Extending High-Level Synthesis with High-Performance Computing Performance Visualization

Jens Huthmann*, Artur Podobas†, Lukas Sommer‡, Andreas Koch‡ and Kentaro Sano*
*Riken Center for Computational Science, Japan.
{jens.huthmann, kentaro.sano}@riken.jp
†Royal Institute of Technology, KTH, Stockholm Sweden.
artur@podobas.net
‡Embedded Systems and Applications Group, TU Darmstadt, Germany.
{sommer, koch}@esa.tu-darmstadt.de

***Abstract*—The recent maturity in High-Level Synthesis (HLS) has renewed the interest of using Field-Programmable Gate-Arrays (FPGAs) to accelerate High-Performance Computing (HPC) applications. Today, several studies have shown performance- and power-benefits of using FPGAs compared to existing approaches for a number of application kernels with ample room for improvements. Unfortunately, modern HLS tools offer little support to gain clarity and insight regarding why a certain application behaves as it does on the FPGA, and most experts rely on intuition or abstract performance models.**

In this work, we hypothesize that existing profiling and visualization tools used in the HPC domain are also usable for understanding performance on FPGAs. We extend an existing HLS tool-chain to support Paraver – a state-of-the-art visualization and profiling tool well-known in HPC. We describe how each of the events and states are collected, and empirically quantify its hardware overhead. Finally, we practically apply our contribution to two different applications, demonstrating how the tool can be used to provide unique insights into application execution and how it can be used to guide optimizations.

***Index Terms*—Visualization, FPGA, HLS, High-Level Synthesis, High-Performance Computing, Performance Optimization**

I. Introduction

The past decades' pursuit for better and more productive High-Level Synthesis (HLS) tools has recently sparked a flurry of innovative research in using Field-Programmable Gate-Arrays (FPGAs) in High-Performance Computing (HPC). Here, FPGAs are being examined as alternative to traditional accelerators, and also to possibly mitigate the effects of Moore's law by providing a silicon substrate whose functionality changes through time. Today, several authors [1], [2], [3], [4] have already demonstrated performance- and/or power-consumption benefits of using FPGAs over server-class general-purpose processors, many-core accelerators (e.g. Xeon PHI), and graphics processing units (GPUs). However, while most works show empirical benefits of using HLS and FPGAs over alternative forms of computing, little clarity is provided into why a certain application performs as it does, and where opportunities to improve are.

High-Performance Computing has a long tradition of a diverse arsenal of profiling and visualization tools, particularly those aimed at understanding bottlenecks and limitations of high-performance applications. Profilers and visualizers have been developed for most parallel programming models, including both thread-based [5], [6] and task-based [7] models.

In this paper, we hypothesize that existing HPC visualization tools are sufficiently general to also be applied for understanding FPGA applications compiled through HLS. Confirming our hypothesis is important for several reasons: **(i)** it would provide the means to better understand the performance HLS tools yield, including identifying bottlenecks (e.g. memory-, compute- or latency-boundness), and **(ii)** FPGAs could be more seamlessly integrated into HPC tool infrastructures.

To test our hypothesis, we extend a state-of-the-art HLS tool to include modules that continuously monitor states and generate events based on the execution. Our HLS tool supports a sub-set of the OpenMP [8] 4.0 accelerator directives, allowing for multiple forms of parallelism (SISD, SIMD, MIMD) and also supports shared-memory synchronization (thread-level barriers and critical sections). Our methods are general and can be adopted for different HLS or visualization tools. We demonstrate our efforts targeting the Paraver tool-chain [6], which represents state-of-the-art in HPC profiling and visualization, and is actively used in a number of HPC centers to understand performance.

In this study we claim the following contributions:

- We describe in detail how to integrate support for an HPC profiling infrastructure into FPGA High-Level Synthesis flows, quantifying area and utilization as well as measuring the impact on performance
- Using two different applications, we demonstrate how our profiling infrastructure can be used to the understand performance of HLS-generated accelerators, showing step-by-step how to reason about and overcome the bottlenecks.

II. Background and Motivation

Understanding the performance of applications through visualization has long been an active research field in HPC. Some noteworthy visualization methodologies include Score-P [9], Vampir [5], and Paraver [6] (see the review by Isaacs et al. [10] for a complete list); these tools are heavily used to port and obtain performance in state-of-the-art HPC systems. Many of these tools work both on general-purpose processors (CPUs) and Graphics Processing Units (GPUs), and are used to provide intuitive understanding behind both application execution and performance.

The most common way of extending a particular programming model or framework (e.g. OpenMP [8]) to support trace-generation is to intercept API function calls and time-stamp the related activities. For example, when a thread is created (e.g. using `pthread_create`), a profiling library intercepts the API call and time-stamps the thread creation and saves it to a log. The log is later formatted and can be viewed using one of said visualization tools, which also provide a rich set of analyses. Due to the generality of modern processors, trace generation is often trivial to implement with low overhead, and much of the research focuses on other challenges, e.g. compression, and how to manage the often tens of GB's of trace-data.

The situation is different for HLS on FPGAs. Here, there are no easily available interception mechanism. While it is possible to add tracing mechanics into the code itself – as is done in CPU tracing – this can severely impact the hardware generation, for example increase initiation intervals (II) of loops through false dependencies, or impact memory behavior or overheads. Furthermore, monitoring stalls and memory-bandwidth is very hard since the application is not exposed to those signals. Ideally, one would change the HLS compiler itself to incorporate tracing and profiling. This option requires access to the source-code of HLS compilers, most of which are closed-source. Furthermore, the impact of fully supporting all features of an HLS generated pipeline from the perspective of hardware constructs known in HPC remains unknown and unmeasured. While trace-generation and profiling indeed are much more challenging on FPGAs than on general-purpose systems, there can also be more rewards. For example, applications running on general-purpose systems are often oblivious of low-level architectural details happening on CPUs (and performance counters are often limited); on the other hand, FPGAs are fully aware over its hardware, and much more (and interesting) information can be obtained from its execution.

Our work aspires to unify performance visualization of FPGAs to that currently existing in HPC, in order to leverage the mature methodology available in HPC and also to help bridge (and hence popularize) the use of FPGAs in HPC. Our work, to the best of our knowledge, is the first effort to fully integrate an HPC visualization framework into HLS compilers in order to reason around performance and efficiency of the HLS-generated code.

III. Nymble HLS Compiler

In this work, the ability to collect information for HPC performance visualization tools in HLS-generated FPGA-accelerators is integrated directly in an automated compile flow. The established academic HLS compiler *Nymble* [11] serves as basis for the compilation flow.

Nymble originally targeted Xilinx devices and was adapted to also target Intel FPGA boards for this work.

A. Compilation Flow

In earlier versions of Nymble, users had to use specialized, custom annotations (C/C++ pragmas) to mark regions of the application to be executed on the FPGA. The necessary data-transfers were automatically inferred by the compiler, pessimistically assuming that all data had to be transferred to the FPGA and back to the host after execution.

In order to make the Nymble HLS compiler more accessible for users from the HPC-domain, a new frontend was added which uses the OpenMP target offloading constructs instead of the custom, Nymble-specific pragmas. These constructs, standardized in version 4.0 and following the OpenMP standard, do not only allow to denote target regions with standardized annotations, but also allow users to clearly specify which and how data has to be transferred, avoiding unnecessary costly data transfers between CPU and FPGA memories.

With the new frontend, it is possible to use any C/C++-program with OpenMP annotations as input to the Nymble HLS compiler. The automated HLS flow will create an accelerator design for the target regions in the application (currently limited to one target region per application) as Verilog HDL code. Together with the architectural template shown in Fig. 1, this accelerator forms a complete FPGA-design that can be synthesized using the vendor's standard tools (Quartus in Intel's case).

As shown in Fig. 1, the generated accelerator has access to two different kinds of memory: Small, but fast local memories and the large external DRAM memory on the FPGA-board, which is also used to exchange data between host and FPGA-accelerator. Data-transfers are automatically handled as specified by the corresponding OpenMP clauses (`map`). The preloader can be used to efficiently pre-load data from the external memory to the local memory for faster access, and the hardware semaphore connected to the Avalon bus is used to handle OpenMP synchronization constructs (`critical` and `barrier`).

B. Execution Model

The execution inside the generated accelerator is organized according to a static schedule computed at synthesis time to determine the start times of individual operations.

However, for some operations, it is not possible to statically determine their operation delay. One example of such *variable-latency operations* (VLO) are (cached) memory accesses, which allow Nymble to access local (BRAM) and external (DRAM) memory. A second example of variable-latency operations are inner (nested) loops with statically unknown trip

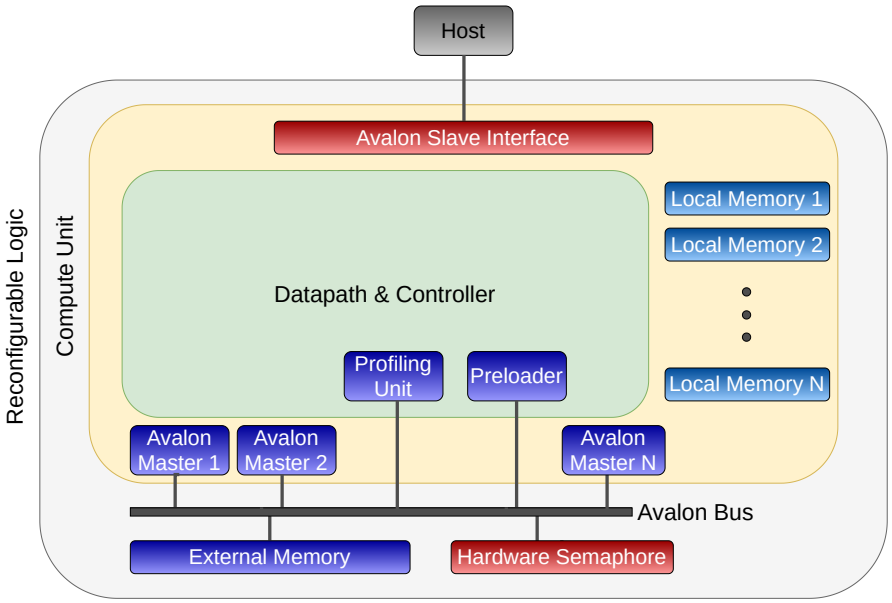

Fig. 1. Overview of the architecture template, including the Datapath and Controller, generated by Nymble, and the integrated profiling unit. All components are connected to four DDR4-banks through the Avalon bus.

count: These nodes are embedded into the dataflow graph of the surrounding loop as a *single* operation node with statically unknown delay. At runtime, the execution of the outer loop's graph is paused during execution of the inner loop.

At synthesis time, the scheduler assumes the expected *minimum* delay for VLOs. To account for longer delays of VLOs during execution, the surrounding hardware execution needs to be suspended (stalled). The number of stalls, e.g., due to external memory accesses, is an important performance figure.

However, fitting each operation with dynamic control signals (e.g., a handshake) would be too expensive in terms of hardware resource consumption. The Nymble controller therefore orchestrates the execution at the granularity of *stages*, which contain all nodes active in a single pipeline stage of the datapath. Controlling the execution at the granularity of stages requires a smaller number of control signals and less controller logic, while still allowing to suspend the execution when a variable-latency operation, e.g., access to external DDR memory, exceeds the delay assumed during scheduling.

The unique feature of the extended Nymble-MT [12] execution model employed in this work is that the model allows for the *simultaneous* execution of multiple hardware threads. Different hardware threads can be active in different stages at the same time, significantly increasing the overall throughput and resource efficiency of the accelerator. With the new OpenMP-based frontend for the Nymble HLS compiler, parallel OpenMP constructs (e.g., `teams`, `teams distribute` or `parallel`) are directly mapped to parallel hardware threads executing simultaneously in the generated accelerator.

In contrast to the basic *C-slow execution model* presented by Leiserson et al. [13], Nymble-MT also uses *thread reordering* to allow faster threads to overtake slower threads during execution. To enable a stage for thread reordering, the stage must be able to hold the context (e.g., intermediate results) of all hardware threads for all operations contained in the stage. As this requires significant amounts of hardware resources, it is not reasonable to generally enable thread reordering for *all* stages. Instead, Nymble-MT selectively enables thread reordering only for those stages that actually contain variable-latency operations, whereas the other stages in between form a *static region*. In the reordering stages, a hardware thread scheduler (HTS) selects one available thread for execution as soon as the following stage becomes available. Huthmann et al. also presented more elaborate techniques to optimize the placement of reordering stages in [14], but this is out of scope for this work.

IV. Extending High-Level Synthesis with HPC Profiling Support in Paraver

In this work, we extend the Nymble HLS compiler to include the capability of sampling and monitoring states and events, captured in the format required by modern HPC visualization tools, by embedding hardware counters for profiling directly into the generated accelerator.

Although the concrete implementation of this work is specific to the Nymble compiler, the general methodology developed in this work is generic enough to be used with and integrated into the compilation flow of other academic (e.g., LegUp [15]) or industrial (e.g., Intel OpenCL [16], Xilinx Vivado HLS [17]) HLS tools. Our additions have negligible impact on the overall compile time.

In this section, we give a brief introduction to Paraver and why we chose it, before going deeper into how different metrics are implemented and collected inside our HLS toolflow.

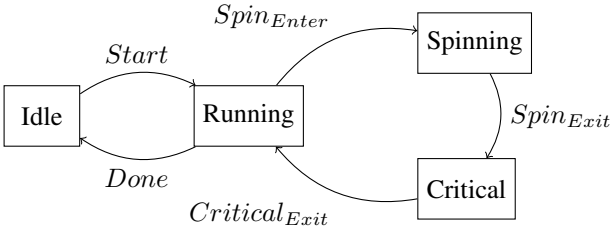

Fig. 2. The state transition diagram of our implementation for recording OpenMP critical sections. It includes profiling both the time spent in acquiring (spinning) on the lock as well as the time spent inside the protected section.

A. Introduction to Paraver

Paraver is a state-of-the-art visualization tool that brings clarity into how applications execute in HPC environments. The premise behind Paraver is that many bottlenecks in applications can easily be identified visually, such as how memory-bound the application is, or the degree of load-(im)balance across threads. Paraver visualizes the execution of actors (e.g., CPUs, Threads, MPI-ranks etc.) in a time-line view, where different colors or values represent the behavior of the thread.

Paraver supports three types of records: states, events, and communication. In this work, we have focused on supporting states and events, and excluded communication since they are primarily used for MPI communication, which for us is subject for future work in visualizing *multi-FPGA* execution.

B. Hardware Collection of Paraver States and Events

As shown in the architectural template diagram in Fig. 1, the profiling unit is integrated into the generated datapath and directly hooks-into and snoops all compute pipelines that compose the accelerator. The profiling unit is backed by the external memory, with the collected performance counters being periodically stored to external memory to avoid overflow of the counters. There they can later be accessed from the host for analysis.

As previously stated, Paraver supports three types of records, out of which we support two: states and events. A state describes the situation a thread is currently in, whereas events are, on the other hand, near-instantaneous measurements of a certain metric.

1) State Recording: States are useful in analyzing high-level details regarding the execution, such as whether the application is well-balanced (all threads contribute equally), or how many serialization points exist in the application (the time spent in critical sections). In our implementation, threads can be in four different states, as shown in Fig. 2, and the state information is available per hardware thread. The two basic states are *Running* and *Idle*. The *running* state indicates that a user (or run-time system) has loaded a context and explicitly started the accelerator. The *idle* state indicates that there is no currently loaded context, and/or the previous context finished executing. Our representation of running/idle is identical to how it is used for CPUs and GPUs.

Next to that, Nymble supports OpenMP `#pragma omp critical` sections. In multi-threaded applications, critical sections are used to provide thread-safe access to data. The absence of critical sections can lead to race conditions, breaking the application. Since the timing of entering, exiting, and waiting for the lock associated with the critical section is important, these are recorded as a state of the pipeline. The state *Spinning* is used to express that a thread is currently waiting to enter the critical section, which is currently occupied by another thread. The state *Critical* is used to track the time a thread spends inside the critical section. As a thread cannot resume computation while waiting to enter a critical section, the time spent in state *Spinning* can be an important performance figure.

The current state for each thread is stored in a register. Because the state can change for multiple threads at once, each time at least one thread changes it state, we record the current state for all threads together with the current clock count. Each state is represented as a 2-bit value: 00 for idle, 01 for running, 10 for critical, and 11 for spinning. The size of each state record is $2 * N_{threads} + 32$ bits wide. Each record is saved into a buffer; when the buffer is nearly full, the buffer is flushed to the external memory, and resumes operations. Currently, the width of the buffer is equal to the data-width of the external memory controller (512-bit), but can be tuned with other sizes.

2) Event recording: In general-purpose processors, events are often recorded by performance counters, and accessed through tools such as PAPI [18]. They include metrics such as memory-bandwidth (GB/s), compute performance (e.g. FLOP/s), or how often resources stall (% stall cycles). Implementation-wise, events are often collected for a certain time (the sampling period), and are then time-stamped and saved at periodic intervals.

In our methodology, we extended the HLS compiler to automatically insert support for collecting metrics as events. Recording an event is different from recording a state, as a threads' state can change at any time during execution, while an event is only acquired at periodic intervals.

For each of the supported events, we added a performance counter module to the accelerator. As we need to aggregate values from multiple sources (stages, operations, and others), this module has two inputs for each source. The event to be recorded from that source, and a condition if the value is valid. In each clock cycle, all valid values are added to the running aggregate. All aggregated events are periodically flushed to external memory. This period is user-adjustable, and is a proxy over fine-grained information that is required, but is also subject to a larger tracer – the higher the period, the more data is produced. Next, we will introduce the different metrics we support and how they are collected.

a) Stalls: As described in Section III-B, the Nymble-MT execution model supports variable-latency operations. If the execution of VLOs exceeds the minimum latency assumed during static scheduling, the pipeline is stalled.

Stalls in HLS environments are different from stalls in a

```
void matmul(DTYPE* A,
            DTYPE* B,
            DTYPE* C,
            int DIM){
  #pragma omp target parallel map(from:C[0:DIM*DIM])\
    map(to:A[0:DIM*DIM], B[0:DIM*DIM]) num_threads(8)
  {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    for(int i=0; i < DIM; ++i){
      for(int j=0; j < DIM; ++j){
        DTYPE sum = 0;
        for(int k=my_id; k < DIM; k += num_threads){
          sum += A[i*DIM+k] * B[k*DIM+j];
        }

        #pragma omp critical
        {
          C[i*DIM + j] = sum;
        }
      }
    }
  }
}
```

Fig. 3. Simple version of GEMM.

```
void matmul(DTYPE* A,
            DTYPE* B,
            DTYPE* C,
            int DIM){
  #pragma omp target parallel map(from:C[0:DIM*DIM])\
    map(to:A[0:DIM*DIM], B[0:DIM*DIM]) num_threads(8)
  {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    for(int i=0; i < DIM; i += num_threads){
      for(int j=0; j < DIM; ++j){
        DTYPE sum = 0;
        for(int k=my_id; k < DIM; k += VECTOR_LEN){
          VECTOR vA = *((VECTOR*) &A[(i*DIM) + k]);
          #pragma unroll VECTOR_LEN
          for(int v = k; v < k + VECTOR_LEN; ++v){
            sum += va[v-k] * B[v * DIM + j];
          }
        }

        #pragma omp critical
        {
          C[i*DIM + j] = sum;
        }
      }
    }
  }
}
```

Fig. 4. Vectorized version of GEMM.

CPU or GPU. While a CPU only has a single pipeline that can stall (a binary metric, the program counter stalls), in a Nymble accelerator every stage that contains a variable-latency operation can stall (a compile-time known discrete maximal value). In our implementation, a stall can happen due to one of two things: **(i)** a memory access takes longer than expected, or **(ii)** a resource (e.g., memory port) is shared across many threads and arbitration leads to a stall.

In this work, stalls are collected as events and the collection of stalls is implemented by snooping the control signals. For visualization purposes, it would be impractical to show per-pipeline-stage stalls, as these may occur in large numbers. Instead, we argue that an aggregated per-thread stall event is more useful from a visual perspective.

A high number of stalls during execution can be an important hint that the applications performance on the FPGA is limited by the execution of variable-latency operations, e.g., due to latency when accessing external memory, or caused by contention when entering a critical section, and can guide optimization of the application.

b) Compute Performance: Compute performance in Nymble can be classified as two types: floating-point and integer performance. Each compute-stage in the pipeline has a number of both integer- and floating-point operations scheduled onto it, determined at compile-time. By snooping the control-bus associated with each compute-stage, we can watch the value of the per-stage activation signal to determine whether the arithmetic units in the stage are active. We can then track the number of active units over time to measure the complete performance. We collect the compute-performance as an event, where each threads' compute performance is aggregated and sampled during an execution time window.

If the profiled compute performance falls short of the expected performance, this can be an indicator that the accelerator is not able to supply the arithmetic operators with data due to excessive memory access latency. Preloading data from external DRAM memory to local BRAM memory can help to improve the overall compute performance.

c) Memory Performance: For inserting the performance counters for memory accesses, there are multiple options: The counters could be placed directly at each memory operation in the pipeline, or they could be placed in the Avalon memory interface of the CU. All memory operations in the pipeline are multiplexed to one Avalon read- and one Avalon write port per thread. Therefore, we decided to place the memory performance counters in the central Avalon interface and evaluate the memory requests coming from the operators, as this reduces the footprint of the memory performance counters.

Tracking the memory bandwidth by collecting the memory requests coming from the operators to the Avalon interface incurs a small time skew. However, to get rid of this skew, we would additionally need to track memory responses, which would significantly increase the footprint of the profiling infrastructure.

Information about the memory throughput of the application over time can provide insights about the application's memory access pattern. Often, replacing many accesses to single data-items with a single read of multiple data-elements (e.g., a submatrix) into fast local memory can significantly improve the memory bandwidth and, in turn, the overall application performance.

```
void matmul(DTYPE* A,
            DTYPE* B,
            DTYPE* C,
            int DIM){
  #pragma omp target parallel map(from:C[0:DIM*DIM])\
    map(to:A[0:DIM*DIM], B[0:DIM*DIM]) num_threads(8)
  {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    for(int i= my_id * BLOCK_SIZE; i < DIM; i += num_threads * BLOCK_SIZE){
      for(int j=0; j < DIM; j += BLOCK_SIZE){
        VECTOR C_local[BLOCK_SIZE];
        for(int k = 0, buffer = 0; k < (DIM+BS); k+=BLOCK_SIZE, ++buffer){
          VECTOR A_local[BUFFER_SIZE][BLOCK_SIZE];
          VECTOR B_local[BUFFER_SIZE][BLOCK_SIZE];
          if(k < DIM){
            for(int m=0; m < BLOCK_SIZE; ++m){
              A_local[buffer%BUFFER_SIZE][m] = *((VECTOR*) &A[(i+m)*DIM+k]);
              B_local[buffer%BUFFER_SIZE][m] = *((VECTOR*) &B[(k+m)*DIM+j]);
            }
          }

          if(buffer != 0){
            for(int x=0; x < BLOCK_SIZE; ++x){
              #pragma unroll BLOCK_SIZE
              for(int y=0; y < BLOCK_SIZE; ++y){
                DTYPE sum = C_local[i][j] * (buffer == 1 ? 0.0f : 1.0f);
                #pragma unroll BLOCK_SIZE
                for(int v=0; v < BLOCK_SIZE; ++v){
                  sum += (A_local[(buffer-1)%BUFFER_SIZE][x][v]
                          * B_local[(buffer-1)%BUFFER_SIZE][v][j]);
                }
                C_local[x][y] = sum;
              }
            }
          }
        }

        for(int k=0; k < BLOCK_SIZE; ++k){
          *((VECTOR*) &C[(i+k)*DIM+j]) += C_local[k];
        }
      }
    }
  }
}
```

Fig. 5. Double buffering version of GEMM.

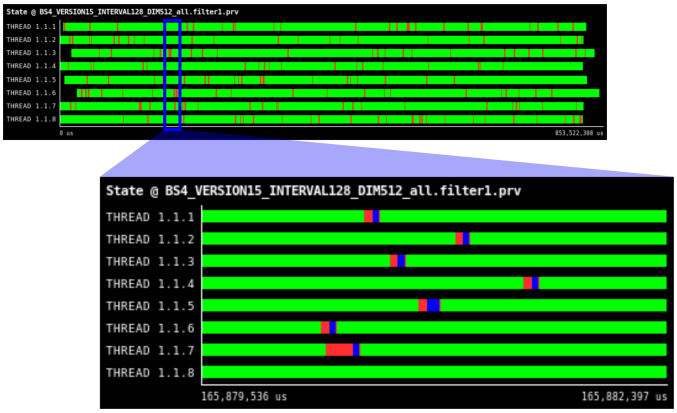

Fig. 6. Paraver state-view showing the state each of the executing hardware threads as a function of the execution time: Green represents a running state, Red represents spinning on a mutex, Blue represents atomic execution inside critical sections, and Black represents an idle thread.

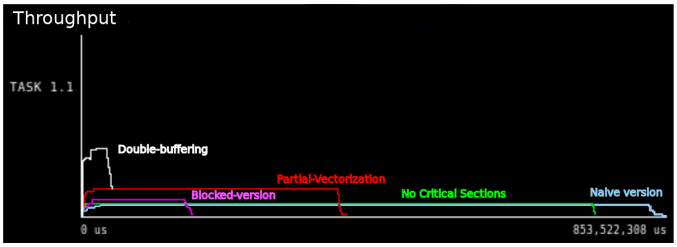

Fig. 7. Visual comparison between the different GEMM versions and their relative bandwidth over their respective execution time (absolute metrics Paraver views are not included for space reasons).

V. Results

A. Experimental Methodology

All experiments were carried out using a Stratix 10 SX-280HN2F43E2VG FPGA on an Intel D5005 PAC card using the Quartus 19.2 software. The OpenMP frontend for Nymble is based on LLVM version 9.

All performance measurements were captured using the performance counters described in the previous section. For visualization we used Paraver 4.8.2. Today, Paraver does not support the notion of *cycles*. For all cases in the graphs where microseconds are used, these are in fact cycles.

We used eight *simultaneous* threads in a single accelerator. More than eight threads in a single accelerator did not increase the performance further, because at this point all computing resources are filled. Adding more threads only increases congestion. Instantiating another accelerator would be possible, but is out of scope for this work.

B. Profiling Overhead and Hardware Footprint

The inclusion of the necessary infrastructure for the state- and event tracing of course incurs a small hardware overhead. For our first case-study, investigation of the post-P&R results shows that the inclusion of our tracing infrastructure increases the number of registers by at most 5.4% (geo.-mean 2.41%) and the number of ALMs by at most 4% (geo.-mean 3.42%), so the footprint of the tracing infrastructure is comparably small. The impact on the operating frequency of the accelerator is negligible (maximum degradation 8 MHz at 140 MHz).

A direct comparison of the different performance counters shows that each of the counters contributes similarly to the hardware overhead, none of the counters could be identified as being remarkably expensive.

For our second study, the number of registers is increased by 1.3% and the number of ALMs by 1.5%. The impact on the operating frequency of the accelerator is even smaller with a degradation of only 1 MHz at 148 MHz.

C. Case-study: Matrix Multiplication

To illustrate how our methodology can be used to guide optimizations and reason about performance of applications compiled with HLS, we turn our attention to one of the perhaps most important computational kernels: the general matrix multiplication (GEMM). In order to demonstrate our methodology, we consider matrices with size 512x512.

The starting point of our case-study is a naive implementation, as shown in Fig. 3. The version is a text-book example of a GEMM implementation, and has been extended to use OpenMP threads to parallelize the innermost loop, while protecting the update to the `C` matrix with a critical section.

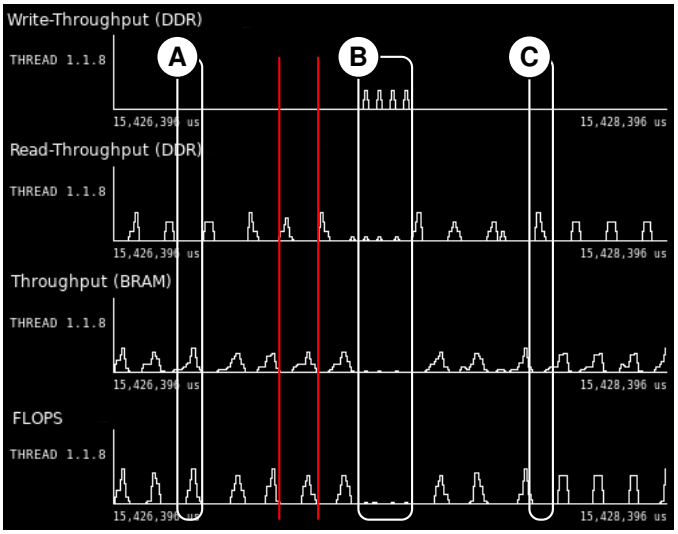

Fig. 8. Figure showing the impact of blocking on the Paraver graph and between two iterations of the matrix compute loop (the red line). We see that we compute only on data stored in local memory (**A**), which is followed by writing the local data back to external storage (**B**), and finally loading the new block from external memory into local storage (**C**).

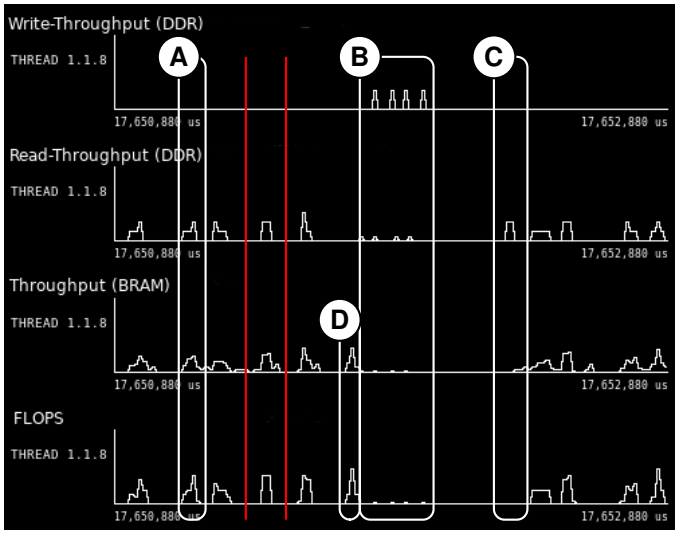

Fig. 9. Figure showing the profiling double-buffering in the Paraver graph. Best contrasted against the results in Figure 8. We now see that the external memory is being read from to prefetch the next block, concurrently with computing the matrix multiplication on the current local block (**A**), except for the final iteration where no prefetching is done, and only computation is seen (**D**). Finally, we also see that segment **C** is working on a different iteration (relative to the red lines, which separate individual iterations) than in Figure 8, which is also due to the improved double-buffering scheme. Segment **B** remains identical between the two figures.

We start by profiling our initial GEMM version. This version, for a matrix of 512x512 elements, takes 853,522,308 cycles to execute. We see the visualization for this version in Figure 6 (top); while threads are mostly running without interference, we do notice that there is a fair amount (1.54 %) of time spent in critical sections and spinning on locks (1.57 %). This – in effect – extends the serial portion of the code, limiting the parallelism (according to Amdahl's law [19]), which is shown in detail when zooming into the trace (Figure 6 (bottom)): thread 7 is spinning on the lock that protects the critical section that thread 6 is currently in. We also see that the memory throughput is quite low (Figure 7) throughout the entire execution.

In the second version (called: *No Critical Sections*), we distribute the work differently, effectively forcing threads to work on elements in the output matrix `C` in isolation, removing the need to protect it. This is a minor (and fairly non-intrusive) change in application code, but removes all critical section- and spin-states, enabling the application to fully execute in parallel (only green states, not shown due to space-limitations) and be memory-bound. We can see in Figure 7 that this version has a slightly better memory throughput, which overall yields a net improvement of 1.14x in execution time over the original version with minor improvements to obtained bandwidth.

Being memory-bound, in the third version (called: *Partial Vectorization*) (Fig. 4) we partially vectorize our application. Here we vectorize the loading of the matrix `A`, and still have `B` non-vectorized (since it needs to be transposed to allow similar vectorization). The vectorization width is 128-bit, which leads to a better memory performance, which can be observed in Figure 7, yielding the expected improvement in achieved memory throughput with wider accesses. This also materialize in a 1.93x faster execution time over the previous version.

The fourth version (called: *Blocked version*), we take a step back and try to apply a commonly used method: blocking the algorithm. Blocking should be an effective strategy, as it trades the expensive external memory operations for better, high-bandwidth, on-chip BRAM bandwidth. Blocking is often used for CPUs [20], GPUs [21], and even FPGAs [22]. Blocking consists of two stages: **(i)** we first load the sub-matrices into local BRAMs, and **(ii)** we compute on the local sub-matrices. Furthermore, we vectorized the computation on the sub-matrices, leading to better compute performance.

These two discrete stages can be seen when visualizing the throughput and compute-performance (FLOP/s) trace in Figure 8– we see the distinct compute phases as "spikes" in the execution trace, which are dependent on the loading of data (the upper throughput trace). The blocked version gives us a net performance improvement of 5.28x over the initial version. Interestingly, we also notice that this version yields a lower external memory throughput compared to our previous partial vectorized version (Figure 7). The reason for this is that we trade much of the external memory bandwidth for local memory bandwidth.

In the final version (called: *double-buffering)*, we remedy the problem of distinct load- and compute-phases. Instead, we re-write our application such that a thread performs the pre-load the *next* iterations' blocks while computing on the blocks currently available. The code for this version is seen in Fig. 5. We can see the impact on performance of this optimization in Figure 9, and we clearly see a different and more beneficial behavior compared to Figure 8: the sequential read and compute

```
DTYPE pi (int steps, int threads){
  DTYPE final_sum= 0.0;
  DTYPE step = 1.0/(DTYPE) steps;
  #pragma omp target parallel map(to : step) \
    map(tofrom: final_sum) num_threads(threads)
  {
    int step_per_thread= steps/omp_get_num_threads();
    int start_i = omp_get_thread_num()*step_per_thread;
    VECTOR sum = {0.0f};
    DTYPE local_step= step;
    for (int i=0; i< step_per_thread; i+=BS_compute) {
      #pragma unroll BS_compute
      for(int j=0; j < BS_compute; j++) {
        DTYPE x = ((DTYPE)(i+start_i+j)+0.5f)*local_step;
        sum[j] += 4.0f / (1.0f+x*x);
      }
    }
    #pragma omp critical
    for(int i=0;i<BS_compute;i++) {
     final_sum+= sum[i];
    }
  }
  return final_sum;
}
```

Fig. 10. Infinite series for π calculation, distributed across multiple threads using OpenMP.

behavior is removed, as the memory accesses and compute are now running in parallel. This also has a beneficial impact on achieved external memory throughput, where it reaches the highest performance out of our GEMM candidates (Figure 7). Overall, this version is 19x faster than our initial version.

D. Case study: Infinite series for π calculation

In the second case study, we show how problems with the scaling of algorithms can be analyzed using the state-view of Paraver. For this, we use an infinite series for calculating π (source shown in Fig. 10), which we distribute onto multiple threads. The sum-reduction of the individual results is done using a critical section. Figures 11, 12, and 13 show the Paraver state traces for 1, 4, and 10 million iterations respectively. For 1,000,000 iterations, the hardware only achieves 0.146 GFLOP/s. From the trace, it can be seen that the overhead of starting the individual threads by the software causes the earliest threads to be finished before last ones are even started. If we increase the iteration count to 4,000,000, all threads are starting to get executed in parallel, resulting in an increased performance of 0.556 GFLOP/s. Further increasing the number of iterations to 10,000,000 gives us a performance of 1.507 GFLOP/s. Unfortunately, since we are using only single-precision computation, further increasing the number of iterations results in numerical instability. Ignoring the instability, increasing the number of iterations to $15{\cdot}10^9$ would give us 36.84 GFLOP/s.

From this case study we can see that the bottleneck for small computational loads is located in the communication between the software and hardware, so we will look into how we can improve our interface.

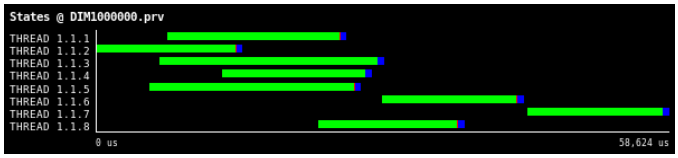

Fig. 11. Paraver state-view (colors explained in Fig. 2) for π with 1 million iterations divided onto 8 threads. Here it can be seen that not all threads are executing simultaneously, resulting in a performance of 0.146 GFLOP/s.

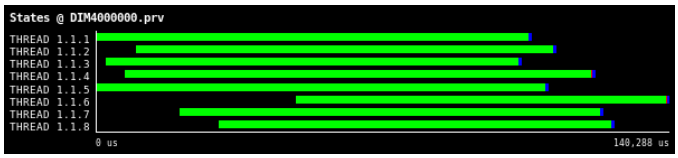

Fig. 12. Paraver state-view (colors explained in Fig. 2) for π with 4 million iterations divided onto 8 threads. Compared to Fig. 11, the threads are now running more and more in parallel, resulting in a performance of 0.556 GFLOP/s.

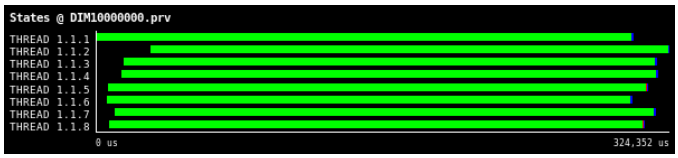

Fig. 13. Paraver state-view (colors explained in Fig. 2) for π with 10 million iterations divided onto 8 threads. Compared to Fig. 12, most of the time is spent running all threads, resulting in a performance of 1.507 GFLOP/s.

VI. Related Work

The SoCLog [23] platform provides real-time acquisition of profiling data for FPGA-based System-on-Chips. The primary metric of collection is activity and the authors visualize said activity on a time-line. They show-case their work on DCT application, showing how different traces of two versions (one un-optimized and one optimized) differs in their framework. Compared to our work, which also captures activity (IDLE/RUNNING), we also incorporate many more metrics to provide a more truthful and informative picture of the execution. Curreri et al. [24] profile FPGA throughput between producer and consumer in HLS applications. The authors visualize the performance as a DAG where nodes represents producers and consumers, and edges data communication (throughput), showing both absolute and relative performance. Compared to our work, we aggregate and abstract throughput performance in both off- and on-chip memory, showing it per-thread and per-FPGA, rather than per-consumer/producer, and we are more consistent with how HPC application display these metrics.

Additionally, we support more metrics than only throughput. Podobas et al. [25] used Paraver to reason about and demonstrate the effect of resource-sharing and arbitration on load-(im)balance in multi-threaded FPGAs similar to our work in this paper. This work extends their work by including a much richer and complex set of events. Calagar et al. [26] researched source-level profiling using their tool Inspect,

which links the generated hardware (Verilog) to source-level (C/C++) constructs in order to provide understanding around what hardware is generated, and what is happening through a familiar `gdb`-like interface. Our work focuses more on the visualization of real-time performance, but in the future could also benefit from linking traces with source-level annotations in a way similar to what they proposed. The OmpSs [27] task-based eco-system supports offloading OpenMP tasks to FPGA accelerators [28], and the authors also support Paraver trace-generation during FPGA execution. Their work focuses mostly on visualization how the host processor orchestrates FPGA execution, e.g., memory transfers to-and-from the FPGA itself rather than what happens *inside* the accelerator (as this work does). Curreri et al. [29] provides a general methodology for profiling High-Level Languages (essentially HLS) for FPGAs, and create a hardware module capable of capturing performance metrics. They demonstrate their framework on a molecular dynamics application in Impulse C. Our work extends this by collecting many more diverse metrics, and we are also not limited to streams, but we also include threads. Both Intel and Xilinx offer visualization tools and profiling information for various parts of their respective HLS flow, including reporting performance bottlenecks (e.g., high initiation intervals) and what memory interfaces were synthesized for what operation (e.g., coalesced, burst, etc.), often linking performance problems back to source code. Our work extends both of these by focusing on profiling dynamic execution in a multi-threaded environment, where we also include concepts of threads, critical sections, and achievable throughput.

VII. Conclusions

In this work, we have developed a profiling infrastructure with OpenMP-based frontend that can be included directly into an HLS toolflow that produces traces that can be used with state-of-the-art performance visualization tools. The infrastructure was included into the Nymble HLS compiler and used with the Paraver visualization tool, but is sufficiently general to be included in other HLS tools and to be used with other HPC visualization tools.

Using two different applications, we demonstrated how the performance visualization can be used to precisely analyze performance bottlenecks, and successfully optimize the performance by restructuring the HLS input code.

In the future, we plan to extend our infrastructure for communication between FPGAs in a multi-FPGA setup and evaluate how the collected traces can be used for profile-guided optimization in the HLS compiler.

References

[1] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 77–84.

[2] A. Podobas and S. Matsuoka, "Designing and accelerating spiking neural networks using opencl for fpgas," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 255–258.

[3] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 153–162.

[4] C. Yang, T. Geng, T. Wang, R. Patel, Q. Xiong, A. Sanaullah, C. Wu, J. Sheng, C. Lin, V. Sachdeva *et al.*, "Fully integrated fpga molecular dynamics simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–31.

[5] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis toolset," in *Tools for high performance computing*. Springer, 2008, pp. 139–155.

[6] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1. Citeseer, 1995, pp. 17–31.

[7] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain graphs: Openmp performance analysis made easy," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–13.

[8] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[9] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[10] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization." in *EuroVis (STARs)*, 2014.

[11] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/software co-compilation with the Nymble system," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, Jul. 2013, pp. 1–8.

[12] J. Huthmann, J. Oppermann, and A. Koch, "Automatic high-level synthesis of multi-threaded hardware accelerators," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–4.

[13] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming (preliminary version)," in *Third Caltech conference on very large scale integration*. Springer, 1983, pp. 87–116.

[14] J. Huthmann and A. Koch, "Optimized high-level synthesis of SMT multi-threaded hardware accelerators," in *2015 International Conference on Field Programmable Technology (FPT)*, 2015, pp. 176–183.

[15] 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.

[16] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *22nd international conference on field programmable logic and applications (FPL)*. IEEE, 2012, pp. 531–534.

[17] Xilinx, "Vivado design suite user guide: High-level synthesis," vol. 2, 2018.

[18] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[20] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 1, pp. 1–14, 2008.

[21] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 133–137.

[22] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point fpga matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 86–95.

[23] I. Parnassos, P. Skrimponis, G. Zindros, and N. Bellas, "Soclog: A real-time, automatically generated logging and profiling mechanism for fpga-based systems on chip," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[24] J. Curreri, G. Stitt, and A. George, "Communication visualization for bottleneck detection of high-level synthesis applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012, pp. 33–36.

[25] A. Podobas and M. Brorsson, "Empowering openmp with automatically generated hardware," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2016, pp. 245–252.

[26] N. Calagar, S. D. Brown, and J. H. Anderson, "Source-level debugging for fpga high-level synthesis," in *2014 24th international conference on field programmable logic and applications (FPL)*. IEEE, 2014, pp. 1–8.

[27] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[28] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers, "Ompss@ zynq all-programmable soc ecosystem," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 137–146.

[29] J. Curreri, S. Koehler, A. D. George, B. Holland, and R. Garcia, "Performance analysis framework for high-level language applications in reconfigurable computing," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 1, pp. 1–23, 2010.