Detecting Kernels Suitable for C-based High-Level Hardware Synthesis

Julian Oppermann and Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt, Germany
Email: {oppermann, koch}@esa.tu-darmstadt.de

***Abstract*—We present SPEXSIM, a software tool for quickly surveying legacy code bases for kernels that could be accelerated by FPGA-based compute units. We specifically aim for low development effort by considering the use of C-based high-level hardware synthesis, instead of complex manual hardware designs. SPEXSIM not only exploits the spatially distributed model of computation commonly used on FPGAs, but can also model the effect of two different microarchitectures commonly used in C-to-hardware compilers.**

I. INTRODUCTION

With improvements in semiconductor technology no longer translating into direct gains in compute performance, alternatives to conventional out-of-order superscalar processors are receiving more attention from users. In recent years, this has been especially true for computing on Graphics Processing Units (GPU), which are now in common use for handling regular (array and vector-based) computations, in scenarios ranging from embedded to high-performance computing.

However, as not all computations map efficiently to GPUs (this includes irregular algorithms, dealing, for example, with sparse or graph-based data structures), further alternatives are being considered. This includes not only Many-Core processors such as the Intel Xeon Phi [1] or Kalray Bostan [2] that have 60–256 cores, but also the use of *reconfigurable computing units*.

The latter follow the idea that, instead of using a hardwired *general-purpose* computing structure programmed in software, it may be more efficient to create an *application-specific* computing structure directly in hardware which can operate autonomously (i.e. no longer needs a software program for control). One trade-off with this approach is that reconfigurable logic devices (most commonly Field-Programmable Gate Arrays) operate only at lower clock frequencies than hardwired circuits. Typical FPGA clock frequencies practically achievable today are 150–250 MHz, compared to the 2.1–3.5 GHz of many commercially available CPUs. On the power side, we observed that FPGAs rarely exceed a power draw of 30–60 W (much of that in high-speed external interfaces for server use, e.g. PCI Express), while high-performance processors such as an Intel E7 v3 or IBM POWER8 often draw 160–190 W [3]. This reduced power draw makes reconfigurable computing attractive not only in embedded scenarios, but also for data center use [4].

A key hindrance in successfully using FPGA-based reconfigurable computing, however, is the difficulty of programming such systems. The term "programming" here is misleading, as what actually needs to be done is the *hardware design* of a custom processing unit for the specific application. This not only requires expertise in digital circuit design and computer architecture, but also the use of specialised languages (e.g. Verilog or VHDL) and a complex design flow (simulation, synthesis, place & route, timing analysis, etc.). Very few application programmers will be familiar with these techniques.

As an alternative, high-level hardware synthesis tools (HLS) [5] aim to enable the *automatic* creation of digital circuits from high-level descriptions. These most commonly use a subset of C or C++, or sometimes a domain-specific language (e.g. MATLAB for signal processing). Despite the many advances in the field, often significant rewriting of the code (restructuring or enrichment with specialised **#pragma**s) is required to get the code to compile (as different HLS tools often have different restrictions on the code, e.g. no pointer operations, only regular control flow etc.). Even when the code does compile, additional rewriting may be required to improve performance (e.g. add directives to enable loop unrolling, request loop pipelining with a specified initiation interval, localise memories etc.).

When FPGAs were used to accelerate very specific computations of an application, it generally sufficed to invest the development effort for a conventional hardware design or for HLS-based programming on a few select application regions (so-called *kernels*). With this methodology, excellent efficiency has been demonstrated, e.g. for cryptographical operations, low-level signal processing such as filters or FFTs, or string/sequence pattern matching. But when FPGAs act as general-purpose accelerators, either in reconfigurable system-on-chips [6], or in data center computing [7], surveying large legacy code bases to discover promising areas for "easy" HLS-based acceleration on the FPGAs now available in the system quickly becomes difficult.

The traditional approach would be to profile the application to discover "hot spots" in the code and then optimise these parts of the program. However, due to the clock speed difference, not all code is well suited for acceleration on the FPGA. The FPGA will enable performance in most cases only if a high degree of finely granular parallelism, called *instruction level parallelism* (ILP) in computer architecture, is

present in the code. On a processor, a computation containing these operations would be executed in a *temporally distributed* fashion, re-using a limited number of hardware units (on the order of eight for current super-scalar processors) to execute them sequentially. The computing paradigm commonly used on FPGAs, however, relies on the *spatial distribution* of the computation. In the extreme case, each operation in the code will be mapped to a dedicated hardware unit. In code with a high degree of ILP, these units will actually be able to operate in parallel. Some input programs can thus result in very deep hardware processing pipelines [8] with hundreds of stages, each containing multiple operators, all executing in parallel. In such cases, the high degree of compute parallelism on the FPGA easily outweighs the clock frequency difference to the hard-wired CPU, allowing an accelerator running at 180 MHz on the FPGA to achieve a speedup of 7.3x over a multi-threaded quad core CPU clocked at 2.4 GHz (and drawing 5x the power of the FPGA).

Quickly surveying large code bases for such acceleration potential thus requires a tool that actually takes the spatial computing nature of the FPGA into account. This is made even more complex by the need to consider the flexibility of realizing *arbitrary* processing hardware: The same computation can be realised in many different microarchitectures on the FPGA. Extreme examples include running the algorithm actually in software on a *soft-core* processor implemented on the FPGA (generally not very efficient), or map them to very efficient lock-step systolic arrays, or fully dynamically scheduled compute structures [9] that can potentially deal well with very complex control flows.

We present SPEXSIM, a software tool that can quickly analyse large code bases to determine which kernels are interesting for FPGA acceleration using HLS. SPEXSIM not only estimates the temporally and spatially distributed execution times for each kernel, it also considers two different micro architectural models for its analysis. The two models were chosen since they are representative of the microarchitectures actually generated by current HLS tools. The *Blockwise* model is employed, e.g. by LegUp [10], while the *Pipelined* model is used in Nymble [11]. As will be seen in the evaluation (Section V), different kernels might map better to one or the other of the execution models.

We do not claim that SPEXSIM can accurately predict actual speed-ups of FPGA implementations, as attempting this would have to take too many variables (including e.g. detailed models of the memory hierarchy) into account. Also, it would require a full high-level tool flow, which in itself can take minutes to hours (e.g. when using advanced loop pipelining using modulo scheduling). Instead, our tool is intended to guide developers to focus their manual examination on very specific areas of the code in order to exploit the FPGAs present in their target compute platforms.

II. Related Work

Sotomayor et al. [12] recently presented AKI, a tool to detect hotspots in an application and classify their potential to be parallelised and mapped to a heterogeneous computing system (consisting of CPUs, GPUs and FPGAs), according to static souce code metrics. In contrast, SPEXSIM is tailored specifically to analyse the amount of fine-grained parallelism in sequential algorithms than can be exploited by high-level synthesis for FPGAs.

Guo et al. [13] investigated the components, including the available ILP to benefit a spatially distributed computation, of the speed-up that FPGA accelerators can achieve over processors in image processing applications. At the time when VLIW architectures were popular in CPU design, Jouppi and Wall [14] simulated several execution schemes employed by general-purpose processors (e.g. super-scalar, super-pipelined, VLIW) to measure the available instruction level parallelism in a set of benchmark programs. A similar study, also in the context of CPUs, was conducted by Butler et al. [15].

III. Simulation of Spatial Execution

The proposed pertinence analysis is based on the comparison of the estimated runtime of *kernels* when executed with different *execution models*. In general, a kernel is a region of the input code that has a significant influence on the overall runtime of a given application, and therefore would potentially benefit from hardware acceleration.

We currently consider all loops to be potential kernels. However, our approach could be restricted to perform the runtime estimation only on loops preselected by other analyses, or could be extended to operate on a broader range of single-entry regions, e.g. a mix of loop and non-loop code marked by user specified pragmas.

A. Representation of kernels

Compiler frameworks targeting general purpose processors traditionally utilise a control flow graph (CFG) [16] as an intermediate representation (IR) for optimisations and as input to the code generation subsystem. The CFG is also the usual starting point for high-level synthesis (HLS) systems, as these rely heavily on existing compiler frameworks from the software domain. Being a common element in the two types of compilation flows we want to compare, the CFG is the ideal IR to define our pertinence analysis on.

A CFG consists of *basic blocks*, i.e. sequences of RISC-like instructions without control-flow changes, and the control flow transitions between them. In Figure 1e), the CFG for the example loop in Figure 1d) is shown, consisting of two basic blocks.

B. Execution models

The simplest way of executing the computation as represented by a kernel's CFG is to sequentially execute both the basic blocks as well as the instructions contained in them, just as a non-superscalar processor would do. This defines our *CPU* model. Figure 1a) shows the execution of two iterations of the example loop in Subfig. d), according to the CPU model. We assume that typical compiler optimisations were run, especially common subexpression elimination to detect

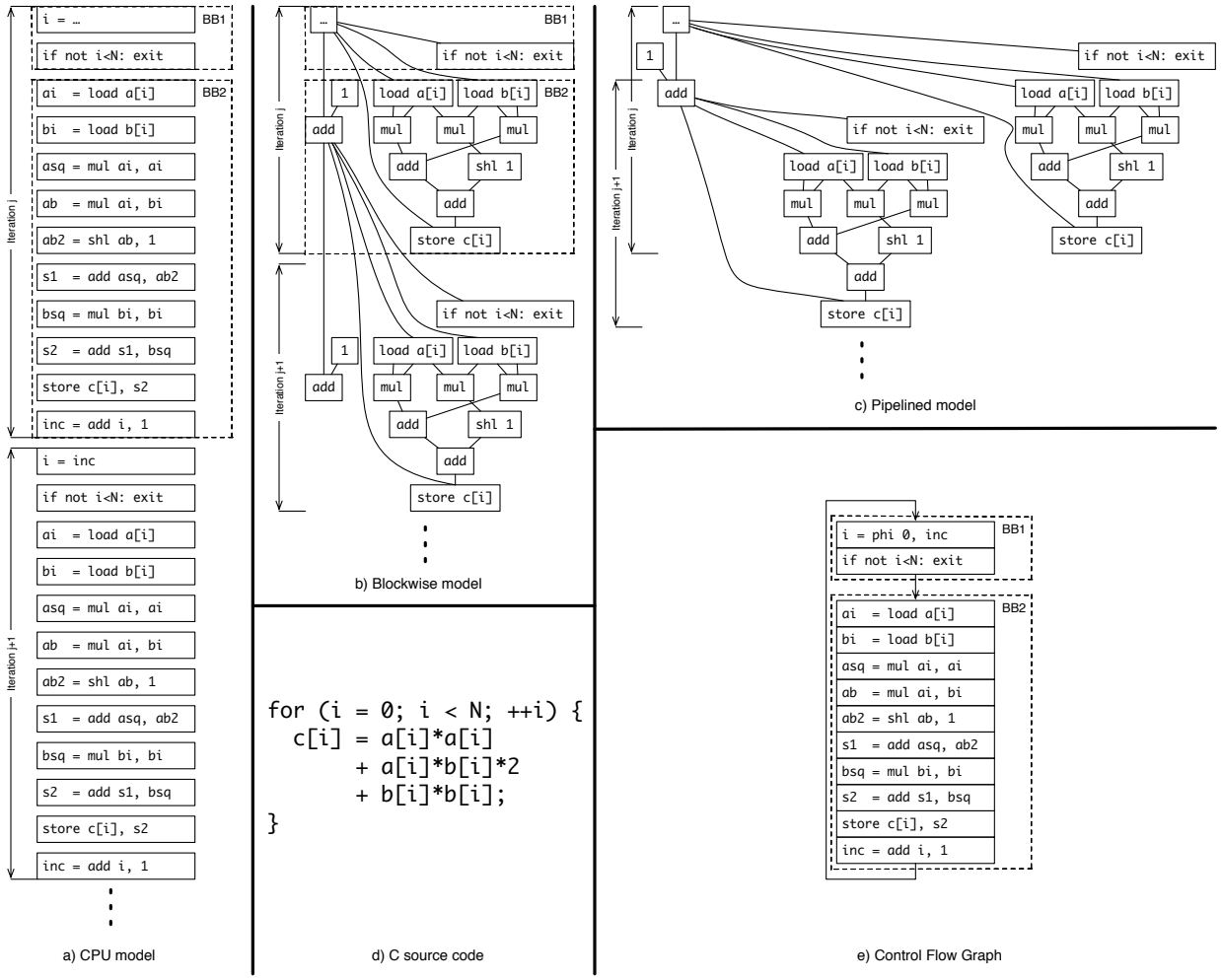

Fig. 1. Execution of two subsequent iterations of an example loop, according the presented execution models

that the results of the load instructions can be reused.

However, BB2 contains much instruction-level parallelism, as not all subexpressions depend on each other. This can be exploited by a blockwise spatial execution on an FPGA: A dataflow graph (DFG), where each instruction is mapped to an individual operator module, is constructed for the computation in each basic block. This defines our *Blockwise* model, exemplified in Figure 1b). We assume that enough hardware resources are availble to perform all independent operations in parallel. Note that iterations j and $j+1$ and blocks BB1 and BB2 are still executed sequentially, but the computation inside the blocks is much shorter due to the spatial execution, resulting in an earlier completion time of $j+1$ compared to the CPU model.

Observe that the example loop's iterations are mostly independent - the increment of the loop counter is the only inter-iteration dependency. This makes the kernel amenable to loop pipeling, i.e. issuing a new iteration after a fixed amount of time, resulting in a partial overlapping of iterations, as shown in Figure 1c). This defines our *Pipelined* model.

Loop pipelining requires the presence of a dataflow graph per loop, across the basic block structure. To this end, control-flow within the loop has to be transformed into dataflow as well, usually by adding predicates to operations that may not be executed speculatively (e.g. memory accesses). Note that in Figure 1c), these predicates are not shown for brevity. As a secondary effect, the larger one-graph-per-loop scope may expose more ILP than was available on the basic block level.

C. Estimation

We now present a kernel runtime estimation based on the execution models introduced in the previous section. The basic idea is to compute a numerical value (in an abstract time unit) that shall be interpreted as the execution time of each block or loop, and then multiply it with a factor representing how often a given block or loop is executed in a typical run of the program.

To this end, we define latency(I) to represent the number of time steps that an instruction I needs to complete, and determine the execution frequency β_B for each basic block B. We discuss the sources of these parameters in Section IV.

Note that the instructions in a basic block are always executed together. We therefore do not need to determine how

often each individual instruction is typically executed, but can use the enclosing block's execution frequency to the same end.

1) CPU model: For the CPU model, we estimate that the execution time $E_{\text{CPU}}(B)$ of a single basic block B is the sum of the latencies of the instructions contained in it, and that the execution time $E_{\text{CPU}}(K)$ of the kernel K is the weighted sum of the execution times of all its basic blocks:

$$E_{\text{CPU}}(B) = \sum_{I \in B} \text{latency}(I) \tag{1}$$

$$E_{\text{CPU}}(K) = \sum_{B \in K} \beta_B \cdot E_{\text{CPU}}(B) \tag{2}$$

2) Blockwise model: In order to apply the Blockwise model to a kernel K, we have to construct the dataflow graph for each of the kernel's blocks and schedule the operations in this graph, i.e. a start time t_I has to be assigned to each operation I. The estimated execution time $E_{\text{FPGA\_bw}}(B)$ for a basic block B is equal to the time when the last operation finishes. We perform a simple as-soon-as-possible (ASAP) scheduling in order to minimise that time.

The schedule has to adhere to a number of precedence constraints:

- An operation can start only after all its operands have finished. This precedence is ensured by the dataflow edges.
- The schedule has to preserve the order of memory accesses in the presence of flow, anti and output dependencies. We use LLVM's alias analysis framework to determine whether a given pair of memory accesses possibly operates on the same memory location, and add a precedence edge accordingly.
- We conservatively handle calls to other functions like barriers, and introduce precedence edges for memory accesses and other calls that come before or after them.

The estimation $E_{\text{FPGA\_bw}}(K)$ for a kernel K is again computed as the weighted sum of its blocks' execution times.

$$E_{\text{FPGA\_bw}}(B) = \max_{I \in B}(t_I + \text{latency}(I)) \tag{3}$$

$$E_{\text{FPGA\_bw}}(K) = \sum_{B \in K} \beta_B \cdot E_{\text{FPGA\_bw}}(B) \tag{4}$$

3) Pipelined model: Before we can construct a mock-up of the kernel in accordance to the Pipelined model, we transform its contents to a hierarchical set of control-dataflow graphs [17]. Each graph represents a loop, containing the operations in the loop as well as special operations for each nested loop.

Again, we schedule each graph (now representing a whole loop instead of a single basic block) with the ASAP strategy, subject to the following precedence constraints:

- An operation can start only after all its operands have finished. This precedence is ensured by the dataflow edges.
- The schedule has to handle intra-iteration flow, anti and output dependencies between all pairs of memory accesses. We use LLVM's dependence analysis to determine whether such dependencies are present, and add the appropriate precedence edges.
- We conservatively handle calls to other functions and starts of nested loops like barriers, and introduce precedence edges for memory accesses, other calls and nested loops that come before or after them.

In addition to the intra-iteration dependencies, due to the overlapping execution of loop iterations, it is required to consider inter-iteration dependencies from earlier iterations as well. These dependencies occur naturally in the computation of loop-dependent values, like incrementing the iteration variable of a for-loop. However, memory accesses may contribute such dependencies as well, for example, when a memory location is read in one iteration after it was written in the previous iteration. We query LLVM's dependence analysis for these inter-iteration dependencies and record them. Conservatively, we assume that all dependencies have to hold between immediately neighbouring iterations. Modulo scheduling is a technique to compute a schedule that adheres to all intra- and inter-iteration dependencies when new loop iterations are started after a fixed *initiation interval (II)* [18], [19]. A small II results in a greater number of overlapping iterations, and in consequence, an earlier completion of the whole loop.

In order to estimate how amenable a loop is for pipelining, we use a simplified lower bound to estimate the II instead of performing actual modulo scheduling: It has to be greater than or equal to the length of the longest inter-iteration dependency edge in the graph. In addition to the already known ASAP times, we calculate the as-late-as-possible (ALAP) times of each operation. The length of an inter-iteration dependency edge $J \to I$ is then defined as $\text{ASAP}(J) - \text{ALAP}(I)$.

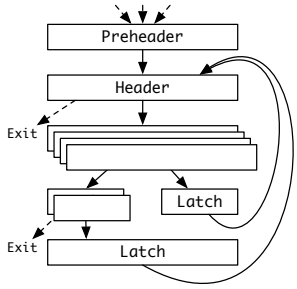

Fig. 2. Basic blocks of interest in a loop

Let P be L's preheader block, i.e. a unique predecessor block to the actual loop header, and let latches(L) denote the loop's latch blocks, i.e. the blocks that end with a branch back to L's header block, as shown in Figure 2. Then

$$n_{\text{avg}}(L) = \left(\sum_{B \in \text{latches}(L)} \beta_B \right) / \beta_P \tag{5}$$

is the average number of iterations for the loop, computed as the ratio of accumulated latch block execution frequencies β_B over the preheader's execution frequency β_P. This is an approximate measure of the times the program jumped back to the loop header over the times the loop was started.

For a single start of L, we know that the last iteration is started after $n_{\text{avg}} - 1$ initiation intervals, and one complete execution of the datapath is needed before the loop is finished. Thus:

$$E_{\text{FPGA\_pl, single}}(L) = (n_{\text{avg}}(L) - 1) \cdot II + (\max_{I \in L} t_I + \text{latency}(I)) \tag{6}$$

To get the estimation $E_{\text{FPGA\_pl}}(L)$ for the loop L, we multiply the single start estimation $E_{\text{FPGA\_pl, single}}(L)$ by the number of starts, as expressed by the preheader's execution frequency β_P:

$$E_{\text{FPGA\_pl}}(L) = \beta_P \cdot E_{\text{FPGA\_pl, single}}(L) \tag{7}$$

A kernel might have an acyclic part A that is executed only once. We account for it with:

$$E_{\text{FPGA\_pl}}(A) = \max_{I \in A}(t_I + \text{latency}(I)) \tag{8}$$

Putting it all together, the pipelined model's estimate $E_{\text{FPGA\_pl}}(K)$ for a kernel K is:

$$E_{\text{FPGA\_pl}}(K) = \left(\sum_{L \in \text{loops}(K)} E_{\text{FPGA\_pl}}(L) \right) + E_{\text{FPGA\_pl}}(A) \tag{9}$$

IV. Implementation

LLVM [20] is a state-of-the-art compiler framework on which both academic and commercial high-level synthesis systems [21] are based. Its intermediate representation (LLVM-IR [22]) is thus a viable environment to base our static analysis on. The level of abstraction in LLVM-IR resembles that of assembly code for a generic RISC processor. The basic blocks of a function are organised explicitly in a CFG. All functions as well as any global variables of a compilation unit, e.g. the input program, are grouped together in a module, which is the top-level construct in LLVM-IR. Loops are not modelled explicitly in the IR, but can be discovered by a built-in analysis pass.

SPEXSIM consists of 1) a modified version of LLVM's C frontend clang that annotates loops with their source file name and the line number on which the loop statement begins, and 2) a standalone tool linking against the LLVM libraries. Within this tool, we implemented the simulation of the execution models as well as the kernel extraction and our proposed instrumentation/profiling infrastructure (as presented in the next sections) as custom passes. These passes rely on the analyses (e.g. loop discovery, alias and dependence analysis) and transformation utilities available in the LLVM framework. Furthermore, we allow the user to apply all available optimisation passes to the input program by forwarding the corresponding command line parameters to LLVM's built-in optimisation driver.

A. Kernel extraction

In order to make the full range of LLVM optimisations applicable to the kernels without diluting their borders, it is required to extract the kernels into their own functions, and replace their original occurrence with a call to the extracted function.

Kernels can be nested directly (e.g. a loop inside another loop) or indirectly (e.g. a loop inside a function that is called from within another loop). In both cases, the kernels have to be estimated in a top-down fashion, e.g. the outermost kernels are extracted and simulated first (including all computations belonging to nested kernels). Afterwards, the kernels in the just extracted functions have to be discovered and extracted. This is continued until no unextracted kernels remain.

1) Depth: We found it useful to assign a nesting depth d_K to each kernel K, in order to specify the extraction order before actually extracting the kernels. We perform a top-down walk of the call graph, i.e. visiting all callers of a function `bar()` before visiting `bar()`, starting at the `main()` function. For every function, we determine an initial depth as the maximum depth of its call sites, or 0 in case of the main function. Then, we walk the loop tree inside the current function, increasing the depth for each kernel we encounter. Alongside the walk, we remember the maximum nesting depth among all kernels as d_{max}.

2) Successive extraction: For each nesting depth $d \in \{1, \ldots, d_{max}\}$ we generate a new intermediate version of the program by extracting the kernels K with $d_K = d$ based upon the previous extraction step. A copy of this intermediate version can then be arbitrarily optimised, instrumented and simulated, without influencing the next extraction step.

Figure 3 exemplifies this process. From left to right, the successive extraction, based on the discovered depth property, is shown. The example illustrates the advantage of this extraction order: At the time a kernel K is estimated, no other kernel inside or transitively called from within K has been extracted, ensuring that the estimation results will be the same as if K was the only kernel in the program. Furthermore, the module stays executable in all intermediate steps, making it possible to instrument and run it at any time.

B. Latencies

We employ an optimistic and a practical latency model:

In the optimistic latency model, we assume that all instructions need only one time step to complete, with the exception of some pseudo instructions (e.g. phi nodes, cast operations) that are needed for the consistency of the IR, but do not result in actual code or hardware. Function calls are accounted for with the estimation for the called function, or are associated with a fixed cost in case the callee is external or unknown.

The practical latency model is based on instruction latencies measured on an Intel Skylake processor [23] for the CPU model and operator latencies of pipelined FloPoCo [24] modules targeting 200 MHz for the FPGA models. Function calls are handled analogously to the optimistic model. Missing datapoints have been interpolated based on the authors'

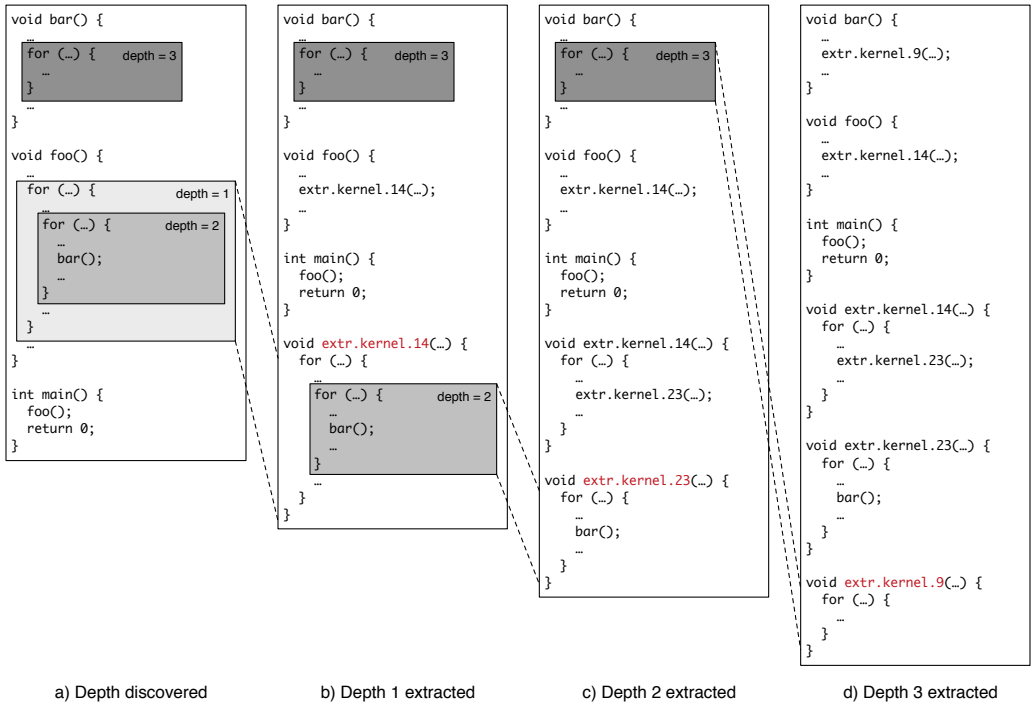

Fig. 3. Kernel depth and successive extraction

experience, e.g. that logical bitwise operations usually can be done combinatorically in 0 time steps on an FPGA.

C. Execution frequencies

The execution models require that an execution frequency β_B is available, estimating how often each basic block B, and therefore each LLVM-IR instruction, is executed in a typical run of the input program. The frequencies are always scaled to represent how often a block is executed on average during a single activation of its parent function F. For example, a block B that is part of a conditional branch is usually not executed every time F is called, resulting in $\beta_B < 1$. On the other hand, if B is part of a loop, it will potentially be executed multiple times during an activation of F, resulting in $\beta_B > 1$.

We currently support the following two means of acquiring these frequencies.

1) LLVM BlockFrequency analysis: LLVM includes the BlockFrequency analysis pass that uses heuristics to estimate how often certain branches are taken, and computes an execution frequency per basic block that is usable in our execution models. By using this analysis, the estimation process remains completely static. In principle, the accuracy of the BlockFrequency analysis can be improved by activating an instrumentation mechanism in the frontend. We found, however, that the resulting profiling information is hard to keep consistent across the kernel extraction and possible optimisation steps.

2) Custom instrumentation: To this end, we designed our own simple instrumentation framework based on inserting counters in each basic block. Inserting the instrumentation code at the IR level allows us to account for all transformations that were applied to a kernel. This mechanism delivers exact execution frequencies, at the slight disadvantage that these are tied to a particular set of runtime parameters and the resulting program execution.

V. Results

Tables II and III show the results of running SPEXSIM, linked against LLVM 3.7, on the publicly available CHStone [25] and MachSuite [26] benchmark suites. We used our custom instrumentation scheme and applied LLVM's -O3 optimisation preset to the kernels prior to the estimation. Kernels are identified by their filename and starting line number. The kernels are sorted by the fraction (Column '%') of the total program execution time they take in the software-based model. We show the top 30 kernels for both latency models. Column 7 and 9 display the estimated speed-up of Blockwise (Column 7) and Pipelined (Column 9) microarchitectures over the processor when FPGA and CPU are running at the *same* clock frequency.

A number of aspects must be considered when discussing these results. First, remember that these potential speedups are based on *C-based* HLS, translating the original software code into a hardware block without actually optimizing the algorithm for hardware. Thus, the speed-ups are rather low on a per-clock-cycle basis. For many of the kernels in this listing, *much* faster implementations designed as custom hardware are available. For example, custom FFT implementations ran more than 10x faster on a single FPGA than on a 6-core CPU clocked at 2.67 GHz [27]. That performance far exceeds the best-case speed-up of 3.31x (in terms of clock cycles) predicted by SPEXSIM for using C-based HLS.

Second, the threshold for interesting speed-ups is largely dependent on the clock frequency disparity between CPU and

FPGA. This might be a factor of just 3 (e.g. an embedded Cortex-A9 processor running at 667 MHz vs. an FPGA running at 220 MHz), but it also could exceed 10 (CPU running at more than 2.5 GHz) in a high-performance computing use-case. A number of kernels (e.g. sha, viterbi, nw) are predicted to have speed-up potential in the embedded use-case. Far fewer kernels would be competitive in the high-performance use-case (e.g. gemm, knn) from a pure performance perspective. But especially in data center use-cases, energy efficiency is becoming a key performance indicator. Conservatively assuming a 5x difference in power draw between FPGA (30 W) and CPU (150 W), even a kernel such as merge, which might run roughly at at third of the CPU performance (estimated 3.13x clock-cycle speed up, but 10x clock frequency differential), but would draw only a fifth of the power, still might come out ahead in terms of energy efficiency when targeting an FPGA. We compare against a single CPU core because HLS input programs often are purely sequential. Note that the (coarse-grained) parallelisation techniques required to make use of additional CPU cores would also be applicable to an FPGA design, e.g. by instantiating as many copies of the generated accelerator as fit on the reconfigurable device.

Third, when it is actually applicable, the Pipelined microarchitecture on the FPGA is capable of a much higher performance than the Blockwise architecture. However, it is efficient for only a few of the benchmarks (e.g. gemm, knn, viterbi, etc.). But having the flexibility of the FPGA, a high-level synthesis system can pick the micro architecture best suited for the specific kernel. As an example, the Nymble collection of compilers can generate five different microarchitectures (blockwise, pipelined, multi-threaded, resource-shared, and value-speculating), allowing a tight match between the capabilities of the microarchitecture and the needs of each kernel.

VI. Conclusion

We have introduced SpExSim as a tool to quickly survey large legacy code bases, searching for kernels potentially profiting from a low development effort mapping to an FPGA using C-based high-level synthesis. The tool targets a spatially distributed model of computation and considers two different hardware microarchitectures for the estimates. The interesting kernels discovered by SpExSim can then be manually examined for additional optimisation potential (e.g. adding performance-enhancing HLS directives).

References

[1] Intel Corporation, "The Intel Xeon Phi Product Family." [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf

[2] KALRAY Corporation, "MPPA2-256 (Bostan) processor." [Online]. Available: http://www.kalrayinc.com/kalray/products/#processors

This work was partially funded by the EU in the FP7 research project *REPARA* (ICT-609666).

[3] Johan De Gelas, "The Intel Xeon E7-8800 v3 Review: The POWER8 Killer?" [Online]. Available: http://www.anandtech.com/show/9193/the-xeon-e78800-v3-review/6

[4] Intel Corporation, "Intel Acquisition of Altera." [Online]. Available: https://newsroom.intel.com/press-kits/intel-acquisition-of-altera/

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, 2011.

[6] Xilinx Inc., "Zynq-7000 all programmable soc." [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[7] A. Putnam, A. M. Caulfield, E. S. Chung *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[8] J. Huthmann, P. Müller, F. Stock, D. Hildenbrand, and A. Koch, "Accelerating high-level engineering computations by automatic compilation of geometric algebra to hardware accelerators," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, 2010.

[9] H. Gädke and A. Koch, "Accelerating speculative execution in high-level synthesis with cancel tokens," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008.

[10] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, "From software to accelerators with LegUp high-level synthesis," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013.

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

[12] R. Sotomayor, L. M. Sanchez, J. G. Blas, A. Calderon, and J. Fernandez, "Aki: Automatic kernel identification and annotation tool based on c++ attributes," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, 2015.

[13] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of fpgas over processors," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, 2004.

[14] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, 1989.

[15] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3, 1991.

[16] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2006.

[17] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[18] R. B. Rau, "Iterative modulo scheduling," *International Journal of Parallel Programming*, vol. 24, no. 1, 1996.

[19] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," 2014.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, CGO*, 2004.

[21] R. Nane, V. M. Sima, C. Pilato *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[22] LLVM Compiler Infrastructure Project, "LLVM Language Reference Manual." [Online]. Available: http://llvm.org/docs/LangRef.html

[23] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," 2016. [Online]. Available: http://agner.org/optimize/instruction_tables.pdf

[24] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, 2011.

[25] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis," *Journal of Information Processing*, vol. 17, no. 4, 2009.

[26] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," 2014.

[27] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among CPU, GPU, and FPGA," 2013. [Online]. Available: http://m.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking_Final.pdf

Benchmark	File	Line	E_{SPP}	%	$E_{\text{FPGA\_bw}}$	$\frac{E_{\text{SPP}}}{E_{\text{FPGA\_bw}}}$	$E_{\text{FPGA\_pl}}$	$\frac{E_{\text{SPP}}}{E_{\text{FPGA\_pl}}}$
CHStone/dfsin	dfsin.c	173	92870	100 %	59799	1.55	66317	1.40
MachSuite/kmp/kmp	kmp.c	31	491995	100 %	426193	1.15	359561	1.37
CHStone/dfadd	dfadd.c	215	3641	100 %	2349	1.55	4375	0.83
CHStone/sha	sha.c	210	801077	100 %	307241	2.61	214295	3.74
CHStone/mips	mips.c	139	17359	99 %	11623	1.49	49418	0.35
CHStone/dfdiv	dfdiv.c	144	2122	99 %	1411	1.50	1985	1.07
CHStone/dfmul	dfmul.c	137	1465	99 %	903	1.62	1545	0.95
MachSuite/nw/nw	nw.c	30	560154	99 %	288526	1.94	164356	3.41
MachSuite/sort/radix	sort.c	84	1460642	94 %	1055170	1.38	1680818	0.87
MachSuite/md/grid	md.c	16	1076680	92 %	780502	1.38	384121	2.80
MachSuite/viterbi/viterbi	viterbi.c	18	8576719	90 %	6263620	1.37	1263375	6.79
MachSuite/backprop/backprop	backprop.c	264	34302220	90 %	17592430	1.95	12321660	2.78
CHStone/jpeg	decode.c	423	1478800	90 %	829447	1.78	1018783	1.45
MachSuite/sort/merge	sort.c	38	563191	86 %	341895	1.65	180184	3.13
MachSuite/gemm/ncubed	gemm.c	8	3174722	81 %	1855810	1.71	286916	11.06
MachSuite/gemm/blocked	gemm.c	15	2158882	74 %	1245474	1.73	1069404	2.02
CHStone/blowfish	bf.c	840	447726	67 %	196824	2.27	1912959	0.23
CHStone/aes	aes_dec.c	116	17789	62 %	5279	3.37	2758	6.45
MachSuite/aes/aes	aes.c	191	7219	62 %	2223	3.25	2693	2.68
CHStone/adpcm	adpcm.c	846	34812	53 %	12352	2.82	11356	3.07
CHStone/sha	sha.c	164	401295	50 %	154381	2.60	107784	3.72
MachSuite/stencil/stencil3d	stencil.c	14	662583	47 %	206223	3.21	119794	5.53
CHStone/gsm	lpc.c	134	8531	47 %	7148	1.19	6845	1.25
MachSuite/stencil/stencil2d	stencil.c	7	454114	46 %	117813	3.85	86440	5.25
CHStone/motion	getbits.c	168	717	46 %	251	2.86	204	3.51
CHStone/adpcm	adpcm.c	850	30050	46 %	6602	4.55	8709	3.45
CHStone/motion	getbits.c	66	738	43 %	347	2.13	141	5.23
CHStone/blowfish	bf_skey.c	144	209924	32 %	89603	2.34	85510	2.45
MachSuite/md/knn	md.c	24	156930	30 %	104962	1.50	11267	13.93
MachSuite/fft/strided	fft.c	8	203874	23 %	114768	1.78	61504	3.31

TABLE I
ESTIMATION RESULTS FOR LOOPS FROM CHSTONE AND MACHSUITE WITH OPTIMISTIC LATENCIES

Benchmark	File	Line	E_{SPP}	%	$E_{\text{FPGA\_bw}}$	$\frac{E_{\text{SPP}}}{E_{\text{FPGA\_bw}}}$	$E_{\text{FPGA\_pl}}$	$\frac{E_{\text{SPP}}}{E_{\text{FPGA\_pl}}}$
CHStone/dfsin	dfsin.c	173	127134	100 %	70640	1.80	135484	0.94
MachSuite/kmp/kmp	kmp.c	31	623575	100 %	755203	0.83	1012846	0.62
CHStone/sha	sha.c	210	1258823	100 %	541503	2.32	330813	3.81
CHStone/dfadd	dfadd.c	215	5380	99 %	3780	1.42	10458	0.51
CHStone/mips	mips.c	139	25294	99 %	20359	1.24	182408	0.14
CHStone/dfdiv	dfdiv.c	144	3847	99 %	2120	1.81	3822	1.01
MachSuite/nw/nw	nw.c	30	773786	99 %	534414	1.45	573956	1.35
CHStone/dfmul	dfmul.c	137	2242	99 %	1428	1.57	3276	0.68
MachSuite/md/grid	md.c	16	2976547	94 %	2181327	1.36	1180726	2.52
MachSuite/backprop/backprop	backprop.c	264	77406150	93 %	60706090	1.28	53545340	1.45
MachSuite/sort/radix	sort.c	84	2182114	92 %	2866162	0.76	4391218	0.50
MachSuite/viterbi/viterbi	viterbi.c	18	16566160	91 %	14857430	1.12	3736464	4.43
CHStone/jpeg	decode.c	423	2322856	86 %	1714310	1.35	3044369	0.76
MachSuite/sort/merge	sort.c	38	903136	82 %	702156	1.29	376685	2.40
MachSuite/gemm/ncubed	gemm.c	8	5804354	81 %	4235522	1.37	1356292	4.28
MachSuite/gemm/blocked	gemm.c	15	4849954	78 %	5161250	0.94	4997724	0.97
CHStone/blowfish	bf.c	840	732306	66 %	351010	2.09	2318951	0.32
MachSuite/aes/aes	aes.c	191	11838	57 %	7405	1.60	10248	1.16
CHStone/adpcm	adpcm.c	846	56608	53 %	24502	2.31	26416	2.14
CHStone/gsm	lpc.c	134	15236	51 %	22201	0.69	21898	0.70
CHStone/aes	aes_dec.c	116	29624	50 %	8168	3.63	9522	3.11
CHStone/sha	sha.c	164	633874	50 %	272405	2.33	166799	3.80
CHStone/motion	getbits.c	168	1514	47 %	645	2.35	358	4.23
CHStone/adpcm	adpcm.c	850	49822	46 %	15002	3.32	22227	2.24
MachSuite/stencil/stencil2d	stencil.c	7	821286	45 %	203619	4.03	172372	4.76
MachSuite/md/knn	md.c	24	467202	41 %	285698	1.64	42755	10.93
CHStone/motion	getbits.c	66	1254	38 %	983	1.28	779	1.61
MachSuite/stencil/stencil3d	stencil.c	14	939784	37 %	318723	2.95	256594	3.66
CHStone/blowfish	bf_skey.c	144	359946	33 %	108548	3.32	91675	3.93
MachSuite/fft/strided	fft.c	8	420990	29 %	381028	1.10	332894	1.26

TABLE II
ESTIMATION RESULTS FOR LOOPS FROM CHSTONE AND MACHSUITE WITH PRACTICAL LATENCIES