Journal of Supercomputing manuscript No.
(will be inserted by the editor)

SpExSim: Assessing Kernel Suitability for C-based High-Level Hardware Synthesis

Julian Oppermann · Lukas Sommer · Andreas Koch

Received: date / Accepted: date

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, including pipelined architectures with modulo scheduling. The estimations have been validated against actual hardware generated by two current HLS tools.

Keywords reconfigurable computing · FPGA · hardware acceleration · high-level synthesis · estimation · legacy code

1 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

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

Julian Oppermann
Hochschulstr. 10
64289 Darmstadt, Germany
Tel.: +49-6151-16-22432
E-mail: oppermann@esa.tu-darmstadt.de

Lukas Sommer
E-mail: sommer@esa.tu-darmstadt.de

Andreas Koch
E-mail: koch@esa.tu-darmstadt.de

(GPU), which are now in common use for handling regular (array and vector-based) computations.

However, as not all computations map efficiently to GPUs (e.g. 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 or Kalray Bostan [1], but also the use of *reconfigurable computing units*.

A key hindrance in successfully using FPGA-based reconfigurable computing, however, is the fact that only very few application programmers are familiar with the techniques required to design an accelerator for such a system.

As an alternative, high-level hardware synthesis tools (HLS) [3] aim to enable the *automatic* creation of digital circuits from high-level descriptions. Despite the many advances in the field, the use of HLS tools often requires significant rewriting of the code to make it compatible with the tools' requirements (e.g. no pointer operations or variable-bounded loops) or to improve performance (e.g. by using loop unrolling). When FPGAs are to be employed as general-purpose accelerators, either in reconfigurable system-on-chips [19], or in data centre computing [14], developers need to be able to survey large legacy code bases to discover promising areas for "easy" HLS-based acceleration on FPGAs.

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 increase performance in most cases only if a high degree of finely granular parallelism, called *instruction-level parallelism* (ILP), 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, whereas the computing paradigm commonly used on FPGAs relies on the *spatial distribution* of the computation: In code with a high degree of ILP, operations can be mapped to dedicated hardware units operating in parallel, with the high degree of compute parallelism allowing the FPGA to compensate the clock frequency difference to the hard-wired CPU.

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 becomes even more complex due to the fact that the same computation can be realised in many different microarchitectures on an FPGA.

We therefore present SpExSim, a software tool that can quickly analyse large code bases to determine which kernels are interesting for FPGA acceleration using HLS. 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. SpExSim not only estimates the spatially distributed execution times for each kernel, it also

[1] http://www.kalrayinc.com/kalray/products/#processors

considers two different micro architectural models used by current HLS tools for its analysis. The *Blockwise* model is employed, e.g. by LegUp [1], while the *Pipeline* model is used in Nymble [7]. As will be seen in the evaluation (Section 6), 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 quickly guide developers to focus their manual examination on very specific areas of the code in order to exploit FPGA-based compute accelerators.

2 Related Work

Sotomayor et al. [17] 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 source code metrics. In contrast, SpExSim combines dynamic profiling and static analysis and is tailored specifically to analyse the amount of fine-grained parallelism in sequential algorithms that can be exploited by high-level synthesis for FPGAs.

Performance modelling for FPGAs has been addressed by a number of approaches [18, 16, 13, 6]. These approaches typically employ an analytic model to predict different performance characteristics, such as computational throughput, memory bandwidth, or hardware area for the FPGA-based implementation of a given algorithm. The aim of these tools is to aid the developer in improving an FPGA implementation of an algorithm by providing up-front estimations of performance impacts caused by different optimisations, such as loop unrolling or memory coalescing.

However, none of these tools guides the developer in selecting parts of an software-only application for hardware/software co-execution. SpExSim, in contrast, was designed to identify the parts of an application most suitable for FPGA-based hardware acceleration in a heterogeneous system, based on the existing software source code.

3 Estimation of Spatial Execution

The proposed analysis is based on the comparison of the estimated runtime of kernels when executed with different *execution models*. 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.

3.1 Representation of kernels

Compiler frameworks targeting general purpose processors traditionally utilise a control flow graph (CFG) 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 analysis on.

The nodes of a CFG are *basic blocks*, i.e. sequences of RISC-like operations without control-flow changes, whereas the graph's edges represent the control flow transitions between the blocks. In Figure 1b), the CFG for the example loop in Listing 1a) is shown, consisting of two basic blocks.

3.2 Execution models

Starting from the CFG, typical HLS-systems construct another IR, which is then used for the synthesis of the hardware accelerator. In SpExSim we provide a model for the two most common choices of IRs used in HLS, the *Blockwise* model and the *Pipeline* model. The following sections describe both models in detail.

3.2.1 Blockwise model

In Figure 1b), BB2 contains much instruction-level parallelism, as not all subexpressions depend on each other. In order to exploit this fine-grained intra-block parallelism by a blockwise spatial execution on an FPGA, this parallelism is expressed by a distinct dataflow graph (DFG) *per block* (as opposed to the per-function/per-kernel representation in the pipeline model). Each operation is mapped to an individual hardware operator module.
In the DFG, a number of precedence constraints has to be represented by edges:

- An operation can start only after all its operands have been computed. 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 potentially 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.

An example for the Blockwise model is shown in Figure 1c). Note that iterations j and $j+1$ and blocks BB1 and BB2 are still executed sequentially, but

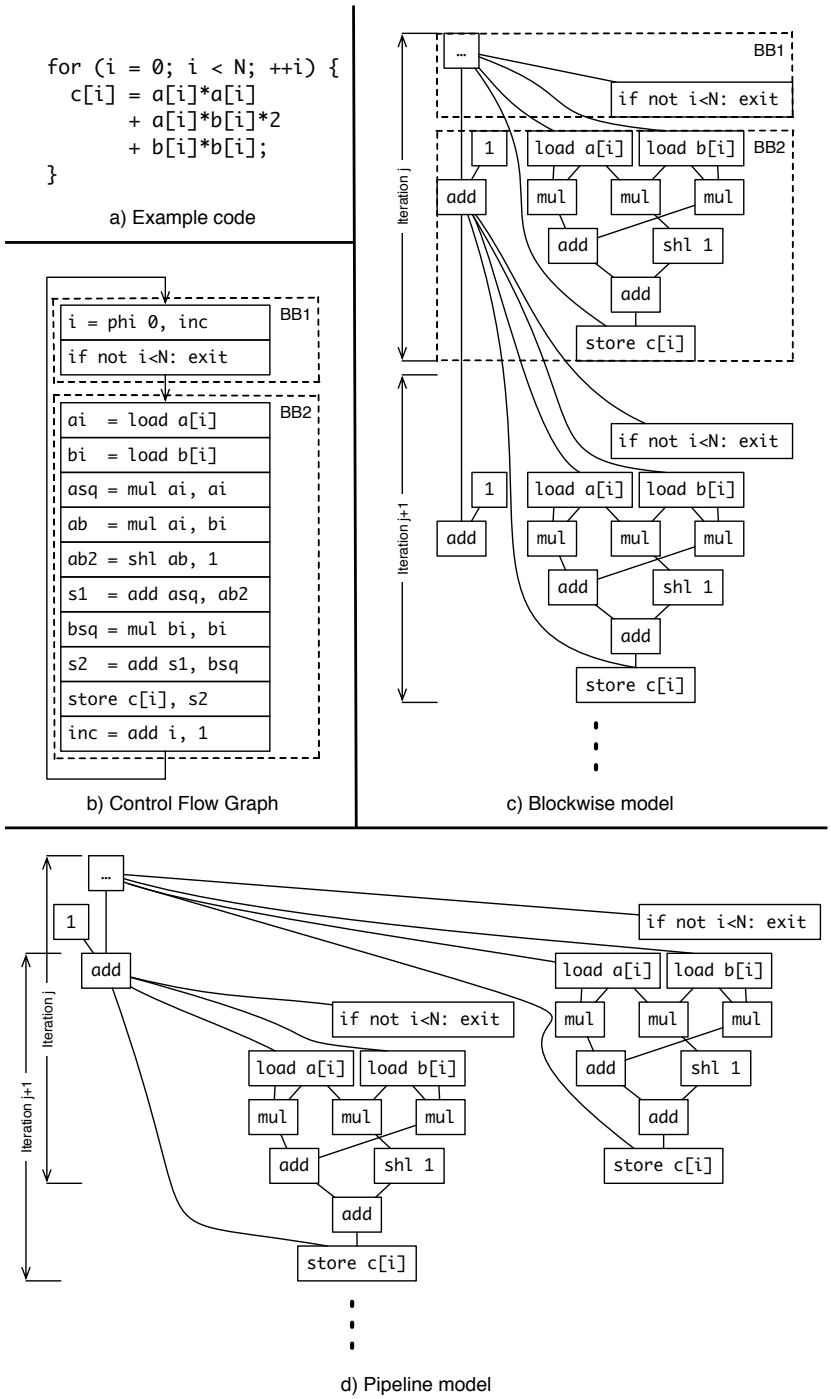

Fig. 1: Execution of two subsequent iterations of an example loop, according the presented execution models

the computation inside the blocks is much shorter due to the spatial execution, resulting in an earlier completion time of $j + 1$.

3.2.2 Pipeline model

In the same example, the loop iterations are mostly independent - the increment of the loop counter is the only inter-iteration dependency. This makes the kernel amenable to loop pipelining, i.e. issuing a new iteration after a fixed amount of time, the so-called *initiation interval (II)*, ideally before the current iteration completes. As a consequence, the iterations of the loop are partially overlapped, as shown in Figure 1d).
Loop pipelining requires the presence of an inter-block dataflow graph covering the entire loop [4], which may also expose more ILP than was available in the per-block DFGs in the Blockwise model. Each graph in the resulting hierarchical set of so-called control-dataflow-graphs (CDFG) comprises not only the operations in the loop and special operations for each nested loop, but must also incorporate the loop's controlflow, as the CFG-structure of the loop is fully resolved in a CDFG. Control flow is therefore represented by conditional dataflow and by adding predicates to operations that may not be executed speculatively (e.g. memory accesses). Different branches in the CFG are expressed as parallel dataflows, where the length of all dataflows must be balanced.

The constructed graph again has to model 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 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.

Note that for brevity in Figure 1d), control-flow predicates are not shown.
In addition to the intra-iteration dependencies, due to the overlapping execution of loop iterations, we need 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. Inter-iteration dependencies are represented by backedges in the CDFG.

3.2.3 Scheduling

Now that we have obtained a set of graphs for each of our models, we need to assign a starting time t_{op} to each operation *op* to assess the overall latency of a block or loop, respectively. Besides the precedence constraints represented by edges in the graphs, the scheduler has to adhere to a number of additional constraints:

- Resource constraints: Often the maximum number of concurrent operations of a kind is constrained in HLS systems. For example, the number of available ports to memory is typically limited. We therefore introduce resource constraints, limiting the maximum number of concurrent accesses to memory in each clock cycle.
- Operator chaining: In HLS, a sequence of operations can be scheduled to the same clock cycle, if their overall combinatorial latency does not exceed the maximum clock period. We model this HLS optimisation technique using simple weight-based approach: Each operation is assigned a weight based on the expected combinatorial latency of the operation. The scheduler is then allowed to chain a sequence of operations, as long as their accumulated weight does not exceed a user-defined, per-cycle maximum weight.

Another information required by the scheduler is the number of time steps that an operation *op* needs to complete, which we define as latency(*op*). In Section 4 we discuss how this parameter is determined.
For the resource-constrained scheduling of the graphs, we implement a list-scheduler, which provides us with a sufficient quality of scheduling results in reasonable time. Our list-scheduler uses a mobility-based priority function, where the mobility of each node is computed from the as-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) schedules [12].
While the list-scheduler obeys all inter-iteration dependencies, it violates the resource constraints in case multiple iterations of a loop are overlapped. To counter this, we additionally provide an implementation of the *Moovac*-scheduler [11], a modulo-scheduler optimising the II for maximum overlap. This makes better use of the pipelining opportunities in the kernel, resulting in an earlier completion of the entire loop.

3.3 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 determine the execution frequency β_B for each basic block B by dynamic profiling, as discussed in Section 4. Note that the operations

in a basic block are always executed together. We therefore do not need to determine how often each individual operation is typically executed, but can use the enclosing block's execution frequency to the same end.

3.3.1 Blockwise model

In order to estimate the runtime of a kernel K in the Blockwise model, we use the schedule we have computed for each block's DFG as discussed in the previous section. The estimated execution time $E_{\mathrm{FPGA\_bw}}(B)$ for a basic block B is equal to the time when the last operation finishes.

$$E_{\mathrm{FPGA\_bw}}(B) = \max_{op \in B}(t_{op} + \mathrm{latency}(op)) \tag{1}$$

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

$$E_{\mathrm{FPGA\_bw}}(K) = \sum_{B \in K} \beta_B \cdot E_{\mathrm{FPGA\_bw}}(B) \tag{2}$$

3.3.2 Pipeline model

In order to estimate how amenable a loop L is for pipelining, we use the II and operation start times t_{op} resulting from the CDFG's schedule, as discussed in Section 3.2.

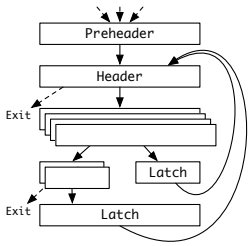

Fig. 2: Basic blocks of interest in a loop L

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

$$n_{\mathrm{avg}}(L) = \left(\sum_{B \in \mathrm{latches}(L)} \beta_B \right) / \beta_P \tag{3}$$

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:

$$E_{\text{FPGA\_pl, single}}(L) = (n_{\text{avg}}(L) - 1) \cdot II + (\max_{op \in L} t_{op} + \text{latency}(op)) \quad (4)$$

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

$$E_{\text{FPGA\_pl}}(L) = \beta_P \cdot E_{\text{FPGA\_pl, single}}(L) \quad (5)$$

A kernel might have an arbitrarily complex acyclic part A that is executed only once. We also construct a CDFG for this acyclic part, schedule it, and account for its execution time with:

$$E_{\text{FPGA\_pl}}(A) = \max_{op \in A}(t_{op} + \text{latency}(op)) \quad (6)$$

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

$$E_{\text{FPGA\_pl}}(K) = \left(\sum_{L \in \text{loops}(K)} E_{\text{FPGA\_pl}}(L) \right) + E_{\text{FPGA\_pl}}(A) \quad (7)$$

4 Implementation

LLVM [9] is a state-of-the-art compiler framework on which both academic and commercial high-level synthesis systems [10] are based. Its intermediate representation LLVM-IR is a viable environment for our proposed analysis tool. The level of abstraction in LLVM-IR resembles that of assembly code for a generic RISC processor. The basic blocks of a function are explicitly organised 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 estimations according to 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.

4.1 Kernel extraction

In order to apply the full range of LLVM optimisations, but limited to the scope of individual loops, these kernels are extracted into separate functions.

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 should be extracted in a top-down fashion, i.e. beginning with the outermost ones. With this 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.

4.1.1 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}.

4.1.2 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 prior to the estimation step, without influencing the next extraction step.

Figure 3 illustrates this process. From left to right, the successive extraction, based on the discovered depth property, is shown. The module stays executable in all intermediate steps, making it possible to instrument and run it at any time.

4.2 Latencies

In order to schedule the graphs, we need to determine the latency of each operation, i.e. the number of time steps needed for an operation's completion. We deduced a realistic latency model from different sets of latency parameters used in existing HLS systems, e.g. Legup or Nymble. The latency model associates each operation with a latency, e.g. a single-precision floating point multiplication takes 8 cycles in the Nymble model. Operations may have a latency of zero, e.g. combinatorial operations such as integer additions, and several pseudo operations (e.g. type casts) that are needed for the consistency

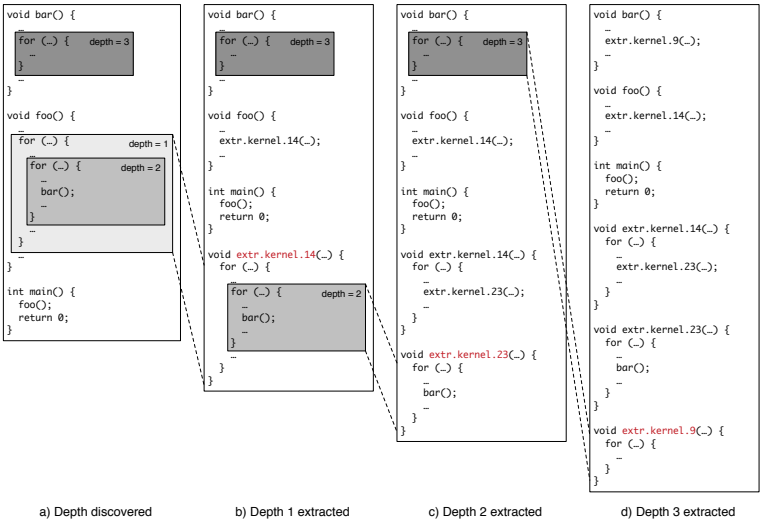

Fig. 3: Kernel depth and successive extraction

of the IR, but do not result in actual 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.

4.3 Execution frequencies

The execution models require that an execution frequency β_B is available, estimating how often each basic block B, and therefore each operation, 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 $\beta_B > 1$.

To acquire these frequencies, 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 common to all dynamic profiling approaches that these are tied to a particular set of runtime parameters.

5 Model validation

In order to confirm the validity of our execution models, we now compare the estimations computed by the SpExSim tool to cycle counts obtained by RTL

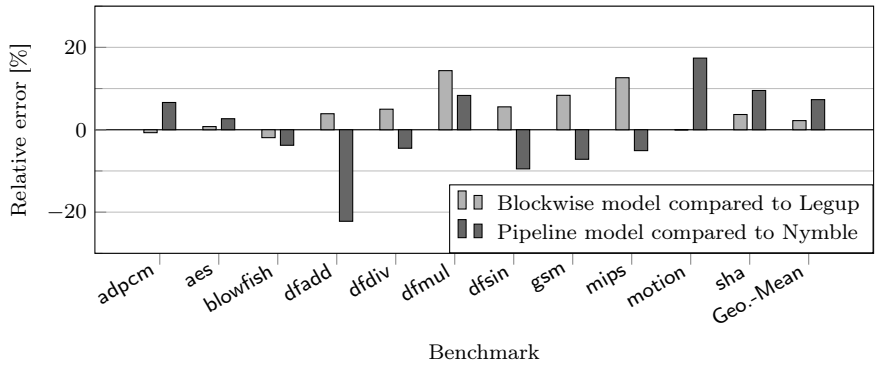

Fig. 4: Relative errors in estimated cycle counts (compared to simulated HLS-generated hardware) for benchmarks from the CHStone suite

simulation for actual hardware accelerators generated by the HLS tools LegUp 4.0 [1] and Nymble [7].

Note that the execution models only apply to the actual computation, and SpExSim intentionally does not try to model tool-specific properties besides the employed microarchitecture, such as the memory/cache architecture (e.g. [8]) or additional delays inherent to the controller implementation. However, these omitted properties can have a large impact of the simulated cycle counts, so for the sake of this comparison, we created two specialised versions of our tool to mimic LegUp and Nymble as closely as possible.

HLS tools usually leverage common functionality from software compiler frameworks for analysing and optimising the input program. We observed that variations in this preprocessing step, i.e. the LLVM version as well as the selection and order of applied passes, had great influence on the accuracy of the estimations. To this end, we tapped into the HLS tools' compilation flows and invoked SpExSim on the intermediate representation of the program just before the actual synthesis step.

We use benchmark applications from the CHStone suite [5] that come with pairs of golden inputs and outputs. The estimations are based on the exact execution frequencies that are obtained by our custom instrumentation method, and correspond to a single execution of the respective application with these predefined inputs.

LegUp For the comparison with LegUp, we compiled and simulated the CHStone benchmarks with LegUp's pure hardware flow that translates the entire program, i.e. its `main()` function, to an RTL description. Accordingly, we instructed SpExSim to perform the estimation for the `main()` function as the only kernel. We also mimic LegUp's loop unrolling behaviour by using the same threshold value of 192.

During the estimation, we limit the scheduler to place chains of up to 4 combinatorial operations into a time step. Additionally, the scheduler obeys the limit of at most two concurrent memory accesses per time step as imposed by the memory system generated by LegUp. Depending on the scope (e.g. "global" or "local") a memory location is accessible from, reading from memory takes either 1 or 2 cycles [2]. We enhanced our latency model to respect this by approximating the actual partitioning algorithm with a simple heuristic.

Nymble Nymble expects the user to add `#pragma`s to the source code to specify which part of the program should be compiled to a hardware description, and extracts this region into a separate function. All function calls within the scope of the hardware `#pragma`s are fully inlined into the hardware function. We let SpExSim perform the estimation on this hardware function.

We deactivated the cache simulation that is normally used in the simulation environment for Nymble-generated accelerators, and instead assume that memory accesses finish within a single clock cycle. The number of concurrent memory accesses is limited to 1 load and 1 store per clock cycle. As in the LegUp-specific SpExSim version, chains of up to 4 combinatorial operations are placed into a single time step.

As Nymble is not yet able to exploit the modulo schedules as computed by the Moovac algorithm to the same extent as the Pipeline model in SpExSim, we use the list schedulers in both systems and do not apply loop pipelining in this validation. This setup still allows us to show that the estimation of a loop's single-iteration latency conforms to its actual execution within the generated accelerator.

The plot in Figure 4 shows the relative errors of the Blockwise estimation compared to the simulation of a LegUp-generated hardware accelerator, and the Pipeline estimation compared to a Nymble-generated hardware module. The geometric mean of the absolute values of the errors is 2.26 % for the Blockwise model, and 7.31 % for the Pipeline model. Please note that for the largest relative errors (dfmul/Blockwise, dfadd, motion/Pipeline), the absolute difference between estimation and simulation is less than 300 cycles.

6 Results

Now that we have successfully validated our estimation for both execution models in the previous section, we now discuss SpExSim's use to survey large codebases for HLS-opportunities in greater detail. For heterogeneous platforms, it is common to only extract compute-intensive parts of a program to the hardware accelerator, so we present estimation results per kernel. To this end, we introduce the notion of a *significant kernel*, which we define as a loop whose execution makes up for at least 20% of the overall program execution time.

Testcase	Nesting depth	Source location	Testcase	Nesting depth	Source location
adpcm	1	adpcm.c:850	dfmul	1	dfmul.c:137
		adpcm.c:846	dfsin	1	dfsin.c:173
aes	1	aes_dec.:116	gsm	1	lpc.c:134
blowfish	1	bf.c:840	mips	1	mips.c:139
dfadd	1	dfadd.c:215	motion	0	main
dfdiv	1	dfdiv.c:144	sha	1	sha.c:210

Table 1: CHStone kernel characteristics

As described in Section 4, we successively extract kernels at an increasing nesting depth from the input program, in order to find the sections of code most suitable for HLS. For the purpose of this evaluation we then identify the nesting depth with the highest number of significant kernels, allowing us to study the input program at the finest level of granularity possible. Tables 1 and 2 list the significant kernels for the applications from the CHStone [5] and MachSuite [15] benchmark suites and give the respective nesting depths at which the kernels have been extracted.

For each significant kernel we compute two characteristics: The *Spatial factor* [2] gives the speed-up of a spatial execution in the blockwise model over a purely sequential execution and indicates the possible improvement resulting from the spatial distributed computing paradigm used on FPGAs. The *Pipeline factor* [3] on the other hand reports the speed-up of a pipelined execution in the Pipeline model over the execution in the Blockwise model and characterises a program's suitability for loop pipelining.

Figure 5 gives the resulting factors for all significant kernels extracted from input programs from the CHStone and MachSuite benchmark suite. These estimates were computed by SpExSim, which optimised the kernels with LLVM's presets `-O3` and `-std-link-opts`, but neither used loop unrolling nor inlining. The schedulers were parameterised to allow one concurrent load and store operation each, and chained up to 4 combinatorial operations in one time step. For the Pipelined model, actual modulo schedules were computed by the Moovac scheduler [11] with a time limit of 2 minutes per scheduling attempt.

The results indicate that all significant kernels would benefit from a spatially distributed execution model. On top of that, SpExSim estimates that 9 kernels would benefit from an accelerator that uses a pipelined microarchitecture. Note that in order to compute the speedup of the Pipeline model over the sequential execution, one needs to multiply the Spatial factor with the Pipeline factor, e.g. for `stencil/2d`, the estimated speedup (disregarding clock-frequency differences) is $3.22 \cdot 1.58 = 5.09$.

[2] Spatial factor $= \frac{\text{Exec.-Time(Sequential)}}{\text{Exec.-Time(Blockwise)}}$

[3] Pipeline factor $= \frac{\text{Exec.-Time(Blockwise)}}{\text{Exec.-Time(Pipeline)}}$

Testcase	Nesting depth	Source location	Testcase	Nesting depth	Source location
aes	1	aes.c:191	md/grid	1	md.c:16
backprop	1	backprop.c:264	md/knn	1	md.c:24
bfs/bulk	0	main	nw	1	nw.c:30
bfs/queue	0	main	sort/merge	1	sort.c:38
fft/strided	1	fft.c:8	sort/radix	1	sort.c:84
fft/transpose	1	fft.c:244	spmv/crs	0	main
	1	fft.c:127	spmv/ellpack	0	main
gemm/blockeed	1	gemm.c:15	stencil/stencil2D	1	stencil.c:7
gemm/ncubed	1	gemm.c:8	stencil/stencil3D	0	main
kmp	1	kmp.c:31	viterbi	1	viterbi.c:18

Table 2: MachSuite kernel characteristics

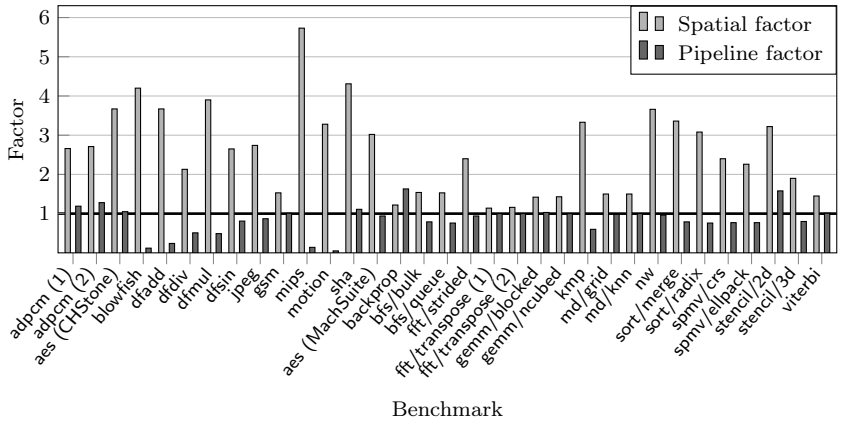

Fig. 5: Factors for applications from CHStone and MachSuite

7 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 and a pipelined model of computation based on estimations for two different hardware microarchitectures. The estimations for both microarchitectures have been validated to lie within 2.26% and 7.31% respectively. As our evaluation showed, many high-level language programs contain regions of code suitable for HLS. The interesting kernels discovered by SpExSim can then be manually examined for additional optimisation potential (e.g. adding performance-enhancing HLS directives), yielding even further speedups.

References

1. Canis, A., Choi, J., Fort, B., Lian, R., Huang, Q., Calagar, N., Gort, M., Qin, J.J., Aldham, M., Czajkowski, T., Brown, S., Anderson, J.: From software to accelera-

tors with LegUp high-level synthesis. In: 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2013). DOI 10.1109/CASES.2013.6662524
2. Canis, A.C.: Legup: Open-source high-level synthesis research framework. Ph.D. thesis, University of Toronto (2015)
3. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for fpgas: From prototyping to deployment. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **30**(4) (2011)
4. De Micheli, G.: Synthesis and optimization of digital circuits. McGraw-Hill (1994)
5. Hara, Y., Tomiyama, H., Honda, S., Takada, H.: Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. Journal of Information Processing **17**(4) (2009). DOI 10.2197/ipsjjip.17.242
6. Holland, B., Nagarajan, K., George, A.D.: Rat: Rc amenability test for rapid performance prediction. ACM Trans. Reconfigurable Technol. Syst. **1**(4), 22:1–22:31 (2009). DOI 10.1145/1462586.1462591
7. Huthmann, J., Liebig, B., Oppermann, J., Koch, A.: Hardware/software co-compilation with the nymble system. In: Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on (2013)
8. Lange, H., Wink, T., Koch, A.: MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp. 1–6. IEEE (2011)
9. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, CGO (2004). DOI 10.1109/CGO.2004.1281665
10. Nane, R., Sima, V.M., Pilato, C., et al.: A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2016). DOI 10.1109/TCAD.2015.2513673
11. Oppermann, J., Koch, A., Reuter-Oppermann, M., Sinnen, O.: Ilp-based modulo scheduling for high-level synthesis. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '16, pp. 1:1–1:10. ACM (2016). DOI 10.1145/2968455.2968512
12. Pangrle, B.M., Gajski, D.D.: Design tools for intelligent silicon compilation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **6**(6), 1098–1112 (1987). DOI 10.1109/TCAD.1987.1270350
13. Park, J., Diniz, P.C., Shayee, K.R.S.: Performance and area modeling of complete fpga designs in the presence of loop transformations. IEEE Transactions on Computers **53**(11), 1420–1435 (2004). DOI 10.1109/TC.2004.101
14. Putnam, A., Caulfield, A.M., Chung, E.S., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture (2014)
15. Reagen, B., Adolf, R., Shao, Y.S., Wei, G.Y., Brooks, D.: MachSuite: Benchmarks for accelerator design and customized architectures (2014). DOI 10.1109/IISWC.2014.6983050
16. da Silva, B., Braeken, A., D'Hollander, E.H., Touhafi, A.: Performance modeling for fpgas: Extending the roofline model with high-level synthesis tools. Int. J. Reconfig. Comput. **2013**, 7:7–7:7 (2013). DOI 10.1155/2013/428078
17. Sotomayor, R., Sanchez, L.M., Blas, J.G., Calderon, A., Fernandez, J.: Aki: Automatic kernel identification and annotation tool based on c++ attributes. In: Trustcom/BigDataSE/ISPA, 2015 IEEE, vol. 3 (2015)
18. Wang, Z., He, B., Zhang, W., Jiang, S.: A performance analysis framework for optimizing opencl applications on fpgas. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 114–125 (2016). DOI 10.1109/HPCA.2016.7446058
19. Xilinx Inc.: Zynq-7000 all programmable soc. URL http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html