Deferring Accelerator Offloading Decisions to Application Runtime

Gavin Vaz, Heinrich Riebler, Tobias Kenter, Christian Plessl
Department of Computer Science, University of Paderborn, 33098 Paderborn, Germany
Email: { gavin.vaz — heinrich.ribler — kenter — christian.plessl }@uni-paderborn.de

Abstract—Reconfigurable architectures provide an opportunity to accelerate a wide range of applications, frequently by exploiting data-parallelism, where the same operations are homogeneously executed on a (large) set of data. However, when the sequential code is executed on a host CPU and only data-parallel loops are executed on an FPGA coprocessor, a sufficiently large number of loop iterations (trip counts) is required, such that the control- and data-transfer overheads to the coprocessor can be amortized. However, the trip count of large data-parallel loops is frequently not known at compile time, but only at runtime just before entering a loop. Therefore, we propose to generate code both for the CPU and the coprocessor, and to defer the decision where to execute the appropriate code to the runtime of the application when the trip count of the loop can be determined just at runtime. We demonstrate how an LLVM compiler based toolflow can automatically insert appropriate decision blocks into the application code. Analyzing popular benchmark suites, we show that this kind of runtime decisions is often applicable. The practical feasibility of our approach is demonstrated by a toolflow that automatically identifies loops suitable for vectorization and generates code for the FPGA coprocessor of a Convey HC-1. The toolflow adds decisions based on a comparison of the runtime-computed trip counts to thresholds for specific loops and also includes support to move just the required data to the coprocessor. We evaluate the integrated toolflow with characteristic loops executed on different input sizes.

I. INTRODUCTION

With the rise of heterogeneous computing using accelerators in mobile and general-purpose systems and the advent of FPGAs in the data center [1], the question of optimal application partitioning across the HW/SW boundary is as important as ever. As FPGAs become increasingly accessible through techniques like high-level synthesis or overlay architectures, fast and automated yet accurate techniques are required to determine which parts of an application can benefit from being offloaded to an FPGA accelerator. Traditional methods have relied either on static code analysis or on profiling data to characterize suitable application hotspots. If the application's behavior cannot be statically determined at compile time or if it is heavily input data or parameter dependent, partitioning decisions need to be taken heuristically, possibly leaving significant optimization potential on the table.

Therefore we propose to defer the actual decision to offload a particular code section to an accelerator to program runtime, where dynamic information about the execution is available, for example, the loop trip count for data-dependent hot loops. We propose to automatically insert code that performs these runtime decisions into the application code. During a static partitioning process at compile time, code for both targets, CPU and FPGA accelerator, is generated. At runtime, before some potentially accelerable code section is entered, the amount of computation and the size of data to be worked on can be determined in many cases and can be used for making better partitioning decisions than static analysis. For this purpose, we build a toolflow based on the LLVM compiler infrastructure [2], which allows us to target all applications that can be compiled to, or are distributed in the LLVM intermediate representation (LLVM IR).

In this work we investigate the possible benefits of offloading decisions at runtime in two experiments. First we evaluate how often common compute workloads offer an opportunity for runtime decisions under the assumption that offloaded code typically corresponds to loops or loop nests. For this evaluation we let our LLVM-based toolflow analyze the code of three benchmark suites and determine which loops can be precisely characterized at runtime, but not at compile time. It turns out that this is the case for a considerable fraction of all detected loops. We also evaluate the overheads that are inserted into the applications this way.

Second, we evaluate the benefit of runtime decisions with an actual toolflow targeting the reconfigurable Convey HC-1 compute platform. The integrated toolflow can automatically identify loops suitable for vectorization and generate efficient code for the host CPU and a vector coprocessor implemented as an FPGA overlay. We have extended this toolflow to analyze for detected loops whether their offloading decision should be taken at runtime and insert the required decision code. When offloading decisions are deferred to runtime, data movement between CPU and coprocessor also needs to be organized at runtime as well. Alongside our runtime decisions we generate code for proper data movement and again profit from analysis that uses runtime information to determine which amount of data needs to be transferred. We apply our toolflow to a set of different, runtime-dependent loops with different nesting structures and evaluate the performance for different data dimensions. Our results show that runtime decisions can improve the overall performance of the system as compared to always executing vectorizable code either on the CPU or the coprocessor.

The remainder of this paper is structured as follows. In Section II we discuss related work, in particular other approaches to systems with offloading decisions at runtime. In Section III we present our approach of inserting the offloading decision right into the code of the application. In Section IV, we show with an analysis of common benchmark suites, how widely applicable runtime decisions are. In Section V we present the integrated toolflow with runtime decisions and code generation for a reconfigurable coprocessor and describe the
data migration strategy implemented for this platform. Then, we compare the performance of special aspects and the overall integrated toolflow in Section VI, and finally draw a conclusion in Section VII.

II. RELATED WORK

Static hardware/software partitioning is a well-known approach that has been used to achieve performance improvements as well as energy savings as compared to a software-only approach. For example, Niemann and Marwedel present early work on modeling the problem with integer linear programming [3], Lópezez-Vallejo and López present a survey on heuristic approaches [4], and Cardoso et al. [5] discuss partitioning from the perspective of compilers for reconfigurable computing. General methods for improving performance are achieved by partitioning the application and running the computational intensive parts (kernels) on the accelerator. However, the speedup depends on:

- How often the kernels are executed
- Problem size (input data)

This means that the speedup is not constant, and can differ for the same kernel depending on the input data size [6]. This affects overall system speedups and depending on the input size and communication overhead we might even see a slowdown. To tackle this problem, Stitt et al. [7] introduced dynamic hardware/software partitioning that can adapt the partitions to the actual usage of an applications during runtime. Their runtime system includes a profiler, decompiler and synthesis for a simplified configurable logic. Beisel et al. [8] propose another runtime system by using a shared library interposing mechanism that is able to transparently accelerate applications. In their approach, they use a compatible library that implements the same interface as the native library but executes the code on the accelerator. The advantage of such an approach is that the application does not need to be modified. Additionally, application developers do not have to deal with the complexities of heterogeneous architectures but can simply use them. By using this approach, they are able to decide whether to execute the native code or the accelerated code depending on different policies at runtime. To be able to effectively decide if the computation should be offloaded to the accelerator, it is important to estimate the performance gains of a heterogeneous system. Matoga et al. [6] propose a flexible shared library profiler based on the shared library interposing techniques that is capable of estimating such performance gains. They do this by first collecting execution characteristics of the application by generating a tracing library. By preloading this library, they are able to collect a sufficiently large execution trace of the application, which they then combine with the performance models for different implementations in order to estimate the speedups that can be obtained. Bispo et al. [9] extract patterns of machine instruction traces (loops) at basic block level and map them to a reconfigurable processing unit (RPU) implemented with a FPGA. They use offline simulation to detect suitable loops and then make use of offline partitioning and synthesis, while the replacement is done online (at runtime). The replacement is done transparently by monitoring the instructions during execution and modifying the program counter.

Our approach is somehow similar to these approaches. First, the acceleration is transparent for the application and no modifications are required. Second, our mechanism is also at runtime. But since we are targeting source code in LLVM IR, our approach is much more fine-grained. Library calls are restricted to function level, so the granularity of this approach is limited compared to our approach, which can directly extract hot loops. Additionally many kernels cannot be targeted by their approach, because they are directly embedded in the source code and not implemented through library calls.

Additionally, the locality of data also plays a major role in the overall system performance. The required data should always be present as near as possible to the processing unit to avoid costly recurring data transfers over the bandwidth. The overall system performance is maximized, when the data transfer overhead can be minimized. Thus fast and accurate data migration methods become a crucial factor. Lange et al. [10] present an approach to handle pointer-based algorithms in a common address space between CPU and coprocessor. Meswani et al. [11] propose a simple performance model for the Convey HC-1. However, they just outline the importance of data migration with a preliminary analysis and do not support any runtime decisions. Wang [12] and Zhao et al. [13] use hybrid memory architectures with custom data migration algorithms implemented in the memory controller of a GPU to improve energy efficiency.

III. GENERATION OF RUNTIME DECISIONS

In this section, we explain how the trip counts of loops (as one important criterion for guiding offloading decisions) can be determined at runtime by utilizing the LLVM compilation infrastructure and how the offloading decisions can be inserted into the application code.

Regardless of the actual accelerator and acceleration method used, the payoff of offloading the execution of a loop may heavily depend on its trip count. When the loop’s iteration space depends only on constants, we follow the LLVM terminology and denote this property as compile-time tripcount, meaning that decisions can be taken statically at compile time. In traditional approaches, all other loops need to be characterized by profiling. However, the iteration space of many loops depends on variables that are not changed during the loop execution itself, so their trip count is actually fixed at the point where the loop is entered. We denote this loop property as runtime tripcount. In a third class of loops however, for example when a complex, data-dependent exit condition is computed in every loop iteration, the loop has an unknown tripcount.

Our analysis is implemented with the help of LLVM’s ScalarEvolution (SCEV) analysis pass. SCEV analysis is specialized to recognize induction variables in loop iterators. By using the LLVM API, we are able to get an expression for the trip count for a specific loop. For trip counts that can be computed at compile time, the API returns a single constant value SCEVConstant defining the number of iterations for that loop or an expression with constant operands. In other cases, the trip count is expressed as being dependent on a runtime variable SCEVUnknown or an expression containing at least one of those variables. If however SCEV analysis cannot express the trip count, it returns SCEVCannotCompute.
Listing 1. Nested loop example with unknown trip count at compile time.

```csharp
const int WIDTH = 1024;
void heavyFunc(int height)
for(int i=0; i<WIDTH; i++)
for(int j=0; j<height; j++)
// Computation.
```

Listing 2. Generated IR code for computing `count3`.

```csharp
%add = add i32 %WIDTH, %height
%div = div i32 %add, 2
%smx = smax i32 %div, 50
%sub = sub i32 %smx, 50
%cnt = div i32 %sub, 2
```

However, the other two expressions depend on the variable `height` and at analysis time they are only available in our custom LLVM pass, so we require a mechanism to integrate it into the binary so that they can be evaluated at runtime. The idea is to generate LLVM IR for the expression so that we can insert into our runtime check. For this, we make use of LLVM’s SCEVExpander class, which is capable of generating LLVM IR from a given ScalarEvolution expression. The generated LLVM IR is a sequence of LLVM instructions that compute the trip count where the last instruction holds the actual value of the loop trip count at runtime, as illustrated for the computation of `count3` in Listing 2. In order to get the total number of iterations of the innermost loop inside this loop nest, this number can be multiplied by the corresponding trip counts of the two outer loops. By evaluating all expressions before the complete loop nest, repeated execution of this evaluation can be avoided.

IV. APPlicability of Runtime Decisions

In this section we investigate the possible impact of deferring a part of the partitioning decisions and selecting the appropriate computing resource at application runtime. We first evaluate, how often runtime decisions can be used in common applications and then investigate what the overhead of adding this runtime decisions is.

A. Runtime Decidable Loops in Common Benchmarks

In order to investigate how often the method of computing trip counts at runtime is applicable for real-world applications, we analyze common benchmarks for general-purpose computing and embedded systems: SPEC CPU2006 [14], MiBench [15] and MediaBench [16]. Utilizing the LLVM analysis infrastructure, we detect all loops in the benchmarks and classify their tripcount into the three categories: compile-time, runtime and unknown. Note, that this examination does not take into account the suitability of loops for vectorization or any other form of acceleration. It rather shows the overall opportunity of a runtime decision for real-world applications.

The MediaBench source code is obtained from the LLVM 3.4 testing infrastructure (test-suite) and the other code corresponds to the original sources. We compiled all benchmarks with optimization level -O3 into LLVM IR, as LLVM IR is the input format for our analysis pass. For the C/C++ files we used clang/clang++ (version 3.4) and for the F/F90 files the GFortran compiler (version 3.4) from the DragonEgg project.

The results are presented in Table I. The ratio of compile-time, runtime and unknown trip counts for C/C++ code is somewhat similar across all three considered benchmarks. About a quarter of the loops can be described by scalar expressions for a runtime-check. For Fortran (F/F90) the ratio differs significantly. The majority of all loops (64%) can be used for a runtime-check. Further analysis (see Table II) revealed that Fortran loops are affected strongly by compiler optimizations. Without optimization (level -00) only 25% of the loops have runtime tripcounts, which is very close to the result of C/C++ applications compiled with -O3. Loops in C/C++ applications showed insignificant dependence on the optimization level.

Overall, our results show, that the trip counts of almost half of all loops can be determined at runtime, but not by static approaches. If we consider only C/C++ files, about a quarter of all trip counts are usable for our hybrid approach.

| TABLE I. Analysis of Loop Tripcount Properties for Different Benchmarks. |
|-----------------------------|-------|-------|------|------|
|                             | loops | compile-time | runtime | unknown |
| SPEC CPU’06                 |       |                |        |        |
| - C/C++                     | 26,131 | 3,196 (12%)  | 5,301 (20%) | 17,634 (67%) |
| - F/F90                     | 43,890 | 4,065 (9%)   | 28,028 (64%) | 11,797 (27%) |
| - TOTAL                     | 70,021 | 7,261 (10%)  | 33,329 (44%) | 29,431 (42%) |
| MiBench                     |       |                |        |        |
| - C                         | 5,080  | 617 (12%)     | 1,285 (25%) | 3,178 (63%) |
| MediaBench I                |       |                |        |        |
| - C                         | 683    | 182 (27%)     | 180 (26%)  | 321 (47%)  |
| TOTAL                       | 75,784 | 8,060 (11%)   | 34,794 (46%) | 32,930 (43%) |

The MediaBench source code is obtained from the LLVM 3.4 testing infrastructure (test-suite) and the other code corresponds to the original sources. We compiled all benchmarks with optimization level -O3 into LLVM IR, as LLVM IR is the input format for our analysis pass. For the C/C++ files we used clang/clang++ (version 3.4) and for the F/F90 files the GFortran compiler (version 3.4) from the DragonEgg project.

The results are presented in Table I. The ratio of compile-time, runtime and unknown trip counts for C/C++ code is somewhat similar across all three considered benchmarks. About a quarter of the loops can be described by scalar expressions for a runtime-check. For Fortran (F/F90) the ratio differs significantly. The majority of all loops (64%) can be used for a runtime-check. Further analysis (see Table II) revealed that Fortran loops are affected strongly by compiler optimizations. Without optimization (level -00) only 25% of the loops have runtime tripcounts, which is very close to the result of C/C++ applications compiled with -O3. Loops in C/C++ applications showed insignificant dependence on the optimization level.

Overall, our results show, that the trip counts of almost half of all loops can be determined at runtime, but not by static approaches. If we consider only C/C++ files, about a quarter of all trip counts are usable for our hybrid approach.
TABLE II. COMPARISON OF LOOP TRIPCOUNT PROPERTIES FOR -O0 VS. -O3 OPTIMIZATION IN FORTRAN.

<table>
<thead>
<tr>
<th>SPEC CPU’06</th>
<th>loops</th>
<th>compile-time</th>
<th>runtime</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>-FF90 (-O3)</td>
<td>43,890</td>
<td>4,065 (99%)</td>
<td>28,028 (64%)</td>
<td>11,797 (27%)</td>
</tr>
<tr>
<td>-FF90 (-O0)</td>
<td>38,455</td>
<td>3,336 (99%)</td>
<td>9,715 (25%)</td>
<td>25,404 (66%)</td>
</tr>
</tbody>
</table>

B. Overhead of Runtime Decisions

As presented in the last subsection, we saw that in order to be able to take offloading decisions at runtime, our LLVM pass modifies the application by injecting instructions before each applicable loop nest. In order to measure the overhead for executing this additional code, we consider the SPEC benchmarks. We use SPEC as it has the most number of loops (see Table I) and is well supported by the LLVM test suite. We first compiled the SPEC benchmarks and executed them on the CPU. We then used our LLVM pass to inject the runtime code into the applications. However, we used a sufficiently high threshold such that the code is always executed on the CPU. In order to have a sufficiently large data set, we executed the benchmark 100 times for both the native as well as the modified test sets. On comparing the results, we found that the overhead for computing the trip counts and for a simple threshold comparison was negligible (less than 1%).

V. CASE STUDY: RUNTIME DECISIONS FOR A RECONFIGURABLE VECTOR COPROCESSOR

In this section, we present a case study that demonstrates how the runtime offloading decisions have been integrated into a toolflow that automatically partitions applications between a host CPU and a vector coprocessor implemented on the FPGA of a Convey HC-1 system. The foundations of this toolflow have been laid in our previous work [17], which focused on creating a fully automated, hardware/software partitioning toolflow and vectorizer for the Convey vector coprocessor. For the previous toolflow, we investigated only offloading decisions for compile-time tripcount loops and statically selected all loops for offloading that exceed a fixed iteration threshold. As discussed in Section IV, such static decisions are not sufficient in many real-world applications. In the remainder of this section we briefly describe the target architecture and toolflow and then present the integration of runtime decisions for offloading and data migration into the code generation backend. An evaluation of the effect of runtime decisions will be presented in Section IV.

A. Convey HC-1 Architecture

Our target system is the Convey HC-1 hybrid-core platform [18], which is based on an x86 Intel Xeon CPU that is attached to a configurable FPGA-based coprocessor via the Intel Frontside Bus (Figure 1). A defining characteristic of this system is that both the CPU and the coprocessor have local memories that are integrated in a NUMA memory subsystem with a cache-coherent, global virtual memory space. The configurable coprocessor implements application-specific instructions denoted as personalities. A personality is a loadable bundle of optimized instructions to augment the x86-64 instruction set extension. In this work we target the Convey Vector Personality, which implements a floating-point vector coprocessor. Using this personality eases the programming effort considerably, because it allows users to develop applications as one would for standard general-purpose computing systems. The corresponding Convey vectorizing compiler translates C/C++ code that is augmented with compiler directives (pragmas) to advise the compiler about the code blocks to be vectorized and offloaded and the data structures involved. The result of the compilation is a dual target executable for the host CPU and the coprocessor. In order to get rid of the need for pragmas and improve some aspects of code generation, we developed an alternative toolflow for this target.

B. Compilation Toolflow

Our toolflow generates heterogeneous binaries that can be executed on the host CPU and the coprocessor. A schematic overview of our toolflow is depicted in Figure 2. The input to the toolflow is an LLVM IR (intermediate representation) binary of the application. The main component of our toolflow is the Partitioning Pass, which is implemented as an LLVM ModulePass that works on the LLVM IR and performs the following steps:

1) Identify basic blocks that are feasible for execution on the coprocessor (no system calls, recursive function calls, I/O, etc.).
2) Extract basic blocks that belong to loops into separate functions.
3) Determine if the loops can be vectorized (inner/outer loop vectorization).
4) Add a function call interface via which the coprocessor code can be called.
5) Partition the code into: coprocessor functions, modified host code functions, and host-coprocessor interface.

Once the application has been partitioned, the modified host functions are translated into x86 assembly code by using the standard LLVM backend (llc). For implementing the function call interfaces through which the host can call the offloaded code on the vector coprocessor, we use the Convey Compiler (cnyCC). The Convey Compiler generates the function signatures, sets up the corresponding stacks and inserts coprocessor availability checks. By using the Convey Compiler instead of LLVM for creating the function call interface, we can avoid re-implementing the calling conventions of Convey’s application binary interface. Hence, the Partition Pass generates a host-coprocessor interface which is a .cpp file that contains stubs with the appropriate signatures of all functions that can be
run on the coprocessor. The Code Generation Pass generates vectorized coprocessor assembly code for the coprocessor functions based on the vectorization strategy that was previously determined by the Partition Pass. We then use a Python script that integrates our assembly code (generated by the Code Generation Pass) with the coprocessor (assembly) stubs that were generated by the Convey Compiler. We finally use the Convey Compiler tools to link all the generated assembly and object files to generate the heterogeneous binary. For a more detailed discussion of the toolflow and vectorization approach refer to [17].

C. Runtime Decisions for Offloading and Data Migration

For integrating runtime decisions into the application, we have followed the method presented in Section III. Using LLVM’s ScalarEvolution information, we add code to the application that computes the trip count for all loops. To integrate this information into the application, we have extended the Partitioning Pass with a Runtime Code Injection module to modify the CPU functions that are suitable candidates for offloading by inserting runtime decision code before the entry block of the CPU function. Thus, whenever the function is called, the runtime check evaluates the loop trip count and decides whether it is beneficial to run the code on the coprocessor. If the runtime check decides that the code should run on the coprocessor, we then migrate the required data to the coprocessor, call the corresponding coprocessor function, and then migrate data back to the host. Otherwise the execution is performed on the CPU. Figure 3 shows the control flow of the application with our runtime check. Whenever coprocessor code is available for a part of the application, the runtime check is performed to decide if the computation should be offloaded or not.

In addition to offloading decisions we also use the availability of information about the trip count of offloaded loop nests for optimizing the data transfer between CPU and coprocessor. From a functional point of view, explicit data migration is not required, because the Convey HC-1 provides a cache-coherent, shared memory between CPU and coprocessor memory. However, the memory subsystem of the HC-1 has a NUMA (non-uniform memory access) characteristic, that is, bandwidth and latency of memory accesses are strongly dependent on whether the data is stored in the local memory attached of the computing resource. To ensure that the data is present in the local memory of the computing resource requesting it, Convey provides us with API calls that migrate the data to either the host or coprocessor memory. Hence, we migrate the data from the host to the coprocessor memory before executing the coprocessor code, and once the execution on the coprocessor is complete, we migrate the data back to the host memory so that the remaining CPU code is not slowed down by having to read data from the coprocessor memory.

For optimizing these memory transfers, we analyze each function that can be offloaded to the coprocessor and determine what data will be required during its lifetime. For this purpose, we do not only consider statically available information but also tripcount information that is known at runtime. The Partitioning Pass then modifies the LLVM IR to encapsulate the call to the coprocessor with calls to Convey’s data migration API to first migrate the data to the coprocessor and then back to the host memory. An example for the modified LLVM IR can be seen in Listing 3. We use the cny_migrate_data function to migrate the data. The function parameters are; the starting address of the data to be migrated, its size, and where it needs to be migrated. In this example, arrays aa and a
were identified to be migrated during the static analysis of the a2dOuter_convey function.

In our approach, we make use of a buffer to store images. The size of the buffer is set to the maximum image resolution that we support (7680x4320). We do this as the image size is known only at runtime and we need to allocate a buffer that is large enough. In the case of smaller images, they only occupy a portion of the buffer. However as the image resolutions can vary dramatically (Table III), migrating the entire buffer to the coprocessor is very expensive. In our approach, we only migrate those portions of the buffer that are required by the coprocessor code. We determine the section of the buffer that has to be migrated at runtime by taking into account the maximum loop bounds within which the buffer is accessed. We demonstrate how the data size is computed with the help of a simple two dimensional array aa (buffer) as shown in Figure 4. The array aa has dimensions of 7680x4320. Let us assume that the coprocessor function heavyFunc_cny works on an image of dimensions 1920x1080. This only uses a subsection of the buffer i.e. elements aa[0][0] up to aa[1079][1919] as shown in the figure. By analyzing the code, we can determine this, and migrate only the required subsection of the buffer. Ideally we would only want to move those elements that are required, but the memory is organized in pages and during data migration, the entire memory pages are migrated [20]. To avoid multiple transfers, we prefer to have one large structure as compared to several smaller ones and so we migrate a 7680x1080 subsection of the buffer. By using this approach, we are able to save the cost of migrating the entire buffer to and from the coprocessor memory, which in turn results in better overall application performance.

**Listing 3. Runtime data migration.**

```c
// migrate data to the coprocessor memory
@cny_migrate_data(%aa, %size_aa, %coproc)
@cny_migrate_data(%a, %size_a, %coproc)
// dispatch call to coprocessor
%cny_call =
call @heavyFunc_cny(%N, %width, %height)
// migrate data back to the host memory
@cny_migrate_data(%aa, %size_aa, %cpu)
@cny_migrate_data(%a, %size_a, %cpu)
// CPU code
```

**Figure 4.** Data migration a two dimensional array aa[1080][1920].

**Table III. Different Applications and their Resolutions.**

<table>
<thead>
<tr>
<th>Application</th>
<th>Image Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVD</td>
<td>720x480</td>
</tr>
<tr>
<td>HD DVD</td>
<td>1280x720</td>
</tr>
<tr>
<td>Blu-Ray</td>
<td>1920x1080</td>
</tr>
<tr>
<td>4K</td>
<td>4096x2160</td>
</tr>
<tr>
<td>UHDTV</td>
<td>7680x4320</td>
</tr>
</tbody>
</table>

**VI. Evaluation**

In this section we first, evaluate the effectiveness of our data migration scheme and then later on evaluate our runtime approach with respect to the traditional partitioning approach.

For our evaluation, we make use of the Convey toolflow described in Section V-B. We evaluate our approach by using real world image dimensions as inputs to our kernels. We have chosen the image dimensions based on popular image resolutions used in the real world (see Table III).

To evaluate the impact that different kinds of compute patterns have on the system, we used a synthetic test suite of characteristic based on stereo matching algorithms [21]. This test suite consists of functions with loops that can be vectorized either by vectorizing the inner loop (denoted as Inner) or the outer-loop (denoted as Outer/Middle). Besides functions working on the 2D image space (denoted as 2D), we also consider functions working on 3D volumes (3D). Functions of this type occur for example during stereo matching to analyze a 3D matching space called cost volume between two 2D images.

**A. Data Migration**

Our test suite functions make use of a buffer from which they read/write data during their computational lifecycle. The size of the buffer that is actually used is dependent on the image size. In the traditional approach, this size cannot be determined at compile-time as the function might operate on images of varying dimensions. Hence, in the traditional approach, the entire buffer needs to be migrated to the coprocessor memory.

The aim of our data migration strategy is to able to improve the performance of the coprocessor code by determining the amount of data to be migrated to the coprocessor memory at runtime instead of migrating the entire buffer to the coprocessor memory. In order to evaluate this, we first measure the time taken by the coprocessor (including data transfer times) to execute the test suite code for different data sizes. In this case, the entire buffer is migrated to the coprocessor. We then enable our runtime data migration module and record the execution times of the test suite. The speedup of using the data migration strategy is computed by taking a ratio of the coprocessor execution time without data migration strategy, we achieve up to 9x speedups for our data migration strategy.

In Figure 5, we can see that by using our on demand data migration strategy, we achieve up to 9x speedups for smaller image dimensions. We can also observe the trend that as the image dimensions get closer to that of the buffer size, the speedups seen are reduced as more and more data has to be migrated to the coprocessor in such cases. It is also interesting to note that the speedup is not only dependent on the amount of data migrated but also on the computation...
pattern. We see that our data migration strategy produces only a marginal performance improvement for the computational intensive $a2dCensus$.

Thus, by migrating data on demand based on the runtime values, we reduce the data migration overhead and functions that were previously not profitable to run on the coprocessor (because of large data migration overheads) can now be executed on the coprocessor.

### B. Runtime System

In the traditional approach, once code has been vectorized and compiled for the coprocessor it is always executed on the coprocessor. Our approach considers the size of the data to be processed at runtime and determines if it would be beneficial to run the code on the CPU or on the coprocessor. In order to evaluate this approach, we first measure the time taken by the coprocessor to execute the test suite code for different loop iterations. We then measure the execution time of test suite for our runtime approach. In order to evaluate the performance gain attributed only to the runtime system, the measurement is performed with the data migration module enabled in both cases. The performance gain is computed by taking a ratio of the coprocessor execution time to the execution time using our runtime system. It is important to note that in our test suite, the loop iteration space is bound by the size of the image dimensions (Figure 6). We also compute the performance improvement of our runtime system to that of the CPU in Figure 7.

Figure 6 shows the performance gain that can be achieved by using the runtime system. We see that we can achieve speedups of up to 10x depending on the loop iteration space. The maximum gain is achieved for smaller image dimensions as they are run on the CPU instead of the coprocessor. As the loop iteration space increases, the code is executed on the coprocessor (Figure 7) where we can achieve speedups of up to 3x by executing the code on the coprocessor instead of the CPU.

Table IV shows us where a specific function is executed when our runtime system is used (gray cells). We can see that most of the functions are executed on the CPU for smaller loop iterations. And as the loop iteration size increases above a threshold where the data migration is amortized, the computation is performed on the coprocessor. The computation driven $a2dCensus$ is always executed on the coprocessor as

<table>
<thead>
<tr>
<th>Function</th>
<th>Proc. Unit</th>
<th>DVD</th>
<th>HD DVD</th>
<th>Blu-Ray</th>
<th>4K</th>
<th>UHDTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a2dInner$</td>
<td>CPU</td>
<td>0.03</td>
<td>0.07</td>
<td>0.25</td>
<td>1.10</td>
<td>3.89</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.06</td>
<td>0.13</td>
<td>0.19</td>
<td>0.46</td>
<td>1.23</td>
</tr>
<tr>
<td>$a2dOuter$</td>
<td>CPU</td>
<td>0.10</td>
<td>0.29</td>
<td>0.66</td>
<td>2.8</td>
<td>10.41</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.11</td>
<td>0.20</td>
<td>0.50</td>
<td>1.53</td>
<td>4.74</td>
</tr>
<tr>
<td>$a2dInnerS$</td>
<td>CPU</td>
<td>0.05</td>
<td>0.12</td>
<td>0.26</td>
<td>1.07</td>
<td>3.83</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.22</td>
<td>0.36</td>
<td>0.54</td>
<td>1.13</td>
<td>2.64</td>
</tr>
<tr>
<td>$a2dMiddleS$</td>
<td>CPU</td>
<td>0.12</td>
<td>0.29</td>
<td>0.66</td>
<td>2.79</td>
<td>10.38</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.27</td>
<td>0.43</td>
<td>0.84</td>
<td>2.22</td>
<td>6.12</td>
</tr>
<tr>
<td>$a2dInner$</td>
<td>CPU</td>
<td>0.06</td>
<td>0.18</td>
<td>0.40</td>
<td>1.66</td>
<td>5.96</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.25</td>
<td>0.42</td>
<td>0.64</td>
<td>1.33</td>
<td>2.94</td>
</tr>
<tr>
<td>$a2dMiddle$</td>
<td>CPU</td>
<td>0.11</td>
<td>0.30</td>
<td>0.68</td>
<td>2.87</td>
<td>10.62</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.30</td>
<td>0.49</td>
<td>0.94</td>
<td>2.46</td>
<td>6.68</td>
</tr>
<tr>
<td>$a2dCensus$</td>
<td>CPU</td>
<td>3.89</td>
<td>10.61</td>
<td>24.14</td>
<td>104.30</td>
<td>393.21</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>2.65</td>
<td>7.97</td>
<td>12.44</td>
<td>49.96</td>
<td>198.65</td>
</tr>
<tr>
<td>$a2dAggInner$</td>
<td>CPU</td>
<td>0.02</td>
<td>0.05</td>
<td>0.12</td>
<td>0.48</td>
<td>1.71</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.13</td>
<td>0.21</td>
<td>0.33</td>
<td>0.73</td>
<td>1.83</td>
</tr>
<tr>
<td>$a2dAggOuter$</td>
<td>CPU</td>
<td>0.02</td>
<td>0.05</td>
<td>0.11</td>
<td>0.48</td>
<td>1.69</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.14</td>
<td>0.21</td>
<td>0.32</td>
<td>0.72</td>
<td>1.74</td>
</tr>
<tr>
<td>$a2dAD$</td>
<td>CPU</td>
<td>0.01</td>
<td>0.04</td>
<td>0.08</td>
<td>0.33</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.10</td>
<td>0.13</td>
<td>0.20</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>$a3dADInner$</td>
<td>CPU</td>
<td>0.08</td>
<td>0.25</td>
<td>0.56</td>
<td>2.1</td>
<td>8.2</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.29</td>
<td>0.48</td>
<td>0.71</td>
<td>1.48</td>
<td>3.3</td>
</tr>
<tr>
<td>$a3dADMiddle$</td>
<td>CPU</td>
<td>0.08</td>
<td>0.25</td>
<td>0.56</td>
<td>2.32</td>
<td>8.25</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.29</td>
<td>0.48</td>
<td>0.72</td>
<td>1.48</td>
<td>3.42</td>
</tr>
</tbody>
</table>
VII. CONCLUSION

In this paper, we have presented an automated runtime system that can decide to execute application code on the CPU or on a FPGA-based coprocessor depending on a threshold that can be derived from the behavior of the application. Our runtime approach is superior to the approaches that execute code only on the CPU or only on a FPGA. By moving the decision of where the code should be executed from compile time to runtime, we achieve speedups that were previously not achievable by using the traditional approach. We show that by using our runtime system along with our data migration strategy, a speedup of up to 3x compared to “CPU only” and up to 10x compared to “coprocessor only” can be achieved. In this paper, we focus on the performance. However, we also believe that this approach could result in significant energy savings and we will explore this aspect in our future work.

ACKNOWLEDGEMENT

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901) and the European Union Seventh Framework Programme under grant agreement no. 610996 (SAVE).

REFERENCES


