Abstract: Heterogeneous platforms with FPGAs have started to be employed in the High-Performance Computing (HPC) field to improve performance and overall efficiency. These platforms allow the use of specialized hardware to accelerate software applications, but require the software to be adapted in what can be a prolonged and complex process. The main goal of this work is to describe and evaluate mechanisms that can transparently transfer the control flow between CPU and FPGA within the scope of HPC. Combining such a mechanism with transparent software profiling and accelerator configuration could lead to an automatic way of accelerating regular applications. In this work, a mechanism based on the ptrace system call is proposed, and its performance on the Intel Xeon+FPGA platform is evaluated. The feasibility of the proposed approach is demonstrated by a working prototype that performs the transparent control flow transfer of any function call to a matching hardware accelerator. This approach is more general than shared library interposition at the cost of a small time overhead in each accelerator use (about 1.3 ms in the prototype implementation).

Keywords: heterogeneous computing; reconfigurable computing; transparent acceleration; FPGA; HPC

1. Introduction

In recent years, the claim that Moore’s law is dead has been frequently made. Although the statement may be disputed [1], there is no doubt that the miniaturization of transistors will not go on forever [2,3], and that increased integration makes power and thermal management of computing resources more challenging [4]. With each new technology node, more attention has been given to heterogeneous platforms, which contain different types of processing units, like CPUs, GPUs, and FPGAs [5]. In these platforms, the more intensive and regular parts of the computation (“hot spots”) are executed faster and more efficiently by the more specialized units. Since FPGAs can be configured to match a specific application’s behavior and may achieve high levels of parallelism, they have enjoyed growing acceptance in the High-Performance Computing (HPC) domain [6,7].

A major problem that inhibits further adoption of such platforms is the need to rewrite or adapt the application source code to take advantage of the newly available resources. In the absence of special support, this may entail a very complex and time-consuming process, which may be infeasible due to expertise, time, or cost constraints.

The ideal scenario would be one in which applications do not need to be explicitly prepared to benefit from accelerators implemented in FPGAs—existing binary executables would simply run and automatically take advantage of available accelerators, without their developers having to consider accelerator availability. Indeed, accelerators could be created long after the program is developed. This effect of “transparent acceleration” can be achieved by combining the following three mechanisms:

1. Non-intrusive profiling of the target application;
2. Automatic configuration of the appropriate accelerators;
3. Transparent transfer of the control flow at run time.
The entire process would then proceed as shown in Figure 1. When a program is started, the transparent profiling mechanism would be used to obtain information about the execution of the program. The obtained information would then be used to extract possible execution bottlenecks (hot spots), and the transparent accelerator configuration mechanism could retrieve (or generate) appropriate accelerator configurations for these hot spots (a possible approach is outlined in [8] in the context of embedded systems). From this moment on, the control flow, that is, the order in which statements, instructions, or function calls of an imperative program are executed or evaluated, could be transferred to an already configured accelerator. A transparent mechanism for transferring control flow would ensure that the program experiences a speedup, as hotspots on available accelerators execute more efficiently, all without user intervention.

This work focuses on the third point, transparent control flow transfer in the context of HPC platforms—specifically, the Xeon + FPGA platform [9].

Several proposals have been made for a mechanism capable of transparently transferring control flow between a CPU and an accelerator, but most of them target embedded platforms [10–12]. Embedded platforms benefit from a heterogeneous computational model, but allow much more fine-grained control, leading to approaches that are not directly transferable to HPC. For example, the co-processor BERET [11] requires software to be compiled in a special process that tags instructions where each hot spot begins. When a tagged instruction is later retrieved, the fetch stage of the processor identifies it by checking to see if it is in a list of all tagged instructions. An input trigger is then sent to the BERET co-processor along with some configuration data. Another approach is proposed in [12], where a local memory bus injector is responsible for triggering the transfer of the control flow. This block watches the instruction bus, and when it detects the start of a hotspot, it modifies the instruction flow to invoke a subroutine that takes over the transfer. Approaches such as those described have not been developed for HPC platforms because they require a redesign of the processor.

Previous approaches applicable to HPC only handle hotspot functions. The Shared Library Interposing (SLI) approach [13] takes advantage of the Linux dynamic linker. It works by using the environment variable LD_PRELOAD to instruct the dynamic linker to load custom shared libraries instead of the default ones. The new custom shared libraries contain alternative accelerated versions of the original functions, which handle the transfer of the execution to and back from an accelerator using an accelerator-specific software interface. An example of an accelerated shared library is NVBLAS, which implements GPU-accelerated versions of routines from BLAS. By using LD_PRELOAD to load NVBLAS instead of the default BLAS implementation, the application’s control flow is transparently transferred to a GPU at runtime. It is a simple mechanism that works, but is affected by a specific limitation: it can only be used to accelerate hot spots that are in shared libraries. The Courier tool chain [14] also employs a similar approach, and suffers from the same limitation.

This work proposes a generic transparent control-flow transfer mechanism that can be used with traditional operating systems such as Linux, and hence in the context of HPC.
Such a mechanism is responsible for stopping a running program, transferring data and control flow to an accelerator built on FPGA, and later transferring data and control flow back to the CPU. The program should continue as if its execution was entirely in software (without using an accelerator). For the prototype implementation, the Xeon + FPGA platform was used, where a server-grade Xeon E5-2600 v4 processor is tightly coupled to an Arria 10 FPGA on the same chip [9]. The two are connected with high-bandwidth communication links, including a QuickPath Interconnect (QPI) bus, that enables main memory sharing in a cache-coherent manner. This is a particularly interesting architecture because CPU and FPGA can work together without having to transfer data back and forth, allowing for finer-grained workload partitioning of the workload.

The newly proposed mechanism relies on the Linux system call `ptrace` [15] to control the process to be accelerated and cause its execution flow to be transferred to an accelerator. Results obtained with a prototype implementation show that the approach is viable and can be used effectively to handle arbitrary hotspot functions, not just those located in shared library routines. Moreover, as discussed in Section 4, the approach can be extended to handle hotspots that are not necessarily subroutines of the original code (such as the “megablocks” of [16,17]).

The remainder of this work is arranged as follows. Section 2 describes both the proposed mechanism and its implementation. Section 3 contains the results obtained by applying the proposed mechanism to two different applications and their interpretation. Finally, Section 4 contains a discussion on the results, some concluding remarks, and directions for future work.

2. Control Flow Transfer Mechanism

This section first presents an overview of the proposed control flow transfer mechanism, followed by a description of our implementation.

2.1. Overview

The proposed mechanism has a new process running alongside the program to be accelerated. The new process (the “manager”) uses the Linux system call `ptrace` to monitor and control the process to be accelerated (the “target”). Due to Linux security features, for the manager to be able to start tracing the target, it is generally enough for both processes to belong to the same user, but some Linux configurations impose that a process can only trace its own child processes. The idea is that the manager can use `ptrace` to stop the target before it executes a hotspot, and handle the transfer of control to and from an accelerator. When the accelerator stops execution, the target resumes execution on the CPU. A diagram of the system architecture described is shown in Figure 2. This ultimately behaves similarly to the embedded oriented approaches briefly described in Section 1. The main difference is that this alternative is based on software rather than dedicated hardware, using services provided by the operating system and requiring no hardware changes. The accelerator is expected to provide both an interface in the form of a collection of Control and Status Registers (CSRs) through which the parameters of the accelerated task can be configured, and to access the main memory directly to read/write data.
2.2. Implementation of Control Flow Transfer

The target’s execution is halted when it reaches a breakpoint inserted by the manager. This is achieved in a very similar way to how some debuggers work. They too insert breakpoints and then wait for them to be reached by taking advantage of ptrace. The manager uses ptrace to dynamically change the program and insert a trap instruction at the beginning of each hotspot, as shown in Listing 1 for a single hotspot. In this example, the first instruction of the hotspot was coded using 5 bytes. As a trap instruction is coded in a single byte, the original instruction is replaced with the trap instruction, followed by four NOP instructions.

Listing 1. Inserting a trap instruction at the start of a hotspot.

```c
uint8_t trap_instruction[8] = {0xCC, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
uint8_t old_instruction[8];
uint64_t break_addr = get_break_addr(i);

// read 8 bytes from tracee’s memory into old_instruction using ptrace
peek_text(tracee_pid, (void*)break_addr, &old_instruction, 8)

// replace a 5 byte instruction with an INT3 (0xCC) followed by NOPs (0x90)
// keep the bytes that follow the replaced instruction
trap_instruction[5] = old_instruction[5];
trap_instruction[6] = old_instruction[6];
trap_instruction[7] = old_instruction[7];

// write the new 8 bytes in the tracee’s memory using ptrace
poke_text(tracee_pid, (void*)break_addr, &trap_instruction, 8)
```

When a trap instruction is executed, the processor enters kernel mode; the kernel then sends a SIGTRAP signal back to the process. Every time that a process being traced receives a signal, the operating system gives control to its tracer process—in this case, control is given to the manager.

As the manager can use ptrace to insert breakpoints arbitrarily and later control from where the target is resumed, this approach is not as limited as the SLI one [13]. In fact, other routines that are not contained within shared libraries and even blocks of code that do not represent whole functions could be accelerated. Despite this advantage, it is expected that the use of another process that is frequently stopping and controlling the target application will lead to overheads that are not present in the SLI approach. These overheads are addressed in Section 3.

When a hotspot is reached, the manager must transfer the execution flow to the accelerator. In our implementation platform, accelerators must be allocated by a process, and can only access the address space of that process. Assuming that the accelerator needs
access to the main memory, the manager process cannot access the accelerator directly. To solve this problem, the manager injects into the target the code that handles this transfer. This is accomplished by using the Linux Dynamic Linking Loader and compiling an implementation of all the steps required to transfer the control flow into a shared library. The manager can make the target execute the dlopen function [18] (as proposed in [19]), which dynamically loads a shared library into the caller’s address space (available on Unix-like systems).

Dynamic loading of the shared library is achieved as shown in Figure 3. First, the manager stops the target’s process. The state of the process in that moment would be as represented at Stage 1 of the Figure. Then, the manager changes the next instruction to a call to dlopen, which is shown at Stage 2. Stage 3 shows the state of the process after the execution of dlopen. Finally, as shown at Stage 4, the manager removes the call to dlopen and restores the initial value of the instruction pointer.

Figure 3. Target’s process during the transfer code injection: Stage 1 shows the initial state of the target’s process. At Stage 2, the manager alters the process so that the next instruction is a call to dlopen. Stage 3 shows the process after dlopen is executed. At Stage 4, the manager removes the dlopen call and returns the instruction pointer to its original value.

The described injection step only needs to be performed at the beginning of the target’s execution, and from that point on, the transfer code is ready for execution when a hotspot is reached. The manager can make the target execute the injected code on demand by employing the strategy used to make it execute the function dlopen() [19].

The operation of the manager process is summarized in Figure 4. After starting, it stops the target and uses the method just described to inject the code that handles the control transfer to and from available accelerators. Before letting the target resume execution, the manager also introduces all the necessary breakpoints. It then lets the target resume execution and waits for any breakpoint to be reached. At this point, the manager makes the target execute the injected code. When the code finishes executing, the manager lets the target resume, and waits for the next breakpoint to be hit.

Figure 4. Manager operation flowchart.

The specific code that is injected into the target is dependent on the API provided by the hardware vendor to interact with the FPGA, both for programming the FPGA
and to interact with the circuits implemented there. For the FPGA in the Xeon + FPGA platform, Intel provides the Open Programmable Acceleration Engine (OPAE) [20], which is a software layer specifically designed to integrate accelerators into software applications. OPAE exposes a user-space C library that allows accelerators to be discovered, allocated, and finally, used. Successful allocation of an accelerator to an application makes the Control and Status Registers (CSRs) of the accelerator available for reading and/or writing. The accelerator can use the application’s address space to access the main memory while preserving cache coherence.

Given these concepts about the platform’s software interface (for the specific hardware platform employed in this work), the simplest way to transfer data between the software and the accelerator is to use CSRs to transfer non-variable sized data and let the accelerator directly access the main memory when variable-sized data, such as arrays, need to be accessed. Therefore, the injected code has to handle the discovery and allocation of an available accelerator, and write to the accelerator CSRs the data with fixed size and the starting addresses of data with variable size. Then, it simply busy-waits for a specific CSR to signal that the accelerator finished processing, as there is no interrupting mechanism in the Xeon+FPGA platform. A code sample of the control transfer code that is injected into the target is shown in Listing 2. In this code sample, length, IV, and key are fixed-sized data that the accelerator needs to perform its intended functionality. The address of the data array is also written to an accelerator CSR. In this example, when reading CSR number 0, the value zero is obtained while the accelerator is running.

### Listing 2. Control transfer code.

```c
void transfer_control_aes(uint8_t key[], uint8_t iv[], uint8_t data[], uint32_t length) {
    OPAE_SVC.WRAPPER fpga(AFU_ACCEL_UUID); // Find and connect to the accelerator
    CSR_MGR csrs(fpga); // Connect the CSR manager
    // allocate shared memory buffer
    auto data_handle = fpga.attachBuffer(data, getpagesize() * ((length/4096) + 1));

    // write into CSRs non-variable sized data, including size and address of arrays
    csrs.writeCSR(0, (uint64_t)data); // Address of src
    csrs.writeCSR(1, (uint64_t)data); // Address of dest
    csrs.writeCSR(2, (uint64_t)length); // data length
    csrs.writeCSR(4, (uint64_t)*(uint64_t*)iv); // IV 0
    csrs.writeCSR(5, (uint64_t)*(uint64_t*)((iv+)8)); // IV 1
    csrs.writeCSR(6, (uint64_t)*(uint64_t*)key); // key 0
    csrs.writeCSR(7, (uint64_t)*(uint64_t*)((key+)8)); // key 1
    csrs.writeCSR(8, (uint64_t)*(uint64_t*)((key+16))); // key 2
    csrs.writeCSR(9, (uint64_t)*(uint64_t*)((key+24))); // key 3
    csrs.writeCSR(3, (uint64_t)1); // Run~signal

    while (0 == csrs.readCSR(0)) _mm_pause(); // spin wait
    return;
}
```

### 2.3. Proof-of-Concept Implementation

A simple framework that implements the described mechanism was built as a proof of concept and as a means to evaluate its performance. In this implementation, only single-threaded applications were considered for acceleration, and only entire functions can be transferred to the accelerator. The framework relies on several parameters that are obtained from a JSON configuration file, which are meant to be generated by binary profiling and analysis tools. Table 1 describes the parameters that are used to automatically generate a case-specific version of the manager’s source code. The new version of the manager’s code contains an including statement for the transfer code library header file and also the name of the specific function that contains the transfer code. The parameters are also used to identify the start address of all hot spots, which, in this case, are calls to hot spot functions. The automatic generation uses the tool `objdump` to find these addresses. The addresses are stored in a new, complete version of the JSON configuration file, and later loaded by
the manager at run-time. A diagram showing the described tool flow for configuring the manager is provided in Figure 5. An example of the complete description file after the hotspot starting addresses have been added as shown in Listing 3.

**Figure 5.** Tool flow for manager configuration.

**Listing 3.** Contents of a sample configuration file loaded by the manager at runtime.

```json
{
    "functionAddr": "0000000000401700 ",
    "functionCalls": [" 40115e", " 40132e", " 40114c"],
    "targetName": "aes_ctr_soft",
    "functionName": "aes_ctr_soft",
    "functionArgs": ["type": "uint8_t **", "name": "key",
    "evaluate": 0, "thres": 01],
    "accLibString": "/libaes_ctr_acc",
    "accLibPath": "to_inject/libaes_ctr_acc.so",
    "accHeaderPath": "to_inject/aes_ctr_acc.h",
    "accFunctionName": "aes_ctr_acc"
}
```

**Table 1.** Parameters in the JSON configuration file and the corresponding description.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>functionAddr</td>
<td>Address of the hot spot function</td>
</tr>
<tr>
<td>functionCalls</td>
<td>Array containing addresses where the hot spot function is called</td>
</tr>
<tr>
<td>targetName</td>
<td>Name of the target’s process executable</td>
</tr>
<tr>
<td>functionName</td>
<td>Name of the hot spot function to be accelerated</td>
</tr>
<tr>
<td>functionArgs</td>
<td>Array describing each one of the hot spot function arguments</td>
</tr>
<tr>
<td>accLibString</td>
<td>Name of the shared library containing the transfer code</td>
</tr>
<tr>
<td>accLibPath</td>
<td>Path to the shared library containing the transfer code</td>
</tr>
<tr>
<td>accHeaderPath</td>
<td>Path to the header file of the shared library</td>
</tr>
<tr>
<td>accFunctionName</td>
<td>Name of the function containing the transfer code</td>
</tr>
</tbody>
</table>

Some additional functionality has also been implemented, such as the ability to set thresholds that the hotspot function arguments (e.g., data size) must meet in order for control flow transfer to occur. Similar functionality could be implemented for an analogous system capable of accelerating other types of code segments (e.g., “megablocks” [16]) by imposing constraints on key register values. By imposing these constraints, it is possible to tune when a hotspot should be executed in software and when it is beneficial to transfer its execution to the FPGA accelerator. In general, when the data to be processed is small, the overhead introduced by transferring the control flow cancels the gains obtained by the accelerator. Another implemented feature is automatic fallback to software when the
accelerator is not immediately available (it may be in use by another process). A specific error code is returned, which the manager uses to determine whether the target should perform the original hotspot function.

3. Experimental Results

To evaluate and characterize the behavior, the proposed mechanism was applied to two different applications that were not prepared to use accelerators. Only one hotspot per application was considered. One performs AES 256 CTR mode encryption as described in [21], and the other performs matrix multiplication. The SLI approach was also applied to both applications so that the additional overheads expected from the proposed mechanism could be measured. To ensure that the performance differences are only due to the mechanism used, the code in SLI’s alternative shared library is the same as that injected into the target process by the proposed mechanism. The accelerators are also the same.

The measured performance metrics are the time taken for the initial steps (injection and breakpoint configuration) when the proposed mechanism is used, and the average execution time of the application for a set of input data sizes under different configurations. The different configurations are:

- Software—original application (not subject to acceleration).
- FPGA Shared Library Interposing—the application runs with an alternative accelerated shared library.
- FPGA Proposed Mechanism—implemented framework without an acceleration threshold.
- FPGA + Software Proposed Mechanism—implemented framework with an acceleration threshold.

A simple accelerator was developed for each of the two applications, which complies with the control flow transfer method used by the injected code. All software were compiled with using the recommended optimization flags for the particular Xeon processor used. Care was also taken to ensure that operating system user limits, such as memory limits, did not affect the results.

3.1. AES Encryption Case

The AES 256 CTR mode encryption accelerator uses multiple instances of an open-source AES encryption kernel provided by [22]. The rest of the accelerator consists mainly of logic that handles reading and writing data to memory, as well as CTR mode logic. The software version used for comparison was obtained from [23].

Figure 6 shows the average execution times and the corresponding speedups. As expected, the original software-only version of the application manages to be the fastest when only a small amount of data needs to be processed (data size inferior to 35 KB). When compared with SLI, using the proposed mechanism introduces an additional overhead that does not seem to correlate with the size of the data processed. In this case, the additional overhead averages 1.2 ms and the small measured variations are likely a byproduct of this experiment being run on an operating system that has many other tasks running at the same time. The impact of this delay can only be evaluated when compared to the speedup that the accelerator provides, which in turn depends on the amount of data being processed.

The zoomed-in section of Figure 6a shows that if the framework is configured to use the accelerator only when it is likely to be worthwhile (total size of the data to be encrypted is less than 35 KB), the execution time becomes closer to that of the software-only version. The zoomed section also shows that when the accelerator is used and the input data size is between 130 KB and 565 KB, the average execution time is subject to some variations that do not occur otherwise. The phenomenon occurs independently of the transfer mechanism, suggesting that the two aspects are unrelated. The cause is unknown, but is probably related to the hardware platform used for acceleration.

Figure 6b shows the speedup factors resulting from the ratio of software execution times. Using the proposed mechanism results in lower speedups compared to SLI, but this graph shows that the difference becomes less significant as the data size increases.
The time taken by the initial configuration phase (code injection and breakpoint configuration) varies considerably between executions. This variability is likely due to complex system tasks, like dynamic linking (the call to `dlopen`). The minimum recorded delay was 10 ms. On average, it is 20 ms and the maximum is 50 ms.

3.2. Matrix Multiplication Case

A deliberately simple matrix multiplication accelerator was developed. The accelerator alternates between directly accessing data from two matrices and writes the result directly to memory (without special buffering on the accelerator side). The software version is the CBLAS `sgemm` routine—specifically, the `netlib` reference implementation [24].

Figure 7 is analogous to the one shown for the AES benchmark, and presents similar results. Again, as expected, the software version is the fastest when small matrices are multiplied. In this case, the additional overhead caused by the proposed mechanism is, on average, 0.7 ms, which is almost half of the overhead in the AES case. This shows that the introduced overhead varies with each application, which is to be expected since the manager is customized for each application. Looking at Figure 7b, we see that the speedup obtained while using the proposed approach is very similar to the one obtained from SLI, which can be explained by the longer execution times of matrix multiplication when compared to AES encryption. In both approaches, speedup increases with matrix size up until size $720 \times 720$, then it slightly decreases until it stabilizes for sizes around $1000 \times 1000$. This phenomenon is likely caused by cache-related issues, but is unrelated to the proposed mechanism as it is also experienced by the SLI approach.

In this case, the initial configuration phase took between 8 ms and 320 ms and, on average, it took 38 ms. The variability of the recorded results here is even higher than in the AES case. The high variability makes it difficult to compare this duration with that from the AES application. It is expected that different applications will be affected by different delays, as the injected code may be of different sizes and may require a different number of breakpoints to be set up. In this case, this difference is insignificant compared to the variable delay introduced by the operating system.
4. Discussion

The results show that using the proposed mechanism leads to a trade-off between hotspot selection flexibility and performance. Compared to the SLI approach, the proposed mechanism does not restrict hot spots to be contained within shared libraries. A hot spot can, in principle, be any piece of code, such as megablocks [16]), even though the current framework only supports transferring complete functions to an accelerator for simplicity. The proposed mechanism has a larger initial overhead that can be amortized over larger datasets.

While using ptrace is easier when the process being traced is single-threaded, the proposed mechanism could also be applied to multi-threaded applications, since ptrace works on a thread-by-thread basis. This means that the manager could simultaneously control multiple threads of the target. For example, the manager could use multiple threads to divide work, with each thread controlling one of the target’s threads; alternatively, each thread could be responsible for controlling multiple target threads. This extension is beyond the scope of the work presented here.

Using the proposed mechanism results in a lower speedup than using SLI due to two different penalties. One occurs only at the start of the application to be sped up, while the other affects every control flow transfer. The values that these penalties take in the tested scenarios should not be too problematic. In the context of HPC, applications typically run for long periods of time, which reduces the significance of these delays.

Although the experiments performed in this work concern only the Xeon + FPGA platform, the proposed mechanism should be applicable to any platform on which software can allocate and use accelerators, even if accelerators are implemented with devices other than FPGAs. The main difference would be in the code injected into the target process, which would need to be adapted to the specific acceleration platform and its software interface.

In summary, the results show that supercomputing applications can benefit from transparent use of accelerators through dynamic monitoring via the system call ptrace. Compared to a previously proposed approach, the use of ptrace can significantly increase the flexibility of hotspot selection. Moreover, accelerator selection can, in principle, be
made dependent on the runtime context (e.g., the size of the data). However, besides a possible reduction in speedup, some potential drawbacks have also been identified:

- **Architecture/platform dependency**: using ptrace to modify and control another process is highly dependent on the hardware architecture, as well as the specific Application Binary Interface (ABI). However, the use of dedicated FPGA-based accelerators is also very specific, so the impact of this drawback may be limited in practice.
- **Debugging is hindered**: Since a process can only be traced by one process at a time, it becomes impossible to debug the target with a debugger based on ptrace, while the manager simultaneously controls the target process. Again, accelerators are likely to be used only after the software-only application has been tested and debugged, so this drawback also has limited scope.

Further experiments are needed to better characterize the overheads introduced and their relationship to the application being accelerated. Future work should go in this direction, but also to improve the implemented framework on several levels. The most important ones are integration with automatic profiling and accelerator configuration mechanisms. Support for multi-threaded applications is also relevant and should be explored.

**Author Contributions**: Conceptualization, J.C.F. and D.G.; methodology, J.C.F.; software, D.G.; validation, D.G.; data curation, D.G.; writing—original draft preparation, D.G.; writing—review and editing, J.C.F. and D.G.; supervision, J.C.F. All authors have read and agreed to the published version of the manuscript.

**Funding**: This work is financed by National Funds through the FCT—Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project «PTDC/EEI-HAC/30848/2017».

**Acknowledgments**: The presented results were obtained on resources hosted at the Paderborn Center for Parallel Computing (PC²) in the context of the Intel Hardware Accelerator Research Program (HARP2).

**Conflicts of Interest**: The authors declare no conflict of interest.

**Abbreviations**
The following abbreviations are used in this manuscript:

- ABI: Application Binary Interface
- CSR: Control & Status Register
- HPC: High-Performance Computing
- OPAE: Open Programmable Acceleration Engine
- SLI: Shared Library Interposing

**References**


21. Dworkin, M.J. *Recommendation for Block Cipher Modes of Operation*; NIST: Gaithersburg, MD, USA, 2007; [CrossRef]

