and the software performance is measured using a single thread.

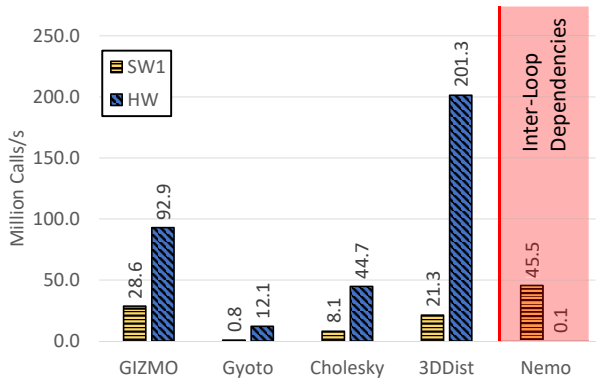

Fig. 8: Kernel performance comparisons between generated hardware and original software

Figure 9 presents the same information as the accelerator performance improvements over single-thread performance. In reality, this number is how many CPU threads FarSlayer is able to free up for other useful work, without any programmer effort. In both cases, we see that the automatically generated kernel is able to improve the performance of the software kernel even under the PCIe performance bottleneck. We also note that the performance of the Nemo benchmark is very bad, much lower than even the single-thread software performance. This is because of the blocking nature of the kernel, the hardware performance is completely bottlenecked by the hardware kernel call latency.

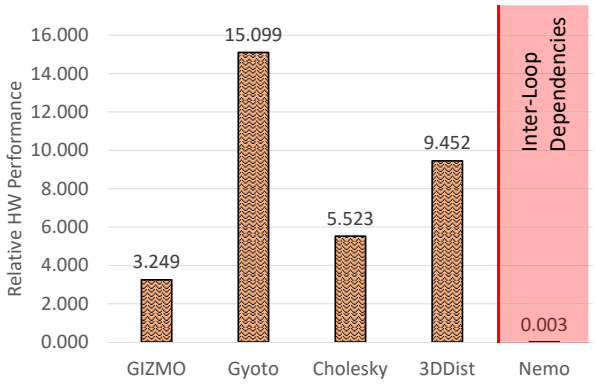

Fig. 9: Performance improvements over single-thread software

C. Performance Discussions

For the favorable benchmarks which the compiler reports good potential acceleration, FarSlayer is able to deliver 3 to 15 threads worth of performance for *free*, without almost any human effort beyond recompiling. While this is likely not comparable to the order of magnitude performance improvements potentially achievable with hand-optimized kernels and cross-layer software modifications, such an undertaking would have likely been a much more costlier and longer project than simply running the FarSlayer compiler to explore potential kernels overnight. As mentioned earlier, our target scenario for FarSlayer is conveniently taking advantage of a shared cloud or cluster environment with accelerators already installed. For this purpose, we believe FarSlayer demonstrates attractive performance improvements.

Furthermore, we believe FarSlayer is a valuable tool even outside such cluster environments, since the tool can be used to explore the range of free performance improvements even without purchasing an FPGA accelerator card. Potential users can simply run FarSlayer on their projects to determine whether an FPGA accelerator will be a worthwhile investment, and once accelerators cards are purchased for free performance improvements, further optimizations outside FarSlayer can be explored without additional hardware cost.

VII. Conclusion and Future Work

In this work, we present FarSlayer, a source-to-source compiler which can deliver performance benefits by using commercial, PCIe-attached FPGA accelerator cards, without human experts or effort. We demonstrate using multiple openly available scientific computing software as well as in-house implementations of important algorithms to demonstrate that FarSlayer can deliver significant performance benefits on

scientific computing software without additional effort. We expect FarSlayer to greatly lower the hurdle of using FPGAs for acceleration, improving their utility.

We are currently working on multiple improvements on the FarSlayer toolchain which will further improve its performance and usability benefits. For example, we are working on low-overhead compression and algorithms for improving the effective bandwidth of PCIe, as well as expand static analysis to either pre-compiled binary libraries, or to parse the source code of included pre-compiled libraries.

REFERENCES

[1] M. C. Smith, J. S. Vetter, and S. R. Alam, "Scientific computing beyond cpus: Fpga implementations of common scientific kernels," in *2005 MAPLD International Conference*, 2005.

[2] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan, "Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data," in *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2011, pp. 248–255.

[3] C. Leber, B. Geib, and H. Litz, "High frequency trading acceleration using fpgas," in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 317–322.

[4] R. Mueller and J. Teubner, "Fpga: What's in it for a database?" in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 999–1004.

[5] G. Sun and S.-W. Jun, "Columnburst: a near-storage accelerator for memory-efficient database join queries," in *roceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020.

[6] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[7] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 10–19.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 365–376.

[9] A. Fuchs and D. Wentzlaff, "The accelerator wall: Limits of chip specialization," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 1–14.

[10] C. Plessl, "Bringing fpgas to hpc production systems and codes," in *Fourth International Workshop on Heterogeneous High-performance Reconfigurable Computing, workshop at Supercomputing*, 2018.

[11] S. D. S. Center, "Nrp user guide," https://www.sdsc.edu/support/user_guides/nrp.html, accessed: 2023-03-04.

[12] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: Successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–42, 2022.

[13] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using vivado hls," in *2013 International conference on field-programmable technology (FPT)*. IEEE, 2013, pp. 362–365.

[14] G. Stitt, A. George, H. Lam, M. Smith, V. Aggarwal, G. Wang, C. Reardon, B. Holland, S. Koehler, and J. Coole, "An end-to-end tool flow for fpga-accelerated scientific computing," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 68–77, 2011.

[15] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *2010 18th IEEE annual international symposium on field-programmable custom computing machines*. IEEE, 2010, pp. 127–134.

[16] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 33–36.

[17] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2019, pp. 214–229.

[18] P. F. Hopkins, "A new class of accurate, mesh-free hydrodynamic simulation methods," *Monthly Notices of the Royal Astronomical Society*, vol. 450, no. 1, pp. 53–110, apr 2015. [Online]. Available: https://doi.org/10.1093%2Fmnras%2Fstv195

[19] F. H. Vincent, T. Paumard, E. Gourgoulhon, and G. Perrin, "GYOTO: a new general relativistic ray-tracing code," *Classical and Quantum Gravity*, vol. 28, no. 22, p. 225011, oct 2011. [Online]. Available: https://doi.org/10.1088%2F0264-9381%2F28%2F22%2F225011

[20] P. Teuben, "The Stellar Dynamics Toolbox NEMO," in *Astronomical Data Analysis Software and Systems IV*, ser. Astronomical Society of the Pacific Conference Series, R. A. Shaw, H. E. Payne, and J. J. E. Hayes, Eds., vol. 77, Jan. 1995, p. 398.

[21] L. Josipović, A. Guerrieri, and P. Ienne, "Invited tutorial: Dynamatic: From c/c++ to dynamically scheduled circuits," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 1–10.

[22] P. Coussy, C. Chavet, P. Bomel, H. Dominique, E. Senn, and E. Martin, "Gaut: A high-level synthesis tool for dsp applications," 06 2008.

[23] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 127–134.

[24] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown, and J. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, 09 2013.

[25] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, Dec 2021, pp. 1327–1330.

[26] C. Pilato, A. Cazzaniga, G. Durelli, A. Otero, D. Sciuto, and M. D. Santambrogio, "On the automatic integration of hardware accelerators into fpga-based embedded systems," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 607–610.

[27] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Trans. Parallel Distributed Syst.*, vol. 3, pp. 166–178, 1992.